Beej의 네트워크 프로그래밍 안내서

Brian “Beej Jorgensen” Hall

Contents

Chapter 1

도입부

이봐요! 소켓 프로그래밍 때문에 힘든가요? man페이지로 공부하기가 좀 지나치게 어려운가요? 멋진 인터넷 프로그래밍을 하고싶지만 connect()를 호출하기 전에 bind()를 호출해야 한다는 것을 알아내기 위해서 한 무더기의 struct를 헤집고다닐 시간이 없나요?

음, 그런데 제가 그 귀찮은 일을 이미 다 해놨습니다. 그리고 그 정보를 여러분에게 공유하고 싶어서 죽을 지경입니다. 제대로 찾아오셨습니다. 이 문서는 보통 수준의 C 프로그래머가 귀찮은 네트워킹을 처리할 수 있게 도와줄 것입니다.

그리고 확인하실 것: 제가 드디어 미래의 기술을 따라잡았고(정말 겨우 시간을 맞췄죠) IPv6에 대한 안내를 갱신했습니다. 재밌게 보십시오!

1.1 읽는이에게

이 문서는 완전한 참고문서가 아닌 튜토리얼로서 작성되었습니다. 아마도 이제 막 소켓 프로그래밍을 시작해서 발받침이 필요한 사람이 읽기에 적합할 것입니다. 이것은 분명히 어떤 의미로든 완벽하고 완전한 소켓프로그래밍 가이드는 아닙니다.

그럼에도 희망적으로, 이 문서를 읽고나면 man page가 이해되기 시작할 것입니다.

1.2 실행환경과 컴파일러

이 문서에 포함된 코드는 Gnu의 gcc 컴파일러를 사용하는 리눅스 PC에서 컴파일 되었습니다. 그러나 그 코드들은 gcc를 사용하는 어떤 실행환경에서도 빌드가 되어야 합니다. 그 말인즉 당신이 윈도우를 위해서 프로그램을 만들고 있다면 해당사항이 없다는 의미입니다. 그런 경우라면 윈도우즈 프로그래밍을 위한 절을 참고하십시오.

1.3 공식 홈페이지와 책 구매

이 문서의 공식 위치는 아래와 같습니다:

- https://beej.us/guide/bgnet/

이 곳에서 예제 코드와 이 안내서의 여러 언어로 된 번역본도 찾을 수 있습니다.

사람들이 책이라고 부르는 잘 제본된 인쇄된 복사본을 사고싶다면 여기에 방문하십시오:

- https://beej.us/guide/url/bgbuy

구매해주신다면 저의 먹물밥으로 먹고 사는 라이프 스타일을 유지하는 일에 도움이 되므로 감사하겠습니다.

1.4 Solaris/SunOS 프로그래머들을 위한 노트

Solaris 또는 SunOS를 위해서 컴파일할 때, 정확한 라이브러리에 링크하기 위해서 약간의 추가적인 명령줄 스위치를 지정해야 합니다. 그러기 위해서 컴파일 명령의 끝에 "-lnsl -lsocket -lresolv"를 아래와 같이 덧붙이십시오.

$ cc -o server server.c -lnsl -lsocket -lresolv

여전히 에러가 있다면, 그 명령 뒤에 -lxnet를 덧붙여보십시오. 그것이 정확히 무엇을 하는지는 모르지만, 몇몇 사람들은 그렇게 해야 했다고 합니다.

당신이 문제를 발견할 수도 있는 또 다른 곳은 setsockopt()을 호출하는 곳입니다. 제 리눅스 장치와 함수 원형이 다릅니다. 그러므로 아래의 코드 대신:

int yes=1;

이렇게 입력하십시오:

char yes='1';

제가 Sun 장치를 가지지 않았으므로, 위에 적은 내용들을 시험해보지는 않았습니다. 저 내용들은 단지 사람들이 저에게 이메일로 알려준 것입니다.

1.5 Windows 프로그래머들을 위한 노트

안내서의 이 부분을 적는 시점에 저는 더이상 제가 싫어한다는 이유로 Windows를 욕하는 일은 하지 말자고 다짐했습니다. 공평해야 하니까 미리 말해두자면 윈도우는 널리 사용되고 있고 분명히 완전히 멀쩡한 운영체제입니다.

추억은 미화된다고 하던데, 이 경우엔 사실인 듯 합니다.(아니면 제가 나이를 먹어서 그런가봅니다.) 제가 말할 수 있는 것은 마이크로소프트의 운영체제를 제 개인적 작업에 10년 이상 쓰지 않은 결과, 저는 더 행복하다는 것입니다. 얼마나 편하냐면 저는 의자에 기대서 편하게 "그럼요, 윈도우 써도 좋죠!"라고 말할 수 있습니다. 사실 그렇게 말하자니 어금니를 꽉 깨물게 되는군요.

그래서 저는 여전히 윈도우 대신 Linux[1], BSD[2], 아니면 다른 종류의 유닉스를 써 보라고 권하고 싶습니다.

하지만 사람들은 좋아하던 것을 계속 좋아하는 법이고, 윈도우를 쓰는 친구들은 이 문서의 정보가 그들에게도 보통 적용된다는 것을 알면 기뻐할 것입니다. 때때로 약간의 차이는 있겠지요.

당신이 진지하게 고려해봐야 할 다른 것은 Windows Subsystem for Linux[3] 입니다. 이것은 간단히 말하자면 Windows 10에 리눅스 VM 비슷한 것을 깔게 해 줍니다. 그것도 당신이 충분히 준비할 수 있게 해 줄 것이고, 예제 프로그램을 있는 그대로 빌드할 수 있게 해 줄 것입니다.

당신이 할 수 있는 다른 멋진 일은 Cygwin[4]을 설치하는 것입니다. 이것은 Windows를 위한 유닉스 도구 모음입니다. 그렇게 하면 예제 프로그램을 수정 없이 컴파일 할 수 있다고 들었습니다만 직접 해 보지는 않았습니다.

그러나 여러분 중 몇몇은 순수한 Windows방식으로 이걸 하고싶을지도 모르겠습니다. 그렇다면 아주 배짱이 두둑한 일이 되겠군요. 그렇게 하고싶다면 당장 집 밖으로 가서 유닉스를 돌릴 기계를 사십시오! 장난입니다. 요새는 윈도우에 (좀 더) 친화적으로 행동하려고 노력하고 있습니다.

이게 당신이 해야 할 일입니다. 첫 번째로 제가 이 문서에서 언급하는 거의 모든 시스템 헤더 파일을 무시하십시오. 그 대신 아래의 헤더파일을 포함하십시오.

#include <winsock2.h>
#include <ws2tcpip.h>

winsock은 "새로운"(1994년 기준으로) 윈도우즈 소켓 라이브러리입니다.

불행하게도 당신이 windows.h를 인클루드하면 그것이 자동으로 버전1인 오래된 winsock.h 를 끌어오고 winsock2.h와 충돌을 일으킬 것입니다. 정말 재밌지요.

그러므로 만약 windows.h를 인클루드해야 한다면 그것이 오래된 헤더를 포함하지 않도록 아래의 매크로를 정의해야 합니다.

#define WIN32_LEAN_AND_MEAN // 이렇게 적으십시오.

#include <windows.h> // 이제 이걸 인클루드해도 됩니다.
#include <winsock2.h> // 이것도요.

[1] https://www.linux.com/
[2] https://bsd.org/
[3] https://learn.microsoft.com/en-us/windows/wsl/
[4] https://cygwin.com/

잠깐! 소켓 라이브러리를 쓰기 전에 WSAStartup()을 호출해야 합니다. 이 함수에게 사용하길 원하는 Winsock 버전(예를 들어 2.2)을 넘겨주고 결과값을 확인해서 쓰고자 하는 버전이 사용 가능한지 확인해야 합니다.

그 작업을 하는 코드는 아래와 비슷할 것입니다.

```
#include <winsock2.h>

{
   WSADATA wsaData;

   if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
      fprintf(stderr, "WSAStartup failed.\n");
      exit(1);
   }

   if (LOBYTE(wsaData.wVersion) != 2 ||
      HIBYTE(wsaData.wVersion) != 2)
   {
      fprintf(stderr,"Versiion 2.2 of Winsock is not available.\n");
      WSACleanup();
      exit(2);
   }
```

저기 보이는 WSACleanup() 호출부를 주목하십시오. Winsock라이브러리를 다 쓴 후에 저것을 호출해야 합니다.

또한 컴파일러에게 ws2_32.lib라는 Winsock2 라이브러리를 링크하라고 말해줘야 합니다. VC++에서는 프로젝트 메뉴에서 설정으로 가서 링크탭을 클릭하고 "오브젝트/라이브러리 모듈" 이라는 제목이 붙은 상자를 찾으십시오. 그리고 거기에 "ws2_32.lib"나 당신이 원하는 다른 라이브러리를 추가하십시오. (옮긴이 주) 최신 Visual Studio에서는 [이 링크|https://learn.microsoft.com/en-us/cpp/build/reference/dot-lib-files-as-linker-input?view=msvc-170]를 참고해보십시오.

직접 해 본 것은 아닙니다.

그렇게 하고나면, 이 튜토리얼의 나머지 예제들은 거의 그대로 쓸 수 있을 것입니다. 몇 가지 예외가 있는데 소켓을 닫기 위해서 close()를 쓸 수 없습니다. 대신 closesocket()을 써야합니다. 또한 select()는 파일 설명자가 아닌 소켓 설명자에만 쓸 수 있습니다. (stdin의 0 같은 파일 설명자)

당신이 쓸 수 있는 소켓 클래스도 있습니다. CSocket입니다. 자세한 정보는 컴파일러의 도움말 페이지를 참고하십시오.

Winsock에 대한 정보를 더 얻고싶다면 마이크로소프트의 공식 홈페이지를 참고하십시오.

마지막으로 윈도우에는 fork()가 없다고 들었습니다. 불행히도 제 예제코드 중 일부는 fork()를 사용합니다. 아마 그것을 동작하게 하려면 POSIX라이브러리에 링크하거나 다른 작업이 필요할 것입니다. 아니면 CreateProcess() 를 대신 쓸 수도 있습니다. fork()는 인수를 받지 않지만 CreateProcess()는 인수를 4800만개 정도 받습니다. 그게 부담스럽다면 CreateThread() 이 조금 더 쓰기 쉬울겁니다. 불행히도 멀티스레딩에 대한 논의는 이 문서의 범위를 벗어납니다. 저는 이 정도 까지밖에 말씀드릴 수가 없습니다.

정말 마지막으로, Steven Mitchell이 몇몇 예제들을 Winsock으로 변환했습니다.[5] 확인해보십시오.

1.6 이메일 정책

저는 대체로 이메일로 오는 문의사항에 답을 드리고자 하니 이메일 보내기를 주저하지 마십시오. 그러나 응답을 보장하지는 못합니다. 저는 바쁜 삶을 살고 있고 제가 당신이 가진 궁금증에 대답할 수 없는 때가 많이 있습니다. 그런 경우라면 저는 그 메시지를 그냥 삭제합니다. 개인적인 감정이 아닙니다. 그저 당신이 필요로 하는 자세한 답을 할 시간이 없을 것이라 생각하기 때문입니다.

규칙을 제시하자면 질문이 복잡할수록 제가 응답할 가능성이 적어질 것입니다. 메일을 보내기 전에 질문의 범위를 좁히고 적절한 정보(실행환경, 컴파일러, 당신이 접하는 에러메시지, 문제 해결에 도움이 될 만한 다른 정보)를 첨부해주신다면 제 응답을 받을 확률이 올라갈 것입니다. 더 자세한 지침은 ESR의 문서인 How To Ask Questions The Smart Way[6]을 참고하십시오.

[5] https://www.tallyhawk.net/WinsockExamples/
[6] http://www.catb.org/~esr/faqs/smart-questions.html

당신이 제 회신을 받지 못한다면, 문제를 더 파고들어보고, 답을 찾기 위해 노력해보십시오. 그래도 확실한 답을 얻지 못했다면 그동안 알아낸 정보를 가지고 저에게 다시 메일을 보내십시오. 어쩌면 제가 답을 드릴 수 있을지도 모릅니다.

저에게 메일을 보낼 때 이렇게 해라 저렇게 해라 말이 많았습니다만 이 안내서에 지난 몇 년 동안 보내주신 모든 칭찬에 정말로 감사한다는 사실을 말씀드리고 싶습니다. 그것은 정말로 정신적인 힘이 됩니다. 이 안내서가 좋은 일에 쓰였다는 말을 듣는 일은 저를 기쁘게 합니다. :-) 감사합니다!

1.7 미러링

이 웹사이트를 공개로든 사적으로든 미러링하는 것은 정말로 환영합니다. 이 웹사이트를 공개적으로 미러링하고 제가 메인 페이지에 링크를 걸게 하고 싶다면 beej@beej.us 로 메일을 보내주십시오.

1.8 Note for Translators 번역가들을 위한 노트

이 안내서를 다른 언어로 번역하고 싶다면 beej@beej.us에게 메일을 보내주십시오. 당신의 번역본의 링크를 제 메인 페이지에 걸어두겠습니다. 당신의 이름과 연락처 정보를 번역본에 추가하셔도 좋습니다.

이 원본 마크다운 문서는 UTF-8로 인코드 되었습니다.

아래의 Copyright, Distribution, and Legal 절을 참고하십시오.

제가 번역본을 호스트하길 바란다면, 말씀해주십시오. 당신이 호스트하길 원한다면 그것도 링크하겠습니다. 어느 쪽이든 좋습니다.

1.9 Copyright, Distribution, and Legal

(Translator's Note : This section has not been translated to keep it's legal information) (역자 주 : 이 절은 법적 정보를 보존하기 위해 번역하지 않았습니다.)

1.10 헌사

이 안내서를 쓸 수 있도록 저를 과거에 도와주신, 그리고 미래에 도와주실 모든 분들에게 감사합니다. 이 안내서를 만들기 위해 사용한 자유 소프트웨어와 패키지(GNU, Linux, Slackware, vim, Python, Inkscape, pandoc, 기타 등등...)를 만든 모든 분들에게 감사드립니다. 또한 이 안내서의 발전을 위한 제안을 해주시고 응원의 말씀을 보내주신 수천명의 사람들에게 감사드립니다.

이 안내서를 컴퓨터 세계에서 나의 가장 위대한 영웅이자 영감을 주는 이들에게 바칩니다. Donald Knuth, Bruce Schneier, W. Richard Stevens, Steve The Woz Wozniak, 독자 여러분, 그리고 모든 자유 및 공개 소프트웨어 커뮤니티

1.11 출판 정보

이 책은 GNU 도구를 적재한 Arch Linux장치에서 Vim 편집기를 사용해서 Markdown 으로 작성되었습니다. 표지 "미술"과 다이어그램은 Inkscape로 작성되었습니다. Markdown은 Python과 Pandoc, XeLaTex를 통해 HTML과 Latex/PDF로 변환되었습니다. 문서에는 Liberation 폰트를 사용했습니다. 툴체인은 전적으로 자유/공개 소프트웨어를 사용해서 구성했습니다.

1.12 옮긴이의 말

이 문서의 첫 한국어 번역 버전은 박성호(tempter@fourthline.com)님이 1998/08/20(yyyy/MM/d)에 인터넷의 어딘가에 게시하신 것으로 보입니다. 현재는 KLDP에 있습니다.

이 문서의 두 번째 한국어 번역 버전은 김수봉(연락처 없음)님이 2003/12/15(yyyy/MM/d)에 KLDP에 게시하셨습니다. 첫 번역 문서를 html에서 wiki형태로 변환했다고 적혀 있습니다.

지금 읽고 계시는 세 번째 버전은 정민석(javalia.javalia@gmail.com)이 작업했으며, 2023년 5월 28일부터 작업했습니다.

이 문서의 번역본은 1998년에 최초로 한국어 버전이 작성된 이래로 한국인이 Socket 프로그래밍에 대해서 참고할 수 있는 문서 중 리눅스 man page를 제외하면 가장 1차적인 문서였습니다. 실제로 많은 소켓 프로그래밍 관련 글과 출판된 책의 예제 코드도 이 문서의 것을 차용하고 있습니다. 원문은 세월이 흐르면서 조금씩 개정되어 내용이 상세해지고 IPv6에 대해서도 다루고 있으나 번역본은 20년 전의 모습으로 멈추어 있었습니다. 소켓 프로그래밍을 공부하던 시절 가장 큰 도움을 받은 문서의 번역본이 개정되지 않음을 안타깝게 생각해서 이렇게 새로운 번역본을 만들게 되었습니다. 이 글이 새로운 네트워크 라이브러리를 개발하는 프로그래머에게 도움이 되기를 바랍니다.

이 문서는 원문의 3.1.5 버전을 기반으로 번역되었습니다. 원문에 등장하는 고유명사는 해당 명사에 통용되는 한국어 번역이 없는 한 원어 그대로 실었습니다. 영어 일반명사는 음역을 기본으로 하였으나, 일부는 의역하기도 하였고 혼동을 줄이기 위해서 병기한 부분도 있습니다. 작업 과정에서 이전 번역자들의 원문을 유지하지는 못했습니다. 원문/번역본/새 번역본 사이의 대조/교정 작업은 개인적인 시간을 짜내서 진행하는 이 일에는 너무 큰 작업이었습니다. 이러한 사정으로 인해 이전 번역자들의 작업이 직접적으로 유지되지는 못하지만, 다른 프로그래머들을 위해 수 십년 전 글을 남기신 번역자 분들의 노력을 이어받고 그 작업을 존중하는 마음으로 다음 몇 년간 사람들이 읽을 수 있는 번역을 제공하기 위해 노력했습니다.

읽어주셔서 감사합니다.

Chapter 2

소켓이란 무엇인가?

여러분은 "소켓"이란 단어를 자주 들을 것입니다. 그리고 어쩌면 그것이 정확히 무슨 뜻인지 궁금하시겠지요. 그것은 표준 유닉스 파일 설명자를 통해서 다른 프로그램과 이야기하는 방법을 의미합니다.

무슨 말이냐고요?

좋습니다. 아마 어떤 유닉스 해커들이 "어으, 유닉스에선 모든 것이 파일이야!" 라고 말하는 것을 들어보셨을 것입니다. 그들이 말하는 바는 유닉스 프로그램이 어떤 종류의 입출력을 하든, 파일 설명자에 읽거나 쓰는 방식으로 동작한다는 의미입니다. 파일 설명자는 단순히 열려있는 파일과 연관된 정수입니다. 그러나 (이 부분이 중요합니다.) 그 파일은 네트워크 연결이나 선입선출, 파이프, 터미널, 디스크에 있는 진짜 파일이나 다른 어떤 것이든 될 수 있습니다. 유닉스의 모든 것 은 파일입니다! 그러니 인터넷 너머의 다른 프로그램과 통신하고 싶다면 파일 설명자를 통해서 하는 것이 당연합니다.

"그래서 이 네트워크 통신을 위한 파일 설명자를 어디에서 구하죠, 똑똑이양반?" 이라고 아마 지금 생각하실 듯 합니다. 답을 드리자면 socket() 시스템 루틴을 호출하면 된다는 겁니다. 그것은 소켓 설명자를 반환하고, 여러분은 특화된 send()와 recv() (man send, man recv) 소켓 함수를 써서 그 소켓을 통해 통신합니다.

"그런데 잠시만요!" 아마 다른 의문이 생길 것입니다. "그게 그냥 파일 설명자라면, 어째서 그냥 평범한 read()와 write() 함수를 사용해서 소켓 통신을 하면 안되는 것입니까?" 짧은 답은 "그래도 됩니다!"입니다. 긴 답은 "그래도 되지만, send()과 recv()이 데이터 전송을 더 많이 조정할 수 있게 해 줍니다"가 되겠습니다.

다음이 궁금하신가요? 세상에는 온갖 종류의 소켓이 있습니다. DARPA 인터넷 주소(인터넷 소켓), 로컬 노드의 경로(유닉스 소켓), CCITT X.25 주소(무시해도 상관없는 X.25주소), 그리고 여러분이 실행중인 유닉스의 종류에 따라 다른 많은 종류의 소켓들. 이 문서는 첫 번째 것에 대해서만 다룹니다. 바로 인터넷 소켓입니다.

2.1 두 종류의 인터넷 소켓

인터넷 소켓이 두 종류라니 무슨 소리냐고요? 사실 거짓말입니다. 더 많은 종류가 있습니다. 그러나 여러분을 겁먹게 하기 싫었습니다. 여기서는 두 종류에 대해서만 이야기하겠습니다. 여러분에게 "Raw Socket"이라는 것이 있으며 그것이 아주 강력하고 한 번쯤 살펴보시길 권한다고 말하는 지금 이 문장을 제외하고 말입니다.

"이제 됐고 그 두 종류가 도대체 뭡니까?" 하나는 "Stream Socket(스트림 소켓)"이고 다른 하나는 "Datagram Socket(데이터그램 소켓)" 입니다. 앞으로 이 둘을 각각 "SOCK_STREAM" 과 "SOCK_DGRAM"으로 칭하겠습니다. 데이터그램 소켓은 때때로 비연결형/비연결성 소켓이라고 불립니다. (그러나 그것도 정말로 필요하다면 connect() 함수를 사용할 수 있습니다. 아래의 connect()를 참고하하십시오.)

스트림 소켓은 신뢰성있는 양방향 연결 통신 스트림입니다. 이 소켓에 두 개의 아이템을 "1, 2"의 순서로 출력하면, 반대쪽 끝에 "1, 2"의 순서로 도착합니다. 또한 스트림소켓은 에러가 발생하지 않습니다. 이것은 정말로 확실한 내용이어서, 저는 다른 사람들이 이것에 대해서 반박한다면 귀를 막고 노래나 부르겠습니다.

"무엇이 스트림 소켓을 사용하나요?" telnet이나 ssh 응용프로그램에 대해서 들어보셨습니까? 그것들이 스트림 소켓을 사용합니다. 여러분이 입력하시는 모든 문자가 입력하신 순서 그대로 도착해야 합니다. 또한, 웹브라우저가 쓰는 Hypertext Transfer Protocol (HTTP)도 스트림 소켓을 사용합니다. 정말로, 어떤 웹사이트의 80번 포트에 텔넷으로 연결한 후 "GET / HTTP/1.0"을 입력하고 엔터를 두 번 치면 HTML을 돌려줄 것입니다.

> 여러분이 telnet을 설치하지 않았고 설치하고 싶지 않거나, 설치된 telnet이 클라이언트에 연결하는 것에 대해 까다롭게 군다면 이 안내서는 telnet과 유사한 프로그램인 telnot[1]과 같이 제공됩니다. 이 안내서가 필요로 하는 일은 다 할 수 있을 것입니다. 텔넷이 사실은 spec'd networking protocol[2]이며, telnot 은 이 프로토콜을 전혀 구현하지 않는다는 사실을 기억하십시오.

"스트림 소켓이 어떻게 이렇게 수준높은 데이터 전송 품질을 달성하나요?" 스트림 소켓은 "Transmission Control Protocol"혹은 "TCP" (TCP에 대한 지나치게 자세한 정보는 RFC 793[3]를 참고하십시오) 라고 불리는 프로토콜을 사용합니다. TCP는 여러분의 데이터가 순서대로 도착하고 오류가 없음을 보장합니다. "TCP"를 "TCP/IP"의 반절로 이미 들어보셨을 것입니다. "IP"는 "Internet Protocol(인터넷 프로토콜)"의 약어입니다. (RFC 791[4]을 살펴보십시오) IP는 주로 인터넷 라우팅을 맡으며 데이터 무결성에는 보통 책임이 없습니다.

"굉장하네요. 데이터그램 소켓은 뭔가요? 왜 비연결성이죠? 뭐가 다른가요? 왜 신뢰성이 없나요?" 대답은 아래와 같습니다. 데이터그램을 전송하면 도착할 수도, 도착하지 않을 수도 있습니다. 도착은 하되 순서대로 도착하지 않을 수도 있습니다. 도착한다면, 패킷 안에 있는 데이터에는 에러가 없습니다.

데이터그램 소켓도 라우팅을 위해서 IP를 사용할 것입니다. 그러나 TCP를 사용하지는 않습니다. 데이터그램 소켓은 "User Datagram Protocol" 또는 "UDP"를 사용합니다. (RFC 768[5]를 참고하십시오)

"왜 비연결성인가요?" 간단히 말하자면 스트림 소켓과 달리 열린 연결을 유지할 필요가 없기 때문입니다. 패킷을 만들고, 목적지 정보를 담은 IP헤더를 붙이고 보냅니다. 연결이 필요하지 않습니다. 데이터그램 소켓은 일반적으로 TCP 스택을 사용할 수 없거나 패킷 몇 개가 유실되어도 세상이 끝장나지는 않는 상황에서 쓰입니다. 몇몇 예제 프로그램은 다음과 같습니다. tftp (trivial file transfer protocol, FTP의 동생), dhcpcd (DHCP 클라이언트), 다중 사용자 게임, 오디오 스트리밍, 화상 회의, 등등

"잠시만요! tftp나 dhcpcd는 하나의 호스트에서 다른 호스트로 이진 응용프로그램을 전송하기 위해 쓰이잖아요! 응용프로그램이 도착지에서 제대로 동작하길 바라면 데이터가 유실되면 안 되는 것 아닌가요? 데이터를 제대로 보내기 위해서 마법이라도 쓰나요?"

머글 여러분, tftp와 유사한 프로그램들은 UDP위에서 자신만의 프로토콜을 구현합니다. 예를 들어 tftp프로토콜은 그들이 보내는 모든 패킷에 대해서 수신자가 "잘 받았습니다"라고 말하는 패킷("ACK" 패킷)을 돌려줄 것을 요구합니다. 원본 패킷의 송신자가 일정 시간, 가령 5초 안에 응답을 받지 못한다면 송신자는 ACK응답을 받을 때까지 패킷을 재전송합니다. 이 확인 절차는 아주 중요해서 신뢰할 수 있는 SOCK_DGRAM 응용프로그램을 만들 때 아주 중요합니다.

게임이나 오디오, 비디오같은 신뢰할 수 없는 응용프로그램의 경우 유실된 패킷을 그냥 무시하거나 혹은 똑똑하게 무마하려고 합니다. (퀘이크 사용자들은 이런 유실로 인한 영향을 전문 용어로 짜증나는 렉 이라고 부릅니다. 여기에서 쓰인 "짜증나는" 은 아주 극단적인 욕설을 대체한 표현입니다.)

신뢰할 수 없는 기반 프로토콜을 왜 사용하는지 궁금하십니까? 두 가지입니다. 속도와 속도. 발사 후 망각(fire-and-forget) 방식이 무엇이 안전하게 도착했는지 추적하고 데이터가 올바른 순서로 왔는지 등을 확인하는 것보다 빠릅니다. 대화 메시지를 보낸다면 TCP는 훌륭한 선택입니다. 세계 안에서 초당 40번의 위치 정보 갱신을 전송한다면 한두 개의 정보가 유실되어도 상관없습니다. 그렇다면 UDP가 좋은 선택입니다.

2.2 저수준 넌센스와 네트워크 이론

프로토콜의 계층구조에 대해서 이야기했으니 네트워크가 실제로 어떻게 동작하는지 이야기할 차례입니다. SOCK_DGRAM패킷이 어떻게 만들어지는지 약간의 예제를 보여드리겠습니다. 실용적으로는 이 절을 생략해도 좋습니다. 그러나 좋은 배경지식입니다.

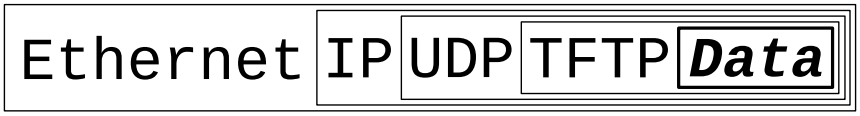

Figure 2.1: 데이터 캡슐화(Data Encapsulation).

데이터 캡슐화 에 대해 배울 차례입니다. 이것은 아주 중요합니다. 너무 중요해서 치코 캘리포니아 주립대에서 네트워크 수업을 듣는다면 이것에 대해 배우게 될 수도 있습니다. 간단히 말하자면 이렇습니다. 패킷이 태어나면 패킷은 첫 번째 프로토콜(예를 들어 TFTP 프로토콜)에

[1] https://beej.us/guide/bgnet/examples/telnot.c
[2] https://tools.ietf.org/html/rfc854
[3] https://tools.ietf.org/html/rfc793
[4] https://tools.ietf.org/html/rfc791
[5] https://tools.ietf.org/html/rfc768

의해 헤더로 감싸집니다("캡슐화") (때때로 푸터로도 감싸집니다). 그리고 전체가 다시 다음 프로토콜에 의해 감싸집니다(말하자면 UDP같은 것). 그리고 다시 IP로 감싸지고, 또 다시 하드웨어(물리) 계층(예를 들어 이더넷(Ethernet)에 의해 최종적인 프로토콜로 감싸집니다.

다른 컴퓨터가 패킷을 받으면 하드웨어는 이더넷 헤더를 벗겨내고, 커널이 IP와 UDP헤더를 벗겨냅니다. 그리고 TFTP프로그램이 TFTP헤더를 벗겨내고, 마침내 데이터를 가지게 됩니다.

드디어 악명높은 계층화 네트워크 모델(Layered Network Model) 에 대해서 이야기할 수 있습니다. 이 네트워크 모델은 네트워크 기능의 계(system) 를 묘사하며 다른 모델에 비해서 많은 장점을 가지고 있습니다. 예를 들어 여러분은 물리적으로 데이터가 어떻게 전송되는지(직렬통신(Serial), Thin Ethernet(얇은 동축케이블을 쓰는 이더넷의 변형), AUI(Attachment Unit Interface), 등등)(역자 주 : 여기에 언급되는 물리적 통신 단자들은 대개 현재는 특수한 산업현장이 아니면 쓰이지 않습니다. 여러분이 이런 것에 대해서 모르신다고 해도 전혀 지장이 없다는 의미입니다.)에 대해서 전혀 신경쓰지 않고 소켓 프로그램을 완전히 똑같은 모습으로 짤 수 있습니다. 그 이유는 저수준에 있는 프로그램들이 그것을 자동으로 처리해주기 때문입니다. 실제 네트워크 하드웨어와 망 구성방식(topology)는 소켓 프로그래머에게 투명(역자 주 : 알 필요가 없거나 알 수 없음)합니다.

길게 말하지 않고 이제 전체 모델의 계층을 제시하겠습니다. 네트워크 과목 시험을 위해서 이것을 기억하십시오.

- 응용(Application)
- 표현(Presentation)
- 세션(Session)
- 전송(Transport)
- 네트워크(Network)
- 데이터 링크(Data Link)
- 물리(Physical)

물리 계층은 하드웨어입니다(직렬통신, 이더넷 등). 응용 계층은 여러분이 상상할 수 있는 한 최대한 물리 계층에서 먼 것입니다. 사용자가 실제로 네트워크와 상호작용 하는 부분을 의미합니다.

사실 이 모델은 너무나 일반적이어서 정말로 원한다면 자동차 정비 안내서에도 쓸 수 있을 것입니다. 유닉스와 좀 더 일치하는 계층화 모델은 이렇습니다.

- 응용 계층(Application Layer) (텔넷, ftp 등.)
- 호스트 간 전송 계층(Host-to-Host Transport Layer) (TCP, UDP)
- 인터넷 계층(Internet Layer) (IP와 라우팅)
- 네트워크 접근 계층(Network Access Layer) (이더넷, 와이파이(wi-fi), 기타 등등)

이 시점까지 오면 여러분은 아마도 이 계층들이 원본 데이터의 캡슐화에 어떻게 대응하는지 아실 수 있을 듯 합니다.

간단한 패킷을 만들기 위해서 얼마나 많은 일이 일어나는지 아시겠습니까? 그리고 이것을 패킷 헤더에 적기 위해서 "cat"명령으로 하나하나 직접 적어야 합니다! 농담입니다. 스트림 소켓을 위해서 할 일은 그저 send() 로 데이터를 보내는 것 뿐입니다. 데이터그램 소켓을 위해서 할 일은 여러분이 원하는 방식으로 패킷을 캡슐화하고 sendto() 로 내보내는 일 뿐입니다. 커널이 여러분을 위해서 전송 계층과 인터넷 계층을 만들어주고 하드웨어가 네트워크 접근 계층을 만들어줄 것입니다. 현대 기술은 정말 멋지지요!

네트워크 이론에 대한 우리의 짧은 공부는 이렇게 끝납니다. 아, 라우팅에 대해서 말씀드리는 것을 잊었습니다. 그러나 라우팅에 대해서는 아무것도 다루지 않을 생각입니다. 라우터(router)가 IP헤더에 이르기까지 패킷을 까고, 라우팅 테이블 (routing table)을 조회하고, 어쩌고 저쩌고 등에 대한 내용은 하나도 이야기하지 않을 것입니다. 정말로 진짜로 알고싶다면 IP RFC[6]을 참고하세요. 평생 모르고 살아도 문제는 없습니다.

[6] https://tools.ietf.org/html/rfc791

Chapter 3

IP 주소, 구조체들, 데이터 처리(Munging)

변화를 위해 코드에 대해 이야기하는 부분이 되었습니다.

그러나 코드가 아닌 이야기를 조금 더 하겠습니다. 먼저 IP 주소와 포트에 대한 이야기를 조금 해서 이해하고 넘어가겠습니다. 그 다음 소켓 API가 어떻게 IP주소와 다른 정보를 저장하고 조작하는지 다루겠습니다.

3.1 IP 주소, 4판과 6판(버전4와 버전6)

벤 케노비(역자 주 : 스타워즈의 케릭터)를 아직 오비-완 케노비라고 부르던 좋은 옛 시절에는 인터넷 프로토콜 제4판 또는 IPv4라고 부르던 좋은 네트워크 라우팅 체계가 있었습니다. 그것은 4개의 바이트(또는 네 개의 "옥텟" (역자 주 : 8비트로 이루어진 1바이트를 명시적으로 지칭하는 표현))로 이루어지고 보통 192.0.2.111 같은 "점과 숫자" 형태로 기록되던 주소를 가졌습니다.

여러분은 아마도 그것을 보셨을 것입니다.

사실 이 글을 적는 시점에는 인터넷의 거의 모든 사이트가 IPv4를 사용합니다.

오비-완을 비롯해 모두가 행복했습니다. 모든 것이 훌륭했습니다. Vint Cerf 같은 부정적인 사람이 IPv4주소가 고갈되기 직전이라고 모두에게 경고하기 전까지 말입니다.

(다가오는 IPv4의 종말과 어둠을 모두에게 경고한 것 외에도 Vint Cerf[1]는 인터넷의 아버지로 아주 잘 알려져 있습니다. 그래서 저는 그의 판단에 반기를 들 수가 없습니다.)

주소가 고갈된다니 이게 가능한 일인가? 32비트 IPv4 주소는 수십억개인데 정말로 가능한지 의문일겁니다. 정말로 세상에 수십억 개의 컴퓨터가 있을까요?

있습니다.

여기에 더해서, 컴퓨터가 정말 적던 초기에는 모두가 수십억은 불가능할 정도로 큰 수라고 생각했습니다. 그래서 일부 큰 조직에 관대하게도 수백만 개의 아이피 주소를 할당해 주었습니다. (그런 식으로 많은 IP 주소를 할당받은 조직의 이름을 몇 개 대자면 Xerox, MIT, Ford, HP, IBM, GE, AT&T, 그리고 애플(Apple)이라고 하는 작은 회사도 있었습니다.)

사실 몇몇 임시방편이 없었다면 아이피는 예전에 다 떨어졌을 것입니다.

그러나 우리는 모든 사람, 모든 컴퓨터, 모든 계산기, 모든 전화기, 모든 주차 정산기, 모든 강아지와 멍멍이(왜 아니겠습니까?)에게 IP주소가 있는 시대를 살고 있습니다.

그리고 그렇게 해서 IPv6이 태어났습니다. 그리고 Vint Cerf는 아마도 불사의 존재이겠지만(그의 물리적 형체가 사라져야 한다면 그는 이미 Internet2의 깊은 곳에서 일종의 초인공지능 ELIZA[2] 프로그램 같은 모습으로 존재하고 있을 것입니다.), 다시 한 번 다음 버전의 인터넷 프로토콜에서 주소가 부족해서 그가 "내가 말했지"라는 식으로 이야기하는 것을 듣고 싶어할 사람은 없습니다.

이게 무슨 의미냐고요?

[1] https://en.wikipedia.org/wiki/Vint_Cerf
[2] https://en.wikipedia.org/wiki/ELIZA

우리에게 아주 많은 주소가 필요하다는 뜻입니다. 두 배도 아니고, 십억배도 아니고, 천조배도 아니고, _7양9천자(역자 주 : 양은 10의 28승을 의미하는 수단위)_배가 필요합니다.

"Beej, 그게 사실인가요? 저에게는 큰 수를 신뢰하지 않을 많은 이유가 있습니다."라고 말씀하시겠지요. 그러니까 32비트와 128비트 사이에는 그다지 큰 차이가 없어보일지도 모르겠습니다. 96비트가 더 있을 뿐이니까요. 하지만 우린 지금 거듭제곱에 대해서 이야기해야합니다. 32비트가 대략 40억개의 숫자들을 의미합니다. 128비트는 대략 11업 (역자 주 : 1업은 10의 76승을 의미)입니다. (정확히는 2의 128승입니다) 그것은 이 우주의 모든 별에 대해서 IPv4인터넷이 백만 개씩 있는 것과 비슷합니다.

IPv4의 점과 숫자 표기는 잊어버리십시오. 이제 우리에겐 콜론으로 구별하는 2바이트 조각으로 구성되는 16진수 표기법이 있습니다. 예시는 아래와 같습니다.

2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551

그게 다가 아닙니다! 대부분의 경우에 여러분은 0이 아주 많이 들어있는 주소를 가지게 될 것입니다. 그리고 그 0들을 두 개의 콜론 사이에 압축해서 넣을 수 있습니다. 또한 각각의 바이트 쌍의 앞에 오는 0은 생략할 수 있습니다. 예를 들어 아래의 각 주소쌍은 동일한 의미입니다.

2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1

::1주소는 재귀 주소(루프백 주소) 입니다. 그것은 언제나 "내가 실행중인 이 기계"를 의미합니다. IPv4에서 루프백 주소는 127.0.0.1입니다.

마지막으로 IPv6주소에는 여러분이 보실지도 모르는 IPv4호환 모드가 있습니다. 예시를 원하신다면 이렇습니다. 192.0.2.33을 IPv6주소로 표기한다면 아래와 같습니다. "::ffff:192.0.2.33".

이건 아주 재미있는 일입니다.

사실 너무 재미있다고 할 수 있는데, IPv6의 제작자들이 수자(역자 주 : 10의 24승) 개의 주소를 기사도적으로 잘라내어 예약된 주소로 지정했기 때문입니다. 그러나 우리에게는 그러고도 너무 많은 주소가 남아있어서 남은 주소를 세기도 귀찮을 정도입니다. 아직도 은하계의 모든 남녀노소, 강아지, 주차정산기에 배정할 주소가 남아있습니다. 은하계의 모든 행성에 주차정산기가 있다는 것은 확실합니다. 저를 믿으십시오.

3.1.1 Subnets(서브넷 또는 부분망)

관리적인 측면에서 때때로 "아이피 주소의 이 비트까지의 첫 부분은 네트워크 부분 이고 나머지는 호스트 부분" 이라고 칭하는 것이 편할 때가 있습니다.

예를 들어 IPv4주소에서 192.0.2.12를 가지고 있다면 우리는 첫 3바이트가 네트워크 주소이고 마지막 바이트가 호스트 주소라고 말할 수 있습니다. 조금 다르게 말하면 192.0.2.0 네트워크에 있는 호스트 12에 대해서 말한다고 할 수 있습니다.(네트워크 부분에서 우리가 호스트 바이트를 0으로 바꾼 것에 주목하십시오.)

더 오래된 정보를 알려드리겠습니다! 고대에는 이 서브넷에 "클래스"가 있었습니다. 각각 주소의 첫 번째, 두 번째, 세 번째 바이트까지가 네트워크 부분임을 의미했습니다. 여러분이 네트워크 부분에 1바이트를 받고 호스트 부분에 3바이트를 받을 정도로 운이 좋다면 여러분의 네트워크에 24비트 규모(대략 천육백만)의 호스트를 가질 수 있었습니다. 이것이 "클래스 A"네트워크입니다. 반대쪽 끝을 "클래스 C"라고 했습니다. 여기에서는 3바이트가 네트워크 부분이고 1바이트가 호스트입니다.(256개의 호스트이고, 예약된 주소때문에 몇 개를 더 빼야 합니다.)

그래서 보시다시피, A클래스는 몇 개 없고, C클래스는 엄청나게 많았으며, B클래스는 중간정도였습니다.

IP 주소의 네트워크 부분은 넷마스크 라는 것으로 표시되는데, IP주소에서 네트워크 번호를 얻어내기 위해서 넷마스크와 비트단위 AND연산을 하게 되어있습니다. 넷마스크는 대체로 255.255.255.0처럼 보입니다. (예를 들어 넷마스크가 저렇고 여러분의 IP가 192.0.2.12라면 네트워크는 192.0.2.12 AND 255.255.255.0이고 결과는 192.0.2.0입니다.)

불행히도 이것은 인터넷의 결과적인 필요에 비해 섬세하지 못했던 것으로 드러났습니다. C 클래스 네트워크는 꽤 빠르게 고갈되고 있었고 A클래스는 이미 다 소진되었으니 물어볼 것도 없습니다. 이 상황을 타개하기 위해서 8이나 16, 24개의 비트 뿐 아니라 임의의 비트를 넷마스크로 쓸 수 있는 능력이 필요했습니다. (예를 들어 255.255.255.252 같은 넷마스크로 30비트의 네트워크 부분과 2비트의

호스트(4개의 호스트)를 허락할 필요가 있었습니다.) (넷마스크는 언제나 여러 개의 비트 1 뒤에 뒤따라오는 여러 개의 비트 0임을 기억하십시오.)

그러나 255.192.0.0같은 긴 문자열을 넷마스크로 쓰는 것은 불편했습니다. 첫 번째로 사람들이 보기에 그것이 몇 비트인지 알기가 힘들었고 두 번째로 너무 길었습니다. 그래서 새로운 방식이 등장했고 훨씬 좋았습니다. IP 주소 뒤에 빗금(역자 주 : 슬래시 또는 /)을 적고 네트워크 비트의 수를 십진수로 적는 것입니다. 예시는 이렇습니다: 192.0.2.12/30

IPv6을 위해서는 이렇게 할 것입니다: 2001:db8::/32 또는 2001:db8:5413:4028::9db9/64

3.1.2 포트 번호

친절하게도 아직 기억해주신다면, 계층화 네트워크 모델에서 호스트 대 호스트 전송 계층(TCP and UDP)과 분리된 인터넷 계층 (IP)이 있음을 아실 것입니다. 다음 문단으로 가기 전에 한 번 더 살펴보셔도 좋습니다.

(IP 계층이 사용하는) IP주소 말고도 TCP (stream sockets)와 UDP (datagram sockets)는 우연히도 한 가지 주소를 더 사용하는 것을 알 수 있습니다. 그것은 바로 포트 번호 입니다. 이것은 16비트 숫자이며 연결을 위한 로컬 주소같은 것입니다.

IP 주소를 호텔의 도로명 주소라고 생각하고, 포트 번호를 방 번호라고 생각하십시오. 이것은 꽤 적절한 비유입니다. 어쩌면 나중에 자동차 산업과 관련된 다른 비유를 떠올릴 수 있을지도 모르겠습니다.

여러분이 이메일 수신과 웹서비스를 처리하는 컴퓨터를 가지고 있다고 해봅시다. 하나의 IP주소로 어떻게 그 두 일을 구분할 수 있겠습니까?

인터넷의 서로 다른 서비스들은 서로 다른 "잘 알려진" 포트번호를 가지고 있습니다. 전체 목록은 거대한 IANA 포트 목록[3] 또는, 여러분이 유닉스 장치를 사용하신다면 /etc/services파일에서 볼 수 있습니다. HTTP(웹)는 80번 포트를 사용하고, telnet은 23번을, SMTP는 25를 쓰고 게임인 DOOM[4]은 666번(역자 주 : 둠은 지옥에서 온 악마와 싸우는 내용의 게임이며, 666은 기독교에서 악마의 숫자로 알려져 있습니다) 포트를 사용했습니다. 1024번 아래의 포트는 흔히 특별한 것으로 취급되어, 사용하기 위해서는 보통 운영체제 특권이 필요합니다.

이게 전부입니다!

3.2 바이트 순서

왕국의 명령으로, 앞으로는 "엉터리"와 "큰 것 먼저"인 두 개의 바이트 순서가 있을 것이다!

농담입니다. 그러나 한 쪽이 다른 쪽보다 더 좋습니다. :-)

사실 이것에 대해 딱 잘라 말할 방법은 없습니다. 그러니 허풍을 좀 떨겠습니다: 여러분의 컴퓨터는 뒤에서 바이트들을 여러분이 생각하는 것과 반대되는 순서로 저장하고 있었을 수도 있습니다. 아무도 이것을 여러분에게 알려주고 싶어하지 않았을 것입니다.

중요한 것은 인터넷 세계의 모든 사람들이 b34f같은 16진수 2바이트 수를 표현하고자 할 때 b3이 앞에 오고 4f이 뒤에 오는 연속된 바이트로 저장하는 것에 대체로 동의했다는 사실입니다. 이것은 말이 되기도 하고, Wilford Brimley[5] 라면 이것에 대해서 "마땅히 해야 할 일"이라고 할 것입니다. 큰 쪽이 앞에 저장되는 이 방식을 Big-Endian(빅엔디언) 이라고 합니다.

안타깝게도 전 세계 이곳저곳에 흩어진 일부 컴퓨터들, 그러니까 인텔 혹은 인텔 호환 프로세서 컴퓨터들은 바이트를 반대 순서로 저장합니다. 그래서 b34f는 f4뒤에 b3이 있는 순차적 바이트들로 저장됩니다. 이런 저장법을 Little-Endian(리틀 엔디언) 이라고 합니다.

아직 용어 해설이 조금 남았습니다! 더 멀쩡한 빅엔디언은 _Network Byte Order (네트워크 바이트 순서)라고도 합니다. 네트워크에서 우리가 그렇게 바이트를 전송하기 때문입니다.

여러분의 컴퓨터는 숫자를 Host Byte Order(호스트 바이트 순서) 로 저장합니다. 인텔 80×86이라면 그것은 리틀엔디언입니다. 모토롤라 68k라면 호스트 바이트 순서는 빅엔디언입니다. PowerPC라면 호스트 바이트 순서는.. 상황에 따라 다릅니다! (역자 주 : 현재 널리 쓰이는 x86-64 프로세서들은 리틀 엔디언이며, 이것은 흔히 쓰이는 ARM프로세서와 최근 애플이 사용을 시작한 M1, M2 등의 ARM변종에서도 동일하다. 리틀/빅엔디언 여부에 관계 없이 한 바이트 내에서는 무조건 MSB가 앞에 온다는 것도 기억해야 한다. 결론적으로 대부분의 컴퓨터들의 호스트 바이트 오더가 네트워크 바이트 오더와 다르다.)

패킷을 만들거나 자료 구조를 채워넣는 많은 경우에 여러분은 여러분의 2바이트 또는 4바이트 숫자들이 네트워크 바이트 순서로 확실히 기록되도록 해야합니다. 하지만 원시 호스트 바이트 순서를 모른다면 어떻게 이런 작업을 할 수 있을까요?

[3] https://www.iana.org/assignments/port-numbers
[4] https://en.wikipedia.org/wiki/Doom_(1993_video_game)
[5] https://en.wikipedia.org/wiki/Wilford_Brimley

좋은 소식입니다! 그냥 호스트 바이트 순서가 늘 틀렸다고 가정하고, 그것을 네트워크 바이트 순서로 재정렬하는 함수에 넣으십시오. 그 함수가 필요하다면 마법의 변환과정을 처리하고, 여러분의 코드는 서로 다른 바이트 정렬 방식을 가진 기계 사이에서 호환성을 가질 것입니다.

좋습니다! 여러분이 변환할 수 있는 숫자에는 두 가지 종류가 있습니다: short(2바이트) 과 long(4바이트)입니다. 위에서 말한 처리함수들은 부호 없는 종류에도 쓰일 수 있습니다. 호스트 바이트 순서로 기록된 short을 네트워크 바이트 순서로 변환하고 싶다면, "host"의 "h"로 시작하고 "to"를 이어서 적고 "network"의 "n"을 적고 "short"의 "s"를 적으십시오. 다 붙이면 htons()이 됩니다. (읽는 법 : Host to Network Short)

정말 쉽지요...

"n"과 "h", "s", "l"의 모든 조합을 원하는대로 쓸 수 있습니다. 정말로 바보같은 것만 제외하고 말입니다. 예를 들어 stolh() 그러니까 "Short to Long Host"는 없습니다. 대신 이런 것들이 있습니다:

함수	설명
htons()	host to network short
htonl()	host to network long
ntohs()	network to host short
ntohl()	network to host long

간단히 말하자면 숫자가 랜선을 타고 나가기 전에 네트워크 바이트 순서로 변환해야 하며 랜선에서 들어올 때에 호스트 바이트 순서로 변환해야 합니다.

64비트 종류에 대해서는 저는 잘 모릅니다. 그리고 만약 부동소수점에 대한 것을 원하신다면 한참 아래에 있는 직렬화 절을 참조하십시오.

달리 말하지 않는 이상 이 문서의 숫자들은 호스트 바이트 순서라고 생각하십시오.

3.3 struct들

마침내 여기까지 왔습니다. 프로그래밍에 대해서 말할 차례입니다. 이 절에서는 소켓 인터페이스가 사용하는 다양한 데이터 형식에 대해서 논할 것이며 그 중 몇몇은 정말로 어렵습니다.

쉬은 것 부터 시작하겠습니다: 소켓 설명자입니다. 소켓 설명자는 아래의 형식입니다.

int

그냥 평범한 int입니다.

여기서부터 이상해집니다. 저와 함께 꾹 참고 따라오십시오.

나의 첫 번째 구조체™—struct addrinfo. 이 구조체는 꽤나 최근의 발명품입니다. 이것은 이후의 사용을 위한 소켓 주소 구조체를 준비하기 위해 사용됩니다. 또한 호스트 이름 찾기나 서비스 이름 찾기 에도 사용됩니다. 나중에 실제 사용 예시를 보시면 이해가 되겠지만 지금은 연결을 만들 때 맨 처음 호출하는 것들 중 하나라고 알아두십시오.

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

이 구조체에 내용을 조금 채워넣은 후에 getaddrinfo() 을 호출할 것입니다. 이 함수는 여러분에게 필요한 모든 것들로 채워진, 이 구조체의 링크드 리스트를 반환할 것입니다.

ai_family필드에서 IPv4나 IPv6을 강제할 수 있고 무엇이든 상관없다면 AF_UNSPEC 으로 둘 수 있습니다. 이것은 여러분의 코드가 IP버전에 무관해지도록 해 주기에 좋습니다.

이것이 링크드 리스트임을 기억하십시오: ai_next는 다음 요소를 가리킵니다. 여러분이 고를 수 있는 여러 개의 결과가 돌아올 수 있다는 의미입니다. 저라면 쓸 수 있는 첫 번째 것을 쓰겠습니다. 그러나 여러분은 다른 비즈니스 요구사항이 있을지도 모릅니다. 저는 이것에 대해서 잘 모릅니다!

struct addrinfo안의 ai_addr이 struct sockaddr 에 대한 포인터임을 보실 수 잇습니다. 여기서부터 IP 주소 구조체의 내부를 살펴볼 때에 지저분해지기 시작하는 곳입니다.

여러분은 대체로 이 구조체들에 쓰기 작업을 할 일이 없을 것입니다. 대체로 여러분의 struct addrinfo을 채우기 위해서 getaddrinfo()을 호출하는 것이 여러분이 해야 할 일의 전부입니다. 그러나 그 안에서 값을 꺼내오기 위해서는 그 안을 들여다봐야만 하므로 이제부터 설명하겠습니다.

(struct addrinfo가 발명되기 전의 코드에서는 모든 정보를 손으로 채워넣어야 했습니다. 저 거친 야생에는 정확히 그런 일을 하는 IPv4코드를 많이 보실 수 있습니다. 이 안내서의 오래된 버전을 포함한 여러 곳에서 말입니다.)

몇몇 struct는 IPv4용이고, 어떤 것은 IPv6용이며 어떤 것은 양쪽 모두에 필요합니다. 뭐가 무엇인지도 적어두겠습니다.

어쨌든 struct sockaddr은 여러 종류의 소켓을 위한 소켓 주소 정보를 저장합니다.

```
struct sockaddr {
    unsigned short    sa_family;    // 주소 계열, AF_xxx
    char              sa_data[14];  // 14 바이트의 프로토콜 주소
};
```

sa_family는 몇 가지 것들 중 하나가 될 수 있는데, 우리가 이 문서에서 하는 모들 일에 대해서는 AF_INET (IPv4) 이나 AF_INET6 (IPv6)가 될 것입니다. sa_data는 소켓을 위한 목적지 주소와 포트 번호가 담겨 있습니다. 이것에 주소를 직접 적어넣는 일은 지루하고 불편합니다.

struct sockaddr을 상대하기 위해서 프로그래머들은 IPv4를 위한 병렬적인 구조체인 struct sockaddr_in (“Internet”의 “in”)을 만들었습니다.

그리고 이것이 중요한 부분입니다: struct sockaddr_in에 대한 포인터는 struct sockaddr에 대한 포인터르 형변환 될 수 있고 그 반대도 가능합니다. 그래서 connect()가 struct sockaddr*을 원하더라도 struct sockaddr_in을 사용할 수 있고 마지막에 형변환만 하면 되는 것입니다!

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)

struct sockaddr_in {
    short int          sin_family;  // 주소 계열, AF_INET
    unsigned short int sin_port;    // 포트 번호
    struct in_addr     sin_addr;    // 인터넷 주소
    unsigned char      sin_zero[8]; // sockaddr 구조체와 같은 크기로 만든다
};
```

이 구조체는 소켓 주소의 요소들을 참조하는 일을 쉽게 해 줍니다. sin_zero (struct sockaddr과 길이를 맞추기 위해서 덧대진 것)은 memset()을 이용해서 0으로 설정되어야 함을 기억하십시오. 또한 sin_family은 struct sockaddr의 sa_family에 대응되며 “AF_INET”으로 설정되어야 함을 기억하십시오. 마지막으로 sin_port은 반드시 네트워크 바이트 순서 로 기록해야 함을 기억하십시오.(htons()을 써야한다는 의미입니다.)

더 깊게 파고들어가봅시다. struct in_addr에는 sin_addr필드가 있습니다. 저것이 무엇일까요? 지나치게 과장할 필요는 없지만 저것은 지금껏 있었던 가장 무서운 공용체 (역자 주 : 하나의 메모리 구역을 서로 다른 데이터타입처럼 읽고 쓸 수 있게 해 주는 C언어의 기능) 중 하나입니다.

```
// (IPv4 전용--IPv6를 위해서는 in6_addr 구조체를 참조하라)

// 인터넷 주소 (역사적인 이유로 존재하는 구조체)
struct in_addr {
    uint32_t s_addr; // 32비트 정수이다 (4 바이트)
};
```

와! 사실 이것은 공용체 였습니다. 그러나 이제 그 시절은 지났습니다. 잘된 일입니다. 그러니까 만약 여러분이 struct sockaddr_in 형으로 ina를 선언했다면 ina.sin_addr.s_addr이 (네트워크 바이트 순서로 적힌) 4바이트의 IP주소를 가리킬 것입니다. 여러분의 시스템이 struct in_addr에 대해서 여전히 형편없는 공용체를 사용해도 여러분은 제가 위에서 한 것과 동일한 방식으로 4바이트 IP주소를 참조할 수 있습니다.(이것은 #define 덕분입니다.)

IPv6에 대해서도 유사한 struct가 있습니다:

```
// (IPv6 전용--IPv4를 위해서는 sockaddr_in 구조체와 in_addr 구조체를 참조하라)

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // 주소 계열, AF_INET6
    u_int16_t       sin6_port;     // 포트 번호, 네트워크 바이트 순서
    u_int32_t       sin6_flowinfo; // IPv6 흐름 정보
    struct in6_addr sin6_addr;     // IPv6 주소
    u_int32_t       sin6_scope_id; // 스코프 아이디
};

struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 주소
};
```

IPv4용 구조체가 그렇듯이 IPv6 구조체도 IPv6주소와 포트번호를 가진 것에 주목하십시오.

지금은 IPv6의 흐름 정보나 Scope ID 필드에 관한 내용은 다루지 않을 것입니다. 이것은 초보자용 안내서이기 때문입니다. :-)

마지막으로 중요한 것은, 여기에 IPv4와 IPv6의 모든 구조체를 담기에 충분히 크게 설계된 struct sockaddr_storage라는 또 하나의 단순한 구조체가 있다는 사실입니다. 때때로 어떤 함수 호출에 대해서 여러분은 그 함수가 여러분의 struct sockaddr를 IPv4주소로 채울지 아니면 IPv6주소로 채울지 알 수 없는 경우가 있습니다. 그러므로 이 병렬 구조체를 넘기면 됩니다. 이것은 좀 더 크다는 점을 제외하면 struct sockaddr과 아주 비슷하며, 여러분은 이것을 여러분이 원하는 형식으로 형변환 할 수 있습니다.

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // 주소 계열

    // 이것들은 모두 패딩이고 구현에 특정적인 내용입니다. 무시하십시오.
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

중요한 것은 여러분이 ss_family 필드에서 주소 계통을 볼 수 있다는 사실입니다. 그것이 AF_INET또는 AF_INET6인지 확인하십시오(IPv4또는 IPv6인지 확인하기 위해서). 그 뒤 필요하다면 struct sockaddr_in 또는 struct sockaddr_in6으로 형변환할 수 있을 것입니다.

3.4 IP 주소, 파트 2

여러분에게는 다행스럽게도, IP 주소를 다룰 수 있게 해주는 많은 함수가 있습니다. 손으로 직접 종류를 알아내고 long에 <<연산자로 값을 채워넣을 필요가 없습니다.(역자 주 : <<은 비트 옮기기 연산자이며 큰 메모리 영역에 작은 값을 집어넣고자 할 때 흔히 사용한다.)

우선 여러분이 struct sockaddr_in ina를 가지고 있다고 합시다. 그리고 저장하고 싶은 두개의 주소, "10.12.110.57"와 "2001:db8:63b3:1::3490"가 있다고 합시다. 여러분이 사용해야 하는 함수는 inet_pton() 입니다. 이것은 숫자와 점 표기법으로 적힌 IP주소를 여러분이 AF_INET또는 AF_INET6 를 지정하는 것에 따라서 struct in_addr또는 struct in6_addr으로 변환합니다. ("pton"는 "presentation to network"(역자 주 : 표현에서 네트워크로)의 약어이며 쉽게 기억하고 싶다면 "printable to network"라고 해도 됩니다.) 변환은 아래와 같이 이루어집니다:

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

(짧은 노트 : 이 일을 하는 예전 방식은 inet_addr()이나 inet_aton()함수를 사용했습니다. 이것들은 이제 구형이고 IPv6과는 동작하지 않습니다.)

위의 코드 예제는 그다지 견고하지 않은데 오류 확인이 없기 때문입니다. inet_pton() 은 오류가 발생하면 -1을 돌려주고 주소가 엉망이면 0을 돌려줍니다. 그러니 결과물을 사용하기 전에 복귀값이 0보다 큰지 확인하십시오.

좋습니다. 이제 여러분은 IP주소 문자열을 그것의 이진 표현으로 바꿀 수 있습니다. 반대로는 어떻게 하는지 궁금하신가요? struct in_addr구조체를 가지고 있고 그것의 숫자와 점 표기법을 출력하길 원하신다면 어떻게 해야할까요? (또는 struct in6_addr을, 그러니까... 16진수와 콜론 표기법으로 출력한다면 어떻게 해야할까요?) 이 경우 여러분은 inet_ntop() 을 사용해야 합니다. ("ntop"는 "network to presentation"을 의미하며 쉽게 기억하려면 "network to printable"이라고 부르셔도 됩니다.) 예제는 아래와 같습니다:

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // IPv4 문자열을 담아둘 공간
struct sockaddr_in sa;      // 이곳에 무엇인가 담겨있다고 가정하자

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // IPv6 문자열을 담아둘 공간
struct sockaddr_in6 sa6;    // 이곳에 무엇인가 담겨있다고 가정하자

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

이 함수를 호출하려면 주소 종류(IPv4 또는 IPv6)와 주소, 결과를 담을 문자열에 대한 포인터, 그 문자열의 최대 길이를 넘겨줘야 합니다. (두 개의 매크로인 INET_ADDRSTRLEN과 INET6_ADDRSTRLEN이 편리하게도 가장 긴 IPv4또는 IPv6문자열을 담아둘 문자열의 크기를 가지고 있습니다.)

(이 일을 하는 오래된 방식에 대한 다른 이야기: 이 변환 작업을 하는 역사적인 함수는 inet_ntoa()입니다. 마찬가지로 구식이고 IPv6에는 작동하지 않습니다.)

마지막으로 이 함수들은 숫자 형태의 IP 주소에만 사용할 수 있습니다. 이 함수들은 "www.example.com"같은 호스트이름에 대한 네임서버 DNS 탐색을 하지 않습니다. 그 작업을 위해서는 다음에 보실 getaddrinfo()을 써야합니다.

3.4.1 사설(또는 분리된) 망

많은 곳들이 보호를 목적으로 네트워크를 외부로부터 숨기는 방화벽을 가지고 있습니다. 그리고 흔히 이 방화벽들은 Network Address Translation 또는 NAT이라는 절차를 통해서 "내부" IP 주소를 (세상의 다른 사람들이 아는)"외부" IP주소로 변환합니다.

벌써 긴장되나요? "저 사람은 이 이상한 것들로 무슨 이야기를 하려는거지?"

진정하고 무알콜(아니면 알콜이 있는)음료수를 준비하세요. NAT은 여러분을 위해서 투명하게 처리되므로(역자 주 : 알 필요가 없게, 보이지 않게) 초보자는 NAT에 대해서 신경쓸 필요도 없습니다. 그러나 저는 여러분이 보는 네트워크 숫자로 인해 헷갈릴 일이 없도록 방화벽 뒤의 네트워크에 대해서 이야기하고 싶었습니다.

예를 들어 제 집에는 방화벽이 있습니다. 저에게는 디지털 가입자 회선(DSL) 회사가 저에게 배정해 준 두 개의 정적 IPv4주소가 있습니다. 그런데 제 네트워크에 있는 컴퓨터는 일곱 대 입니다. 이것이 어떻게 가능하냐고요? 두 개의 컴퓨터가 같은 아이피를 쓸 수는 없습니다. 그렇게 되면 데이터가 어디로 가야할지 알 수 없게 됩니다.

답은 이렇습니다: 컴퓨터들은 아이피 주소를 공유하지 않습니다. 그것들은 2천4백만개의 아이피 주소가 할당된 사설 네트워크에 속해 있습니다. 그것들 전체가 저만을 위한 것입니다. 최소한 저에게는 그렇습니다. 원리는 이렇습니다:

제가 원격 컴퓨터에 로그인하면, 그것은 제가 192.0.2.33에서 로그인했다고 말해줍니다. 그 주소는 제 인터넷 서비스 제공자가 저에게 준 공용 아이피 주소입니다. 그러나 제가 로컬 컴퓨터에게 저의 주소를 물어보면 그것은 10.0.0.5라고 대답합니다. 누가 IP주소를 번역해주는

것일까요? 그렇습니다. 바로 방화벽입니다. 그것이 NAT을 수행하는 것입니다.

10.x.x.x는 완전히 외부와 차단된 네트워크 또는 방화벽 뒤의 네트워크만이 사용하도록 예약된 몇 개의 네트워크 대역 중 하나입니다. 어떤 사설 네트워크 숫자가 사용 가능한지는 RFC 1918[6]에 제시되어 있습니다. 그러나 여러분이 보실 일반적인 것들은 10.x.x.x 또는 192.168.x.x입니다. x에는 보통 0에서 255사이의 수가 들어갑니다. 좀 덜 일반적인 것으로 172.y.x.x가 있습니다. y에는 16부터 31사이의 수가 옵니다.

NAT동작을 수행하는 방화벽 뒤에 있는 망(네트워크)이 이런 사설 네트워크 중 하나 여야만 하는 것은 아닙니다. 그러나 보통 그런 종류입니다.

(재미있는 사실! 제 외부 아이피 주소가 실제로 192.0.2.33인 것은 아닙니다. 192.0.2.x 네트워크는 문서에 "진짜" 아이피 주소가 쓰였다고 믿게 하기 위해서 예약된 대역입니다. 바로 이 안내서에 쓰인 것 처럼 말입니다! 우왕!)

IPv6도 어떤 의미로는 사설망을 가지고 있습니다. 그것들은 fdXX:으로 시작합니다. (미래에는 fcXX:으로 시작할 수도 있습니다.) RFC 4193[7] 문서에 따르자면 그렇습니다. 그러나 NAT과 IPv6은 일반적으로 같이 쓰이지 않습니다. (IPv6 to IPv4 게이트웨이(gateway)같은 것을 만들지 않는 다면 말입니다. 그리고 그것은 이 문서의 범위를 넘어섭니다.) 아무튼 이론적으로는 여러분에게는 너무나 많은 주소가 있어서 더이상 NAT을 쓸 필요가 없을 것입니다. 그럼에도 여러분이 외부와 통하지 않는 주소를 할당하고 싶다면, 위에 적은 대역을 쓸 수 있다는 의미입니다.

[6] https://tools.ietf.org/html/rfc1918
[7] https://tools.ietf.org/html/rfc4193

Chapter 4

IPv4에서 IPv6으로 점프하기

"아무튼 저는 IPv6에서 동작하려면 제 코드의 어디를 바꿔야 하는지 알고싶단 말입니다! 당장 알려주세요!"

좋아요! 좋습니다!

여기 적을 내용의 대부분은 제가 위에서 다룬 내용들이지만 이것은 참을성 없는 분들을 위한 짧은 버전입니다. (물론 이것보다 더 많은 내용이 있겠지만, 이 안내서에 있는 내용은 이정도입니다.)

1. 우선 struct sockaddr 정보를 얻어내기 위해서 수작업 대신 getaddrinfo() 함수를 사용하십시오. 이렇게 하면 여러분은 IP버전에 신경쓰지 않을 수 있고, 뒤따르는 많은 후속 작업을 할 필요가 없게 해 줍니다.
2. IP 버전에 관계된 것을 하드코딩하는 곳을 찾아낼 때마다 헬퍼 함수로 감싸두는 처리를 해 두십시오.
3. AF_INET를 AF_INET6로 바꾸십시오.
4. PF_INET를 PF_INET6로 바꾸십시오.
5. INADDR_ANY 대입을 in6addr_any대입으로 바꾸십시오. 이런 차이가 있습니다:

   ```
   struct sockaddr_in sa;
   struct sockaddr_in6 sa6;

   sa.sin_addr.s_addr = INADDR_ANY;  // use my IPv4 address
   sa6.sin6_addr = in6addr_any; // use my IPv6 address
   ```

   또한 struct in6_addr을 선언할 때 IN6ADDR_ANY_INIT을 초기값으로 사용할 수 있습니다. 아래와 같이 합니다.

   ```
   struct in6_addr ia6 = IN6ADDR_ANY_INIT;
   ```

6. struct sockaddr_in 대신에 struct sockaddr_in6을 사용하시고, 필요한 필드에 "6"을 적절히 덧붙이십시오. (위의 structs을 참고하십시오) sin6_zero필드는 없습니다.
7. struct in_addr 대신에 struct in6_addr를 사용하시고, 필요한 필드에 "6"을 적절히 덧붙이십시오. (위의 structs을 참고하십시오)
8. inet_aton()이나 inet_addr() 대신에 inet_pton()을 사용하십시오.
9. inet_ntoa() 대신에 inet_ntop()을 사용하십시오.
10. gethostbyname()대신에 더 뛰어난 getaddrinfo()를 사용하십시오.
11. gethostbyaddr() 대신에 더 뛰어난 getnameinfo()를 사용하십시오. (gethostbyaddr()가 IPv6을 위해서도 여전히 작동하기는 합니다).
12. INADDR_BROADCAST는 더 이상 작동하지 않습니다. 대신 IPv6 멀티캐스트를 사용하십시오.

끝!

Chapter 5

시스템 콜이 아니면 죽음을

이 절에서 우리는 유닉스 장치나 기타 소켓 API를 지원하는 다른 장치(BSD, 윈도우즈, 리눅스, 맥, 여러분이 가진 다른 장치)에서 네트워크 기능에 접근할 수 있게 해 주는 시스템 호출(System call)(과 다른 라이브러리 호출 (Library Call))에 대해서 다룰 것입니다. 이런 함수 중 하나를 호출하면 커널이 넘겨받고 여러분을 위해 모든 일을 자동으로 마법같이 처리합니다.

대부분의 사람들이 어려워하는 점은 이 함수들을 어떤 순서로 호출해야 하는가 입니다. 이미 찾아보셨겠지만 그런 쪽으로는 man페이지는 아무 쓸모도 없습니다. 그 끔찍한 상황을 해결하기 위해 시스템콜들을 정확히(대략) 여러분의 프로그램에서 호출해야 하는 순서 그대로 아래에 이어지는 절들에 제시했습니다.

그러니까 여기에 있는 몇몇 예제 코드와 우유, 과자(이것들은 직접 준비하셔야 합니다) 그리고 두둑한 배짱과 용기만 있다면 여러분은 인터넷의 세계에서 존 포스텔의 아들처럼 데이터를 나를 수 있게 될 것입니다.(역자 주 : John Postel은 인터넷의 초기에 큰 기여를 한 컴퓨터 과학자 중 한 명입니다.)

(아래의 예제 코드들은 대개 필수적인 에러코드를 간략함을 얻기 위해서 생략했음을 기억하십시오. 그리고 예제 코드들은 대개 getaddrinfo()의 호출이 성공하고 연결리스트로 적절한 결과물을 돌려준다고 가정합니다. 이런 상황은 독립 실행형 프로그램에서는 제대로 처리되어 있으니, 그것들을 지침으로 삼으십시오.)

5.1 getaddrinfo()—발사 준비!

, 이것은 여러 옵션을 가진 진짜 일꾼입니다. 그러나 사용법은 사실 꽤 간단합니다. 이것은 여러분이 나중에 필요로 하는 struct들을 초기화합니다.

역사 한토막 : 예전에는 DNS 검색을 위해서 gethostbyname()을 호출해야 했습니다. 그리고 그 정보를 수작업으로 struct sockaddr_in에 담고 이후의 호출에서 사용해야 했습니다.

고맙게도 더 이상은 그럴 필요가 없습니다. (여러분이 IPv4와 IPv6환경 모두에서 동작하는 코드를 짜고싶다면 그래서도 안 됩니다!) 요새는 getaddrinfo()이라는 것이 있어서 DNS 와 서비스 이름 검색, struct내용 채워넣기 등을 포함해서 여러분이 필요로 하는 모든 일을 해 줍니다.

이제 살펴봅시다!

(역자 주 : 아래에서부터 입니다, 하세요 등의 표현 대신 이다, 하라 등의 간결한 어미를 섞어서 씁니다.)

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

int getaddrinfo(const char *node,     // e.g. "www.example.com" 또ㅡㄴ IP
                const char *service,  // e.g. "http" 또는 포트 숫자를 ""안에 감싸서 넣는다.
                const struct addrinfo *hints,
                struct addrinfo **res);
```

이 함수에는 3개의 입력 매개변수를 넘겨줍니다. 그리고 결과 연결리스트의 포인터인 res를 돌려받습니다.

node매개변수는 접속하려는 호스트 이름이나 IP주소입니다.

다음 매개변수는 service입니다. 이것은 "80"같은 포트 번호나 "http", "ftp", "telnet" 또는 "smtp"같은 특정한 서비스 이름이 될 수 있습니다. (IANA 포트 목록[1] 혹은 여러분이 유닉스 장치를 쓴다면 /etc/services에서 볼 수 있습니다)

마지막으로 hints매개변수는 여러분이 관련된 정보로 이미 채워넣은 struct addrinfo 를 가리킵니다.

여기에 여러분이 호스트 IP주소의 포트 3490을 듣고자 할 때의 함수 호출 예제가 있습니다. 이것이 듣기 작업이나 네트워크 설정을 하지는 않음을 기억하십시오. 이것은 단지 나중에 사용할 구조체들을 설정할 뿐입니다.

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // 결과를 가리킬 것이다

memset(&hints, 0, sizeof hints); // 구조체를 확실히 비워두라
hints.ai_family = AF_UNSPEC;     // IPv4 이든 IPv6 이든 상관없다
hints.ai_socktype = SOCK_STREAM; // TCP 스트림 소켓
hints.ai_flags = AI_PASSIVE;     // 내 주소를 넣어달라

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
   fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
   exit(1);
}

// servinfo는 이제 1개 혹은 그 이상의 addrinfo 구조체에 대한 연결리스트를 가리킨다

// ... servinfo가 더이상 필요없을 때까지 모든 작업을 한다...

freeaddrinfo(servinfo); // 연결리스트를 해제
```

ai_family을 AF_UNSPEC으로 설정해서 IPv4든 IPv6이든 신경쓰지 않음을 나타낸 것에 주목하라. 만약 특정한 하나를 원한다면 AF_INET이나 AF_INET6을 쓸 수 있다.

AI_PASSIVE도 볼 수 있다. 이것은 getaddrinfo()에게 소켓 구조체에 내 로컬 호스트의 주소를 할당해달라고 말해준다. 이것은 여러분이 하드코딩할 필요를 없애주기에 좋다. (아니면 위에서 NULL을 넣은 getaddrinfo()의 첫 번째 매개변수에 특정한 주소를 넣을 수 있다.)

이렇게 함수를 호출한다. 오류가 있다면(getaddrinfo()이 0이 아닌 값을 돌려준다면) 보다시피 그 오류를 gai_strerror()함수를 통해서 출력할 수 있다. 만약 모든 것이 제대로 동작한다면 servinfo는 각각이 우리가 나중에 쓸 수 있는 struct sockaddr나 비슷한 것을 가진 struct addrinfo의 연결리스트를 가리킬 것이다. 멋지다!

마지막으로 getaddrinfo()가 은혜롭게 우리에게 할당해 준 연결리스트를 다 썼다면 우리는 freeaddrinfo()을 호출해서 그것을 할당 해제할 수 있습니다. (반드시 해야합니다.)

여기에 여러분이 특정한 주소, 예를 들어 "www.example.net"의 3490포트에 잡속하고자 하는 클라이언트일 경우의 호출 예제가 있습니다. 다시 말씀드리지만 이것으로는 실제 연결이 이루어지지 않습니다. 그러나 이것은 우리가 나중에 사용할 구조체를 설정해줍니다.

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // 결과물을 가리킬 것임

memset(&hints, 0, sizeof hints); // 반드시 비워둘 것
hints.ai_family = AF_UNSPEC;     // IPv4나 IPv6은 신경쓰지 않음
hints.ai_socktype = SOCK_STREAM; // TCP 스트림 소켓

// 연결 준비
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);
```

[1] https://www.iana.org/assignments/port-numbers

```
// servinfo는 이제 1개 혹은 그 이상의 addrinfo 구조체에 대한 연결리스트를 가리킨다

// 등등.
```

servinfo은 모든 종류의 주소 정보를 가진 연결리스트라고 계속 이야기하고 있다. 이 정보를 보기 위한 짧은 시연 프로그램을 작성해보자. 이 짧은 프로그램[2] 은 여러분이 명령줄에 적는 호스트의 IP주소들을 출력한다.

```
/*
** showip.c -- 명령줄에서 주어진 호스트의 주소들을 출력한다.
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: showip hostname\n");
        return 1;
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // 버전을 지정하려면 AF_INET또는 AF_INET6을 사용
    hints.ai_socktype = SOCK_STREAM;

    if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
        return 2;
    }

    printf("IP addresses for %s:\n\n", argv[1]);

    for(p = res;p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;

        // 주소 자체에 대한 포인터를 받는다. IPv4와 IPv6은 필드가 다르다.
        if (p->ai_family == AF_INET) { // IPv4
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
            ipver = "IPv4";
        } else { // IPv6
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
```

[2] https://beej.us/guide/bgnet/examples/showip.c

```
        addr = &(ipv6->sin6_addr);
        ipver = "IPv6";
      }

      // IP주소를 문자열로 변환하고 출력한다.
      inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
      printf("  %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res); // 연결 목록을 해제한다.

    return 0;
}
```

보다시피 이 코드는 당신이 명령줄에 넘기는 것이 무엇이든 getaddrinfo()을 호출한다. 그리고 res에 연결목록의 포인터를 넘겨준다. 그래서 우리는 이 목록을 순회해서 출력하거나 다른 일을 할 수 있다.

(저 예제코드에는 IP 버전에 따라 다른 종류의 struct sockaddr을 처리해야 하는 흉한 부분이 있다. 그 점에 대해서 사과한다. 그러나 더 나은 방법이 있는지는 모르겠다.)

실행 예제! 모두가 스크린샷을 좋아합니다.

```
$ showip www.example.net
IP addresses for www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
IP addresses for ipv6.example.com:

  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

이제 저것을 다룰 수 있으니, getaddrinfo()에서 얻은 결과를 다른 소켓 함수에 넘기고 결과적으로는 네트워크 연결을 성립할 수 있도록 해 보자! 계속 읽어보라!

5.2 socket()—파일 설명자를 받아오라!

더 이상 미룰 수가 없을 듯 하다. 이제 socket() 시스템 콜에 대해서 이야기해야 한다. 개요는 이렇다.

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

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

그러나 이 인수들이 무엇인지 모를 것이다. 이것들은 어떤 종류의 소켓을 원하는지 정할 수 있게 해 준다.(IPv4 또는 IPv6, 스트림 혹은 데이터그램, TCP 혹은 UDP)

사용자들이 그 값을 직접 적어야 했고, 지금도 그렇게 할 수 있다. (domain은 PF_INET이나 PF_INET6이고, type은 SOCK_STREAM또는 SOCK_DGRAM 이며, protocol은 주어진 type에 적절한 값을 자동으로 선택하게 하려면 0을 넘겨주거나 “tcp”나 “udp” 중 원하는 프로토콜의 값을 얻기 위해서 getprotobyname() 을 쓸 수도 있다.)

(이 PF_INET은 sin_family필드에 넣어주는 AF_INET와 유사한 것이다. 이것을 이해하려면 짧은 이야기가 필요하다. 아주 먼 옛날에는 어쩌면 하나의 주소 계통(Address Family) (“AF_INET”안에 들어있는 “AF”)가 여러 종류의 프로토콜 계통(Protocol Family)(“PF_INET”의 “PF”))을 지원할 것이라고 생각하던 시절이 있었다. 그런 일은 일어나지 않았다. 그리고 모두 행복하게 오래오래 잘 살았다. 이런 이야기다. 그래서 할 수 있는 가장 정확한 일은 struct sockaddr_in에서 AF_INET을 쓰고 socket()에서 PF_INET을 사용하는 것이다.

아무튼 이제 충분하다. 여러분이 정말로 하고싶은 일은 getaddrinfo()을 호출한 결과로 돌아오는 값을 아래와 같이 socket()에 직접 넘겨주는 것이다.

```
int s;
struct addrinfo hints, *res;

// 탐색 시작
// ["hints"구조체는 이미 채운 것으로 친다]
getaddrinfo("www.example.com", "http", &hints, &res);

// 다시 말하지면 원래는 (이 안내서의 예제들이 하듯이) 첫 번째 것이 좋다고
// 가정하는 대신 getaddrinfo()에 대해서 오류 확인을 하고
// "res"링크드 리스트를 순회해야 한다.
// client/server절의 진짜 예제들을 참고하라.

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

socket()은 단순하게 이후의 시스템 호출에서 쓸 수 있는 소켓 설명자 를 돌려준다. 오류가 있으면 -1을 돌려준다. 전역 변수인 errno가 오류의 값으로 설정된다. (자세한 정보는 errno 의 맨페이지를 참고하라.)

좋다. 그러면 이제 이 소켓을 어디에 쓰는가? 정답은 아직 못 쓴다는 것이다. 실제로 쓰기 위해서는 안내서를 더 읽고 이것이 동작하게 하기 위한 시스템 호출을 더 해야 한다.

5.3 bind()—나는 어떤 포트에 있는가?

소켓을 가지면 여러분의 기계의 포트에 연동하고 싶을 것이다. (이 작업은 보통 여러분이 listen() 으로 특정 포트에서 들어오는 연결을 듣고자(listen) 할 때 이루어진다. —다중 사용자 네트워크 게임들은 "192.168.5.10의 3490포트에 연결합니다"라고 말할 때 이런 작업을 한다.) 포트 번호는 커널이 특정 프로세스의 소켓 설명자를 들어오는 패킷과 연관짓기 위해서 사용한다. 만약 여러분이 connect()만 할 생각이라면 bind()는 불필요하다. 그러나 재미를 위해 읽어두자.

이것이 bind() 시스템 콜의 개요다.

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

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

sockfd은 socket()이 돌려준 소켓 파일 설명자이다. my_addr은 여러분의 주소, 말하자면 포트와 IP주소를 가진 struct sockaddr 에 대한 포인터이다. addrlen은 그 주소의 바이트 단위 길이이다.

으엑. 한 번에 많이 배웠다. 프로그램이 실행되는 호스트의 3490번 포트에 소켓을 바인드하는 예제를 보자.

```
struct addrinfo hints, *res;
int sockfd;

// 먼저 getaddrinfo()으로 구조체에 정보를 불러온다.

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // IPv4나 IPv6 중 아무 것이나 쓴다
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // IP는 나의 아이피로 채운다.

getaddrinfo(NULL, "3490", &hints, &res);

// 소켓을 만든다.

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

```
// getaddrinfo()에 넘겼던 포트에 바인드한다.

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

AI_PASSIVE플래그를 써서 프로그램에게 실행중인 호스트의 IP에 바인드하라고 알려줍니다. 특정한 로컬 IP주소에 바인드하고싶다면 AI_PASSIVE을 버리고 getaddrinfo()의 첫 번째 인수로 IP주소를 넣으라.

bind()도 오류가 발생하면 -1을 돌려주고 errno을 오류의 값으로 설정한다.

많은 오래된 코드들이 bind()을 호출하기 전에 struct sockaddr_in을 직접 채워넣는다. 이것은 분명히 IPv4 전용이지만 같은 일을 IPv6에 대해서도 못 할 이유는 없다. 단지 getaddrinfo()을 쓰는 편이 일반적으로 더 쉽다. 어쨌든 예전 코드는 이런 방식이다.

```
// !!! 이것은 예전 방식이다 !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);    // short, 네트워크 바이트 순서
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

위의 코드에서 당신의 로컬 IP 주소에 바인드하고 싶었다면(위의 AI_PASSIVE처럼) s_addr필드에 INADDR_ANY을 대입할 수 있다. IPv6버전의 INADDR_ANY은 당신의 struct sockaddr_in6의 sin6_addr필드에 대입해야 하는 전역변수인 in6addr_any이다. (변수 초기화식에 쓸 수 있는 IN6ADDR_ANY_INIT이라는 매크로도 있다.)

bind()을 쓸 때 주의해야 할 것 : 포트 번호는 낮은 것을 쓰지 말 것. 1024번 아래의 모든 포트는 예약되어 있다(슈퍼유저가 아닌 이상)! 그 위의 포트 번호는 (다른 프로그램이 이미 쓰고 있지 않다면) 65535까지 아무 것이나 쓸 수 있다.

눈치챌 수 있듯이 때때로 서버를 다시 실행하려고 하면 bind()가 실패하고 "주소가 이미 사용중입니다.."라고 할 때가 있다. 그것은 연결되었던 소켓 중 일부가 여전히 커널에서 대기중이고 포트를 사용하고 있다는 것을 의미한다. 여러분은 그것이 정리될 때까지 1분 정도를 기다리거나 당신의 프로그램이 포트를 재사용할 수 있도록 하는 코드를 넣을 수도 있다.

```
int yes=1;
//char yes='1'; // 솔라리스는 이것을 사용

// "주소가 이미 사용중입니다"라는 오류 메시지를 제거
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes) == -1) {
   perror("setsockopt");
   exit(1);
}
```

bind()에 대해서 한마디 더: 이 함수를 호출할 필요가 전혀 없는 경우도 있습니다. connect()를 호출해서 원격 장치에 연결하려고 하고, 로컬 포트에 대해서는 신경쓰지 않는다면(telnet의 경우 처럼 원격지 포트만 신경쓰는 경우) connect()가 자동으로 소켓이 바인드되지 않았는지 확인하고 필요하다면 사용하지 않은 로컬 포트에 bind()해줄 것입니다.

5.4 connect()—이봐, 안녕!

몇 분만 여러분이 텔넷 응용프로그램이 되었다고 생각해봅시다. 여러분의 사용자들이 소켓 파일 설명자를 얻기 위해서 여러분에게 명령을 내립니다 (영화 트론 에서처럼요). 여러분은 그에 따라 socket()을 호출합니다. 다음으로 사용자가 여러분에게 "10.12.110.57"의 "23"번 포트(텔넷 표준 포트)에 연결하라고 합니다. 어떻게 해야할까요?

응용프로그램 여러분, connect()에 대한 절을 읽는 중이라니 운이 좋습니다! 이 절은 원격 호스트에 어떻게 연결하는지에 대해 알려줍니다. 거침없이 읽어봅시다! 낭비할 시간이 없습니다!

connect()에 대한 호출은 아래와 같습니다:

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

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

sockfd는 socket()함수 호출이 돌려주는 우리의 친근한 이웃인 소켓 파일 설명자 입니다. serv_addr는 struct sockaddr이고 목적지 포트와 아이피 주소를 담고 있습니다. addrlen은 서버 주소 구조체의 바이트단위 길이를 담고 있습니다.

모든 정보는 멋진 getaddrinfo()호출의 결과에서 추출할 수 있습니다.

이해가 되기 시작합니까? 여기서는 대답을 들을 수 없으니 그럴 것이라 생각하겠습니다. "www.example.com"의 3490포트로 소켓 연결을 만드는 예제를 살펴봅시다:

```
struct addrinfo hints, *res;
int sockfd;

// getaddrinfo()으로 주소 구조체를 채웁니다:

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// 소켓을 만듭니다:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// 연결합니다!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

다시 이야기하자면 구식 프로그램들은 connect()에 넘겨줄 struct sockaddr_in 을 직접 채워넣었습니다. 그렇게 하고싶다면 해도 됩니다. 위에 있는 bind() 절 에서 비슷한 내용을 참고하십시오.

connect()의 복귀값을 확인하는 것을 잊지 마십시오. 오류가 발생하면 -1을 돌려주고 errno변수를 설정할 것입니다.

우리가 bind()를 호출하지 않았음에 주목하십시오. 간단히 말하자면 우리의 로컬 포트 번호에 대해서는 신경쓰지 않습니다. 우리가 어디로 가는지만 신경씁니다 (원격지 포트). 커널이 우리 대신 로컬 포트를 고를 것입니다. 우리가 접속하는 사이트는 이 정보를 자동으로 우리에게서 얻어냅니다. 신경쓰실 필요가 없습니다.

5.5 listen()—누가 연락 좀 해주실래요?

이제 흐름이 변할 때입니다. 우리가 원격지 호스트에 접속하고 싶지 않은 경우라면 어떻게 하시겠습니까? 재미로 하는 말이지만, 들어오는 연결을 기다리고 그것을 어떤 방식으로 다루고자 한다면 어떻게 하시겠습니까? 그 과정은 두 단계입니다. 먼저 listen()를 호출하고, accept()를 씁니다.(아래를 참고하십시오.)

listen()함수 호출은 꽤 단순하지만 약간의 설명이 필요합니다:

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

sockfd은 socket()시스템 함수 호출로 얻어온 평범한 소켓 파일 설명자입니다. backlog는 들어오는 큐에 허용되는 연결의 숫자입니다. 이것이 무슨 뜻인지 궁금하십니까? 들어오는 연결들은 여러분이 accept()를 해주기 전까지(아래를 참고하십시오) 이 큐 안에서 기다릴

것이고 이것은 몇 개의 연결이 대기할 수 있는가를 정합니다. 대개의 시스템은 이 값을 조용히 20 정도로 제한합니다. 그러나 5나 10정도의 값으로 설정해도 괜찮을 것입니다.

또 평소와 다름없이 listen()도 오류가 발생할 경우 -1을 돌려주고 errno을 설정할 것입니다.

아마도 상상하실 수 있겠지만 서버가 특정 포트에서 실행되도록 하기 위해서는 listen()을 호출하기 전에 bind()을 호출해야 합니다. (여러분의 친구들에게 어떤 포트로 연결해야 할지 말해줄 수 있어야 합니다.) 이런 식입니다.

```
getaddrinfo();
socket();
bind();
listen();
/* accept()는 아래에 온다 */
```

I'll just leave that in the place of sample code, since it's fairly self-explanatory. (The code in the accept() section, below, is more complete.) The really tricky part of this whole sha-bang is the call to accept().

5.6 accept()—"3490포트에 접속해주셔서 감사합니다.."

각오하십시오! accept()함수는 조금 이상합니다. 이렇게 돌아갑니다: 아주 먼 곳에 있는 누군가가 여러분의 장치에 connect() 함수로 연결하려고 합니다. 여러분은 특정 포트에서 listen()을 실행하고 있습니다. 그들의 연결은 accept()로 받아들여질 때까지 대기열에 쌓일 것입니다. 여러분은 accept()을 해서 대기중인 연결을 받아들이겠다고 알려줍니다. accept()는 이 연결만을 위해서 쓸 완전히 새로운 소켓 파일 설명자 를 돌려줄 것입니다. 그렇습니다! 갑자기 send()와 recv()를 쓸 수 있는 두 개 의 소켓 파일을 가지게 된 것입니다.

호출은 아래와 같이 합니다:

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

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

sockfd은 listen()을 하고있는 소켓 설명자입니다. 어렵지 않습니다. addr은 대개 로컬 struct sockaddr_storage에 대한 포인터입니다. 여기에 들어오는 연결의 정보가 들어가게 됩니다(그리고 그것을 통해서 어떤 호스트가 어떤 포트에서 여러분을 호출하고 있는지 알 수 있습니다.) addrlen은 sockaddr_storage을 accept()에 넘기기 전에 sizeof(struct sockaddr_storage)으로 설정되어야 하는 로컬 정수 변수입니다. accept()는 addr에 addrlen의 크기 이상의 바이트를 적지 않을 것입니다. 예상하셨습니까? accept()도 오류가 발생하면 -1을 돌려주고 errno에 값을 설정합니다. 전혀 예상하지 못하셨으리라 생각합니다.

전과 마찬가지로 한 번에 많은 내용입니다. 여러분의 독서를 위한 예제 코드 조각입니다:

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

#define MYPORT "3490"  // 사용자들이 접속할 포트
#define BACKLOG 10     // 대기열에 몇 개의 연결이 대기할 수 있는가

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! 이 호출들에 대한 오류 확인을 잊지 마십시오 !!

    // getaddrinfo()으로 정보를 채워넣습니다:
```

```
  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC;  // IPv4또는 IPv6, 아무것이나 씁니다.
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;     // 나를 위해 자동으로 내 IP를 채워넣을 것.

  getaddrinfo(NULL, MYPORT, &hints, &res);

  // 소켓을 만들고, 바인드하고, 듣기 시작:

  sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
  bind(sockfd, res->ai_addr, res->ai_addrlen);
  listen(sockfd, BACKLOG);

  // 들어오는 연결을 받습니다:

  addr_size = sizeof their_addr;
  new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

  // new_fd 소켓 설명자에서 통신할 준비 완료!
  .
  .
  .
```

다시 말하지만 모든 send()와 recv() 호출에 대해서 new_fd를 사용할 것입니다. 만약 단 한 개의 연결만을 받아들이길 원한다면 추가적인 연결이 같은 포트를 통해 들어오는 것을 막기 위해서 sockfd을 close()처리할 수 있습니다.

5.7 send()와 recv()—Talk to me, baby!

(역자 주 : Talk to me, baby!는 Elmore James의 노래입니다. 그러나 원저자의 의도가 이것인지 확실하지는 않습니다.) 이 두 함수들은 스트림 소켓이나 연결된 데이터그램 소켓을 통해 통신하기 위해서 쓰입니다. 일반적인 연결되지 않은 데이터그램 소켓을 쓰고싶다면 sendto()과 recvfrom() 절을 보시면 됩니다.

send() 함수:

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

sockfd 은 데이터를 보내고 싶은 소켓 설명자(socket()으로 만들었든 accept()로 만들었든)입니다. msg는 당신이 보낼 데이터에 대한 포인터이며, len은 그 길이입니다.

예제 코드는 이렇게 될 수 있겠습니다:

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

한 방에 모든 것을 보냈습니다. 다시 강조하지만 오류가 발생하면 -1이 반환되고 errno가 오류 번호로 설정됩니다.

recv()함수는 많은 면에서 유사합니다:

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

sockfd은 읽어들일 소켓 설명자이며, buf는 정보를 읽어들일 버퍼이고, len은 버퍼의 최대 길이이고 flags는 여기서도 0으로 설정될 수 있습니다. (플래그 정보에 대해서는 recv()의 man page를 참고하십시오.)

recv()는 실제로 버퍼에 읽어들인 바이트의 수를 돌려주거나 오류가 발생할 경우 (errno 를 적절한 값으로 설정하고) -1을 돌려줍니다.

잠깐! recv()는 0을 돌려줄 수 있습니다. 이것은 한 가지 의미입니다: 원격지 측에서 당신에 대한 연결을 닫은 것입니다! 복귀값 0은 recv()가 연결이 끊어졌음을 알려주는 방식입니다.

자, 정말 쉽지않습니까? 이제 여러분은 스트림 소켓에서 자료를 주고받을 수 있습니다. 와! 이제 여러분은 유닉스 네트워크 프로그래머입니다!

5.8 sendto()와 recvfrom()—Talk to me, DGRAM-방식

"이제 다 깔끔하고 좋네요"라고 말씀하시는 소리가 들립니다. "그렇지만 연결이 없는 데이터그램 소켓은 어떻게 처리하지요?"라고도 하시는군요. 문제 없습니다, 토모다치여(역자 주 : 원문은 amigo). 딱 맞는 것이 있습니다.

데이터그램 소켓은 원격지 호스트에 연결되어 있지 않으므로, 패킷을 보낼 때에 필요한 정보는 조금 다릅니다. 그렇습니다. 목적지 주소가 필요합니다. 이런 식입니다:

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

보시다시피 send()와 같지만 두 개의 정보가 더 있습니다. to는 목적지의 IP주소와 포트를 담은 struct sockaddr 이며(아마도 여러분이 형변환해서 사용하실 struct sockaddr_in이나 struct sockaddr_in6 또는 struct sockaddr_storage일 것입니다.) tolen은 내부적으로는 int이며 간단하게 sizeof *to나 sizeof(struct sockaddr_storage)로 설정하면 됩니다.

목적지 주소 구조체를 얻으려면 getaddrinfo()이나 아래의 recvfrom()을 사용하시거나 수작업으로 값을 채워넣을 수도 있습니다.

send()와 마찬가지로 sendto()도 실제로 보낸 바이트 수를 돌려줍니다. (그 말은 보내려고 한 바이트의 수보다 적은 수가 돌아올 수도 있다는 의미입니다.) 오류가 발생하면 -1을 돌려줍니다.

이와 유사한 관계가 recv()과 recvfrom()입니다. recvfrom()의 개요는 이렇습니다:

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

또 다시 이것은 몇 개의 추가적인 필드가 있는 recv()과 같습니다. from은 근원지 장치의 아이피 주소와 포트로 채워진 로컬 struct sockaddr_storage에 대한 포인터입니다. fromlen은 로컬 int에 대한 포인터이며 sizeof *from이나 sizeof(struct sockaddr_storage)으로 초기화되어야 합니다. 함수가 반환될 때 fromlen은 from에 실제로 저장된 주소의 길이로 설정되어 있을 것입니다.

recvfrom()은 받은 바이트의 갯수를 반환하며 오류가 나면 (errno를 적절히 설정하고) -1을 돌려줍니다.

여기 질문이 하나 있을 것입니다: 왜 우리는 struct sockaddr_storage을 소켓의 타입으로 사용하는가? 왜 그냥 struct sockaddr_in을 쓸 수 없는가? 이유는 보시다시피 우리가 IPv4나 IPv6중 하나에 얽메이고 싶지 않기 때문입니다. 그래서 우리는 양쪽 모두에 충분히 크고 일반적인 struct sockaddr_storage을 사용합니다.

(그럼... 여기에서 다른 질문 하나: 왜 struct sockaddr을 모든 주소를 담을 수 있을 정도로 크게 만들지 않았는가? 우리는 일반 목적의 struct sockaddr_storage 을 다시 일반 목적의 struct sockaddr으로 형변환하고 있습니다! 이런 동작은 과하고 불필요해 보입니다. 여기에 대한 대답은 그냥 이 struct sockaddr은 만들어질 때부터 그렇게 크지 않았다는 것이고, 이제와서 그것을 바꾸는 것은 문제의 소지가 있다는 것입니다. 그래서 그들은 그냥 새로운 타입을 만들었습니다.) (역자 주 : struct sockaddr은 소켓 통신의 초기에 만들어진 구조체이므로 IPv6을 담기에 충분하지 않은 것은 당연한 일입니다.)

만약 여러분이 데이터그램 소켓을 connect()하게 되면 모든 통신에 send()와 recv()을 쓸 수 있음을 기억하십시오. 소켓 자체는 여전히 데이터그램 소켓일 것이고 패킷은 여전히 UDP를 사용할 것이지만 소켓 인터페이스가 자동으로 여러분을 위해서 목적지와 원천지 정보를 추가할 것입니다.

5.9 close()와 shutdown()—내 앞에서 꺼져!

휴! 여러분은 하루 종일 send()와 recv()을 사용했고, 이제 충분합니다. 이제 여러분의 소켓 설명자를 닫을 준비가 되었습니다. 이건 쉽습니다. 그냥 평범한 유닉스 파일 설명자 닫기 함수인 close() 를 쓸 수 있습니다:

close(sockfd);

이것은 해당 소켓에 대한 후속 읽기와 쓰기를 방지할 것입니다. 원격지에서 이 소켓을 쓰거나 읽으려는 모든 시도는 오류를 반환할 것입니다.

소켓이 어떻게 닫히는지 좀 더 조절하고 싶은 경우에 shutdown() 함수를 사용할 수 있습니다. 이것은 특정 방향으로의 통신만 끊는 일을 할 수 있으며 양쪽 모두 막을 수도 있습니다(마치 close()가 하듯이). 개요는 이렇습니다:

int shutdown(int sockfd, int how);

sockfd는 종료하고 싶은 소켓 파일 설명자이고, how는 다음 중 하나입니다:

how	효과
0	후속 수신이 금지됩니다.
1	후속 송신이 금지됩니다.
2	후속 송수신이 금지됩니다. (close()처럼)

shutdown()은 성공시에 0을 반환하고, 오류가 발생하면 (errno를 적절한 값으로 설정하고) -1을 반환합니다.

연결되지 않은 데이터그램 소켓에 기꺼이 shutdown()을 해주신다면, 그것은 단순히 해당 소켓에 send()와 recv()를 사용할 수 없도록 만들 것입니다(데이터그램 소켓에 connect()를 사용하면 이 두 함수를 사용할 수 있음을 기억하십시오).

shutdown()이 실제로 파일 설명자를 닫지는 않음에 주목하십시오. 소켓 설명자를 해제하기 위해서는 close()를 호출해야 합니다.

별 것 없군요.

(예외적으로 여러분이 윈도우즈와 Winsock을 사용하실 경우 close()대신 closesocket()을 호출해야 합니다.)

5.10 getpeername()—누구십니까?

이 함수는 너무 쉽습니다.

너무 쉬워서 이 함수에 별도의 장을 주지도 않았습니다. 아무튼 알려드리겠습니다.

getpeername()함수는 연결된 스트림 소켓의 반대편 끝에 누가 있는지를 알려줄 것입니다. 개요입니다:

#include <sys/socket.h>

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

sockfd는 연결된 스트림 소켓의 설명자입니다. addr은 연결의 반대편 끝에 대한 정보를 담을 struct sockaddr (또는 struct sockaddr_in) 에 대한 포인터입니다. addrlen은 int에 대한 포인터이며 sizeof *addr이나 sizeof(struct sockaddr)으로 초기화되어야 합니다. (역자 주 : 이 함수도 IPv6과 동작하기 위해서 struct sockaddr_storage 을 사용할 수 있습니다.)

이 함수는 오류가 발생하면 -1을 돌려주고 errno를 알맞게 설정합니다.

여러분이 상대방의 주소를 가지면 그것을 inet_ntop(), getnameinfo() 또는 gethostbyaddr()에 넣어서 화면에 출력하거나 추가적인 정보를 가져올 수 있습니다. 그들의 로그인 이름을 가져올 수는 없습니다. (좋습니다, 좋아요. 만약 저쪽 컴퓨터가 ident 데몬을 실행중이라면 가능합니다. 그러나 그것은 이 문서의 범위를 넘어섭니다. 더 자세한 정보를 원한다면 RFC 1413[3]을 참고하십시오.)

5.11 gethostname()—나는 누구인가?

getpeername()보다 더 쉬운 것이 바로 gethostname() 함수입니다. 이것은 여러분의 프로그램이 실행되고 있는 컴퓨터의 이름을 돌려줍니다. 돌려받은 이름은 위에 있는 getaddrinfo() 을 써서 여러분의 로컬 장치의 IP주소를 알아내는 일에 쓰일 수 있습니다.

[3] https://tools.ietf.org/html/rfc1413

이보다 더 재미있는 일이 있을 수 있겠습니까? 사실 몇 가지 생각나긴 합니다만 소켓 프로그래밍에 대한 것이 아니군요. 아무튼 정리하자면 이렇습니다:

```
#include <unistd.h>

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

인수들은 단순합니다: hostname은 함수가 반환하는 호스트 이름을 담을 char의 배열에 대한 포인터입니다. size는 hostname배열의 길이입니다.

함수는 성공적인 완료 후에 0을 반환하고, 오류에 대해서는 흔히 그렇듯 errno를 설정하고 -1을 반환합니다.

Chapter 6

클라인트-서버 배경지식

클라이언트-서버에 대해 이야기할 차례이다. 통신망에 있는 거의 모든 것들은 서버 프로세스에게 이야기하는 클라이언트 프로세스를 상대하거나 그 반대이다. telnet을 예로 들어보자. 여러분이 텔넷(클라이언트)로 원격지 호스트의 23번 포트에 접속할 때 그 호스트의 프로그램(telnetd라고 불리는 서버)이 생명을 얻는다. 그것이 들어오는 텔넷 요청을 처리하고 당신에게 로그인 프롬프트를 띄워주는 등의 일을 처리한다.

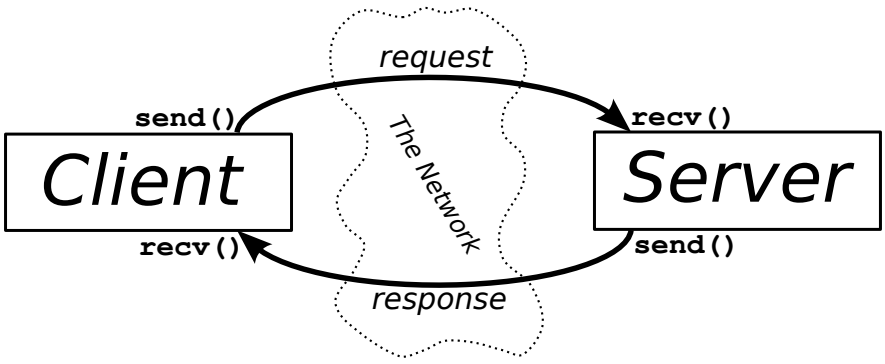

Figure 6.1: 클라이언트 - 서버 상호작용

위의 도표에 클라이언트와 서버의 정보 교환이 정리되어 있습니다.

클라이언트-서버 쌍은 SOCK_STREAM이나 SOCK_DGRAM 또는 다른 어떤 것이라도 말할 수 있음을 기억하십시오.(둘이 같은 방식으로 말하기만 한다면) 클라이언트-서버 쌍의 좋은 예시는 telnet/telnetd, ftp/ftpd 또는 Firefox/Apache 입니다. 당신이 ftp를 쓸 때마다 당신의 요청을 받아들이는 원격지 프로그램인 ftpd가 있습니다.

흔히 한 대의 장치에는 오직 하나의 서버만이 있을 것이며 그 서버는 fork() 를 통해서 여러 클라이언트를 처리할 것입니다.(역자 주 : 한 대의 장치에서 여러 개의 서버를 실행하는 많은 방법이 있지만 이 문서의 초판은 90년대에 작성되었습니다.) 기본적인 과정은 아래와 같습니다. 서버가 연결을 기다리고, accept()한 후, 요청을 처리할 자식 프로세스를 fork()합니다. 이것이 다음 절에서 우리의 예제 서버가 하는 일입니다.

6.1 단순한 스트림 서버

이 서버가 하는 일은 스트림 연결에 “Hello, world!”을 전송하는 것 뿐입니다. 이 서버를 시험하기 위해서 할 일은 하나의 창에서 이것을 실행한 후 다른 창에서 텔넷에 아래 명령어로 접속하는 일 뿐입니다.

$ telnet remotehostname 3490

remotehostname은 당신이 실행하는 장치의 아이피입니다.

The server code[1]:

```
/*
** server.c -- a stream socket server demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "3490"  // 사용자들이 접속할 포트

#define BACKLOG 10   // 몇 개의 대기중인 연결이 유지될 것인가

void sigchld_handler(int s)
{
    // waitpid()이 errno를 덮어쓸 수 있으므로 저장했다가 되살린다.
    int saved_errno = errno;

    while(waitpid(-1, NULL, WNOHANG) > 0);

    errno = saved_errno;
}


// IPv4 또는 IPv6 sockaddr을 받아온다.
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd, new_fd;  // sock_fd에서 대기하고 들어오는 연결은 new_fd에 저장
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // 접속자의 주소 정보
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
```

[1] https://beej.us/guide/bgnet/examples/server.c

```
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use my IP

if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
}

// 모든 결과를 조회하고 쓸 수 있는 첫 번째 것을 사용
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
            sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        continue;
    }

    break;
}

freeaddrinfo(servinfo); // 이 구조체는 더 이상 필요없음

if (p == NULL)  {
    fprintf(stderr, "server: failed to bind\n");
    exit(1);
}

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

sa.sa_handler = sigchld_handler; // 죽은 프로세스를 다 거둬들이자
sigemptyset(&sa.sa_mask);
sa.sa_flags =   ;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}
```

```
    printf("server: waiting for connections...\n");

    while(1) {  // 주 accept() 루프
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if (new_fd == -1) {
            perror("accept");
            continue;
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s);
        printf("server: got connection from %s\n", s);

        if (!fork()) { // 자식 프로세스이다.
            close(sockfd); // 자식은 리스너가 필요없다.
            if (send(new_fd, "Hello, world!", 13, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // 부모는 이것이 필요없다.
    }

    return 0;
}
```

궁금한 독자들을 위해 덧붙이자면 구문의 명료함을 위해서 하나의 큰 main()함수 안에 모든 코드를 다 적었습니다. 원한다면 더 작은 함수들로 나누어도 좋습니다.

(아마도 이 sigaction()을 처음 볼 수도 있는데 괜찮다. 이 코드는 fork()된 자식 프로세스가 종료되면서 생기는 좀비 프로세스를 거둬들이는 데 사용된다. 좀비 프로세스를 많이 만들고 거둬들이지 않으면 시스템 관리자가 흥분할 것이다.)

다음 절에 나오는 클라이언트를 사용해서 이 서버로부터 데이터를 얻을 수 있다.

6.2 단순한 스트림 클라이언트

이 녀석은 서버보다도 더 쉽다. 이 클라이언트가 하는 일은 당신이 명령줄에 지정한 호스트의 3490번 포트로 접속하는 것이다. 이것은 서버가 보낸 문자열을 받는다.

The client source[2]:

```
/*
** client.c -- a stream socket client demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
```

[2] https://beej.us/guide/bgnet/examples/client.c

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

#include <arpa/inet.h>

#define PORT "3490" // 클라이언트가 접속할 포트

#define MAXDATASIZE 100 // 한 번에 받을 수 있는 최대 바이트 갯수

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct addrinfo hints, *servinfo, *p;
    int rv;
    char s[INET6_ADDRSTRLEN];

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

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // 모든 결과를 순회하면서 쓸 수 있는 가장 첫 번째 것을 씀
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("client: connect");
            continue;
```

```
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
    printf("client: connecting to %s\n", s);

    freeaddrinfo(servinfo); // 이 구조체는 더 이상 필요 없음

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

    buf[numbytes] = '\0';

    printf("client: received '%s'\n",buf);

    close(sockfd);

    return 0;
}
```

클라이언트를 실행하기 전에 서버를 실행하지 않으면 connect()는 "Connection refused"를 반환한다는 점을 기억하라. 아주 유용하다.

6.3 데이터그램 소켓

위에서 sendto()과 recvfrom()에 대해 논의할 때 UDP 데이터그램 소켓의 기본에 대해서 이미 알아보았다. 그러므로 바로 2개의 예제 프로그램을 제시하겠다. talker.c와 listener.c이다.

listener는 장치에서 포트 4950으로 들어오는 패킷을 대기한다. talker는 지정한 장치의 해당 포트로 사용자가 명령줄에 입력한 내용을 담은 패킷을 보낸다.

데이터그램 소켓은 연결이 없고 소켓을 이더넷에 발송한 후 성공 여부는 신경쓰지 않기 때문에 클라이언트와 서버에 IPv6을 사용하도록 명시할 것이다. 이렇게 하면 서버가 IPv6에서 듣고 클라이언트가 IPv4에서 발송해서 데이터를 받을 수 없는 상황을 피할 수 있을 것이다. (우리의 TCP 스트림 소켓 세상에서도 이런 불일치가 발생할 수 있지만 connect()에서 하나의 주소 체계에 대해 에러를 발생시키고 다른 주소체계를 쓰도록 해준다.)

여기에 listener.c의 소스코드가 있다.[3]:

```
/*
** listener.c -- a datagram sockets "server" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

[3] https://beej.us/guide/bgnet/examples/listener.c

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

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

#define MAXBUFLEN 100

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
    struct sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET6; // set to AF_INET to use IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("listener: socket");
            continue;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("listener: bind");
```

```
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "listener: failed to bind socket\n");
        return 2;
    }

    freeaddrinfo(servinfo);

    printf("listener: waiting to recvfrom...\n");

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

    printf("listener: got packet from %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("listener: packet is %d bytes long\n", numbytes);
    buf[numbytes] = '\0';
    printf("listener: packet contains \"%s\"\n", buf);

    close(sockfd);

    return 0;
}
```

getaddrinfo()에서 우리가 마침내 SOCK_DGRAM을 사용한다는 것에 주목하라. 또한, listen()와 accept()이 필요하지 않다는 점도 기억하라. 이것이 연결 없는 데이터그램 소켓을 사용할 때의 장점 중 하나이다. Notice that in our call to getaddrinfo() we're finally using SOCK_DGRAM. Also, note that there's no need to listen() or accept(). This is one of the perks of using unconnected datagram sockets!

Next comes the source for talker.c[4]:

```
/*
** talker.c -- a datagram "client" demo
*/

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

[4] https://beej.us/guide/bgnet/examples/talker.c

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

#define SERVERPORT "4950"    // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;

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

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET6; // set to AF_INET to use IPv4
    hints.ai_socktype = SOCK_DGRAM;

    if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and make a socket
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("talker: socket");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "talker: failed to create socket\n");
        return 2;
    }

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
            p->ai_addr, p->ai_addrlen)) == -1) {
        perror("talker: sendto");
        exit(1);
    }

    freeaddrinfo(servinfo);

    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
    close(sockfd);

    return 0;
```

}

이것이 전부다! 하나의 장치에서 listener를 실행하고 talker를 다른 장치에서 실행하라.(역자 주 : 하나의 장치에서도 순서만 맞게 실행하면 문제는 없다. 여러 터미널을 동시에 열 수 있는 다양한 방법이 있다.) 그것들이 통신하는 것을 지켜보라. 핵가족 전체를 위한 전연령 엔터테인먼트다.

이번에는 서버르 실행할 필요도 없다! talker를 혼자 실행시키면 패킷을 행복하게 발송하고, 아무도 반대쪽에서 recvfrom()을 호출하지 않는다면 그저 패킷은 사라질 뿐이다. 기억하라 : UDP 데이터그램 소켓으로 보낸 데이터는 도착을 보장하지 않는다.!

전에 몇 번 말한 사소한 것 한 가지를 빼면 전부다: 연결된 데이터그램 소켓이 그것이다. 그것에 대해서 여기에서 말해야하는데, 이 문서의 데이터그램에 대한 부분이 바로 여기이기 때문이다. 위의 talker 가 listener의 주소를 지정하고 connect()를 호출한다고 하자. 그 순간부터 talker는 connect()로 지정한 주소로만 데이터를 보내고 받을 수 있다. 이런 이유로 sendto()와 recvfrom()을 쓸 필요가 없다. 단순히 send() 와 recv()를 쓰면 된다.

Chapter 7

약간 더 고급스러운 기술

이것들은 진짜로 고급 기술인 것은 아니지만 우리가 지금까지 다룬 기본적인 것들을 벗어나고 있다. 사실 당신이 여기까지 왔다면 스스로가 기본적인 유닉스 네트워크 프로그래밍을 꽤 잘 한다고 생각해도 된다. 축하한다.

이제 당신이 소켓에 대해서 배우고 싶어할 좀 더 비밀스러운 것들의 세상으로 용감하게 가보자. 열심히 해 보자!

7.1 블로킹

블로킹. 그것에 대해서 이미 들어보았다. 그것은 도대체 무엇인가? 간단히 말하면 "블록"은 "잠자기"에 대한 기술적 용어이다. 위에서 listener를 실행하면 패킷이 도착할 때까지 그대로 기다리고 있다는 것을 눈치챘을 것이다. 내부적으로는 recvfrom()이 호출되고, 데이터가 없으므로 recvfrom()는 데이터가 도착할 때까지 "블록"된 상태인 것이다.(즉 그대로 잠들어 있게 된다.)

많은 함수들이 블록상태가 된다. accept()는 블로킹 함수이다. 모든 recv() 함수들도 블록 동작을 하는 함수이다.(역자 주 : 원문에서는 block자체를 동사로 쓴다.) 그것들이 블록동작을 할 수 있는 이유는 그렇게 할 수 있게 허락을 받았기 때문이다. socket()으로 소켓 설명자를 처음 만들 때 커널이 이 소켓을 블로킹 소켓으로 설정한다. 만약 소켓이 블록 동작을 하지 않길 원한다면 fcntl() 에 대한 호출을 해야한다. :

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

소켓을 논블로킹으로 설정하면 정보를 얻기 위해서 소켓을 효과적으로 "조사(원문 : poll)" 할 수 있다. 논 블로킹 소켓을 읽으려고 할 때 정보가 없다면 그것이 블록 동작을 하는 것은 허락되지 않는다. 그것은 -1을 반환할 것이고 errno은 EAGAIN 이나 EWOULDBLOCK로 설정될 것이다.

(잠깐, EAGAIN 이나 EWOULDBLOCK 를 돌려준다니 무엇을 확인해야 한다는 말인가? 명세서에는 사실 당신의 시스템이 어떤 값을 돌려줘야 하는지가 정의되어 있지 않다. 그러므로 이식성을 위해서는 둘을 모두 확인해야 한다.)

그러나 일반적으로 말하자면 이런 방식의 조사는 좋은 생각이 아니다. 당신의 프로그램이 소켓의 자료를 기다리면서 바쁜 대기 상태가 되면 당신은 보통의 프로그램보다 훨씬 CPU시간을 많이 사용할 것이다. (역자 주 : 특별한 제한을 걸지 않으면 최대 단일 코어 하나를 100% 점유할 수 있다.) 읽기 작업을 기다리는 정보가 있는지 확인하기 위한 조금 더 우아한 해결책은 poll()에 대해 다루는 다음 절에 있다.

7.2 poll()—동기 입출력 다중화

당신이 정말로 하고자 해야하는 일은 소켓 한 무더기 를 한 번에 감시하고 그 중에 데이터가 준비된 것을 처리하는 것이다. 이런 방식을 통해서 당신은 모든 소켓을 지속적으로 조사하지 않아도 여러 개의 소켓 중 어떤 것이 데이터가 준비되었는지 알 수 있다.

> 경고 : poll()은 엄청나게 많은 수의 연결을 처리할 때 끔찍하게 느려진다. 이런 상황에서는 libevent[1] 같은 이벤트 라이브러리 를 사용하면 더 좋은 성능을 얻을 수 있다. 이런 라이브러리는 당신의 운영체제에서 사용할 수 있는 가장 빠른 방법을 사용하려고 시도할 것이다.

그래서 어떻게 조사를 피할 수 있는가? 놀랍게도 poll() 시스템 함수를 사용해서 조사를 피할 수 있다. (역자 주 : poll은 투표, 설문, 그러한 부류의 조사라는 뜻이다.) 간단히 말하자면 운영체제에게 모든 번거로운 작업을 우리 대신 하고 어떤 소켓에 자료가 도착하면 알려달라고 부탁하는 것이다. 그 동안 우리의 프로세스는 대기 상태가 될 수 있고 시스템 자원을 아낄 수 있다.

전체적인 계획은 우리가 감시하고 싶은 소켓 설명자와 우리가 감시하고 싶은 이벤트의 종류에 대한 정보를 담은 struct pollfd의 배열을 보관하는 것이다. 운영체제는 해당하는 종류의 이벤트가 발생(예를 들어 "소켓에 읽을 자료가 있다!" 같은 이벤트)하거나 사용자가 지정한 제한 시간이 지날 때까지 poll() 호출을 블록할 것이다.

유용하게도 listen() 상태인 소켓은 새로운 연결이 accept()될 수 있는 상태가 되었을 때 "ready to read"를 반환할 것이다.

이만하면 충분히 떠들었다. 이것을 쓰는 방법은 어떨지 보자.

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

fds는 우리의 정보(어떤 소켓을 무엇을 위해 감시할지)의 배열이다. nfds는 배열에 담긴 요소의 갯수이다. timeout은 밀리초 단위의 제한시간이다. 이것은 이벤트가 발생한 요소의 갯수를 돌려준다.

위에 등장하는 구조체는 무엇인지 살펴보자:

```
struct pollfd {
  int fd;       // 소켓 설명자
  short events;  // 우리가 관심있는 이벤트의 비트맵
  short revents;  // poll()이 반환하는 시점에 발생한 이벤트의 비트맵
};
```

즉 이것의 배열을 하나 설정하고 각각의 fd필드를 우리가 관찰하고 싶은 소켓 설명자로 설정한다. 그리고 각각의 events필드는 우리가 관심있는 이벤트로 설정하는 것이다.

events필드는 아래 값들을 비트단위 논리합 계산한 결과값이다.

매크로	설명
POLLIN	이 소켓이 데이터를 recv()할 준비가 되면 알려달라.
POLLOUT	이 소켓에 블로킹 없이 데이터를 send()할 수 있으면 알려달라.

struct pollfd의 배열을 준비하면 poll()에 그것을 넘길 수 있다. 배열의 크기와 밀리초 단위의 제한시간도 같이 넘겨야 한다.(영원히 기다리려면 음수를 지정하면 된다.)

poll()이 반환하면 이벤트가 발생했음을 나타내는 POLLIN이나 POLLOUT이 설정되었는지 보기위해서 revents 필드를 확인할 수 있다.

(실제로는 poll()호출로 할 수 있는 것들이 더 있다. 자세한 내용은 아래의 poll() 맨페이지를 참고하라.)

여기 표준 입력에서 데이터를 읽어들일 수 있을 때까지(예를 들어 당신이 줄바꿈을 입력할 때까지) 2.5초를 기다리는 예제가 있다.an example[2]

```
#include <stdio.h>
#include <poll.h>
```

[1] https://libevent.org/
[2] https://beej.us/guide/bgnet/examples/poll.c

```
int main(void)
{
    struct pollfd pfds[1]; // 더 많은 것들을 관찰하고 싶다면 더 크게 하라.

    pfds[0].fd = 0;          // 표준 입력
    pfds[0].events = POLLIN; // 읽을 준비가 되면 알려달라.

    // 만약 다른 것들도 관찰하고 싶다면
    //pfds[1].fd = some_socket; // 임의의 소켓 설명자
    //pfds[1].events = POLLIN;  // 읽을 준비가 되면 알려달라.

    printf("엔터키를 누르거나 제한시간 도달을 위해 2.5초를 기다리라\n");

    int num_events = poll(pfds, 1, 2500); // 2.5초 제한 시간

    if (num_events == 0) {
        printf("Poll 시간 초과!\n");
    } else {
        int pollin_happened = pfds[0].revents & POLLIN;

        if (pollin_happened) {
            printf("파일 설명자 %d을 읽을 준비가 되었다\n", pfds[0].fd);
        } else {
            printf("예상하지 못한 이벤트가 발생했다: %d\n", pfds[0].revents);
        }
    }

    return 0;
}
```

poll()이 pfds배열에서 이벤트가 발생한 요소의 갯수를 돌려준다는 것을 다시 기억하라. 배열의 어떤 요소에서 이벤트가 발생했는지는 알려주지 않지만 몇 개의 revents 필드가 0이 아닌 값으로 설정되었는지는 알려준다. 이것을 활용해서 반환된 숫자만큼의 0이 아닌 revents를 읽은 후에는 스캔을 중단할 수 있다.

몇 가지 질문이 떠오를 것이다: poll()에 넘겨준 집합에 새 파일 설명자를 추가하려면 어떻게 해야하는가? 이를 위해서 단순히 당신의 모든 필요에 부합할 만큼 충분한 크기의 배열을 만들거나 추가적인 공간이 필요할 때 realloc()을 사용하라.

집합에서 요소를 제거하려면 어떻게 해야하는가? 이것을 위해서 당신은 배열의 마지막 요소를 삭제할 요소에 덮어씌우고, poll()의 count에 하나 더 적은 값을 전달하라. (역자 주 : 이것은 배열에서 임의의 요소 1개를 빠르게 제거하기 위해서 일반적으로 사용하는 방법이다.) 다른 한 가지 방법은 fd필드를 음수로 설정하는 것이며 poll()은 해당 요소를 무시할 것이다.

이 모든 것을 당신이 telnet할 수 있는 하나의 채팅 서버에 합치려면 어떻게 해야할까?

우리가 할 일은 리스너 소켓을 시작한 후에 그것을 poll()할 파일 설명자 집합에 추가하는 일이다. (그 파일설명자는 들어오는 연결이 있을 때 읽기 준비된 상태가 될 것이다.)

그 후에 새로운 연결을 우리의 struct pollfd 배열에 추가하면 된다. 만약 배열의 크기가 부족하다면 동적으로 키우면 된다.

연결이 닫힌 후에는 그것을 배열로부터 제거한다.

연결이 읽기 준비되면 우리는 그것에서 데이터를 읽어들인 후 다른 모든 연결에 전송한다. 그렇게 해서 사용자들은 서로가 입력한 내용을 볼 수 있다.

이제 이 폴 서버[3]를 한 번 시험해보라. 이것을 하나의 창에서 실행한 후 몇 개의 다른 터미널 창에서 telnet localhost 9034를 실행해보라. 당신이 하나의 창에서 입력하는 것을 (당신이 엔터키를 누른 후에) 다른 창들에서 볼 수 있어야한다.

[3] https://beej.us/guide/bgnet/examples/pollserver.c

그것 뿐 아니라 당신이 CTRL-]를 누른 후 quit을 입력해서 telnet을 종료할 경우 서버는 연결종료를 감지하고 그 연결을 파일 설명자 배열에서 제거할 것이다.

```
/*
** pollserver.c -- 형편없는 다인 대화 서버
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <poll.h>

#define PORT "9034"   // 우리가 듣는(listening) 포트

// Get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

// Return a listening socket
int get_listener_socket(void)
{
    int listener;     // 듣는 소켓 설명자
    int yes=1;        // setsockopt() SO_REUSEADDR을 위해서는 아래를 보라
    int rv;

    struct addrinfo hints, *ai, *p;

    // 소켓을 받아서 바인드하자
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (listener < 0) {
            continue;
        }
```

```
        // 귀찮은 "주소가 이미 사용중입니다"에러메시지를 제거한다
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }

    freeaddrinfo(ai); // 더 이상 필요없다.

    // 여기가 실행되면 우리가 바인드하지 못했다는 의미다
    if (p == NULL) {
        return -1;
    }

    // 리슨
    if (listen(listener, 10) == -1) {
        return -1;
    }

    return listener;
}

// 집합에 새 파일 설명자를 추가한다
void add_to_pfds(struct pollfd *pfds[], int newfd, int *fd_count, int *fd_size)
{
    // 공간이 부족하면 pfds 배열을 늘린다.
    if (*fd_count == *fd_size) {
        *fd_size *= 2; // 두배로 한다.

        *pfds = realloc(*pfds, sizeof(**pfds) * (*fd_size));
    }

    (*pfds)[*fd_count].fd = newfd;
    (*pfds)[*fd_count].events = POLLIN; // 읽을 준비가 되었는지 확인

    (*fd_count)++;
}

// 집합에서 하나의 인덱스를 제거한다
void del_from_pfds(struct pollfd pfds[], int i, int *fd_count)
{
    // 마지막에서 하나를 삭제 대상 인덱스로 복사해온다
    pfds[i] = pfds[*fd_count-1];

    (*fd_count)--;
}

// 메인
int main(void)
```

```
{
    int listener;     // 리슨 소켓 설명자

    int newfd;        // 새롭게 accept()한 소켓 설명자
    struct sockaddr_storage remoteaddr; // 클라이언트 주소
    socklen_t addrlen;

    char buf[256];    // 클라이언트 데이터를 위한 버퍼

    char remoteIP[INET6_ADDRSTRLEN];

    // 5개의 연결을 위한 공간을 가지고 시작한다
    // (필요해지면 realloc할 것이다.)
    int fd_count = 0;
    int fd_size = 5;
    struct pollfd *pfds = malloc(sizeof *pfds * fd_size);

    // 초기화 후 리스닝 소켓을 얻는다
    listener = get_listener_socket();

    if (listener == -1) {
        fprintf(stderr, "리스닝 소켓 얻기 실패\n");
        exit(1);
    }

    // 리스너를 집합에 추가
    pfds[0].fd = listener;
    pfds[0].events = POLLIN; // 들어오는 연결을 읽을 준비가 되면 보고하라

    fd_count = 1; // 리스너를 위한 설정

    // 주 반복문
    for(;;) {
        int poll_count = poll(pfds, fd_count, -1);

        if (poll_count == -1) {
            perror("poll");
            exit(1);
        }

        // 읽어들일 데이터를 찾기 위해서 존재하는 연결을 순회
        for(int i = 0; i < fd_count; i++) {

            // 무엇인가 읽을 준비가 되었는지 확인
            if (pfds[i].revents & POLLIN) { // 하나를 찾았다!!

                if (pfds[i].fd == listener) {
                    // 리스너를 읽을 준비가 되었다면 새 연결을 처리한다

                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                        (struct sockaddr *)&remoteaddr,
                        &addrlen);
```

```
            if (newfd == -1) {
                perror("accept");
            } else {
                add_to_pfds(&pfds, newfd, &fd_count, &fd_size);

                printf("폴서버: 새로운 연결 %s"
                    " 소켓 %d\n",
                    inet_ntop(remoteaddr.ss_family,
                        get_in_addr((struct sockaddr*)&remoteaddr),
                        remoteIP, INET6_ADDRSTRLEN),
                    newfd);
            }
        } else {
            // 리스너가 아닐 경우 일반적인 클라이언트다
            int nbytes = recv(pfds[i].fd, buf, sizeof buf, 0);

            int sender_fd = pfds[i].fd;

            if (nbytes <= 0) {
                // 오류가 발생했거나 연결이 클라이언트에 의해 닫혔다
                if (nbytes == 0) {
                    // 연결이 닫혔다.
                    printf("폴서버: 소켓 %d 이 끊어짐\n", sender_fd);
                } else {
                    perror("recv");
                }

                close(pfds[i].fd); // 잘가!

                del_from_pfds(pfds, i, &fd_count);

            } else {
                // 클라이언트로부터 뭔가 좋은 데이터를 받았다

                for(int j = 0; j < fd_count; j++) {
                    // 모두에게 보내자!
                    int dest_fd = pfds[j].fd;

                    // 리스너와 보낸 사람을 제외한다
                    if (dest_fd != listener && dest_fd != sender_fd) {
                        if (send(dest_fd, buf, nbytes, 0) == -1) {
                            perror("send");
                        }
                    }
                }
            }
        } // 클라이언트로부터 온 데이터를 처리하는 부분의 끝
    } // poll()에서 읽을 준비가 된 부분의 끝
  } // 파일 설명자 순회의 끝
} // for(;;)의 끝--절대 안 끝나겠지만!

  return 0;
}
```

다음 절에서는 비슷하지만 오래된 함수인 select()를 살펴볼 것이다. select()와 poll() 모두 비슷한 기능과 성능을 제공하고 쓰는 방식만 조금 다르다. select()쪽이 조금 더 이식성이 좋을지도 모르나 사용하기에는 조금 더 어색할 것이다. 당신의 시스템에서 지원되기만 한다면 더 마음에 드는 쪽을 선택하라.

7.3 select()—동기화된 I/O 멀티플렉싱, 예전 방식

이 함수는 이상하지만 아주 유용하다. 다음과 같은 상황을 생각해보라: 당신은 서버이고 들어오는 연결을 감지함과 동시에 이미 가진 연결로부터 계속 자료를 읽어들이고 싶다.

별 문제가 없다고 말할지도 모른다. 그냥 accept()와 몇 개의 recv()를 쓰면 될 뿐이다. 정말로 그럴까? accept()호출이 블록 상태로 들어갔다면 어떻게 하겠는가? 어떻게 recv()로 동시에 데이터를 받을 수 있겠는가? "논 블로킹 소켓을 써라!" 역시 안 될 말이다. CPU를 모조리 쓰고 싶지는 않을 것이다. 그럼 어떻게 해야하는가?

select()가 여러 소켓을 동시에 관찰할 수 있는 능력을 준다. 그것이 어떤 것이 읽을 준비가 되었는지, 어떤 것이 쓸 준비가 되었는지, 그리고 정말로 관심이 있다면 어떤 것에 오류가 발생했는지까지 알려줄 것이다.

> 경고 한마디: select()가 이식성이 아주 좋지만 연결이 아주 많은 상황에서는 끔찍하게 느려진다. 그런 상황에서는 당신의 시스템에서 쓸 수 있는 가장 빠른 방법을 시도하는 libevent[4] 같은 이벤트 라이브러리를 쓰면 더 나은 성능을 얻을 수 있다.

잡담은 그만하고 select()의 개요를 제시하겠다.

```
#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 특정한 readfds과 writefds 그리고 exceptfds로 이루어진 파일 설명자의 "집합들"을 관찰한다. 만약 당신이 표준 입력과 몇 개의 소켓 설명자로부터 읽어들일 수 있는지 확인하고 싶다면 readfds 집합에 0과 sockfd를 추가하라. numfds는 가장 큰 파일 설명자에 1을 더한 값으로 설정해야 한다. 이 예제에서는 sockfd+1이 될 것이며 이유는 당연히 그것이 표준 입력(0)보다 클 것이기 때문이다.

select()가 반환할 때 readfds는 당신이 선택한 파일 설명자 중에서 읽기를 위해 준비된 것들을 반영하기 위해서 변해있을 것이다. 당신은 그것들을 아래에 나오는 FD_ISSET()매크로로 검사할 수 있다. When select() returns, readfds will be modified to reflect which of the file descriptors you selected which is ready for reading. You can test them with the macro FD_ISSET(), below.

더 진행하기 전에 이 집합들을 어떻게 조작하는지에 대해 이야기할 것이다. 각 집합은 fd_set형이다. 이 자료형에 대해서 아래의 매크로들을 쓸 수 있다.

함수	설명
FD_SET(int fd, fd_set *set);	fd를 set에 더한다.
FD_CLR(int fd, fd_set *set);	fd를 set에서 제거한다.
FD_ISSET(int fd, fd_set *set);	fd이 set에 있다면 참을 돌려준다.
FD_ZERO(fd_set *set);	set의 모든 요소를 제거한다.

마지막으로 이 이상한 struct timeval은 무엇일까? 누군가 당신에게 자료를 보낼 때까지 무한히 기다리고 싶지 않을 때가 있다. 아마도 매 96초마다 실제로는 아무 일도 일어나지 않았어도 "진행중..."이라고 출력하고 싶을 수도 있다. 이 시간 구조체가 제한시간을 지정할 수 있도록 해 준다. 시간이 초과하고 select() 가 준비된 파일 설명자를 찾지 못할 경우, 그것은 반환하고 당신은 처리를 계속할 수 있다.

struct timeval는 아래와 같은 필드를 가지고 있다:

```
struct timeval {
   int tv_sec;    // 초
   int tv_usec;   // 마이크로초
};
```

[4] https://libevent.org/

단순히 tv_sec을 기다리고 싶은 초로, tv_usec을 기다리고 싶은 마이크로초로 설정하라. 그렇다. 마이크로 초다. 밀리초가 아니다. 1밀리초는 1,000마이크로초다. 그리고 1,000밀리초는 1초이다. 그러므로 1초는 1,000,000초이다. 왜 "usec"일까? "u"는 우리가 "마이크로"를 뜻하기 위해서 쓰는 그리스 문자 μ(뮤)와 닮았기 때문이다. 또 함수가 반환할 때 timeout은 남아있는 시간을 보여주기 위해서 업데이트 될 수도 있다. 이것은 당신이 실행중인 유닉스의 종류에 따라 다르다.

와! 우리는 마이크로초 해상도의 타이머를 가졌다! 사실 별로 기대하지 않는 것이 좋다. 당신이 struct timeval을 아무리 작게 설정해도 당신의 표준 유닉스 타임슬라이스 (역자 주 : 커널이 프로세스 스케쥴링의 최소 단위로 쓰는 시간)만큼은 기다려야 한다.

다른 흥미로운 것들: struct timeval을 0으로 설정하면 select()는 당신의 집합에 있는 모든 파일 설명자를 조사한 즉시 시간초과가 될 것이다. 매개변수 timeout을 NULL 로 설정하면 절대 시간초과가 되지 않으며 파일 설명자가 준비될 때까지 기다릴 것이다. 마지막으로 만약 특정 집합을 기다릴 필요가 없다면 그 집합은 select()를 호출할 때 NULL로 설정하면 된다.

아래의 코드 조각[5]은 표준 입력에 뭔가 나타날 때까지 2.5초를 기다린다:

```
/*
** select.c -- a select() demo
*/

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

#define STDIN 0  // 표준 입력의 파일 설명자

int main(void)
{
    struct timeval tv;
    fd_set readfds;

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

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

    // writefds와 exceptfds는 신경쓰지 않는다:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("키가 눌렸다!\n");
    else
        printf("시간이 초과되었다.\n");

    return 0;
}
```

만약 당신이 줄 단위로 버퍼처리되는 터미널을 사용한다면 엔터를 누르지 않으면 제한시간이 초과될 것이다.

이제 여러분 중 일부는 이것이 데이터그램 소켓의 데이터를 기다리는 아주 훌륭한 방법이라고 생각할 것이다. 그리고 맞다. 맞을 수도 있다. 일부 유닉스에서는 select를 이 목적으로 쓸 수 있고, 일부에서는 그럴 수 없다. 그 방식을 시도하고 싶다면 당신의 로컬 맨페이지 내용을 참고해야 한다.

일부 유닉스는 제한시간이 초과되기까지 남은 시간을 반영하기 위해서 당신의 struct timeval를 업데이트한다. 그러나 다른 것들은 그렇게 하지 않는다. 만약 이식성있는 코드를 작성하고자 한다면 그것에 의존해서는 안된다. (경과한 시간을 알고싶다면gettimeofday() 을 사용하라. 실망스럽겠지만 그것이 올바른 방법이다.)

[5] https://beej.us/guide/bgnet/examples/select.c

만약 읽기 집합에 잇는 소켓이 연결을 닫는다면 어떤 일이 생길까? 그 경우 select()는 그 소켓 설명자를 "읽기 준비된 상태"로 설정할 것이다. 실제로 그 소켓에 recv()하면 recv()는 0을 돌려줄 것이다. 그것이 클라이언트가 연결을 닫았음을 알아내는 방법이다.

select()에 관한 흥미로운 이야기 하나 더: 만약 listen()작업중인 소켓을 가지고 있을 경우, 그 소켓의 파일 설명자를 readfds 집합에 넣어서 새로운 연결이 있는지 알 수 있다.

지금까지 전능한 select()함수에 대한 간략한 개관이었다.

그러나 대중적 요구가 있으므로 아래에 심도있는 예제를 첨부한다. 불행하게도 위의 아주 단순한 예제와 아래의 에제에는 상당한 차이가 있다. 그렇지만 한 번 살펴보고 뒤따르는 설명을 읽어보라.

이 프로그램[6] 은 단순한 다중 사용자 챗 서버처럼 동작한다. 하나의 창에서 이것을 실행한 후 다른 창에서 telnet을 통해 접속하라. ("telnet hostname 9034") 하나의 telnet세션에서 뭔가 입력하면 나머지 모두에서 그 내용이 나타나야 한다.

```
/*
** selectserver.c -- 허술한 다중 사용자 대화 서버
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PORT "9034"   // 우리가 듣는 포트

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    fd_set master;    // 마스터 파일 설명자 리스트
    fd_set read_fds;  // select()를 위한 임시 파일 설명자 리스트
    int fdmax;        // 가장 큰 파일 설명자 번호

    int listener;     // 듣는 소켓 설명자
    int newfd;        // 새롭게 accept() 처리한 소켓 설명자
    struct sockaddr_storage remoteaddr; // 클라이언트 주소
    socklen_t addrlen;

    char buf[256];    // 클라이언트 데이터를 위한 버퍼
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];
```

[6] https://beej.us/guide/bgnet/examples/selectserver.c

```
int yes=1;        // setsockopt() SO_REUSEADDR를 위해서는 아래를 보라
int i, j, rv;

struct addrinfo hints, *ai, *p;

FD_ZERO(&master);    // 마스터와 임시 집합을 초기화
FD_ZERO(&read_fds);

// 소켓을 하나 받아와서 바인드한다.
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
    fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
    exit(1);
}

for(p = ai; p != NULL; p = p->ai_next) {
    listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (listener < 0) {
        continue;
    }

    // 짜증나는 "address already in use" 오류 메시지를 제거한다.
    setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

    if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
        close(listener);
        continue;
    }

    break;
}

// 이곳이 실행되면 바인드가 되지 않은 것이다.
if (p == NULL) {
    fprintf(stderr, "selectserver: failed to bind\n");
    exit(2);
}

freeaddrinfo(ai); // 더 이상 필요없다.

// 듣는다.
if (listen(listener, 10) == -1) {
    perror("listen");
    exit(3);
}

// 리스너를 마스터 집합에 추가한다.
FD_SET(listener, &master);

// 가장 큰 파일 설명자를 기록한다.
```

```
    fdmax = listener; // 현재까지는 이것이다.

    // 주 반복문
    for(;;) {
        read_fds = master; // 복사한다.
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
        }

        // 존재하는 연결을 순회하며 읽을 데이터가 있는지 확인한다.
        for(i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == listener) {
                    // 새 연결을 처리한다.
                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                        (struct sockaddr *)&remoteaddr,
                        &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                        FD_SET(newfd, &master); // 마스터 집합에 추가한다.
                        if (newfd > fdmax) {    // 가장 큰 것을 기록한다.
                            fdmax = newfd;
                        }
                        printf("selectserver: new connection from %s on "
                            "socket %d\n",
                            inet_ntop(remoteaddr.ss_family,
                                get_in_addr((struct sockaddr*)&remoteaddr),
                                remoteIP, INET6_ADDRSTRLEN),
                            newfd);
                    }
                } else {
                    // 클라이언트에서 온 자료를 처리한다.
                    if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                        // 오류가 발생했거나 클라이언트에 의해 연결이 종료되었다.
                        if (nbytes == 0) {
                            // 연결이 종료되었다.
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i); // 잘가!
                        FD_CLR(i, &master); // 마스터 집합에서 삭제
                    } else {
                        // 클라이언트로부터 데이터가 들어왔다.
                        for(j = 0; j <= fdmax; j++) {
                            // 모두에게 보낸다!
                            if (FD_ISSET(j, &master)) {
                                // 리스너와 그 자신을 제외
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) == -1) {
```

```
                        perror("send");
                    }
                }
            }
        }
    }
} // 무한루프의 끝. 절대 끝나지 않는다고 생각할 것이다!

return 0;
}
```

코드에 master와 read_fds 두 개의 파일 설명자 집합이 있음에 주목하라. 전자인 master는 새 연결을 듣는 소켓 설명자와 현재 연결된 모든 소켓의 설명자를 가진다.

master를 가지는 이유는 select()가 사실 당신이 넘기는 집합을 변형해서 읽기 준비된 소켓을 반영하기 때문이다. 우리는 하나의 select()호출과 다음 호출 사이에서 연결들을 계속 기억해야 하므로 이것들은 다른 곳에 안전하게 보관해야 하는 것이다. 그래서 우리는 실제로 쓰기 전에 master를 read_fds에 복사하고 select()를 호출하는 것이다.

하지만 그것은 우리가 새로운 연결을 받을 때마다 그것을 master집합에 추가해야 한다는 의미가 아닌가? 맞다! 그리고 연결이 닫힐 때마다 master집합에서 제거해야 하지않은가? 맞다, 그렇게 해야한다.

listener소켓이 읽을 준비가 되었는지 확인한다는 사실에 주목하라. 준비가 되어있다면 대기중인 새로운 연결이 있다는 의미이고, accept()한 후에 master집합에 추가한다. 비슷하게 클라이언트 연결을 읽을 준비가 되고 recv()가 0을 돌려준다면 클라이언트가 연결을 닫았다는 사실을 알 수 있고, 우리는 그 연결을 master 집합에서 제거해야 한다.

클라이언트에 대한 recv()가 0이 아닌 값을 돌려준다면 우리는 어떤 데이터가 도착했다는 것을 알 수 있다. 그러므로 자료를 받은 후에 master목록을 순회하면서 모든 나머지 연결된 클라이언트들에게 그 자료를 보낸다.

지금까지 전능한 select()함수에 대한 별로 단순하지 않은 개관이었다.

모든 리눅스 팬들을 위한 짧은 이야기: 드문 몇몇 상황에서 때때로 리눅스의 select() 는 실제로는 읽을 준비가 되어있지 않음에도 "읽을 준비가 되었다"고 하면서 반환한다. 이것은 select()가 읽기 동작에 대한 블로킹이 없을 것이라고 말함에도 read() 가 블록될 것임을 의미한다. 아무튼 해결책은 읽을 소켓에 O_NONBLOCK 플래그를 설정해서 EWOULDBLOCK오류가 발생하도록 하는 것이다. (이 오류가 생겨도 무시해도 된다.) 소켓을 논블로킹 모드로 설정하는 방법에 대해서는 fcntl() 참조 페이지를 참고하라.

추가로 약간의 여담을 하자면 select()와 상당히 비슷한 일을 하지만 파일 설명자 집합을 다른 방식으로 처리하는 poll()이라는 다른 함수가 있다. 한 번 확인해 보라!

7.4 부분적인 send() 처리하기

위에서 다룬 send()에 대한 절에서 send()가 당신이 전송 요청한 바이트들을 모두 보내지는 않을 수도 있다고 말한 것을 기억하는가? 당신은 512바이트를 보내길 원해도 복귀값은 412일 수 있다는 의미이다. 남은 100바이트에는 무슨 일이 생긴 것인가?

음, 그것들은 여전히 당신의 작은 버퍼에 남아서 보내지길 기다리고 있다. 당신의 통제 밖에 있는 상황때문에 커널이 데이터를 한 덩어리로 전부 보내지는 않기로 결정한 것이다. 그리고 친구여, 이제 남은 데이터를 보내는 일은 당신에게 달려있는 것이다.

그 일을 하는 함수를 만드는 한 가지 방법은 이렇다:

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

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // 몇 바이트를 보냈는가
    int bytesleft = *len; // 보내야하는 데이터는 얼마나 남아있는가
    int n;
```

```
    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }

    *len = total; // 실제로 보낸 바이트 수를 기록해서 돌려준다.

    return n==-1?-1:0; // 실패시에는 -1을, 성공시에는 0을 돌려준다.
}
```

이 예제에서 s는 당신이 데이터를 보내고 싶은 소켓이고 buf는 자료를 담은 버퍼이다. len은 버퍼에 담긴 바이트의 갯수를 담은 int에 대한 포인터이다.

함수는 오류가 발생하면 -1을 돌려준다(errno는 send()에 대한 호출로 인해 여전히 설정되어 있다). 또한 실제로 전송된 바이트의 갯수가 len을 통해 반환된다. 이 값은 오류가 발생하지 않는 이상 당신이 전송하라고 요청한 바이트의 수와 같다. sendall()은 데이터를 전송하기 위해서 최선을 다하지만 오류가 발생하면 바로 당신에게 알려줄 것이다.

완결성을 위해 이 함수에 대한 호출 예제가 여기 있다:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("오류가 발생해서 %d바이트만 전송했습니다!\n", len);
}
```

수신자 측에 패킷의 일부가 도착하면 무슨 일이 벌어질까? 만약 패킷이 가변 길이일 경우 수신자는 어떻게 하나의 패킷이 끝나고 다른 하나가 시작되는 것을 알까? 그렇다. 실세계의 시나리오는 뒤지게 고통스럽다. 당신은 아마도 캡슐화 를 해야 할 것이다. (시작부에 있던 데이터 캡슐화 절을 기억하는가?) 자세한 내용을 알고싶다면 계속 읽어보자!

7.5 직렬화—데이터를 포장하는 방법

네트워크를 통해 문자열 데이터를 보내는 것은 꽤 쉽다는 것을 이제 알 것이다. 하지만 만약 int나 float같은 "이진" 자료를 전송하려고 하면 어떻게 해야하는가? 몇 가지 방법이 있다.

1. sprintf()같은 함수를 써서 수를 텍스트로 변환하고 텍스트를 전송한다. 수신자는 strtol()같은 함수를 써서 텍스트를 다시 숫자로 변환한다.
2. send()에 데이터를 가리키는 포인터를 전달해서 원시 데이터를 그대로 전송한다.
3. 데이터를 호환성 있는 바이너리 형태로 인코드한다. 수신자는 디코드한다.

오늘밤의 특별 사시회!

[커튼이 올라간다]

Beej가 말합니다: "저는 위의 세 번째 방법을 좋아합니다."

[끝]

(이 절을 기쁘게 시작하기에 앞서, 이 일을 하기 위한 라이브러리들이 이미 있다는 말을 미리 해야겠다. 이식성 있고 오류가 없는 당신만의 라이브러리를 만드는 작업은 꽤 어려운 일이 될 것이다. 그러므로 그런 작업을 직접 하기 전에 그것들을 살펴보고 해야하는 다른 일을 처리하는 것이 나을 것이다. 필자는 그런 것이 어떻게 동작하는지 궁금할 독자들을 위해서 관련된 내용을 여기에 담을 뿐이다.)

사실 위에 언급한 모든 방법에 각각의 장점과 단점이 있다. 그러나 위에 말한대로, 필자는 일반적으로 세 번째 방법을 선호한다. 그러나 먼저 다른 두 가지 방법의 장단점에 대해서 조금 더 알아보자.

수를 보내기 전에 텍스트로 인코딩하는 첫 번째 방법은 랜선을 타고 오는 정보를 출력하고 읽어보기가 쉽다는 장점이 있다. Internet Relay Chat (IRC)[7] 처럼 인간이 읽을 수 있는 프로토콜은 때때로 전송 대역폭에 민감하지 않은 상황에서 사용하기에 아주 훌륭하다. 그러나 그것은 변환이 느리다는 단점이 있고, 결과물은 언제나 원본 수보다 더 많은 공간을 차지한다는 단점이 있다.

두 번째 방법: 원시 데이터(원문: raw data)를 넘기기. 이것은 꽤 간단(하고 위험)하다: 보낼 데이터에 대한 포인터를 얻은 후 그것에 send를 호출한다.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0); /* 위험--이식성 없음!(역자 주 : 이식성은 컴퓨터 프로그램이나 소스코드가 서로 다른 구조를 가진 컴퓨터에서 동작하는 특성을 의미한다.) */
```

수신자는 이것을 아래와 같이 받는다:

```
double d;

recv(s, &d, sizeof d, 0);  /* 위험--이식성 없음! */
```

빠르고, 간단하다—문제될 것이 없지않은가? 사실, 모든 아키텍처들이 double (이나 다른 예로는 int)을 동일한 비트 표현이나 심지어 동일한 바이트 순서로 표시하는 것은 아니라는 문제가 있다! 위의 코드는 절대로 이식성이 없다. (잠깐—이식성이 필요 없는 상황도 있지 않을까? 그렇다면 이 방식은 좋고 빠른 방법이 된다.)

정수 자료형을 포장할 때 htons()-수를 네트워크 바이트 순서로 변환해주는 종류의 함수를 어떻게 쓰는지, 그리고 그것이 왜 필요한지 이미 살펴보았다. 불행하게도 float자료형에 대해서는 유사한 함수가 없다. 희망이 없는 것일까?

두려워 말라!(잠시 두려움을 느꼈는가? 두렵지 않았는가? 아주 조금도?) 우리에게 방법이 있다: 우리는 데이터를 원격지에서 풀어낼 수 있는 알려진 방식으로 포장(원문 : pack)(또는 "marshal" 또는 "직렬화" 그것도 아니면 그런 일에 대한 천만개의 다른 이름)을 할 수 있다.

"알려진 이진 형식"은 무엇일까? 우리는 이미 htons()의 예제를 보았다. 그것은 수를 호스트의 형식이 무엇이든간에 네트워크 바이트 순서로 변환(또는 "인코드", 이것이 더 이해하기 쉽다면)한다. 수를 원래대로 돌려놓기(디코드) 위해서 수신자는 ntohs()를 호출해야 한다.

하지만 필자가 조금 전에 비-정수 타입에 대해서는 그런 함수가 없다고 하지 않았던가? 그렇다. 그리고 C에서 이것을 처리하는 표준 방법이 없기 때문에 이것은 조금 까다로운 일이다. (파이썬 팬들은 이런 작업을 할 필요가 없을 것이다.)

필요한 작업은 데이터를 알려진 형식으로 포장하고 유선상으로 실어보내는 것이다. 예를 들어서 float를 포장하는 작업을 위한 간단하고 지저분하고 개선할 점이 많은 예제코드[8] 가 있다:

```
#include <stdint.h>

uint32_t htonf(float f)
{
    uint32_t p;
    uint32_t sign;

    if (f < 0) { sign = 1; f = -f; }
    else { sign = 0; }

    p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31); // 전체 부분과 부호
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // 소수점

    return p;
}

float ntohf(uint32_t p)
{
    float f = ((p>>16)&0x7fff); // 전체
```

[7] https://en.wikipedia.org/wiki/Internet_Relay_Chat
[8] https://beej.us/guide/bgnet/examples/pack.c

```
    f += (p&0xffff) / 65536.0f; // 소수점

    if (((p>>31)&0×1) == 0×1) { f = -f; } // 부호 비트 설정

    return f;
}
```

위의 코드는 float를 32비트 수에 저장하기 위한 단순한 구현이다. 최상위 비트(31)가 수의 부호를 저장하기 위해 쓰인다. ("1"이 음수를 의미한다.) 다음 15비트(30-16)(역자 주 : 원문에서는 7비트라고 적혀 있으나 코드의 내용상 오타로 보임)가 float의 전체 수 부분을 저장하기 위해서 쓰인다. 마지막으로 남은 비트들(15-0)이 수의 소수점 부분을 기록하기 위해서 쓰인다.

사용법은 꽤 직관적이다:

```
#include <stdio.h>

int main(void)
{
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // "네트워크" 형식으로 변환
    f2 = ntohf(netf); // 시험을 위해 원래대로 변환

    printf("Original: %f\n", f);       // 3.141593
    printf(" Network: 0x%08X\n", netf); // 0×0003243F
    printf("Unpacked: %f\n", f2);      // 3.141586

    return 0;
}
```

장점을 보자면, 이 코드는 작고 간단하며 빠르다. 단점을 보자면 이 방식은 공간을 효율적으로 쓰지 않으며 표현 범위가 상당히 제한되어 있다.—32767보다 큰 수를 저장하려고 하면 이 방법은 제대로 동작하지 않을 것이다! 또한 여러분은 위의 예제에서 소수점의 마지막 2자리가 제대로 보존되지 않은 것을 볼 수 있다.

이것을 해결하려면 어떻게 해야할까? 사실 부동소수점 수를 저장하기 위한 표준 은 IEEE-754[9]로 알려져 있다. 대부분의 컴퓨터는 부동소수점 계산을 위해서 내부적으로 이 형식을 사용한다. 그러므로 그런 경우라면 엄밀히 말하자면 변환을 수행할 필요는 없다. 그러나 여러분의 소스코드가 이식성이 있기를 바란다면 그런 가정을 할 수는 없다. (한 편으로 만약 속도를 원한다면 변환이 필요없는 플랫폼에서는 변환 작업을 제거하는 최적화를 해야함을 의미한다. htons()및 그와 유사한 함수들은 그런 방식으로 동작한다.)

여기 단정밀도 부동소수점 및 배정밀도 부동소수점 타입을 IEEE-754로 인코드하는 코드가 있다[10]. (엄밀히는 거의 대부분을 인코드한다. 이 코드는 NaN이나 Infinity를 처리하지 않는다. 그러나 그런 처리가 가능하게 수정할 수도 있다.)

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
    long double fnorm;
    int shift;
    long long sign, exp, significand;
    unsigned significandbits = bits - expbits - 1; // 부호 비트를 위해 1을 뺀다.

    if (f == 0.0) return 0; // 특별한 경우의 처리
```

[9] https://en.wikipedia.org/wiki/IEEE_754

[10] https://beej.us/guide/bgnet/examples/ieee754.c

```
    // 부호를 확인하고 정규화를 시작한다.
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // 정규화된 형태의 f를 얻어내고 지수를 추적한다.
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // 실수부의 부동소수점이 아닌 이진 표현을 구한다.
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // 바이어스를 더한 지수부를 구한다.
    // (역자 주 : IEEE754에서는 지수부를 일정 비트의 정수로 나타내며,
    // 바이어스보다 큰 수는 바이어스와 계산한 차의 절대값 만큼의 양의 지수,
    // 바이어스 미만은 바이어스와 계산한 차의 절대값만큼의 음의 지수를 나타낸다.)
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // 최종 값을 돌려준다.
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}

long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;
    unsigned significandbits = bits - expbits - 1; // 부호 비트를 위해서 -1

    if (i == 0) return 0.0;

    // 실수부를 뽑아낸다.
    result = (i&((1LL<<significandbits)-1)); // 마스크 처리
    result /= (1LL<<significandbits); // 부동소수점으로 변환
    result += 1.0f; // 1을 다시 더한다.

    // 지수부를 처리한다.
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // 부호처리
    result *= (i>>(bits-1))&1? -1.0: 1.0;

    return result;
}
```

32비트(아마도 float)와 64비트(아마도 double) 수를 위한 패킹과 언패킹 매크로를 위에 넣어두었다. 그러나 bits크기의 데이터를 인코드하기위해서 pack754()함수를 직접 호출할 수도 있을 것이다.(expbits 만큼의 지수부가 정규화된 수의 지수로 보존될 것이다.)

여기 사용 예시가 있다:

```
#include <stdio.h>
#include <stdint.h> // uintN_t 형들을 정의한다.
#include <inttypes.h> // PRIx 매크로들을 정의한다.

int main(void)
{
   float f = 3.1415926, f2;
   double d = 3.14159265358979323, d2;
   uint32_t fi;
   uint64_t di;

   fi = pack754_32(f);
   f2 = unpack754_32(fi);

   di = pack754_64(d);
   d2 = unpack754_64(di);

   printf("float before : %.7f\n", f);
   printf("float encoded: 0x%08" PRIx32 "\n", fi);
   printf("float after  : %.7f\n\n", f2);

   printf("double before : %.20lf\n", d);
   printf("double encoded: 0x%016" PRIx64 "\n", di);
   printf("double after  : %.20lf\n", d2);

   return 0;
}
```

위의 코드는 아래의 출력을 생성한다:

```
float before : 3.1415925
float encoded: 0×40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0×400921FB54442D18
double after  : 3.14159265358979311600
```

여러분이 가질 수 있는 또 다른 질문은 어떻게 sturct를 포장하는가이다. 불행히도 컴파일러는 struct의 모든 곳에 자유롭게 패딩을 넣을 수 있다. 그리고 그것은 구조체 전체를 한 번에 네트워크에 전송할 수는 없다는 것을 의미한다. (“이건 되고”, “이건 안되고”를 듣는 것이 지겨운가? 미안하다. 내 친구의 말을 빌리자면 “뭔가 잘못되면 나는 늘 마이크로소프트를 탓한다.” 이 경우에는 아마도 마이크로소프트의 잘못만은 아닐 것이다. 그러나 내 친구의 선언은 완전히 옳다.)

다시 주제로 돌아가서: stuct를 전송하기 위한 최고의 방법은 각각의 필드를 독립적으로 포장한 다음 반대편에 도착하면 다시 struct안에 풀어넣는 것이다.

여러분은 이것이 굉장히 큰 작업일 것이라 예상할 것이다. 맞다. 여러분이 할 일은 데이터를 포장하는 일을 도와줄 도우미 함수를 작성하는 것이다. 재미있을 것이다! 정말로!!

Kernighan(역자 주 : 커니건)과 Pike가 지은 The Practice of Programming[11] 에서 그들은 바로 그 일을 하도록 printf()와 유사한 pack()과 unpack()함수를 작성했다. 그것에 대한 링크를 제공하고 싶지만 그 함수들과 책의 다른 소스코드는 온라인으로 제공되지 않고 있다.

(The Practice of Programming은 아주 좋은 책이다. 필자가 그 책을 추천할 때마다 제우스 신이 고양이를 한 마리씩 구한다. (역자 주 : 아주 좋은 선행이라는 뜻))

[11] https://beej.us/guide/url/tpop

이 시점에서 필자는 프로토콜 버퍼의 C 구현체[12] 에 대한 링크를 제공하려 한다. 필자는 이것을 써 본 적이 없으나 훌륭한 코드로 보인다. 파이썬과 펄 프로그래머들은 같은 일을 하기 위해서 그들의 언어가 가진 pack()과 unpack() 함수를 확인해보길 바란다. 자바는 유사한 방식으로 사용할 수 있는 Serializable 인터페이스를 가지고 있다.

그러나 만약 여러분이 자신만의 패킹 유틸리티를 C언어로 작성하고 싶다면, K&P의 해결책은 패킷을 만들기 위해서 가변 길이 매개변수를 활용하는 printf()와 유사한 함수를 만드는 것이다. 여기 필자가 만든 버전이 있으며[13] 여러분이 그런 것이 어떻게 동작하는지 알기에 충분할 것이다.

(이 코드는 위의 pack754()함수를 참조한다. packi*()함수는 또 다른 정수가 아닌 char의 배열에 수를 담는다는 점을 제외하면 htons()계열 함수와 유사하게 동작한다.)

```
#include <stdio.h>
#include <ctype.h>
#include <stdarg.h>
#include <string.h>

/*
** packi16() -- 16비트를 char 버퍼에 저장한다. (htons()처럼)
*/
void packi16(unsigned char *buf, unsigned int i)
{
   *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- 32비트를 char 버퍼에 저장한다. (htonl()처럼)
*/
void packi32(unsigned char *buf, unsigned long int i)
{
   *buf++ = i>>24; *buf++ = i>>16;
   *buf++ = i>>8;  *buf++ = i;
}

/*
** packi64() -- 64비트를 char 버퍼에 저장한다. (htonl()처럼)
*/
void packi64(unsigned char *buf, unsigned long long int i)
{
   *buf++ = i>>56; *buf++ = i>>48;
   *buf++ = i>>40; *buf++ = i>>32;
   *buf++ = i>>24; *buf++ = i>>16;
   *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() -- 16비트 정수를 char 버퍼에서 풀어낸다. (ntohs()처럼)
*/
int unpacki16(unsigned char *buf)
{
   unsigned int i2 = ((unsigned int)buf[0]<<8) | buf[1];
   int i;

   // change unsigned numbers to signed
```

[12] https://github.com/protobuf-c/protobuf-c
[13] https://beej.us/guide/bgnet/examples/pack2.c

```
    if (i2 <= 0×7fffu) { i = i2; }
    else { i = -1 - (unsigned int)(0xffffu - i2); }

    return i;
}

/*
** unpacku16() -- 16비트 부호없는 정수를 char 버퍼에서 풀어낸다. (ntohs()처럼)
*/
unsigned int unpacku16(unsigned char *buf)
{
    return ((unsigned int)buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- 32비트 정수를 char 버퍼에서 풀어낸다. (ntohl()처럼)
*/
long int unpacki32(unsigned char *buf)
{
    unsigned long int i2 = ((unsigned long int)buf[0]<<24) |
                           ((unsigned long int)buf[1]<<16) |
                           ((unsigned long int)buf[2]<<8)  |
                           buf[3];
    long int i;

    // change unsigned numbers to signed
    if (i2 <= 0×7fffffffu) { i = i2; }
    else { i = -1 - (long int)(0xffffffffu - i2); }

    return i;
}

/*
** unpacku32() -- 32비트 부호없는 정수를 char 버퍼에서 풀어낸다. (ntohl()처럼)
*/
unsigned long int unpacku32(unsigned char *buf)
{
    return ((unsigned long int)buf[0]<<24) |
           ((unsigned long int)buf[1]<<16) |
           ((unsigned long int)buf[2]<<8)  |
           buf[3];
}

/*
** unpacki64() -- 32비트 정수를 char 버퍼에서 풀어낸다. (ntohl()처럼)
*/
long long int unpacki64(unsigned char *buf)
{
    unsigned long long int i2 = ((unsigned long long int)buf[0]<<56) |
                                ((unsigned long long int)buf[1]<<48) |
                                ((unsigned long long int)buf[2]<<40) |
                                ((unsigned long long int)buf[3]<<32) |
                                ((unsigned long long int)buf[4]<<24) |
                                ((unsigned long long int)buf[5]<<16) |
```

```
                        ((unsigned long long int)buf[6]<<8)  |
                        buf[7];
    long long int i;

    // change unsigned numbers to signed
    if (i2 <= 0x7fffffffffffffffu) { i = i2; }
    else { i = -1 -(long long int)(0xffffffffffffffffu - i2); }

    return i;
}

/*
** unpacku64() -- 64비트 부호없는 정수를 char 버퍼에서 풀어낸다. (ntohl()처럼)
*/
unsigned long long int unpacku64(unsigned char *buf)
{
    return ((unsigned long long int)buf[0]<<56) |
           ((unsigned long long int)buf[1]<<48) |
           ((unsigned long long int)buf[2]<<40) |
           ((unsigned long long int)buf[3]<<32) |
           ((unsigned long long int)buf[4]<<24) |
           ((unsigned long long int)buf[5]<<16) |
           ((unsigned long long int)buf[6]<<8)  |
           buf[7];
}

/*
** pack() -- 버퍼의 형식화 문자열이 지시한 방식으로 데이터를 저장한다.
**
**   bits |signed   unsigned   float   string
**   -----+----------------------------------
**      8 |   c        C
**     16 |   h        H         f
**     32 |   l        L         d
**     64 |   q        Q         g
**      - |                      s
**
**  (16비트 부호없는 길이가 자동으로 문자열의 앞에 붙는다.)
*/

unsigned int pack(unsigned char *buf, char *format, ...)
{
    va_list ap;

    signed char c;              // 8비트
    unsigned char C;

    int h;                      // 16비트
    unsigned int H;

    long int l;                 // 32비트
    unsigned long int L;

    long long int q;            // 64비트
```

```
    unsigned long long int Q;

    float f;                    // 부동소수점
    double d;
    long double g;
    unsigned long long int fhold;

    char *s;                    // 문자열
    unsigned int len;

    unsigned int size = 0;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'c': // 8비트
            size += 1;
            c = (signed char)va_arg(ap, int); // 자료형 승급
            *buf++ = c;
            break;

        case 'C': // 부호없는 8비트
            size += 1;
            C = (unsigned char)va_arg(ap, unsigned int); // 자료형 승급
            *buf++ = C;
            break;

        case 'h': // 16비트
            size += 2;
            h = va_arg(ap, int);
            packi16(buf, h);
            buf += 2;
            break;

        case 'H': // 부호없는 16비트
            size += 2;
            H = va_arg(ap, unsigned int);
            packi16(buf, H);
            buf += 2;
            break;

        case 'l': // 32비트
            size += 4;
            l = va_arg(ap, long int);
            packi32(buf, l);
            buf += 4;
            break;

        case 'L': // 부호없는 32비트
            size += 4;
            L = va_arg(ap, unsigned long int);
            packi32(buf, L);
            buf += 4;
```

```
            break;

        case 'q': // 64비트
            size += 8;
            q = va_arg(ap, long long int);
            packi64(buf, q);
            buf += 8;
            break;

        case 'Q': // 부호없는 64비트
            size += 8;
            Q = va_arg(ap, unsigned long long int);
            packi64(buf, Q);
            buf += 8;
            break;

        case 'f': // 부동소수점 16비트
            size += 2;
            f = (float)va_arg(ap, double); // 자료형 승급
            fhold = pack754_16(f); // IEEE 754로 변환
            packi16(buf, fhold);
            buf += 2;
            break;

        case 'd': // 부동소수점 32비트
            size += 4;
            d = va_arg(ap, double);
            fhold = pack754_32(d); // IEEE 754로 변환
            packi32(buf, fhold);
            buf += 4;
            break;

        case 'g': // 부동소수점 64비트
            size += 8;
            g = va_arg(ap, long double);
            fhold = pack754_64(g); // IEEE 754로 변환
            packi64(buf, fhold);
            buf += 8;
            break;

        case 's': // 문자열
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
        }
    }

    va_end(ap);
```

```
    return size;
}

/*
** unpack() -- 형식화 문자열이 지정하는대로 버퍼에 데이터를 풀어놓는다.
**
**   bits |signed   unsigned   float   string
**   -----+----------------------------------
**      8 |   c       C
**     16 |   h       H         f
**     32 |   l       L         d
**     64 |   q       Q         g
**      - |                     s
**
**  (문자열은 저장된 길이에 근거해서 추출된다. 그러나 `s`의 앞에 최대 길이를 앞에 지정할 수 있다.)
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;

    signed char *c;              // 8비트
    unsigned char *C;

    int *h;                      // 16비트
    unsigned int *H;

    long int *l;                 // 32비트
    unsigned long int *L;

    long long int *q;            // 64비트
    unsigned long long int *Q;

    float *f;                    // 부동소수점
    double *d;
    long double *g;
    unsigned long long int fhold;

    char *s;
    unsigned int len, maxstrlen=0, count;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'c': // 8비트
            c = va_arg(ap, signed char*);
            if (*buf <= 0x7f) { *c = *buf;} // 부호를 다시 붙인다
            else { *c = -1 - (unsigned char)(0xffu - *buf); }
            buf++;
            break;

        case 'C': // 부호없는 8비트
            C = va_arg(ap, unsigned char*);
            *C = *buf++;
```

```
            break;

        case 'h': // 16비트
            h = va_arg(ap, int*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'H': // 부호없는 16비트
            H = va_arg(ap, unsigned int*);
            *H = unpacku16(buf);
            buf += 2;
            break;

        case 'l': // 32비트
            l = va_arg(ap, long int*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'L': // 부호없는 32비트
            L = va_arg(ap, unsigned long int*);
            *L = unpacku32(buf);
            buf += 4;
            break;

        case 'q': // 64비트
            q = va_arg(ap, long long int*);
            *q = unpacki64(buf);
            buf += 8;
            break;

        case 'Q': // 부호없는 64비트
            Q = va_arg(ap, unsigned long long int*);
            *Q = unpacku64(buf);
            buf += 8;
            break;

        case 'f': // 부동소수점
            f = va_arg(ap, float*);
            fhold = unpacku16(buf);
            *f = unpack754_16(fhold);
            buf += 2;
            break;

        case 'd': // 32비트 부동소수점
            d = va_arg(ap, double*);
            fhold = unpacku32(buf);
            *d = unpack754_32(fhold);
            buf += 4;
            break;

        case 'g': // 64비트 부동소수점
            g = va_arg(ap, long double*);
```

```
            fhold = unpacku64(buf);
            *g = unpack754_64(fhold);
            buf += 8;
            break;

        case 's': // 문자열
            s = va_arg(ap, char*);
            len = unpacku16(buf);
            buf += 2;
            if (maxstrlen > 0 && len >= maxstrlen) count = maxstrlen - 1;
            else count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

        default:
            if (isdigit(*format)) { // 최대 문자열 길이를 기록
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

        if (!isdigit(*format)) maxstrlen = 0;
    }

    va_end(ap);
}
```

그리고 위의 코드를 시연하는 프로그램[14]이 여기에 있다. 이 프로그램은 buf에 약간의 데이터를 포장한 후 다시 변수에 풀어놓는다. unpack()을 문자열 매개변수로 호출하는 경우(형식 지정자 "s") 버퍼 오버런을 방지하기 위해서 "96s"처럼 최대 길이를 앞에 붙이는 것이 현명하다는 것을 기억하라. 네트워크를 통해 받은 데이터를 풀어놓을 때에는 주의해야 한다. 악의적인 사용자가 당신의 시스템을 공격하기 위해서 악의적으로 구성된 패킷을 보낼 수 있다!

```
#include <stdio.h>

// 부동 소수점 형의 다양한 비트의 변종
// 아키텍처 별로 다르다.
typedef float float32_t;
typedef double float64_t;

int main(void)
{
    unsigned char buf[1024];
    int8_t magic;
    int16_t monkeycount;
    int32_t altitude;
    float32_t absurdityfactor;
    char *s = "Great unmitigated Zot! You've found the Runestaff!";
    char s2[96];
    int16_t packetsize, ps2;

    packetsize = pack(buf, "chhlsf", (int8_t)'B', (int16_t)0, (int16_t)37,
        (int32_t)-5, s, (float32_t)-3490.6677);
```

[14] https://beej.us/guide/bgnet/examples/pack2.c

```
    packi16(buf+1, packetsize); // 시작을 위해 패킷 사이즈를 패킷에 넣어둔다.

    printf("packet is %" PRId32 " bytes\n", packetsize);

    unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude, s2,
      &absurdityfactor);

    printf("'%c' %" PRId32" %" PRId16 " %" PRId32
          " \"%s\" %f\n", magic, ps2, monkeycount,
          altitude, s2, absurdityfactor);

    return 0;
}
```

여러분이 직접 만든 코드를 쓰건 다른 사람이 작성한 것을 쓰건, 매번 각 비트를 수동으로 포장하기보다는 버그를 쉽게 잡아내기 위해서 일반적인 데이터 패킹 루틴을 사용하는 것이 좋은 습관이다.

데이터를 포장할 때에 쓰기 좋은 형식은 무엇일까? 아주 좋은 질문이다. 다행히도 RFC 4506[15], 외부 데이터 표현 표준이 부동소수점과 정수, 배열 등 다양한 자료형에 대해서 이진 형식을 정의한다. 만약 데이터를 직접 처리할 생각이라면 이것을 준수하는 것을 권장한다. 그러나 반드시 그래야 하는 것은 아니다. 패킷 경찰들이 문 앞에 지키고 서 있는 것은 아니다. 최소한 필자는 그렇지 않을 것이라고 생각한다.

어떤 경우에도, 데이터를 보내기 전에 인코드 하는 것이 옳은 일이다.

7.6 망할 데이터 캡슐화

아무튼 데이터 캡슐화가 정말로 의미하는 것은 무엇인가? 가장 단순한 경우 그것은 여러분이 데이터에 약간의 식별 정보나 패킷 길이 혹은 둘 모두를 담은 헤더를 붙여둔다는 뜻이 된다.

헤더가 어떤 모양을 하고 있어야 할까? 사실 여러분의 프로젝트를 끝내기 위해서 필요하다고 느끼는 어떤 이진 데이터면 된다.

와. 정말 막연한 이야기다.

좋다. 예를 들자면 SOCK_STREAM을 사용하는 다중 사용자 대화 프로그램이 있다고 하자. 한 사용자가 뭔가 입력한다면, 두 조각의 정보가 서버에 전달되어야 한다. 무엇을 말했는지, 그리고 누가 말했는지.

여기까지는 좋은가? "그럼 무엇이 문제인가?"라고 여러분은 질문할 것이다.

문제는 메시지가 가변 길이일 수 있다는 점이다. "Tom"이라는 사용자가 "Hi"라고 말할 수 있고 "Benjamin"이라는 또다른 사용자가 "Hey guys what is up?"이라고 말할 수도 있다.

그것이 들어오는대로 클라이언트에게 send()한다고 하자. 여러분의 송출 자료 스트림은 아래와 같을 것이다.

t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?

이런 식일 것이다. 클라이언트가 어떻게 하면 메시지의 시작과 끝을 알 수 있겠는가? 원한다면 모든 메시지가 같은 길이를 갖도록 하고 우리가 구현한 sendall() 함수를 그냥 호출할 수 있을 것이다. 그러나 그렇게 하면 대역폭을 낭비하게 된다! "tom"이 "Hi"라고 말하는 일을 위해서 1024바이트를 send()하고싶지는 않을 것이다.

그래서 우리는 데이터를 작은 헤더와 패킷 구조에 캡슐화 한다. 클라이언트와 서버 모두 이 데이터를 어떻게 포장하고 풀어내는지(때때로 "marshal"과 "unmarshal" 이라고 부른다) 알고있다. 지금 이해할 필요는 없지만 우리는 클라이언트와 서버가 어떻게 통신하는지를 정의하는 프로토콜 을 정의하려고 하고있다.

지금은 사용자의 이름이 '\0'으로 패드된 고정된 8개의 문자라고 가정하자. 데이터는 최대 128개 문자로 구성되는 가변길이 형태라고 하자. 이 상황에서 쓸 수 있는 예제 패킷 구조를 살펴보자.

1. len (1바이트, 부호 없음)—패킷의 전체 길이, 8바이트의 사용자 이름과 대화 데이터의 길이를 센다.
2. name (8 바이트)—사용자의 이름, 필요한 경우 0이 덧대진다.

[15] https://tools.ietf.org/html/rfc4506

3. chatdata (n 바이트)—데이터 자체, 최대 128바이트. 패킷의 길이는 이 데이터의 길이에 8을 더한 값으로 계산되어야 한다.(위에서 언급한 이름 필드의 길이)

필자가 8바이트와 128바이트를 필드의 길이 제한으로 선택한 이유가 궁금한가? 특별한 이유는 없고, 충분히 길 것이라 생각했다. 그러나 아마도 8바이트는 여러분의 필요에는 조금 못 미칠수도 있다. 그런 경우에는 이름 필드의 길이를 30바이트나 다른 값으로 설정할 수 있다. 선택은 여러분의 몫이다.

위의 패킷 정의를 사용하는 첫 번째 패킷은 아래와 같은 정보로 구성될 수 있다. (16진수와 아스키 코드로 표시되었다.):

```
  0A     74 6F 6D 00 00 00 00 00      48 69
(길이)  T  o  m   (패딩)        H  i
```

두 번째 패킷도 비슷하다.

```
  18     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(길이)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(길이는 물론 네트워크 바이트 순서로 기록되어 있다. 이 경우 길이가 단일 바이트이므로 그것이 중요하지는 않지만, 일반적으로는 여러분의 패킷이 가지는 모든 이진 정수가 네트워크 바이트 순서로 기록되기를 원할 것이다.)

이 데이터를 보낼 때 여러분은 위에서 제시된 sendall()과 비슷한 함수를 쓸 수 있다. 그렇게 하면 데이터를 모두 전송하기 위해서 send()를 여러 번 호출하는 한이 있어도 모든 데이터가 전송되는 것을 확신할 수 있다.

마찬가지로 이 데이터를 받을 때에도 약간의 추가적인 작업이 필요하다. 이 데이터를 받을 때에도 부분적인 패킷(예를 들어 위의 벤자민으로부터 recv()로 받은 것이 "18 42 65 6E 6A"뿐일 수도 있다.)을 받을 가능성을 염두에 둬야 한다. 전체 패킷을 받을 때까지 recv()를 반복적으로 호출해야 한다.

그러나 어떻게 해야할까? 우리는 패킷이 완성되기 위해서 받아야 하는 바이트의 총 갯수를 알고있다. 갯수가 패킷의 앞쪽에 붙어있기 때문이다. 우리는 또한 패킷의 최대 크기가 1 + 8 + 128, 즉 137바이트라는 것을 알고있다. (우리가 그렇게 정의했기 때문이다.)

여기에서는 몇 가지 방식으로 일을 할 수 있다. 모든 패킷이 길이 정보로 시작한다는 것을 알고있으므로 패킷의 길이를 얻기 위해서 recv()를 호출할 수 있다. 그리고 길이를 가지고 있으면 전체 패킷을 받을 때까지 남은 길이를 명시하면서 recv() 를(아마도 반복적으로) 호출하는 것이다. 이 방식의 장점은 하나의 패킷을 담기에 충분한 크기의 버퍼만 있으면 된다는 것이고, 단점은 모든 데이터를 받기 위해서 recv()를 최소 두 번 호출해야 한다는 것이다.

다른 옵션은 recv()을 호출할 때 한 패킷의 최대 크기만큼을 받겠다고 지정하는 것이다. 그 후에 받은 자료를 버퍼의 뒤쪽에 쌓아두고, 패킷이 완성되었는지 확인한다. 물론 다음 패킷의 일부를 받을 수 있으므로 그것을 위한 여분의 공간이 필요하다.

이를 위해서 두 개의 패킷을 담기에 충분한 배열을 선언하면 된다. 이것은 패킷이 도착하는대로 재구성하는 일에 사용할 작업 공간이다.

자료를 recv()처리할 때마다 그것을 작업 버퍼에 덧붙이고 패킷이 완성되었는지 확인한다. 버퍼에 담긴 바이트의 갯수가 헤더에 명시된 길이보다 많거나 같은지 확인한다는 뜻이다(사실은 헤더에 헤더 자신의 길이가 포함되지 않으므로 +1을 해야한다). 만약 버퍼의 바이트 수가 1보다 적다면 물론 패킷은 완성되지 않은 것이다. 또한 이 경우 버퍼의 첫 바이트를 읽어들인다고 해도 그것은 쓰레기값이므로 그것을 감안한 처리를 해야한다.

패킷이 완성되면 여러분은 그것으로 여러분이 원하는 일을 할 수 있다. 패킷을 사용하거나, 그것을 여러분의 작업 버퍼에서 제거할 수 있다.

휴! 아직 머릿속이 어지러운가? 여기 원투펀치의 두 번째 부분이 있다. 한 번의 recv()호출로 한 패킷을 넘어서는 분량을 읽어들일 수가 있다. 즉 작업버퍼에 하나의 완전한 패킷과 다음 패킷의 불완전한 부분이 있을 수 있다는 것이다. 제기랄. (그러나 이런 경우를 처리하기 위해서 여러분의 작업 버퍼를 두 개의 패킷을 담기에 충분한 크기로 만들어둔 것이다.)

첫 패킷의 길이를 헤더를 통해 알고있고 작업 버퍼에 있는 바이트의 수를 추적하고 있으므로, 뺄셈을 해서 작업 버퍼에 있는 바이트 중 몇 개가 다음(미완성된) 패킷에 속해있는지 계산할 수 있다. 첫 번째 패킷을 처리한 후에는 그것을 작업 버퍼에서 제거하고 부분적인 두 번째 패킷을 버퍼의 앞쪽으로 옮겨서 다음 recv()를 처리할 준비를 할 수 있다.

(독자 여러분 중 일부는 부분적인 두 번째 패킷을 작업 버퍼의 앞쪽으로 옮기는 것에 시간이 걸리고, 환형 버퍼를 사용하면 그 작업이 필요하지 않다는 것에 주목할 것이다. 다른 독자들에게는 불행하게도, 환형 버퍼에 대한 논의는 이 글의 범위를 벗어난다. 흥미가 있다면 데이터 구조 책을 집어들고 거기서부터 시작할 수 있을 것이다.)

쉽다고 한 적은 없다. 사실은, 쉽다고 했다. 또 실제로도 그렇다. 단지 연습이 필요하고 오래지않아 익숙해질 것이다. 엑스칼리버에 맹세한다!

7.7 브로드캐스트(Broadcast) 패킷 — Hello, World!

지금까지 이 안내서에서는 데이터를 하나의 호스트에서 다른 호스트로 보내는 일에 대해서 이야기했다. 그러나 적절한 권한이 있다면 한 번에 여러 호스트에게 자료를 보낼 수 있다!

UDP(TCP는 안 된다)와 표준 IPv4에서 이것은 브로드캐스팅(Broadcasting) 이라는 매커니즘으로 가능하다. IPv6에서 브로드캐스팅은 지원되지 않으며, 더 상위의 기술인 멀티캐스팅(Multicasting)을 사용해야 한다. 그러나 이번에는 그것에 대해서 다루지 않을 것이다. 촉망받는 미래에 대해서는 그만 이야기하자. 우리는 32비트의 현재에 갇혀있다.

잠깐! 그러나 무작정 브로드캐스팅을 시작할 수는 없다. 네트워크에 브로드캐스트 패킷을 전송하기 전에 SO_BROADCAST 소켓 옵션을 설정해야 한다. 이것은 미사일 발사 스위치에 달아두는 플라스틱 덮개같은 것이다. 그만큼 강력한 도구라는 뜻이다.

아무튼 진지하게 말하자면 브로드캐스트 패킷을 쓰는 일에는 위험이 따른다. 브로드캐스트 패킷을 받는 모든 시스템은 반드시 그 데이터가 어떤 포트를 목적지로 삼는지 알아내기 위해서 패킷의 데이터 캡슐화 계층이라는 양파껍질을 벗겨내야 한다는 점이 바로 그것이다. 그렇게 하고 난 후에야 시스템은 데이터를 포트에 건네줄지 아니면 무시할지를 결정한다. 어떤 경우건 그것은 브로드캐스트 패킷을 받는 각각의 장치에게 큰 작업이고, 상당히 많은 장치들이 불필요한 작업을 할 수 있다. 게임 둠이 처음 세상에 나왔을 때 그것의 네트워크 코드에 대한 불평이 이런 것이었다.

자, 고양이 가죽을 벗기는 일에도 여러 방법이 있을 수 있으니 잠깐 기다려보자. (역자 주 : 서양의 속담) 잠깐, 무슨 그런 속담이 다 있는가? 정말로 고양이 가죽을 벗기는 일에 여러 방법이 있는가? 그리고 브로드캐스트 패킷을 보내는 일에도 여러 방법이 있는가? 핵심을 말하자면 이것이다. 어떻게 브로드캐스트 메시지의 목적지 주소를 지정할 수 있는가? 두 개의 일반적인 방법이 있다.

1. 데이터를 특정 서브넷의 브로드캐스트 주소로 보낸다. 이것은 주소의 모든 호스트 부분 비트가 1로 설정된 서브넷 네트워크 주소이다. 예를 들어 필자의 네트워크는 집에서 192.168.1.0이고 넷마스크는 255.255.255.0이다. 그러므로 주소의 마지막 바이트가 호스트 번호이다(넷마스크에 따라 첫 세 바이트가 네트워크 주소이기 때문이다). 그러므로 필자의 브로드캐스트 주소는 192.168.1.255 이다. 유닉스에서는 ifconfig 명령이 이 모든 정보를 줄 것이다. (궁금한 분을 위해 적자면 브로드캐스트 주소를 얻기 위한 비트단위 논리연산은 네트워크 번호 OR (NOT 넷마스크)이다.) 여러분은 이 종류의 브로드캐스트 패킷을 로컬 네트워크 뿐 아니라 원격 네트워크에도 보낼 수 있다. 그러나 이 경우 목적지의 라우터가 패킷을 무시할 가능성이 존재한다. (이런 패킷을 무시하지 않으면 공격자가 브로드캐스트 통신을 과다하게 발송할 수 있다.)
2. 데이터를 "전역" 브로드캐스트 주소로 보낸다. 이것은 255.255.255.255, 통칭 INADDR_BROADCAST다. 많은 장치들은 이것을 자동으로 여러분의 네트워크 번호와 비트단위 AND연산 해서 네트워크 브로드캐스트 주소로 변환할 것이다. 그러나 일부는 그렇게 하지 않을 것이다. 그것은 장치별로 다르다. 역설적이게도 라우터들은 이 종류의 브로드캐스트 패킷을 로컬 네트워크 너머로 전송하지 않는다.

SO_BROADCAST 소켓 옵션을 지정하지 않고 브로드캐스트 주소에 데이터를 보내려고 하면 어떤 일이 생길까? 오래됐지만 유용한 talker와 listener 를 실행해보고 무슨 일이 생기는지 보자.

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

별로 좋지 않은 상황이다. SO_BROADCAST을 설정하지 않았기 때문이다. 설정을 한 뒤에는 원하는 곳 어디에든 sendto()를 할 수 있다.

사실 그것이 브로드캐스트를 할 수 있는 UDP 응용프로그램과 그렇지 않은 응용프로그램의 유일한 차이다. 그러나 오래된 talker 응용프로그램에 SO_BROADCAST 소켓 옵션을 설정하는 부분을 하나 추가해보자. 이 프로그램을 broadcaster.c[16]라고 부를 것이다.

```
/*
** broadcaster.c -- talker.c와 같은 데이터그램 클라이언트, 다만
**                  이 프로그램은 브로드캐스트를 할 수 있다.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

[16] https://beej.us/guide/bgnet/examples/broadcaster.c

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

#define SERVERPORT 4950 // 사용자들이 연결할 포트

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // 연결자(Connector)의 주소 정보
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // 동작하지 않으면 이것을 써 보라

    if (argc != 3) {
        fprintf(stderr,"usage: broadcaster hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // 호스트 정보를 받아온다
        perror("gethostbyname");
        exit(1);
    }

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

    // 이 호출이 브로드캐스트 패킷을 보낼 수 있게 만든다
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // 호스트 바이트 순서
    their_addr.sin_port = htons(SERVERPORT); // 숏, 네트워크 바이트 순서
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

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

    printf("sent %d bytes to %s\n", numbytes,
        inet_ntoa(their_addr.sin_addr));
```

```
    close(sockfd);

    return 0;
}
```

이 프로그램과 "평범한" UDP 클라이언트/서버의 상황에는 어떤 차이가 있을까? 아무 것도 없다! (이 경우에는 클라이언트가 브로드캐스트 패킷을 보낼 수 있다는 점을 빼면) 앞서와 마찬가지로 이전에 언급한 UDP listener를 한 창에 실행하고 다른 창에 broadcaster를 실행하라. 위에서 실패한 전송이 성공할 것이다.

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

listener가 수신한 패킷에 반응하는 것을 볼 수 있어야 한다. (만약 listener가 반응하지 않는다면 그것이 IPv6주소에 연결되어서 그럴 수 있다. listener.c의 AF_INET6를 AF_INET로 바꿔서 IPv4를 강제해보라.)

자, 여기까지도 조금 재미있었다. 그러나 여러분이 가진 다른 장치 중 같은 네트워크에 있는 것에서 listener를 실행해서 각 장치에 1개씩 실행되게 한 후에 broadcaster에 브로드캐스트 주소를 넣고 다시 실행해보자. sendto()를 한 번만 실행했음에도 두 개의 listener 모두가 패킷을 받는다! 멋지다!

만약 listener가 실행중인 장치의 아이피 주소를 목적지로 발송된 데이터는 받는데 브로드캐스트 주소로 보낸 데이터는 받지 못한다면 아마도 방화벽이 장치에 있어서 패킷을 막고 있을 것이다. (그래요, Pat, Bapper. 이게 제 샘플 코드가 동작하지 않을 수 있는 이유라는 것을 나보다 먼저 깨달아줘서 고마워요. 안내서에 여러분을 언급하겠다고 했지요. 바로 이 부분에 적었습니다.)

다시 말하지만 브로드캐스트 패킷을 다룰 때에는 주의하라. LAN(역자 주 : Local Area Network) 에 있는 모든 장치들이 recvfrom() 수행 여부와 상관없이 패킷을 처리해야 하므로 전체 컴퓨터 네트워크에 상당한 부하를 줄 수 있다. 브로드캐스트 패킷은 반드시 가끔씩만, 그리고 적절한 상황에서만 쓰여야 한다.

Chapter 8

일반적인 질문들

이 헤더파일들은 어디서 찾을 수 있는가?

여러분의 시스템에 헤더파일이 없다면 아마도 그것이 필요하지 않을 것이다. 사용중인 플랫폼의 설명서를 참고하라. Windows를 위해 작업하고 있다면 #include <winsock.h>만 하면 된다.

bind()가 "Address already in use" 를 보고하면 어떻게 해야하는가?

리스닝 소켓에 setsockopt()를 SO_REUSEADDR옵션과 함께 사용해야 한다. 예제가 필요하다면 bind()에 관한 절과 select()에 관한 절을 참고하라.

시스템에 열린 소켓의 목록을 얻으려면 어떻게 해야하는가?

netstat를 사용하라. 완전한 정보에 대해서는 man 을 참고해야 하지만 아래와 같이 입력해도 약간의 유용한 출력을 얻을 수 있다.

$ netstat

비결은 어떤 소켓이 어떤 프로그램과 연결되어 있는지 알아내는 것이다. :-)

라우팅 테이블을 보려면 어떻게 해야하는가?

route명령(대개의 리눅스 장치에서 /sbin에 있다) 을 실행하라. 아니면 netstat -r 명령을 실행하라. 혹은 ip route 명령일 수도 있다.

컴퓨터가 하나밖에 없다면 어떻게 클라이언트와 서버 프로그램을 실행하는가? 네트워크 프로그램을 작성하려면 네트워크가 있어야 하는 것 아닌가?

여러분에게는 다행히고 사실상 모든 장치가 커널에 자리잡고 네트워크 카드인 척 하는 루프백 네트워크 "장치"를 구현한다. (이것은 라우팅 테이블에서 "lo"라는 이름으로 표시되는 인터페이스이다.)

여러분이 "goat"라는 이름의 장치에 로그인했다고 하자. 클라이언트를 하나의 창에서 실행하고 서버를 다른 창에서 실행하자. 아니면 서버를 백그라운드에서 실행하고("server &") 클라이언트를 같은 창에서 실행하자. 루프백 장치는 여러분이 client goat와 client localhost("localhost"는 여러분의 /etc/hosts파일에 정의되어 있을 것이다.) 중 어떤 것이든 할 수 있게 해 줄 것이고 네트워크 없이도 서버와 대화하는 클라이언트 프로그램을 시험할 수 있을 것이다.

간단히 말하자면 네트워크 없는 단일 장치에서 코드를 실행하기 위해서 코드를 변경할 필요는 없다.

원격지 측에서 연결을 닫았는지 어떻게 알 수 있는가?

recv()가 0을 돌려주는 것으로 알 수 있다.

"ping" 유틸리티를 만들려면 어떻게 해야하는가? ICMP는 무엇인가? raw 소켓과 SOCK_RAW 에 대해서는 어디에서 더 알아볼 수 있는가?

raw 소켓에 대한 모든 질문은 W. Richard Stevens' UNIX Network Programming books 에서 답을 얻을 수 있다. 또한 온라인으로 사용 가능한[1] Stevens' UNIX Network Programming source code에서 ping/ 하위디렉터리를 살펴보라.

[1] http://www.unpbook.com/src.html

connect()에 대한 제한시간을 변경하거나 단축할 수 있는가?

W. Richard Stevens이 여러분에게 줄 수 있는 답과 동일한 답을 주는 대신, UNIX Network Programming source code의 lib/connect_nonb.c[2] 를 안내해주겠다.

요점은 socket()으로 소켓 설명자를 만든 후 논 블로킹으로 만든 후에 connect()를 호출할 때 모든 것이 잘 돌아간다면 connect()는 즉시 -1을 반환할 것이고 errno는 EINPROGRESS로 설정될 것이라는 것이다. 그 후 select()를 호출할 때 소켓 설명자를 읽기와 쓰기 집합에 모두 넣으면서 여러분이 원하는 제한시간을 지정하면 된다. 시간초과가 발생하지 않으면 connect() 이 완료되었다는 의미다. 이 시점에서 getsockopt()을 SO_ERROR옵션과 함께 호출해서 connect()호출의 반환값을 얻을 수 있고, 오류가 없었다면 그 값은 0이어야 한다.

마지막으로 여러분은 아마도 해당 소켓에 데이터를 전송하기 전에 소켓을 다시 블로킹 모드로 설정하고 싶을 것이다.

이 방식은 프로그램이 연결을 시작하는 동안 다른 일을 할 수 있게 해 주는 장점도 있음에 주목하라. 예를 들어 500ms정도의 짧은 제한시간을 설정한 후 select()를 다시 호출하자. select()가 20번 정도 시간초과를 일으킨다면 연결을 포기할 때가 되었음을 알 수 있다.

말했듯이 완벽하게 훌륭한 예제가 필요하다면 Stevens의 코드를 참고하라.

Windows를 위해 빌드하려면 어떻게 하는가?

먼저 윈도우를 삭제한 후 리눅스나 BSD를 설치하라. };-). 사실 그럴 필요는 없고, 도입부의 윈도우즈에서 빌드하기를 위한 절을 살펴보라.

Solaris/SunOS에서 빌드하려면 어떻게 하는가? 컴파일을 시도하면 계속 링커 오류가 발생한다!

링커 에러는 Sun사의 장치들이 자동적으로 소켓 라이브러리를 링크하지 않기 때문에 발생한다. 컴파일을 위해서는 도입부의 Solaris/SunOS를 위한 절 을 참고하라.

왜 select()가 시그널을 받으면 실패하는가?

시그널은 중단된 시스템 콜이 errno를 EINTR로 설정하고 -1을 반환하게 만든다. sigaction()으로 시그널 핸들러를 설정하면 SA_RESTART 플래그를 설정할 수 있는데 이것이 방해받은(Interrupted) 시스템 콜이 재개되게 해 줄 것이다.

태생적으로 이런 방식이 늘 작동하는 것은 아니다.(역자 주 : 시스템콜/인터럽션과 관련된 처리는 시스템마다 다를 수 있다.)

내가 선호하는 해결책은 goto문을 사용한다. 물론 이것이 교수님들을 아주 짜증나게 할 수 있지만 쓸만하다.

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
   if (errno == EINTR) {
      // 어떤 시그널이 우리에게 인터럽트를 걸었다. 그러니 재시작하자.
      goto select_restart;
   }
   // 진짜 오류는 여기서 처리하자
   perror("select");
}
```

물론 여기에서 goto를 쓸 필요는 없다. 처리를 위해 다른 구조를 쓸 수 있다. 그러나 필자는 goto문이 사실 더 깔끔하다고 생각한다. (역자 주 : 그래도 역자는 goto문을 쓰는 일을 추천하지 않는다.)

recv()에 대한 호출에 시간제한을 적용하려면 어떻게 해야하는가?

select()를 사용하라! 그것이 읽어들이려는 소켓 설명자에 시간제한 매개변수를 지정하라 수 있게 해준다. 아니면 모든 기능을 아래와 같이 하나의 함수에 감쌀 수 있다.

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

int recvtimeout(int s, char *buf, int len, int timeout)
{
   fd_set fds;
```

[2] http://www.unpbook.com/src.html

```
    int n;
    struct timeval tv;

    // 파일 설명자 집합 생성
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // 시간 제한을 위한 timeval 구조체 생성
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // 데이터를 받거나 시간이 초과될 때까지 기다린다
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error

    // 여기에 데이터가 있어야한다. 그러니 평범한 recv()를 한다
    return recv(s, buf, len, 0);
}
.
.
.
// recvtimeout()에 대한 호출 예시:
n = recvtimeout(s, buf, sizeof buf, 10); // 시간제한 10초

if (n == -1) {
    // 오류가 발생했다
    perror("recvtimeout");
}
else if (n == -2) {
    // 시간이 초과되었다
} else {
    // 버퍼에 데이터가 들어있다
}
.
.
.
```

recvtimeout()이 시간초과 상황에서 -2를 돌려준다는 점에 주목하라. 왜 0이 아닌지 궁금한가? recv()가 원격지 연결이 닫혔을 때 0을 돌려준다는 점을 떠올려보자. 그러니 그 값은 쓸 수가 없고, -1은 "오류"를 의미한다. 그래서 필자는 시간초과를 가리키는 값으로 -2를 썼다.

소켓에 데이터를 보내기 전에 암호화하거나 압축하려면 어떻게 해야하는가?

데이터를 암호화 하기 위한 간편한 방법 중 하나는 SSL (secure sockets layer)를 사용하는 것이다. 그러나 그것은 이 안내서의 범위를 벗어난다. (더 많은 정보가 필요하면 OpenSSL project[3] 를 확인하라.)

그러나 만약 여러분이 자신만의 압축기나 암호화 체계를 만들거나 써 보고 싶다면, 여러분의 데이터가 양 끝 사이에서 정해진 단계를 거친다는 점을 생각해보라. 각 단계는 데이터를 특정한 방법으로 바꾼다.

1. 서버가 데이터를 파일(또는 다른 것)에서 읽어들인다
2. 서버가 데이터를 암호화/압축한다(여러분이 이 단계를 추가한다)
3. 서버가 암호화된 데이터를 send()한다

반대편은 이렇다.

1. 클라이언트가 암호화된 데이터를 recv()한다

[3] https://www.openssl.org/

2. 클라이언트가 데이터를 복호화/압축해제한다(여러분이 이 단계를 추가한다)
3. 클라이언트가 데이터를 파일(또는 다른 것)에 쓴다

만약 압축과 암호화를 모두 할 생각이라면 압축을 먼저 해야한다는 점을 기억하라. :-) (역자 주 : 최소한 순서는 일관되어야 한다.)

클라이언트가 서버가 했던 작업을 제대로 거꾸로 수행하기만 한다면 여러분이 얼마나 많은 단계를 추가하던 데이터는 무사히 도착할 것이다.

그러므로 여러분이 필자의 코드를 쓰기 위해서 할 일은 데이터를 네트워크에서 읽고 보내는 코드의 중간 지점을 찾아내서 암호화를 하는 코드를 끼워넣는 것이다.

"PF_INET"이 계속 등장하는데 무엇인가? AF_INET와 관계가 있는가?

그렇다. 관계가 있다. 자세한 사항은 socket()에 대한 절을 참고하라.

클라이언트에게서 셸 커맨드를 받아서 실행하는 서버는 어떻게 만드는가?

단순함을 위해서 클라이언트가 connect()와 send() 후에 연결을 close() 처리한다고 가정하자. (즉 클라이언트가 연결을 닫지 않고 후속 시스템 콜을 보내는 일은 없다는 의미이다.)

클라이언트의 과정은 이렇다.

1. 서버에 connect()
2. send("/sbin/ls > /tmp/client.out")
3. 연결에 대한 close() 처리

한편 서버는 데이터를 받아서 실행한다.

1. 클라이언트의 연결 요청에 대한 accept()
2. 명령 문자열에 대한 recv(str)
3. 연결에 대한 close()
4. 명령을 실행하기 위해서 system(str)

주의하라! 클라이언트가 말하는 것을 서버가 실행한다는 것은 원격 셸 접근 권한을 주는 것과 비슷한 일이고 그들이 서버에 접속할 때 여러분의 계정으로 무엇인가 할 수 있다는 의미이다. 위의 예제에서 클라이언트가 "rm -rf ~"를 보내면 어떻게 되겠는가? 여러분의 계정이 가진 모든 것을 삭제할 것이다!

그러니 지혜롭게 여러분이 안전하다고 확신하는 몇 개의 유틸리티, 예를 들어 foobar 외의 것을 클라이언트가 실행하지 못하도록 하는 것이 좋다.

```
if (!strncmp(str, "foobar", 6)) {
   sprintf(sysstr, "%s > /tmp/server.out", str);
   system(sysstr);
}
```

그러나 불행히도 이것만으로는 여전히 위험하다. 클라이언트가 "foobar; rm -rf ~" 를 입력한다면 어떻게 되겠는가? 가장 안전한 방식은 명령의 매개변수에 들어가는 숫자나 영문자가 아닌 모든 문자(필요하다면 공백 문자도) 앞에 탈출 ("\") 문자를 붙이는 것이다.

보다시피 보안은 클라이언트가 보낸 것을 서버가 실행할 때에 큰 문제가 된다.

나는 꽤 큰 데이터를 보내는데 recv()를 해보면 한번에 536바이트나 1460바이트 씩만 받아온다. 그러나 이것을 로컬 장치에서 실행하면 한 번에 모든 데이터를 받아온다. 왜 이런 것인가?

MTU에 도달한 것이다. 이것은 물리계층이 전송가능한 최대 크기이다. 로컬 장치에서는 루프백 장치를 쓰기에 8K나 그 이상의 크기도 문제없이 다룰 수 있다. 그러나 이더넷에서는 헤더를 포함해 1500바이트가 한계이다. 모뎀을 쓴다면 (마찬가지로 헤더를 포함해)576바이트가 한계이다.

일단 모든 데이터가 전송되었음을 확실히 해야한다. (자세한 정보는sendall() 함수의 구현을 확인하라.) 전송이 잘 되었음이 확실하다면 모든 데이터를 읽어들일 때까지 recv()를 반복문 내부에서 호출해야 한다.

여러 번의 recv()호출을 통해 완전한 패킷을 수신하는 작업에 대해 자세한 정보가 필요하다면 망할 데이터 캡슐화 절을 참고하라.

나는 윈도우 장치를 써서 fork()시스템 호출이 없고 struct sigaction같은 것도 없다. 어떻게 해야하는가?

이것이 있다면 그것은 컴파일러와 함께 있는 POSIX 라이브러리에 있을 것이다. 필자는 윈도우 장치를 가지고 있지 않으므로 그에 대해 정확한 답을 줄 수 없다. 그러나 기억하기로는 마이크로소프트가 POSIX 호환성 계층을 만들었고 fork()도 거기에 있을 것이다. (어쩌면

sigaction도 있을 것이다.) (역자 주 : 그러나 윈도우에서는 윈도우의 처리법을 사용하는 것이 의도한 결과를 정확히 만드는 더 나은 방법이 될 것입니다.)

VC++에 딸려오는 도움말에서 "fork"나 "POSIX"를 검색하고 도움이 될만한 것이 있는지 살펴보라. (역자 주 : VC++ 자체도 Visual Studio가 가진 기능 중 일부의 오래된 이름에 불과합니다. 이 글이 최초에 작성된 것은 90년대임을 기억하십시오.)

그것이 전혀 작동하지 않는다면, fork()/sigaction과 관련된 것들을 떼어내고 그것의 Win32 대응인 CreateProcess()로 교체하라. 필자는 CreateProcess()를 어떻게 쓰는지는 모른다. 그것은 수억개의 인수를 받지만 아마도 VC++과 같이 오는 문서에 설명이 있을 것이다.

나는 방화벽 뒤에 있다. 방화벽 너머의 사람들이 나의 IP 주소를 알고 나의 장치에 접근하게 하려면 어떻게 해야하는가?

불행히도 방화벽의 목적은 방화벽 바깥의 사람들이 방화벽 안의 장치에 접근하는 것을 막는 것이다. 그러므로 그것을 허용하는 것은 보안에 헛점을 만든다.

그러나 모든 것이 안 된다고 말하려고 이 이야기를 꺼낸 것은 아니다. 방화벽이 마스커레이딩이나 NAT처리같은 것을 한다면 여전히 connect()로 방화벽 너머에 접근할 수 있다. 여러분의 프로그램이 언제나 연결을 게시하는 쪽이 되도록 한다면 문제는 없을 것이다.

만약 그것으로는 충분하지 않다면, 시스템 관리자에게 부탁해서 방화벽에 구멍을 내서 여러분에게 연결할 수 있도록 해야한다. 방화벽은 그것의 NAT프로그램이나 프록시 등을 써서 여러분에게 연결을 전달(Forward) 해줄 수 있다.

방화벽의 구멍은 가볍게 볼 것이 아니라는 점을 기억하라. 나쁜 사람들에게 내부 네트워크에 대한 접근 권한을 주지 않도록 해야한다. 초보자라면 소프트웨어를 안전하게 만드는 것이 생각보다 어렵다는 것을 알아야한다.

여러분의 시스템 관리자가 필자를 탓하는 일이 없게 해달라. ;-)

패킷 스니퍼는 어떻게 작성하는가? 어떻게 하면 내 이더넷 인터페이스를 무차별 모드로 설정할 수 있는가?

모르는 이들을 위해 설명하자면, 네트워크 카드가 "무차별 모드(promiscuous mode)" 일 때 목적지 주소가 실행중인 장치가 아닌 패킷까지 전부 운영체제에 전달한다. (우리는 IP 주소가 아닌 이더넷 계층 주소에 대해서 이야기하는 것이다. 그러나 이더넷은 IP보다 낮은 계층이므로, 사실상 모든 IP주소에 대한 통신이 전달된다. 더 자세한 내용은 저수준 넌센스와 네트워크 이론을 참고하라.)

이것이 패킷 스니퍼 동작의 기본 원리이다. 패킷 스니퍼는 인터페이스를 무차별 모드로 만들고, 운영체제는 그 장치를 통해 전달되는 모든 패킷을 받게 된다. 여러분은 이런 데이터를 읽을 수 있는 몇 가지 종류의 소켓을 쓸 수 있다.

불행히도 질문에 대한 답은 플랫폼에 따라 다르다. 그러나 인터넷을 찾아보면, 예를 들어 "windows promiscuous ioctl"을 검색한다면 도움이 되는 정볼르 얻을 수 있을것이다. 리눅스를 위해서는 useful Stack Overflow thread[4] 같은 정보가 있다.

어떻게 하면 TCP나 UDP소켓에 대해서 사용자 정의한 제한시간 값을 사용할 수 있는가?

시스템에 따라 다르다. 여러분의 시스템이 어떤 기능을 지원하는지 알아내기 위해서 (그리고 그것을 setsockopt()에 쓰기 위해서) SO_RCVTIMEO나 SO_SNDTIMEO 같은 것을 인터넷에서 찾아봐야 할 것이다.

리눅스의 맨페이지는 alarm()나 setitimer()를 대체재로 쓸 것을 권한다.

어떤 포트가 사용 가능한 상태인지는 어떻게 알아내는가? "공식적인" 포트 번호 목록같은 것이 있는가?

보통 이것은 문제가 되지 않는다. 여러분이 웹 서버를 작성한다고 하면, 80번같이 잘 알려진 포트를 쓰는 것이 좋다. 여러분만의 특별한 목적의 서버를 작성한다면 무작위의 포트 번호(그러나 1023보다 큰 것으로)를 고르고 시도해보라.

만약 포트가 이미 사용중이라면 bind()를 시도할 때 "Address already in use" 오류가 발생할 것이다. 다른 포트를 고르라. (여러분의 소프트웨어의 사용자가 설정 파일이나 명령줄 스위치로 대체 포트를 지정할 수 있게 하는 것이 좋다.)

인터넷 할당 번호 관리 기관(the Internet Assigned Numbers Authority, IANA) 이 관리하는 공식 포트 번호[5] 가 있다. (1023보다 큰) 어떤 번호가 저 목록에 없다고 해서 그 포트를 쓸 수 없는 것은 아니다. 예를 들어 Id Software의 DOOM은 "mdqs"(이것이 무엇이든) 와 같은 포트를 쓴다. 중요한 것은 같은 장치 의 누구도 당신이 그 포트를 쓰고 싶을 때 그 포트를 쓰고 있지 않으면 된다는 것이다.

[4] https://stackoverflow.com/questions/21323023/
[5] https://www.iana.org/assignments/port-numbers

Chapter 9

Man Pages

유닉스의 세상에는 많은 설명서들이 있다. 거기에는 여러분이 쓸 수 있는 개별 함수에 대한 짧은 절이 있다.

물론 manual은 타자로 치기에 너무 길 것이다. 그 말인즉 필자를 포함해 유닉스 세상의 누구도 그 만큼을 치고 싶어하지 않는다. 물론 필자는 본인이 얼마나 간결한 것을 선호하는지에 대해서 이렇게 저렇게 길게 늘여서 말할 수 있을 것이다. 그러나 필자는 본인이 얼마나 완전히 놀라우리만치 사실상 총체적인 모든 상황에서 간결하기를 선호하는지에 대해서 긴 웅변을 늘어놓는 대신 짧게 말하도록 하겠다.

[박수 소리]

이 글을 읽어주는 독자들에게 감사한다. 필자가 말하는 바는 이런 페이지들이 유닉스 세상에서 "man page"라고 불린다는 것이다. 그리고 독자 여러분의 읽는 재미를 위해서 필자의 개인적인 축약판을 여기에 포함해두었다. 그 말인 즉 이 함수들 대개는 필자의 사용법보다 훨씬 더 범용적이지만 필자는 인터넷 소켓 프로그래밍에 연관된 부분만을 제시할 것이라는 의미이다.

그러나 잠깐! 필자의 man page에서 잘못된 부분은 이것만이 아니다.

- 그것은 불완전하고 안내서의 기본적인 부분만을 보여준다.
- 현실 세계에는 여기에 있는 것보다 훨씬 많은 man page들이 있다.
- 여러분의 시스템에 있는 것과 다를 수 있다.
- 여러분의 시스템에 있는 특정 함수에 대해서는 헤더파일이 다를 수 있다.
- 여러분의 시스템에 있는 특정 함수에 대해서는 함수 매개변수가 다를 수 있다.

진짜 정보를 원한다면 man whatever를 입력해서 여러분의 로컬 유닉스 맨 페이지를 참고하라. "whatever"는 예를 들자면 "accept"처럼 여러분이 큰 관심을 갖는 사안이다. (마이크로포스트 비주얼 스튜디오도 그들의 도움말 부분에 이것과 유사한 것을 가지고 있다고 생각한다. 그러나 "man"이 "help"보다 1바이트 더 간결하므로 더 좋다. 이번에도 유닉스가 이겼다!)

그래서 이 정보들에 흠이 있다면 애초에 안내서에 첨부한 이유는 무엇인가? 말하자면 몇 가지 이유가 있다. 그러나 그 중 가장 좋은 이유는 바로 (a) 이 버전들은 네트워크 프로그래밍을 위해 특별히 재구성되었고 원본보다 이해하기 쉬우며 (b) 이 버전들에는 예제가 있다!

예제 이야기가 나왔으니 더 이야기하자면 코드의 길이가 너무 길어지는 점을 염려해서 오류 검사 코드를 모두 넣지는 않았다. 그러나 여러분은 여러분의 시스템 콜이 100% 성공할 것이라고 가정하지 않는 이상 오류 확인을 언제나 해야한다. 그리고 사실 그런 확신이 있어도 검사를 하는 것이 좋다!

9.1 accept()

Accept an incoming connection on a listening socket

Synopsis

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

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

Description

Once you've gone through the trouble of getting a SOCK_STREAM socket and setting it up for incoming connections with listen(), then you call accept() to actually get yourself a new socket descriptor to use for subsequent communication with the newly connected client.

The old socket that you are using for listening is still there, and will be used for further accept() calls as they come in.

Parameter	Description
s	The listen()ing socket descriptor.
addr	This is filled in with the address of the site that's connecting to you.
addrlen	This is filled in with the sizeof() the structure returned in the addr parameter. You can safely ignore it if you assume you're getting a struct sockaddr_in back, which you know you are, because that's the type you passed in for addr.

accept() will normally block, and you can use select() to peek on the listening socket descriptor ahead of time to see if it's "ready to read". If so, then there's a new connection waiting to be accept()ed! Yay! Alternatively, you could set the O_NONBLOCK flag on the listening socket using fcntl(), and then it will never block, choosing instead to return -1 with errno set to EWOULDBLOCK.

The socket descriptor returned by accept() is a bona fide socket descriptor, open and connected to the remote host. You have to close() it when you're done with it.

Return Value

accept() returns the newly connected socket descriptor, or -1 on error, with errno set appropriately.

Example

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, MYPORT, &hints, &res);
```

```
// make a socket, bind it, and listen on it:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);

// now accept an incoming connection:

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

// ready to communicate on socket descriptor new_fd!
```

See Also

socket(), getaddrinfo(), listen(), struct sockaddr_in

9.2 bind()

Associate a socket with an IP address and port number

Synopsis

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

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

Description

When a remote machine wants to connect to your server program, it needs two pieces of information: the IP address and the port number. The bind() call allows you to do just that.

First, you call getaddrinfo() to load up a struct sockaddr with the destination address and port information. Then you call socket() to get a socket descriptor, and then you pass the socket and address into bind(), and the IP address and port are magically (using actual magic) bound to the socket!

If you don't know your IP address, or you know you only have one IP address on the machine, or you don't care which of the machine's IP addresses is used, you can simply pass the AI_PASSIVE flag in the hints parameter to getaddrinfo(). What this does is fill in the IP address part of the struct sockaddr with a special value that tells bind() that it should automatically fill in this host's IP address.

What what? What special value is loaded into the struct sockaddr's IP address to cause it to auto-fill the address with the current host? I'll tell you, but keep in mind this is only if you're filling out the struct sockaddr by hand; if not, use the results from getaddrinfo(), as per above. In IPv4, the sin_addr.s_addr field of the struct sockaddr_in structure is set to INADDR_ANY. In IPv6, the sin6_addr field of the struct sockaddr_in6 structure is assigned into from the global variable in6addr_any. Or, if you're declaring a new struct in6_addr, you can initialize it to IN6ADDR_ANY_INIT.

Lastly, the addrlen parameter should be set to sizeof my_addr.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
// modern way of doing things with getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:
// (you should actually walk the "res" linked list and error-check!)

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

```
// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
// example of packing a struct by hand, IPv4

struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// you can specify an IP address:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// or you can let it automatically select one:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

See Also

getaddrinfo(), socket(), struct sockaddr_in, struct in_addr

9.3 connect()

Connect a socket to a server

Synopsis

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

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

Description

Once you've built a socket descriptor with the socket() call, you can connect() that socket to a remote server using the well-named connect() system call. All you need to do is pass it the socket descriptor and the address of the server you're interested in getting to know better. (Oh, and the length of the address, which is commonly passed to functions like this.)

Usually this information comes along as the result of a call to getaddrinfo(), but you can fill out your own struct sockaddr if you want to.

If you haven't yet called bind() on the socket descriptor, it is automatically bound to your IP address and a random local port. This is usually just fine with you if you're not a server, since you really don't care what your local port is; you only care what the remote port is so you can put it in the serv_addr parameter. You can call bind() if you really want your client socket to be on a specific IP address and port, but this is pretty rare.

Once the socket is connect()ed, you're free to send() and recv() data on it to your heart's content.

Special note: if you connect() a SOCK_DGRAM UDP socket to a remote host, you can use send() and recv() as well as sendto() and recvfrom(). If you want.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
// connect to www.example.com port 80 (http)

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;

// we could put "80" instead on "http" on the next line:
getaddrinfo("www.example.com", "http", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect it to the address and port we passed in to getaddrinfo():
```

```
connect(sockfd, res->ai_addr, res->ai_addrlen);
```

See Also

socket(), bind()

9.4 close()

Close a socket descriptor

Synopsis

#include <unistd.h>

int close(int s);

Description

After you've finished using the socket for whatever demented scheme you have concocted and you don't want to send() or recv() or, indeed, do anything else at all with the socket, you can close() it, and it'll be freed up, never to be used again.

The remote side can tell if this happens one of two ways. One: if the remote side calls recv(), it will return 0. Two: if the remote side calls send(), it'll receive a signal SIGPIPE and send() will return -1 and errno will be set to EPIPE.

Windows users: the function you need to use is called closesocket(), not close(). If you try to use close() on a socket descriptor, it's possible Windows will get angry... And you wouldn't like it when it's angry.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// a whole lotta stuff...*BRRRONNNN!*
.
.
.
close(s);  // not much to it, really.
```

See Also

socket(), shutdown()

9.5 getaddrinfo(), freeaddrinfo(), gai_strerror()

Get information about a host name and/or service and load up a struct sockaddr with the result.

Synopsis

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

int getaddrinfo(const char *nodename, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
  int    ai_flags;        // AI_PASSIVE, AI_CANONNAME, ...
  int    ai_family;       // AF_xxx
  int    ai_socktype;     // SOCK_xxx
  int    ai_protocol;     // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP

  socklen_t  ai_addrlen;    // length of ai_addr
  char   *ai_canonname;     // canonical name for nodename
  struct sockaddr  *ai_addr; // binary address
  struct addrinfo  *ai_next; // next structure in linked list
};
```

Description

getaddrinfo() is an excellent function that will return information on a particular host name (such as its IP address) and load up a struct sockaddr for you, taking care of the gritty details (like if it's IPv4 or IPv6). It replaces the old functions gethostbyname() and getservbyname().The description, below, contains a lot of information that might be a little daunting, but actual usage is pretty simple. It might be worth it to check out the examples first.

The host name that you're interested in goes in the nodename parameter. The address can be either a host name, like "www.example.com", or an IPv4 or IPv6 address (passed as a string). This parameter can also be NULL if you're using the AI_PASSIVE flag (see below).

The servname parameter is basically the port number. It can be a port number (passed as a string, like "80"), or it can be a service name, like "http" or "tftp" or "smtp" or "pop", etc. Well-known service names can be found in the IANA Port List[1] or in your /etc/services file.

Lastly, for input parameters, we have hints. This is really where you get to define what the getaddrinfo() function is going to do. Zero the whole structure before use with memset(). Let's take a look at the fields you need to set up before use.

The ai_flags can be set to a variety of things, but here are a couple important ones. (Multiple flags can be specified by bitwise-ORing them together with the | operator). Check your man page for the complete list of flags.

AI_CANONNAME causes the ai_canonname of the result to the filled out with the host's canonical (real) name. AI_PASSIVE causes the result's IP address to be filled out with INADDR_ANY (IPv4) or in6addr_any (IPv6); this causes a subsequent call to bind() to auto-fill the IP address of the struct sockaddr with the address of the current host. That's excellent for setting up a server when you don't want to hardcode the address.

If you do use the AI_PASSIVE, flag, then you can pass NULL in the nodename (since bind() will fill it in for you later).

[1] https://www.iana.org/assignments/port-numbers

Continuing on with the input parameters, you'll likely want to set ai_family to AF_UNSPEC which tells getaddrinfo() to look for both IPv4 and IPv6 addresses. You can also restrict yourself to one or the other with AF_INET or AF_INET6.

Next, the socktype field should be set to SOCK_STREAM or SOCK_DGRAM, depending on which type of socket you want.

Finally, just leave ai_protocol at 0 to automatically choose your protocol type.

Now, after you get all that stuff in there, you can finally make the call to getaddrinfo()!

Of course, this is where the fun begins. The res will now point to a linked list of struct addrinfos, and you can go through this list to get all the addresses that match what you passed in with the hints.

Now, it's possible to get some addresses that don't work for one reason or another, so what the Linux man page does is loops through the list doing a call to socket() and connect() (or bind() if you're setting up a server with the AI_PASSIVE flag) until it succeeds.

Finally, when you're done with the linked list, you need to call freeaddrinfo() to free up the memory (or it will be leaked, and Some People will get upset).

Return Value

Returns zero on success, or nonzero on error. If it returns nonzero, you can use the function gai_strerror() to get a printable version of the error code in the return value.

Example

```
// code for a client connecting to a server
// namely a stream socket to www.example.com on port 80 (http)
// either IPv4 or IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
  exit(1);
}

// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
  if ((sockfd = socket(p->ai_family, p->ai_socktype,
       p->ai_protocol)) == -1) {
    perror("socket");
    continue;
  }

  if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
    perror("connect");
    close(sockfd);
    continue;
  }
```

```
    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no connection
    fprintf(stderr, "failed to connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

```
// code for a server waiting for connections
// namely a stream socket on port 3490, on this host's IP
// either IPv4 or IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use my IP address

if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("bind");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no successful bind
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

See Also

gethostbyname(), getnameinfo()

9.6 gethostname()

Returns the name of the system

Synopsis

#include <sys/unistd.h>

int gethostname(char *name, size_t len);

Description

Your system has a name. They all do. This is a slightly more Unixy thing than the rest of the networky stuff we've been talking about, but it still has its uses.

For instance, you can get your host name, and then call gethostbyname() to find out your IP address.

The parameter name should point to a buffer that will hold the host name, and len is the size of that buffer in bytes. gethostname() won't overwrite the end of the buffer (it might return an error, or it might just stop writing), and it will NUL-terminate the string if there's room for it in the buffer.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

See Also

gethostbyname()

9.7 gethostbyname(), gethostbyaddr()

Get an IP address for a hostname, or vice-versa

Synopsis

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

struct hostent *gethostbyname(const char *name); // DEPRECATED!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Description

PLEASE NOTE: these two functions are superseded by getaddrinfo() and getnameinfo()! In particular, gethostbyname() doesn't work well with IPv6.

These functions map back and forth between host names and IP addresses. For instance, if you have "www.example.com", you can use gethostbyname() to get its IP address and store it in a struct in_addr.

Conversely, if you have a struct in_addr or a struct in6_addr, you can use gethostbyaddr() to get the hostname back. gethostbyaddr() is IPv6 compatible, but you should use the newer shinier getnameinfo() instead.

(If you have a string containing an IP address in dots-and-numbers format that you want to look up the hostname of, you'd be better off using getaddrinfo() with the AI_CANONNAME flag.)

gethostbyname() takes a string like "www.yahoo.com", and returns a struct hostent which contains tons of information, including the IP address. (Other information is the official host name, a list of aliases, the address type, the length of the addresses, and the list of addresses—it's a general-purpose structure that's pretty easy to use for our specific purposes once you see how.)

gethostbyaddr() takes a struct in_addr or struct in6_addr and brings you up a corresponding host name (if there is one), so it's sort of the reverse of gethostbyname(). As for parameters, even though addr is a char*, you actually want to pass in a pointer to a struct in_addr. len should be sizeof(struct in_addr), and type should be AF_INET.

So what is this struct hostent that gets returned? It has a number of fields that contain information about the host in question.

Field	Description
char *h_name	The real canonical host name.
char **h_aliases	A list of aliases that can be accessed with arrays—the last element is NULL
int h_addrtype	The result's address type, which really should be AF_INET for our purposes.
int length	The length of the addresses in bytes, which is 4 for IP (version 4) addresses.
char **h_addr_list	A list of IP addresses for this host. Although this is a char**, it's really an array of struct in_addr*s in disguise. The last array element is NULL.
h_addr	A commonly defined alias for h_addr_list[0]. If you just want any old IP address for this host (yeah, they can have more than one) just use this field.

Return Value

Returns a pointer to a resultant struct hostent on success, or NULL on error.

Instead of the normal perror() and all that stuff you'd normally use for error reporting, these functions have parallel results in the variable h_errno, which can be printed using the functions herror() or hstrerror(). These work just like the classic

errno, perror(), and strerror() functions you're used to.

Example

```
// THIS IS A DEPRECATED METHOD OF GETTING HOST NAMES
// use getaddrinfo() instead!

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int i;
    struct hostent *he;
    struct in_addr **addr_list;

    if (argc != 2) {
        fprintf(stderr,"usage: ghbn hostname\n");
        return 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) {  // get the host info
        herror("gethostbyname");
        return 2;
    }

    // print information about this host:
    printf("Official name is: %s\n", he->h_name);
    printf("    IP addresses: ");
    addr_list = (struct in_addr **)he->h_addr_list;
    for(i = 0; addr_list[i] != NULL; i++) {
        printf("%s ", inet_ntoa(*addr_list[i]));
    }
    printf("\n");

    return 0;
}
```

```
// THIS HAS BEEN SUPERSEDED
// use getnameinfo() instead!

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
```

```
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
printf("Host name: %s\n", he->h_name);
```

See Also

getaddrinfo(), getnameinfo(), gethostname(), errno, perror(), strerror(), struct in_addr

9.8 getnameinfo()

Look up the host name and service name information for a given struct sockaddr.

Synopsis

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

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
        char *host, size_t hostlen,
        char *serv, size_t servlen, int flags);
```

Description

This function is the opposite of getaddrinfo(), that is, this function takes an already loaded struct sockaddr and does a name and service name lookup on it. It replaces the old gethostbyaddr() and getservbyport() functions.

You have to pass in a pointer to a struct sockaddr (which in actuality is probably a struct sockaddr_in or struct sockaddr_in6 that you've cast) in the sa parameter, and the length of that struct in the salen.

The resultant host name and service name will be written to the area pointed to by the host and serv parameters. Of course, you have to specify the max lengths of these buffers in hostlen and servlen.

Finally, there are several flags you can pass, but here a a couple good ones. NI_NOFQDN will cause the host to only contain the host name, not the whole domain name. NI_NAMEREQD will cause the function to fail if the name cannot be found with a DNS lookup (if you don't specify this flag and the name can't be found, getnameinfo() will put a string version of the IP address in host instead).

As always, check your local man pages for the full scoop.

Return Value

Returns zero on success, or non-zero on error. If the return value is non-zero, it can be passed to gai_strerror() to get a human-readable string. See getaddrinfo for more information.

Example

```
struct sockaddr_in6 sa; // could be IPv4 if you want
char host[1024];
char service[20];

// pretend sa is full of good information about the host and port...

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf("   host: %s\n", host);    // e.g. "www.example.com"
printf("service: %s\n", service); // e.g. "http"
```

See Also

getaddrinfo(), gethostbyaddr()

9.9 getpeername()

Return address info about the remote side of the connection

Synopsis

#include <sys/socket.h>

int getpeername(int s, struct sockaddr *addr, socklen_t *len);

Description

Once you have either accept()ed a remote connection, or connect()ed to a server, you now have what is known as a peer. Your peer is simply the computer you're connected to, identified by an IP address and a port. So...

getpeername() simply returns a struct sockaddr_in filled with information about the machine you're connected to.

Why is it called a "name"? Well, there are a lot of different kinds of sockets, not just Internet Sockets like we're using in this guide, and so "name" was a nice generic term that covered all cases. In our case, though, the peer's "name" is it's IP address and port.

Although the function returns the size of the resultant address in len, you must preload len with the size of addr.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
// assume s is a connected socket

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// deal with both IPv4 and IPv6:
if (addr.ss_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)&addr;
    port = ntohs(s->sin_port);
    inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr);
} else { // AF_INET6
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
    port = ntohs(s->sin6_port);
    inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

See Also

gethostname(), gethostbyname(), gethostbyaddr()

9.10 errno

Holds the error code for the last system call

Synopsis

```
#include <errno.h>

int errno;
```

Description

This is the variable that holds error information for a lot of system calls. If you'll recall, things like socket() and listen() return -1 on error, and they set the exact value of errno to let you know specifically which error occurred.

The header file errno.h lists a bunch of constant symbolic names for errors, such as EADDRINUSE, EPIPE, ECONNREFUSED, etc. Your local man pages will tell you what codes can be returned as an error, and you can use these at run time to handle different errors in different ways.

Or, more commonly, you can call perror() or strerror() to get a human-readable version of the error.

One thing to note, for you multithreading enthusiasts, is that on most systems errno is defined in a threadsafe manner. (That is, it's not actually a global variable, but it behaves just like a global variable would in a single-threaded environment.)

Return Value

The value of the variable is the latest error to have transpired, which might be the code for "success" if the last action succeeded.

Example

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
   perror("socket"); // or use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
   // an error has occurred!!

   // if we were only interrupted, just restart the select() call:
   if (errno == EINTR) goto tryagain;  // AAAA! goto!!!

   // otherwise it's a more serious error:
   perror("select");
   exit(1);
}
```

See Also

perror(), strerror()

9.11 fcntl()

Control socket descriptors

Synopsis

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

Description

This function is typically used to do file locking and other file-oriented stuff, but it also has a couple socket-related functions that you might see or use from time to time.

Parameter s is the socket descriptor you wish to operate on, cmd should be set to F_SETFL, and arg can be one of the following commands. (Like I said, there's more to fcntl() than I'm letting on here, but I'm trying to stay socket-oriented.)

cmd	Description
O_NONBLOCK	Set the socket to be non-blocking. See the section on blocking for more details.
O_ASYNC	Set the socket to do asynchronous I/O. When data is ready to be recv()'d on the socket, the signal SIGIO will be raised. This is rare to see, and beyond the scope of the guide. And I think it's only available on certain systems.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Different uses of the fcntl() system call actually have different return values, but I haven't covered them here because they're not socket-related. See your local fcntl() man page for more information.

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // set to non-blocking
fcntl(s, F_SETFL, O_ASYNC);     // set to asynchronous I/O
```

See Also

Blocking, send()

9.12 htons(), htonl(), ntohs(), ntohl()

Convert multi-byte integer types from host byte order to network byte order

Synopsis

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Description

Just to make you really unhappy, different computers use different byte orderings internally for their multibyte integers (i.e. any integer that's larger than a char). The upshot of this is that if you send() a two-byte short int from an Intel box to a Mac (before they became Intel boxes, too, I mean), what one computer thinks is the number 1, the other will think is the number 256, and vice-versa.

The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to "big-endian" before sending them out. Since Intel is a "little-endian" machine, it's far more politically correct to call our preferred byte ordering "Network Byte Order". So these functions convert from your native byte order to network byte order and back again.

(This means on Intel these functions swap all the bytes around, and on PowerPC they do nothing because the bytes are already in Network Byte Order. But you should always use them in your code anyway, since someone might want to build it on an Intel machine and still have things work properly.)

Note that the types involved are 32-bit (4 byte, probably int) and 16-bit (2 byte, very likely short) numbers. 64-bit machines might have a htonll() for 64-bit ints, but I've not seen it. You'll just have to write your own.

Anyway, the way these functions work is that you first decide if you're converting from host (your machine's) byte order or from network byte order. If "host", the the first letter of the function you're going to call is "h". Otherwise it's "n" for "network". The middle of the function name is always "to" because you're converting from one "to" another, and the penultimate letter shows what you're converting to. The last letter is the size of the data, "s" for short, or "l" for long. Thus:

Function	Description
htons()	host to network short
htonl()	host to network long
ntohs()	network to host short
ntohl()	network to host long

Return Value

Each function returns the converted value.

Example

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// convert and send
```

```
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // this expression is true
```

9.13 inet_ntoa(), inet_aton(), inet_addr

Convert IP addresses from a dots-and-number string to a struct in_addr and back

Synopsis

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// ALL THESE ARE DEPRECATED! Use inet_pton()  or inet_ntop() instead!!

char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

Description

These functions are deprecated because they don't handle IPv6! Use (inet_ntop())[#inet_ntopman] or (inet_pton())[#inet_ntopman] instead! They are included here because they can still be found in the wild.

All of these functions convert from a struct in_addr (part of your struct sockaddr_in, most likely) to a string in dots-and-numbers format (e.g. "192.168.5.10") and vice-versa. If you have an IP address passed on the command line or something, this is the easiest way to get a struct in_addr to connect() to, or whatever. If you need more power, try some of the DNS functions like gethostbyname() or attempt a coup d'État in your local country.

The function inet_ntoa() converts a network address in a struct in_addr to a dots-and-numbers format string. The "n" in "ntoa" stands for network, and the "a" stands for ASCII for historical reasons (so it's "Network To ASCII"—the "toa" suffix has an analogous friend in the C library called atoi() which converts an ASCII string to an integer).

The function inet_aton() is the opposite, converting from a dots-and-numbers string into a in_addr_t (which is the type of the field s_addr in your struct in_addr).

Finally, the function inet_addr() is an older function that does basically the same thing as inet_aton(). It's theoretically deprecated, but you'll see it a lot and the police won't come get you if you use it.

Return Value

inet_aton() returns non-zero if the address is a valid one, and it returns zero if the address is invalid.

inet_ntoa() returns the dots-and-numbers string in a static buffer that is overwritten with each call to the function.

inet_addr() returns the address as an in_addr_t, or -1 if there's an error. (That is the same result as if you tried to convert the string "255.255.255.255", which is a valid IP address. This is why inet_aton() is better.)

Example

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope

some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"

// and this call is the same as the inet_aton() call, above:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

See Also

inet_ntop(), inet_pton(), gethostbyname(), gethostbyaddr()

9.14 inet_ntop(), inet_pton()

Convert IP addresses to human-readable form and back.

Synopsis

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

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

Description

These functions are for dealing with human-readable IP addresses and converting them to their binary representation for use with various functions and system calls. The "n" stands for "network", and "p" for "presentation". Or "text presentation". But you can think of it as "printable". "ntop" is "network to printable". See?

Sometimes you don't want to look at a pile of binary numbers when looking at an IP address. You want it in a nice printable form, like 192.0.2.180, or 2001:db8:8714:3a90::12. In that case, inet_ntop() is for you.

inet_ntop() takes the address family in the af parameter (either AF_INET or AF_INET6). The src parameter should be a pointer to either a struct in_addr or struct in6_addr containing the address you wish to convert to a string. Finally dst and size are the pointer to the destination string and the maximum length of that string.

What should the maximum length of the dst string be? What is the maximum length for IPv4 and IPv6 addresses? Fortunately there are a couple of macros to help you out. The maximum lengths are: INET_ADDRSTRLEN and INET6_ADDRSTRLEN.

Other times, you might have a string containing an IP address in readable form, and you want to pack it into a struct sockaddr_in or a struct sockaddr_in6. In that case, the opposite function inet_pton() is what you're after.

inet_pton() also takes an address family (either AF_INET or AF_INET6) in the af parameter. The src parameter is a pointer to a string containing the IP address in printable form. Lastly the dst parameter points to where the result should be stored, which is probably a struct in_addr or struct in6_addr.

These functions don't do DNS lookups—you'll need getaddrinfo() for that.

Return Value

inet_ntop() returns the dst parameter on success, or NULL on failure (and errno is set).

inet_pton() returns 1 on success. It returns -1 if there was an error (errno is set), or 0 if the input isn't a valid IP address.

Example

```
// IPv4 demo of inet_ntop() and inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);
```

```
printf("%s\n", str); // prints "192.0.2.33"

// IPv6 demo of inet_ntop() and inet_pton()
// (basically the same except with a bunch of 6s thrown around)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// now get it back and print it
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // prints "2001:db8:8714:3a90::12"

// Helper function you can use:

//Convert a struct sockaddr address to a string, IPv4 and IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr),
                    s, maxlen);
            break;

        case AF_INET6:
            inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
                    s, maxlen);
            break;

        default:
            strncpy(s, "Unknown AF", maxlen);
            return NULL;
    }

    return s;
}
```

See Also

getaddrinfo()

9.15 listen()

Tell a socket to listen for incoming connections

Synopsis

#include <sys/socket.h>

int listen(int s, int backlog);

Description

You can take your socket descriptor (made with the socket() system call) and tell it to listen for incoming connections. This is what differentiates the servers from the clients, guys.

The backlog parameter can mean a couple different things depending on the system you on, but loosely it is how many pending connections you can have before the kernel starts rejecting new ones. So as the new connections come in, you should be quick to accept() them so that the backlog doesn't fill. Try setting it to 10 or so, and if your clients start getting "Connection refused" under heavy load, set it higher.

Before calling listen(), your server should call bind() to attach itself to a specific port number. That port number (on the server's IP address) will be the one that clients connect to.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 10); // set s up to be a server (listening) socket

// then have an accept() loop down here somewhere
```

See Also

accept(), bind(), socket()

9.16 perror(), strerror()

Print an error as a human-readable string

Synopsis

```
#include <stdio.h>
#include <string.h>   // for strerror()

void perror(const char *s);
char *strerror(int errnum);
```

Description

Since so many functions return -1 on error and set the value of the variable errno to be some number, it would sure be nice if you could easily print that in a form that made sense to you.

Mercifully, perror() does that. If you want more description to be printed before the error, you can point the parameter s to it (or you can leave s as NULL and nothing additional will be printed).

In a nutshell, this function takes errno values, like ECONNRESET, and prints them nicely, like “Connection reset by peer.”

The function strerror() is very similar to perror(), except it returns a pointer to the error message string for a given value (you usually pass in the variable errno).

Return Value

strerror() returns a pointer to the error message string.

Example

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // some error has occurred
   // prints "socket error: " + the error message:
   perror("socket error");
}

// similarly:
if (listen(s, 10) == -1) {
   // this prints "an error: " + the error message from errno:
   printf("an error: %s\n", strerror(errno));
}
```

See Also

errno

9.17 poll()

Test for events on multiple sockets simultaneously

Synopsis

#include <sys/poll.h>

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);

Description

This function is very similar to select() in that they both watch sets of file descriptors for events, such as incoming data ready to recv(), socket ready to send() data to, out-of-band data ready to recv(), errors, etc.

The basic idea is that you pass an array of nfds struct pollfds in ufds, along with a timeout in milliseconds (1000 milliseconds in a second). The timeout can be negative if you want to wait forever. If no event happens on any of the socket descriptors by the timeout, poll() will return.

Each element in the array of struct pollfds represents one socket descriptor, and contains the following fields:

```
struct pollfd {
  int fd;       // the socket descriptor
  short events;   // bitmap of events we're interested in
  short revents;  // when poll() returns, bitmap of events that occurred
};
```

Before calling poll(), load fd with the socket descriptor (if you set fd to a negative number, this struct pollfd is ignored and its revents field is set to zero) and then construct the events field by bitwise-ORing the following macros:

Macro	Description
POLLIN	Alert me when data is ready to recv() on this socket.
POLLOUT	Alert me when I can send() data to this socket without blocking.
POLLPRI	Alert me when out-of-band data is ready to recv() on this socket.

Once the poll() call returns, the revents field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present:

Macro	Description
POLLERR	An error has occurred on this socket.
POLLHUP	The remote side of the connection hung up.
POLLNVAL	Something was wrong with the socket descriptor fd—maybe it's uninitialized?

Return Value

Returns the number of elements in the ufds array that have had event occur on them; this can be zero if the timeout occurred. Also returns -1 on error (and errno will be set accordingly).

Example

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];
```

```
s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// pretend we've connected both to a server at this point
//connect(s1, ...)...
//connect(s2, ...)...

// set up the array of file descriptors.
//
// in this example, we want to know when there's normal or out-of-band
// data ready to be recv()'d...

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band

ufds[1].fd = s2;
ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout
rv = poll(ufds, 2, 3500);

if (rv == -1) {
   perror("poll"); // error occurred in poll()
} else if (rv == 0) {
   printf("Timeout occurred! No data after 3.5 seconds.\n");
} else {
   // check for events on s1:
   if (ufds[0].revents & POLLIN) {
      recv(s1, buf1, sizeof buf1, 0); // receive normal data
   }
   if (ufds[0].revents & POLLPRI) {
      recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band data
   }

   // check for events on s2:
   if (ufds[1].revents & POLLIN) {
      recv(s1, buf2, sizeof buf2, 0);
   }
}
```

See Also

select()

9.18 recv(), recvfrom()

Receive data on a socket

Synopsis

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

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
        struct sockaddr *from, socklen_t *fromlen);
```

Description

Once you have a socket up and connected, you can read incoming data from the remote side using the recv() (for TCP SOCK_STREAM sockets) and recvfrom() (for UDP SOCK_DGRAM sockets).

Both functions take the socket descriptor s, a pointer to the buffer buf, the size (in bytes) of the buffer len, and a set of flags that control how the functions work.

Additionally, the recvfrom() takes a struct sockaddr*, from that will tell you where the data came from, and will fill in fromlen with the size of struct sockaddr. (You must also initialize fromlen to be the size of from or struct sockaddr.)

So what wondrous flags can you pass into this function? Here are some of them, but you should check your local man pages for more information and what is actually supported on your system. You bitwise-or these together, or just set flags to 0 if you want it to be a regular vanilla recv().

Macro	Description
MSG_OOB	Receive Out of Band data. This is how to get data that has been sent to you with the MSG_OOB flag in send(). As the receiving side, you will have had signal SIGURG raised telling you there is urgent data. In your handler for that signal, you could call recv() with this MSG_OOB flag.
MSG_PEEK	If you want to call recv() "just for pretend", you can call it with this flag. This will tell you what's waiting in the buffer for when you call recv() "for real" (i.e. without the MSG_PEEK flag. It's like a sneak preview into the next recv() call.
MSG_WAITALL	Tell recv() to not return until all the data you specified in the len parameter. It will ignore your wishes in extreme circumstances, however, like if a signal interrupts the call or if some error occurs or if the remote side closes the connection, etc. Don't be mad with it.

When you call recv(), it will block until there is some data to read. If you want to not block, set the socket to non-blocking or check with select() or poll() to see if there is incoming data before calling recv() or recvfrom().

Return Value

Returns the number of bytes actually received (which might be less than you requested in the len parameter), or -1 on error (and errno will be set accordingly).

If the remote side has closed the connection, recv() will return 0. This is the normal method for determining if the remote side has closed the connection. Normality is good, rebel!

Example

```
// stream sockets and recv()

struct addrinfo hints, *res;
```

```
int sockfd;
char buf[512];
int byte_count;

// get host info, make socket, and connect it
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// all right! now that we're connected, we can receive some data!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
```

```
// datagram sockets and recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// get host info, make socket, bind it to port 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, "4950", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

// no need to accept(), just recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
   inet_ntop(addr.ss_family,
      addr.ss_family == AF_INET?
         ((struct sockadd_in *)&addr)->sin_addr:
         ((struct sockadd_in6 *)&addr)->sin6_addr,
      ipstr, sizeof ipstr);
```

See Also

send(), sendto(), select(), poll(), Blocking

9.19 select()

Check if sockets descriptors are ready to read/write

Synopsis

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

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

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Description

The select() function gives you a way to simultaneously check multiple sockets to see if they have data waiting to be recv()d, or if you can send() data to them without blocking, or if some exception has occurred.

You populate your sets of socket descriptors using the macros, like FD_SET(), above. Once you have the set, you pass it into the function as one of the following parameters: readfds if you want to know when any of the sockets in the set is ready to recv() data, writefds if any of the sockets is ready to send() data to, and/or exceptfds if you need to know when an exception (error) occurs on any of the sockets. Any or all of these parameters can be NULL if you're not interested in those types of events. After select() returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions.

The first parameter, n is the highest-numbered socket descriptor (they're just ints, remember?) plus one.

Lastly, the struct timeval, timeout, at the end—this lets you tell select() how long to check these sets for. It'll return after the timeout, or when an event occurs, whichever is first. The struct timeval has two fields: tv_sec is the number of seconds, to which is added tv_usec, the number of microseconds (1,000,000 microseconds in a second).

The helper macros do the following:

Macro	Description
FD_SET(int fd, fd_set *set);	Add fd to the set.
FD_CLR(int fd, fd_set *set);	Remove fd from the set.
FD_ISSET(int fd, fd_set *set);	Return true if fd is in the set.
FD_ZERO(fd_set *set);	Clear all entries from the set.

Note for Linux users: Linux's select() can return "ready-to-read" and then not actually be ready to read, thus causing the subsequent read() call to block. You can work around this bug by setting O_NONBLOCK flag on the receiving socket so it errors with EWOULDBLOCK, then ignoring this error if it occurs. See the fcntl() man page for more info on setting a socket to non-blocking.

Return Value

Returns the number of descriptors in the set on success, 0 if the timeout was reached, or -1 on error (and errno will be set accordingly). Also, the sets are modified to show which sockets are ready.

Example

```
int s1, s2, n;
```

```
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// pretend we've connected both to a server at this point
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);

// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// since we got s2 second, it's the "greater", so we use that for
// the n param in select()
n = s2 + 1;

// wait until either socket has data ready to be recv()d (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
   perror("select"); // error occurred in select()
} else if (rv == 0) {
   printf("Timeout occurred! No data after 10.5 seconds.\n");
} else {
   // one or both of the descriptors have data
   if (FD_ISSET(s1, &readfds)) {
      recv(s1, buf1, sizeof buf1, 0);
   }
   if (FD_ISSET(s2, &readfds)) {
      recv(s2, buf2, sizeof buf2, 0);
   }
}
```

See Also

poll()

9.20 setsockopt(), getsockopt()

Set various options for a socket

Synopsis

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

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

Description

Sockets are fairly configurable beasts. In fact, they are so configurable, I'm not even going to cover it all here. It's probably system-dependent anyway. But I will talk about the basics.

Obviously, these functions get and set certain options on a socket. On a Linux box, all the socket information is in the man page for socket in section 7. (Type: "man 7 socket" to get all these goodies.)

As for parameters, s is the socket you're talking about, level should be set to SOL_SOCKET. Then you set the optname to the name you're interested in. Again, see your man page for all the options, but here are some of the most fun ones:

optname	Description
SO_BINDTODEVICE	Bind this socket to a symbolic device name like eth0 instead of using bind() to bind it to an IP address. Type the command ifconfig under Unix to see the device names.
SO_REUSEADDR	Allows other sockets to bind() to this port, unless there is an active listening socket bound to the port already. This enables you to get around those "Address already in use" error messages when you try to restart your server after a crash.
SOCK_DGRAM	Allows UDP datagram (SOCK_DGRAM) sockets to send and receive packets sent to and from the broadcast address. Does nothing—NOTHING!!—to TCP stream sockets! Hahaha!

As for the parameter optval, it's usually a pointer to an int indicating the value in question. For booleans, zero is false, and non-zero is true. And that's an absolute fact, unless it's different on your system. If there is no parameter to be passed, optval can be NULL.

The final parameter, optlen, should be set to the length of optval, probably sizeof(int), but varies depending on the option. Note that in the case of getsockopt(), this is a pointer to a socklen_t, and it specifies the maximum size object that will be stored in optval (to prevent buffer overflows). And getsockopt() will modify the value of optlen to reflect the number of bytes actually set.

Warning: on some systems (notably Sun and Windows), the option can be a char instead of an int, and is set to, for example, a character value of '1' instead of an int value of 1. Again, check your own man pages for more info with "man setsockopt" and "man 7 socket"!

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
int optval;
int optlen;
```

```
char *optval2;

// set SO_REUSEADDR on a socket to true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// bind a socket to a device name (might not work on all systems):
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);

// see if the SO_BROADCAST flag is set:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
   print("SO_BROADCAST enabled on s3!\n");
}
```

See Also

fcntl()

9.21 send(), sendto()

Send data out over a socket

Synopsis

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

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
        int flags, const struct sockaddr *to,
        socklen_t tolen);
```

Description

These functions send data to a socket. Generally speaking, send() is used for TCP SOCK_STREAM connected sockets, and sendto() is used for UDP SOCK_DGRAM unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of sendto() define where the packet is going.

With both send() and sendto(), the parameter s is the socket, buf is a pointer to the data you want to send, len is the number of bytes you want to send, and flags allows you to specify more information about how the data is to be sent. Set flags to zero if you want it to be "normal" data. Here are some of the commonly used flags, but check your local send() man pages for more details:

Macro	Description
MSG_OOB	Send as "out of band" data. TCP supports this, and it's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal SIGURG and it can then receive this data without first receiving all the rest of the normal data in the queue.
MSG_DONTROUTE	Don't send this data over a router, just keep it local.
MSG_DONTWAIT	If send() would block because outbound traffic is clogged, have it return EAGAIN. This is like a "enable non-blocking just for this send." See the section on blocking for more details.
MSG_NOSIGNAL	If you send() to a remote host which is no longer recv()ing, you'll typically get the signal SIGPIPE. Adding this flag prevents that signal from being raised.

Return Value

Returns the number of bytes actually sent, or -1 on error (and errno will be set accordingly). Note that the number of bytes actually sent might be less than the number you asked it to send! See the section on handling partial send()s for a helper function to get around this.

Also, if the socket has been closed by either side, the process calling send() will get the signal SIGPIPE. (Unless send() was called with the MSG_NOSIGNAL flag.)

Example

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;
```

```
// first with TCP stream sockets:

// assume sockets are made and connected
//stream_socket = socket(...
//connect(stream_socket, ...

// convert to network byte order
temp = htonl(spatula_count);
// send data normally:
send(stream_socket, &temp, sizeof temp, 0);

// send secret message out of band:
send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB);

// now with UDP datagram sockets:
//getaddrinfo(...
//dest = ... // assume "dest" holds the address of the destination
//dgram_socket = socket(...

// send secret message normally:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
    (struct sockaddr*)&dest, sizeof dest);
```

See Also

recv(), recvfrom()

9.22 shutdown()

Stop further sends and receives on a socket

Synopsis

#include <sys/socket.h>

int shutdown(int s, int how);

Description

That's it! I've had it! No more send()s are allowed on this socket, but I still want to recv() data on it! Or vice-versa! How can I do this?

When you close() a socket descriptor, it closes both sides of the socket for reading and writing, and frees the socket descriptor. If you just want to close one side or the other, you can use this shutdown() call.

As for parameters, s is obviously the socket you want to perform this action on, and what action that is can be specified with the how parameter. how can be SHUT_RD to prevent further recv()s, SHUT_WR to prohibit further send()s, or SHUT_RDWR to do both.

Note that shutdown() doesn't free up the socket descriptor, so you still have to eventually close() the socket even if it has been fully shut down.

This is a rarely used system call.

Return Value

Returns zero on success, or -1 on error (and errno will be set accordingly).

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...do some send()s and stuff in here...

// and now that we're done, don't allow any more sends()s:
shutdown(s, SHUT_WR);
```

See Also

close()

9.23 socket()

Allocate a socket descriptor

Synopsis

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

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

Description

Returns a new socket descriptor that you can use to do sockety things with. This is generally the first call in the whopping process of writing a socket program, and you can use the result for subsequent calls to listen(), bind(), accept(), or a variety of other functions.

In usual usage, you get the values for these parameters from a call to getaddrinfo(), as shown in the example below. But you can fill them in by hand if you really want to.

Parameter	Description
domain	domain describes what kind of socket you're interested in. This can, believe me, be a wide variety of things, but since this is a socket guide, it's going to be PF_INET for IPv4, and PF_INET6 for IPv6.
type	Also, the type parameter can be a number of things, but you'll probably be setting it to either SOCK_STREAM for reliable TCP sockets (send(), recv()) or SOCK_DGRAM for unreliable fast UDP sockets (sendto(), recvfrom()). (Another interesting socket type is SOCK_RAW which can be used to construct packets by hand. It's pretty cool.)
protocol	Finally, the protocol parameter tells which protocol to use with a certain socket type. Like I've already said, for instance, SOCK_STREAM uses TCP. Fortunately for you, when using SOCK_STREAM or SOCK_DGRAM, you can just set the protocol to 0, and it'll use the proper protocol automatically. Otherwise, you can use getprotobyname() to look up the proper protocol number.

Return Value

The new socket descriptor to be used in subsequent calls, or -1 on error (and errno will be set accordingly).

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;     // AF_INET, AF_INET6, or AF_UNSPEC
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM or SOCK_DGRAM

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket using the information gleaned from getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

See Also

accept(), bind(), getaddrinfo(), listen()

9.24 struct sockaddr and pals

Structures for handling internet addresses

Synopsis

```
#include <netinet/in.h>

// All pointers to socket address structures are often cast to pointers
// to this type before use in various functions and system calls:

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


// IPv4 AF_INET sockets:

struct sockaddr_in {
    short          sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char           sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;         // load with inet_pton()
};


// IPv6 AF_INET6 sockets:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // load with inet_pton()
};


// General socket address holding structure, big enough to hold either
// struct sockaddr_in or struct sockaddr_in6 data:

struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
```

```
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Description

These are the basic structures for all syscalls and functions that deal with internet addresses. Often you'll use getaddrinfo() to fill these structures out, and then will read them when you have to.

In memory, the struct sockaddr_in and struct sockaddr_in6 share the same beginning structure as struct sockaddr, and you can freely cast the pointer of one type to the other without any harm, except the possible end of the universe.

Just kidding on that end-of-the-universe thing...if the universe does end when you cast a struct sockaddr_in* to a struct sockaddr*, I promise you it's pure coincidence and you shouldn't even worry about it.

So, with that in mind, remember that whenever a function says it takes a struct sockaddr* you can cast your struct sockaddr_in*, struct sockaddr_in6*, or struct sockadd_storage* to that type with ease and safety.

struct sockaddr_in is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (AF_INET), a port in sin_port, and an IPv4 address in sin_addr.

There's also this sin_zero field in struct sockaddr_in which some people claim must be set to zero. Other people don't claim anything about it (the Linux documentation doesn't even mention it at all), and setting it to zero doesn't seem to be actually necessary. So, if you feel like it, set it to zero using memset().

Now, that struct in_addr is a weird beast on different systems. Sometimes it's a crazy union with all kinds of #defines and other nonsense. But what you should do is only use the s_addr field in this structure, because many systems only implement that one.

struct sockadd_in6 and struct in6_addr are very similar, except they're used for IPv6.

struct sockaddr_storage is a struct you can pass to accept() or recvfrom() when you're trying to write IP version-agnostic code and you don't know if the new address is going to be IPv4 or IPv6. The struct sockaddr_storage structure is large enough to hold both types, unlike the original small struct sockaddr.

Example

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);

// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
```

```
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

See Also

accept(), bind(), connect(), inet_aton(), inet_ntoa()

Chapter 10

더 많은 참고문헌

여기까지 왔고, 더 많은 것을 원하는가? 네트워크에 대해서 더 배우려면 어디로 가야할까? (역자 주 : 이 장은 일부만 번역했습니다.)

10.1 책들

옛날식의, 정말로 손에 쥘 수 있는, 종이로 만든 책들이다. 아래에 있는 훌륭한 책 중 몇몇을 읽어보라. 이것들은 필자에게 좋은 보상을 주는 유명한 도서 유통업자들의 제휴된 링크로 여러분을 안내할 것이다. 좀 더 관대한 독자라면 페이팔로 beej@beej.us에 기부를 해주길 바란다. :-)

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Addison-Wesley Professional and Prentice Hall. ISBNs for volumes 1-2: 978-0131411555[https://beej.us/guide/url/unixnet1], 978-0130810816 [https://beej.us/guide/url/unixnet2].

Internetworking with TCP/IP, volume I by Douglas E. Comer. Published by Pearson. ISBN 978-0136085300[1].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3 (and a 3-volume set): 978-0201633467[2], 978-0201633542[https://beej.us/guide/url/tcpi2], 978-0201634952 [https://beej.us/guide/url/tcpi3], (978-0201776317[3]).

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 978-0596002978[4].

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 978-0321637734[5].

10.2 웹 참고문헌

인터넷에는 아래와 같은 정보가 있다.

BSD Sockets: A Quick And Dirty Primer[6] (Unix system programming info, too!)

The Unix Socket FAQ[7]

TCP/IP FAQ[8]

The Winsock FAQ[9]

[1] https://beej.us/guide/url/intertcp1
[2] https://beej.us/guide/url/tcpi1
[3] https://beej.us/guide/url/tcpi123
[4] https://beej.us/guide/url/tcpna
[5] https://beej.us/guide/url/advunix
[6] https://cis.temple.edu/~giorgio/old/cis307s96/readings/docs/sockets.html
[7] https://developerweb.net/?f=70
[8] http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/
[9] https://tangentsoft.net/wskfaq/

And here are some relevant Wikipedia pages:

Berkeley Sockets[10]

Internet Protocol (IP)[11]

Transmission Control Protocol (TCP)[12]

User Datagram Protocol (UDP)[13]

Client-Server[14]

Serialization[15] (packing and unpacking data)

10.3 RFCs

RFCs[16]—진짜 진탕이다! 이것들은 인터넷에 사용된 할당된 번호와 프로그래밍 API, 프로토콜 등을 설명하는 문서다. 여러분의 즐거움을 위해 그 중 일부에 대한 링크를 포함했다. 팝콘을 한 바가지 가져와서 생각모자를 써 보자.(역자 주 : 어떤 것을 진지하게 생각한다는 의미)

RFC 1[17] —The First RFC; this gives you an idea of what the "Internet" was like just as it was coming to life, and an insight into how it was being designed from the ground up. (This RFC is completely obsolete, obviously!)

RFC 768[18] —The User Datagram Protocol (UDP)

RFC 791[19] —The Internet Protocol (IP)

RFC 793[20] —The Transmission Control Protocol (TCP)

RFC 854[21] —The Telnet Protocol

RFC 959[22] —File Transfer Protocol (FTP)

RFC 1350[23] —The Trivial File Transfer Protocol (TFTP)

RFC 1459[24] —Internet Relay Chat Protocol (IRC)

RFC 1918[25] —Address Allocation for Private Internets

RFC 2131[26] —Dynamic Host Configuration Protocol (DHCP)

RFC 2616[27] —Hypertext Transfer Protocol (HTTP)

RFC 2821[28] —Simple Mail Transfer Protocol (SMTP)

RFC 3330[29] —Special-Use IPv4 Addresses

[10] https://en.wikipedia.org/wiki/Berkeley_sockets
[11] https://en.wikipedia.org/wiki/Internet_Protocol
[12] https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[13] https://en.wikipedia.org/wiki/User_Datagram_Protocol
[14] https://en.wikipedia.org/wiki/Client-server
[15] https://en.wikipedia.org/wiki/Serialization
[16] https://www.rfc-editor.org/
[17] https://tools.ietf.org/html/rfc1
[18] https://tools.ietf.org/html/rfc768
[19] https://tools.ietf.org/html/rfc791
[20] https://tools.ietf.org/html/rfc793
[21] https://tools.ietf.org/html/rfc854
[22] https://tools.ietf.org/html/rfc959
[23] https://tools.ietf.org/html/rfc1350
[24] https://tools.ietf.org/html/rfc1459
[25] https://tools.ietf.org/html/rfc1918
[26] https://tools.ietf.org/html/rfc2131
[27] https://tools.ietf.org/html/rfc2616
[28] https://tools.ietf.org/html/rfc2821
[29] https://tools.ietf.org/html/rfc3330

RFC 3493[30] —Basic Socket Interface Extensions for IPv6

RFC 3542[31] —Advanced Sockets Application Program Interface (API) for IPv6

RFC 3849[32] —IPv6 Address Prefix Reserved for Documentation

RFC 3920[33] —Extensible Messaging and Presence Protocol (XMPP)

RFC 3977[34] —Network News Transfer Protocol (NNTP)

RFC 4193[35] —Unique Local IPv6 Unicast Addresses

RFC 4506[36] —External Data Representation Standard (XDR)

The IETF has a nice online tool for searching and browsing RFCs[37].

[30] https://tools.ietf.org/html/rfc3493
[31] https://tools.ietf.org/html/rfc3542
[32] https://tools.ietf.org/html/rfc3849
[33] https://tools.ietf.org/html/rfc3920
[34] https://tools.ietf.org/html/rfc3977
[35] https://tools.ietf.org/html/rfc4193
[36] https://tools.ietf.org/html/rfc4506
[37] https://tools.ietf.org/rfc/

Index