Beej's Guide to Network Programming 简体中文版

Brian “Beej Jorgensen” Hall (http://beej.us/guide/bgnet/)
中文译者：Aaron Liao（http://beej-cn.netdpi.net）
Version：April 29, 2014

鸣谢

原着鸣谢

感谢古往今来帮助过我写完这份教程的人。感谢 Ashley 帮我将封面设计做成最棒的程序员风格、感谢产出自由（free）软件与套件的所有人，让我可以做出这份教程：GNU 、Slackware 、vim 、Python 、Inkscape 、Apache FOP 、Firefox 、Red Hat 与许多其它软件、最後特感谢数以千计的你们所写来的改善建议，以及文字上的鼓励。

我将这份教程献给我心目中电脑领域的大牛及鼓励我的人：Donald Knuth 、Bruce Schneier 、W. Richard Stevens 与 The Woz 、我的读者、以及整个自由与开源码软件社区。

这本书是以 XML 撰写，使用装载着 GNU tools 的 Slackware Linux 系统与 vim 编辑器。

封面的”艺术”与图片是使用 Inkscape 产生的，使用自定义的 Python scripts 将 XML 转换为 HTML 与 XSL-FO。接着用 Apache FOP 将 XSL-FO 输出档转为 Liberation 字体的 PDF 文档，Toolchain 100% 都使用自由与开源软件。

译注：上述的封面应是本教程纸质书的封面。

以下声明保留原文：
Unless otherwise mutually agreed by the parties in writing, the author offers the work as-is and makes no representations or warranties of any kind concerning the work, express, implied, statutory or otherwise, including, without limitation, warranties of title, merchantibility, fitness for a particular purpose, noninfringement, or the absence of latent or other defects, accuracy, or the presence of absence of errors, whether or not discoverable.

Except to the extent required by applicable law, in no event will the author be liable to you on any legal theory for any special, incidental, consequential, punitive or exemplary damages arising out of the use of the work, even if the author has been advised of the possibility of such damages.

This document is freely distributable under the terms of the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License. See the Copyright and Distribution section for details.
Copyright © 2012 Brian "Beej Jorgensen" Hall

目录

1. 前言 - 4
2. 何谓 Socket – 10
3. IP address 、结构与数据转换 - 17
4. 从 IPv4 移植为 IPv6 - 30
5. System call 或 Bust - 32
6. Client-Server 基础 - 52
7. 高等技术 - 68
8. 常见的问题 - 102
9. Man 手册 - 111
10. 参考文献 - 167

1. 前言

嘿！Socket 编程让你感到挫折吗？这份教程是不是只摘录了 man 手册，会很难吗？你想要写很酷的网路程序，可是你没有时间能费力读大量的数据结构，而且还得知道在调用 connect() 之前一定要先调用 bind() 的顺序等。

好，猜到了吗！其实我已经完成了这件痛苦的事情，我正要与你们分享这些资料，所以你来对地方了。本教程的目的是提供一份网路编程简介，给想要了解网路程序的 C 程序员。

在这边做个小结，我在这份教程里已经放上了最新的资料〔其实还好啦〕，而且增加了对 IPv6 的介绍！好好享用它吧！

1.1 本书的读者

本教程是一份导览（tutorial），不是全方位的参考手册。这份教程其实没什麼了不起，不仅不够全面，也没有完整到足以做为 socket 编程完全参考手册。

不过期盼本教程能让大家不要害怕 man 手册 ... :-)

1.2 平台与编译器

本教程谈到的代码是在 Linux PC 上，使用 GNU 的 gcc 编译器所编译的。然而，它应该在任何有 gcc 的平台上都能编译。不过实际上，如果你在 Windows 上编程，这可能会有点不太一样，请参考下列关於 Windows 编程的章节。

1.3 官方网页与书本

本教程的官方链接是 http://beej.us/guide/bgnet/。在这里你也能找到代码的示例，以及各种语言的译本。

如果需要购买价格比较便宜的复本〔有人称为”书”〕，请到 http://beej.us/guide/url/bgbuy。我很感谢您的购买，因为这可以帮忙我继续依靠写教程过活！

1.4 Solaris/SunOS 程序员要注意的事情

当编译 Solaris 或 SunOS 平台的代码时，你需要指定一些额外的命令行参数，以链接（link）正确的程序库（library）。为了达到这个目的，可以在编译命令後面简单加上”-lnsl -lsocket -lresolv”，类似这样：

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

如果还是有错误信息，你可以再加上一个”-lnext”到命令行的後端。我不太清楚这样做了什麼事，不过有些人是会这样用。

你可能会遇到的另一个问题是调用 setsockopt()。这个原型与在我 Linux 系统上的不一样，所以可以这样替换：

```
int yes=1;
```

输入这行：

```
char yes='1';
```

因为我没有 Sun 系统，所以我无法测试上面的资料，这只是有人用 email 跟我说的。

1.5 Windows 程序员要注意的事情

本教程以前只讨论一点 Windows，纯粹是我很不喜欢。不过我应该要客观的说 Windows 其实提供很多基本安装，所以显然是个完备的操作系统。

人家说：小别胜新婚，这里我相信这句话是对的〔或许是年纪的关系〕。不过我只能说，我已经十几年没有用 Microsoft 的操作系统来做自己的工作了，这样我很开心！

其实我可以安然地打安全牌告诉你：”没问题阿，你尽量去用 Windows 吧！”… 没错，其实我是咬着牙根说这些话的。

所以我还是在拉拢你来试试 Linux [1]、BSD [2]，或一些 Unix 风格的系统。

不过人们各有所好，而 Windows 的用戶也乐於知道这份教程的内容能用在 Windows，

只是需要改变一点代码而已。

你可以安装一个酷玩意儿－ Cygwin [3]，这是让 Windows 平台使用的 Unix 工具集。我曾在秘密情报网听过，这个能让全部的代码不经过修改就能编译。

不过有些人可能想要用纯 Windows 的方法来做。只能说你很有勇气，而你所要做的事就是：立刻去弄个 Unix！喔，不是，我开玩笑的。这些日子以来，大家一直认为我对 Windows 是很友善的。

你所要做的事情就是〔除非你安装了 Cygwin！〕：首先要忽略我这边提过的很多系统 header（标头档），而你唯一需要 include （引用）的是：

```
#include <winsock.h>
```

等等，在你用 socket 程序库做任何事情之前，必须要先调用 WSAStartup()。代码看起来像这样：

```
#include <winsock.h>
{
 WSADATA wsaData; // if this doesn't work
 //WSAData wsaData; // then try this instead

 // MAKEWORD(1,1) for Winsock 1.1, MAKEWORD(2,0) for Winsock 2.0:

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

你也必须告诉编译器要链接（link） Winsock 程序库，在 Winsock 2.0 通常称为 wsock32.lib 或 winsock32.lib 或 ws2_32.lib。在 VC++ 底下，可以透过项目（Project）菜单，在设置（Settings）底下 ...。按下 Link tag，并找到”Object/library modules”的标

题。新增”wsock32.lib”（或者你想要用的程序库）到表中。

最後，当你用好 socket 程序库时，你需要调用 WSACleanup()，细节请参考在线帮助手册。

只要你做好这些工作，本教程後面的示例应该都能顺利编译，只有少部分例外。

还有一件事情，你不能用 close() 关闭 socket，你要用 closesocket() 来取代。而且 select() 只能用在 socket descriptors 上，不能用在 file descriptors（像 stdin 就是 0）。

还有一种你能用的 socket 类型，CSocket，细节请查询你的编译器手册。

要取得更多关於 Winsock 的信息可以先阅读 Winsock FAQ [4]。

最後，我听说 Windows 没有 fork() 系统调用，我在一些示例中会用到。你可能需要连结到 POSIX 程序库或要让程序能动的一些程序库，或许你也可以用 CreateProcess() 来取代。fork() 不需要参数，但是 CreateProcess() 却需要大约 480 亿个参数。如果你不想用，CreateThread() 会稍微比较容易理解 … 不过多线程（multithreading）的讨论则不在本教程的范畴中。我只能尽量提到而已，你明白的！

译注：作者说 480 亿个参数只是想表达 CreateProcess() 需要的参数很多。

1.6 来信原则

通常我很乐意帮助解决来信的问题，所以请尽管写信来，不过，我不一定会回信，我很忙，而且我还有三次无法回答你们的问题。在这种情况下，我通常只会把信息删掉，这并没有针对什麼人；我只是没空可以详细答覆问题。

同样的原则，越复杂的问题我就越不想回答。如果你很想要收到答覆，你可以在寄信之前先简化你的问题，并确定你引述了相关的资料〔比如平台、编译器、得到的错误信息，以及任何你想的到可以协助我找出问题的资料〕。对於更多的要点，请阅读 ESR 的文件，提问的智慧（How To Ask Questions The Smart Way）[5]。

若你没有收到答覆，请自行多多 hack（研究），试着找出答案，如果真的还是无法解决，那麼再写信给我，并提供你找到的资料，期盼会有足够资料可以让我帮忙解决。

现在我一直缠着你说要怎麼写信给我，以及哪些情况千万不要写信给我，我只想让你知道，我诚心的感谢这几年本教程所收到的赞美。它真的是个轻薄短小的教程，而且让我很高兴听到大家说它非常实用！:-) 感谢您！

1.7 镜像站台（Mirroring）

欢迎镜像本站，无论公开或私人的。若你公开镜像本站，并想要我从官方的网站连结，请送个信息到 beej@beej.us。

1.8 译者该注意的

如果你想要将本指南翻译为其它语言，请寄信到 beej@beej.us，我会从官方主页连结你的译本，请随意加上你的名字与联络资料到译本中。

请注意下列〔版权与散布〕一节所列的授权限制，如果你想要我放译本，跟我说就好了；若你想要自己架站，我会鏈接到你所提供的鏈接，任何方式都行。

1.9 版权与出版（Copyright and Distribution）

Beej 的网路编程指南（Beej's Guide to Network Programming）版权是属於 Copyright © 2012 Brian “Beej Jorgensen” Hall。对於特定的代码与译本在下面有例外，本作品基於 Creative Commons Attribution- Noncommercial- No Derivative Works 3.0 License 授权。欲检视该授权的复本请参考 http://creativecommons.org/licenses/by-nc-nd/3.0/ 或者写封信到这个住址：Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA. 该授权对於 “No Derivative Works” 这部分的例外如下：这份教程可以自由翻译成任何语言，提供的译本要正确，而重新列印时需要保持本教程的完整性。原本的教程授权规范亦会套用於译本上。译本可以包含译者的名字与联络资料。本教程所介绍的 C 来源代码在此公开发表（public domain），并完全免於任何授权限制。欢迎老师们免费推荐或提供本教程的复本给你们的学生使用。

需要更多消息请联系 beej@beej.us

译注：中文读者可来信给（Aaron Liao）aaron@netdpi.net。

以下是 1.9 节，版权声明的原文内容：

Beej's Guide to Network Programming is Copyright © 2012 Brian “Beej Jorgensen” Hall.
With specific exceptions for source code and translations, below, this work is licensed under the Creative Commons Attribution- Noncommercial- No Derivative Works 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.
One specific exception to the “No Derivative Works” portion of the license is as follows: this guide may be freely translated into any language, provided the translation is accurate, and the guide is reprinted in its entirety. The same license restrictions apply to the translation as to the original guide. The translation may also include the name and contact information for the translator.
The C source code presented in this document is hereby granted to the public domain, and is completely free of any license restriction. Educators are freely encouraged to recommend or supply copies of this guide to their students. Contact beej@beej.us for more information.

参考资料

[1] http://www.linux.com/

[2] http://www.bsd.org/

[3] http://www.cygwin.com/

[4] http://tangentsoft.net/wskfaq/

[5] http://www.catb.org/~esr/faqs/smart-questions.html

2. 何谓 Socket

你一直听到人家在讲”sockets”（套接字），你可能也想知道这些是什麼东西。

好的，其实它们是：利用标准 UNIX file descriptors（文件描述符）与其它程序沟通的一种方式。

什麼？

OK，你可能有听过有些黑客（hacker）说过：”我的天呀！在 UNIX 系统中的任何东西都可以视为文件！”

他说的是事实。当 UNIX 程序要做任何类型的 I/O 时，它们会读写 file descriptor。File descriptor 单纯只是跟已开启文件有关的整数。只是〔关键在於〕，该文件可以是一个网路连接、FIFO、pipe（管道）、terminal（终端）、真实的磁盘文件、或只是相关的东西。在 UNIX 所见都是文件！所以当你想要透过 Internet（互联网）跟其它的程序沟通时，你需要透过一个 file descriptor 来达成，这点你一定要相信。

”那麼，Smarty-Pants 先生，我在哪里可以取得这个用在网路通訊的 file descriptor 呢？”

这可能是你现在心里的问题，我会跟你说的：你能调用 socket() system routine（系统例程）。它会传回 socket descriptor，你可以用精心设计的 send() 与 recv() socket calls〔man send、man recv〕来透过 socket descriptor 进行通讯。

”不过，嘿嘿！”

现在你可能在想：”既然只是个 file descriptor，为什麼我不能用一般的 read() 与 write() call 透过 socket 进行通讯，而要用这什麼鬼东西？”

简单说：”可以！”

详细点的说法是：”可以，不过 send() 与 recv() 让你能对数据传输有更多的控制权”。

接下来呢？

这麼说吧：有很多种 sockets，如 DARPA Internet Sockets（互联网地址）、本地节点的路

径名（path names on a local node，UNIX Sockets）、CCITT X.25 地址（你可以放心忽略 X.25 Sockets），可能还有其它的，要看你用的是哪种 UNIX 系统。在这里我们只讨论第一种：Internet Sockets。

2.1. 两种 Internet Sockets

这是什麼？有两种 Internet sockets 吗？

是的，喔不，我骗你的啦。其实有更多种 Internet sockets，只是我不想吓唬你。所以这里我只打算讨论两种，不过我还会告诉你”Raw Sockets”，这是很强大的东西，所以你应该要好好研究一下它们。

译注：
一般的 socket 只能读取 transport layer 传输层以上〔不含〕的数据，raw socket 一般用在设计 network sniffer，可以让应用程序取得网路数据包底层的数据〔如 TCP 层、IP 层，甚至 link layer socket 可以读取到 link layer 层〕，并用以分析数据包。这份教程不会谈到这类的编程，有兴趣的读者可自行参考：Unix Network Programming Vol. 1、TCP/IP 网路程式实验与设计或 libpcap。

好吧，不聊了。那到底是有哪两种 Internet sockets 呢？

其中一个是”Stream Sockets”（串流式 Sockets）；而另一个是”Datagram Sockets”（讯息式 Sockets），之後我们分别以”SOCK_STREAM”与”SOCK_DGRAM”来表示。Datagram sockets 有时称为”无连接的 sockets”（connectionless sockets）（虽然它们也可以用 connect()，如果你想这麼做的话，请见後面章节的 connect()）。

Stream sockets 是可靠的、双向连接的通讯串流。若你以”1、2”的顺序将两个项目输出到 socket，它们在另一端则会以”1、2”的顺序抵达。而且不会出错。

哪里会用到 stream sockets 呢？

好的，你应该听过 telnet 软件吧，不是吗？它就是用 stream sockets。你所输入的每个字都需要按照你所输入的顺序抵达，有吗？网页浏览器所使用的 HTTP 协议也是用 stream sockets 取得网页。的确，若你以 port 80 telnet 到一个网站，并输入”GET / HTTP/1.0”，然後按两下 Enter，它就会输出 HTML 给你！

Stream sockets 是如何达成如此高品质的数据传送呢？

它们用所谓的”The Transmission Control Protocol”（传输控制协议），就是常见的”TCP”（TCP 的全部细节请参考 RFC 793[6]）。TCP 保证你的数据可以依序抵达而且不会出错。你以前可能听过”TCP”是”TCP/IP”比较优的部分，这边的”IP”是指”Internet Protocol”（互联网协议，请见 RFC 791[7]）。IP 主要处理 Internet routing（互联网路由），通常不保障数据的完整性。

酷喔。那 Datagram socket 呢？为什麼它们号称无连接呢？这边有什麼好主意？为什麼它们是不可靠的？

好，这里说明一下现况：如果你送出一个 datagram（信息数据包），它可能会顺利到达、可能不会按照顺序到达，而如果它到达了，数据包的数据就是正确的。

译注：
TCP 会在传输层对将上层送来的过大数据分割成多个 TCP 段（TCP segments），而 UDP 本身不会，UDP 是信息导向的（message oriented），若 UDP 信息过大时（整体数据包长度超过 MTU），则会由 host 或 router 在 IP 层对数据包进行分割，将一个 IP packet 分割成多个 IP fragments。IP fragmention 的缺点是，到达端的系统需要做 IP 数据包的重组，将多个 fragments 重组合并为原本的 IP 数据包，同时也会增加数据包遗失的可能性。如将一个 IP packet 分割成多个 IP fragments，只要其中一个 IP fragment 遗失了，到达端就会无法顺利重组 IP 数据包，因而造成数据包的遗失，若是高可靠度的应用，则上层协议需重送整个 packet 的数据。

[6] http://tools.ietf.org/html/rfc793

[7] http://tools.ietf.org/html/rfc791

Datagram sockets 也使用 IP 进行 routing（路由），不过它们不用 TCP；而是用”UDP，User Datagram Protocol”（用户数据包协议，请见 RFC 768 [8]）。

为什麼它们是无连接的？

好，基本上，这跟你在使用 stream socket 时不同，你不用维护一个开启的连接，你只需打造数据包、给它一个 IP header 与目的资料、送出，不需要连接。通常用 datagram socket 的时机是在没有可用的 TCP stack 时；或者当一些数据包遗失不会造成什麼重大事故时。这类应用程序的例子有：tftp（trivial file transfer protocol，简易文件传输协议，是 FTP 的小兄弟），多人游戏、串流音乐、影像会议等。

”等一下！tftp 和 dhcpd 是用来在一台主机与另一台之间传输二进制的应用数据！你如果想要应用程序能在数据抵达时正常运作，那数据就不能遗失阿！这是什麼黑魔法？”

好，我的人类朋友，tftp 与类似的程序会在 UDP 的上层使用它们自己的协议。比如：tftp 协议会报告每个收送的数据包，到达端必须送回一个数据包表示：”我收到了！”〔一个”ACK”数据包〕。若原本数据包的传送端在五秒内没有收到回应，这表示它该重送这个数据包，直到收到 ACK 为止。在实作可靠的 SOCK_DGRAM 应用程序时，这个回报的过程很重要。

对於无需可靠度的（unreliable）应用程序，如游戏、音效、或影像，你只需忽略遗失的数据包，或也许能试着用技巧弥补回来。（雷神之锤的玩家都知道的一个技术名词的影响：accursed lag。在这个例子中，”accursed”（受到诅咒）这个字代表各种低级的意思）。

为什麼你要用一个不可靠的底层协议？

有两个理由：第一个理由是速度，第二个理由还是速度。忘了这个数据包是比较快的方式，相较之下，持续追踪全部的数据包否安全抵达，并确保依序抵达是比较慢的。如果你想要传送聊天讯息，TCP 很好；不过如果你想要替全世界的玩家，每秒送出 40 个位置更新的数据，且若遗失一到两个数据包并不会有太大的影响时，此时 UDP 是一个好的选择。

[8] http://tools.ietf.org/html/rfc768

译注：
stream（串流式）socket 是指应用程序要传输的数据就如水流（串流）在水管中传输一般，经由这个 stream socket 流向目的，串流式 socket 是数据会由传输层负责处理遗失、依序送达等工作，以在传输层确保应用程序所送出的数据能够可靠且依序抵达，而应用程序若对数据有可靠与依序的需求时，使用 stream socket 就不用自行处理这类的工作。
datagram（信息式）socket 是基於讯息导向的方式传送数据，应用程序送出的每笔数据会如平信的概念送出，由於遶送数据包的路径可能会随着网路条件而改变，每笔数据抵达的顺序不一定会按照送出的顺序抵达，并且如平信般，信件可能在递送过程遗失，而寄件人并无法知道是否递送成功。
初步简单知道应用这两种 sockets 的时机：当需要数据能完整送达目地时，就使用 stream socket，若是部分数据遗失也无妨时，就可以使用 datagram socket。

2.2 底层漫谈与网路理论

因为我只着重於协议的分层，该是谈谈网路是如何真正的运作的时候了，并呈现一些如何

打造 SOCK_DGRAM 数据包的例子。实务上，你或许可以忽略这一节，然而，这里有很好的观念背景。

Data Encapsulation.

嘿！孩子们，该是学习数据封装（Data Encapsulation）的时候了。这很重要。它就是非常的重要，所以如果你是在加州这里上的网路课程，也只能学到皮毛。

基本上我们会讲到这些：数据包的诞生、将数据包打包〔”封装”〕到第一个协议〔所谓的 TFTP 协议〕的 header 中〔几乎是最底层了〕，接着将全部的东西〔包含 TFTP header〕封装到下一个协议中〔所谓的 UDP〕，接着下一个协议〔IP〕，最後衔接到硬件〔实体〕层上面的协议〔所谓的 Ethernet，乙太网路〕。

当另一台电脑收到数据包时，硬件会解开 Ethernet header，而 kernel 会解开 IP 与 UDP header，再来由 TFTP 程序解开 TFTP header，最後程序可以取得数据。

现在我最後要谈个声名狼藉的分层网路模型（Layered Network Model），亦称”ISO/OSI”。这个网路模型介绍了一个网路功能系统，有许多其它模型的优点。例如，你可以写刚好一样的 socket 程序，而不用管数据在实体上是怎麼传送的〔Serial 、 thin Ethernet 、 AUI 之类〕。因为在底层的程序会帮你处理这件事。真正的网路硬件与拓朴对 socket 程序设计师而言是透明的。

不罗嗦，我将介绍这个成熟模型的分层。为了网路课程的测验，要记住这些。

- Application（应用层）
- Presentation（表现层）
- Session（会谈层）
- Transport（传输层）
- Network（网路层）
- Data Link（数据链结层）
- Physical（实体层）

实体层就是硬件（serial 、 Ethernet 等）。而应用层你可以尽可能的想像，这是个用户与网路互动的地方。

现在这个模型已经很普及，所以你如果愿意的话，或许可以将它当作是一本汽车修理指南来使用。与 Unix 比较相容的分层模型有：

- 应用层（Application layer：telnet 、 ftp 等）
- 主机到主机的传输层（Transport layer：TCP 、 UDP）
- 互联网层（Internet layer：IP 与路由遶送）
- 网路存取层（Network Access Layer：Ethernet 、 wi-fi 、诸如此类）

此时，你或许能知道这几层是如何对应到原始数据的封装。

看看在打造一个简单的数据包需要多少工作呢？

天阿！你得自己用”cat”将数据填入数据包的 header 里！

开玩笑的啦。

你对 stream socket 需要做的只有用 send() 将数据送出。而在 datagram socket 需要你做的是，用你所选择的方式封装该数据包，并且用 sendto() 送出。Kernel 会自动帮你建立传输层与网路层，而硬件处理网路存取层。啊！真现代化的技术。

所以该结束我们短暂的网路理论之旅了。

喔！对了，我忘记告诉你我想要谈谈 routing（路由）了。恩，没事！没关系，我不打算全部讲完。

Router（路由器）会解开数据包的 IP header，参考自己的 routing table（路由表）...。如果你真的很想知道，你可以读 IP RFC [9]。如果你永远都不想碰它，其实你也可以过得很好。

[9] http://tools.ietf.org/html/rfc791

译注：
若读者想要深入了解互联网或 TCP/IP 观念，以下是译者推荐的参考文献，这些都是网路 TCP/IP 观念的经典着作。

[C1] Behrouz Forouzan, TCP/IP Protocol Suite, 4 edition, Mcgraw-Hill Inc., 2009.
[C2] Kevin R. Fall and W. Richard Stevens, TCP/IP Illustrated, The Protocols, Vol. 1, 2 edition, Addison-Wesley Inc., 2011.
[C3] Douglas E. Comer, Internetworking with TCP/IP principles, protocols, and architecture, Vol. 1, 6 edition, Pearson education Inc., 2013.
[C4] Pat Eyler, Networking Linux: A Practical Guide to TCP/IP, New Riders Inc., 2001.

3. IP address 、结构与数据转换

这里是好玩的地方，我们要开始谈代码了。

不过，我们一开始要讨论的代码会比较少！

耶！因为我想要先讲点 IP address（地址）与 port（端口），这样才会比较有谱；接着我们会讨论 socket API 如何储存与控制 IP address 和其它数据。

3.1. IPv4 与 IPv6

在 Ben Kenobi 还是叫 Obi Wan Kenobi 的那段过去的美好时光，有个很棒的 network routing system（网路路由系统），称为 Internet Protocol Version 4（互联网协议第四版），又称为 IPv4。它的地址是由四个 bytes 组成（亦称为四个”octets”），而格式是由句点与数字组成，像是这样：192.0.2.111。

你或许曾经看过。

实际上，在撰写本文时，几乎整个 Internet（互联网）的每个网站都还是使用 IPv4。

每个人跟 Obi Wan 都很开心，一切都是如此美好，直到某个名为 Vint Cerf 的人提出质疑，警告所有人 IPv4 address 即将耗尽。

Vint Cerf [10] 除了提出即将到来的 IPv4 危机警告，他本身还是有名的 Internet 之父，所以我真的没资格能评论他的判断。

你说的是耗尽 address 吗？会发生什麼事呢？其实我的意思是，32-bit 的 IPv4 address 有几十亿个 IP address，我们真的有几十亿台的电脑在用吗？

是的。在一开始大家也是认为这样就够用了，因为当时只有一些电脑，而且每个人认为几十亿是不可能用完的大数目，还很慷慨的分给某些大型组织几百万个 IP address 供他们自己使用〔例如：Xerox 、 MIT 、 Ford 、 HP 、 IBM 、 GE 、 AT&T 及某个名为 Apple 的小公司，族繁不及备载〕。

不过现实状况是，如果不是有些变通的方法，我们早就用光 IPv4 地址了。

我们现在生活於每个人、每台电脑、每部计算机、每只电话、每部停车计时收费器、以

及每条小狗〔为什麼不行？〕都有一个 IP address 的年代，因此，IPv6 诞生了。

因为 Vint Cerf 可能是不朽的，〔即使他的肉体终究应该会回归自然，我也希望不要，不过他的精神或许已经以某种超智慧的 ELIZA [11] 程式存在於 Internet2 的核心〕，应该没有人想要因为下一代互联网协议又没有足够的地址，然後又听到他说：”我要告诉你们一件事 ...” 。

那你有什麼建议吗？

我们需要更多的地址，我们需要不止两倍以上的地址、不止几十亿倍、千兆倍以上，而是 79 乘以 百万 乘以 十亿 乘以 兆倍以上的可用地址！你们大家将会见识到的。

你说：” Beej，真的吗？我还是有许多可以质疑这个大数字的理由。”

好的，32 bits 与 128 bits 的差异听起来似乎不是很多；它只多了 96 个 bits 而已，不是吗？不过请记得，我们所谈的是等比数列；32 bits 表示个 40 亿的数字〔2 的 32 次幂〕，而 128 bits 表示的大约是 340 个兆兆兆的数字〔2 的 128 次幂〕，这相当於宇宙中的每颗星星都能拥有一百万个 IPv4 Internets。

大家顺便忘了 IPv4 的句号与数字的长相吧；现在我们有十六进制的表示法，每两个 bytes 间以冒号分隔，类似这样：

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

这还不是全部呢！大部分的时候，你的 IP address 里面会有很多个零，而你可以将它们压缩到两个冒号间，你也可以在每个 byte pair（字节对）上保留零。例如，这些地址的配对是相等的：

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

[10] http://en.wikipedia.org/wiki/Vinton_Cerf

[11] http://en.wikipedia.org/wiki/ELIZA

地址 ::1 是个 loopback（回路网络接口）地址，它永远只代表”我现在执行的这台电脑”，在 IPv4 中，lookback 地址是 127.0.0.1。

最後，你可能会遇到 IPv6 与 IPv4 兼容的模式。例如，如果你愿意的话，你可以将 IPv4 address 192.0.2.33 以 IPv6 地址表示，可以使用如下的符号：”::ffff:192.0.2.33”。

我们所谓的自信，实际上，因为自信，所以 IPv6 的发明人很有把握的将兆来兆去的地址用於保留用途，不过说实在的，我们有这麼多地址，谁能算清楚呢？

还剩下很多地址可以分配给星系中每个行星的每个男人、女人、小孩、小狗跟停车计时收费器。相信我，星系中的每个行星都有行车计时收费器。你明白这是真的。

3.1.1 Sub network (子网)

为了结构化的理由，有时我们这样宣告是很方便的：”IP address 的前段是 IP address 的 network（网路），而後面的部分是 host（主机）。”

例如：在 IPv4，你可能有 192.0.2.12，而我们可以说前面三个 bytes 是 network，而最後一个 byte 是 host。或者换个方式，我们能说 host 12 位在 network 192.0.2.0。〔请参考我们如何将 host byte 清为零〕。

接下来要讲的是过时的资料了！

真的吗？

很久很久以前，有 subnets（子网）的”class”（分类），在这里，地址的第一个、前二个或前三个 bytes 都是属於 network 的一部分。如果你很幸运可以拥有一个 byte 的 network，而另外三个 bytes 是 host 地址，那在你的网路上，你有价值 24 bits 的 host number〔大约两千四百万个地址左右〕。这是一个”Class A”（A 类）网路；相对则是一个”Class C”（C 类）的网路，network 有三个 bytes 、而 host 只有一个 byte〔256 个 hosts，而且还要再扣掉两个保留的地址〕。

所以，如同你所看到的，只有一些 Class A 网路，一大堆的 Class C 网路，以及一些中等的 Class B 网路。

IP address 的网络地址位数由 netmask（网路掩码）决定，你可以将 IP address 与 netmask 进行 AND bitwise，就能得到 network 的值。Netmask 一般看起来像是 255.255.255.0〔如：若你的 IP 是 192.0.2.12，那麼使用这个 netmask 时，你的 network 就会是 192.0.2.12 AND 255.255.255.0 所得到的值：192.0.2.0〕。

无庸置疑的，这样的分类对於 Internet 的最终需求而言并不够细腻；我们已经以相当快的速度在消耗 Class C 网路，这是我们都知道一定会耗尽的 Class，所以不用费心去想了。补救的方式是，要能接受任意个 bits 的 netmask，而不单纯是 8、16 或 24 个而已。所以你可以有个 255.255.255.252 的 netmask，这个 netmask 能切出一个 30 个 bits 的 network 及 2 个 bits 的 host，这个 network 最多有四台 hosts〔注意，netmask 的格式永远都是：前面是一连串的 1，然後，後面是一连串的 0〕。

不过一大串的数字会有点不好用，比如像 255.192.0.0 这样的 netmask。首要是人们无法直觉地知道有多少个 bits 的 1；其次是这样真的很不严谨。因此，後来的新方法就好多了。你只需要将一个斜线放在 IP address 後面，接着後面跟着一个十进制的数字用以表示 network bits 的数目，类似这样：192.0.2.12/30。

或者在 IPv6 中，类似这样：2001:db8::/32 或 2001:db8:5413:4028::9db9/64。

3.1.2. Port Number（连接埠号码）

如果你还记得我之前跟你说过的分层网路模型（Layered Network Model），它将网路层（IP）与主机到主机间的传输层〔TCP 与 UDP〕分开。

我们要加快脚步了。

除了 IP address 之外〔IP 层〕，有另一个 TCP〔stream socket〕使用的地址，刚好 UDP〔datagram socket〕也是。它就是 port number，这是一个 16-bit 的数字，就像是连线的本地端地址一样。

将 IP address 想成饭店的地址，而 port number 就是饭店的房间号码。这是贴切的比喻；或许以後我会用汽车工业来比喻。

你说想要有一台电脑能处理收到的电子邮件与网页服务－你要如何在一台只有一个 IP 地址的电脑上分辨呢？

好，Internet 上不同的服务都有已知的（well-known）port numbers。你可以在 Big IANA Port 表 [12] 中找到，或者若你在 Unix 系统上，你可以参考文件 /etc/services。HTTP（网站）是 port 80 、 telnet 是 port 23 、 SMTP 是 port 25，而 DOOM 游戏 [13] 使用 port 666 等，诸如此类。Port 1024 以下通常是有特地用途的，而且要有操作系统管理员权限才能使用。

揌，这就是 port number 的介绍。

3.2 Byte Order（字节的顺序）

长久以来都有两种 byte orderings，不过後来才知道，根本差多了。

我开玩笑的，不过其中一个真的比另一个好 :-)

这真的不太好解释，所以我只会扯蛋：你的电脑可能背着你用相反的顺序来储存 bytes。

我知道！没有人跟你说。

Byte Order 其实就是，在 Internet 世界中的每个人一般都已经同意的，如果你想要用两个 bytes 的十六进制数字来表示，比如说 b34f，你可以将它以 b34f 的顺序储存。很合理，而 Wilford Brimley [14] 会跟你说，这麼做是对的。这个数字是先储存比较大的那一边（big end），所以称为 Big-Endian。

毫无疑问地，世界上的电脑那麼多，像 Intel 或 Intel 兼容的中央处理器就是将 bytes 反过来储存，所以 b34f 存在内存中的顺序就是 4fb3，这样的储存方式称为 Little-Endian。

不过，等等。我还没解释名词！照理说，Big-Endian 又称为 Network Byte Order，因为这个顺序与我们网路型别的顺序一样。

你的电脑会以 Host Byte Order 储存数字，如果是 Intel 80x86，Host Byte Order 是 Little-Endian；若是 Motorola 68k，则 Host Byte Order 是 Big-Endian；若是 PowerPC，Host Byte Order 就是 ... 恩，这要看你的 PowerPC 而定。

大多数当你在打造数据包或填写数据结构时，你需要确认你的两个数字跟四个数字都是 Network Byte Order。只是如果你不知道本地端的 Host Byte Order，那该怎麼做呢？

好消息是你只需要假设 Host Byte Order 不正确，然後每次都透过一个函数将值设定为

Network Byte Order。如果有必要，该函数会进行魔法的转换，而这个方式可以让你的代码能方便的移植到不同 endian 的机器上。

你可以转换两种型别的数值：short〔两个 bytes〕与 long〔四个 bytes〕。这些函数也可以用在 unsigned 变量。比如说，你想要将 short 从 Host Byte Order 转换为 Network Byte Order，用”h”代表”host”，用”n”代表”network”，而”s”代表”short”，所以是：h-to-n-s，或者 htons()〔读做：”Host to Network Short”〕。

这真是太简单了...

你可以用任何你想要的方式来组合”n”、”h”、”s”与”l”，不过别用太蠢的组合，比如：没有这样的函数 stolh()〔”Short to Long Host”〕，没有这种东西，不过有：

```
htons() host to network short
htonl() host to network long
ntohs() network to host short
ntohl() network to host long
```

基本上，你需要在送出以前将数值转换为 Network Byte Order，并在收到之後将数值转回 Host Byte Order。

抱歉，我不知道 64-bit 的改变，如果你想要做浮点数的话，可以参考第 7-4 节。

[12] http://www.iana.org/assignments/port-numbers

[13] http://en.wikipedia.org/wiki/Doom_(video_game)

[14] http://en.wikipedia.org/wiki/Wilford_Brimley

如果我没特别强调的话，本文中的数值默认值是 Host Byte Order。

3.3. 数据结构

很好，终於讲到这里了，该是谈谈编程的时间了。在本节，我会介绍 socket 接口的各种数据型别，因为它们有些会不太好理解。

首先是最简单的：socket descriptor，型别如下：

int

就是一般的 int。

从这里开始会有点不好理解，所以不用问太多，直接读过就好。

我的第一个 StructTM －struct addrinfo，这个数据结构是最近的发明，用来准备之後要用的 socket 地址数据结构，也用在主机名（host name）及服务名（service name）的查询。当我们之後开始实际应用时，才会开始觉得比较靠谱，现在只需要知道你在建立连接调用时会用到这个数据结构。

```
struct addrinfo {
    int ai_flags; // AI_PASSIVE, AI_CANONNAME 等。
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol; // 用 0 当作 "any"
    size_t ai_addrlen; // ai_addr 的大小，单位是 byte
    struct sockaddr *ai_addr; // struct sockaddr_in 或 _in6
    char *ai_canonname; // 典型的 hostname
    struct addrinfo *ai_next; // 链表、下个节点
};
```

你可以载入这个数据结构，然後调用 getaddrinfo()。它会返回一个指针，这个指针指向一个新的链表，这个链表有一些数据结构，而数据结构的内容记载了你所需的东西。

你可以在 ai_family 栏位中设定强制使用 IPv4 或 IPv6，或者将它设定为 AF_UNSPEC，AF_UNSPEC 很酷，因为这样你的程序就可以不用管 IP 的版本。

要注意的是，这是个链表：ai_next 是指向下一个成员（element），可能会有多个结果让你选择。我会直接用它提供的第一个结果，不过你可能会有不同的个人考量；先生！我不是万事通。

你会在 struct addrinfo 中看到 ai_addr 栏位是一个指向 struct sockaddr 的指针。这是我们开始要了解 IP 地址结构中有哪些细节的地方。有时候，你需要的是调用 getaddrinfo()

帮你填好 struct addrinfo。然而，你必须查看这些数据结构，并将值取出，所以我在这边会进行说明。

〔还有，在发明 struct addrinfo 以前的代码都要手动填写这些数据的每个栏位，所以你会看到很多 IPv4 的代码真的用很原始的方式去做这件事。你知道的，本教程在旧版也是这样做〕。

有些 structs 是 IPv4，而有些是 IPv6，有些两者都是。我会特别注明清楚它们属於哪一种。

总之，struct sockaddr 记录了很多 sockets 类型的 socket 的地址资料。

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

sa_family 可以是任何东西，不过在这份教程中我们会用到的是 AF_INET〔IPv4〕或 AF_INET6〔IPv6〕。sa_data 包含一个 socket 的目地地址与 port number。这样很不方便，因为你不会想要手动的将地址封装到 sa_data 里。

为了处理 struct sockaddr，程序设计师建立了对等平行的数据结构：struct sockaddr_in〔"in"是代表"internet"〕用在 IPv4。

而这有个重点：指向 struct sockaddr_in 的指针可以转型（cast）为指向 struct sockaddr 的指针，反之亦然。所以即使 connect() 需要一个 struct sockaddr *，你也可以用 struct sockaddr_in，并在最後的时候对它做型别转换！

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)
struct sockaddr_in {
    short int sin_family; // Address family, AF_INET
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr; // Internet address
    unsigned char sin_zero[8]; // 与 struct sockaddr 相同的大小
};
```

这个数据结构让它很容易可以参考（reference）socket 地址的成员。要注意的是 sin_zero〔这是用来将数据结构补足符合 struct sockaddr 的长度〕，应该要使用 memset()函数将 sin_zero 整个清为零。还有，sin_family 是对应到 struct sockaddr 中的 sa_family，并应该设定为”AF_INET”。最後，sin_port 必须是 Network Byte Order〔利用 htons()〕。

让我们再更深入点！你可以在 sruct in_addr 里看到 sin_addr 栏位。

那是什麼？

好，别太激动，不过它是其中一个最恐怖的 union：

```
// (仅限 IPv4 — Ipv6 请参考 struct in6_addr)
// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

哇！好耶，它以前是 union，不过这个包袱现在似乎已经不见了。因此，若你已将 ina 宣告为 struct sockaddr_in 的型别时，那麼 ina.sin_addr.s_addr 会参考到 4-byte 的 IP address（以 Network Byte Order）。要注意的是，如果你的系统仍然在 struct in_addr 使用超恐怖的 union，你依然可以像我上面说的，精确地参考到 4-byte 的 IP address〔这是因为 #define〕。

那麼 IPv6 会怎样呢？

IPv6 也有提供类似的 struct，比如：

```
// (IPv6 only--see struct sockaddr_in and struct in_addr for IPv4)
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]; // IPv6 address
};
```

要注意到 IPv6 协议有一个 IPv6 address 与一个 port number，就像 IPv4 协议有一个 IPv4 address 与 port number 一样。

我现在还不会介绍 IPv6 的流量资料，或是 Scope ID 栏位 … 这只是一份入门教程嘛 :-)

最後要强调的一点，这个简单的 struct sockaddr_storage 是设计用来足够储存 IPv4 与 IPv6 structures 的 structure。〔你看看，对於某些 calls，你有时无法事先知道它是否会使用 IPv4 或 IPv6 address 来填好你的 struct sockaddr。所以你用这个平行的 structure 来传递，它除了比较大以外，也很类似 struct sockaddr ，因而可以将它转型为你所需的型别〕。

```
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];
};
```

重点是你可以在 ss_family 栏位看到地址家族（address family），检查它是 AF_INET 或 AF_INET6（是 IPv4 或 IPv6）。之後如果你愿意的话，你就可以将它转型为 sockaddr_in 或 struct sockaddr_in6。

3.4. IP 地址，Part II

还好你运气不错，有一堆函数让你能够控制 IP address，而不需要亲自用 long 与 << 运算符来处理它们。

咱们说，你有一个 struct sockaddr_in ina，而且你有一个 ”10.12.110.57” 或 ”2001:db8:63b3:1::3490” 这样的一个 IP address 要储存。你想要使用 inet_pton() 函数将 IP address 转换为数值与句号的符号，并依照你指定的 AF_INET 或 AF_INET6 来决定要储存在 struct in_addr 或 struct in6_addr。〔”pton”的意思是”presentation to network”，你可以称之为”printable to network”，如果这样会比较好记的话〕。

这样的转换可以用如下的方式：

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6
inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

〔小记：原本的老方法是使用名为 inet_addr() 的函数或另一个名为 inet_aton() 的函数；这些都过时了，而且不适合在 IPv6 中使用〕。
目前上述的代码片段还不是很可靠，因为没有错误检查。inet_pton() 在错误时会返回 -1，而若地址被搞砸了，则会返回 0。所以在使用之前要检查，并确认结果是大於 0 的。
好了，现在你可以将 IP address 字符串转换为它们的二进位表示。
还有其它方法吗？

如果你有一个 struct in_addr 且你想要以数字与句号印出来的话呢？
〔呵呵，或者如果你想要以”十六进制与冒号”打印出 struct in6_addr〕。在这个例子中，你会想要使用 inet_ntop()函数〔”ntop”意谓”network to presentation”－如果有比较好记的话，你可以称它为”network to prinable”〕，像是这样：

```
// IPv4:
char ip4[INET_ADDRSTRLEN]; // 储存 IPv4 字符串的空间
struct sockaddr_in sa; // pretend this is loaded with something
inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
printf("The IPv4 address is: %s\n", ip4);
// IPv6:
char ip6[INET6_ADDRSTRLEN]; // 储存 IPv6 字符串的空间
struct sockaddr_in6 sa6; // pretend this is loaded with something
inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);
printf("The address is: %s\n", ip6);
```

当你调用它时，你会传递地址的型别〔IPv4 或 IPv6〕，该地址是一个指向储存结果的字符串，与该字符串的最大长度。〔有两个 macro（宏）可以很方便地储存你想储存的最大 IPv4 或 IPv6 地址字符串大小：INET_ADDRSTRLEN 与 INET6_ADDRSTRLEN〕。

〔另一个要再次注意的是以前的方法：以前做这类转换的函数名为 inet_ntoa()，它已经过期了，而也在 IPv6 中也不适用〕。

最後，这些函数只能用在数值的 IP address 上，它们不需要 DNS nameserver 来查询主机名，如”www.example.com”。你可以使用 getaddrinfo() 来做这件事情，如同你稍後会看到的。

3.4.1 Private（或 disconnected）Network

很多地方都有防火墙（firewall），由它们的保护将网路隐藏於世界的其它地方。而有时，防火墙会用所谓的网路地址转换（NAT，Network Address Translation）的方法，将”internal”（内部的）IP 地址转换为”external”（外部的）〔世界上的每个人都知道的〕IP address。你又开始紧张了吗？”他又要扯到哪里去了？”

好啦，放轻松，去买瓶汽水〔或酒精〕饮料，因为身为一个初学者，你还可以不要理会 NAT，因为它所做的事情对你而言是透明的。不过我想在你开始对所见的网路数量开始感到困惑以前，谈谈防火墙身後的网路。

比如，我家有一个防火墙，我有两个 DSL 电信公司分配给我的 static IPv4 地址，而我家的网路有七部电脑要用。这有可能吗？两台电脑不能共用同一个 IP address 阿，不然数据就不知道该送去哪一台电脑了！

答案揭晓：它们不会共用同一个 IP address，它们是在一个配有两千四百万个 IP address 的 private network 上，全部都是我的。

好，都是我的，有这麼多地址可以让大家用来上网，而这里要讲的就是为什麼：

如果我登入到一台远端的电脑，它会说我从 192.0.2.33 登入，这是我的 ISP 提供给我的 public IP。不过若是我问我自己本地端节点的电脑，它的 IP address 是什麼时，他会说是 10.0.0.5。是谁转换 IP 的呢？答对了，就是防火墙！它做了 NAT！

10.x.x.x 是其中一个少数保留的网路，只能用在完全无法连上 Internet 的网路〔disconnected network〕，或是在防火墙後的网路。你可以使用哪个 private network 编号的细节是记在 RFC 1918 [15]中，不过一般而言，你较常见的是 10.x.x.x 及 192.168.x.x，这里的 x 是指 0-255。较少见的是 172.y.x.x，这里的 y 范围在 16 与 31 之间。

在 NAT 防火墙後的网路可以不必用这些保留的网路，不过它们通常会用。

〔真好玩！我的外部 IP 真的不是 192.0.2.33，192.0.2.x 网路保留用来虚构本教程要用的”真实”IP address，就像本教程也是虚构的一样 Wowzers！〕

IPv6 也很合理的会有 private network。它们最早的前缀是 fdxx:〔或者未来可能是 fcXX：〕，如同 RFC 4193 [16]。NAT 与 IPv6 通常不会混用，然而〔除非你在做 IPv6 到 IPv4 的 gateway，这就不在本教程的讨论范围内了〕，理论上，你会有很多地址可以使用，所以根本不再需要使用 NAT。不过，如果你想要在不会路由到外面的网路〔封闭网路〕上分配地址给你自己，就用 NAT 吧。

[15] http://tools.ietf.org/html/rfc1918

[16] http://tools.ietf.org/html/rfc4193

4. 从 IPv4 移植为 IPv6

”可是我只想知道要改代码的哪些地方就可以支援 IPv6 了！快点告诉我！”

OK！OK！

我几乎都是以过来人的身分来讲这边的每件事，这次不考验各位的耐心，来个简洁的短文。〔当然有更多比这篇短的文章，不过我是跟自己的这份教程来比较〕。

1. 首先，请试着用 getaddrinfo() 来取得 struct sockaddr 的资料，取代手动填写这个数据结构。这样你就可以不用管 IP 的版本，而且能省略之後很多步骤。

2. 找出全部与 IP 版本相关的任何代码，试着用一个有用的函数将它们包起来（wrap up）。

3. 将 AF_INET 更改为 AF_INET6。

4. 将 PF_INET 更改为 PF_INET6。

5. 将 INADDR_ANY 更改为 in6addr_any，这里有点不太一样：

```
struct sockaddr_in sa;
struct sockaddr_in6 sa6;

sa.sin_addr.s_addr = INADDR_ANY;     // 使用我的 IPv4 地址
sa6.sin6_addr = in6addr_any;   // 使用我的 IPv6 地址
```

还有，在宣告 struct in6_addr 时，IN6ADDR_ANY_INIT 的值可以做为初始值，像这样：

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

6. 使用 struct sockaddr_in6 取代 struct sockaddr_in，确定要将”6”新增到适当的栏位〔参考上面的 structs〕，但没有 sin6_zero 栏位。

7. 使用 struct in6_addr 取代 struct in_addr，要确定有将”6”新增到适当的栏位〔参考上面的 structs〕。

8. 使用 inet_pton() 替换 inet_aton() 或 inet_addr()。

9. 使用 inet_ntop() 替换 inet_ntoa()。

10. 使用很牛的 getaddrinfo() 取代 gethostbyname()。

11. 使用很牛的 getnameinfo() 取代 gethostbyaddr()〔虽然 gethostbyaddr()在 IPv6 中也能正常运作〕。

12. 不要用 INADDR_BROADCAST 了，请多使用 IPv6 multicast 来替换。

就是这样。

译注：
IPv6 可参考萩野纯一郎, IPv6 网路编程, 博硕, 2004.

5. System call 或 Bust

我们在本章开始讨论如何让你存取 UNIX 系统或 BSD 、 Windows 、 Linux 、 Mac 等系统的 system call（系统调用）、 socket API 及其它 function calls（函数调用）等网路功能。当你调用其中一个函数时，kernel 会接管，并且自动帮你处理全部的工作。

多数人会停滞在这里是因为不知道要用什麼样的顺序来调用这些函数，而你在找 man 手册时会觉得很难用。好，为了要帮忙解决这可怕的困境，我已经试着在下列的章节精确地布局（layout） system call，你在编程时只要照着一样的顺序调用就可以了。

为了要链接一些代码，需要一些牛奶跟饼乾〔这恐怕你要自行准备〕，以及一些决心与勇气，而你就能将数据发送到互联网上，彷佛是 Jon Postel 之子般。

〔请注意，为了简洁，下列许多代码片段并没有包含错误检查的代码。而且它们很喜欢假设调用 **getaddrinfo()** 的结果都会成功，并会返回链表（link-list）中的一个有效资料。这两种情况在单独运行的程序都有严谨的定位，所以，我们还是将它们当作模型来使用吧。〕

5.1. getaddrinfo()－准备开始！

这是个有很多选项（options）的工作马（workhorse）函数，但是却相当容易上手。它帮你设定之後需要的 struct。

谈点历史：它前身是你用来做 DNS 查询的 gethostbyname()。而当时你需要手动将资料写入 struct sockaddr_in，并在你的调用中使用。

感谢老天，现在已经不用了。〔如果你想要设计能通用於 IPv4 与 IPv6 的程序也不用！〕在现代，你有 getaddrinfo() 函数，可以帮你做许多事情，包含 DNS 与 service name 查询，并填好你所需的 structs。

让我们来看看！

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

int getaddrinfo(const char *node, // 例如： "www.example.com" 或 IP
const char *service, // 例如： "http" 或 port number
const struct addrinfo *hints,
struct addrinfo **res);
```

你给这个函数三个输入参数，结果它会返回一个指向链表的指针给你 — res。*node* 参数是要连接的主机名，或者一个 IP address（地址）。

下一个参数是 *service*，这可以是 port number，像是 ”80”，或者特定的服务名〔可以在你 UNIX 系统上的 IANA Port List [17] 或 /etc/services 文件中找到〕，像是 ”http” 或 ”ftp” 或 ”telnet” 或 ”smtp” 诸如此类的。

最後，*hints* 参数指向一个你已经填好相关资料的 struct addrinfo。

这里是一个调用示例，如果你是一部 server（服务器），想要在你主机上的 IP address 及 port 3490 运行 listen。

要注意的是，这边实际上没有做任何的 listening 或网路设置；它只有设置我们之後要用的 structures 而已。

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo; // 将指向结果

memset(&hints, 0, sizeof hints); // 确保 struct 为空
hints.ai_family = AF_UNSPEC; // 不用管是 IPv4 或 IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE; // 帮我填好我的 IP
```

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

// servinfo  目前指向一个或多个  struct addrinfos  的链表

// ...  做每件事情，一直到你不再需要  servinfo    ....

freeaddrinfo(servinfo); //  释放这个链表
```

注意一下，我将 *ai_family* 设置为 AF_UNSPEC，这样代表我不用管我们用的是 IPv4 或 IPv6 address。如果你想要指定的话，你可以将它设置为 AF_INET 或 AF_INET6。

还有，你会在这里看到 AI_PASSIVE flag；这个会告诉 getaddrinfo() 要将我本地端的地址（address of local host）指定给 socket structure。这样很好，因为你就不用写固定的地址了〔或者你可以将特定的地址放在 getaddrinfo() 的第一个参数中，我现在写 NULL 的那个参数〕。

然後我们进行调用，若有错误发生时〔getaddrinfo 会返回非零的值〕，如你所见，我们可以使用 gai_strerror() 函数将错误打印出来。若每件事情都正常运作，那麼 *serinfo* 就会指向一个 struct addrinfos 的链表，表中的每个成员都会包含一个我们之後会用到的某种 struct sockaddr。

最後，当我们终於使用 getaddrinfo() 配置的链表完成工作後，我们可以〔也应该〕要调用 freeaddrinfo() 将链表全部释放。

这边有一个调用示例，如果你是一个想要连接到特定 server 的 client，比如是：”www.example.net”的 port 3490。再次强调，这里并没有真的进行连接，它只是设置我们之後要用的 structure。

```
int status;
struct addrinfo hints;
```

```
struct addrinfo *servinfo; // 将指向结果

memset(&hints, 0, sizeof hints); // 确保 struct 为空
hints.ai_family = AF_UNSPEC; // 不用管是 IPv4 或 IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

// 准备好连接
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo 现在指向有一个或多个 struct addrinfos 的链表
```

我一直说 *serinfo* 是一个链表，它有各种的地址资料。让我们写一个能快速 demo 的程序，来呈现这个资料。这个小程序 [18] 会打印出你在命令行中所指定的主机 IP address：

```
/*
** showip.c -- 显示命令行中所给的主机 IP address
*/
#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; // AF_INET 或 AF_INET6 可以指定版本
    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;

        // 取得本身地址的指针，
        // 在 IPv4 与 IPv6 中的栏位不同：
        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;
            addr = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }
```

```
        // convert the IP to a string and print it:
        inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
        printf(" %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res); // 释放链表

    return 0;
}
```

如你所见，代码使用你在命令行输入的参数调用 getaddrinfo()，它填好 *res* 所指的链表，并接着我们就能重复那行并打印出东西或做点类似的事。

〔有点不好意思！我们在讨论 struct sockaddrs 它的型别差异是因 IP 版本而异之处有点鄙俗。我不确定是否有较优雅的方法。〕

在下面运行示例！来看看大家喜欢看的运行画面：

```
$ 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
```

现在已经在我们的掌控之下，我们会将 getaddrinfo() 返回的结果送给其它的 socket 函数，而且终於可以建立我们的网路连接了！

让我们继续看下去！

5.2. socket()－取得 File Descriptor！

我想可以不用再将 socket() 摆放在旁边凉快了，我一定要讲一下 socket() system call，这边是代码片段：

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

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

可是这些参数是什麼？

它们可以让你设定想要的 socket 类型〔IPv4 或 IPv6，stream 或 datagram 以及 TCP 或 UDP〕。

以前的人得将这些写入固定的值，而你也可以这样做。〔*domain* 是 PF_INET 或 PF_INET6，*type* 是 SOCK_STREAM 或 SOCK_DGRAM，而 *protocol* 可以设置为 0，用来帮给予的 type 选择适当的协议。或者你可以调用 getprotobyname() 来查询你想要的协议，”tcp” 或”udp” 〕。

〔PF_INET 就是你在初始化 struct sockaddr_in 的 *sin_family* 栏位会用到的，它是 AF_INET 的亲戚。实际上，它们的关系很接近，所以其实它们的值也都一样，而许多程序设计师会调用 socket()，并以 AF_INET 替换 PF_INET 来做为第一个参数传递。
现在，你可以去拿点牛奶跟饼乾，因为又是说故事时间了。

在很久很久以前，人们认为它应该是地址家族（address family），就是 ”AF_INET” 中的 ”AF” 所代表的意思；而地址家族也要支援协议家族（protocol family）的几个协议，这件事并没有发生，而之後它们都过着幸福快乐的日子，结束。

所以最该做的事情就是在你的 struct sockaddr_in 中使用 AF_INET，而在调用 socket() 时使用 PF_INET。〕

总之，这样就够了。你真的该做的只是使用调用 **getaddrinfo()** 得到的值，并将这个值直接提供给 socket()，像这样：

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

// 运行查询
// [假装我们已经填好 "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// [再来，你应该要对 getaddrinfo() 进行错误检查，并走到 "res" 链表查询能用的资料，
// 而不是假设第一笔资料就是好的〔像这些示例一样〕
// 实际的示例请参考 client/server 章节。
s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

socket() 单纯返回一个之後 system call 要用的 *socket descriptor* 给你，错误时会返回 -1。errno 全局变量会设置为该错误的值〔细节请见 errno 的 man 手册，而且你需要继续阅读并执行更多与它相关的 system call，这样心里会比较有谱。〕

5.3. bind()－What port am I on？

一旦你有了一个 socket，你会想要将这个 socket 与你本地端的 port 进行关联〔如果你正想要 listen() 特定 port 进入的连接，通常都会这样做，比如：多人网路连线游戏在它们告诉你”连接到 192.168.5.10 port 3490”时这麼做〕。port number 是用来让 kernel 可以比对出进入的数据包是属於哪个 process 的 socket descriptor。如果你只是正在进行 connect()〔因为你是 client，而不是 server〕，这可能就不用。不过还是可以读读，有趣嘛。

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

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

sockfd 是 socket() 传回的 socket file descriptor。*my_addr* 是指向包含你的地址资料、名称及 IP address 的 struct sockaddr 之指针。*addrlen* 是以 byte 为单位的地址长度。

呼！有点比较好玩了。我们来看一个示例，它将 socket bind（绑定）到运行程序的主机上，port 是 3490：

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

// 首先，用 getaddrinfo() 载入地址结构：

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);

// 建立一个 socket：

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

// 将 socket bind 到我们传递给 getaddrinfo() 的 port：

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

使用 AI_PASSIVE flag，我可以跟程序说要 bind 它所在主机的 IP。如果你想要 bind 到指定的本地端 IP address，舍弃 AI_PASSIVE，并改放一个地址到 getaddrinfo() 的第一个参数。

bind() 在错误时也会返回 -1，并将 errno 设置为该错误的值。

许多旧程序都在调用 bind() 以前手动封装 struct sockaddr_in。很显然地，这是 IPv4 才有的，可是真的没有办法阻止你在 IPv6 做一样的事情，一般来说，使用 getaddrinfo() 会比较简单。总之，旧版的程式看起来会像这样：

```
// !!! 这 是 老 方 法 !!!

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, network byte order
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);
```

在上列的代码中，如果你想要 bind 到你本地端的 IP address〔就像上面的 AI_PASSIVE flag〕，你也可以将 INADDR_ANY 指定给 s_addr 栏位。INADDR_ANY 的 IPv6 版本是一个 in6addr_any 全局变量，它会被指定给你的 struct sockaddr_in6 的 sin6_addr 栏位。〔也有一个你能用於 variable initializer（变量初始器）的 IN6ADDR_ANY_INIT macro（宏）〕

另一件调用 bind() 时要小心的事情是：不要用太小的 port number。全部 1024 以下的 ports 都是保留的〔除非你是系统管理员〕！你可以使用任何 1024 以上的 port number，最高到 65535〔提供尚未被其它程序使用的〕。

你可能有注意到，有时候你试着重新运行 server，而 bind() 却失败了，它声称”Address already in use.”（地址正在使用）。这是什麼意思呢？很好，有些连接到 socket 的连接还悬在 kernel 里面，而它占据了这个 port。你可以等待它自行清除〔一分钟之类〕，或者在你的程序中新增代码，让它重新使用这个 port，类似这样：

```
int yes=1;
//char yes='1'; // Solaris 的用户使用这个

// 可以跳过 "Address already in use" 错误信息

if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
perror("setsockopt");
```

```
exit(1);
}
```

最後一个对 bind() 的额外小提醒：在你不愿意调用 bind() 时。若你正使用 connect() 连接到远端的机器，你可以不用管 local port 是多少（以 telnet 为例，你只管远端的 port 就好），你可以单纯地调用 connect()，它会检查 socket 是否尚未绑定（unbound），并在有需要的时候自动将 socket bind() 到一个尚未使用的 local port。

5.4. connect()，嘿!你好。

咱们用几分钟的时间假装你是个 telnet 应用程序，你的用户命令你〔就像 TRON 电影里那样〕取得一个 socket file descriptor。你运行并调用 socket()。接着用户告诉你连接到 ”10.12.110.57” 的 port 23〔标准 telnet port〕。哟！你现在该做什麼呢？

你是很幸运的程序，你现在可以细读 connect() 的章节，如何连线到远端主机。所以努力往前读吧！刻不容缓！

connect() call 如下：

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

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

sockfd 是我们的好邻居 socket file descriptor，如同 socket() 调用所返回的，serv_addr 是一个 struct sockaddr，包含了目的 port 及 IP 地址，而 addrlen 是以 byte 为单位的 server 地址结构之长度。

全部的资料都可以从 getaddrinfo() 调用中取得，它很好用。

这样有开始比较有谱了吗？我在这里没办法知道，所以我只能希望是这样没错。

我们有个示例，这边我们用 socket 连接到 ”www.example.com” 的 port 3490：

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

```
// 首先，用 getaddrinfo() 载入 address structs：

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

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

// 建立一个 socket：

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

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

老学校的程序再次填满了它们自己的 struct sockaddr_ins 并传给 connect()。如果你愿意的话，你可以这样做。请见上面 bind() 章节中类似的提点。

要确定有检查 connect() 返回的值，它在错误时会返回 -1，并设定 errno 变量。

还要注意的是，我们不会调用 bind()。基本上，我们不用管我们的 local port number；我们只在意我们的目地〔远端 port〕。Kernel 会帮我们选择一个 local port，而我们要连接的站台会自动从我们这里取得资料，不用担心。

5.5. listen()－有人会调用我吗？

OK，是该改变步调的时候了。如果你不想要连接到一个远端主机要怎麼做。

我说过，好玩就好，你想要等待进入的连接，并以某种方式处理它们。

这个过程有两个步骤：你要先调用 listen()，接着调用 accept()〔参考下一节〕。

listen 调用相当简单，不过需要一点说明：

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

sockfd 是来自 socket() system call 的一般 socket file descriptor。*backlog* 是进入的队列（incoming queue）中所允许的连接数目。这代表什麼意思呢？好的，进入的连接将会在这个队列中排队等待，直到你 accept() 它们（请见下节），而这限制了排队的数量。多数的系统默认将这个数值限制为 20；你或许可以一开始就将它设置为 5 或 10。

再来，如同往常，listen() 会返回 -1 并在错误时设置 errno。

好的，你可能会想像，我们需要在调用 listen() 以前调用 bind()，让 server 可以在指定的 port 上运行。〔你必须能告诉你的好朋友要连接到哪一个 port！〕所以如果你正在 listen 进入的连接，你会运行的 system call 顺序是：

```
getaddrinfo();
socket();
bind();
listen();
/* accept() 从这里开始 */
```

我只是留下示例程序的位置，因为它相当显而易见。〔在下面 accept() 章节中的代码会比较完整〕。这整件事情真正需要技巧的部分是调用 accept()。

5.6. accept()－”谢谢你调用 port 3490。”

准备好，accept() 调用是很奇妙的！会发生的事情就是：很远的人会试着 connect() 到你的电脑正在 listen() 的 port。他们的连接会排队等待被 accept()。你调用 accept()，并告诉它要取得搁置的（pending）连接。它会返回专属这个连接的一个新 socket file descriptor 给你！那是对的，你突然有了两个 *socket file descriptor*！原本的 socket file descriptor 仍然正在 listen 之後的连线，而新建立的 socket file descriptor 则是在最後要准备给 send() 与 recv() 用的。

调用如下：

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

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

sockfd 是正在进行 listen() 的 socket descriptor。很简单，*addr* 通常是一个指向 local struct sockaddr_storage 的指针，关於进来的连接将往哪里去的资料〔而你可以用它来得知是哪一台主机从哪一个 port 调用你的〕。*addrlen* 是一个 local 的整数变量，应该在将它的地址传递给 accept() 以前，将它设置为 sizeof(struct sockaddr_storage)。accept() 不会存放更多的 bytes（字节）到 *addr*。若它存放了较少的 bytes 进去，它会改变 *addrlen* 的值来表示。

有想到吗？accept() 在错误发生时返回 -1 并设置 errno。不过 BetCha 不这麼认为。

跟以前一样，用一段代码示例会比较好吸收，所以这里有一段示例程供你细读：

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

#define MYPORT "3490" // 使用者将连接的 port
#define BACKLOG 10 // 在队列中可以有多少个连接在等待

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

    // !! 不要忘了帮这些调用做错误检查 !!

    // 首先，使用 getaddrinfo() 载入 address struct：

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // 使用 IPv4 或 IPv6，都可以
```

```
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // 帮我填上我的 IP

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

    // 产生一个 socket，bind socket，并 listen socket：

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

    // 现在接受一个进入的连接：

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

    // 准备好与 new_fd 这个 socket descriptor 进行沟通！
    .
    .
    .
```

一样，我们会将 *new_fd* socket descriptor 用於 send() 与 recv() 调用。若你只是要取得一个连接，你可以用 close() 关闭正在 listen 的 *sockfd*，以避免有更多的连接进入同一个 port，若你有这个需要的话。

5.7. send() 与 recv()— 宝贝，跟我说说话！

这两个用来通讯的函数是透过 stream socket 或 connected datagram ssocket。若你想要使用常规的 unconnected datagram socket，你会需要参考底下的 sendto() 及 recvfrom() 的章节。

send() 调用：

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

sockfd 是你想要送资料过去的 socket descriptor〔不论它是不是 socket() 返回的，或是你用 accept() 取得的〕。*msg* 是一个指向你想要传送资料之指标，而 *len* 是以 byte 为单位的资料长度。而 *flags* 设置为 0 就好。〔更多相关的 flag 资料请见 send() man 手册〕。

一些示例代码如下：

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

send() 会返回实际有送出的 byte 数目，这可能会少於你所要传送的数目！有时候你告诉 send() 要送整笔的资料，而它就是无法处理这麼多资料。它只会尽量将资料送出，并认为你之後会再次送出剩下没送出的部分。

要记住，如果 send() 返回的值与 *len* 的值不符合的话，你就需要再送出字串剩下的部分。好消息是：如果数据包很小〔比 1K 还要小这类的〕，或许有机会一次就送出全部的东西。

一样，错误时会返回 -1，并将 errno 设置为错误码（error number）。

recv() 调用在许多地方都是类似的：

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

sockfd 是要读取的 socket descriptor，*buf* 是要记录读到资料的缓冲区（buffer），*len* 是缓冲区的最大长度，而 *flags* 可以再设置为 0。〔关於 flag 资料的细节请参考 recv() 的 man 手册〕。

recv() 返回实际读到并写入到缓冲区的 byte 数目，而错误时返回 -1〔并设置相对的

errno〕。

等等！ recv() 会返回 0，这只能表示一件事情：远端那边已经关闭了你的连接！recv() 返回 0 的值是让你知道这件事情。

这样很简单，不是吗？你现在可以送回数据，并往 stream sockets 迈进！嘻嘻！你是 UNIX 网路程序员了。

5.8. sendto() 与 recvfrom()— 用 DGRAM 风格跟我说说话

我听到你说，”这全部都是上等的好货” ，” 可是我该如何使用 unconnected datagram socket 呢？”

没问题，朋友。我们正要讲这件事。

因为 datagram socket 没有连线到到远端主机，猜猜看，我们在送出数据包以前会需要哪些资料呢？

对！目的地址！在这里抢先看：

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

如你所见，这个调用基本上与调用 send() 一样，只是多了两个额外的资料。*to* 是一个指向 struct sockaddr〔这或许是另一个你可以在最後转型的 struct sockaddr_in 或 struct sockaddr_in6 或 struct sockaddr_storage〕的指针，它包含了目的 IP address 与 port。*tolen* 是一个 int，可以单纯地将它设置为 sizeof *to 或 sizeof(struct sockaddr_storage)。

为了能自动处理目的地址结构（destination address structure），你或许可以用底下的 getaddrinfo() 或 recvfrom()，或者你也可以手动填上。

如同 send()，sendto() 会返回实际已传送的资料数量（一样，可能会少於你要传送的资料量！）而错误时返回 -1。

recv() 与 recvfrom() 也是差不多的。recvfrom() 的对照如下：

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

一样，它跟 recv() 很像，只是多了两个栏位。*from* 是指向 local struct sockaddr_storage 的指针，这个数据结构包含了数据包来源的 IP address 与 port。*fromlen* 是指向 local int 的指针，应该要初始化为 sizeof *from 或是 sizeof(struct sockaddr_storage)。当函数返回时，*fromlen* 会包含实际上储存於 *from* 中的地址长度。

recvfrom() 返回接收的数据数目，或在发生错误时返回 -1〔并设置相对的 errno〕。

所以这里有个问题：为什麼我们要用 struct sockaddr_storage 做为 socket 的型别呢？为什麼不用 struct sockaddr_in 呢？

因为你知道的，我们不想要让自己绑在 IPv4 或 IPv6，所以我们使用通用的泛型 struct sockaddr_storage，我们知道这样有足够的空间可以用在 IPv4 与 IPv6。

〔所以 ... 这里有另一个问题：为什麼不是 struct sockaddr 本身就可以容纳任何地址呢？我们甚至可以将通用的 struct sockaddr_storage 转型为通用的 struct sockaddr！似乎没什麼关系又很累赘啊。答案是，它就是不够大，我猜在这个时候更动它会有问题，所以他们就弄了一个新的。〕

记住，如果你 connect() 到一个 datagram socket，你可以在你全部的交易中只使用 send() 与 recv()。socket 本身仍然是 datagram socket，而数据包仍然使用 UDP，但是 socket interface 会自动帮你增加目的与来源资料。

5.9. close() 与 shutdown()－从我面前消失吧！

呼！你已经成天都在 send() 与 recv()了。你正准备要关闭你 socket descriptor 的连接，这很简单，你只要使用常规的 UNIX file descriptor close() 函数：

```
close(sockfd);
```

这会避免对 socket 做更多的读写。任何想要对这个远端的 socket 进行读写的人都会收到错误。

如果你想要能多点控制 socket 如何关闭，可以使用 shutdown() 函数。它让你可以切断单向的通信，或者双向〔就像是 close() 所做的〕，这是函数原型：

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

sockfd 是你想要 shutdown 的 socket file descriptor，而 *how* 是下列其中一个值：

```
0 不允许再接收数据
1 不允许再传送数据
2 不允许再传送与接收数据〔就像 close()〕
```

shutdown() 成功时返回 0，而错误时返回 -1（设置相对的 errno）。

若你在 unconnected datagram socket 上使用 shutdown()，它只会单纯的让 socket 无法再进行 send() 与 recv() 调用〔要记住你只能在有 connect() 到 datagram socket 的时候使用〕。

重要的是 shutdown() 实际上没有关闭 file descriptor，它只是改变了它的可用性。如果要释放 socket descriptor，你还是需要使用 close()。

没了。

〔除了要记得的是，如果你用 Windows 与 Winsock，你应该要调用 closesocket() 而不是 close()。〕

5.10 getpeername()－你是谁？

这个函数很简单。

它太简单了，我几乎不想给它一个自己的章节，虽然还是给了。

getpeername() 函数会告诉你另一端连接的 stream socket 是谁，函数原型如下：

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

sockfd 是连接的 stream socket 之 descriptor，*addr* 是指向 struct sockaddr〔或 struct sockaddr_in〕的指针，这个数据结构储存了连线另一端的资料，而 *addrlen* 则是指向 int 的指针，应该将它初始化为 sizeof *addr 或 sizeof(struct sockaddr)。

函数在错误时返回 -1，并设置相对的 errno。

一旦你取得了它们的地址，你就可以用 inet_ntop()、getnameinfo() 或 gethostbyaddr() 印出或取得更多的资料。不过你无法取得它们的登录帐号。

〔好好好，如果另一台电脑运行的是 ident daemon 就可以。〕然而，这个已经超出本教程的范围，更多资料请参考 RFC 1413 [19]。

5.11 gethostname()－我是谁？

比 getpeername() 更简单的函数是 gethostname()，它会返回你运行程序的电脑名，这个名称之後可以用在 gethostbyname()，用来定义你本地端电脑的 IP address。

有什麼更有趣的吗？

我可以想到一些事情，不过这不适合 socket 编程，总之，下面是一段示例：

```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
```

参数很简单：*hostname* 是指向字符数组（array of chars）的指针，它会储存函数返回的主机名（hostname），而 *size* 是以 byte 为单位的主机名长度。

函数在运行成功时返回 0，在错误时返回 -1，并一样设置 errno。

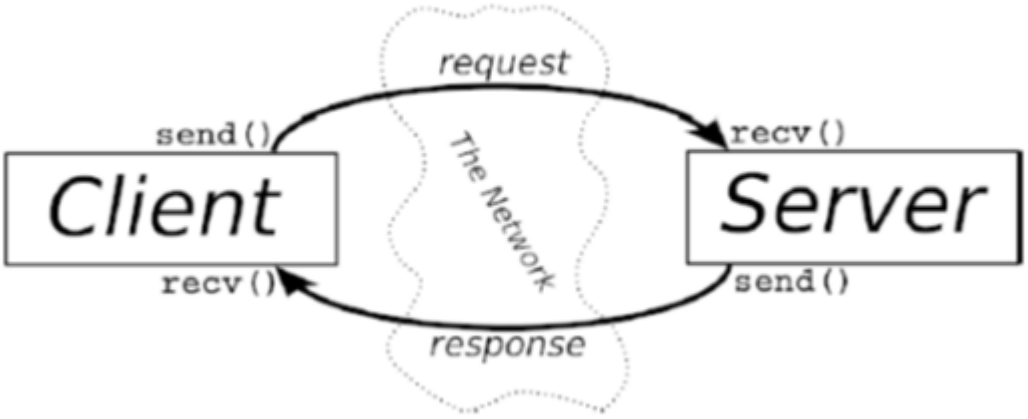
[17] http://www.iana.org/assignments/port-numbers

[18] http://beej.us/guide/bgnet/examples/showip.c

[19] http://tools.ietf.org/html/rfc1413

6. Client-Server 基础

宝贝，这是个 client-server（客户-服务器）的世界。单纯与网路处理 client processes（客户进程）及 server processes（服务器进程）通讯的每件事情有关，反之亦然。以 telnet 为例，当你用 telnet（client）连接到远端主机的 port 23 时，主机上的程序（称为 telnetd server）就开始动了起来，它会处理进来的 telnet 连接，并帮你设定一个登录提示符等。



Client-Server 互动

Client 与 server 间的信息交换摘录於上列的图解中。

需要注意的是 client-server pair 可以使用 SOCK_STREAM 、 SOCK_DGRAM 或其它的（只要它们用一样的协议来沟通）。有一些不错的 client-server pairs 示例，如：telnet/telnetd 、 ftp/ftpd 或 Firefox/Apache。每次你使用 ftp 时，都会有一个 ftpd 远端程序来为你服务。

一台设备上通常只会有一个 server，而该 server 会利用 fork()来处理多个 clients。基本的例程（routine）是：server 会等待连接、 accept() 连接，并且 fork() 一个 child process（子进程）来处理此连接。这就是我们在下一节的 server 示例所做的事情。

6.1. 一个简易的 Stream Server

这个 server 所做的事情就是透过 stream connection（串流连接）送出"Hello, World!\n"字符串。你所需要做就是用一个窗口来测试执行 server，并用另一个窗口来 telnet 到 server：

```
$ telnet remotehostname 3490
```

这里的 remotehostname 就是你运行 server 的主机名。

Server 的代码如下 [20]：

```
/*
** server.c -- 展示一个 stream socket server
*/
#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" // 提供给用戶连接的 port
#define BACKLOG 10 // 有多少个特定的连接队列（pending connections queue）

void sigchld_handler(int s)
{
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

// 取得 sockaddr，IPv4 或 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, new_fd; // 在 sock_fd 进行 listen，new_fd 是新的连接
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // 连接者的地址资料
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // 使用我的 IP

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

    // 以循环找出全部的结果，并绑定（bind）到第一个能用的结果
    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;
}

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

freeaddrinfo(servinfo); // 全部都用这个 structure

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

sa.sa_handler = sigchld_handler; // 收拾全部死掉的 processes
sigemptyset(&sa.sa_mask);
```

```
sa.sa_flags = SA_RESTART;

if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}

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

while(1) { // 主要的 accept() 循环

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()) { // 这个是 child process
        close(sockfd); // child 不需要 listener

        if (send(new_fd, "Hello, world!", 13, 0) == -1)
            perror("send");

        close(new_fd);
```

```
            exit(0);
        }
        close(new_fd); // parent 不需要这个
    }

    return 0;
}
```

趁着你对这个例子还感到很好奇，我为了让句子比较清楚（我个人觉得），所以将代码放在一个大的 main() 函数中，如果你觉得将它分成几个小一点的函数会比较好的话，可以尽管去做。

〔还有，sigaction() 这个东西对你而言应该是蛮陌生的。没有关系，这个代码是用来清理 zombie processes（僵尸进程），当 parent process 所 fork() 出来的 child process 结束时，且 parent process 没有取得 child process 的离开状态时，就会出现 zombie process。如果你产生了许多 zombies，但却无法清除他们时，你的系统管理员就会开始焦虑不安了〕。

你可以利用下一节所列出的 client，来取得 server 的资料。

6.2. 简易的 Stream Client

Client 这家伙比 server 简单多了，client 所需要做的就是：连线到你在命令行所指定的主机 3490 port，接着，client 会收到 server 送回的字符串。

Client 的代码 [21]：

```
/*
/*
** client.c -- 一个 stream socket client 的 demo
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

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

#define PORT "3490" // Client 所要连接的 port
#define MAXDATASIZE 100 // 我们一次可以收到的最大字节数量（number of bytes）

// 取得 IPv4 或 IPv6 的 sockaddr：
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;
    }

    // 用循环取得全部的结果，并先连接到能成功连接的
    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); // 全部皆以这个 structure 完成

    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;
}
```

要注意的是，你如果没有在运行 client 以前先启动 server 的话，connect()会返回 ”Connection refused” ，这很有帮助。

6.3. Datagram Sockets

我们已经在讨论 sendto()与 recvfrom()时涵盖了 UDP datagram socket 的基础，所以我会展示一对示范程序：talker.c 与 listener.c。

Listener 位於一台设备中，等待进入 port 4950 的数据包。Talker 则从指定的机器传送数据包给这个 port，数据包的内容包含用户从命令行所输入的资料。

这里就是 listener.c 的代码 [22]：

```
/*
** listener.c -- 一个 datagram sockets "server" 的 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 <arpa/inet.h>
#include <netdb.h>

#define MYPORT "4950" // 用户所要连线的 port
#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_UNSPEC; // 设定 AF_INET 以强制使用 IPv4
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE; // 使用我的 IP

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

// 用循环找出全部的结果，并 bind 到首先找到能 bind 的
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;
}
```

要注意的是，在我们调用 getaddrinfo()时，我们是使用 SOCK_DGRAM。还要注意到，不需要 listen()或是 accept()，这是使用（无连接）datagram sockets 的一个好处！

接着是 talker.c 的代码 [23]：

```
/*
** talker.c -- 一个 datagram "client" 的 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 <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT "4950" // 用户所要连接的 port

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_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;

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

// 用循环找出全部的结果，并产生一个 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 bind 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;
}
```

全部就这些了！在某个机器上运行 listener，接着在另一台机器运行 talker。观察它们的沟通！这真的超有趣。

这次你甚至不用运行 server！可以只运行 talker，而它只会很开心的将数据包丢到网路上，如果另一端没有人用 recvfrom()来接收的话，这些数据包就只是消失而已。

要记得：使用 UDP datagram socket 传送的数据是不会使命必达的！

我要再提之前提过无数次的小细节：connected datagram socket。我在这里要再讲一下，因为我们正在 datagram 这个章节。

我们说 talker 调用 connect()并指定 listener 的地址。从这开始，talker 就只能从 connect()所指定的地址进行传送与接收。因此，你不用使用 sendto()与 recvfrom()，可以单纯使用 send()与 recv() 就好。

[20] http://beej.us/guide/bgnet/examples/server.c

[21] http://beej.us/guide/bgnet/examples/client.c

[22] http://beej.us/guide/bgnet/examples/listener.c

[23] http://beej.us/guide/bgnet/examples/talker.c

7. 高等技术

本章的技术没有多高明，只是比前几章略高一筹。其实，如果你已经达到这个境界，你应该会觉得自己在 Unix 网路编程已经相当厉害！恭喜！

所以，我们要勇於面对你想学的 socket 新世界，这里会有些比较深奥的东西。

7.1. Blocking（阻塞）

你听过 blocking，只是它在这里代表什麼鬼东西呢？简而言之，”block” 就是 ”sleep（休眠）”的技术术语。你在以前运行 **listener** 时你可能有注意到，它只是在那边等，直到有数据包抵达。

很多函数都会 block，**accept()** 会 block，全部的 recv() 函数都会 block。原因是它们有权这麼做。当你先用 **socket()** 建立 socket descriptor 时，kernel（内核）会将它设置为 blocking。若你不想要 blocking socket，你必须调用 **fcntl()**：

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

将 socket 设置为 non-blocking（非阻塞），你就能 ” poll（轮询）” socket 以取得数据。如果你试着读取 non-blocking socket，而 socket 没有数据时，函数就不会发生 block，而是返回 -1，并将 errno 设置为 EWOULDBLOCK。

然而，一般来说，这样 polling 是不好的想法。如果你让程序一直忙着查 socket 上是否有数据，则会浪费 CPU 的时间，这样是不合适的。比较漂亮的解法是利用下一节

的 **select()** 来检查 socket 是否有数据需要读取。

7.2. select()－同步 I/O 多工

这个函数有点奇怪，不过它很好用。看看下面这个情况：如果你是一个 server，而你想要 listen 正在进来的连接，如同不断读取已建立的连接 socket 一样。

你说：没问题，只要用 **accept()** 及一对 **recv()** 就好了。

慢点，老兄！如果你在 **accept()** call 时发生了 blocking 该怎麼办呢？你要如何同时进行 **recv()** 呢？

”那就使用 non-blocking socket！”

不行！你不会想成为浪费 CPU 资源的罪人吧。

嗯，那有什麼好方法吗？

select() 授予你同时监视多个 sockets 的权力，它会告诉你哪些 sockets 已经有数据可以读取、哪些 sockets 已经可以写入，如果你真的想知道，还会告诉你哪些 sockets 触发了例外。

即使 **select()** 相当有可移植性，不过却是监视 sockets 最慢的方法。一个比较可行的替代方案是 libevent [24] 或者其它类似的方法，封装全部的系统相依要素，用以取得 socket 的通知。

好了，不罗唆，下面我提供了 **select()** 的原型：

```
#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);
```

这个函数以 *readfds* 、 *writefds* 及 *exceptfds* 监视 file descriptor（文件描述符）的 ” sets

（组）”。如果你想要知道你是否能读取 standard input（标准输入）及某个 *sockfd* socket descriptor，只要将 file descriptor 0 与 *sockfd* 新增到 *readfds* set 中。*numfds* 参数应该要设置为 file descriptor 的最高值加 1。在这个例子中，应该要将 *numfds* 设置为 *sockfd+1*，因为它必定大於 standard input（0）。

当 **select()** 回传时，*readfds* 会被修改，用来反映你所设置的 file descriptors 中，哪些已经有数据可以读取，你可以用下列的 **FD_ISSET()** macro（宏）来取得这些可读的 file descriptors。

在继续谈下去以前，我想要说说该如何控制这些 sets。
每个 sets 的型别都是 fd_set，下列是用来控制这个型别的 macro：

```
FD_SET(int fd, fd_set *set);        将 fd 新增到 set。
FD_CLR(int fd, fd_set *set);        从 set 移除 fd。
FD_ISSET(int fd, fd_set *set);      若 fd 在 set 中，返回 true。
FD_ZERO(fd_set *set);               将 set 整个清为零。
```

最後，这个令人困惑的 struct timeval 是什麼东西呢？

好，有时你不想要一直花时间在等人家送数据给你，或者明明没什麼事，却每 96 秒就要印出 ”运行中 ...” 到终端（terminal），而这个 time structure 让你可以设置 timeout 的周期。

如果时间超过了，而 **select()** 还没有找到任何就绪的 file descriptor 时，它会回传，让你可以继续做其它事情。

struct timeval 的栏位如下：

```
struct timeval {
   int tv_sec; // 秒（second）
   int tv_usec; // 微秒（microseconds）
};
```

只要将 *tv_sec* 设置为要等待的秒数，并将 *tv_usec* 设置为要等待的微秒数。是的，就是微秒，不是毫秒。一毫秒有 1,000 微秒，而一秒有 1,000 毫秒。所以，一秒就有 1,000,000 微秒。

为什麼要用 ”usec（微秒）” 呢？

”u” 看起来很像我们用来表示 ”micro（微）” 的希腊字母 μ（Mu）。还有，当函数回传时，会更新 *timeout*，用以表示还剩下多少时间。这个行为取决於你所使用的 Unix 而定。

译注：
因为有些系统平台的 select() 会修改 timeout 的值，而有些系统不会，所以如果要重复调用 select() 的话，每次都应该要重新设置 timeout 的值，以确保程序的行为可以符合预期。

哇！我们有微秒精度的计时器了！

是的，不过别依赖它。无论你将 struct timeval 设置的多小，你可能还要等待一小段 standard Unix timeslice（标准 Unix 时间片段）。

另一件有趣的事：如果你将 struct timeval 的栏位设置为 0，select() 会在轮询过 sets 中的每个 file descriptors 之後，就马上 timeout。如果你将 timeout 参数设置为 NULL，它就永远不会 timeout，并且陷入等待，直到至少一个 file descriptor 已经就绪（ready）。如果你不在乎等待时间，就在调用 **select()** 时将 timeout 参数设置为 NULL。

下列的代码片段 [25] 等待 2.5 秒後，就会出现 standard input（标准输入）所输入的东西：

```
/*
** select.c -- a select() demo
*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
```

```c
#include <unistd.h>
#define STDIN 0 // standard input 的 file descriptor
int main(void)
{
   struct timeval tv;
   fd_set readfds;

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

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

   // 不用管 writefds 与 exceptfds：
   select(STDIN+1, &readfds, NULL, NULL, &tv);

   if (FD_ISSET(STDIN, &readfds))
      printf("A key was pressed!\n");
   else
      printf("Timed out.\n");
   return 0;
}
```

如果你用一行缓冲区（buffer）的终端，那麼你从键盘输入数据後应该要尽快按下 Enter，否则程序就会发生 timeout。

你现在可能在想，这个方法用在需要等待数据的 datagram socket 上很好，而且你是对的：应该是不错的方法。

有些系统会用这个方式来使用 select()，而有些不行，如果你想要用它，你应该要参考你系统上的 man 使用手册说明看是否会有问题。

有些系统会更新 struct timeval 的时间，用来反映 select() 原本还剩下多少时间 timeout；不过有些却不会。如果你想要程序是可移植的，那就不要倚赖这个特性。〔如果你需要追踪剩下的时间，可以使用 **gettimeofday()**，我知道这很令人失望，不过事实就是这样。〕如果在 read set 中的 socket 关闭连接，会怎样吗？

好的，这个例子的 **select()** 回传时，会在 socket descriptor set 中说明这个 socket 是 "ready to read（就绪可读）" 的。而当你真的用 **recv()** 去读取这个 socket 时，**recv()** 则会回传 0 给你。这样你就能知道是 client 关闭连接了。

再次强调 **select()** 有趣的地方：如果你正在 listen() 一个 socket，你可以将这个 socket 的 file descriptor 放在 *readfds* set 中，用来检查是不是有新的连接。

我的朋友阿，这就是万能 **select()** 函数的速成说明。

不过，应观众要求，这里提供个有深度的范例，毫无疑问地，以前的简单范例和这个范例的难易度会有显着差距。不过你可以先看看，然後读後面的解释。

程序 [26] 的行为是简单的多用户聊天室 server，在一个窗口中运行 server，然後在其它多个窗口使用 **telnet** 连接到 server〔"**telnet hostname 9034**"〕。当你在其中一个 **telnet** session 中输入某些文字时，这些文字应该会在其它每个窗口上出现。

```
/*
** selectserver.c -- 一个 cheezy 的多人聊天室 server
*/

#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" // 我们正在 listen 的 port

// 取得 sockaddr，IPv4 或 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; // master file descriptor 表
    fd_set read_fds; // 给 select() 用的暂时 file descriptor 表
    int fdmax; // 最大的 file descriptor 数目

    int listener; // listening socket descriptor
    int newfd; // 新接受的 accept() socket descriptor
    struct sockaddr_storage remoteaddr; // client address
    socklen_t addrlen;

    char buf[256]; // 储存 client 数据的缓冲区
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];
```

```
int yes=1; // 供底下的 setsockopt() 设置 SO_REUSEADDR
int i, j, rv;

struct addrinfo hints, *ai, *p;

FD_ZERO(&master); // 清除 master 与 temp sets
FD_ZERO(&read_fds);

// 给我们一个 socket，并且 bind 它
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;
    }

    // 避开这个错误信息："address already in use"
    setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

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

```
    }

    break;
}

// 若我们进入这个判断式，则表示我们 bind() 失败
if (p == NULL) {
    fprintf(stderr, "selectserver: failed to bind\n");
    exit(2);
}
freeaddrinfo(ai); // all done with this

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

// 将 listener 新增到 master set
FD_SET(listener, &master);

// 持续追踪最大的 file descriptor
fdmax = listener; // 到此为止，就是它了

// 主要循环
for( ; ; ) {
    read_fds = master; // 复制 master

    if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
        perror("select");
        exit(4);
```

```
}

// 在现存的连接中寻找需要读取的数据
for(i = 0; i <= fdmax; i++) {
    if (FD_ISSET(i, &read_fds)) { // 我们找到一个！！
        if (i == listener) {
            // handle new connections
            addrlen = sizeof remoteaddr;
            newfd = accept(listener,
                (struct sockaddr *)&remoteaddr,
                &addrlen);

            if (newfd == -1) {
                perror("accept");
            } else {
                FD_SET(newfd, &master); // 新增到 master set
                if (newfd > fdmax) { // 持续追踪最大的 fd
                    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 {
            // 处理来自 client 的数据
            if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                // got error or connection closed by client
```

```
                    if (nbytes == 0) {
                        // 关闭连接
                        printf("selectserver: socket %d hung up\n", i);
                    } else {
                        perror("recv");
                    }
                    close(i); // bye!
                    FD_CLR(i, &master); // 从 master set 中移除

                } else {
                    // 我们从 client 收到一些数据
                    for(j = 0; j >= fdmax; j++) {
                        // 送给大家！
                        if (FD_ISSET(j, &master)) {
                            // 不用送给 listener 跟我们自己
                            if (j != listener && j != i) {
                                if (send(j, buf, nbytes, 0) == -1) {
                                    perror("send");
                                }
                            }
                        }
                    }
                }
            } // END handle data from client
        } // END got new incoming connection
    } // END looping through file descriptors
} // END for( ; ; )--and you thought it would never end!

    return 0;
}
```

我说过在代码中有两个 file descriptor sets：*master* 与 *read_fds*。前面的 *master* 记录全部

现有连接的 socket descriptors，与正在 listen 新连接的 socket descriptor 一样。

我用 *master* 的理由是因为 **select()** 实际上会改变你传送过去的 set，用来反映目前就绪可读（ready for read）的 sockets。因为我必须在在两次的 **select()** calls 期间也能够持续追踪连接，所以我必须将这些数据安全地储存在某个地方。最後，我再将 *master* 复制到 *read_fds*，并接着调用 **select()**。

可是这不就代表每当有新连接时，我就要将它新增到 *master set* 吗？是的！

而每次连接结束时，我们也要将它从 master set 中移除吗？是的，没有错。

我说过，我们要检查 listen 的 socket 是否就绪可读，如果可读，这代表我有一个待处理的连接，而且我要 **accept()** 这个连接，并将它新增到 *master* set。同样地，当 client 连接就绪可读且 **recv()** 返回 0 时，我们就能知道 client 关闭了连接，而我必须将这个 socket descriptor 从 master set 中移除。

若 client 的 **recv()** 返回非零的值，因而，我能知道 client 已经收到了一些数据，所以我收下这些数据，并接着到 master 清单，并将数据送给其它已连接的每个 clients。

我的朋友们，以上对万能 **select()** 函数的概述，这真是不简单的事情。

另外，这里有个福利：一个名为 **poll** 的函数，它的行为与 **select()** 很像，但是在管理 file descriptor sets 时用不一样的系统，你可以看看 poll。

参考资料

[24] http://www.monkey.org/~provos/libevent/

[25] http://beej.us/guide/bgnet/examples/select.c

[26] http://beej.us/guide/bgnet/examples/selectserver.c

7.3. 不完整传送的後续处理

还记得前面的 **send()** 章节吗？当时我不是提过 **send()** 可能不会将你所要求的数据全部送出吗？也就是说，虽然你想要送出 512 bytes，但是 send() 只送出 412 bytes。那剩下的 100 个 bytes 到哪去了呢？

好的，其实它们还在你的缓冲区里。因为环境不是你能控制的，kernel 会决定要不要用一个 chunk 将全部的数据送出，而现在，我的朋友，你可以决定要如何处理缓冲区中剩下的数据。

你可以写一个像这样的函数：

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

int sendall(int s, char *buf, int *len)
{
    int total = 0; // 我们已经送出多少 bytes 的数据
    int bytesleft = *len; // 我们还有多少数据要送
    int n;

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

    *len = total; // 返回实际上送出的数据量

    return n==-1?-1:0; // 失败时返回 -1 、成功时返回 0
}
```

在这个例子里，*s* 是你想要传送数据的 socket，*buf* 是储存数据的缓冲区，而 *len* 是一个指针，指向一个 int 型别的变数，记录了缓冲区中的数据数量。

函数在错误时返回 -1〔而 *errno* 仍然从调用 **send()** 设置〕。还有，实际送出的数据数量会在 *len* 中回传，除非有错误发生，不然这会跟你所要求要传送的数据量相同。**sendall()** 会尽力将数据送出，不过如果有错误发生时，它就会立刻回传给你。
为了完整性，这边有一个调用函数的示例：

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

```
len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("We only sent %d bytes because of the error!\n", len);
}
```

当数据包的一部分抵达接收端（receiver end）时会发生什麼事情呢？如果数据包的长度是会变动的（variable），接收端要如何知道另一端的数据包何时开始与结束呢？

是的，你或许必须封装（encapsulate）〔还记得数据封装（data encapsulation）章节的开头那边吗？那边有详细说明〕。

7.4. Serialization：如何封装数据

要将文字数据透过网路传送很简单，你已经知道了，不过如果你想要送一些 ”二进制” 的数据，如 int 或 float，会发生什麼事情呢？这里有一些选择。

1. 将数字转换为文字，使用如 **sprintf()** 的函数，接着传送文字。接收者会使用如 **strtol()** 函数解析文字，并转换为数字。
2. 直接以原始数据传送，将指向数据的指针传递给 **send()**。
3. 将数字编码（encode）为可移植的二进制格式，接收者会将它译码（decode）。

先睹为快！只在今晚！

〔序幕〕
Beej 说：”我偏好上面的第三个方法！”
〔结束〕

（在我热切开始本章节之前，我应该要跟你说有现成的程序库可以做这件事情，而要自制个可移植及无错误的作品会是相当大的挑战。所以在决定要自己实作这部分时，可以先四处看看，并做完你的家庭作业。我在这里引用些类似这个作品的有趣的资料。）

实际上，上面全部的方法都有它们的缺点与优点，但是如我所述，通常我偏好第三个方法。首先，咱们先谈谈另外两个的优缺点。

第一个方法，在传送以前先将数字编码为文字，优点是你可以很容易打印出及读取来自网路的数据。有时，人类易读的协定比较适用於频带不敏感（non-bandwidth-intensive）的情况，例如：Internet Relay Chat（IRC）[27]。然而，缺点是转换耗时，且总是需要比原本的数字使用更多的空间。

第二个方法：传送原始数据（raw data），这个方法相当简单〔但是危险！〕：只要将数据指针提供给 send()。

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0); /* 危险，不具可移植性！ */
```

接收者类似这样接收：

```
double d;

recv(s, &d, sizeof d, 0); /* 危险，不具可移植性！ */
```

快速又简单，那有什麼不好的呢？

好的，事实证明不是全部的架构都能表示 double〔或 int〕。〔嘿！或许你不需要可移植性，在这样的情况下这个方法很好，而且快速。〕

当封装整数型别时，我们已经知道 **htons()** 这类的函数如何透过将数字转换为 Network Byte Order（网路字节顺序），来让东西可以移植。毫无疑问地，没有类似的函数可以供 float 型别使用，全部的希望都落空了吗？

别怕！〔你有担心了一会儿吗？没有吗？一点都没有吗？〕

我们可以做件事情：我们可以将数据封装为接收者已知的二进制格式，让接收着可以在远端解压。

我所谓的 ”已知的二进制格式” 是什麼意思呢？

好的，我们已经看过了 **htons()** 范例了，不是吗？它将数字从 host 格式改变〔或是 ”编码” 〕为 Network Byte Order 格式；如果要反转〔译码〕这个数字，接收端会调用 **ntohs()**。

可是我不是才刚说过，没有这样的函数可供非整数型别使用吗？

是的，我说过。而且因为 C 语言并没有规范标准的方式来做，所以这有点麻烦〔that a gratuitous pun there for you Python fans〕。

要做的事情是将数据封装到已知的格式，并透过网路送出。例如：封装 float，这里的东西有很大的改善空间：[28]

```c
#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); // whole part and sign
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // fraction

    return p;
}

float ntohf(uint32_t p)
{
    float f = ((p>>16)&0x7fff); // whole part
    f += (p&0xffff) / 65536.0f; // fraction

    if (((p>>31)&0x1) == 0x1) { f = -f; } // sign bit set

    return f;
}
```

上列的代码是一个 native（原生的）实作，将 float 储存为 32-bit 的数字。High bit（高比特）〔31〕用来储存数字的正负号〔”1”表示负数〕，而接下来的七个比特〔30-16〕是用来储存 float 整个数字的部分。最後，剩下的比特〔15-0〕用来储存数字的小数（fractional portion）部分。

使用方式相当直觉：

```c
#include <stdio.h>

int main(void)
```

```
{
   float f = 3.1415926, f2;
   uint32_t netf;

   netf = htonf(f); // 转换为 "network" 形式
   f2 = ntohf(netf); // 转回测试

   printf("Original: %f\n", f); // 3.141593
   printf(" Network: 0x%08X\n", netf); // 0x0003243F
   printf("Unpacked: %f\n", f2); // 3.141586

   return 0;
}
```

好处是：它很小、很简单且快速，缺点是：它在空间的使用没有效率，而且对范围有严格的限制－试着在那边储存一个大於 32767 的数，它就会不高兴！

你也可以在上面的例子看到，最後一对的十进位空间并没有正确保存。
我们该怎麼改呢？

好的，用来储存浮点数（float point number）的标准方式是已知的 IEEE-754 [29]。多数的电脑会在内部使用这个格式做浮点运算，所以在这些例子里，严格说来，不需要做转换。但是如果你想要你的代码具可移植性，就要假设你不需要转换。〔换句话说，如果你想要让程序很快，你应该要在不需要做转换的平台上进行最佳化！这就是 **htons()** 与它的家族使用的方法。〕

这边有段代码可以将 float 与 double 编码为 IEEE-754 格式 [30]。〔主要的功能，它不会编码 NaN 或 Infinity，不过可以将它改成可以。〕

```
#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; // -1 for sign bit

    if (f == 0.0) return 0; // get this special case out of the way

    // 检查正负号并开始正规化
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // 取得 f 的正规化型式并追踪指数
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // 计算有效位数数据的二进制格式（非浮点数）
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // get the biased exponent
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // 返回最後的解答
    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; // -1 for sign bit

    if (i == 0) return 0.0;
```

```
    // pull the significand

    result = (i&((1LL<<significandbits)-1)); // mask
    result /= (1LL<<significandbits); // convert back to float
    result += 1.0f; // add the one back on

    // deal with the exponent
    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++; }

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

    return result;
}
```

我在那里的顶端放一些方便的 macro 用来封装与解除封装 32-bit〔或许是 float〕与 64-bit〔或许是 double〕的数字，但是 **pack754()** 函数可以直接调用，并告知编码几个比特的数据〔expbits 的哪几个比特要保留给正规化数值的指数。〕

这里是使用范例：

```
#include <stdio.h>
#include <stdint.h> // 定义 uintN_t 型别
#include <inttypes.h> // 定义 PRIx macros

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;
}
```

上面的代码会产生下列的输出：

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

你可能遭遇的另一个问题是你该如何封装 struct 呢？

对你来说没有问题的，编译器会自动将一个 struct 中的全部空间填入。〔你不会病到听成”不能这样做”、”不能那样做”？抱歉！引述一个朋友的话：”当事情出错了，我都会责怪 Microsoft。”这次固然可能不是 Microsoft 的错，不过我朋友的陈述完全符合事实。〕

回到这边，透过网路送出 struct 的最好方式是将每个栏位独立封装，并接着在它们抵达另一端时，将它们解封装到 struct。

你正在想，这样要做很多事情。是的，的确是。一件你能做的事情是写一个有用的函数来帮你封装数据，这很好玩！真的！

在 Kernighan 与 Pike 着作的 ”The Practice of Programming”[31] 这本书，他们实作类似 **printf()** 的函数，名为 **pack()** 与 **unpack()**，可以完全做到这件事。我想要连结到这些函数，但是这些函数显然地无法从网路上取得。

〔The Practice of Programming 是值得阅读的好书，Zeus saves a kitten every time I recommend it。〕

此时，我正要舍弃指向我从未用过的 BSD 授权类型的参数语言 C API（BSD-licensed Typed Parameter Language C API）[32] 的指针，可是看起来整个很可敬。Python 与 Perl 程序设计师想要找出他们语言的 **pack()** 与 **unpack()** 函数，用来完成同样的事情。而 Java 有一个能用於同样用途的 big-ol' Serializable interface。

不过如果你想要用 C 写你自己的封装工具，K&P 的技巧是使用变动参数列（variable argument list），来让类似 **printf()** 的函数建立数据包。我自己所编造的版本 [33] 希望足以供你了解这样的东西是如何运作的。

〔这段代码参考到上面的 **pack754()** 函数，**packi*()** 函数的运作方式类似 **htons()** 家族，除非它们是封装到一个 char 数组（array）而不是另一个整数。〕

```
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

// 供浮点数型别的变动比特
// 随着架构而变动

typedef float float32_t;
typedef double float64_t;

/*
** packi16() -- store a 16-bit int into a char buffer (like htons())
*/
```

```
void packi16(unsigned char *buf, unsigned int i)
{
   *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- store a 32-bit int into a char buffer (like htonl())
*/
void packi32(unsigned char *buf, unsigned long i)
{
   *buf++ = i>>24; *buf++ = i>>16;
   *buf++ = i>>8; *buf++ = i;
}

/*
** unpacki16() -- unpack a 16-bit int from a char buffer (like ntohs())
*/
unsigned int unpacki16(unsigned char *buf)
{
   return (buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- unpack a 32-bit int from a char buffer (like ntohl())
*/
unsigned long unpacki32(unsigned char *buf)
{
   return (buf[0]<<24) | (buf[1]<<16) | (buf[2]<<8) | buf[3];
}

/*
** pack() -- store data dictated by the format string in the buffer
**
** h - 16-bit l - 32-bit
```

```
** c - 8-bit char f - float, 32-bit
** s - string (16-bit length is automatically prepended)
*/
int32_t pack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t h;
    int32_t l;
    int8_t c;
    float32_t f;
    char *s;
    int32_t size = 0, len;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            size += 2;
            h = (int16_t)va_arg(ap, int); // promoted
            packi16(buf, h);
            buf += 2;
            break;

        case 'l': // 32-bit
            size += 4;
            l = va_arg(ap, int32_t);
            packi32(buf, l);
            buf += 4;
            break;

        case 'c': // 8-bit
            size += 1;
            c = (int8_t)va_arg(ap, int); // promoted
```

```
            *buf++ = (c>>0)&0xff;
            break;

        case 'f': // float
            size += 4;
            f = (float32_t)va_arg(ap, double); // promoted
            l = pack754_32(f); // convert to IEEE 754
            packi32(buf, l);
            buf += 4;
            break;

        case 's': // string
            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() -- unpack data dictated by the format string into the buffer
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t *h;
```

```
int32_t *l;
int32_t pf;
int8_t *c;
float32_t *f;
char *s;
int32_t len, count, maxstrlen=0;

va_start(ap, format);

for(; *format != '\0'; format++) {
   switch(*format) {
   case 'h': // 16-bit
      h = va_arg(ap, int16_t*);
      *h = unpacki16(buf);
      buf += 2;
      break;

   case 'l': // 32-bit
      l = va_arg(ap, int32_t*);
      *l = unpacki32(buf);
      buf += 4;
      break;

   case 'c': // 8-bit
      c = va_arg(ap, int8_t*);
      *c = *buf++;
      break;

   case 'f': // float
      f = va_arg(ap, float32_t*);
      pf = unpacki32(buf);
      buf += 4;
      *f = unpack754_32(pf);
      break;
```

```
        case 's': // string
            s = va_arg(ap, char*);
            len = unpacki16(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)) { // track max str len
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

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

    va_end(ap);
}
```

不管你是自己写的程序，或者用别人的代码，基於持续检查 bugs 的理由，能有通用的数据封装机制集是个好主意，而且不用每次都手动封装每个 bit（比特）。
在封装数据时，使用哪种格式会比较好呢？

好问题，很幸运地，RFC 4506 [35]，the External Data Representation Standard 已经定义了一堆各类型的二进位格式，如：浮点数型别、整数型别、数组、原始数据等。如果你打算自己写程序来封装数据，我建议要与标准符合。只是不会强制你一定要这样做。数据包的政策不会刚好就在你门口。至少，我不认为它们会在。

不管怎样，在你送出数据以前，以某种或其它方法将数据编码是对的做事方法。
[27] http://en.wikipedia.org/wiki/Internet_Relay_Chat
[28] http://beej.us/guide/bgnet/examples/pack.c
[29] http://en.wikipedia.org/wiki/IEEE_754

[30] http://beej.us/guide/bgnet/examples/ieee754.c
[31] http://cm.bell-labs.com/cm/cs/tpop/
[32] http://tpl.sourceforge.net/
[33] http://beej.us/guide/bgnet/examples/pack2.c

7.5. 数据封装之子

不管怎样，意思真的是封装数据吗？以最简单的例子而言，这表示你会需要增加一个 header（标头），用来代表识别的资料或数据长度，或者都有。

你的 header 看起来像什麼呢？

好的，它就只是某个用来表示你觉得完成专案会需要的二进制数据。

哇，好抽象。

Okay，举例来说，咱们说你有一个使用 SOCK_STREAM 的多重用户聊天程序。当某个用户输入〔” says” 〕某些字，会有两笔资料要传送给 server：

” 谁” 以及” 说了什麼” 。

到目前为止都还可以吗？

你问：” 会有什麼问题吗？”

问题是讯息的长度是会变动的。一个叫做 ” tom” 的人可能会说 ” Hi（嗨）” ，而另一个叫做” Benjamin（班杰明）” 的人可能说：” Hey guys what is up？（嘿！兄弟最近你好吗？）”

所以你在收到全部的数据之後，将它全部 **send()** 给 clients。你输出的 data stream（数据串流）类似这样：

```
tomHiBenjaminHeyguyswhatisup?
```

类似这样。那 client 要如何知道讯息何时开始与结束呢？

如果你愿意，是可以的，只要让全部的讯息都一样长，并只要调用我们之前实作的 **sendall()** 就行了。但是这样会浪费带宽（bandwidth）！我们并不想用 **send()** 送出了 1024 个 bytes 的数据，却只有携带了 ” tome” 说了 ” Hi” 这样的有效资料。

所以我们以小巧的 header 与数据包结构封装（encapsulate）数据。Client 与 server 都知道如何封装（pack）与解封装（unpack）这笔数据〔有时候称为 ”marshal” 与 ”unmarshal”〕。现在先不要想太多，我们会开始定义一个协议（protocol），用来描述 client 与 server 是如何沟通的！

在这个例子中，咱们假设用户的名称是固定 8 个字元，并用 '\0' 结尾。然後接着让我们假设数据的长度是变动的，最多高达 128 个字元。我们看个可能在这个情况会用到的数据包结构范例。

1. len〔1 个 byte，unsigned（无号）〕：数据包的总长度，计算 8 个 bytes 的用户名称，以及聊天数据。
2. name〔8 个 bytes〕：用户名称，如果有需要，结尾补上 NUL。
3. chatdata〔n 个 bytes〕：数据本身，最多 128 bytes。数据包的长度应该要以这个数据长度加 8〔上面的 name 栏位长度〕来计算。

为什麼我选择 8 个 bytes 与 128 个 bytes 长度的栏位呢？我假设这样就已经够用了，或许，8 个 bytes 对你的需求而言太少了，你也可以有 30 个 bytes 的 name 栏位，总之，你可以自己决定！

使用上列的数据包定义，第一个数据包由下列的资料组成〔以 hex 与 ASCII〕：

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

而第二个也是差不多：

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

〔长度（length）是以 Network Byte Order 储存，当然，在这个例子只有一个 byte，所以没差，但是一般而言，你会想要让你全部的二进制整数能以 Network Byte Order 储存在你的数据包中。〕

当你传送数据时，你应该要谨慎点，使用类似前面的 **sendall()** 指令，因而你可以知道全部的数据都有送出，即便要将数据全部送出会多花几次的 **send()**。

同样地，当你接收这笔数据时，你需要额外做些处理。如果要保险一点，你应该假设你可能只会收到部分的数据包内容〔如我们可能会从上面的班杰明那里收到 ”18 42 65 6E 6A”〕，但是我们这次调用 **recv()** 全部就只收到这些数据。我们需要一次又一次的调用 **recv()**，直到完整地收到数据包内容。

可是要怎麼做呢？

好的，我们可以知道所要接收的数据包它全部的 byte 数量，因为这个数量会记载在数据包前面。我们也知道最大的数据包大小是 1 + 8 + 128，或者 137 bytes〔因为这是我们自己定义的〕。

实际上你在这边可以做两件事情，因为你知道每个数据包是以长度（length）做开头，所以你可以调用 **recv()** 只取得数据包长度。接着，你知道长度以後，你就可以再次调用 **recv()**，这时候你就可以正确地指定剩下的数据包长度〔或者重复取得全部的数据〕，直到你收到完整的数据包内容为止。这个方法的优点是你只需有一个足以存放一个数据包的缓冲区，而缺点是你为了要接收全部的数据，至少调用两次的 **recv()**。

另一个方法是直接调用 **recv()**，并且指定你所要接收的数据包之最大数据量。这样的话，无论你收到多少，都将它写入缓冲区，并最後检查数据包是否完整。当然，你可能会收到下一个数据包的内容，所以你需要有足够的空间。

你所能做的是宣告（declare）一个足以容纳两个数据包的阵列，这是你在数据包到达时，你可以重新建构（reconstruct）数据包的地方。

每次你用 **recv()** 接收数据时，你会将数据接在工作缓冲区（work buffer）的後端，并检查数据包是否完整。在缓冲区中的数据数量大於或等於 数据包 header 中所指定的长度时〔+1，因为 header 中的长度没有包含 length 本身的长度〕。若缓冲区中的数据长度小於 1，那麼很明显地，数据包是不完整的。你必须针对这种情况做个特别处理，因为第一个 byte 是垃圾，而你不能用它来取得正确的数据包长度。

一旦数据包已经完整接收了，你就可以做你该做的处理，将数据拿来使用，并在用完之後将它从工作缓冲区中移除。

呼呼！Are you juggling that in your head yet？

好的，这里是第二次的冲击：你可能在一次的 **recv()** call 就已经读到了一个数据包的结尾，还读到下一个数据包的内容，即是你的工作缓冲区有一个完整的数据包，以及下一个数据包的一部分！该死的家伙。〔但是这就是为什麼你需要让你的工作缓冲区可以容纳两个数据包的原因，就是会发生这种情况！〕

因为你从 header 得知第一个数据包的长度，而你也有持续追踪工作缓冲区的数据量，所以你可以相减，并且计算出工作缓冲区中有多少数据是属於第二个〔不完整的〕数据包的。当你处理完第一个数据包後，你可以将第一个数据包的数据从工作缓冲区中清掉，并将第二个数据包的部分内容移到缓冲区的前面，准备进行下一次的 **recv()**。

〔部分读者会注意到，实际地将第二个数据包的部份数据移动到缓冲区的开头需要花费时间，而程序可以写成利用环状缓冲区（circular buffer），就不需要这样做。如果你还是很好奇，可以找一本数据结构的书来读。〕

我从未说过这很简单，好吧，我有说过这很简单。而你所需要的只是多练习，然後很快的你就会习惯了。我发誓！

7.6. 广播数据包（Broadcast Packet）：Hello World！

到了这里，本文已经谈了如何将数据从一台主机传送到另一台主机。但是，我坚持你可能会需要究极的权力，同时将数据送给多个主机！

用 UDP〔只能是 UDP，TCP 不行〕与标准的 IPv4，可以透过一种叫作广播（broadcasting）的机制达成。IPv6 不支援广播，所以你必须要采用比较高级的技术－群播（multicasting），很遗憾地，我现在不会讨论这个，我受够了异想天开的未来，我们现在还停留在 32-bit 的 IPv4 世界呢！

可是，请等一下！不管你愿不愿意，你不能走呀，开始说说广播吧。

你必须在将广播数据包送到网路之前，先设置 SO_BROADCAST socket 选项。这类似一个推送导弹开关的小塑胶盖！就只是你的手上掌握了多少的权力。

不过认真说来，使用广播数据包是很危险的，因为每个收到广播数据包的系统都要拨开一层层的数据封装，直到系统知道这笔数据是要送给哪个 port 为止。然後系统会开始处理这笔数据或者丢掉它。在另一种情况，对每部收到广播数据包的机器而言这很费工，因为他们都在同一个区域网路（local network），这样会让很多电脑做不少多馀的工作。当 Doom 游戏出现时，就有人在说它的网路程序写的不好。

现在，有很多方法可以解决这个问题 ...

等一下，真的有很多方法吗？

那是什麼表情阿？哎呀，一样阿，送广播数据包的方法很多。所以重点就是：你该如何指定广播讯息的目地地址呢？

有两种常见的方法：

1. 将数据送给子网路（subnet）的广播地址，就是将 subnet's network（子网路网段）的 host（主机）那部分全部填 1，举例来说，我家里的网路是 192.168.1.0，而我的 netmask（网路遮罩）是 255.255.255.0，所以地址的最後一个 byte 就是我的 host number〔因为依据 netmask，前三个 bytes 是 network number〕。所以我的广播地址就是 192.168.1.255。在 Unix 底下，**ifconfig** 指令实际上都会给你这些数据。〔如果你有兴

趣，取得你广播地址的逻辑运算方式是 network_number OR (Not netmask) 〕。你可以用跟区域网路一样的方式，将这类型的广播数据包送到远端网路（remote network），不过风险是数据包可能会被目地端的 router（路由器）丢弃。〔如果 router 没有将数据包丢弃，那麼有个随机的蓝色小精灵会开始用广播流量对它们的区域网路造成水灾。〕

2. 将数据送给 "global（全局的）"广播地址，255.255.255.255，又称为 INADDR_BROADCAST，很多机器会自动将它与你的 network number 进行 AND bitwise，以转换为网路广播地址，但是有些机器不会这样做。Routers 不会将这类的广播数据包转送（forward）出你的区域网路，够讽刺的。

所以如果你想要将数据送到广播地址，但是没有设置 SO_BROADCAST socket 选项时会怎样呢？好，我们用之前的 **talker** 与 **listener** 来炒冷饭，然後看看会发生什麼事情。

```
$ 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
```

是的，没有很顺利 ... 因为我们没有设置 SO_BROADCAST socket 选项，设置它，然後现在你就可以用 **sendto()** 将数据送到你想送的地方了！

事实上，这就是 UDP 应用程序能不能广播的差异点。所以我们改一下旧的 **talker** 应用程序，设置 SO_BROADCAST socket 选项。这样我们就能调用 broadcaster.c 程序了 [36]：

```
/*
** broadcaster.c -- 一个类似 talker.c 的 datagram "client"，
**  差异在於这个可以广播
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```c
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950 // 所要连接的 port

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // 连接者的地址资料
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // 如果上面这行不能用的话，改用这行

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

    if ((he=gethostbyname(argv[1])) == NULL) { // 取得 host 资料
        perror("gethostbyname");
        exit(1);
    }

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

    // 这个 call 就是要让 sockfd 可以送广播数据包
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
```

```
        exit(1);
    }

    their_addr.sin_family = AF_INET; // host byte order
    their_addr.sin_port = htons(SERVERPORT); // short, network byte order
    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;
}
```

这个跟 ”一般的” UDP client/server 有什麼不同呢？

没有！〔除了 client 可以送出广播数据包〕

同样地，我们继续，并在其中一个窗口运行旧版的 UDP **listener** 程序，然後在另一个窗口运行 **broadcaster**，你应该可以顺利运行了。

```
$ 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
```

而你应该会看到 **listener** 回应说它已经收到数据包。〔如果 **listener** 没有回应，可能是因为它绑到 IPv6 地址了，试着将 listener.c 中的 AF_UNSPEC 改成 AF_INET，强制使用

IPv4〕。
好，真令人兴奋，可是现在要在同一个网路上的另一台电脑运行 **listener**，所以你会有两个复本正在运行，每个机器上各有一个，然後再次用你的广播地址来运行 **broadcaster** ... 嘿！你只有调用一次 **sendto()**，但是两个 **listeners** 都收到了你的数据包。酷喔！

如果 **listener** 收到你直接送给它的数据，但不是在广播地址没有数据，可能是因为你本地端（local machine）上有防火墙（firewall）封锁了这些数据包。〔是的，谢谢 Pat 与 Bapper 的说明，让我知道为什麼我的范例程序无法运作。我跟你们说过我会在文件中提到你们，就是这里了，感恩。〕

再次提醒，使用广播数据包一定要小心，因为 LAN 上面的每台电脑都会被迫处理这类数据包，无论它们有没有用 **recvfrom()** 接收，这类数据包会造成整个电脑网路相当大的负担，所以一定要谨慎、适当地使用广播。

[34] http://beej.us/guide/bgnet/examples/pack2.c
[35] http://tools.ietf.org/html/rfc4506
[36] http://beej.us/guide/bgnet/examples/broadcaster.c

8. 常见的问题

我可以从哪边取得那些 header 文件呢?

如果你的系统还没有这些文件，你可能就不需要它们。检查你平台的手册。若你在 Windows 上开发，那麼你只需要 #include <winsock.h>。

在 bind() 回报”Address already in use”（地址已经在使用中）时，我该怎麼办呢？

你必须使用 setsockopt() 对 listen 的 socket 设置 SO_REUSEADDR 选项。请参考 bind() 及 select() 章节的示例。

我该如何取得系统上已经开启的 sockets 表呢？

使用 **netstat**。细节请参考 man 手册，不过你应该只要输入下列的命令就能取得一些不错的资料：

```
$ netstat
```

我该如何检视 routing table（路由表）呢？

使用 **route** 命令（多数的 Linux 系统是在 /sbin 底下），或者 **netstat -r** 命令。

如果我只有一台电脑，我该如何运行 client（客户）与 server（服务器）程序呢？我需要网路来做网路编程吗？

你很幸运，全部的系统都有实现一个 loopback 虚拟网路 ”设备”（device），这个设备位於 kernel（内核），并假装是个网卡〔这个接口就是 routing table 中所列出的 ”lo”〕。

假装你已经登录一个名为”goat”的系统，在一个窗口运行 client，并在另一个窗口运行 server。

或者可以在背景运行 server〔”server &”〕，并在同样的窗口运行 client。

loopback 设备的功能是你可以运行 **client goat** 或 **client localhost**〔因为”localhost”应该已经定义在你的 /etc/hosts 文件〕，而你可以让 client 与 server 通讯而不需要网路。

简而言之，不需要改变任何代码，就可以让程序在无网路的本地端系统上运行！好耶！

我该怎麼识别对方已经关闭连接呢？

你可以辨别出来，因为 recv() 会返回 0。

我该如何实现一个”ping”工具呢？什麼是 ICMP 呢？我可以在哪里找到更多关於 raw socket 与 SOCK_RAW 的资料呢？

你对 raw socket 的全部疑问都可以在 W. Richard Stevens 的 UNIX Network Programming 书本上找到答案。还有，研究 Stevens 的 UNIX Network Programming 代码的 ping 子目录，可以在线下载 [37]。

我该如何改变或减少调用 connect() 的 timeout（超时）时间呢？

我不想跟你说一样的答案：”W. Richard Stevens 会告诉你”，我只能建议你阅读 UNIX Network Programming 代码 [38] 中的 /lib/connect_nonb.c。

主要是你要用 socket() 建立一个 socket descriptor，将它设置为 non-blocking（非阻塞），调用 **connect()**，而如果一切顺利，**connect()** 会立馬返回 -1，并将 errno 设置为 EINPROGRESS。接着你要调用 **select()** 并设置你想要的 timeout 时间，传递读写组（read and write sets）的 socket descriptor。如果 select() 没有发生 timeout，这表示 connect() call 已经完成。此时，你必须使用 **getsockopt()** 设置 SO_ERROR 选择项以取得 connect() call 的返回值，在没有错误时，这个值应该是零。

最後，在你开始透过 socket 传输数据以前，你可能想要再将它设置回 blocking（阻塞）。

要注意的是，这麼做的好处是让你的程序在连接（connecting）期间也可以另外做点事情。比如：你可以将 timeout 时间设定为类似 500 毫秒，并在每次 timeout 发生时更新显示器画面，然後再次调用 select()。当你已经调用了 select() 时，并且 timeout 了，像这样

重复了 20 次，你就会知道应该放弃这个连接了。

如我所述的，请参考 Stevens 的既完美又优秀的代码示例。

我该如何写 Windows 的网路程序呢？

首先，请删除 Windows，并安装 Linux 或 BSD。;-)。不是的，实际上，只要参考前言中的〔Windows 程序设计师要注意的事情〕就可以了。

我该如何在 Solaris/SunOS 上编译程序呢？在我尝试编译时，一直遇到错误！

发生 Linker（链接器）错误是因为 Sun 系统在不会自动编入 socket 程序库。请参考前言中的〔Solaris/SunOS 程序员要注意的事情〕，有如何处理这个问题的示例。

为什麼 select() 一直跟 signal 吵架呢？

Signal 试图要让 blocked system call 返回 -1，并将 errno 设置为 EINTR。当你用 **sigaction()** 设置了一个 signal handler（信号处理例程）时，你可以设置 SA_RESTART flag，这可以在 system call 被中断之後重新打开它。

很自然的是这不会每次都管用。

我最爱的解法是使用一个 goto，你明白这会让你的教授很愤怒，所以放手去做吧！

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
if (errno == EINTR) {
// 某个 signal 中断了我们，所以重新启动
goto select_restart;
}
// 这里处理真正的错误：
perror("select");
```

```
}
```

当然，在这个例子里，你不需使用 goto；你可以用其它的 structures 来控制，但是我认为用 goto 比较乾净。

要怎麼样我才能实作调用 recv() 的 timeout 呢？

使用 **select()**！它可以让你对正在读取的 socket descriptors 指定 timeout 的参数。或者你可以将整个功能包在一个独立的函数中，类似这样：

```
#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;
    int n;
    struct timeval tv;

    // 设置 file descriptor set
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // 设置 timeout 的数据结构 struct timeval
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // 一直等到 timeout 或收到数据
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error
```

```
    // 数据一定有在这里，所以调用一般的 recv()
    return recv(s, buf, len, 0);
}
.
.
.
// 调用 recvtimeout() 的示例：
n = recvtimeout(s, buf, sizeof buf, 10); // 10 second timeout

if (n == -1) {
    // 发生错误
    perror("recvtimeout");
}
else if (n == -2) {
    // 发生 timeout
} else {
    // 从 buf 收到一些数据
}
.
.
.
```

请注意，**recvtimeout()** 在 timeout 的例子中会返回 -2，那为什麼不是返回 0 呢？好的，如果你还记得，在呼叫 recv() 返回 0 值时所代表的意思是对方已经关闭了连接。所以该返回值已经用过了，而 -1 表示”错误”，所以我选择 -2 做为我的 timeout 表示。

我该如何在将数据送给 socket 以前将数据加密或压缩呢？

一个简单的加密方法是使用 SSL（secure sockets layer），只是这超过本教程的范畴了〔细节请参考 OpenSSL project [39]〕。

不过假设你想要安插或实现你自己的压缩器（compressor）或加密系统（encryption

system)，这只不过是将你的数据想成在两个节点间运行连续的步骤，每个步骤以同样的方式改变数据：

1. server 从文件读取数据〔或是什麼地方〕
2. server 加密/压缩数据〔你新增这个部分〕
3. server 用 send() 送出加密数据

而另一边则是：

1. client 用 recv() 接收加密数据
2. client 译码/解压数据〔你新增这个部分〕
3. client 写数据到文件〔或是什麼地方〕

如果你正要压缩与加密，只要记得先压缩。:-)

只要 client 适当地还原 server 所做的事情，数据在另一端就会完好如初，不论你在中间增加了多少步骤。

所以你用我的代码所需要做的只有：找出读数据与透过网路传送〔使用 **send()**〕这中间的段落，并在那里加上编码的代码。

我一直看到的 ”PF_INET”是什麼呢？他跟 AF_INET 有关系吗？

是的，有关系，细节请参考 socket() 章节。

我该怎麼写一个 server，可以接受来自 client 的 shell 命令并运行命令呢？

为了简化，我们说 client 的连接用 connect()、send() 以及 close()〔即为，没有後续的 system calls，client 没有再次连接。〕

client 的处理过程是：

1. 用 connect() 连接到 server
2. send("/sbin/ls > /tmp/client.out")
3. 用 close() 关闭连接

此时，server 正在处理数据并运行命令：

1. accept() client 的连接
2. 使用 recv(str) 接收命令字符串
3. 用 close() 关闭连接
4. 用 system(str) 运行命令

注意！server 会运行全部 client 所送的命令，就像是提供了远端的 shell 权限，人们可以连接到你的 server 并用你的帐户做点事情。例如：若 client 送出 ”rm -rf ~” 会怎麼样呢？这会删掉你帐户里全部的数据，就是这样！

所以你学聪明了，你会避免 client 使用任何危险的工具，比如 **foobar** 工具：

```
if (!strncmp(str, "foobar", 6)) {
sprintf(sysstr, "%s > /tmp/server.out", str);
system(sysstr);
}
```

可是你还是不安全，没错：如果 client 输入 ”**foobar; rm -rf ~**” 呢？
最安全的方式是写一个小机制，将命令参数中的非字母数字字符前面放个〔”\”〕字符〔如果适合的话，要包括空白〕。
如你所见，当 server 开始运行 client 送来的东西时，安全（security）是个问题。

我正在传送大量数据，可是当我 recv() 时，它一次只收到 536 bytes 或 1460 bytes。可是如果我在我本地端运行，它就会一次就收到全部的数据，这是怎麼回事呢？
你碰到的是 MTU，即 physical medium（物理媒体）能处理的最大尺寸。在本地端上，你用的是 loopback 设备，它可以处理 8K 或更多数据也没有问题。但是在 Ethernet（以太网），它只能处理 1500 bytes〔有 header〕，你碰到这个限制。透过 modem 的话，MTU 是 576 bytes〔一样，有 header〕，你遇到比较低的限制。

你必须确认有送出全部的数据。〔细节请参考 **sendall()** 函数的实务〕。一旦你有确认，那麼你就需要在循环中调用 recv()，直到收到全部的数据。

对於使用多重调用 **recv()** 来接收完整数据包的细节，请参考数据封装之子（Son of Data Encapsulation）一节。

我用的是 Windows 系统，而且我没有 fork() system call 或任何的 struct sigaction 可以用，该怎麼办呢？
如果你问的是它们在哪里，它们会在 POSIX 程序库里，这个会包装在你的编译器中。

因为我没有 Windows 系统，所以我真的无法回答你，不过我似乎记得 Microsoft 有一个 POSIX 兼容层，那里会有 **fork()**。〔而且甚至会有 sigaction。〕

在 VC++ 的手册搜寻 ”fork” 或 ”POSIX”，看它是否能给你什麼线索。

如果这样一点都没有用，拿掉 fork()/sigaction 这些东西，用 Win32 中等价的函数来替换：**CreateProcess()**。我不知道怎麼用 **CreateProcess()**，它有多的数不清的参数，不过在 VC++ 的资料中应该可以找到怎麼使用它。

我在防火墙（firewall）後面，我该如何让防火墙外面的人知道我的 IP 地址，让他们可以连接到我的电脑呢？

毫无疑问地，防火墙的目的就是要防止防火墙外面的人连到防火墙里面的电脑，所以你让他们进来基本上会被认为是安全漏洞。

但也不是说完全不行，有一个方法，你仍然可以透过防火墙频繁的进行 **connect()**，如果防火墙是使用某种伪装（masquerading）或 NAT 或类似的方式。你只要让程序一直在做初始化连接，那麼你有机会成功的。

如果这样还不是很满意，你可以要求系统管理员在防火墙开一个小洞（hole），让人们可以连进你的电脑。防火墙可以透过 NAT 软件或 proxy（代理）或类似的方法将数据包转送给你。

要留意，不要对防火墙中的一个小洞掉以轻心。你必须确保你不会放坏人进来存取内网；如果你是新手，做软件安全的难度是远远超过你的想像。
不要让你的系统管理员对我发脾气 ;-)

我该怎麼写 packet sniffer 呢？我要怎麼将我的 Ethernet interface 设置为 promiscuous mode（混杂模式）呢？

这些事情是在底层运作的，当网卡设置为”promiscuous mode”时，它会转送全部的数据给操作系统，而不只是地址属於这台电脑的数据包而已。〔我们这里谈的是 Ethernet 层的地址，而不是 IP 地址，可是因为 ethernet 是在 IP 底层，所以全部的 IP 地址实际上都会转送。细节请参考”底层漫谈与网路理论”一节〕。

这是 packet sniffer 如何运作的基础，它将网卡设置为 promiscuous mode，接着 OS 会收到经过网线的每个数据，你会有一个可以用来读取数据的某种型别 socket。
毫无疑问地，这个问题的答案依平台而异，不过如果你用百度或 Google 搜寻，例如：”windows promiscuous ioctl”，你或许会在某个地方找到，看起来跟 Linux Journal [40] 中写的一样好的。

我该如何为 TCP 或 UDP socket 设定一个自订的 timeout 值呢？

这个按照你的系统而定，你可以在网上搜寻 SO_RCVTIMEO 与 SO_SNDTIMEO〔用在 **setsockopt()**〕，看看是否你的系统有支持这样的功能。
Linux man 手册建议使用 **alarm()** 或 **setitimer()** 作为替代品。

我要如何辨别哪些 ports 可以使用呢？有没有”官方”的 port numbers 呢？

通常这不会有问题，如果你正在写像 web server 这样的程序，那麼在你的程序使用 port 80 是个好主意。如果你只是想要写自己的 server，那麼随机选择一个 port〔不过要大於 1023〕，然後试试看。
如果 port 已经在使用中，你将会在尝试 **bind()** 时遇到”Address already in use”错误。选择另一个 port。〔利用 config 配置文件或命令行参数，让你的软件用户能指定 port 也是个不错的想法〕。
有一个官方的 port nubmer [41] 表，由 Internet Assigned Numbers Authority（IANA）所维护的。在表中的 ports〔超过 1023〕并不代表你不能使用，比如，Id 软件的 DOOM 跟”mdqs” 用一样的 port，不管那是什麼，最重要的是*在同一台机器上*没有人用掉你要用的 port。

[37] http://www.unpbook.com/src.html

[38] http://www.unpbook.com/src.html

[39] http://www.openssl.org/

[40] http://interactive.linuxjournal.com/article/4659

[41] http://www.iana.org/assignments/port-numbers

9. Man 手册

9.1. accept()

接受从 listening socket 进来的连接

函数原型

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


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

说明

一旦你完成取得 SOCK_STREAM socket 并将 socket 设置好可以用来 **listen()** 进来的连接，接着你就能调用 **accept()** 让自己能取得一个新的 socket descriptor，做为後续与新连接 client 的通讯。

原本用来 listen 的 socket 仍然还是会留着，当有新连接进来时，一样是用 **accept()** call 来接受新的连接。

s	**listen()** 中的 socket descriptor。
addr	这里会填入连线到你这里的 client 地址。
addrlen	这里会填入 addr 参数中传回的数据结构大小。如果你确定你知道一定会收到的是 struct sockaddr_in，你可以放心忽略这个参数，因为这就是你原本传递的 addr 型别。

accept() 通常会 block（闭锁），而你可以使用 **select()** 事先取得 listen 中的 socket descriptor 状态，检查 socket 是否就绪可读（ready to read）。若为就绪可读，则表示有新的连接正在等待被 **accept()**！另一个方式是将 listen 中的 socket 使用 **fcntl()** 设定 O_NONBLOCK flag，然後 listen 中的 socket descriptor 就不会造成 block，而是传回 -1，并将 errno 设置为 EWOULDBLOCK。

由 **accept()** 传回的 socket descriptor 是如假包换的 socket descriptor，开启并与远端主机连接，如果你要结束与 client 的连接，必须用 **close()** 关闭。

返回值

accept() 传回新连接的 socket descriptor，错误时传回 -1，并将 errno 设置适当的值。

示例

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// 首先：使用 getaddrinfo() 填好地址结构：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // 使用 IPv4 或 IPv6，都可以
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // 帮我填好我的 IP

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

// 建立一个 socket 、 bind 它，并对它进行 listen：

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

// 现在接受进入的连接：

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

// 准备与 socket descriptor new_fd 沟通！
```

参考

socket(), **getaddrinfo()**, **listen()**, struct sockaddr_in

以下尚未中文化。

9.2. bind()

Associate a socket with an IP address and port number

Prototypes

```
#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

Prototypes

```
#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

Prototypes

```
#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.

Prototypes

```
#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 List42 or in your /etc/services file.

Lastly, for input parameters, we have hints. This is really where you get to define what the **getaddinfo()** 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.)

[42] http://www.iana.org/assignments/port-numbers

Continuing on with the input paramters, 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) {
        close(sockfd);
        perror("connect");
        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

Prototypes

```
#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

Prototypes

```
#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.

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 or 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 SUPERCEDED
// 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.

Prototypes

```
#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

Prototypes

```
#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

Prototypes

```
#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

Prototypes

```
#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.)

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

Prototypes

```
#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:

```
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

Prototypes

```
#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()** or **inet_pton()** 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.

Prototypes

```
#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 funcion **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 **getaddinfo()** 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

Prototypes

```
#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

Prototypes

```
#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

Prototypes

```
#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:

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:

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] = 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

Prototypes

```
#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().

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

Prototypes

```
#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:

```
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.
```

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(s1, buf2, sizeof buf2, 0);
        }
}
```

See Also

poll()

9.20. setsockopt(), getsockopt()

Set various options for a socket

Prototypes

```
#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:

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.
SO_BROADCAST	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, is filled out for you by **getsockopt()** and you have to specify it for **setsockopt()**, where it will probably be sizeof(int).

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

Prototypes

```
#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:

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

Prototypes

```
#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

Prototypes

```
#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.

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

Prototypes

```
#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 **getaddinfo()** 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()**

10. 参考文献

你终於读到这里了，现在你可能正在大喊着：”还有吗！”。

可以从哪里获得更多的这类资料呢？

10.1 、书籍

在一些老学校，你很容易就能借到纸质书，请试着阅读下列推荐的这几本好书。

我曾经加入一个很有名的网路书商，不过他们新的客户追踪系统与已出版的文件无法兼容。因此，我不能收到任何反馈了。所以，如果你同情我的处境，请使用 paypal 赞助我〔beej@beej.us〕:-)

- Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 0131411551 [43], 0130810819 [44].
- Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0131876716 [45], 0130319961 [46], 0130320714 [47].
- 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): 0201633469 [48], 020163354X [49], 0201634953 [50], (0201776316 [51]).
- TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 0596002971 [52].
- Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201433079 [53].

[43] http://beej.us/guide/url/unixnet1

[44] http://beej.us/guide/url/unixnet2

[45] http://beej.us/guide/url/intertcp1

[46] http://beej.us/guide/url/intertcp2

[47] http://beej.us/guide/url/intertcp3

[48] http://beej.us/guide/url/tcpi1

[49] http://beej.us/guide/url/tcpi2

[50] http://beej.us/guide/url/tcpi3

[51] http://beej.us/guide/url/tcpi123

[52] http://beej.us/guide/url/tcpna

[53] http://beej.us/guide/url/advunix

10.2. 网站参考资料

在网路上：

- BSD Sockets: A Quick And Dirty Primer [54]〔也是 Unix 系统编程资讯！〕
- The Unix Socket FAQ [55]
- Intro to TCP/IP [56]
- TCP/IP FAQ [57]
- The Winsock FAQ [58]

[54] http://www.frostbytes.com/~jimf/papers/sockets/sockets.html

[55] http://www.developerweb.net/forum/forumdisplay.php?f=70

[56] http://pclt.cis.yale.edu/pclt/COMM/TCPIP.HTM

[57] http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/

[58] http://tangentsoft.net/wskfaq/

这里是相关的维基百科：

- Berkeley Sockets [59]
- Internet Protocol (IP) [60]
- Transmission Control Protocol (TCP) [61]

- User Datagram Protocol (UDP) [62]
- Client-Server [63]
- Serialization [64]〔封装与解封装数据〕

[59] http://en.wikipedia.org/wiki/Berkeley_sockets

[60] http://en.wikipedia.org/wiki/Internet_Protocol

[61] http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[62] http://en.wikipedia.org/wiki/User_Datagram_Protocol

[63] http://en.wikipedia.org/wiki/Client-server

[64] http://en.wikipedia.org/wiki/Serialization

10.3. RFC

RFC [65] 真的非常低阶！这些文件说明了分配的数字、编程 API 、及 Internet 上使用的协议。

我已经在这里帮你引用了一些 RFCs 的链接，所以你可以拿桶爆米花并开始你的学习之旅：

RFC 1 [66] — 第一篇 RFC；它会告诉你的事：比如什麼是 “Internet” 呢？因它融入生活，而深入了解它是如何设计的，这类的想法。〔显然这份 RFC 已经过时了！〕

RFC 768 [67]—The User Datagram Protocol (UDP)

RFC 791 [68]—The Internet Protocol (IP)

RFC 793 [69]—The Transmission Control Protocol (TCP)

RFC 854 [70]—The Telnet Protocol

RFC 959 [71]—File Transfer Protocol (FTP)

RFC 1350 [72]—The Trivial File Transfer Protocol (TFTP)

RFC 1459 [73]—Internet Relay Chat Protocol (IRC)

RFC 1918 [74]—Address Allocation for Private Internets

RFC 2131 [75]—Dynamic Host Configuration Protocol (DHCP)

RFC 2616 [76]—Hypertext Transfer Protocol (HTTP)

RFC 2821 [77]—Simple Mail Transfer Protocol (SMTP)

RFC 3330 [78]—Special-Use IPv4 Addresses

RFC 3493 [79]—Basic Socket Interface Extensions for IPv6

RFC 3542 [80]—Advanced Sockets Application Program Interface (API) for IPv6

RFC 3849 [81]—IPv6 Address Prefix Reserved for Documentation

RFC 3920 [82]—Extensible Messaging and Presence Protocol (XMPP)

RFC 3977 [83]—Network News Transfer Protocol (NNTP)

RFC 4193 [84]—Unique Local IPv6 Unicast Addresses

RFC 4506 [85]—External Data Representation Standard (XDR)

IETF 有一个很棒的在线工具，可以搜寻与浏览 RFC [86]。

[65] http://www.rfc-editor.org/

[66] http://tools.ietf.org/html/rfc1

[67] http://tools.ietf.org/html/rfc768

[68] http://tools.ietf.org/html/rfc791

[69] http://tools.ietf.org/html/rfc793

[70] http://tools.ietf.org/html/rfc854

[71] http://tools.ietf.org/html/rfc959

[72] http://tools.ietf.org/html/rfc1350

[73] http://tools.ietf.org/html/rfc1459

[74] http://tools.ietf.org/html/rfc1918

[75] http://tools.ietf.org/html/rfc2131

[76] http://tools.ietf.org/html/rfc2616

[77] http://tools.ietf.org/html/rfc2821

[78] http://tools.ietf.org/html/rfc3330

[79] http://tools.ietf.org/html/rfc3493

[80] http://tools.ietf.org/html/rfc3542

[81] http://tools.ietf.org/html/rfc3849

[82] http://tools.ietf.org/html/rfc3920

[83] http://tools.ietf.org/html/rfc3977

[84] http://tools.ietf.org/html/rfc4193

[85] http://tools.ietf.org/html/rfc4506

[86] http://tools.ietf.org/rfc/