

高级 TCP/IP 编程

Effective TCP/IP Programming

Jon C.Snader 著

刘江林 译

44 Tips to Improve Your Network Programs



高级 TCP/IP 编程

Effective TCP/IP Programming

Jon C.Snyder 著

刘江林 译

44 Tips to Improve Your Network Programs

中国电力出版社

内 容 提 要

本书涉及了网络编程的所有细节，通过对 TCP/IP 编程精细部分的分析，帮助读者理解网络协议内部是如何与应用程序交互的。全书分为四章，提供了 44 个 TCP/IP 编程技巧，生动详实的探索了网络编程的各个方面。

本书适合中、高级网络程序员阅读，也可供专业计算机人士参考。

图书在版编目（CIP）数据

高级 TCP/IP 编程/（美）斯纳德著；刘江林译。-北京：中国电力出版社，2001.6

ISBN 7-5083-0661-9

I . 高… II . ①斯… ②刘… III . 计算机网络-通信协议-程序设计
IV . TN915.04

中国版本图书馆 CIP 数据核字（2001）第 034249 号

北京版权局著作权登记号 图字 01-2001-1871

本书英文版原名：Effective TCP/IP Programming

Published by arrangement with Addison Wesley Longman, Inc.

All rights reserved.

本书中文版由美国培生集团授权出版,版权所有。

中国电力出版社出版、发行

（北京三里河路 6 号 100044 <http://www.infopower.com.cn>）

三河市实验小学印刷厂印刷

各地新华书店经售

*

2001 年 8 月第一版 2001 年 8 月北京第一次印刷

787 毫米×1092 毫米 16 开本 19.5 印张 441 千字

定价 35.00 元

版 权 所 有 翻 印 必 究

（本书如有印装质量问题，我社发行部负责退换）

前　　言

简介

随着 Internet 爆炸性的增长，无线通信以及网络技术的发展，编写网络应用程序的程序员和工程师也在相应增加。TCP/IP 编程看起来非常简单。应用程序接口（Application Programming Interface, API）十分易懂，即使是刚刚入门的初学者也可以通过使用客户端或服务器的程序模板来编写应用程序。

然而，通常初学者在经历了最初的高效率之后，在细节面前开始停滞不前，并且发现他们的应用程序正遭受着性能或健壮性的考验。网络编程是一个充满着黑暗角落的领域，有些细节也可能会被错误地理解。本书通过对 TCP/IP 编程最精细部分的分析，让光明照亮那些角落并帮助你改正错误。

在读完本书之后，你对网络编程的许多难点会有一个透彻的理解。本书涉及了网络编程的所有细节。通过对这些小细节的理解，读者将获得相应知识，即网络协议的内部工作机制是如何跟应用程序交互的。拥有了这些知识，那些以前看起来令人困惑的程序行为就会变得很容易理解，问题的解决办法也会变得很清晰。

本书的组织方式有点与众不同，我们在一系列的技巧中每次只探讨一个问题。在学习一个特定的难点的过程中，我们会比较深入地去探索 TCP/IP 编程的某一个方面。这样，读完本书之后，读者不仅可以确认和处理一般的问题，而且对 TCP/IP 协议如何与应用程序一起工作和交互会有一个相对全面的认识。

本书的组织方式也许会导致某些地方的脱节。为了方便读者阅读，第一章包含了一个路径图，它解释每章的内容与基本结构。目录中列出了所有的技巧，这也会加强你对文字组织方式的理解。因为每个技巧的标题是以祈使句的形式表达的，所以你也可以考虑使用目录作为一系列的网络编程规则。

另一方面，技巧的组织方式使本书更适合作为一本手册。当在日常工作中遇到问题，可以很容易地去重读相应的技巧，重新认识特定的问题。你将会发现许多主题在多个技巧中涉及到，这种重复可以帮助你巩固概念并使它们看起来更简单自然。

读者对象

本书主要是为有一定基础的初学者和中级网络程序员编写的，但是对有经验的读者

也有一定的参考价值。虽然我们假定读者对网络和基本的套接字 API 有一定的熟悉程度，但在第一章里还是包含了对基本的套接字调用的回顾，并使用它们建立了一个原始的客户端和服务器。技巧 4（开发和使用程序框架）更详细地讲解了不同的客户端和服务器模型，所以即使读者只有很少的网络编程知识，也能够理解本书并从中受益。

几乎所有的例子都是用 C 语言编写的，所以读者必须对基本的 C 语言有一个很好的理解，这样你就会从本书中的程序中获益匪浅。在技巧 31（记住世界并不全是 C 语言的）里，我们展示了用 Perl 语言编写的例子，但是我们假定你没有 Perl 语言的知识。同样，本书中有一些小的 shell 程序的例子，但是理解它们也无需有 shell 编程经验。

例子和文字试图做到与平台无关。除了少数例子外，其他的都可以在 UNIX 或 Win32 系统下编译并运行。即使那些不在 UNIX 或 Win32 系统下工作的程序员，也会很轻松地把那些例子移植到他们工作的平台上。

排版约定

在我们学习的过程中，将建立和运行许多小的程序，这些程序是为我们正在探讨的问题而举出的例子。在显示交互的输入和输出时，我们使用下面的约定：

- 我们输入的英文设置为加黑 Courier 字体。
- 系统输出的英文设置为普通 Courier 字体。
- 不作为实际输入和输出部分的注释设置为斜体。

下面是从技巧 9 中摘取的一个例子：

```
bsd: $ tcprw localhost 9000
hello
received message 1
hello again
tcprw: readline failed: Connection reset by peer (54)
bsd: $
```

*this is printed after a 5-second delay
the server is killed here*

注意，我们在 shell 提示符中包含了系统名称。在前面的例子里，我们看到 tcprw 运行在名称为 bsd 的主机上。

在我们介绍一个新 API 函数时，我们把自己的或标准系统调用封装到一个方框里。标准的系统调用封装到实线方框里，如下所示：

```
#include<sys/socket.h>          /* UNIX */
#include<winsock2.h>            /* Windows */
int connect(SOCKET s,const struct sockaddr *peer, int peer_len);
>Returns:0 on success, -1(UNIX)or nonzero(Windows)on failure
```

我们自己开发的 API 函数封装到虚线方框里，如下所示：

```
#include "etc.h"  
SOCKET tcp_server( char *host, char *port );  
>Returns:a listening socket(terminates on error)
```

附加的注释和放于脚注里的资料设置为小字体并和本段一样缩进。通常这些材料可以在第一遍阅读时忽略掉。

最后，我们用尖括号把 URL 括起来，如下所示：

<<http://www.freebsd.org>>

可获得的源代码和勘误表

本书中所有例子的源代码都可以从<<http://www.netcom.com/~jsnader>>上获得电子版本。因为本书中的例子是直接从程序中抽取出来的，你可以得到它们并在它们上面做自己的试验。框架和库代码也可以下载来供自己使用。

在同一 Web 站点还可以获得勘误表。

致谢

通常，作者都要感谢他们的家庭成员在写书过程中给予的帮助，现在我知道他们为什么要这样做了。如果没有我的妻子 Maria 的帮助，本书的文字也许不会令人感到满意。如果没有她承担的比我的“50%”大得多的校对，也许我就不会有足够的时间来完成本书。对此她所干的额外家务和忍受的孤独夜晚，任何感激的话都不为过。

作者的另一个宝贵的资源是那些苦苦读完早期草稿的审阅者。本书的技术审阅者找出了很多错误。既有技术错误也有排版错误，他们改正我错误的理解，建议我采用新的方法，告诉我以前不知道的东西，有时甚至给我鼓励。在此我感谢 Chris Cheeland、Bob Gilligan (FreeGate 公司)、Peter Haverlock (北电网络)、S. Lee Henry (Web Publishing 公司)、Mukesh Kacker (Sun 公司)、Uri Raz 和 Rich Stevens，感谢他们的辛勤工作和建议。本书因为他们的帮助而做得更好。

最后，我要感谢我的编辑 Karen Gettman、项目编辑 Mary Hart、产品协调员 Tyrrell Albaugh 和拷贝编辑 Cat Ohala。和他们一起工作充满了乐趣，他们对我这个第一次写书的作者的帮助很大。

欢迎读者把评论、建议和校正反馈给我。

Tampa, Florida
1999 年 12 月

Jon C. Snader
jsnader@ix.netcom.com
<http://www.netcom.com/~jsnader>

目 录

前 言

第一章 简介	1
一些约定	1
本书结构	2
客户端-服务器体系结构	4
基本的套接字 API 回顾	5
第二章 基本知识	15
技巧 1 理解基于连接和无连接协议之间的差异	15
技巧 2 理解子网和 CIDR	20
技巧 3 理解私有地址和 NAT	30
技巧 4 开发和使用应用程序“框架”	33
技巧 5 选择套接字接口而不是 XTI/TLI	48
技巧 6 记住 TCP 是一个流协议	50
技巧 7 不要低估 TCP 的性能	59
技巧 8 不要彻底改造 TCP	70
技巧 9 注意 TCP 是可靠的协议但并非是不会出错的协议	73
技巧 10 记住 TCP/IP 不是轮询	81
技巧 11 为来自对等方的不合要求的行为做准备	98
技巧 12 不要认为成功的 LAN 策略一定可以移植到 WAN 上	106
技巧 13 学习协议是如何工作的	113
技巧 14 不要把 OSI 七层参考模型看得太重要了	114
第三章 创建高效和健壮的网络程序	119
技巧 15 理解 TCP 写操作	119
技巧 16 理解 TCP 顺序释放操作	124
技巧 17 考虑让 inetd 启动应用程序	132
技巧 18 考虑让 tcpmux“指定”服务器的已知端口	142
技巧 19 考虑使用两个 TCP 连接	153
技巧 20 考虑使应用程序事件驱动（1）	161

技巧 21 考虑使应用程序事件驱动 (2)	170
技巧 22 不要使用 TIME-WAIT ASSASSINATION 关闭连接	179
技巧 23 服务器应当设置 SO_REUSEADDR 选项	183
技巧 24 尽量使用大型写操作代替多个小规模写操作	188
技巧 25 理解如何使 <code>connect</code> 调用具有超时机制	195
技巧 26 避免数据拷贝	202
技巧 27 在使用之前置 <code>sockaddr_in</code> 为零	217
技巧 28 不要忘记字节的性别	218
技巧 29 不要在应用程序中对 IP 地址和端口号硬编码	221
技巧 30 理解已连接 UDP 套接字	226
技巧 31 记住这个世界并不全是 C 语言	230
技巧 32 理解缓冲区大小的影响	235
第四章 工具和资源	239
技巧 33 熟悉 ping 实用程序	239
技巧 34 学会使用 <code>tcpdump</code> 或一个类似的工具	241
技巧 35 学会如何使用 <code>traceroute</code>	250
技巧 36 学会使用 <code>ttcp</code>	256
技巧 37 学会使用 <code>lsof</code>	260
技巧 38 学会使用 <code>netstat</code>	262
技巧 39 学会使用系统调用跟踪工具	270
技巧 40 创建和使用捕获 ICMP 消息的工具	277
技巧 41 读 Stevnes 的书	286
技巧 42 阅读代码	288
技巧 43 访问 RFC Editor 主页	290
技巧 44 经常访问新闻组	291
附录 A 各种 UNIX 代码	293
<code>etcpt.h</code> 头文件	293
<code>daemon</code> 函数	295
<code>signal</code> 函数	296
附录 B 各种 Windows 代码	298
Window 兼容性函数	299

第一章

简介

本书的目的是帮助有一定基础的初学者和中级网络程序员成为一个熟练的网络编程人员，并且最终成为大师。向大师级网络程序员的转变在很大程度上靠的是经验和特定知识的积累，当然这些东西有时是很模糊的概念。除了时间和实践之外没有别的东西能够提供经验，而本书可以在知识的积累方面帮助你。

当然，网络编程是一个很大的领域，采用网络技术在两台或更多的机器之间进行通信时，存在很多的障碍。这些障碍从简单的如串行链接到复杂的如 IBM 的系统网络体系结构 (System Network Architecture)，应有尽有。今天，我们清楚地知道 TCP/IP 协议是构造网络的首选技术。这个选择很大程度上是因为 Internet 以及它最流行的应用程序：万维网 (World Wide Web)。

虽然 Web 使用了应用程序 (Web 浏览器和服务器) 和协议 (如 HTTP)。但实际上它并不是一个应用程序也不是一个协议。我们的意思是，Web 是 Internet 上的最流行的可视的网络技术应用程序。

即使在 Web 诞生之前，TCP/IP 也是一种很流行的构架网络的方法。因为它是一个开放的标准并且可以在不同供应商的机器之间进行互连，所以人们越来越多地使用它来建设网络和网络应用程序。到 1990 年底，TCP/IP 已经成为网络的主导技术，而且这种趋势似乎将在一段时间内保持下去。因此，本书把精力集中在 TCP/IP 以及运行 TCP/IP 的网络上。

如果希望掌握网络编程，就必须首先汲取一些背景知识，这些知识对更全面地理解和评价本书的程序是很有用的。首先介绍初学者通常会遇到的几个问题，这些问题大都是由于一些误解造成的，其中包括对 TCP/IP 某些方面和用于在机器之间进行通信的 API 的不完全理解。所有这些问题都是客观存在的，它们永远是困惑的源泉，而且在网络新闻组里也是经常涉及到的主题。

一些约定

本书中的文本材料和程序，除了一些显而易见的之外，都试图做到可以在 UNIX (32 位或 64 位) 和 Microsoft Win32 API 之间移植。我们对 16 位的 Windows 应用程序不予考虑。也就是说，几乎所有的材料和大多数的程序在其他的环境里也是适用的。

这个易移植的愿望在代码示例中导致了一些不便。UNIX 程序员也许对套接字描述符的概念斜眼相看，因为套接字描述符是用 SOCKET 而不是用 int 类型定义的；而 Windows 程序员也会注意到我们严格的提交控制台应用程序。这些约定将在技巧 4 中讲述。

同样地，我们尽量避免在套接字上使用 read 和 write 函数，这是因为 Win32 API 不支持在套接字上调用这些系统函数。我们经常谈到读或写一个套接字，但是我们只是一般地说说而已。我们用 recv、recvfrom 或 recvmsg 来表示“读”，而用 send、sendto 或 sendmsg 来表示“写”。在特定地表示读系统调用时，我们把它设置为 Courier 字体的 read，该约定对写也一样适用。

是否在本书中包含 IPv6 的材料是最难做出的选择之一，IPv6 在不久的将来会取代 Internet 协议（IP）的现有版本（IPv4）。最后，我们决定不涉及 IPv6。做出这个决定有很多原因，包括：

- 几乎所有本书中的东西都是正确的，不管它是使用 IPv4 还是 IPv6。
- 确实存在的差异在 API 的地址部分本地化了。
- 本书在很大程度上提供的是熟练网络程序员的共同的经验和知识，而且我们确实仍然没有 IPv6 的经验，这是因为 IPv6 仅仅在近期才广泛地得以实现。

因此，如果我们谈到没有限定的 IP 时，指的就是 IPv4。在确实提到 IPv6 的地方，我们会十分小心地明白地指出它是 IPv6。

最后，本书以 8 位数据单元为一个字节，而在网络社区里通常称这些数据为 octets，这是历史原因造成的。过去一个字节的大小是依赖于平台而定的，即使是它大小标准也没有统一一致。为了避免混淆，早期的网络书籍杜撰术语 octet 来表示字节的大小。今天，人们普遍认同一个字节就是 8 位长[Kernighan and Pike 1999]，而使用 octet 似乎显得有些书生气了。

本 书 结 构

在本章的其余的部分中，将介绍基本的套接字 API 和在编写 TCP/IP 应用程序时使用的客户端-服务器体系结构。这是掌握网络编程所必须建立的基础。

第二章讨论了一些有关 TCP/IP 和网络的基本事实和误解。例如，在本章将讲述基于连接协议和无连接协议之间的差异。我们将研究 IP 寻址和子网的一些经常令人困惑的主题，classless interdomain routing（无级域间路由选择，CIDR）和 network address translation（网络地址转换法，NAT）。我们可以了解到 TCP 实际上不保证数据的递交，认识到我们必须为对等方（peer）和用户的错误操作做准备，以及我们的应用程序在广域网（WAN）内运行的结果可能和在局域网（LAN）内不一样。

我们知道 TCP 是一个流协议，也知道这对作为程序员的我们来说意味着什么。同时，我们将了解到 TCP 不自动检测连接的丢失，了解到为什么不检测是好事情，也将了解到我们可以从它上面做些什么。

我们将了解为什么套接字 API 应该总是更喜欢建立在 Transport Layer Interface/X/Open

Transport Interface (TLI/XTI) 之上，了解我们为什么不应把开放系统互联模型（OSI）看得很重要，也将了解 TCP 是一个具有显著效率、优秀性能的协议，了解经常使用 UDP 来重现它的功能是不明智的。

在第二章中，将为几个 TCP/IP 应用程序模型开发框架代码，并使用它来构造一个经常使用的函数的类库。这些框架和类库都是很重要的，因为它们使我们在编写应用程序的时候不担心例行的琐事如地址转换、连接管理等等。因为可以使用这些框架，所以我们就不会被捷径所诱惑，如固定地址和端口号，或忽略错误返回值。

我们在本书中重复地使用这些框架和类库来建立测试例子、示例代码，甚至使用独立工作的应用程序。我们经常通过在自己的框架中增加几行代码来建立一个特殊用途的应用程序或测试例子。

在第三章中，我们比较深入地讨论了几个看起来很琐碎的主题。例如，我们首先讨论了 TCP 的一些操作以及该操作做了一些什么事情。开始看起来这似乎很简单：我们写入 n 个字节，然后 TCP 把它们发送到对等方。然而，正如我们将会看到的，实际情况并非如此。TCP 有一个复杂的规则集决定它是否可以把写入的数据立即发送出去。如果可以的话，代价又是多少呢？如果要编写既健壮又高效的程序，那么理解这些规则以及它们是如何跟我们的应用程序交互的就很必要了。

读数据和连接终止需要考虑同样的事情。我们将分析这些操作并学会如何执行一个有序的终止过程，如何保证数据不会丢失。我们也将分析 connect 操作并学会如何指定超时，以及如何在 UDP 应用程序中使用它。

我们将会学习 UNIX 超级服务器端口监视程序（inetd）的使用，以及了解到它是如何极大地减少编写具有网络功能的应用程序必须做的工作。同时，我们也会了解到我们是如何使用 tcpmux 来减少分配已知端口给服务器时的风险。我们将展示 tcpmux 是如何工作的以及如何在没有该功能的系统上建造自己版本的 tcpmux。

本书还将深入地讨论下列不易理解的主题：TIME-WAIT 状态、Nagle 算法、选择缓冲区大小以及 SO_REUSEADDR 套接字选项的正确用法。我们将了解如何使我们的程序成为应用程序事件驱动的程序以及如何才能为每一个事件提供独立的定时器。我们也会分析一些即使是高级的网络程序员也会犯的错误，并且将会学习一些可以用来提高应用程序性能的技术。

最后，我们介绍了网络和脚本语言。通过使用一些 Perl 脚本，说明了使用 Perl 是可以很容易地编写有用的网络实用程序和测试驱动程序的。

第四章介绍了两个领域。首先分析了几个工具，这些工具是每一个网络程序员必须知道的东西。开始介绍的是经典的 ping 实用程序，并说明了它是如何可以用在一些基本的疑难解答上面的。接着泛泛地分析了网络监听工具并特别分析了 tcpdump。在整个第四章中，我们将会看到运用 tcpdump 来诊断应用程序问题和困惑的几个例子。也会学习 traceroute 以及应用它来研究 Internet 的一个很小部分的形状。

ttcp 实用程序（该程序是 ping 程序的作者 Mike Muuss 的合作者）是一个研究网络性能和某些 TCP 参数对性能影响的很有用的工具。我们使用它来演示一些诊断技术。另一个公

共享工具 lsof 在匹配网络并连接到打开这些连接的进程时是很有用的。很多时候，lsof 提供了一些很有用的信息，这些信息在没有 lsof 时不花大力气是不可获得的。

我们将详细地分析 netstat 实用程序，以及它提供的很多不同类型的信息。我们也会分析系统调用跟踪程序如 ktrace 和 truss。

我们在第四章中编写一个分析工具来结束该章第一部分的讨论，这个工具可以拦截并显示 Internet Control Message Protocol（网际控制报文协议，ICMP）数据报。它不仅为我们的工具箱里增加了一个有用的工具，同时也是一个使用原始套接字（raw sockets）的例子。

在第四章的第二部分中我们介绍了一些可用来加深我们的知识、加深对 TCP/IP 和网络的理解的资源。我们将会了解 Rich Stevens 编写的著名书籍、我们可以研究和学习的网络代码、可以从 Internet Engineering Task Force（因特网特别工作组，IETF）获得的 request for comments（请求评议，因特网提议的标准，RFC）集合以及 Usenet 新闻组。

客户端-服务器体系结构

尽管我们经常说起客户端和服务器，但在一般的情况下，一个特定的程序到底是客户端还是服务器不总是很清楚。程序经常更像对等的双方，它们之间互相交换信息，无法看出给客户端提供的信息。通过 TCP/IP 来看，区别就明显了。服务器监听 TCP 连接或客户端主动提供的 UDP 数据报。从客户端的角度来看，我们可以说客户端是首先“说话”的一方。

在本书中我们考虑三种典型的客户端-服务器的情形，如图 1.1 所示。在第一种情形中，

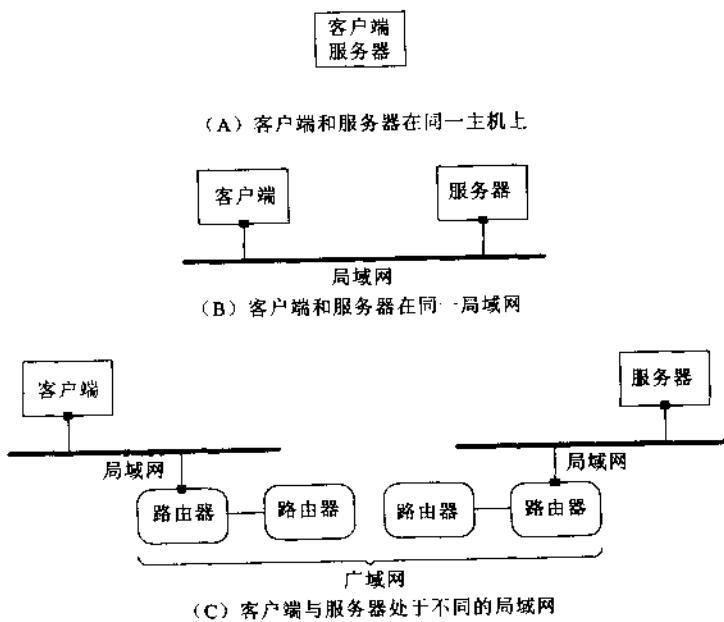


图 1.1 典型的客户端-服务器结构的几种情形

客户端和服务器运行在同一台机器上，如图 1.1 (A) 所示。因为没有涉及到物理网络，所以这是最简单的一种情形。输出的数据照常沿着 TCP/IP 栈下行，但是它不放在网络设备输出队列上，而是在内部返回，沿着 TCP/IP 栈上行作为输入。

在开发时，这种设置有很多优点，即使客户端和服务器最终是运行在不同的机器上。首先，可以很容易粗略地判断客户端和服务器应用程序的性能，因为在这种情况下没有网络延时；其次，这种方法提供了一种理想化的实验环境，包不会丢失、延迟或不按照顺序来递交。

正如我们将会在技巧 7 里了解的那样，至少是在大多数的时间里，我们必须强调，即使在这种环境下也可能导致 UDP 数据报的丢失。

最后，如果我们在同一台机器上调试的话，那么开发工作就会经常变得更容易而且更方便。

当然，客户端和服务器即使是在成为产品的环境下也可以运行在同一台机器上。请参阅技巧 26，了解基于这种情况的一个例子。

在第二种客户端-服务器设置中，如图 1.1 (B) 所示，客户端和服务器运行在同一个局域网内的不同机器上。这个环境涉及到真正的网络，但是这个环境仍然还是近乎理想化的。数据包几乎不丢失，实际上也不会乱序到达。这是一个通常的产品环境，在很多情况下应用程序就是专为这种环境设计的，而不会运行在其他的环境下。

基于这种情况的一个常见的例子就是打印服务器。一个小的局域网中可能为几台主机只配置一个打印机。其中一个主机（或者打印机内置的 TCP/IP 栈）充当服务器，接收来自其他主机的客户端的打印请求，并把这些数据放到缓冲区中等候打印机打印。

第三种类型客户端-服务器的情形涉及到被广域网如图 1.1 (C) 分割的客户端和服务器。广域网可以是 Internet 或者是公司的 intranet，但是最关键的是两个应用程序不在同一个局域网内，而且从一个应用程序发出到另一个应用程序的 IP 数据报必须通过一个或几个路由器。

这种环境比前两种复杂得多。随着广域网内流量的增加，路由器队列经常临时存储数据包直到它们可以转发出去才开始填充。当路由器的队列空间不够时，它们就开始丢弃数据包，这就会导致重传，进而导致重复和乱序传递数据包。这些问题都不是理论上的，而且都比较常见，如同我们将会在技巧 38 里看到的那样。

我们将会在技巧 12 里详细地讨论局域网环境和广域网环境之间的差异，但是目前我们仅仅只能说它们的表现几乎完全不同。

基本的套接字 API 回顾

在本节中，我们将回顾基本的套接字 API 并使用它来创建初步的客户端和服务器应用程序。尽管这些应用程序上是“没有血肉的”，但它们确实说明了 TCP 客户端和服务器的本质特性。

让我们以简单的客户端所必须的 API 调用开始。图 1.2 是每个客户端必须使用的基本套接字调用的示意图。如图 1.2 所示，对等方（peer）的地址在 sockaddr_in 结构里被指定。并传递到 connect。

```
#include <sys/socket.h>      /* UNIX */
#include <winsock2.h>        /* Windows */

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

>Returns:Socket descriptor on success, -1(UNIX) or INVALID_SOCKET (Windows) on failure
```

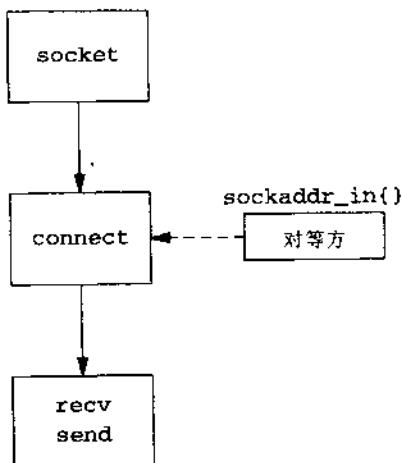


图 1.2 客户端的基本套接字调用

通常我们必须做的第一件事情是为连接获得一个套接字，通过 socket 系统调用来完成。

套接字 API 是与协议无关的，并且它可以支持几个不同的通信域（communication domain）。domain 参数是一个常数，表示所希望的通信域。

两个最普通的域是 AF_INET（或 Internet）域和 AF_LOCAL（或 AF_UNIX）域。在本书中我们只考虑 AF_INET 域。AF_LOCAL 域用于同一机器上的进程间通信（interprocess communication, IPC）。

关于 domain 常数应当为 AF_* 还是 PF_* 的争论一直存在。PF_* 的支持者指出 PF_* 曾用于 4.1c/2.8/2.9BSD 的至今已经不存在的 socket 调用版本中，并指出 PF 代表的是 protocol family 这一事实。AF_* 的支持者指出核心套接字代码和 AF_* 常数的 domain 参数相匹配。因为这两套常数是同样地定义的——实际上，其中一个是根据另一个定义的——所以我们无论使用哪一个都没有应用上的差别。

type 参数指示将要创建的套接字的类型。最常用的值以及我们在本书中使用的值列出如下：

- **SOCK_STREAM**——这些套接字提供了一个可靠的、全双工的面向连接的字节流。在 TCP/IP 中，它指的是 TCP。
- **SOCK_DGRAM**——这些套接字提供了一个不可靠的、效率高的数据报服务。在 TCP/IP 中，它指的是 UDP。
- **SOCK_RAW**——这些套接字允许访问 IP 层中的一些数据报。它们用于特殊的用途，如监听 ICMP 消息。

protocol 域指示对套接字应当使用哪一个协议。对于 TCP/IP，这通常用套接字类型来显式地指定，并且该参数设置为 0。在一些情况下，如 raw 套接字，存在好几个可能的协议，你必须指定其中一个为将要使用的协议。我们将会在技巧 40 里看到这样的一个例子。

对于最简单的 TCP 客户端，另外一个 socket API 要求我们建立一个同我们对等方的会话是 **connect**，它用于建立连接：

```
#include <sys/socket.h>      /* UNIX */
#include <winsock2.h>        /* Windows */
int connect( SOCKET s, const struct sockaddr *peer, int peer_len );
```

Returns:0 on success, -1 (UNIX) or nonzero (Windows) on failure

s 参数是 socket 调用返回的套接字描述符。**peer** 参数指向一个地址结构，它保存着对等方的地址和其他信息。对于 AF_INET 域来说，这是一个 sockaddr_in 结构。让我们花费几分钟来看一下那个简单的例子。**peer_len** 参数是 **peer** 指向的结构的大小。

一旦建立了一个连接，我们就可以传输数据了。在 UNIX 平台下，我们可以简单地使用套接字描述符来调用 **read** 和 **write** 函数，这与我们使用文件描述符是一样的。正如我们前面提到的一样，不幸的是，Windows 并没有在套接字环境下重载这些系统调用，我们必须使用 **recv** 和 **send** 来代替它们。除了有一个额外的参数，这些调用是与 **read** 和 **write** 一样的：

```
#include <sys/socket.h>      /* UNIX */
#include <winsock2.h>        /* Windows */
int recv( SOCKET s, void *buf, size_t len, int flags );
int send( SOCKET s, const void *buf, size_t len, int flags );
```

Returns:number of bytes transferred on success, -1 on failure

s、**buf** 和 **len** 参数是与 **read** 和 **write** 中的参数一样的。**flags** 参数可以采用的值通常是跟系统有关的，但是 UNIX 和 Windows 都支持下列值：

- **MSG_OOB**——如果设置为该值，这个标志指示就要发送或读取紧急的数据。
 - **MSG_PEEK**——该标志用于读取进来的数据，但不把数据从接收缓冲区里删除。
- 调用该函数后，数据仍然可以被以后的读操作获得。

- **MSG_DONTROUTE**——该标志指示系统内核忽略通常的路由功能。它通常仅用于路由程序中或者用于诊断。

在处理 TCP 时，这些调用通常是必需的。然而，对于 UDP 的使用，`recvfrom` 和 `sendto` 调用是很有用的。这些调用与 `recv` 和 `send` 是亲兄弟，但是它们允许我们在发送一个 UDP 数据报时指定目的地址以及在读取一个 UDP 数据报时取出源地址：

```
#include <sys/socket.h>          /* UNIX */
#include <winsock2.h>           /* Windows */
int recv from( SOCKET s, void *buf, size_t len, int flags );
    struct sockaddr *from, int *fromlen );
int send to( SOCKET s, const void *buf, size_t len, int flags );
    const struct sockaddr *to, int tolen );
>Returns:number of bytes transferred on success, -1 on failure
```

前面的 4 个参数——`s`、`buf`、`len` 和 `flags`——与在 `recv` 和 `send` 调用中是一样的。`recvfrom` 调用中的 `from` 参数指向一个套接字地址结构，系统内核在这个结构中存储进来的数据报的源地址信息。该地址的长度存储在 `fromlen` 指向的整数中。应当注意的是 `fromlen` 是一个指向整数的指针。

同理，在 `Sendto` 调用中 `to` 参数指向一个套接字地址结构，它包含数据报的目的地，`tolen` 参数是此地址结构的长度，注意 `tolen` 仅仅是一个整数，不是一个指针。

现在我们看一个简单的 TCP 客户端（图 1.3）。

simplec.c

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <stdio.h>

6 int main( void )
7 {
8     struct sockaddr_in peer;
9     int s;
10    int rc;
11    char buf[ 1 ];

12    peer.sin_family = AF_INET;
```

```

13 peer.sin_port = htons( 7500 );
14 peer.sin_addr.s_addr = inet_addr( "127.0.0.1" );

15 s = socket( AF_INET, SOCK_STREAM, 0 );
16 if ( s < 0 )
17 {
18     perror( "socket call failed" );
19     exit( 1 );
20 }

21 rc = connect( s, ( struct sockaddr * )&peer, sizeof( peer ) );
22 if ( rc )
23 {
24     perror( "connect call failed" );
25     exit( 1 );
26 }
27 rc = send( s, "1", 1, 0 );
28 if ( rc <= 0 )
29 {
30     perror( "send call failed" );
31     exit( 1 );
32 }
33 rc = recv( s, buf, 1, 0 );
34 if ( rc <= 0 )
35     perror( "recv call failed" );
36 else
37     printf( "%c\n", buf[ 0 ] );
38 exit( 0 );
39 }

```

simplec.c

图 1.3 一个简单的 TCP 客户端

我们把图 1.3 的代码编写成一个 UNIX 程序，以后再处理复杂的可移植代码和 Windows WSAStartup 逻辑性。正如我们将在技巧 4 中看到的一样，我们可以在一个头文件中隐含大部分的繁杂代码，但是在这之前我们必须首先建立一些方法。而在这一切之前，我们将只使用相对简单的 UNIX 模式。

建立对等方的地址

12~14 我们在 `sockaddr_in` 结构中填写服务器的端口号（7500）和地址。127.0.0.1 是本地地址。它指示服务器和客户端在同一台机器上。

获得套接字并连接到对等方

15~20 我们获得一个 `SOCK_STREAM` 套接字。正如我们前面提到的那样，因为 TCP 是一个流协议，所以它使用这种类型的套接字。

21~26 我们通过调用 `connect` 函数来跟对等方建立连接。我们使用这个调用来指定对等方的地址。

发送并接收一个字节

37~38 我们首先发送单字节 1 到对等方，然后立即从套接字中读取一个字节，最后我们把该字节写入到标准输出并结束程序。

在使用我们的客户端之前，我们需要一个服务器。服务器的 `socket` 调用有一点不同，如图 1.4 显示的那样。

一个服务器必须在它的已知端口上监听客户端的连接，正如我们将在后面看到的一样，服务器通过 `listen` 调用达到这个目的。但是首先它必须绑定接口的地址和已知端口到它的监听套接字，这可以通过 `bind` 调用达到：

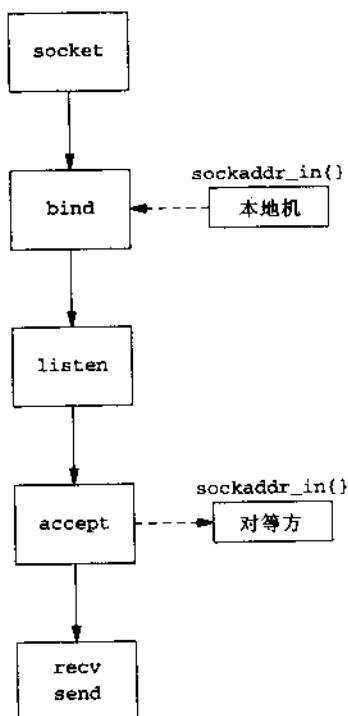


图 1.4 服务器上的基本套接字调用

```
#include <sys/socket.h>      /* UNIX */
#include <winsock2.h>        /* Windows */
int bind( SOCKET s, const struct sockaddr *name, int namelen );
>Returns:0 on success, -1 (UNIX) or SOCKET_ERROR (Windows) on error
```

参数 `s` 是监听套接字的描述符。参数 `name` 和 `namelen` 用于提供监听的端口和接口。通常这个地址设置为 `INADDR_ANY`，这表明连接可以在任何接口上接受。如果一个 multihomed 主机希望只在一个接口上接受连接，那么它可以指定那个接口的 IP 地址。通常 `namelen` 是 `socket_in` 结构的长度。

一旦本地地址绑定到套接字上，我们就必须启动套接字来监听连接请求，这可以通过 `listen` 系统调用完成。该系统调用通常被错误理解，它的唯一功能就是标记套接字正在监听。当一个连接请求到达主机时，系统内核在监听套接字列表上搜索，查找和请求里的目的地址和端口号相匹配的套接字：

```
#include <sys/socket.h>      /* UNIX */
#include <winsock2.h>        /* Windows */
int listen( SOCKET s, int backlog );
>Returns:0 on success, -1 (UNIX) or SOCKET_ERROR (Windows) on error
```

我们希望参数 `s` 被标记为正在监听的套接字的描述符。参数 `backlog` 是可以等待连接的最大数目，但它不是一次在给定端口可以建立的连接的最大数目，而是可以放在队列中等待应用程序来接受它们的连接或部分连接的最大数目（请参阅后面讲述的 `accept` 调用）。

通常参数 `backlog` 可以设置为大于 5，但是现代的实现工具为了支持繁忙的应用程序如 Web 服务器已经将这个最大数目改成很大了。大小是与系统有关的，我们必须检查一下系统文档来找出给定机器的合适值。如果我们指定一个大于最大数目的值，通常要做的操作是减少它到略少于最大数目。

最后的套接字调用是 `accept` 系统调用。它用于接受来自完全连接队列的一个连接。一旦连接被接受，该连接就可以用于数据传输，例如 `recv` 和 `send` 调用。如果成功返回，`accept` 返回一个新套接字的描述符，可以用它来进行数据传输。该套接字和监听套接字有相同的本地端口。本地地址是连接进来的接口。外部端口和地址是客户端连接的接口。

应当注意的是，这两个套接字都有相同的本地端口。因为 TCP 连接是用 4-tuple 来完全指定的，而 4-tuple 是由本地地址、本地端口、外部地址和外部端口组成的，所以这样的设置是不会出现任何问题的。因为两个套接字的（至少）外部地址和端口不同，所以系统内核可以区分它们：

```
#include <sys/socket.h>      /* UNIX */
#include <winsock2.h>          /* Windows */
SOCKET accept( SOCKET s, struct sockaddr *addr, int *addrlen );
>Returns:A connected socket if OK, -1 (UNIX) or INVALID_SOCKET (Windows) on failure
```

参数 s 是监听套接字的套接字描述符。如图 1.4 所示，accept 返回新连接的对等方地址到 addr 指向的 sockaddr_in 结构里。系统内核把该结构的长度放到 addrlen 指向的整数里。通常我们不关心对等方的地址，在这种情况下我们指定 addr 和 addrlen 为 NULL。

现在我们可以看看一个初步的服务器（图 1.5）。这又是一个“没有血肉”的程序，其目的是显示服务器程序的结构以及每个服务器程序必须进行的基本的套接字调用。与客户端代码和图 1.2 一样，我们可以注意到服务器代码严密地遵循着图 1.4 中概括的流程。

✓ 填充地址并获得一个套接字

12~20 我们用服务器的已知地址和端口号来填充 sockaddr_in 结构，local。使用它来调用 bind 函数。对于客户端，我们获得一个 SOCK_STREAM 套接字。这就是我们的监听套接字。

✓ 绑定已知的端口并调用 listen

21~32 我们把 local 中指定的端口号和地址绑定到监听套接字。然后调用 listen 来标记套接字为监听套接字。

✓ 接收连接

33~39 我们调用 accept 来接收新连接。accept 调用一直被阻塞直到一个新的连接已经准备好，然后为那个连接返回一个新套接字。

✓ 传输数据

39~49 首先从客户端读取并打印 1 个字节。然后，发送单一字节 2 给客户端并退出。

我们可以测试自己编写的客户端和服务器，在一个窗口中启动服务器而在另一个窗口中启动客户端。注意首先启动服务器是很重要的，否则客户端将失败并返回一个“连接拒绝”错误：

```
bsd: $ simplec
connect call failed: connection refused
bsd: $
```

因为没有服务器在端口 7500 上监听，而客户端却尝试连接到服务器，所以发生该错误。现在我们在启动客户端之前启动服务器开始监听：

```
bsd: $ simples | bsd: $ simplec
1 | 2
bsd: $ | bsd: $
```

小结

在本章中，我们为本书的其余部分介绍了一个路径图，并简要地回顾了基本的套接字 API。有了这些知识做铺垫，我们就可以继续向前，就可以自信我们已经有了基本的背景知识，就可以继续学习将要到来的各个主题了。

simples.c

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <stdio.h>

5 int main( void )
6 {
7     struct sockaddr_in local;
8     int s;
9     int s1;
10    int rc;
11    char buf[ 1 ];

12    local.sin_family = AF_INET;
13    local.sin_port = htons( 7500 );
14    local.sin_addr.s_addr = htonl( INADDR_ANY );
15    s = socket( AF_INET, SOCK_STREAM, 0 );
16    if ( s < 0 )
17    {
18        perror( "socket call failed" );
19        exit( 1 );
20    }
21    rc = bind( s, ( struct sockaddr * )&local, sizeof( local ) );
22    if ( rc < 0 )
23    {
24        perror( "bind call failure" );
```

```
25     exit( 1 );
26 }
27 rc = listen( s, 5 );
28 if ( rc )
29 {
30     perror( "listen call failed" );
31     exit( 1 );
32 }
33 s1 = accept( s, NULL, NULL );
34 if ( s1 < 0 )
35 {
36     perror( "accept call failed" );
37     exit( 1 );
38 }
39 rc = recv( s1, buf, 1, 0 );
40 if ( rc <= 0 )
41 {
42     perror( "recv call failed" );
43     exit( 1 );
44 }
45 printf( "%c\n", buf[ 0 ] );
46 rc = send( s1, "2", 1, 0 );
47 if ( rc <= 0 )
48     perror( "send call failed" );
49 exit( 0 );
50 }
```

simples.c

图 1.5 简单的 TCP 服务器

第二章

基 本 知 识

技巧 1 理解基于连接和无连接协议之间的差异

在网络编程领域中一个最基本的概念就是面向连接和无连接协议。尽管二者之间没有本质的区别，但它是一个在网络编程初学者中经常导致困惑的普遍存在的问题。问题的一部分是和所处的环境有关的：很明显的，两台需要通信的计算机在某种意义上来说必须是“连接的”，那么“无连接通信”又是什么意思呢？

答案是面向连接和无连接指的是协议。也就是说，这个术语适用于我们是如何通过物理介质来传输数据的，而不适用于物理介质本身。面向连接和无连接协议可以例行公事地同时共享同一个物理介质。

如果二者之间的差异和物理介质传输数据无关，那么它又和什么有关呢？它们之间根本的差异是，对于无连接协议，每一个数据包和另外的数据包都是独立地处理，而对于面向连接的协议，状态信息是被协议实现（implementation）在连续的数据包中维护的。

对于一个无连接协议来说，每个数据包称作数据报（datagram），它都是独立地编址并被应用程序发送出去（请参阅技巧 30）。从协议这个意义上说，每一个数据报是一个独立的实体，它和任何其他的在两个相同的对等方之间发送的数据报都无关。

这并不是说数据报在应用程序意义上是独立的。如果应用程序实现一些比简单的请求/应答协议复杂的功能，客户端发送一个请求到服务器并接收应答，然后它将可能需要维护两个数据报之间的状态。从这一点上来说，是应用程序而不是协议来维护状态信息。请参阅图 3.9，了解需要在来自客户端的数据报之间维护状态信息的无连接服务器的例子。

通常这意味着客户端和服务器不需要进行额外的会话——客户端发出一个请求，服务器返回一个应答。如果客户端在此之后发送另一个请求，那么协议会认为它是一个分开的事物，相对第一个是独立的。

这也意味着协议有可能是不可靠的。也就是说，网络尽最大的努力去传递每一个数据

报，但是难以保证它不会丢失、延迟或传递时顺序发生错误。

另一方面，面向连接的协议却在数据包之间维持着状态信息，应用程序使用这些状态信息来进行额外的会话。这些记住的状态信息使协议能够提供可靠的递交。例如，发送者可以记住什么数据已经发送出去了但是还没有被确认，还可以记住它是什么时候发送出去的。如果在一定的时间间隔后还没有接到应答，发送者就重传该数据。接收者可以记住什么数据已经接收到了，而且可以丢弃重复的数据。如果数据包没有按顺序到达，那么接收者可以先保存它，等待逻辑上先于它的数据包到达。

典型的面向连接的协议包括三个阶段。第一个阶段是对等双方建立连接；第二阶段是数据传输阶段，该阶段对等双方传输数据；最后第三阶段，当对等双方已经结束了数据传输时，就关闭连接。

一个标准的比喻是，使用面向连接的协议就像打电话，而使用无连接协议就像发送信件。当我们给朋友发信时，每一封信都各自写上地址，都是独立的实体。这些信件由邮局处理，不考虑通信者之间的任何其他信件。邮局不保存以前的通信记录——也就是说，不保存信件之间的状态信息。邮局也不保证我们的信件不会丢失、延时或不按顺序到达。这和无连接协议有着惊人的相似之处。

[Haverlock 2000]指出，明信片是一个更好的比喻，因为地址错误的信件会退回给发送者，而明信片（跟典型的无连接协议数据报一样）不会。

现在来看看在我们给朋友打电话而不是发信时会发生什么事情。我们通过拨朋友的电话号码来发起一个会话，朋友应答并说一些类似“Hello”的话，然后我们回答，“Hello, Sally. This is Harry”。我们和朋友聊一会儿，然后双方说再见并挂断电话。这和面向连接的协议很相似。在连接建立期间，发起方联系对等方，开始时互致问候，该阶段协商一些会话时需要使用的参数和选项，然后连接就进入到数据传输阶段。

在电话会话期间，双方都知道他是在跟谁说话，所以他们没有必要总是说“This is Harry saying something to Sally”之类的话。在我们每次说话前也不必拨朋友的电话号码——我们的电话已经建立了连接。与之相似的是在面向连接的协议的数据传输阶段，也没有必要指定我们的地址以及对等方的地址。这些地址是连接为我们维护的状态信息的一部分。我们只需要发送数据，而不必担心寻址或其他协议上的事情。

和电话会话一样，当连接的每一方结束了数据传输时，它就通知对等方。在双方都结束了数据传输时，它们就执行一个顺序的关闭连接。

这个比喻，尽管实用，却并不完美。对于电话系统来说，有一个实际的物理连接。而我们的“连接”却完全是概念上的——它仅仅包括每一方记住的状态信息。为了分析这个区别，请考虑以下空闲连接时一方主机崩溃并重启时会发生什么事情。还会存在一个连接吗？对重启的主机来说当然是没有的。它并不知道前面连接的任何事情。然而，它以前的对等方却仍然认为它自己还是连接的，因为它仍然保存着有关连接的状态信息，而且也不会发生任何事情使这些状态无效。

也许你会纳闷无连接的协议有许多缺点，为什么要使用它呢？我们将会看到，很多时候都会通过使用无连接协议来建立应用程序，使之更加合理。例如，无连接协议可以轻松地支持一对多和多对一的通信。而面向连接的协议通常却需要为此建立独立的连接。然而，更为重要的是，无连接的协议是面向连接的协议建立的基础。明确地说，为了把讨论返回到本书的主题上来，让我们来看看 TCP/IP 协议组。正如我们在技巧 14 里可以看到的那样，TCP/IP 是基于四层协议的堆栈，如图 2.1 所示。

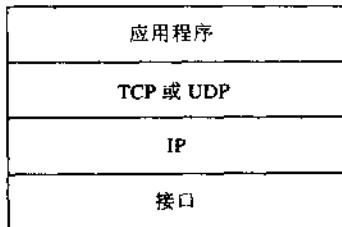


图 2.1 简化的 TCP/IP 协议栈

堆栈的最底层是接口层，它直接和硬件接口。堆栈的最顶层是应用程序，如 telnet、ftp 以及其他标准的和用户编写的应用程序。如图中所示，TCP 和 UDP 是建立在 IP 之上的。这样 IP 层是整个 TCP/IP 协议组建立的基础。然而 IP 层提供了一个很好的高效、不可靠、无连接的服务。它接收来自它上面各层的数据包，把它们封装到一个 IP 数据包里，并路由该数据包到正确的硬件接口上，硬件接口以它自己的方式把数据包发送出去。一旦发送出去，IP 就不再关心数据包。像所有的无连接协议一样，在数据包发送出去之后，就不会记住任何数据包的信息。

这种简单性也是 IP 的主要优点。因为它对下面的物理介质不做任何假定，IP 可以在任何能够传递数据包的物理连接上运行。例如，IP 可以运行在简单的串行链接上，运行在以太和令牌环局域网上，运行在 X.25 和异步传输模式（ATM）广域网上，运行在无线 Cellular Digital Packet Data（CDPD）网络上，以及运行在除此之外的许多其他的网络上。尽管这些网络技术之间有着巨大的差异，但 IP 总是平等地对待它们，就是知道它们可以路由数据包，也不在上面做任何假定。这种隐含的假设是很深奥的。既然 IP 可以运行在任何可以传输数据包的网络之上，那么整个 TCP/IP 协议组也是可以的。

现在让我们看看 TCP 是如何使用这个简单的无连接服务来提供可靠的面向连接的服务的。因为 TCP 发送它的数据包，在 IP 数据报里称作段（segments），这根本不能保证它们能够到达目的地，所以更不用说没有被破坏和顺序正确了。为了提供可靠性，TCP 在基本的 IP 服务里增加了三个服务。首先，它为 TCP 段提供了校验位。这就能保证到达目的地的数据不会在网络上传输时被破坏；第二，它为每个字节分配一个序列号，如果数据不按顺序到达目的地，那么接收者也可以重新把它们组合。

当然，TCP 并不为每一个字节附加一个序列号。所发生的事情是，每一个 TCP 段的头中包含了该段中第一个被序列化的编号。段中其他字节的序列号就可以隐含地知道了。

第三，TCP 提供了一个确认和重传的机制来保证每一个段最终都会被递交到达目的地。

确认/重传机制是到目前为止我们所讨论的三个附加服务中最复杂的一个，因此让我们来分析一下它是如何工作的。

我们在这里忽略了几何细节。这里的讨论仅仅涉及到 TCP 协议许多细微的地方，以及它们是如何用来提供一个可靠和健壮的传输机制的。完整的细节可以查阅 RFC 793 [Postel 1981b] 和 RFC 1122 [Braden 1989]。比较容易理解的说明可以查阅 [Stevens 1994]。RFC 813 [Clark 1982] 从总体上讨论了 TCP 窗口和确认策略。

TCP 连接的每一方维护着一个接收窗口（receive window），该窗口的范围是将要接收对等方的数据的序列号。最低值，代表的是窗口的左边，是下一个将要接收字节的序列号。最高值，代表的是窗口的右边，是 TCP 在缓存空间里已经有的最大编号的字节。使用接收窗口而不仅仅是下一个预计字节的记数，可以通过提供流量控制来增加稳定性。流量控制机制防止 TCP 超过通信对等方的缓冲区空间。

当一个 TCP 段到达时，任何超出接收窗口的带序列号的数据将被丢弃，包括提前到达的数据（这些数据的序列号位于接收窗口的左边）以及没有可获得的缓冲空间的数据（这些数据的序列号位于接收窗口的右边）。如果段中第一个可获得的字节不是下一个预计的字节，那么这个段就不是按顺序到达的，这时大多数的 TCP 具体实现把它放到队列中等待丢失数据的到来。如果段中第一个可获得的字节正是下一个预计的字节，那么数据就可以被应用程序获得，并且下一个预计字节的序列号通过增加段中可接受字节的序列号来更新。也就是说，窗口沿着可接受字节的数目向右滑动。最后，TCP 发送 ACK 信号给对等方，该信号带有下一个预计接收字节的序列号。

例如，在图 2.2 (A) 中，用虚线框起来的方框代表的是接收窗口，从中可以看出下一个预计接收的字节的序列号是 4，可以看出 TCP 希望接收 9 个字节（4~12）。图 2.2 (B) 显示的是字节 4、5、6 和 7 接收之后的接收窗口，该窗口已经向右滑动了 4 个序列号，并且 TCP 的 ACK 指定它预计接收的下一个字节的序列号为 8。



图 2.2 TCP 接收窗口

现在让我们从 TCP 的发送方的观点来看一看相同的情形。除了接收窗口之外，每一个 TCP 同时还维护着一个发送窗口。发送窗口分成两部分：已经发送但是没有被确认的字节，和可以发送但是还没有发送出去的字节。假定字节 1 到 3 已经被确认了，对于图 2.2 (A) 中接收窗口的发送窗口如图 2.3 (A) 所示。字节 4 到 7 发送之后且在确认之

前,发送窗口如图 2.3 (B) 所示。TCP 仍然可以接着发送字节 8 到 12 而不需要对等方的确认。在字节 4 到 7 发送之后, TCP 启动一个重传超时 (retransmission timeout, RTO) 计时器。如果在计时器超时之前字节 4 到 7 还没有得到确认, TCP 就认为它们丢失了, 并把它们重传一次。

因为许多的具体实现不知道哪一个字节是在一个特殊段里发送的, 所以重传的段就有可能包含比原来更多的字节。例如, 如果字节 8 和 9 是在 RTO 计时器超时之前发送的, 具体实现将重传字节 4 到 9。

应当注意的是, RTO 计时器超时了并不意味着原始数据没有到达目的地。它有可能是因为 ACK 丢失造成的, 也有可能是因为原始段在网络中延时过长而在 ACK 到达之前 RTO 计时器就已经超时了造成的。然而, 这些情况不会造成任何问题, 这是因为如果原始数据确实到达了, 那么重传的数据将会超出接收 TCP 方的接收窗口范围, 并被丢弃。

当字节 4 到 7 被确认了, 发送 TCP 方就丢弃它们, 并向右滑动发送窗口, 如图 2.3 (C) 所示。

对于应用程序编程人员来说, TCP 提供了一个可靠的面向连接的协议。请参阅技巧 9, 进一步了解有关可靠性的内容。

另一方面, UDP 提供给应用程序编程人员一个不可靠的无连接服务。实际上, UDP 只是在 IP 协议下面的层次上增加了两个东西。首先, 它提供了一个可选择的校验和来帮助检测数据损坏。尽管 IP 也有检验和, 但是它仅仅计算 IP 数据包的报头部分, 因为这个原因, TCP 和 UDP 也提供了一个校验和来保护它们自己的报头和数据。UDP 增加给 IP 的第二个特性是端口的概念。

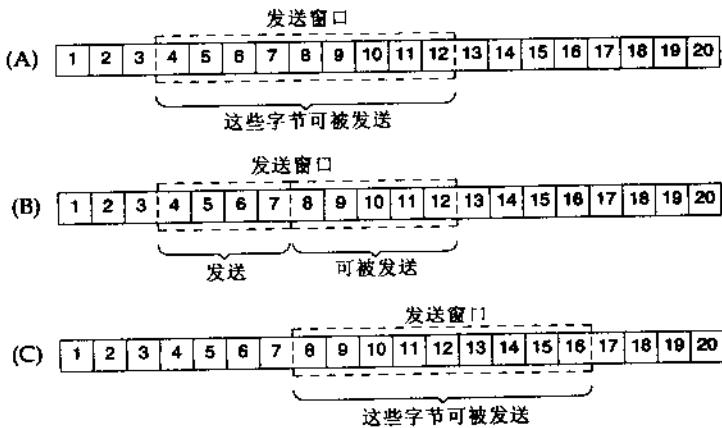


图 2.3 TCP 发送窗口

IP 地址 (该地址通常是在 Internet 标准段以点分隔的十进制表示法中使用, 请参阅技巧 2) 用于把 IP 数据报传递给一个特定的主机。当数据报到达目的主机时, 主机仍然需要把这个数据转交给适当的应用程序。例如, 一个 UDP 数据包可能是用于 echo 服务的, 而

另一个数据包是用于定时服务的。端口提供了一个多路输出数据到适当的应用程序的方法。每一个 TCP 和 UDP 套接字都有一个端口号关联到它上面。应用程序可以通过 bind 调用来显式地设置这个端口，它也可以让操作系统来给它指定一个端口。当数据包到达时，操作系统内核在套接字列表中搜索一个关联到数据包中协议、地址和端口号的套接字。如果匹配，数据就由指定的协议来处理（在这里我们考虑的是 TCP 或 UDP），并使该套接字可以被任何打开了相应套接字的应用程序获得。

如果多个进程或线程打开了一个套接字，那么任何进程或线程都可以读取数据，但是数据一旦读取出来了，对其他的进程或线程来说就再也不可获得。

回到我们的电话/信件的比喻，可以把 TCP 连接中的网络地址看成是办公室里接线总机上的电话号码，而把端口号理解为被呼叫的办公室中特定电话的分机号。同样地，UDP 地址可以看成是公寓楼的地址，而端口号就是公寓楼大厅中各个邮箱中的一个。

小结

我们在本节中探讨了无连接和面向连接协议之间的差异。我们知道不可靠的无连接数据报协议是可靠的面向连接协议构造的基础，我们还讨论了可靠的 TCP 协议是如何建立在不可靠的 IP 协议之上的。

我们也注意到，有的 TCP 连接就完全是概念上的。它由终端保持的状态信息组成，但是不包括“物理的”连接，正如我们打电话一样。

技巧 2 理解子网和 CIDR

一个 IP（版本 4）地址为 32 位长。通常以点分隔的十进制表示法把它们写出来，在这种表示法中，4 字节中的每一个都用一个以点分隔的十进制数字表示。这样地址 0x11345678 可以写为 17.52.86.120。有时因为许多 TCP/IP 具体实现使用标准的 C 语言惯例，以零开头的数字是八进制数字，这时候就需要特别小心。对于这些系统，17.52.86.120 和 017.52.86.120 是不一样的，前者是网络 17 而后者是网络 15。

分类地址

通常情况下，IP 地址分为 5 个类，如图 2.4 所示。这种分割称作分类地址。

D 类地址用于多点广播地址，E 类地址为将来使用而保留。剩下的 A 类地址、B 类地址和 C 类地址用于标识单个的网络和主机。

一个地址的分类是以开始的为 1 的位的个数来标识的。A 类地址的开头有 0 个为 1 的位，B 类地址的开头有 1 个为 1 的位，C 类地址有两个，以此类推。因为余下来的位的解释依赖于地址的类别，所以标识地址的类别是很重要的。

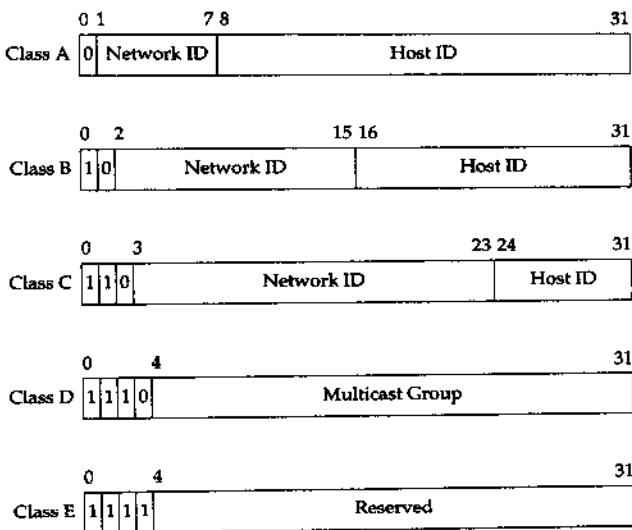


图 2.4 IP 分类地址

A 类地址、B 类地址和 C 类地址的剩下来的位分成两部分。第一部分是网络 ID，它标识该地址所指向的特定网络。在这之后是主机 ID，它标识网络中的一台特定的主机。

类标识位也被看作网络 ID 的一部分。这样，130.50.10.200 是一个 B 类地址，其网络 ID 是 0x8232。

把地址空间分成几类的原因是为了在不浪费地址空间的情况下尽可能地提供灵活性。例如，A 类地址用于具有大量主机（16,777,214）的网络。

有 2^4 台即 16,777,216 台可能的主机 ID，但是主机 0 和 ID 的所有位都为 1 的主机有着特别的含义。ID 中所有位都为 1 的主机是广播地址。发向广播地址的 IP 数据报交付给网络中的所有主机。ID 为 0 的主机的意思是“该主机”仅被一个主机用于作为一个源地址在启动时来找到它的主机号。因此，该主机号总是 $2^n - 2$ ，其中 n 是主机 ID 中位的个数。

因为网络 ID 为 7 位，所以总共有 128 个 A 类网络。

和主机 ID 一样，这些网络中的两个被保留。网络 0 的意思是“该网络”和主机 0 一样仅用于作为一个源地址在启动时来找到网络号。网络 127 是主机的一个内部网络。发向网络 127 的数据报都不能离开源主机，它通常是指 loopback 地址，因为发向它的数据报都被“循环返回”到同一个主机。

跟 A 类网络对应的是 C 类网络。C 类网络的数量很多，但是每一个都只有 254 个可获得的主机 ID。二者相反的是，A 类地址意味着包含大量主机的很少的网络，C 类地址意味着包含很少主机的大量网络。

图 2.5 显示了每一个类中的可能的网络和主机数量，以及其地址范围。我们把网络 127

归入 A 类地址，但是它却不是全局可获得的。

类	网络数	主机数	地址范围
A	127	16777214	0.0.0.1 到 127.255.255.255
B	16384	65534	128.0.0.0 到 191.255.255.255
C	2097252	254	192.0.0.0 到 223.255.255.255

图 2.5 A、B 和 C 类地址的网络、主机以及范围

TCP/IP 协议的设计者在开始时认为应当有数以百计的网络和应当有数以千计的主机。

实际上，如[Huiteema 1995]中讲述的那样，原始的设计仅有我们现在所指的 A 类地址。分为三类是后来为了容纳多于 256 个网络而做的修改。

正如我们所知道的那样，廉价并且无处不在的 PC 的出现导致网络和主机的数量的爆炸式增长。现在 Internet 的规模已经大大超出了它的设计者们的想象。

这样迅速地增长已经暴露了分类地址的一些缺陷。首先的问题是 A 类和 B 类网络的主机数量太多。注意网络 ID 应该是指一个物理网络，如 LAN。但是没有人会在一个单一的物理网络中放置 65,000 台主机而不管余下的 16000000 台。实际上，大型网络都分成通过路由器连接的几个小的段。

作为一个（很）简单的例子，考虑如图 2.6 所示的两个段。

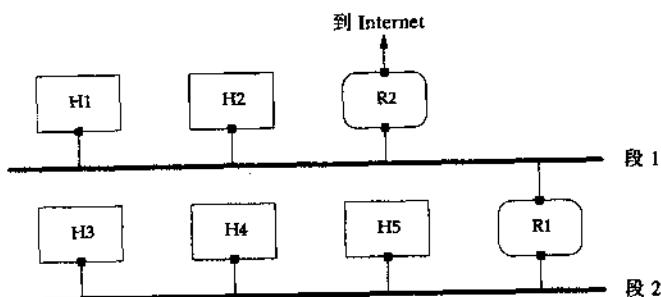


图 2.6 一个分段的网络

如果主机 H1 希望和 H2 通信，它只需要通过使用任何适于下面的网络的方法来映射 H2 的 IP 地址到它的物理地址，并把 IP 数据报“放在电缆上”。

现在假定主机 H1 希望和 H3 通信。H1 不能直接发送一个 IP 数据报给 H3，甚至在 H1 知道或能够找到 H3 的物理地址时也是这样，这是因为 H1 和 H3 是在分开的网络上——如分开的以太网电缆。结果是，H1 必须间接地通过路由器 R1 发送数据报。如果两个段有不同的网络 ID，H1 只需要询问它的路由表来决定使用路由器 R1 处理给段 2 的数据包，并把数据报通过发送给 R1 再转发给 H3。

给两个段分配不同的网络 ID 还有没有其他的方法呢？在标准的分类地址的情况下，有

两种可能。一个就是 H1 的路由表中为段 2 中的每一个主机包含一个条目，指定 R1 作为通向那个主机的中继。段 1 中的每一台主机都必须驻留这个相同的路由表，并且在段 2 中的每一台主机上都必须驻留指定段 1 中主机的相似的路由表。显然该方案不能很好地扩展到很多的主机上。而且，路由表必须手工维护，这很快就可能成为一个管理上的梦魇。由于这些原因，该方案实际上从来都没有采用过。

另外一个可能的方案称作代理 ARP (proxy ARP)，它让 R1 为段 1 假扮 H3、H4 和 H5，为段 2 假扮 H1、H2 和 R2。

代理 ARP 也称作混合地址解析协议和 (promiscuous ARP) ARP 程序。

该方案仅当下面的网络使用地址解析协议 (ARP) 映射 IP 地址为物理地址时才有效。通过 ARP，一个希望映射 IP 地址到物理地址的主机广播一条消息，询问具有目的 IP 地址的主机发送回它的物理地址。ARP 请求被网络中的所有主机获得，当然只有具有正确的 IP 地址的主机才会应答。

通过代理 ARP，当 H1 希望发送一个 IP 数据报给 H3，但不知道 H3 的物理地址时，它就发送一个 ARP 请求询问 H3 的物理地址。因为 H3 位于其他的网络上，它不能接收 ARP 请求，但是充当 H3 的代理的 R1 接收到请求，并用它自己的地址响应。当发向 H3 的数据报到达时，R1 把数据报转发给 H3。对于 H1 来说，就好像 H3 位于同一个物理网络中一样。

正如我们前面提到的，代理 ARP 仅在那些使用 ARP 的网络中有效，它还仅在相对简单的网络拓扑中有效。例如，如果有多个路由器连接段 1 和段 2，设想一下将会发生什么。

在本小节的讨论中，似乎对有多个段的网络的最通常的解决方案就是为每个段分配独立的网络 ID，但是这也带来了它自己的问题。首先，这可能在每个网络中浪费很多 IP 地址。例如，如果多段网络中的每个段有它自己的 B 类地址，那么和每一个 B 类网络地址关联的大多数 IP 地址将会浪费掉。记住，B 类网络地址中可以有 16,384 个主机地址。

其次，直接路由数据报到联合网络的节点需要在它的每一个段的路由表上有一个独立的条目。对我们的小例子来说，这不是什么大问题，但是考虑一下一个有成千上万个段组成的网络，然后考虑一下很多的这些网络，我们将会发现路由表将会变得十分庞大。

这是一个比开始时出现的问题还大的问题。路由器通常只有有限的内存，并把路由表放在接口板上的专用内存中。参阅[Huitema 1995]，了解一些现实世界中由于路由表的增长而导致路由失败中的例子。

请注意如果只有一个网络 ID，这两个问题是如何解决的。任何 IP 地址都不会永久地被闲置，这是因为当需要更多的主机时，我们总是可以再加入一个段来利用它们。在只有一个网络 ID 的网络中，发送 IP 数据报到网络中的任何主机只需要一个路由表条目。

2 子网划分

我们希望有一种解决方法，它包含少量的路由表和由单一网络 ID 连接起来的简易并有

效的 IP 地址空间，通过为每一个段分配独立的网络 ID 来达到目的。我们希望对于外部主机来说这就是单一的网络，但是对于内部主机来说有很多的网络——每个段都有一个网络。

实现这个目标的机制称作子网划分（subnetting）。这个想法令人惊奇地简单。因为外部主机仅使用 IP 地址的网络 ID 来决定路由，主机 ID 的分配可以由系统管理员选择方便的方法来进行。主机 ID 实际上是包含内部结构的 cookie，对于外部网络来说这没有任何意义。

子网划分是通过使用部分主机 ID 指定段（也就是子网）来进行的，外部的主机连接的是段，其余的主机 ID 用来标识特定的主机。例如，考虑 B 类网络 190.59.0.0。我们应该选择使用地址的第三个字节作为子网 ID，并且把第四个字节作为子网中的主机号。图 2.7 (A) 是从网络外部看到的地址结构。主机 ID 是一个不透明的域，它没有明显的子结构。图 2.7 (B) 中，我们看到的是同从内部网络中看到的一样的地址结构，在本图中，主机 ID 域就是由子网 ID 和主机号组成的。

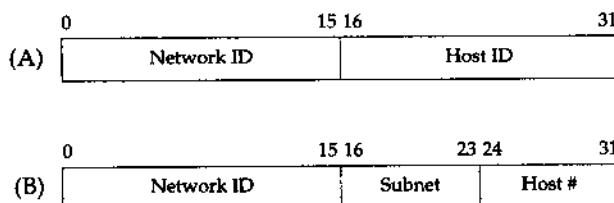


图 2.7 B 类地址的两个视图

尽管我们已经显示了一个 B 类地址，并且把主机 ID 域以字节划分，但是它们都不是必须的。A 类、B 类和 C 类地址可以而且也确实是划分成子网的，它们的划分经常不是以字节为边界的（当然，这对于 C 类地址来说是不可能的）。和每个子网关联的是一个表示连接地址的网络和子网部分的子网掩码。例如，图 2.7B 中的子网掩码必须是 0xffffffff00。该掩码通常写成以点分隔的十进制形式（255.255.255.0），但是如果划分不是以字节为边界的，通常第一种形式更容易使用。

应当注意的是，虽然它称为子网掩码，它实际上指定了地址的网络和子网的部分——也就是说除了主机部分的所有东西。

为了阐明这些观点，让我们假定子网 ID 是 10 位而主机数为 6 位，那么子网掩码必须为 255.255.255.192（0xffffffffc0）。图 2.8 说明了该子网掩码是如何应用于地址 190.50.7.75 来获得网络/子网号 190.50.7.64 的。

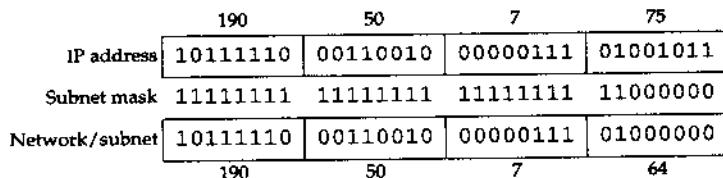


图 2.8 子网掩码按位相与来获取 IP 地址的网络部分

我们可以通过验证地址 190.50.7.75 是网络 190.50.0.0 中子网 29 的第 11 个主机来检验我们的假定。记住这种表示只是在网络内部才知道，而在网络外部，该地址被解释为网络 190.50.0.0 中的第 1867 个主机。

现在让我们来考虑图 2.6 中的路由器是如何利用主机 ID 的子结构来在网络内部路由数据报的。为了方便标识子网 ID，让我们假定我们有一个 B 类网络 190.50.0.0，其子网掩码是 255.255.255.0。这是图 2.7 (B) 中显示的结构。

在图 2.9 中，我们指定图 2.6 中的第一个段的子网 ID 为 1，第二个段的子网 ID 为 2。我们用主机的 IP 地址来标识每一个主机接口。应当注意的是，每一个地址的第三个字节是附加接口的子网。这种表示当然不被网络外部知道。

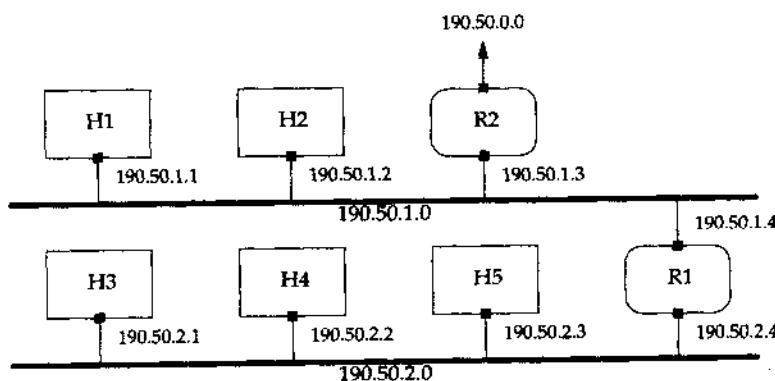


图 2.9 带子网的网络

回忆我们前面的讨论，让我们首先看看主机 H1 想和主机 H3 通信时会发生些什么。H1 获得 H3 的地址 190.50.2.1，把它和子网掩码 255.255.255.0 进行按位相与，产生 190.50.2.0。因为主机 H1 位于子网 190.50.1.0，它知道主机 H3 不能直接连接到，所以查询它的路由表来找到下一个中继时的路由器 R1。

许多具体实现通过在路由表中保存两个子网的子网掩码来合并这两个步骤。在查询路由时，IP 可以发现可以直接连接目的网络，或者需要发送数据报给中间的路由器。

然后主机 H1 映射 R1 的 IP 地址到它的物理地址（例如使用 ARP）并发送数据报给 R1。R1 在它的路由表中查找目的地址，再次使用子网掩码，并发现主机 H3 位于连接接口 190.50.2.4 的子网上。最后，R1 通过映射它的 IP 地址到物理地址来传递数据报给 H3，并把数据报发送到它的 190.50.2.4 接口上。

现在假定 H1 希望发送一个数据报给 H2。当子网掩码应用于 H2 的地址（190.50.1.2）时，它就产生 190.50.1.0，这和 H1 的子网相同。所以，H1 仅需要映射 H2 的 IP 地址到它的物理地址上并把数据报直接发送给 H2。

下面，让我们看看外部网络中的主机 E 想要发送一个数据报给主机 H3 将会发生什么。

事情。因为 190.50.2.1 是一个 B 类地址，所以位于主机 E 的网络边缘的路由器知道 H3 驻留在网络 190.50.0.0 上。因为 R2 是该网络的网关，所以来自主机 E 的数据报最终要到达 R2。现在剩下的步骤和主机 H1 发起的数据报的传递是一样的：R2 使用子网掩码并取出子网号 190.50.2.0，决定下一个中继是 R1，并发送数据报给 R1，R1 把它递交给主机 H3。应当注意的是，主机 E 并不知道网络 190.50.0.0 的内部拓扑结构。它仅仅把它的数据报发送给网关 R2。R2 和网络内部的其他主机才知道子网和路由的信息，这些信息是传送数据到它们时所必须的。

到目前为止我们忽略掉的一个重要的问题是子网掩码是和接口关联的，因此它也是和路由表中的条目相关联的。这意味着不同的子网具有不同的子网掩码是可能的。

举一个例子可以弄清楚这个问题。假定我们的 B 类地址 190.50.0.0 是用于一所大学的，大学里的各个系都分配了一个子网掩码为 255.255.255.0 的子网（图 2.9 仅显示了这个大规模网络的一部分）。计算机科学系分配的是子网 5，其系统管理员决定让学生计算机实验室和系里其他部分有独立的以太网段。他可以向大学的管理部门申请另一个子网号，但是学生计算机实验室仅有几台机器，所以不值得分配整个 C 类地址块给它。这样，管理员希望划分它的子网为两个段。实际上，他希望在子网内创建子网。

为了做到这一点，他增加子网域为 10 位，并使用子网掩码 255.255.255.192。结果地址结构如图 2.10 所示。

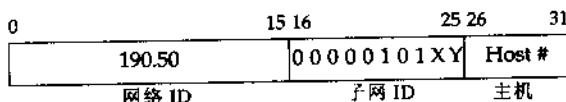


图 2.10 子网 190.50.5.0 的地址结构

子网 ID 的前面 8 位总是 0000 0101 (5)，因为网络的其他部分标识整个子网为子网 5。X 和 Y 位指定地址是属于子网 190.50.5.0 中哪一个以太段的呢？如图 2.11 所示，当 XY=10 时，我们在学生实验室的以太网中寻址，当 XY=01 时，我们在网络中的其他部分中寻址。190.50.5.0 子网的部分拓扑结构如图 2.11 所示。

图 2.11 中最上面的段（子网 190.50.1.0）由路由器 R2 提供外部访问，如图 2.9 所示。子网 190.50.2.0 在图 2.9 中没有显示出来。中间段（子网 190.50.5.128）是学生计算机实验室的以太网。底部段（子网 190.50.64）是系里其他部分的以太网。为了简明起见，每一个机器的主机号对于它的所有接口来说都是一样的，并且方框内的数字代表的是主机或路由器。

接口连接 190.50.5.64 和 190.50.5.128 子网的子网掩码是 255.255.255.192。接口连接 190.50.1.0 子网的子网掩码是 255.255.255.0。

注意这个情况和前面图 2.9 中讨论的情况是如此的相似。正如网络 190.50.0.0 之外的主机不知道第三个字节指定了一个子网一样，网络 190.50.0.0 内子网 190.50.5.0 外的主机也不知道第四字节的头两位是指定子网 190.50.5.0 中的一个子网。

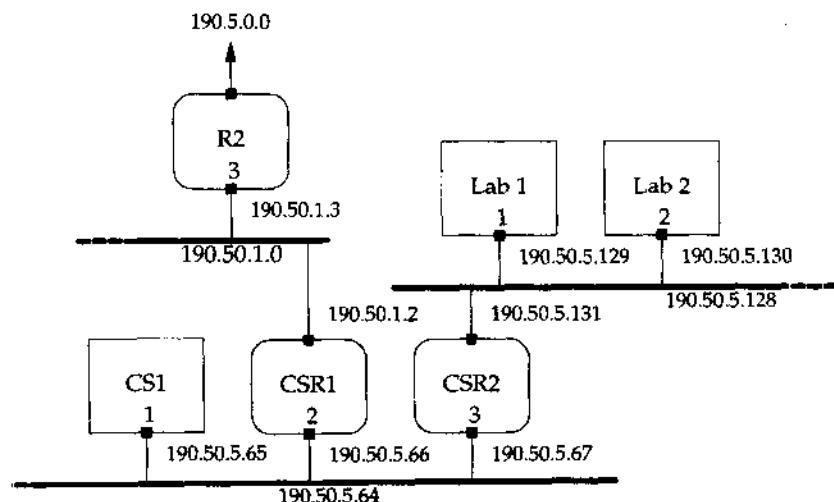


图 2.11 子网的子网

在结束子网的主题之前，让我们简单地讨论一下广播地址。在使用子网时，有四个不同类型的广播地址：有限广播、网络直接广播、子网直接广播和全子网直接广播。

有限广播

有限广播地址是 255.255.255.255。称它为有限是因为使用它的数据报不会被路由器转发。它们总是限制在本地网线之内。它基本上是用在启动时，这个时候主机不知道它的 IP 地址或子网掩码。

当主机使用有限广播地址时发生什么事情是与具体实现有关的。许多具体实现仅在一个接口上发送出数据报。该系统上需要在多个接口上广播的应用程序必须询问操作系统支持广播的配置接口的情况。

网络直接广播

网络直接广播的地址设置为指定网络的网络 ID 以及其主机 ID 的所有位都设置为 1。例如，网络 190.50.0.0 的网络直接广播地址是 190.50.255.255。带有这种类型广播地址的数据报传递给目的网络中的所有主机。

路由器需求 RFC (RFC 1812) [Baker 1995]，要求路由器默认地转发直接广播数据报，但是也有一个选项用来禁止转发。在那些可以利用直接广播的拒绝服务攻击者的眼里，似乎许多路由器都已经禁止转发这些数据报了。

子网直接广播

一个子网直接广播地址指定了网络 ID 和子网 ID。主机号部分设置为全 1。应当注意的是，在不知道子网掩码的情况下是不可能知道一个地址是否为子网直接广播地址的。例如，

如果路由器知道了网掩码是 255.255.255.0，就会把地址 190.50.1.255 看作子网直接广播消息。如果路由器认为子网掩码是 255.255.0.0，那么该地址就会看成是一个广播地址。

对于将会在后面讨论的 CIDR，这种类型的广播地址和网络直接广播地址是不可区分的，RFC 1812 把它们同等对待。

全子网直接广播

这种类型的广播地址指定网络地址，并把子网和主机部分设置为全 1。对于子网直接广播地址，在区别这个地址时必须运用子网掩码的知识。

不幸的是，全子网直接广播地址的使用受到某些失败模式的影响，并且它从来都没有实现过或配置过。对于 CIDR，这种形式的广播地址已经不再使用了，引用 RFC 1812 的话来说，“现在已经转移到历史的垃圾箱里去了”。

所有的这些广播地址都不会用于 IP 数据报的源地址。最后，应当提醒的是早期的 TCP/IP 具体实现，如 4.2BSD，是使用主机 ID 域全为 0 来指定广播地址的。

CIDR

我们已经知道子网划分解决了分类地址的一个问题：路由表条目的无限增长。它通过更好地利用给定网络内的主机 ID 来帮助 IP 地址的消耗减少到很小的程度。

另一个分类地址的严重问题是 B 类网络 ID 的消耗。如图 2.5 所示，B 类地址可获得不多于 17,000 个网络 ID。因为大多数的中等规模以及更大规模的组织需要比 C 类网络 ID 所提供的更多的 IP 地址，所以它们获取 B 类网络 ID。

随着 B 类网络 ID 的消耗，各个组织不得不获取 C 类网络 ID 块，但是这将再次导致需要引入子网划分来解决的问题：Internet 上路由表的增长。

不分类域间路由（classless interdomain routing，CIDR）通过“反向操作”子网划分来解决这个问题。在子网划分中，IP 地址的网络 ID 部分变长了，而 CIDR 使它变短。

让我们来看看 CIDR 是如何工作的。一个需要 1,000 个 IP 地址的组织将会分配共享相同前缀的 4 个连续的 C 类网络 ID。例如，他们可能分配得到从 200.10.4.0 到 200.10.7.0 的 IP 地址。这些网络 ID 的前 22 位都是一样的，并且是作为聚集网络的网络号，在这个例子中是 200.10.4.0。对于子网划分，网络掩码用于标识 IP 地址的网络部分。在前面介绍的聚集网络的例子中，该掩码应当是 255.255.252.0（0xfffffc00）。

跟子网划分不一样的是，这个网络掩码不扩展 IP 地址的网络部分，而是缩短它。因此，CIDR 也称作 supernetting。另外跟子网划分不一样的是，网络掩码向外导出网络部分。它成为指向该网络的每个路由表条目的一部分。

假定一个外部的路由器 R 想转发一个数据报给该聚集网络中的一个主机 200.10.5.33。路由器 R 遍历它的路由表，其中的每一个条目都有一个网络掩码，并用 200.10.5.33 的掩码部分和条目做比较。如果路由器这个网络中的一个条目，它将是 200.10.4.0，网络掩码是 255.255.252.0。当 200.10.5.33 和这个掩码按位相与时，结果是 200.10.4.0，它和条目中的网

络号相匹配，因此路由器就知道了数据报的下一个中继了。

为了防止二义性，最终使用的是匹配得最长的条目。例如，一个路由器可能也有网络掩码为 255.255.0.0 格式为 200.10.0.0 的条目，该条目也和 200.10.5.33 相匹配，但是因为匹配的仅是 16 位而不是第一个条目的 22 位，于是采用第一个条目。

在 Internet 服务提供商 (ISP) 拥有所有前缀为 200.10 的 IP 地址时可能会发生这种情况。第一种类型的条目将路由所有的前缀为 200.10 的数据报给该 ISP。ISP 也可能为了避免额外的中继或因为别的原因而登记更准确的路由信息。

实际上，CIDR 比我们所谈到的情况稍微具有普遍性。称为“不分类”是因为它废除了分类的概念。这样，每个表条目都有一个关联的网络掩码，指定 IP 地址的网络部分。以分类地址的眼光看，掩码可能有缩短或加长地址的网络部分的功效，但是因为 CIDR 废除了分类的概念，我们不再把网络掩码看作简写或扩展，而仅仅把它看作用于指定地址的网络部分。

在实际中，“掩码”仅仅是一个数字，称做前缀 (prefix)，它指定地址的网络部分的以位计的长度。例如，我们前面描述的聚集网络的前缀是 22，我们可以把网络地址写作 200.10.4.0/22，“/22”表示的是前缀。由此看来，分类地址可以看成是 CIDR 的一种特殊情况，它有四种可获得的前缀，编码进入到地址的前面的位中。

CIDR 在指定网络地址长度中提供的很强的灵活性，通过允许地址成块分配并设计成符合网络的需要来帮助有效地分配 IP 地址。我们已经看到 CIDR 是如何能够聚集几个（这是过去经常谈到的）C 类网络为单一的规模更大的网络的。在另一个极端，我们可以使用它来分配一小部分（这也是过去经常谈到的）C 类网络给一个只需要少量地址的小网络。例如，ISP 可能分配给一个小公司单一的 LAN 网络 200.50.17.128/26，它可以支持最多 $2^6 - 2 = 62$ 台主机。

RFC 1518[Rekhter and Li 1993]，讨论了地址聚集以及它对路由表大小的影响。该文档同时也督促 IP 地址前缀（也就是地址的网络部分）应当以分层（hierarchical）的方式来分配，以利于这种聚集。

分层地址聚集可以比作分层文件系统，如 UNIX 和 Windows 中使用的文件系统。正如高层次子目录知道低层次的子目录但不知道低级子目录中的文件一样，高级路由域知道中间域，但是不知道域中单独的网络。例如，假定一个区域提供商传送 200/8 前缀的所有流量，并且拥有前缀 200.1/16、200.2/16 和 200.3/16 的 3 个本地 ISP 连接到它。每个 ISP 将有几个客户，部分地址空间分配给这些客户（如 200.1.5/24 等等）。区域提供商外部的路由器仅需要维护一个路由条目，就是 200/8，就可以到达 200/8 地址范围内的任何主机。他们可以不需要知道 200/8 地址空间是如何划分的就做出他们的路由决定。区域提供商仅需要三个路由条目，每个 ISP 一个。在最低层次：ISP 将为每一个客户提供一个路由表条目。虽然简单，这个例子却抓住了聚集的本质。

RFC 1518 很值得一读，因为它说明通过使用 CIDR 获得巨大的好处。

CIDR 本身以及它的基本原理，是在 RFC1519 中讲述的[Fuller et al. 1993]。该 RFC 也

包括一个详细的 CIDR 的成本/利益分析，并讨论了一些域间路由协议需要的变化。

» 子网划分和 CIDR 的状况

子网划分，如 RFC 950[Mogul and Postel 1985]中讲述的那样，是一个标准的协议（Std. 5）。同样地，运行 TCP/IP 的每个主机都需要支持它。

CIDR（RFC 1517[Hinden 1993], RFC 1518, RFC 1519）是一个建议的标准协议，所以并不是必须的。然而，CIDR 在 Internet 中的使用非常普遍，并且所有的新地址分配都是和它保持一致的。因特网工程指导小组（IESG）选择 CIDR 作为解决路由表大小问题的短期方案。

从长远来看，地址消耗和路由表大小问题都会被 IP 版本 6 解决。IPv6 有一个很大的地址空间（128 位）以及显式的层次结构。更大的地址空间（包括 64 位的接口 ID）保证了在可以预见的未来具有充足的 IP 地址，IPv6 的层次结构使路由表大小合理。

» 小结

在本小结中，我们学习了子网划分和 CIDR，并了解了它们是如何用来解决两个分类地址的问题。子网划分除帮助避免路由表的增长外同时还维护着灵活的寻址方式。CIDR 使地址可以有效地分配，并使地址的层次分配变得更容易。

技巧 3 理解私有地址和 NAT

在因特网普及之前，通常一个组织为它们的网络选择的是随机的 IP 地址。毕竟，这些网络都不会而且“永远也不会”连接到任何外部网络，所以 IP 地址的选择就没有看成是一个问题。当然，情况已经改变了，现在没有连接 Internet 的网络已经是很少了。

现在不再需要为私有网络选择随机的 IP 地址块。RFC 1918[Rekhter, Moskowitz et al. 1996]指定了三个保证永远都不会被分配的保留 IP 地址。这三个地址块是：

- 10.0.0~10.255.255.255 (10/8 前缀)
- 172.16.0.0~172.31.255.255 (172.16/12 前缀)
- 192.168.0.0~192.168.255.255 (192.168/16 前缀)

通过给网络使用这三个地址块之一，就有可能让网络中的任何一台主机都能访问网络中的任何其他主机，而不用担心和全局分配的 IP 地址相冲突。当然，只要网络保持和外部网络不连接，就不用管分配的是什么地址了，但是使用这些私有地址块是很简单的，而且它可以在网络连通 Internet 的那一天受到保护。

如果网络确实连接到外部了，那会发生什么呢？一个拥有私有 IP 地址的主机是如何能够和 Internet 或其他外部网络中的主机通信的呢？这个问题的最通俗的答案是使用网络地址转换（network address translation, NAT）。几种类型的设备支持 NAT，包括路由器、防火

墙和独立 NAT 产品。NAT 通过在私有网络地址和一个或多个全局分配 IP 地址之间解释地址来正常工作。让我们简单地看看它是如何工作的。大多数的 NAT 设备可以配置为支持 3 种模式：

(1) 静态模式——私有网络中所有主机中的一部分有它们的私有 IP 地址，这些私有 IP 地址映射到一个固定的全局分配的地址。

(2) 缓冲池模式——NAT 设备有一个可获得的全局分配的 IP 地址的缓冲池，并且它动态地分配这些地址给需要和外部网络中的主机进行通信的主机。

(3) PAT 模式——也称作端口地址转换，这种方法用于有单一的全局分配地址时，如图 2.12 中所示。通过 PAT，每个私有地址映射到同一个外部地址，但是往外传输的数据包的源端口变成唯一的值，该值用于关联进来的数据包和私有网络地址。

在图 2.12 中，我们显示了使用 10/8 地址块的只有三台主机的一个小网络。该网络同时也有路由器、带标签的 NAT，它有一个私有网络中的地址和一个 Internet 上的地址。

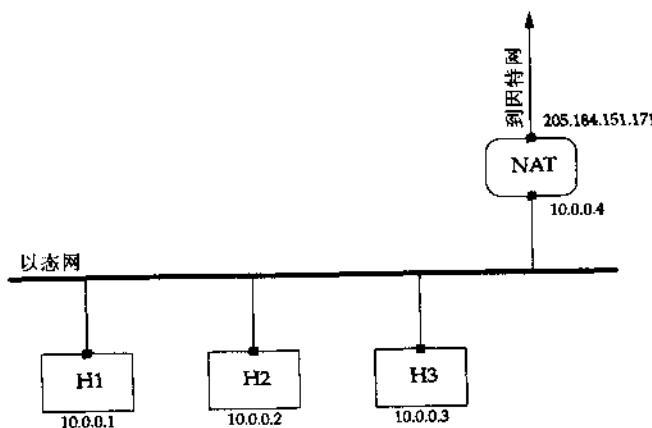


图 2.12 带 NAT 路由的私有网络

因为我们仅显示了和 NAT 关联的一个全局地址，让我们假定本讨论中遵循的配置为使用 PAT 模式。静态和缓冲池模式的情况与 PAT 模式相似，但是更简单，这是因为它们不需要在处理地址转换之时处理额外的端口转换。

让我们假设主机 H2 想发送一个 TCP SYN 段给 204.71.200.69，www.yahoo.com Web 服务器之一，打开一个连接。在图 2.13 (A) 中，我们看到从 H2 发送出来的段的目的地址是 204.71.200.69.80，源地址是 10.0.0.2.9600。

我们在这里使用的是标准的约定，IP 地址写成 A.B.C.D.P 的格式，A.B.C.D 是 IP 地址，而 P 是端口号。

除了源地址是一个私有网络地址之外，这本身没什么不一样。当这个段到达路由器的时候，NAT 必须改变它的源地址为 205.184.151.171，以让 Yahoo 的 Web 服务器知道向哪里发送 SYN/ACK 和其他应答。因为任何发向和来自私有网络上其他主机的 Internet 流量也

将把它们的地址转换成 205.184.151.171，所以 NAT 就必须把源端口转换成一个唯一的端口号，以让它可以发送该连接上进来的流量到正确的主机上。我们可以看到源端口 9400 映射到 5555。这样传递到 Yahoo 的段的目的地址为 204.71.200.69.80，源地址为 205.184.151.171.5555。

正如图 2.13 (B) 显示的那样，当 Yahoo 的应答到达路由器的时候，它的目的地址为 205.184.151.171.5555。NAT 在它的内部状态表中查找该端口，找出端口 5555 是和地址 10.0.0.1.9600 关联的，所以从路由器到主机 H2 的应答段的源地址为 204.71.200.69.80，目的地址为 10.0.0.1.9600。

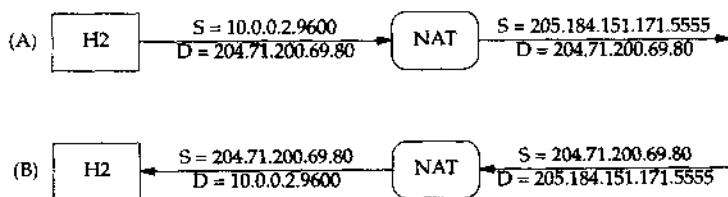


图 2.13 端口地址转换

尽管前面的讨论使 PAT 似乎看起来很直接，但是其中还有一些比它表面上看出来的东西复杂得多的细微之处。例如，改变源地址和端口号就会同时改变 IP 报头校验和以及 TCP 段校验和，所以这两个东西必须要同时调整。

下面的例子同样说明了这种转换是如何引起复杂性的，考虑一下文件传输协议（FTP）[Reynolds and Postel 11985]。当 FTP 客户端想从 FTP 服务器发送或接收一个文件时，它发送给服务器一个 PORT 命令，包含一个地址和端口号，它将在这个地址和端口号上监听来自服务器的（数据）连接。这就是说，首先 NAT 必须知道这个 TCP 段是一个 FTP PORT 命令，并且必须转换该地址和端口；其次，PORT 命令里的地址和端口号是作为 ASCII 字符串发送出去的，因此改变它们就可能改变段的大小。结果是，这就意味着序列号（请参阅技巧 1）将会被改变，而且 NAT 也不得不去跟踪它们，使 ACK 命令中的序列号可以得到调整，而且来自私有网络上主机的额外段可以调整它们的序列号。

尽管存在这些复杂性，NAT 工作得十分正常并被广泛地使用。特别是 PAT，它是小网络在只有一个可获得的 Internet 连接时自然要采用的方法。

小结

在本小节中，我们了解了 NAT 是如何让网络为内部主机使用私有网络地址块中的地址，但仍然可以使自由地和 Internet 进行通信成为可能。特别是 PAT，它对于那些只有一个全局可访问 IP 地址的小网络是十分有用的。不幸的是，因为 PAT 改变了向外发送的数据包的源端口号，所以它就有可能给那些在消息内部传递端口号信息的非标准协议带来诸多问题。

技巧 4 开发和使用应用程序“框架”

大多数的 TCP/IP 应用程序属于下面四个种类之一：

- (1) TCP 服务器
- (2) TCP 客户端
- (3) UDP 服务器
- (4) UDP 客户端

属于同一种类的应用程序具有几乎一样的“设置”代码来初始化它们的网络性质。例如，TCP 服务器必须用它想要的地址和端口号来填充 `sockaddr_in` 结构，获得一个 `SOCK_STREAM` 套接字，绑定它想要的地址和端口号到套接字上，设置 `SO_REUSEADDR` 套接字选项（请参阅技巧 23），调用 `listen`，然后通过 `accept` 调用准备接收一个连接（或多个连接）。

为了正确地完成工作，每一个步骤都需要为错误的返回做检查，而且地址和端口转换代码应当准备接收数字或符号的地址和端口号。这样，每一个 TCP 服务器为了完成这些任务都有几乎一样的按顺序的 100 行代码。处理这个问题的一种方法是抽取设置代码到一个或多个库函数中，应用程序可以调用它们。这是一个很好的策略，而且也是我们在本书中将要采用的一种方法，但是有时应用程序需要一个稍微不同的初始化序列。如果情况是这样的，我们就不得不从头开始，或者从库中抽取相关代码并做相应的改动。

为了应付这些情况，我们建立了一个应用程序框架，它在里面包含了所有必须的代码。我们可以拷贝这个框架到应用程序中，做一些必要的改动，然后继续应用程序本身。没有这样的一个框架，就很有可能采用诸如在应用程序中 hard-coding 地址的捷径（请参阅技巧 29）以及其他一些不好的习惯。一旦我们开发出了这些框架，我们就可以抽取一般的情况到一个库中并为特殊情况保存这个框架。

为了便于移植，我们定义了几个宏，隐藏了 UNIX 和 Windows API 之间的一些差别。例如，UNIX 中关闭套接字的系统调用是 `close`，而在 Windows 平台下，该系统调用是 `closesocket`。这些宏的 UNIX 版本如图 2.14 所示，Windows 版本的宏跟它相似，列在附录 B 中。我们的框架通过包含文件 `skel.h` 来访问这些宏。

skel.h

```

1 #ifndef __SKEL_H__
2 #define __SKEL_H__

3 /* UNIX version */

4 #define INIT()          ( program_name = \
5                           strrchr( argv[ 0 ], '/' ) ) ? \

```

```

6           program_name++ : \
7           ( program_name = argv[ 0 ] )
8 #define EXIT(s)          exit( s )
9 #define CLOSE(s)         if ( close( s ) ) error( 1, errno, \
10                      "close failed" )
11 #define set_errno(e)     errno = ( e )
12 #define isvalidsock(s)   ( ( s ) >= 0 )

13 typedef int SOCKET;

14 #endif /* __SKEL_H__ */

```

skel.h

图 2.14 skel.h

» TCP 服务器框架程序

让我们从 TCP 服务器的框架开始。完成框架之后，我们就可以通过从框架中抽取相应的代码来着手构造我们的类库。图 2.15 是框架的 main 函数的源代码。

tcpserver.skel

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdarg.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <netdb.h>
8 #include <fcntl.h>
9 #include <sys/time.h>
10 #include <sys/socket.h>
11 #include <netinet/in.h>
12 #include <arpa/inet.h>
13 #include "skel.h"

14 char *program_name;

```

```
15 int main( int argc, char **argv )
16 {
17     struct sockaddr_in local;
18     struct sockaddr_in peer;
19     char *hname;
20     char *sname;
21     int peerlen;
22     SOCKET s1;
23     SOCKET s;
24     const int on = 1;

25     INIT();

26     if ( argc == 2 )
27     {
28         hname = NULL;
29         sname = argv[ 1 ];
30     }
31     else
32     {
33         hname = argv[ 1 ];
34         sname = argv[ 2 ];
35     }

36     set_address( hname, sname, &local, "tcp" );
37     s = socket( AF_INET, SOCK_STREAM, 0 );
38     if ( !isValidsock( s ) )
39         error( 1, errno, "socket call failed" );

40     if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR, &on,
41                     sizeof( on ) ) )
42         error( 1, errno, "setsockopt failed" );

43     if ( bind( s, ( struct sockaddr * ) &local,
44                 sizeof( local ) ) )
45         error( 1, errno, "bind failed" );
```

```

46 if ( listen( s, NLISTEN ) )
47     error( 1, errno, "listen failed" );
48 do
49 {
50     peerlen = sizeof( peer );
51     s1 = accept( s, ( struct sockaddr * )&peer, &peerlen );
52     if ( !isValidsock( s1 ) )
53         error( 1, errno, "accept failed" );
54     server( s1, &peer );
55     CLOSE( s1 );
56 } while ( 1 );
57 EXIT( 0 );
58 }

```

*tcpserver.skel*图 2.15 *tcpserver.skel* 的 main 函数

包含文件和全局变量

1~14 我们为将要使用的标准函数包含头文件。

25 INIT 宏执行标准的初始化操作，如为 error 设置全局变量 `program_name`，而且如果是运行在 Microsoft Windows 平台下的话我们就调用 `WSAStartup`。

main

26~35 我们希望服务器用一个地址和端口号或仅仅是一个端口号来调用。如果没有指定地址，`INADDR_ANY` 就绑定到套接字上，从而接收任何接口上的连接。当然，在一个实际的应用程序中，有可能有其他的命令行参数，我们应当在这里处理它们。

36 我们调用 `set_address`，它用请求的地址和端口号来填充 `sockaddr_in` 结构 `local` 变量。`set_address` 函数如图 2.16 所示。

37~45 然后我们得到一个套接字，设置 `SO_REUSEADDR` 选项（请参阅技巧 23），并绑定 `local` 中的地址和端口号到它上面。

46~47 然后通过调用 `listen` 来通知操作系统内核，我们已经准备好了，可以接收来自客户端的连接。

48~56 然后接收连接请求，并为每一个连接调用 `server`。函数 `server` 可以处理连接，把它作为服务器进程的一部分，或者它 `fork` 一个子进程来处理它。在这两种情况中，函数 `server` 返回之后就关闭连接。那个不同寻常的 `do-while` 结构为服务器在第一个连接到来之后提供了一个简单的退出方法。我们仅需要改变

```
while(1);
```

为

```
while(0);
```

下面我们看看 `set_address` 函数。我们将在所有的框架程序中使用这个函数，所以它是通用库函数中自然而然的候选者。

tcpserver.skel

```

1 static void set_address( char *hname, char *sname,
2   struct sockaddr_in *sap, char *protocol )
3 {
4   struct servent *sp;
5   struct hostent *hp;
6   char *endptr;
7   short port;
8
9   bzero( sap, sizeof( *sap ) );
10  if ( hname != NULL )
11  {
12    if ( !inet_aton( hname, &sap->sin_addr ) )
13    {
14      hp = gethostbyname( hname );
15      if ( hp == NULL )
16        error( 1, 0, "unknown host: %s\n", hname );
17      sap->sin_addr = *( struct in_addr * )hp->h_addr;
18    }
19  }
20 else
21   sap->sin_addr.s_addr = htonl( INADDR_ANY );
22 port = strtol( sname, &endptr, 0 );
23 if ( *endptr == '\0' )
24   sap->sin_port = htons( port );
25 else
26 {
27   sp = getservbyname( sname, protocol );
28   if ( sp == NULL )

```

```

29         error( 1, 0, "unknown service: %s\n", sname );
30         sap->sin_port = sp->s_port;
31     )
32 }

```

tcpserver.skel

图 2.16 set_address 函数

set_address

8~9 我们 zero out 这个 sockaddr_in 结构并设置地址族为 AF_INET。

10~19 如果 hname 不是 NULL，我们首先假定它是一个以标准的点分隔的十进制表示法表示的数字地址，并使用 inet_aton 对它进行转换。如果 inet_aton 返回错误，我们接着通过调用 gethostbyname 来解析 hname 为一个地址。如果该函数也失败了，我们就打出诊断信息并退出程序。

20~21 如果调用者不指定一个主机名或地址，我们就设置地址为 INADDR_ANY。

22~24 接着，我们尝试转换 sname 为一个整数。如果该函数成功返回，我们就转换端口号为网络字节顺序（请参阅技巧 28）。

27~30 如果 sname 转换成整数时失败了，我们就假定它是一个服务名称并通过调用 getservbyname 来获得端口号。如果服务是未知的，我们就输出诊断信息并退出程序。注意 getservbyname 返回的端口号已经是网络字节顺序了。

```

#include "etcp.h"

void set_address( char *host, char *port,
                  struct sockaddr_in *sap, char *protocol );

```

因为我们还有机会直接调用 set_address，所以我们在里记录它的函数原型以利于以后参考。

```

#include "etcp.h"

void error( int status, int err, char *format, ... );

```

最后的函数 error 显示在图 2.17 中。这是我们标准的诊断函数。

如果 status 是非零的，error 在打印诊断信息后退出；否则，它直接返回。如果 err 是非零的，它就会被当作 errno 的值，并取出它的值以及相应的字符串附加到诊断信息上。

我们在例子中连续地使用 error 函数，所以我们把它添加到类库中，并把它的原型写入文档之中：

tcpserver.skel

```

1 void error (int status, int err, char *fmt, ...)
2 {
3     va_list ap;
4
5     va_start( ap, fmt );
6     fprintf( stderr, "%s: ", program_name );
7     vfprintf( stderr, fmt, ap );
8     va_end( ap );
9     if ( err )
10        fprintf( stderr, " : %s (%d)\n", strerror( err ), err );
11     if ( status )
12        EXIT( status );
13 }
```

tcpserver.skel

图 2.17 error 函数

我们的框架程序也包括一个 server 函数的占位程序 (stub) :

```
static void server( SOCKET s, struct sockaddr_in *peerp )
{
}
```

我们可以仅仅往这个 server 占位程序中加入一些代码而使之成为一个简单的应用程序。例如，如果我们拷贝 tcpserver.skel 到 hello.c 并用下列代码取代占位程序：

```
static void server( SOCKET s, struct sockaddr_in *peerp )
{
    send( s, "Hello, world\n", 13, 0 );
}
```

我们就获得了这个著名的 C 程序的网络版本。如果我们编译并运行它，然后用 telnet 来连接它，我们就会获得期望的结果：

```
bsd: $ hello 9000 &
[1] 1163
bsd: $ telnet localhost 9000
```

```

Trying 127.0.0.1...
Connected to localhost
Escape character is '^]'.
Hello, world
Connection closed by foreign host.

```

因为 `tcpserver.skel` 表示了 TCP 服务器的一般情况，我们抽取 `main` 中的大多数代码，并把它转换成一个库函数，称作 `tcp_server`，如图 2.18 所示。`tcp_server` 的原型如下：

```

#include "etcp.h"

SOCKET tcp_server( char *host, char *port );
>Returns:a listening socket(terminates on error)

```

参数 `host` 指向一个字符串，该字符串是主机名称或者是主机的 IP 地址。同样地，参数 `port` 指向一个字符串，该字符串是符号化的服务名称或 ASCII 码的端口号。

从现在开始，除非需要为一个特殊的情况来修改框架程序代码，否则我们将使用 `tcp_server`。

library/tcp_server.c

```

1 SOCKET tcp_server( char *hname, char *sname )
2 {
3     struct sockaddr_in local;
4     SOCKET s;
5     const int on = 1;
6
7     set_address( hname, sname, &local, "tcp" );
8     s = socket( AF_INET, SOCK_STREAM, 0 );
9     if ( !isValidsock( s ) )
10        error( 1, errno, "socket call failed" );
11
12     if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR,
13                      ( char * )&on, sizeof( on ) ) )
14        error( 1, errno, "setsockopt failed" );
15
16     if ( bind( s, ( struct sockaddr * ) &local,
17                sizeof( local ) ) )
18        error( 1, errno, "bind failed" );

```

```

16 if ( listen( s, NLISTEN ) )
17     error( 1, errno, "listen failed" );
18
19 }

```

-----library/tcp_server.c

图 2.18 tcp_server 函数

» TCP 客户端框架程序

下面我们看看 TCP 客户端应用程序的框架（图 2.19）。除了 main 函数和用 client 占位程序取代 server 占位程序之外，它和我们的 TCP 服务器程序是一模一样的。

```

tcpclient.skel
1 int main( int argc, char **argv )
2 {
3     struct sockaddr_in peer;
4     SOCKET s;
5
6     INIT();
7
8     set_address( argv[ 1 ], argv[ 2 ], &peer, "tcp" );
9
10    s = socket( AF_INET, SOCK_STREAM, 0 );
11    if ( !isvalidsock( s ) )
12        error( 1, errno, "socket call failed" );
13
14    if ( connect( s, ( struct sockaddr * )&peer,
15                  sizeof( peer ) ) )
16        error( 1, errno, "connect failed" );
17
18    client( s, &peer );
19    EXIT( 0 );
20 }

```

-----tcpclient.skel

图 2.19 tcpclient.skel 的 main 函数

tcpclient.skel

6~9 和 `tcpserver.skel` 一样，我们用请求地址和端口号来填充 `sockaddr_in` 结构，并获得一个套接字。

10~11 我们调用 `connect` 来建立和对等方的连接。

13 当 `connect` 函数成功返回时，我们用新建立的连接套接字和对等方的地址来调用 `client` 占位程序。

然后我们拷贝它到 `hello.c` 加以测试，并用下面的代码来填充 `client` 占位程序：

```
static void client( SOCKET s, struct sockaddr_in *peerp )
{
    int rc;
    char buf[ 120 ];
    for ( ; ; )
    {
        rc = recv( s, buf, sizeof( buf ), 0 );
        if ( rc <= 0 )
            break;
        write( 1, buf, rc );
    }
}
```

这个客户端程序仅仅从套接字中读取字节并把它们写到标准输出，直到服务器发送一个文件结束字节（EOF）为止。如果我们连接到 `hello` 服务器，就会再次获得下面的预期结果：

```
bsd: $ helloc localhost 9000
hello, world
bsd: $
```

正如在 `tcpserver` 框架程序中做的一样，我们抽取 `tcpclient.skel` 中的相应部分到类库中。新函数 `tcp_client` 显示在图 2.20 中，其原型如下：

```
#include "etcpc.h"

SOCKET tcp_client( char *host, char *port );

```

Returns:a connected socket(terminates on error)

和 `tcp_server` 一样，`host` 要么是主机名称要么是它的 IP 地址。参数 `port` 要么是字符串的服务名称要么是它的 ASCII 端口号。

library/tcp_client.c

```

1 SOCKET tcp_client( char *hname, char *sname )
2 {
3     struct sockaddr_in peer;
4     SOCKET s;
5
6     set_address( hname, sname, &peer, "tcp" );
7     s = socket( AF_INET, SOCK_STREAM, 0 );
8     if ( !isValidsock( s ) )
9         error( 1, errno, "socket call failed" );
10
11    if ( connect( s, ( struct sockaddr * )&peer,
12                  sizeof( peer ) ) )
13        error( 1, errno, "connect failed" );
14
15    return s;
16 }

```

*library/tcp_client.c*图 2.20 `tcp_client` 函数

» UDP 服务器框架程序

除了不需要设置 `SO_REUSEADDR` 套接字选项之外，我们的 UDP 服务器框架程序跟 TCP 版本很相似，当然没有必要调用 `accept` 和 `listen` 函数，这是因为 UDP 是一个无连接协议（请参阅技巧 1）。框架程序的 `main` 函数如图 2.21 所示。框架程序的其余部分和 `tcpserver.skel` 一模一样。

udpserver.skel

```

1 int main( int argc, char **argv )
2 {
3     struct sockaddr_in local;
4     char *hname;
5     char *sname;
6     SOCKET s;
7
8     INIT();

```

```

8   if ( argc == 2 )
9   {
10     hname = NULL;
11     sname = argv[ 1 ];
12   }
13 else
14 {
15   hname = argv[ 1 ];
16   sname = argv[ 2 ];
17 }

18 set_address( hname, sname, &local, "udp" );
19 s = socket( AF_INET, SOCK_DGRAM, 0 );
20 if ( !isValidsock( s ) )
21   error( 1, errno, "socket call failed" );

22 if ( bind( s, ( struct sockaddr * ) &local,
23           sizeof( local ) ) )
24   error( 1, errno, "bind failed" );

25 server( s, &local );
26 EXIT( 0 );
27 }

```

*udpserver.skel*图 2.21 *udpserver.skel* 的 main 函数

udpserver.skel

18 我们调用 `set_address` 函数来用地址和服务器将接收数据报的端口号填充 `sockaddr_in` 结构 `local`。注意我们指定的是“`udp`”而不是“`tcp`”。

19~24 我们获得 `SOCK_DGRAM` 套接字并绑定 `local` 中的地址和端口到该套接字上。

25 我们通过调用服务器占位程序来等待数据报的到来。

为了构造“hello world”程序的 UDP 版本，我们拷贝框架程序到 `udphello.c` 中，并用下列代码来代替服务器占位程序：

```

static void server( SOCKET s, struct sockaddr_in *localp )
{
    struct sockaddr_in peer;
    int peerlen;
    char buf[ 1 ];

    for ( ; ; )
    {
        peerlen = sizeof( peer );
        if ( recvfrom( s, buf, sizeof( buf ), 0,
                      (struct sockaddr * )&peer, &peerlen ) < 0 )
            error( 1, errno, "recvfrom failed" );
        if ( sendto( s, "hello, world\n", 13, 0,
                    (struct sockaddr * )&peer, peerlen ) < 0 )
            error( 1, errno, "sendto failed" );
    }
}

```

我们在使用这个服务器之前，需要开发一个 UDP 客户端框架程序（图 2.23），所以现在我们仅抽取 main 的最后部分到库函数 udp_server 中：

```

#include "etcp.h"

SOCKET udp_server( char *host, char *port );

```

Returns:a UDP socket bound to *host* on port *port*

参数 host 和 port 照例指向包含主机名称或 IP 地址的字符串，以及服务名称或端口号。

library/udp_server

```

1 SOCKET udp_server( char *hname, char *sname )
2 {
3     SOCKET s;
4     struct sockaddr_in local;
5
5     set_address( hname, sname, &local, "udp" );
6     s = socket( AF_INET, SOCK_DGRAM, 0 );
7     if ( !isValidsock( s ) )
8         error( 1, errno, "socket call failed" );
9     if ( bind( s, ( struct sockaddr * ) &local,

```

```

10     sizeof( local ) ) )
11     error( 1, errno, "bind failed" );
12 return s;
13 }

```

----- library/udp_server

图 2.22 udp_server 函数

➤ UDP 客户端框架程序

UDP 客户端框架程序的 main 函数仅仅用被请求对等方的地址和端口号来填充 peer，并获得一个 SOCK_DGRAM 套接字。函数 main 如图 2.23 所示。框架程序的其余部分和 udpserver.skel 一模一样。

----- udpclient.skel

```

1 int main( int argc, char **argv )
2 {
3     struct sockaddr_in peer;
4     SOCKET s;
5
6     INIT();
7
8     set_address( argv[ 1 ], argv[ 2 ], &peer, "udp" );
9     s = socket( AF_INET, SOCK_DGRAM, 0 );
10    if ( !isValidsock( s ) )
11        error( 1, errno, "socket call failed" );
12
13    client( s, &peer );
14    exit( 0 );
15 }

```

----- udpclient.skel

图 2.23 udpclient.skel 的 main 函数

我们去测试这个框架程序的同时，通过拷贝 udpclient.skel 到 udphello.c 来测试 udphello，client 占位程序如下所示：

```

static void client( SOCKET s, struct sockaddr_in *perrp )
{

```

```

int rc;
int peerlen;
char buf[ 120 ];
peerlen = sizeof( *peerp );
if ( sendto( s, "", 1, 0, ( struct sockaddr * )peerp,
    peerlen ) < 0 )
    error( 1, errno, "sendto failed" );
rc = recvfrom( s, buf, sizeof( buf ), 0,
    ( struct sockaddr * )peerp, &peerlen );
if ( rc >= 0 )
    write( 1, buf, rc );
else
    error( 1, errno, "recvfrom failed" );
}

```

函数 `client` 仅发送一个空字节给服务器，读取结果数据报，并把它写到标准输出上，然后退出。发送一个空字节满足 `udphello` 程序中的 `recvfrom` 函数，该函数返回以便 `udphello` 可以发送它的数据报。

当我们运行这两个程序时，就会获得通常的问候：

```

bsd: $ udphello 9000 &
[1] 448
bsd: $ udphelloc localhost 9000
hello, world
bsd: $

```

和往常一样，我们从 `main` 中抽取设置代码到类库中。注意我们的类库函数，称作 `udp_client`（如图 2.24），有第三个参数，它是 `sockaddr_in` 结构的地址，它将会用前两个参数中指定的地址和端口号来填充：

```

#include "etcp.h"

SOCKET udp_client( char *host, char *port );
    struct sockaddr_in *sap );

```

Returns:a UDP socket and filled in sockaddr_in structure

library/udp_client.c

```

1 SOCKET udp_client( char *hname, char *sname,
2   struct sockaddr_in *sap )
3 {
4   SOCKET s;
5
6   set_address( hname, sname, sap, "udp" );
7   s = socket( AF_INET, SOCK_DGRAM, 0 );
8   if ( !isvalidsock( s ) )
9     error( 1, errno, "socket call failed" );
10  return s;
11 }
```

library/udp_client.c

图 2.24 udp_client 函数

小结

尽管本节比较长，但是它说明了建造框架程序以及库代码是如何的简单。所有的框架程序都很相似，主要是 main 中的几行设置代码有点差异。这样，一旦我们第一次把它们编写好了，其余的工作无非就是拷贝并做一些小的改动。只要通过简单填充占位函数就可以轻松而迅速地创建一些基本的客户端和服务器，这更说明了这种方法具有很好的能力。

使用框架程序来创建函数的类库给了我们一个基本类库，在这个基础上我们可以轻松地创建应用程序并编写小的测试程序来测试它们。本书的其余部分到处都可以看到这些例子。

技巧 5 选择套接字接口而不是 XTI/TLI

在 UNIX 的世界里，有两个主要的 API，它们用于通信协议如 TCP/IP 的接口。这两个 API 是：

- Berkeley 套接字
- X/Open 传输接口（X/Open Transport Interface，XTI）

套接字接口是加利福尼亚大学伯克利分校为它们的 UNIX 操作系统的新版本开发的。它首先是在 4.2BSD (1983) 中广泛使用的，在 4.3BSD Reno (1990) 中进行了一些改进，现在这个接口实际上在每一个 UNIX 版本中都提供。一些其他的操作系统也提供了一个套接字 API。在 Microsoft Windows 的世界里很流行的 Winsock API，就是从 BSD 套接字 API 继承过来的 [Winsock Group 1997]。

XTI API 是传输层接口 (transport layer interface, TLI) 的超集，它首先是在 AT&T 的 UNIX 系统 V 版本 3.0 (SVR3) 中提供的。TLI 设计为与协议无关的，主要是为了让它能够轻松地支持新的协议。它的设计受到 OSI 协议（请参阅技巧 14）的影响很大。在那个时候，人们广泛地认为这些协议将很快取代 TCP/IP，因此接口的设计从 TCP/IP 程序员的观点来看就不是最优的。更糟的是，尽管 TLI 函数名称和套接字 API 中函数名称很相似（除了它们是以 `t_` 开头之外），但是相应函数的语意有时却不同。

XTI 接口受到比较长久的欢迎，这很大程度上归功于它和 Novell 互联网包交换/顺序包交换协议 (Internetwork Packet Exchange/Sequenced Packet Exchange, IPX/SPX) 的联合使用。当为 IPX/SPX 编写的程序移植到 TCP/IP 上时，最好的方法就是使用相同的 TLI 接口 [Kacker 1999]。

《UNIX 网络编程》[Stevens 1998]第一卷第四部分提供了十分不错的 XTI 和 STREAMS 编程的介绍。当我们看到 Stevens 花了 100 多页的篇幅来讨论 XTI API 时，就会对 XTI 和套接字接口很大的语意差别有一个很好的认识。

因为人们相信 OSI 协议将会取代 TCP/IP，许多 UNIX 供应商建议新的程序使用 TLI API 来编写。甚至多个供应商建议不要在以后的版本中支持套接字接口。当然，人们关于 TCP/IP 和套接字接口将进入穷途末路的认识最后被证明是不成熟的。

现在 OSI 协议基本上已经不再使用了，但是 TLI 以及它的继承者 XTI 仍然在 System V 继承下来的 UNIX 系统中使用。对于 UNIX 应用程序来说，这就出现了使用套接字接口好还是 XTI 好的问题。

在我们分析这些接口的每一个好处时，认识到它们只是“接口”是很重要的。对于 TCP/IP 应用程序员来说，它们仅仅是跟 TCP/IP 栈通信的一种方法。因为实现通信协议的是 TCP/IP 栈，所以对于应用程序来说对等方使用的是哪种 API 根本没有任何区别。也就是说，使用套接字的应用程序可以无缝地和使用 XTI 的应用程序实现互操作。SVR4 系统对两个接口都支持，而且通常是作为类库来实现的，通过 STREAMS 子系统来和同一个 TCP/IP 栈实现接口。

让我们首先看看 XTI。XTI 确实在网络编程中有一定的位置。因为它的协议无关性的设计，使用 TLI 或 XTI 增加一个新的协议到 UNIX 系统，所以可以不需要访问操作系统内核的源代码。协议的设计者仅需要实现传输提供者如 STREAMS 多路器，把它链接到操作系统内核，然后使用 XTI 来访问它。

如何编写诸如 STREAMS 的模块，以及通常的 TLI 和 STREAMS 编程，在[Rago 1993]中有详细的讨论。

注意这种情况是十分有限的：我们必须实现一个操作系统没有提供的协议，而且我们不能对操作系统内核进行访问。

当然，我们在 SVR4 或其他支持 STREAMS 和 TLI/XTI 的系统下也必须开发这个新协议。从 Solaris 2.6 开始，Sun 公司提供和套接字 API 相同的功能。

除了刚才描述的情况中的优势之外，人们有时也讨论使用 XTI/TLI 编写协议无关的代码更为容易[Rago 1996]。当然这种“更容易”是主观的，在《UNIX 网络编程》[Stevens 1998] 的 11.9 节中，Stevens 使用套接字提供了一个简单的协议无关的时间服务器，它支持 IPv4、IPv6 以及 UNIX 域套接字。

最后，有时在两个接口都支持时人们也认为套接字接口位于 TLI/XTI 之上，这样 TLI/XTI 效率更高，这种认识是不正确的。正如前面所讲的那样，基于 SVR4 的系统通常以一套直接和下面的 STREAMS 子系统通信的类库来实现的。实际上，从 Solaris 2.6 开始（Solaris 是 SVR4 版本的 Sun 版本），套接字调用是在操作系统内核实现的，可以通过系统调用直接访问。

套接字接口最大的优势就是可移植性。因为 TLI/XTI 所能够提供的东西套接字接口实际上总是提供的，所以使用套接字可以获得最大的可移植性。不管我们的应用程序是否只运行在 UNIX 系统上这都是正确的，这是因为大多数支持 TCP/IP 的操作系统都提供了套接字接口，而只有少数的非 UNIX 系统提供 TLI/XTI。例如，编写一个可以在 UNIX 和 Microsoft Windows 之间可移植的应用程序是相当直接的，这是因为 Windows 支持 Winsock，Winsock 是套接字接口的具体实现。

套接字的另一个优势就是该接口比 TLI/XTI 更容易使用。因为 TLI/XTI 受到 OSI 协议的限制设计为更通用的接口，所以和套接字比较，它要求应用程序编程人员要做更多的工作。甚至是 TLI/XTI 支持者都承认对于 TCP/IP 应用程序来说，套接字是一个更好的接口。

下面的段落摘自 Solaris 2.6 子过程类库介绍手册，它提出了选择 API 的极好的建议：

在所有的情况下，我们更多地建议使用套接字 API 而不是 XTI 和 TLI API。如果需要移植到其他的 XPGV4v2 系统，应用程序就必须使用 libxnet 接口。如果不进行这种移植，就建议使用 libsocket 和 libns1 中的套接字接口而不是 libxnet 中的接口。在 XTI 和 TLI API 之间，建议使用 XTI 接口（在 libxnet 中可获得）而不是使用 TLI 接口（在 libns1 中可获得）。

小结

除了一些特殊的情况之外，很少有什么原因迫使我们在 TCP 编程中必须使用 TLI/XTI。套接字提供了一个更容易的、可移植的接口，而且这些接口之间的性能确实不存在任何差异。

技巧 6 记住 TCP 是一个流协议

TCP 是一个流协议（stream protocol），这意味着数据是作为字节流递交给接收者的，没有内在的“消息”或“消息边界”的概念。从这方面来考虑，读 TCP 数据就像从一个串行端口读数据一样——读数据时永远也不知道一个给定的读调用将会返回多少字节。

为了说明这一点，让我们假设主机 A 和主机 B 的应用程序之间有一个 TCP 连接，并

且主机 A 上的应用程序发送消息给主机 B。进一步假设主机 A 要发送两个消息，通过两次调用 send 来发送，每个消息调用一次。我们很自然地认为两个消息从主机 A 传送到主机 B 是作为两个独立实体的，每个消息放在它自己的数据包里，如图 2.25 所示。

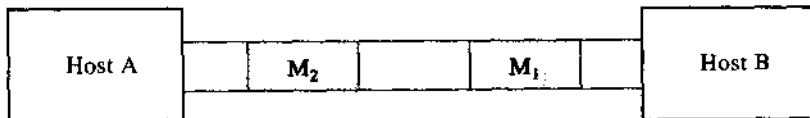


图 2.25 发送两个消息的不正确的模式

不幸的是，数据传输实际上有可能并不遵循这种模式。主机 A 上的应用程序调用 send。我们想像的是应用程序写入的数据在一个数据包里传送到主机 B。实际上，send 通常只是拷贝数据到主机 A 的 TCP/IP 栈里然后返回。如果要发送数据的话，TCP 就会决定将要立即发送多少出去。该决定是复杂的，并且依赖于多种因素，如发送窗口（在这个时间点上主机 B 希望接收的数据数量）、阻塞窗口（网络阻塞的估计）、路径最大传输单元（主机 A 和主机 B 之间的网络路径上能够传输数据的最大数量），以及连接的输出队列上有多少数据正在排队。请参阅技巧 15，了解更多的关于这方面的知识。图 2.26 显示的是数据在主机 A 上被 TCP 打包的 4 种可能的方法。在图 2.26 中，M11 和 M12 表示的是 M1 的第一和第二部分，M21 和 M22 的意义类似。如图 2.26 描述的那样，TCP 没有必要在一个数据包里发送一个消息的全部数据。

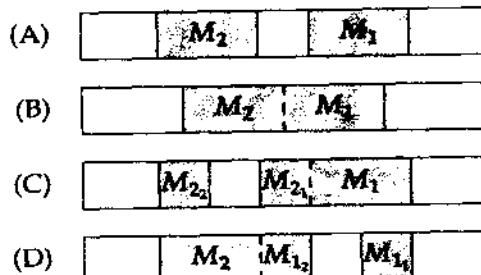


图 2.26 两条消息的四种可能的打包方法

现在让我们以主机 B 上的应用程序的观点来看看情况到底是如何的。通常，主机 B 上的应用程序在任何给定的 recv 调用上都不会假定 TCP 可获得多少数据。例如，当主机 B 上的应用程序通过发布读操作来获得第一个消息时，有可能出现下列四种情况之一：

实际上，存在不止四种的情况，但是我们忽略了比如错误和 EOF 等情况的可能性。同时，我们假定应用程序读取可获得的所有数据。

(1) 数据没有为读准备好，应用程序阻塞或者 recv 返回一个指示说明数据不可获得。严格地说，到底发生什么事情依赖于套接字是否已经标记为阻塞以及主机 B 上的操作系统

时如何定义 `recv` 系统调用的语意的。

(2) 应用程序获得的是消息 M1 中的部分数据但不是全部。例如，如果在发送方 TCP 以图 2.26 (D) 中显示的方式分组打包数据时，就会发生这种情况。

(3) 应用程序获得且只获得消息 M1 中的所有数据。如果数据是以图 2.26 (A) 中显示方式分组打包的话，就会发生这种情况。

(4) 应用程序获得消息 M1 中的所有数据以及消息 M2 中的部分数据。当数据是以图 2.26 (B) 或图 2.26 (C) 中显示的方式分组打包的时，就会发生这种情况。

应当注意的是，这里有一个时间问题。如果主机 B 上的应用程序直到主机 A 已经发送第二条消息之后一段时间才发布读取第一条消息的操作，那么该操作就可以获得两条消息。这种情况和图 2.26 (B) 中显示的情况相似。正如上面这些情况显示的那样，在特定时候可以读取数据的数量通常是不确定的。

再次提醒你，TCP 是一个流协议，虽然数据是在 IP 数据包中传输的，但是一个数据包中的数据跟调用 `send` 函数传递多少数据给 TCP 没有直接的关系。而且，接收数据的应用程序也没有可靠的方法来决定数据是如何分组打包的，这是因为几个包可能在 `recv` 调用之间到达。

甚至接收数据的应用程序的相应能力很强，这种情况也可能发生。例如，如果一个数据包丢失了（今天的 Internet 经常发生这种情况，请参阅技巧 12），而后面的包安全到达，TCP 就会保存后面数据包中的数据直到第一个数据包重传过来并正确地接收到。只有在这个时候，我们才能说应用程序可以获得所有的数据。

TCP 记录着它发送了多少字节以及多少字节已经被确认了，而不是它们是如何分组打包的。实际上，一些具体实现可以在一个丢失数据包的重传中发送或多于或少于原来数据包中的数据。这个事实足够重要，让我们重申一遍：

对于 TCP 应用程序来说没有“数据包”之类的概念。设计为以任何方式依赖 TCP 如何将数据分组打包的应用程序应当重新设计。

因为任何给定的读操作中返回数据的数量是不可预测的，我们必须在应用程序中应付这种情况。通常这不是一个问题。例如，有可能我们正在使用一个标准的 IP 库函数如 `fgets` 来读取数据。在这种情况下，`fgets` 负责为我们把字节流分成行。请参阅图 3.6 中的这种情况的一个例子。其他时候，我们确实不得不担心消息的边界。在这些情况下，我们必须在应用程序级别保留消息边界。

最简单的情况是定长度的消息。对于这些消息，我们仅需要读取消息中的指定数目的字节。以前面的讨论为例，执行下面的一个读操作是不够的：

```
recv(s, msg, sizeof(msg), 0);
```

因为该读操作可能返回少于 `sizeof(msg)` 个字节（图 2.26D）。处理这个情况的标准方

法如图 2.27 所示。

library/readn.c

```

1 int readn( SOCKET fd, char *bp, size_t len)
2 {
3     int cnt;
4     int rc;
5
6     cnt = len;
7     while ( cnt > 0 )
8     {
9         rc = recv( fd, bp, cnt, 0 );
10
11         if ( rc < 0 )           /* read error? */
12         {
13             if ( errno == EINTR ) /* interrupted? */
14                 continue;      /* restart the read */
15             return -1;          /* return error */
16         }
17         if ( rc == 0 )           /* EOF? */
18             return len - cnt;   /* return short count */
19         bp += rc;
20         cnt -= rc;
21     }
22     return len;
23 }
```

library/readn.c

图 2.27 readn 函数

readn 函数的用法和 read 函数的用法相似，但是它直到读取了 len 个字节后才返回，最后对等方或者接收到一个 EOF 字节，或者发生了错误。我们记录它的定义为：

```
#include "etcp.h"

int readn( SOCKET s, char *buf, size_t len );

```

Returns:number of bytes read, or -1 on error

请不要奇怪，`readn` 和从串行口以及其他基于流的数据源读取指定数目的字节具有相同的代码，这是因为在任何指定的时间内可获得的数据量是不可预知的。实际上，`readn`（用 `int` 代替 `SOCKET`，`read` 代替 `recv`）可以用于这些所有的情况。

if 语句

```
if ( errno == EINTR ) /*interrupted?*/
    continue;           /*restart the read*/
```

如果它被一个信号中断的话，那就在 11 行和 12 行重新启动 `recv` 调用。因为一些系统会自动地重新启动中断系统调用，所以对于这些系统来说不需要这两行。但是，这两行代码不会造成任何危害，所以最好尽量保留它们。

对于那些必须支持可变长度消息的应用程序，可以使用两种解决方法。首先，记录可以用记录结束标记来分隔。我们前面所描述的通过使用标准的 I/O 函数如 `fgets` 来把消息分成单独的行，就属于这种情况。在标准 I/O 的情况下，换行符号是以一种自然的方法作为记录结束标记的。然而，这种方法通常是有问题的。首先，除非发送方在它的消息体中不使用记录结束标记，否则发送函数不得在消息中扫描这些标记，或者使用转义字符，或者把它们转换成不会被误解为记录结束标记的编码。例如，如果选择记录分隔符 RS 作为记录结束标记，发送方就必须在消息体中搜索任何出现的 RS 字符并使用转义字符，也就是说在它们前面使用 “\”。这就意味着数据不得不为转义字符增加位置。当然，任何出现的换码字符也必须被转义。所以，如果我们使用 “\” 作为转义字符，消息体中出现的 “\” 不得不变为 “\\”。

在接收方，必须再次扫描整个消息，这次是删除转义字符并搜索（没有被转义字符标记的）记录结束标记。因为记录结束标记的使用可能需要将整个消息扫描两遍，所以最好限制记录结束标记的使用，仅让它在有“自然的”记录结束标记的情况下使用，如换行字符用于分隔文本行记录。

另一个解决可变长度记录的方法是在每一个消息前面附加一个消息头，包含（至少是）后面消息的长度，如图 2.28 显示的那样。



图 2.28 可变长度记录的格式

接收数据的应用程序把消息读成两部分。首先，它读取固定长度的消息头，从消息头中抽取可变部分的长度；然后读取可变长度部分。图 2.29 说明了消息头只包含记录长度的简单情况。

library/readvrec.c

```

1 int readvrec( SOCKET fd, char *bp, size_t len )
2 {
3     u_int32_t reclen;
4     int rc;
5
6     /* Retrieve the length of the record */
7
8     rc = readn( fd, ( char * )&reclen, sizeof( u_int32_t ) );
9     if ( rc != sizeof( u_int32_t ) )
10        return rc < 0 ? -1 : 0;
11     reclen = ntohl( reclen );
12     if ( reclen > len )
13     {
14         /*
15          * Not enough room for the record--
16          * discard it and return an error.
17         */
18
19         while ( reclen > 0 )
20         {
21             rc = readn( fd, bp, len );
22             if ( rc != len )
23                 return rc < 0 ? -1 : 0;
24             reclen -= len;
25             if ( reclen < len )
26                 len = reclen;
27         }
28     }
29
30     set_errno( EMSGSIZE );
31     return -1;
32 }
33
34 /* Retrieve the record itself */

```

```

29 rc = readn( fd, bp, reclen );
30 if ( rc != reclen )
31     return rc < 0 ? -1 : 0;
32 return rc;
33 }

```

library/readvrec.c

图 2.29 读取可变长度记录的函数

读取记录长度

6~8 记录的长度读入到 `reclen` 中，如果 `readn` 不返回一个整型的数据就返回 0 (EOF)，如果发生了错误就返回-1。

9 记录的大小从网络字节顺序转换成主机字节顺序。参阅技巧 28，了解更多有关这方面的知识。

检查记录是否合适

10~27 然后，通过检查调用者的缓冲区大小来检验它是否足够保存整条记录。如果缓冲区中的空间不够，该记录就会被丢弃，随后读取 `len`-大小的段到缓冲区中。在丢弃该记录之后，`errno` 设置为 `EMSGSIZE`，而 `readrec` 返回-1。

读取记录

29~32 最后，读取记录本身。根据 `readn` 是否返回错误、短整型的记数或成功返回，`readrec` 返回-1、0 或 `reclen` 给调用者。

因为 `readrec` 是一个很有用的函数，我们将在其他技巧中使用它，我们记录它的定义为：

```

#include "etcp.h"

int readvrec( SOCKET s, char *buf, size_t len );

```

Returns: number of bytes read, or -1 on error

图 2.30 是一个简单服务器的示意图，它使用 `readvrec` 函数从一个 TCP 连接中读取可变长度的记录，并把它们写入到标准输出。

vrs.c

```

1 #include "etcp.h"

2 int main( int argc, char **argv )
3 {
4     struct sockaddr_in peer;

```

```

5   SOCKET s;
6   SOCKET s1;
7   int peerlen = sizeof( peer );
8   int n;
9   char buf[ 10 ];

10 INIT();
11 if ( argc == 2 )
12     s = tcp_server( NULL, argv[ 1 ] );
13 else
14     s = tcp_server( argv[ 1 ], argv[ 2 ] );
15 s1 = accept( s, ( struct sockaddr * )&peer, &peerlen );
16 if ( !isValidsock( s1 ) )
17     error( 1, errno, "accept failed" );
18 for ( ; ; )
19 {
20     n = readvrec( s1, buf, sizeof( buf ) );
21     if ( n < 0 )
22         error( 0, errno, "readvrec returned error" );
23     else if ( n == 0 )
24         error( 1, 0, "client disconnected\n" );
25     else
26         write( 1, buf, n );
27 }
28 EXIT( 0 );      /* not reached */
29 }

```

vrs.c

图 2.30 vrs——使用 readrec 的服务器的示例

10~17 服务器初始化并接收一个连接。

20~24 调用 readrec 来读取下一个可变长度的记录。如果发生了错误，就输出诊断消息并读取下一个记录。如果 readrec 返回 EOF，就输出一个消息，然后服务器退出。

26 把记录写入到标准输出。

图 2.31 显示了从标准输入中读取消息的对应客户端，附加消息的长度，并把它返回给服务器。

定义 packet 结构

当我们调用 send 时，我们通过定义 packet 结构来保存消息以及消息的长度。u_int_32_t 是一个无符号 32 位整型数据类型。因为 Windows 没有定义这种数据类型，所以 Windows 版本的 skel.h 就有一个类型定义。

我们必须注意这个例子的另一个潜在的问题。我们假定编译器严格地按照我们指定的那样压缩结构中的数据而不做任何填充。因为第二个记录元素是一组字节，所以该假设在大多数的系统上是可行的。但是我们应当警觉有关编译器是如何压缩结构中的数据的假设的问题。我们在技巧 24 里还会讨论这个问题，在技巧 24 里我们讨论立即发送两个或更多的不同段的方法。

vrc.c

```

1 #include "etcp.h"

2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     int n;
6     struct
7     {
8         u_int32_t reclen;
9         char buf[ 128 ];
10    } packet;

11 INIT();
12 s = tcp_client( argv[ 1 ], argv[ 2 ] );
13 while ( fgets( packet.buf, sizeof( packet.buf ), stdin )
14       != NULL )
15 {
16     n = strlen( packet.buf );
17     packet.reclen = htonl( n );
18     if ( send( s, ( char * )&packet,
19               n + sizeof( packet.reclen ), 0 ) < 0 )
20         error( 1, errno, "send failure" );
21 }
22 EXIT( 0 );
23 }
```

vrc.c

图 2.31 vrc——发送可变长度消息的客户端

连接、读数据和发送数据行

12 客户端通过调用 `tcp_client` 连接到服务器。
 13~21 调用 `fgets` 从标准输入中读取一行，并把它放入到消息数据包里的 `buf` 域中。数据行的长度通过调用 `strlen` 来获得，该值在转换成网络字节顺序后，放在消息数据包中的 `reclen` 域中。最后，调用 `send` 来发送数据包给服务器。

请参阅技巧 24，了解将消息分成两个或多个部分发送出去的方法。我们在 `sparc` 工作站上启动服务器，然后在 `bsd` 系统上启动客户端来测试这些程序。运行结果并排显示，好让我们比较客户端的输入以及服务器端的响应输出。程序在第四行也显示错误消息：

Bsd: \$ vrc sparc 8050	sparc: \$ vrs 8050
123	123
123456789	123456789
1234567890	vrs: readvrec returned error:
	Message too long (97)
12	
^C	vrs: client disconnected

因为服务器的缓冲区是 10 字节大小，当我们发送 11 个字节 `1,...,0,<LF>` 时，`readvrec` 返回错误。

小结

对 TCP 发送一个字节流是对没有记录边界的概念的错误理解，也是初级网络编程人员开始经常犯的错误。我们可以通过说明 TCP 没有用户可见的“数据包”的概念来总结这个重要的事实。TCP 仅传送字节流，我们不能可靠地预测一个特定的读操作到底返回多少字节。在本节中，我们探讨了几个策略，我们的应用程序可以使用这些策略来处理这个问题。

技巧 7 不要低估 TCP 的性能

因为 TCP 是一个复杂的协议，它在基本的 IP 数据报服务上增加了可靠性和流量控制，而 UDP 仅增加了校验和，UDP 看起来似乎比 TCP 更快。基于这个假想，许多应用程序编程人员认为他们应当使用 UDP 来获得可接受的性能。在有些情况下，UDP 确实比 TCP 快很多，但是并非所有的情况都如此。我们将在后面看到，TCP 的性能有时实质上比 UDP 还要快。

对于通常的网络编程，任何协议的性能都依赖于网络、应用程序、负载以及其他因素，

这些因素当然包括具体实现的性能。知道哪一个协议和算法将最优地执行的最好方法就是在相同的条件下测试应用程序。当然，这并不总是可行的，但是我们可以使用我们的框架程序创建简单的程序模仿所期望的网络流量来获得对程序性能的良好的认识。

在我们讨论创建一些测试例子之前，让我们考虑一下为什么在某些情况下以及在哪些情况下 UDP 的性能大大高于 TCP。首先，因为 TCP 是一个更复杂的协议，所以它比 UDP 要做更多的处理。

[Stevens 1996] 披露 4.4BSD 的 TCP 具体实现包含将近 4,500 行 C 代码，而 UDP 的代码才不到 800

行。当然，这两个协议的正常代码长度相当短，但是这些数字确实给了我们一个代码复杂度的相对指标。

通常情况下，尽管 CPU 处理两个协议的主要操作都是拷贝数据和数据校验（请参阅技巧 26），但是我们不希望在这里看到很大的差异。实际上，在[Partridge 1993]中，Jacobson 描述了一个 TCP 的实验版本，其正常的接收段代码长度只有 30 (RISC) 机器指令长（数据校验和拷贝数据到用户缓冲区的代码除外，这二者是一起进行的）。

我们要提到的下一件事是提供可靠性，接收端 TCP 必须发送 ACK 给发送端 TCP。这就增加了双方必须做的处理工作，但是没有我们想像的那么多。首先，接收方可以在任何必须发送给对等方的数据中捎带 ACK 消息。实际上，许多 TCP 的具体实现都是延迟几毫秒来发送 ACK 消息，以防本方的应用程序对近来的段返回一个应答。其次，TCP 并不需要为每个段都返回一个 ACK 消息。在正常情况下，大多数 TCP 的具体实现每隔一个段返回一个应答。

RFC 1122[Braden 1989]，主机需求 RFC，建议至少延迟 0.5 秒发送 ACK 消息，只要第双数个段被确认就可以了。

TCP 和 UDP 之间的其他主要差异是 TCP 是一个面向连接的协议（请参阅技巧 1），因此必须处理连接的建立和撤消。在通常情况下连接建立需要交换 3 个段，撤消连接通常需要交换 4 个段，但是除了后面的几个之外，其他所有的段都可以附带在传输数据的段中。

我们假设连接撤消的时间大多数都被数据交换吸收了，把精力主要集中的连接建立时发生的事情上。如图 2.32 所示，客户端通过给服务器发送一个 SYN (同步) 段开始连接建立。这个段不仅包含了某些连接参数如将要接受的最大段大小 (maximum segment size, MMS) 以及初始接收窗口的大小，而且还指定客户端为将要传送的数据使用的一个序列号。服务器响应，传送自己的 SYN 段并附带对客户端的 SYN 消息的确认 ACK 给客户端。最后，客户

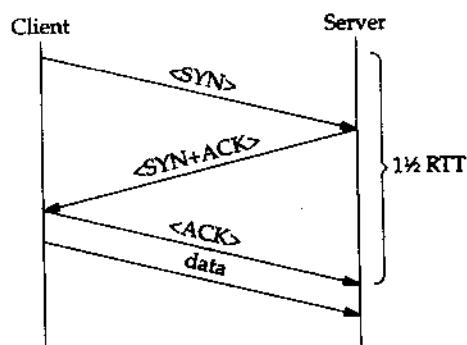


图 2.32 连接建立

端确认服务器的 SYN，连接建立就结束了。这时，客户端（服务器）就可以发送它的第一个段了。

图中 RTT 表示一个往返旅行时间 (round-trip time) ——数据包从一个主机到对等方并返回所花费的时间。正如我们所看到的那样，连接建立花费的时间是一个半往返旅行时间。

如果客户端和服务器之间的连接是长时间的（例如，如果客户端和服务器之间有大量的数据需要传输），一个半 RTT 的时间就被数据传输的时间摊销得微不足道了，不会很大地影响性能。然而，如果我们有一个简单的事物，客户端只发送一个请求，然后服务器响应，那么连接建立的时间就占去了整个事物时间的很大一部分。这样，准确地说如果当应用程序只涉及到短的请求/应答会话的话，UDP 的性能就比 TCP 好很多，而如果连接的时间很长并且涉及到大量的数据传输的话，TCP 的性能就比 UDP 好很多。

为了测试 TCP 和 UDP 的相对性能以及给出创建小测试程序的例子，我们编写了一系列简单的服务器和客户端。我们的目的并不是提供一个全面的基准测试，而是想证明一下在一个大量数据传输的情况下两个协议的性能究竟如何。也就是说，我们的兴趣在于从一个应用程序到另一个应用程序之间传输大量数据时两个协议之间的相对性能。FTP 协议就是我们都很熟悉的一个大量数据传输的例子。

2. UDP 发送方和接收方

对于 UDP，客户端任何数量的数据报，服务器读、记数并丢弃它们。UDP 的客户端如图 2.33 所示。

udpsource.c

```

1 #include "etcpc.h"
2 int main( int argc, char **argv )
3 {
4     struct sockaddr_in peer;
5     SOCKET s;
6     int rc;
7     int datagrams;
8     int dgramsiz = 1440;
9     char buf[ 1440 ];

10    INIT();
11    datagrams = atoi( argv[ 2 ] );
12    if ( argc > 3 )
13        dgramsiz = atoi( argv[ 3 ] );
14    s = udp_client( argv[ 1 ], "9000", &peer );
15    while ( datagrams-- > 0 )

```

```

16  {
17      rc = sendto( s, buf, dgramsz, 0,
18                  ( struct sockaddr * )&peer, sizeof( peer ) );
19      if ( rc <= 0 )
20          error( 0, errno, "sendto failed" );
21  }
22  sendto( s, "", 0, 0,
23          ( struct sockaddr * )&peer, sizeof( peer ) );
24  EXIT( 0 );
25 }
```

udpsource.c

图 2.33 发送任意数量数据报的客户端

10~14 发送数据报的数量以及可选的数据报的大小，是从命令行读取出来的，UDP 套接字使用 *peer* 中请求服务器的地址来建立。和技巧 29 相违背的是，端口号硬编码为 9000。

15~21 数据报的请求数目发送给服务器。

22~23 最后一个 0 字节的数据报发送给服务器。它充当服务器的 EOF 指示符。

图 2.34 中给出的服务器更为简单。

udpsink.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     int rc;
6     int datagrams = 0;
7     int rcbufsiz = 5000 * 1440;
8     char buf[ 1440 ];

9     INIT();
10    s = udp_server( NULL, "9000" );
11    setsockopt( s, SOL_SOCKET, SO_RCVBUF,
12                ( char * )&rcbufsz, sizeof( int ) );
13    for( ; ; )
14    {
15        rc = recv( s, buf, sizeof( buf ), 0 );
16        if ( rc <= 0 )
```

```

17         break;
18     datagrams++;
19 }
20 error( 0, 0, "%d datagrams received\n", datagrams );
21 EXIT( 0 );
22 }
```

udpsink.c

图 2.34 数据报接收方

10 服务器设置为在任何接口的端口 9000 上接收数据报。

11~12 接收缓冲区设置为足够保存 50001440 字节的数据报。

虽然我们设置了缓冲区的大小为 7200000 字节，但是不能保证操作系统将实际分配那么多内存。

我们的主机 bsd 实际上设置缓冲区大小为 41600 字节。这就解释了我们在后面将要看到的为什么数据报会丢失的现象。

13~19 服务器一直读数据报并记数，直到接收到一个空数据报或出现了错误。

20 把接收到的数据报的数目写入到标准错误输出。

» TCP 发送方和接收方

正如在技巧 32 里解释的那样，如果套接字的发送和接收缓冲区的大小比较合理，我们就能从 TCP 里获得更好的性能。这样我们希望设置服务器的套接字接收缓冲区以及客户端的套接字发送缓冲区。

因为我们的 `tcp_server` 和 `tcp_client` 函数使用默认的套接字缓冲区大小，所以我们就从技巧 4 的 TCP 客户端和服务器框架程序开始。因为 TCP 需要知道连接建立期间的缓冲区大小，所以我们必须在服务器端调用 `listen` 之前以及在客户端调用 `connect` 之前就设置缓冲区的大小。这是我们为什么不能使用 `tcp_server` 和 `tcp_client` 的原因：当这两个函数返回时，已经调用了 `listen` 和 `connect` 函数。我们在图 2.35 中列出了客户端的源程序。

tcpsource.c

```

1 int main( int argc, char **argv )
2 {
3     struct sockaddr_in peer;
4     char *buf;
5     SOCKET s;
6     int c;
7     int blks = 5000;
8     int sndbufsz = 32 * 1024;
```

```
9  int sndsz = 1440; /* default ethernet mss */

10 INIT();
11 opterr = 0;
12 while ( ( c = getopt( argc, argv, "s:b:c:" ) ) != EOF )
13 {
14     switch ( c )
15     {
16         case 's' :
17             sndsz = atoi( optarg );
18             break;

19         case 'b' :
20             sndbufsz = atoi( optarg );
21             break;

22         case 'c' :
23             blks = atoi( optarg );
24             break;

25         case '?' :
26             error( 1, 0, "illegal option: %c\n", c );
27     }
28 }

29 if ( argc <= optind )
30     error( 1, 0, "missing host name\n" );

31 if ( ( buf = malloc( sndsz ) ) == NULL )
32     error( 1, 0, "malloc failed\n" );
33 set_address( argv[ optind ], "9000", &peer, "tcp" );
34 s = socket( AF_INET, SOCK_STREAM, 0 );
35 if ( !isvalidsock( s ) )
36     error( 1, errno, "socket call failed" );

37 if ( setsockopt( s, SOL_SOCKET, SO_SNDBUF,
38     ( char * )&sndbufsz, sizeof( sndbufsz ) ) )
```

```

39     error( 1, errno, "setsockopt SO_SNDBUF failed" );

40 if ( connect( s, ( struct sockaddr * )&peer,
41             sizeof( peer ) ) )
42     error( 1, errno, "connect failed" );

43 while( blks-- > 0 )
44     send( s, buf, sndsz, 0 );
45 EXIT( 0 );
46 }

```

tcpsource.c

图 2.35 充当发送方的 TCP 客户端的 main 函数

main

12~30 我们重复调用 getopt 来读取并处理命令行参数。因为以后还会使用这个程序，所以我们在编码时特别考虑了可配置性。命令行选项允许我们设置套接字发送方缓冲区的大小，每次写操作要发送数据的数量，以及要执行写操作的总数量。

31~42 除了增加了一个 setsockopt 调用来设置发送缓冲区大小以及为每个写操作发送的指定数量的数据分配一个缓冲区之外，还有我们标准的 TCP 服务器设置代码。应当注意的是，因为我们并不关心传送的是什么数据，所以我们不需要初始化 buf 指向的缓冲区。

43~44 我们将以指定次数地调用 send 函数。

服务器的 main 函数，如图 2.36 所示，是通过标准的 TCP 服务器框架程序加入 getopt 调用来获得套接字接收缓冲区的大小，并调用 setsockopt 来设置它。

tcpsink.c

```

1 int main( int argc, char **argv )
2 {
3     struct sockaddr_in local;
4     struct sockaddr_in peer;
5     int peerlen;
6     SOCKET sl;
7     SOCKET s;
8     int c;
9     int rcvbufsz = 32 * 1024;
10    const int on = 1;

11    INIT();

```

```
12 opterr = 0;
13 while ( ( c = getopt( argc, argv, "b:" ) ) != EOF )
14 {
15     switch ( c )
16     {
17         case 'b' :
18             rcvbufsz = atoi( optarg );
19             break;
20
21         case '?' :
22             error( 1, 0, "illegal option: %c\n", c );
23     }
24
25     set_address( NULL, "9000", &local, "tcp" );
26     s = socket( AF_INET, SOCK_STREAM, 0 );
27     if ( !isValidsock( s ) )
28         error( 1, errno, "socket call failed" );
29
30     if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR,
31                      ( char * )&on, sizeof( on ) ) )
32         error( 1, errno, "setsockopt SO_REUSEADDR failed" );
33
34     if ( bind( s, ( struct sockaddr * ) &local,
35                sizeof( local ) ) )
36         error( 1, errno, "bind failed" );
37
38     listen( s, 5 );
39
40     peerlen = sizeof( peer );
```

```

41     s1 = accept( s, ( struct sockaddr * )&peer, &peerlen );
42     if ( !isValidsock( s1 ) )
43         error( 1, errno, "accept failed" );
44     server( s1, rcvbufsz );
45     CLOSE( s1 );
46 } while ( 0 );
47 EXIT( 0 );
48 }

```

-tcpsink.c

图 2.36 充当接收方的 TCP 服务器的 main 函数

直到接收 EOF (请参阅技巧 16) 或发生了错误, server 函数一直读取数据并记数。它分配一个等于套接字接收缓冲区大小的缓冲区, 试图一次读取尽可能多的数据。server 函数如图 2.37 所示。

```

1 static void server( SOCKET s, int rcvbufsz )
2 {
3     char *buf;
4     int rc;
5     int bytes = 0;
6
7     if ( ( buf = malloc( rcvbufsz ) ) == NULL )
8         error( 1, 0, "malloc failed\n" );
9     for ( ; ; )
10    {
11        rc = recv( s, buf, rcvbufsz, 0 );
12        if ( rc <= 0 )
13            break;
14        bytes += rc;
15    }
16    error( 0, 0, "%d bytes received\n", bytes );
17 }

```

-tcpsink.c

图 2.37 server 函数

为了帮助我们分析在大量数据传输的情况下 TCP 和 UDP 的相对性能, 我们在 bsd 系统下运行客户端, 而在其他不同的主机上运行服务器。当然, 主机 bsd 和本地主机是一样

的，但是正如我们将要看到的那样，结果却有很大的差异。我们在同一台主机上运行客户端和服务器来获得对没有网络影响下的 TCP 和 UDP 的性能的认识。在这两种情况下，UDP 数据报或 TCP 段都封装在 IP 数据报中并发送给 loopback 接口 lo0，该接口回送这些数据报给 IP 输入路径，如图 2.38 所示。

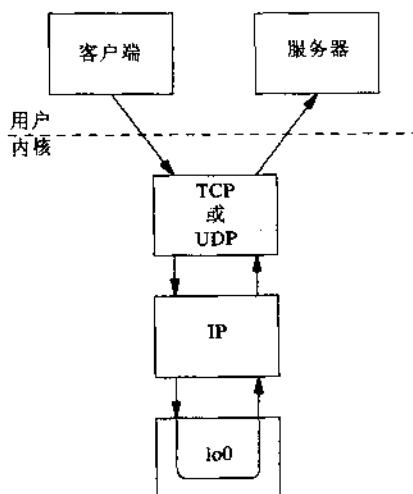


图 2.38 loopback 接口

首先，我们通过传输 1440 字节的数据报（UDP 中）或写入 1440 字节（TCP 中）来运行每个测试程序 50 次。选择 1440 字节大小是因为它近似 TCP 在以太网上传输的最大段的大小。

这个数字来源于一个以太帧至多可以传输 1500 数据字节这样一个事实。因为每个 IP 和 TCP 报头需要 20 字节，所以只剩下 1460 字节。我们为 TCP 选项保留其他 20 个字节。因为在 bsd 系统上，TCP 发送 12 个字节的选项，所以它的实际最大段大小应当是 1448 字节。

图 2.39 总结了 50 次运行的平均结果。表中为每一个协议给出了 3 次测试。Clock 时间是启动客户端和客户端关闭之间花费的时间。User 时间是应用程序在用户空间上执行所花费的时间。System 时间是应用程序执行操作系统内核代码所花费的时间。MB/Sec 栏是发送字节的总数量除以时钟时间的值。UDP 下的 Drops 栏是每次测试数据报丢失的平均数量。

服务器	TCP				UDP				
	Clock	User	System	MB/Sec	Clock	User	System	MB/Sec	Drops
bsd	2.88	0.0292	1.4198	2.5	1.9618	0.0316	1.1934	3.67	336
localhost	0.9558	0.0096	0.6316	7.53	1.9748	0.031	1.1906	3.646	272
sparc	7.1882	0.016	1.6226	1.002	5.8284	0.0564	0.844	1.235	440

图 2.39 TCP 和 UDP 性能比较（1440 字节）

我们注意到的第一件事就是服务器为本机时 TCP 的性能比服务器是 bsd 时高出许多。而 UDP 却不会发生这种情况，在 UDP 下没有重大的性能差别。为了理解为什么 TCP 当客户端在本机时其性能高出一大块，我们调用带-i 选项的 netstat（请参阅技巧 38）。两个相关的输出行（删除了无关的信息）是：

Name	Mtu	Network	Address
ed0	1500	172.30	bsd
lo0	16384	127	localhost

我们可以看到 bsd 的 MTU 是 1500，而本机的 MTU 是 16384。

该操作是 BSD 衍生出来的 TCP 的具体实现中所特有的。它在 Solaris 系统下是不会发生的。当到 bsd 的路由开始建立的时候，路由代码假定它是在 LAN 上，这是因为 IP 地址的网络部分和以太网接口的网络部分是一样的。直到第一次使用路由时，TCP 才发现它是在同一台主机上并把它切换到 loopback 接口。然而，直到那个时候，路由规格包括 MTU 才设置到 LAN 的接口上。

这意味着，当我们发送数据给本机时，TCP 可以发送总共 16384 个字节的段（或者是 $16384 - 20 - 20 - 12 = 16332$ 个字节的数据）；然而在发送给 bsd 时，数据总的字节是前面所讲到的 1448。发送这么大的段意味着发送次数的减少，同时也意味着处理量减少了，IP 和 TCP 给每个段增加的报头的开销也减少了。我们可以看到二者之间的区别：本机的速度比 bsd 的速度快 3 倍。

我们从结果中可以注意到的第二件事情就是，在本机上 TCP 几乎有 UDP 的两倍的速度快。再次指出，这是因为 TCP 可以合并多个 1440 字节，并把它写入到一个段中，而 UDP 每次只发送 1440 字节的数据报。

最后，我们注意到，在局域网上 UDP 大约比 TCP 快 20%，但是 UDP 丢失了大量的数据报。即使是我们在同一台机器上运行客户端和服务器也会发生数据报的丢失，数据报丢失是因为缓冲区空间不够造成的。虽然我们以尽可能快的速度发送 5000 个数据报这似乎可以称为病态的，但是这些结果值得我们去记住。它们说明了：即使我们在同一台机器上运行客户端和服务器的程序，UDP 也不能保证任何特定的数据报安全地到达。

本机和 bsd 结果的对比说明了我们发送数据报的大小也会影响相对性能。例如，如果我们以写入 300 字节的数据报（TCP 中）以及传送 300 字节的数据报（UDP 中）的方式来重新运行测试程序，我们就可以在图 2.40 中发现，TCP 在本机和 LAN 上的性能都要比 UDP 高。

服务器	TCP				UDP				
	Clock	User	System	MB/Sec	Clock	User	System	MB/Sec	Drops
bsd	1.059	0.0124	0.445	1.416	1.6324	0.0324	0.9998	.919	212
sparc	1.5552	0.0084	1.2442	.965	1.9118	0.0278	1.4352	.785	306

图 2.40 TCP 和 UDP 性能比较（300 字节）

从这些例子中得出的重要事实就是，给 TCP 和 UDP 的相对性能做一个假设是错误的。改变一下条件，甚至是以一种似乎微小的方式改变一下条件，就可能极大地改变性能的结果。做出使用哪一个协议是最好的选择方法就是在每一种情况下测试一下我们的应用程序，来看看它们的性能（但是请参阅技巧 8）。当这种方法不太实用时，我们可以编写小的测试工具，正如我们前面所做的那样，来获得有关性能的一些认识。

作为一个实用的东西，现代 TCP 的具体实现都是“需要有多快就有多快”。事实是，TCP 可以轻松地在 100MB 的 FDDI 网络上以“wire speed”运行，而且最近的一些实验在 PC 硬件上已经达到了接近千兆比特的速度[Gallatin et al. 1999]。

1999 年 7 月 29 日，Duke 大学的研究者在 DEC/Compaq Alpha-based XP1000 工作站和 Myrinet 的千兆网络上获得千兆比特的速度。研究者使用的是为零拷贝套接字修改的标准 FreeBSD TCP/IP 栈。相同的实验还在使用旧版本的 Myrinet 网络的 PII 450MHzPC 上获得高于 800Mb/s 的速度。
详情请参阅<<http://www.cs.duke.edu/ari/trapeze>>.

小结

我们已经看到了 UDP 并不一定比 TCP 提供更好的性能。许多因素都可以影响两个协议的相对性能，我们应当总是测试我们的应用程序来看看它们在每个协议上的性能究竟如何。

技巧 8 不要彻底改造 TCP

正如我们在技巧 7 里看到的那样，UDP 在简单请求/应答的应用程序中的性能比 TCP 好很多。这就引导我们考虑使用 UDP 作为那些基于事务的应用程序的内部协议。然而，UDP 协议是一个不可靠的协议，所以应用程序必须自己保障可靠性。

这意味着应用程序必须考虑在网络传输中丢失或破坏的数据报。许多初级网络程序员认为这在 LAN 环境中是一个遥远的事情，因此他们可以有效地忽略这些事情发生的可能性。然而，正如我们在技巧 7 里看到的那样，即使我们是给同一台机器上的应用程序发送数据报，它们也有可能丢失。因此，保护应用程序免受丢失数据报的干扰仍然是不能忽略的问题。

如果应用程序在 WAN 上运行，那还会存在数据报不按顺序到达的可能性。这在从发送者到接收者之间有多个路径时存在是经常发生的。

以前面讨论的观点来看，任何比较健壮的 UDP 应用程序都必须提供（1）在一定时间内没有接收到应答时重传请求，（2）保证应答和请求的对应关系是正确的。

第一个要求可以通过在发送每个请求时设置一个计时器来达到，该计时器称作重传时间超时（retransmission time-out, RTO）计时器。如果计数器在应答到达之前超时了，该请求就必须重新发送。我们将在技巧 20 中讨论处理这种情况的一些更有效的方法。第二个要求可

以通过附加一个序列号到每一个请求上并要求服务器在它的应答里返回该序列号来满足。

如果应用程序将要运行在 Internet 上，那么一个固定的 RTO 值将不是一个很好的选择，因为两个主机之间的 RTT（信号来往时间）值即使在一个很短的时间间隔内也会有很大变化。因此，我们希望当网络条件变化时能够校正 RTO 计时器。而且，即使 RTO 计时器没有超时，也应当合理地为重传增加它的值，因为旧的值好像太小了。这使我们在有必要进行重传时为 RTO 考虑一些指教级的缓冲空间。

其次，如果应用程序将要使用的不仅仅是简单的请求/应答协议，其中客户端发送一个请求然后在发送任何其他的数据报之前等待应答的到达，或者如果服务器的应答可以由多个数据报组成，那么我们需要提供某些类型的流量控制。例如，如果服务器是一个人事数据库应用程序，而客户端的请求是“把工程部门的所有职员的名字和地址发送给我”，应答可能包括多条记录，那么每一条作为一个单独的数据报发送过来。如果没有流量控制，服务器应答的大量的数据报就可能超出客户端的缓冲区。TCP 使用的滑动窗口流量控制（计算数据报的数量而不是字节的多少）就是进行流量控制的普通方法。

最后，如果应用程序涉及到连续传送多个数据报，我们就需要考虑流量控制了。不考虑流量控制就向网络中注入数据报的应用程序很容易给网络中的其他用户带来严重的吞吐量下降问题，并且有可能导致网络失败。

当我们审视应用程序使基于 UDP 协议变得可靠必须采用的所有步骤时，我们发现自己已经极大地改造了 TCP。有时候这是不可避免的。毕竟，有些简单的基于事务的应用程序的 TCP 连接建立和撤消所需的时间可能达到甚至超过了发送和接收实际数据所需的时间。

一个经常可以看到的例子就是域名系统 (DNS)，它用于将一个域名映射到 IP 地址。例如，当我们在 Web 浏览器里输入 www.rfc-editor.org 时，浏览器里的 DNS 客户端发送一个 UDP 数据报给 DNS 服务器，请求和该名称关联的 IP 地址。服务器响应包含适当的 128.9.160.27 IP 地址的数据报。DNS 将在技巧 29 里做进一步的讨论。

而且，我们应当通过仔细地分析每一个应用程序来评估我们这么做是否合理。如果一个应用程序需要 TCP 的可靠性，那么 TCP 就是最好的解决方案。

首先，在一个用户应用程序中重复 TCP 的功能是不可能获得比较好的效率的。这部分是因为 TCP 的具体实现是大量的实验和研究的结果。经过了这么多年，TCP 已经发展成了可以适应不同的环境和网络（包括 Internet）下的最细微的功能。

同时这也是因为 TCP 几乎总是在内核环境中运行。为了弄清楚这为什么会影响性能，让我们假定当应用程序的 RTO 计时器超时的时候到底会发生什么事情。首先，操作系统内核必须唤醒应用程序，这就需要切换到用户空间的上下文（context）；接着应用程序必须重新发送数据，这需要切换回操作系统内核的另一个上下文，必须把数据报从用户空间拷贝到内核里；然后内核为数据报采用正确的路由，把它转移到适当的网络接口，并返回到应用程序，这又需要另一个切换上下文。这时应用程序必须安排再次设置 RTO 计时器，因此需要另一个切换到内核的上下文。

现在考虑一下当 TCP 的 RTO 计时器超时的时候会发生什么。因为内核已经有了数据

的存储拷贝，所以没有必要重新从用户空间里拷贝过来，而且不需要切换上下文。TCP 仅仅重新发送数据。这里的大部分工作主要涉及从内核缓冲区移动数据到网络接口上。因为 TCP 已经保存了这些信息，所以不必要计算路由。

避免在用户级别实现 TCP 的功能的另一个原因关系到当服务器的应答丢失时会发生什么事情。因为客户端没有接收到应答，所以它就会超时并重新发送请求。这意味着服务器再次处理该请求，而这是服务器所不希望的。例如，考虑一个请求服务器从一个银行帐号把资金转到另一个帐号的客户端，如果使用了 TCP，那么重新发送的逻辑就不在应用程序中，因此服务器没有第二次看到请求。

我们在这里把网络失败或者其中一个主机崩溃的可能性丢在一边。请参阅技巧 9，了解更多的关于这方面的东西。

事务处理的应用程序和涉及到使用 TCP 或 UDP 来实现它们的一些问题在 RFC 955 [Braden 1985]中有详细的讨论。在该文档中，Braden 探讨了在无连接但不可靠的 UDP 和可靠但是面向连接的 TCP 协议之间存在第三个协议的需要。在这个 RFC 中的这些考虑使 Braden 推出了 TCP 事物扩展（TCP Extensions for Transactions，T/TCP），我们将在下面讨论它。

使应用程序不仅具有 TCP 的可靠性，而且还要避免连接建立所带来的开销的一个方法就是使用 T/TCP。这是 TCP 的一个扩展，它（通常）通过分散 TCP 为建立连接使用的三次握手以及缩短连接撤销的 TIME-WAIT 状态来达到和 UDP 一样的事务处理。

T/TCP 的基本原理，以及它的具体实现所需的概念在 RFC 1379 [Braden 1992a]中有详尽的描述。RFC 1644 [Braden 1994]不仅包含了 T/TCP 的功能规范，也讨论了一些具体实现的问题。[Stevens 1996]讨论了 T/TCP、讨论了它相对于 UDP 的性能、还讨论了如何改变套接字 API 来支持它以及它在 4.4BSD TCP/IP 栈中的具体实现。

不幸的是，T/TCP 并没有广泛使用，它在 FreeBSD 中实现了，Linux 2.0.32 内核以及 SunOS 4.1.3 有专门为 T/TCP 设计的补丁。

Rich Stevens 建立了一个 T/TCP 个人主页<<http://www.kohala.com/start/ttcp.html>>，其中包括了在本书中所讲到的以及其他的一些 T/TCP 资源。

小结

我们已经讨论了在 UDP 上面创建一个可靠的协议所需的步骤。尽管有一些应用程序，如 DNS，确实是这样做了，但是我们看到了要正确地完成这个任务就必须对 TCP 做很大改动。因为基于 UDP 的协议未必比 TCP 更有效，所以实现了这些功能未必有意义。

我们也简单地讨论了 T/TCP，它是 TCP 的扩展，为基于事务处理的应用程序而进行了优化。虽然 T/TCP 解决了使用 TCP 编写事务处理程序带来的许多问题，但是它并没有广泛地被使用。

技巧 9 注意 TCP 是可靠的协议

但并非是不出错的协议

正如我们前面多次讨论到的那样，TCP 是一个可靠的协议。通常可以这样表述：“TCP 保证了发送数据的传输”。这个表述尽管是很普通，却十分不恰当。

首先，即使是简单地考虑一下也知道它并不正确。例如，假设在数据传输时断开一台主机的网络连接，TCP 并不会为传输剩下的数据做出任何的努力。网络紊乱确实会发生，主机也确实会崩溃，而且用户确实也会在 TCP 连接仍然活动时关闭它们的机器。这些事件以及其他类似事件都会使 TCP 不能传输它已经从应用程序中接收到的数据。

然而，更为重要的是 TCP 的“保证传输”的概念在粗心的网络程序员的脑海中已经产生了一些微妙的影响。当然，没有人真的相信 TCP 有魔力总是能让数据安全地到达它的目的地。对 TCP 保证传输的信任更准确地说是表明了，不需要在编程时处处留心，也没必要考虑失败模式，毕竟 TCP 是保证连接的。

可靠性——什么是可靠性，什么不是可靠性

在我们讨论 TCP 可能发生的失败模式的种类之前，让我们弄清楚究竟什么是 TCP 的可靠性。如果 TCP 不保证传输我们提交给它的所有数据，那么它保证的又是什么呢？首先提出来的问题是：对谁保证？图 2.41 显示了来自应用程序 A 的数据流通过 TCP/IP 栈下行，通过几个中间的路由器，沿着应用程序 B 的主机的 TCP/IP 栈上行，最终到达应用程序 B。当一个 TCP 段离开应用程序 A 的主机的 TCP 层时，该段就封装进一个 IP 数据报中传输到它的对等主机。它所经过的路径可能要它通过几个路由器，但是如图 2.41 所示，这些路由器没有 TCP 层——它们仅转发 IP 数据报。

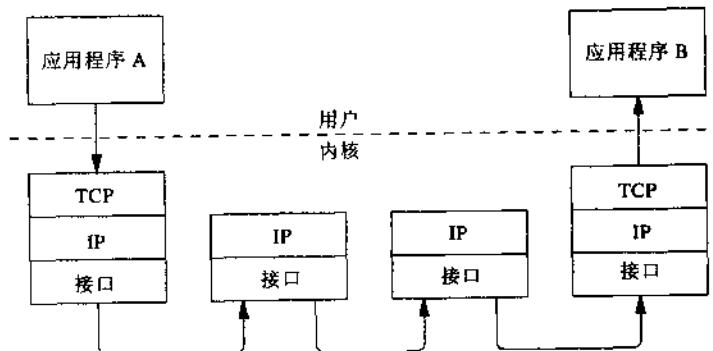


图 2.41 有中间路由器的网络

一些“路由器”实际上就是通用计算机，它们有完整的 TCP/IP 栈，但是在这些情况下，它们的路由功能不涉及到 TCP 或应用程序层。

因为我们知道 IP 是一个不可靠的协议，所以我们应当清楚在数据传输的路径上可以讨论保证传输的第一个地方就是应用程序 B 的主机的 TCP 层。当一个段到达应用程序 B 的主机的 TCP 层时，我们确实知道唯一的事情就是段已经到达了，它有可能被破坏了，有可能是重复的数据，有可能是乱序的，还有可能因为其他的一些原因不可接受该段。注意发送方 TCP 对到达接收方 TCP 的段不会做任何保证。

然而，接收方 TCP 将会对发送方的 TCP 做出保证，该保证就是它确认的任何数据以及在它之前到达的所有数据在 TCP 层上已经正确地接收到了，然后发送方 TCP 可以安全地丢弃数据的拷贝。这并不意味着该数据已经传递给应用程序了，也不意味它一定会传递到。例如，接收方主机可能在确认数据之后但而在应用程序读取数据之前崩溃。我们值得进一步去考虑这个问题：TCP 提供的通知发送方数据已经接收到的唯一的消息就是 ACK。发送方应用程序并没有单独告诉 TCP 数据是否被对等方应用程序实际接收到了。正如我们将在后面提到的那样，这代表了编写应用程序的人应当注意的一种可能的失败模式。

可以讨论保证传输的另一个地方是应用程序 B。正如我们所看到的那样，应用程序 A 对发送的所有数据是否会到达目的地不会做任何保证。TCP 能够对应用程序 B 保证的事情是到达的所有数据是有序并且是没有被破坏的。

数据保证没有被破坏所能够到达的程度取决于 Internet 校验和能够检测错误。因为该校验和是字节对的 16 位偶校验和，它能检测 15 位或少于 15 位的突发错[Plummer 1978]。在统一分布数据的假设条件下，TCP 校验和接受破坏段的概率不高于 $1/(2^{16}-1)$ 。然而，[Stone et al. 1998]显示在一些比如典型的 TCP 段的真实数据中，Internet 校验和失误的概率在一些条件下可能会大大高于理论情况。

失败模式

我们已经看到前面给出的一种失败模式：TCP 确认的数据实际上有可能不会到达它的目的应用程序。正如我们将要讨论的大多数失败模式一样，这是一个相对少见的事件，即使当它发生时，影响也可能是良性的。重要的是，网络程序员能意识到这种可能性，能够在这种情况或任何其他的失败模式出现不愿看到的结果时保护应用程序。我们需要避免的是抱有 TCP 为我们处理了所有的事情、没有必要担心我们应用程序协议的健壮性这样的一种态度。

对于前面讨论的失败模式，解决方法是显而易见的。如果应用程序知道对等方已经接收到了一个特定的消息这事确实很重要，那么对等方就必须通知发送方这个事实。通常这个确认是隐含的。例如，如果一个客户端要求服务器传送一些数据，然后服务器响应，那么该响应就是原来请求的隐含确认。请参阅技巧 19，了解处理显式服务器确认的一个可行的方法。

更困难的一个问题是，如果服务器不确认接收到的数据，客户端将会做什么呢？当然，这个问题的答案很大程度上依赖于特定的应用程序，而且也许没有一个通用的解决方案。然而，我们应当注意到仅仅是重新发送请求并不总是可以接受的：正如我们在技巧 8 里讨论的那样，我们不希望两次请求资金转帐。数据库系统使用三阶段提交协议来处理这种类型的问题，一些应用程序可以借用这个策略来保证操作“至多执行一次”。这种协议的一个例子就是 OSI 参考栈的应用程序层中提供的一般应用程序服务元素（CASE）的并发、提交和恢复（concurrency, commitment, and recovery, CCR）服务。[Jain and Agrawala 1993] 讨论了 CRR 协议以及它是如何工作的。

TCP 是一个终端点到终端点的协议，这意味着通信的双方只关心自己提供了一个可靠的传输机制。然而，认识到“终端点”是对等双方的 TCP 层而不是对等的应用程序是很重要的。需要终端到终端确认的应用程序必须自己提供。

让我们看一下其他“通常”的失败模式。只要双方保持连接，TCP 就能保证按顺序递交数据并且数据不会被破坏。只有当连接中断时失败故障才会发生。什么类型的事件可能导致这样的中断呢？发生中断的情况有三个：

- (1) 可能发生永久的或暂时的网络紊乱。
- (2) 对等方的应用程序可能崩溃。
- (3) 对等方应用程序运行的主机可能崩溃。

正如我们将要看到的那样，其中每一个事件在发送方应用程序中以不同的方法显示。

» 网络紊乱

网络紊乱可能因多种原因而发生，从路由器失败或主干网链接失败到有人踩到以太网线上把接头弄松了。终端点（endpoint）之外发生的失败通常是暂时的，这是因为路由协议就是为找出问题所在地并绕过它而设计的。

终端点的意思是应用程序之一驻留的局域网或主机。

因为当在终端点发生了问题时，通常没有别的可选择的路径，所以在修理之前问题就一直存在。

除非中间的路由器发送一个 ICMP 消息来说明目的网络或主机不可到达，否则应用程序和它们的 TCP/IP 栈都不会立刻意识到发生了紊乱（技巧 10）。在这种情况下，发送方最后超时并重新发送没有确认的段，直到在发送方 TCP 放弃，TCP 放弃之后就中断连接并报告错误。在传统的 BSD 栈中，数据段重发 12 次之后 TCP 就会放弃（大约 9 分钟）。如果一个读操作挂起了，它就返回一个错误条件，errno 设置为 ETIMEDOUT。如果没有挂起读操作，下一个写操作就会失败，返回一个 SIGPIPE 信号。如果该信号被忽略或捕捉到了就返回一个 EPIPE 错误。

如果中间的路由器不能转发包含段的 IP 数据报，它就给原来的主机发送一个 ICMP 消息说明网络或主机不可到达。在这种情况下，一些 TCP 的具体实现把 ENETUNREACH 或

EHOSTUNREACH 作为错误返回。

对等方崩溃

接下来，让我们看看如果对等方应用程序崩溃或中断了究竟会发生什么。从我方应用程序的观点可以认识到的第一件事情就是，对等方崩溃和对等方调用 `close`（或 Windows 应用程序里的 `closesocket`）和 `exit` 是不可区分的。在这两种情况下，对等方的 TCP 都是发送一个 FIN 消息给我们的 TCP。FIN 消息可以充当 EOF，指示发送方没有任何数据可以发送了。没有必要说明发送方已经退出了或者是不希望接收任何数据了。请参阅技巧 16，了解更多有关这方面的东西。应用程序是如何知道接收到 FIN 消息依赖于 FIN 到达时应用程序正在干的事情。为了说明所发生的不同事情，下面创建一个小的 TCP 客户端，该客户端从标准输入读取一行数据，发送给服务器，并读取服务器的应答，把它写入到标准输出。图 2.42 显示的就是该客户端。

8~9 应用程序初始化为 TCP 客户端，连接到命令行里指定的服务器和端口号。

10~15 应用程序一直从标准输入读取数据行，并把它们发送给服务器，直到接收到一个 EOF 字符。

16~20 在给服务器发送数据之后，应用程序读取一行应答。`readline` 函数通过从套接字中一直读取数据直到发现一个新行字符来读取一行数据。`readline` 函数显示在技巧 11 的图 2.55 中。如果调用 `readline` 失败或者返回 EOF（技巧 16），应用程序输出诊断信息并退出。

22 否则，应用程序把应答写到标准输出上。

-tcprw.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     int rc;
6     int len;
7     char buf[ 120 ];
8
9     INIT();
10    s = tcp_client( argv[ 1 ], argv[ 2 ] );
11    while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
12    {
13        len = strlen( buf );
14        rc = send( s, buf, len, 0 );
15        if ( rc < 0 )
16            error( 1, errno, "send failed" );
17    }
18    closesocket( s );
19    return 0;
20 }
```

```

16     rc = readline( s, buf, sizeof( buf ) );
17     if ( rc < 0 )
18         error( 1, errno, "readline failed" );
19     else if ( rc == 0 )
20         error( 1, 0, "server terminated\n" );
21     else
22         fputs( buf, stdout );
23 }
24 EXIT( 0 );
25 }

```

-tcprw.c

图 2.42 以行读取和写入的 TCP 客户端

为了运行该客户端，下面编写了一个服务器，该服务器仅仅循环地从客户端读取来自客户端的数据行，并返回指示已经读取了多少行的消息。为了模拟服务器端的处理延时，在接收的两个消息之间睡眠 5 秒钟然后发送一个应答。服务器如图 2.43 所示。

-count.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     SOCKET s1;
6     int rc;
7     int len;
8     int counter = 1;
9     char buf[ 120 ];

10    INIT();
11    s = tcp_server( NULL, argv[ 1 ] );
12    s1 = accept( s, NULL, NULL );
13    if ( !isValidsock( s1 ) )
14        error( 1, errno, "accept failed" );
15    while ( ( rc = readline( s1, buf, sizeof( buf ) ) ) > 0 )
16    {
17        sleep( 5 );
18        len = sprintf( buf, "received message %d\n", counter++ );

```

```

19     rc = send( s1, buf, len, 0 );
20     if ( rc < 0 )
21         error( 1, errno, "send failed" );
22 }
23 EXIT( 0 );
24 }
```

-count.c

图 2.43 消息记数服务器

为了弄清楚服务器崩溃时会发生什么事情，在 bsd 的不同窗口里启动服务器和客户端：

```

bsd: $ tcprw localhost 9000
hello
received message 1                                this is printed after a 5-second delay
                                                the server is killed here

hello again
tcprw: readline failed: Connection reset by peer (54)
bsd: $
```

我们发送一个消息给服务器，5 秒钟后收到应答。同时，我们杀死服务进程，模拟服务器崩溃，在客户端，好像什么都没发生。这是因为客户端被锁定在调用 fgets 上，而 TCP 无法通知客户端它已从另一端接收到了一个 EOF 信号。如果没有其他操作，客户端保持锁定在调用 fgets 上永远不知道对等方已被终结。

接下来，我们在客户端键入另一行，客户端立即失败，通知我们服务器以重新启动连接。当客户端从 fgets 返回，并不知晓从服务器来的 EOF 信号，此时发生了什么事呢？因为应用程序在接收到一个 FIN 信号后送出数据是完全合理的，客户端 TCP 堆栈尝试发送第二行给服务器。当 TCP 接收到数据，因为连接已经不存在，它返回一个 RST 信号。服务终结。当客户端调用 readline，内核返回一个 ECONNRESET 通知客户端一个 reset 信号被接收。图 2.44 显示了这个过程的时间线。

现在让我们看看在一个呼叫和应答过程中服务器崩溃会造成什么后果。我们再次启动服务器和客户端在 bsd 上的不同窗口：

```

bsd: $ tcprw localhost 9000
hello
                                                the server is killed here

tcprw:server terminated
bsd: $
```

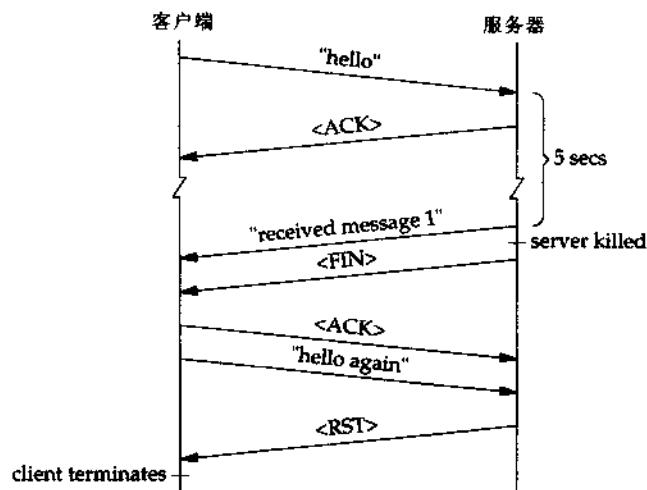


图 2.44 服务器崩溃时间线

客户端发送一行数据给服务器，然后在服务器从 sleep 调用醒来之前中断服务器。这模拟了在服务器结束处理请求之前崩溃的情况。这次，客户端获得了一个即时错误，指示服务器已经中断了。在这个例子中，客户端在调用 readline 时收到 FIN 消息就被阻塞，TCP 通过让 readline 返回 EOF 字符来通知客户端。这些事件的时间顺序如图 2.45 所示。

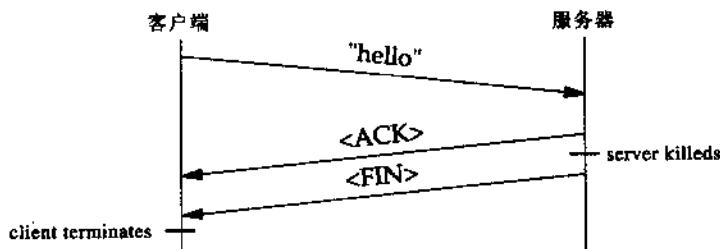


图 2.45 服务器崩溃客户端挂起

如果忽略连接重起错误并继续发送数据，就会发生第三种类型的错误。为了模拟这种情况，改变 readline 后面的 error 调用来输出一个诊断消息，但是程序并不退出。为了做到这一点，把图 2.42 中的第 17 行改为：

```
error( 0, errno, "readline failed" );
```

重新运行第一个测试：

```

bsd: $ tcprw localhost 9000
hello
received message 1
                                         the server is killed here

hello again
tcprw: readline failed: Connection reset by peer (54)
                                         the client ignores the line, but
                                         the connection is dropped by TCP

hello for the last time
Broken pipe
                                         client terminated by SIGPIPE signal
bsd: $

```

当输入第二行时，客户端立刻报告和以前相同的错误（对等方重起连接），但是它并不中断，而是调用另一个函数 fgets 来获取更多的输入发送给服务器。当输入第三行时，客户端立即中断，然后 shell 告诉客户端因为 SIGPIPE 信号而中断。这次，发送第二行数据导致发送和前面一样的 RST 消息，但是忽略了它。然而接收 RST 消息之后，TCP 中断连接，因此当我们试图发送第三行数据时，就会立即终止客户端，并返回 SIGPIPE 信号。这种情况下事件发生的时间顺序和图 2.44 中的差不多。唯一的差别就是客户端是在写数据时终止而不是在读数据之后终止。

当然，优秀的程序通常不会忽略所发生的错误，但是即使是在十分健壮的应用程序里也会发生这种错误模式。考虑一下一个连续执行两个或更多的读操作而中间没有插入读操作的程序。FTP 是一个通常的例子。如果应用程序的对等方崩溃了，对等方的 TCP 就会发送一个 FIN 消息。因为应用程序仅仅发送数据而不接收数据，所以 FIN 消息就通知不到它那里。应用程序发送的下一个段将导致对等方 TCP 发送 RST 消息。应用程序再次接收不到 RST 消息，因为它没有等待的读操作。在对等方崩溃后的第二次写时，应用程序接收到一个 SIGPIPE 信号，如果该信号接收到了或忽略掉了，就会获得 EPIPE 错误。

这种情况在执行多个写操作而没有读操作的应用程序中经常发生，应当理解它的隐含条件。通常是在对等方终止后第二次写时应用程序才接收到通知。因为第一次写操作导致连接重置，所以该写操作写入的数据全部丢失。

根据所处的环境不同，也有可能发生另一种情况。例如，如果我们在 sparc 上运行服务器而在 bsd 上运行客户端，那么重新运行第一个例子，将会得到如下结果：

```

bsd: $ tcprw sparc 9000
hello
received message 1
                                         this is printed after a 5-second delay

```

the server is killed here

```
hello again
tcprw:server terminated
bsd: $
```

这次，客户端在服务器终止时发现服务器产生的 EOF 字符。在发送第二行时仍然产生 RST 消息，但是因为 LAN 的延时，客户端可以调用 readline 并发现服务器已经在接收 RST 消息之前在 bsd 上终止了，如果我们插入：

```
sleep(1);
```

图 2.42 中的第 14 和 15 行之间模拟的是客户端处理或系统加载，当在同一台主机上运行客户端和服务器时获得的是相同的结果。

对等方主机崩溃

下面将要讨论的最后一个失败模式是对等方主机崩溃。它和对等方应用程序崩溃是不同的，因为对等方的 TCP 不能通过 FIN 信号通知本方应用程序对等方已经不再运行。

直到对等方的主机重新启动，该错误和网络紊乱失败模式很相似：对等方 TCP 不再响应。对于网络紊乱，本方应用程序的 TCP 继续传输没有确认的段。最后，如果对等方主机不重新启动，它就放弃并返回 ETIMEDOUT 错误给本方应用程序。

如果对等方主机在本方 TCP 放弃并撤消连接之前重新启动，将会发生什么事情？在这种情况下，本方重传的段之一到达对等方重新启动的主机，而对等方主机没有任何有关这个连接的记录。在这种情况下，TCP 规范[Postel 1981b]要求接收消息的主机返回一个 RST 消息给发送消息的主机，这将导致发送方主机撤消连接，之后应用程序可以看到 ECONNRESET 错误（如果它有读等待）或下一个读操作导致 SIGPIPE 信号或 EPIPE 错误。

本节我们讨论了 TCP 可靠性的概念。我们知道了“保证递交”之类的事情是没有的，实际上 TCP 很容易受到几种类型失败的影响。这些失败模式都不是致命的，但是我们必须准备去处理它们。

技巧 10 记住 TCP/IP 不是轮询

那些初次接触 TCP/IP 协议组但是有其他网络技术经验的网络程序员经常感到很迷惑，甚至很沮丧地发现 TCP 不给应用程序提供即时的网络连接中断的通知。一些程序员据此断定 TCP 不适用于一般的应用程序到应用程序的通信。本节将讨论 TCP 为什么不提供通知，不通知的优缺点是什么，以及应用程序编程人员必须检测什么类型的连接中断。

正如我们在技巧 9 里看到的那样，网络紊乱或系统崩溃可能使双方应用程序不能互相通信，而其中一个应用程序却不能立刻意识到这点。发送数据给对等方的应用程序可能在知道 TCP 放弃重发之前才会发现连接中断。这将花费很长的时间——对 BSD 派生的系统来说大约是 9 分钟。如果应用程序没有发送数据，它可能永远也不会发现网络已经中断了。例如，应用程序可能是一个正在等待对等方发出下一个请求的服务器，因为客户端不能和服务器通信，下一个请求永远不会到达，甚至在客户端的 TCP 放弃并撤消连接，导致客户端中断，服务器也没有意识到这一点。

其他的通信协议，如 SNA 和 X.25，当连接中断时，确实给应用程序提供通知。下面考虑一下它们是怎么做的。比简单的直接点对点专有链接复杂的任何协议都必须使用一种轮询协议来连续地测试对等方是否存在。轮询-选择协议可能会采用显式地发送“你有要发送给我的任何数据吗？”诸如此类的消息的形式，或者它们会采用后台静态帧的形式来监测虚拟线路的是否仍然存在。在任何情况下，这种能力都不会免费达到的。它需要消耗网络带宽。每一个轮询消息都会消耗网络资源，而这些资源本来可以用于“有效负载”数据的传输。

很明显，对可获得的网络带宽的消耗是 TCP 不提供网络中断立即通知的一个原因。因为大多数的应用程序不需要即时的通知，所以没有必要以降低带宽的代价来提供这个功能。需要以一种及时的方式知道对等方不可到达的应用程序可以实现它们自己的机制来发现网络中断，如后面介绍那样。

人们对 TCP/IP 协议组内置通知功能也有哲学上的反对。TCP/IP 设计中使用的一个基本原则是终端对终端参数[Saltzer et al. 1984]，该参数应用到网络上时可以粗略地表述为所有的智能应当尽可能地接近连接的终端点，而网络本身应当相对没有智能。例如，这就是为什么 TCP 自己处理错误控制而不是依靠网络来提供它的原因。正如技巧 1 中所讲的那样，IP（以及在它之上建造的 TCP）对下层网络的依赖很少。当这个原则应用到监控对等应用程序之间的连接时，就意味着应用程序应当提供它自己需要的功能，而不是不管应用程序是否需要这个功能都由下层提供。[Huitema 1995]给出了一个有关终端对终端的有趣的参数以及该参数在 Internet 中的应用。

然而，TCP 不提供及时连接中断通知的最重要的原因是 TCP 需要利用它的一个主要设计目标：网络突然中断时仍可以维持通信的能力。TCP 是国防部发起的一项研究的成果，该项研究要求提供一个遇到战争或自然灾害引起的网络中断时仍然可以维持计算机之间可靠的通信的网络协议。通常网络紊乱是暂时的，路由器也可能找到连接的另一条路径。通过允许连接的暂时中断，甚至在终端应用程序意识到中断之前 TCP 就已经处理好了紊乱。

迫使应用程序实现连接监控的一个缺点就是代码必须内置到每一个应用程序（需要监控连接的应用程序），而且一些不成熟的实现可能浪费带宽或者导致其他对网络和其他用户有害的影响。然而，即使在这里，都可以说服别人通过在应用程序中提供监控，可以很好地满足应用程序的需要，并且可以和应用程序协议尽可能地无缝工作。

Keep-Alive

实际上，TCP 确实提供了一个称作 keep-alive 的机制用于检测死连接，但是正如我们将要看到的那样它并不经常用于应用程序。如果应用程序启用 keep-alive 机制时，TCP 就会在连接已经空闲了一定时间间隔后发送一个特殊的段给对等方。如果对等方主机可到达而且对等方应用程序仍然运行，对等方 TCP 就会响应一个 ACK 应答。在这种情况下，TCP 发送 keep-alive 重置空闲时间为零，并且应用程序不会接收到消息交换的任何通知。

如果对等方主机可以到达但是对等方应用程序没有运行，对等方 TCP 就响应 RST 消息，发送 keep-alive 消息的 TCP 撤消连接并返回 ECONNRESET 错误给应用程序。这通常是对等方主机崩溃后重启的结果，因为如技巧 9 中讨论的那样，如果仅是对等方应用程序中断或崩溃了，对等方 TCP 可能已经发送 FIN 消息了。

如果对等方主机没有响应 ACK 或 RST 消息，发送 keep-alive 消息的 TCP 继续发送 keep-alive 探询消息，直到它认为对等方不可到达或已经崩溃了。这时它就撤消连接并通知应用程序 ETIMEDOUT 错误，如果路由器已经返回主机或网络不可到达的 ICMP 消息的话，就返回 EHOSTUNREACH 或 ENETUNREACH 错误。

需要即时通知网络中断的应用程序使用 keep-alive 功能的第一个问题是和时间间隔有关的。RFC 1122[Braden 1989]认为，如果 TCP 实现了 keep-alive，它就必须有一个至少是 2 小时的空闲间隔的默认值，之后它才能发送 keep-alive 探询消息。那么，因为对等方的 ACK 消息并不是可靠地递交，它必须在放弃连接之前重复发送探询消息。4.4BSD 具体实现再撤消连接之前以 75 秒的时间间隔发送 9 个探询消息。

这是一个实现细节。RFC 1122 没有规定撤消连接之前应当以什么样的频率发送多少探询消息，它仅仅是规定了具体实现不能把单个探询消息没有响应理解为连接已经中断了。

这意味着 BSD 派生的具体实现使用的是默认的 2 小时 11 分钟 15 秒时间来发现连接已经中断了。这个时间值只有在我们认识 keep-alive 是用于释放被死连接占有的资源时才有意义。例如，当客户端连接到服务器而客户端主机崩溃时就有可能发生这样的连接。如果没有 keep-alive 机制，服务器就会永远等待客户端的下一个请求，这是因为它永远没有接收到 FIN 消息。

因为基于 PC 系统的用户仅仅是关闭计算机或 modem 而不是正确地关闭应用程序，所以这种情况正越来越普遍。

一些具体实现允许改变一个或两个时间间隔值，但是这几乎都是在系统范围的基础上实现的。也就是说，这些值的改变影响系统上所有的 TCP 连接，这是 keep-alive 作为一个连接监控机制没有实际使用的主要原因：默认的时间段太长了，如果改变了默认值，它们就失去了清除长时间死连接这一最初的意义。

有一个新的 POSIX 套接字选项 TCP_KEEPALIVE，它允许在每一个连接的基础上指定

超时时间间隔，但是它没有广泛地实现。

`keep-alive` 的另一个问题是它们不仅仅检测死连接，同时也撤消它们。这有可能是应用程序所希望的，但是也有可能不是应用程序所希望的。

Heartbeats

通过在应用程序中实现一个相同的机制就可以轻松地解决使用 `keep-alive` 监控连接中断带来的问题。根据应用程序的不同最好的解决方法也不同，在这里我们可以看到应用程序提供这个机制所带来的灵活性。例如，考虑两个极端的情况：

- (1) 客户端和服务器交换几个不同类型的消息，每一个消息有标识消息类型的消息头。
- (2) 一字节流提供数据给对等方的应用程序没有内在的记录或消息的概念。

第一种情况相对简单。引入一个新的消息类型 `MSG_HEARTBEAT`，该消息可以由一方发送给另一方。接收到 `MSG_HEARTBEAT` 消息之后，应用程序仅仅是返回该消息给对方。正如我们将要看到的那样，它给了我们极大的自由度。连接的一方或者双方都可以监控连接，而实际上只有一方发送 `heartbeat` 消息。

首先看看客户端和服务器都使用的头文件（图 2.46）。

heartbeat.h

```

1 #ifndef __HEARTBEAT_H__
2 #define __HEARTBEAT_H__

3 #define MSG_TYPE1    1          /* application specific msg */
4 #define MSG_TYPE2    2          /* another one */
5 #define MSG_HEARTBEAT 3        /* heartbeat message */

6 typedef struct           /* message structure */
7 {
8     u_int32_t type;       /* MSG_TYPE1, ... */
9     char data[ 2000 ];
10 } msg_t;

11 #define T1             60      /* idle time before heartbeat */
12 #define T2             10      /* time to wait for response */

13 #endif /* __HEARTBEAT_H__ */

```

heartbeat.h

图 2.46 heartbeat 头文件

3~5 这些列出来的常量定义了客户端和服务器交换的不同类型消息。本例中只有 MSG_HEARTBEAT 消息是有意义的。

6~10 `typedef` 定义了客户端和服务器之间传递的消息的结构。再次说明，本例只使用了 `type` 域。实际的应用程序可能会根据需要定制该结构。请参阅图 2.31 中的注释，了解 `u_int32_t` 的意义以及假设压缩结构的危害性。

11 该常量定义了在客户端发送 `heartbeat` 消息给对等方之前连接可以空闲的时间量。我们任意地选择了 60 秒，但是实际的应用程序可能会根据它的需要以及网络的类型来选择一个值。

12 另一个时间值是客户端等待 `heartbeat` 消息响应的时间量。

接下来，图 2.47 显示的是客户端的程序，它初始化 `heartbeat` 消息。这个选项完全是任意的，而且我们可以同样地选择服务器作为初始化的一方。

hb_client.c

```

1 #include "etcp.h"
2 #include "heartbeat.h"

3 int main( int argc, char **argv )
4 {
5     fd_set allfd;
6     fd_set readfd;
7     msg_t msg;
8     struct timeval tv;
9     SOCKET s;
10    int rc;
11    int heartbeats = 0;
12    int cnt = sizeof( msg );

13    INIT();
14    s = tcp_client( argv[ 1 ], argv[ 2 ] );
15    FD_ZERO( &allfd );
16    FD_SET( s, &allfd );
17    tv.tv_sec = T1;
18    tv.tv_usec = 0;
19    for ( ; ; )
20    {
21        readfd = allfd;
22        rc = select( s + 1, &readfd, NULL, NULL, &tv );

```

```

23     if ( rc < 0 )
24         error( 1, errno, "select failure" );
25     if ( rc == 0 )      /* timed out */
26     {
27         if ( ++heartbeats > 3 )
28             error( 1, 0, "connection dead\n" );
29         error( 0, 0, "sending heartbeat #%d\n", heartbeats );
30         msg.type = htonl( MSG_HEARTBEAT );
31         rc = send( s, ( char * )&msg, sizeof( msg ), 0 );
32         if ( rc < 0 )
33             error( 1, errno, "send failure" );
34         tv.tv_sec = T2;
35         continue;
36     }
37     if ( !FD_ISSET( s, &readfd ) )
38         error( 1, 0, "select returned invalid socket\n" );
39     rc = recv( s, ( char * )&msg + sizeof( msg ) - cnt,
40                cnt, 0 );
41     if ( rc == 0 )
42         error( 1, 0, "server terminated\n" );
43     if ( rc < 0 )
44         error( 1, errno, "recv failure" );
45     heartbeats = 0;
46     tv.tv_sec = T1;
47     cnt -= rc;           /* in-line readn */
48     if ( cnt > 0 )
49         continue;
50     cnt = sizeof( msg );
51     /* process message */
52 }
53 }
```

hb_client.c

图 2.47 带有 heartbeat 的基于消息的客户端

✍ 初始化

- 13~14 执行标准的初始化并以命令行中指定的主机和端口号连接到服务器。
 15~16 为连接套接字设置 select 掩码。
 17~18 初始化计时器为 T1 秒，如果在 T1 秒内没有接收到消息，select 就返回时间超时错误。
 21~22 设置读 select 掩码，然后在调用 select 时阻塞，直到套接字中有了数据或者时间超时。

✍ 处理时间超时

27~28 如果应用程序已经发送多于三个连续的 heartbeats 消息而没有应答，就断定连接已经中断了。在这个例子中，应用程序仅仅是退出，但是实际的应用程序可以采用任何它认为合适的措施。

29~33 如果应用程序仍然没有耗尽 heartbeat 探询消息，就给对等方发送一个 heartbeat 消息。

34~35 设置计时器为 T2 秒。如果应用程序在该时间范围内没有从对等方接收到消息，应用程序发送另一个 heartbeat 消息，或者根据 heartbeats 的值断定连接已经中断了。

✍ 消息处理

37~38 如果 select 返回的不是连接到对等方的套接字，应用程序退出并返回致命错误。
 39~40 应用程序调用 recv 读取一个消息。以下包含 cnt 变量的程序行和代码本质上是 readn 的内嵌版本。应用程序没有直接调用 readn，这是因为它可以不确定地禁用 heartbeat 计时。

41~44 如果应用程序接收到一个 EOF 字符或者读错误，它就输出诊断消息并退出。
 45~46 因为应用程序已经接收到对等方的消息，它就设置 heartbeat 记数为零并重置记数器为 T1 秒。

47~50 该代码实现了内嵌的 readn。应用程序用读取的数据来减少 cnt。如果有更多的数据可以读取，应用程序继续调用 select；否则，应用程序为完整的消息重置 cnt，并结束刚才已经读取消息的处理。

图 2.48 显示的是本例子的服务器程序。我们选择让服务器也监控连接，但没有必要严格地使用它。

hb_server.c

```

1 #include "etcp.h"
2 #include "heartbeat.h"

3 int main( int argc, char **argv )
4 {

```

```
5   fd_set allfd;
6   fd_set readfd;
7   msg_t msg;
8   struct timeval tv;
9   SOCKET s;
10  SOCKET s1;
11  int rc;
12  int missed_heartbeats = 0;
13  int cnt = sizeof( msg );

14 INIT();
15 s = tcp_server( NULL, argv[ 1 ] );
16 s1 = accept( s, NULL, NULL );
17 if ( !isValidSock( s1 ) )
18     error( 1, errno, "accept failed" );
19 tv.tv_sec = T1 + T2;
20 tv.tv_usec = 0;
21 FD_ZERO( &allfd );
22 FD_SET( s1, &allfd );
23 for ( ; ; )
24 {
25     readfd = allfd;
26     rc = select( s1 + 1, &readfd, NULL, NULL, &tv );
27     if ( rc < 0 )
28         error( 1, errno, "select failure" );
29     if ( rc == 0 )      /* timed out */
30     {
31         if ( ++missed_heartbeats > 3 )
32             error( 1, 0, "connection dead\n" );
33             error( 0, 0, "missed heartbeat # %d\n",
34                     missed_heartbeats );
35         tv.tv_sec = T2;
36         continue;
37     }
38     if ( !FD_ISSET( s1, &readfd ) )
39         error( 1, 0, "select returned invalid socket\n" );
```

```

40     rc = recv( s1, ( char * )&msg + sizeof( msg ) - cnt,
41             cnt, 0 );
42     if ( rc == 0 )
43         error( 1, 0, "client terminated\n" );
44     if ( rc < 0 )
45         error( 1, errno, "recv failure" );
46     missed_heartbeats = 0;
47     tv.tv_sec = T1 + T2;
48     cnt -= rc;           /* in-line readn */
49     if ( cnt > 0 )
50         continue;
51     cnt = sizeof( msg );
52     switch ( ntohs( msg.type ) )
53     {
54     case MSG_TYPE1 :
55         /* process type1 message */
56         break;
57     case MSG_TYPE2 :
58         /* process type2 message */
59         break;
60     case MSG_HEARTBEAT :
61         rc = send( s1, ( char * )&msg, sizeof( msg ), 0 );
62         if ( rc < 0 )
63             error( 1, errno, "send failure" );
64         break;
65     default :
66         error( 1, 0, "unknown message type (%d)\n",
67                ntohs( msg.type ) );
68     }
69 }
70 EXIT( 0 );
71 }

```

hb_server.c

图 2.48 带有 heartbeat 的基于消息的服务器

初始化

- 14~18 执行标准的初始化并接收来自对等方的连接。
 19~20 设置初始计时器值为 T_1+T_2 秒。因为对等方在不活动 T_1 秒之后发送一个 heartbeat 消息，所以程序通过增加 T_2 秒给超时值来允许额外的时间。
 21~22 接下来，程序为连接到对等方的套接字的可读性初始化 select 掩码。
 25~28 程序调用 select 并处理任何错误返回。

超时处理

- 31~32 如果程序错过了多于三个连续的 heartbeat 消息，它就断定连接已经中断了并退出。和客户端一样，这时服务器可以采取任何其他合适的操作。
 35 计时器重置为 T_2 秒。直到现在，客户端应当每隔 T_2 秒发送 heartbeat 消息。程序在那个时间量之后就会超时，计算丢失的 heartbeat 消息。

消息处理

- 38~39 程序做与在客户端里所做的相同的无效套接字检查。
 40~41 跟客户端一样，程序调用内嵌的 readn 函数。
 42~45 如果 recv 返回一个 EOF 或错误，程序输出诊断消息并退出。
 46~47 因为程序已经从对等方接收到一个消息，而且程序知道连接仍然是活动的，所以程序重置丢失的 heartbeat 消息数为零，计数器值也重置为 T_1+T_2 秒。
 48~51 该代码和客户端的相似，实现了内嵌的 readn 函数。
 60~64 如果这是一个 heartbeat 消息，程序把它返回给对等方。当对等方接收到这个消息时，双方都知道连接仍然是活动的。

为了测试这些程序，在 sparc 上运行 hb_server 而在 bsd 上运行 hb_client。在 hb_client 已经连接到 hb_server 之后，中断 sparc 跟网络的连接。服务器和客户端的输出并排显示如下：

<pre>sparc: \$ hb_server 9000 hb_server: missed heartbeat #1 hb_server: missed heartbeat #2 hb_server: missed heartbeat #3 hb_server: connection dead sparc: \$</pre>	<pre>bsd: \$ hb_client sparc 9000 hb_client: sending heartbeat #1 hb_client: sending heartbeat #2 hb_client: sending heartbeat #3 hb_client: connection dead dsb: \$</pre>
---	--

另一个例子

当一方发送给另一方一串没有内部消息或记录边界的数据时我们在上一个例子中使用的模式就不能很好地工作了。问题是，如果程序发送一个 heartbeat 消息，该消息正好出现在其他数据的中间，就不得不检验一遍，而且有可能如技巧 6 里讨论的那样被忽略掉。为

了避免这些复杂情况，下面采用完全不同的方法。

本方法的思想就是为 heartbeat 消息使用独立的连接。使用一个连接来监控另一个连接好像有些怪异，但是请记住要检测的是对等方主机是否崩溃或者网络是否中断了。如果发生了这两个事件之一，就会影响到两个连接，否则两个都不会影响。一个普通的解决方法是使用独立的线程来控制 heartbeat 函数。另一个方法是使用将在技巧 20 里开发的通用的时间机制。然而，出于简单以及避免担心 Win32 线程 API 和 UNIX 线程 API 之间的差异的考虑，这里给出的代码仅修改了在前面例子中使用的 select 代码。

新版本的客户端和服务器和原来的基本相似。主要的区别是在 select 代码中，该代码现在必须处理两个套接字，而且在附加的代码中设置额外的连接。客户端在连接到服务器之后，就发送给服务器一个端口号，服务器在这个端口上监听 heartbeat 连接。这种方案和 FTP 服务器给客户端建立数据连接的方案相似。

如果使用 NAT（请参阅技巧 3）来转换内部网络地址为公用网络地址时，这就有可能引发问题。

和 FTP 不一样的是，NAT 软件不知道改变该端口号为一个重映射的端口号。在这种情况下，最简单的方法就是给应用程序分配另一个“已知”端口。

图 2.49 中给出的是客户端的初始化和连接代码。

hb_client2.c

```

1 #include "etcp.h"
2 #include "heartbeat.h"

3 int main( int argc, char **argv )
4 {
5     fd_set allfd;
6     fd_set readfd;
7     char msg[ 1024 ];
8     struct timeval tv;
9     struct sockaddr_in hblisten;
10    SOCKET sdata;
11    SOCKET shb;
12    SOCKET slisten;      ▶
13    int rc;
14    int hblistenlen = sizeof( hblisten );
15    int heartbeats = 0;
16    int maxfd1;
17    char hbmsg[ 1 ];

```

```

18 INIT();
19 slisten = tcp_server( NULL, "0" );
20 rc = getsockname( slisten, ( struct sockaddr * )&hblisten,
21 &hblistenlen );
22 if ( rc )
23     error( 1, errno, "getsockname failure" );
24 sdata = tcp_client( argv[ 1 ], argv[ 2 ] );
25 rc = send( sdata, ( char * )&hblisten.sin_port,
26 sizeof( hblisten.sin_port ), 0 );
27 if ( rc < 0 )
28     error( 1, errno, "send failure sending port" );
29 shb = accept( slisten, NULL, NULL );
30 if ( !isValidsock( shb ) )
31     error( 1, errno, "accept failure" );
32 FD_ZERO( &allfd );
33 FD_SET( sdata, &allfd );
34 FD_SET( shb, &allfd );
35 maxfd1 = ( sdata > shb ? sdata: shb ) + 1;
36 tv.tv_sec = T1;
37 tv.tv_usec = 0;

```

hb_client2.c

图 2.49 客户端初始化和连接代码

✍ 初始化和连接

19~23 首先程序一端口号 0 调用 `tcp_server`, 这将导致操作系统内核为程序分配一个暂时的端口（请参阅技巧 18）；然后程序调用 `getsockname` 来获取操作系统内核选择的端口号。程序做这些事情，以使仅有一个已知端口关联到服务器上。

24~28 接下来的步骤是连接到服务器并发送端口号给服务器，该端口号用于 `heartbeat` 连接。

29~31 调用 `accept` 后一直阻塞，直到服务器和客户端建立 `heartbeat` 连接。在一个具备产品性能的程序中，应当在这个调用的附近给出一个计时器，以防服务器永远不建立 `heartbeat` 连接而挂起。程序也应当检查建立 `heartbeat` 连接的主机实际上是第 24 行连接的服务器。

32~37 程序初始化 `select` 掩码和计时器。

客户端的其余部分如图 2.50 所示。在这部分我们看到了消息和 `heartbeat` 处理。

hb_client2.c

```
38 for ( ; ; )
39 {
40     readfd = allfd;
41     rc = select( maxfd1, &readfd, NULL, NULL, &tv );
42     if ( rc < 0 )
43         error( 1, errno, "select failure" );
44     if ( rc == 0 )      /* timed out */
45     {
46         if ( ++heartbeats > 3 )
47             error( 1, 0, "connection dead\n" );
48         error( 0, 0, "sending heartbeat #%" );
49         rc = send( shb, "*", 1, 0 );
50         if ( rc < 0 )
51             error( 1, errno, "send failure" );
52         tv.tv_sec = T2;
53         continue;
54     }
55     if ( FD_ISSET( shb, &readfd ) )
56     {
57         rc = recv( shb, hbmgs, 1, 0 );
58         if ( rc == 0 )
59             error( 1, 0, "server terminated (shb)\n" );
60         if ( rc < 0 )
61             error( 1, errno, "bad recv on shb" );
62     }
63     if ( FD_ISSET( sdata, &readfd ) )
64     {
65         rc = recv( sdata, msg, sizeof( msg ), 0 );
66         if ( rc == 0 )
67             error( 1, 0, "server terminated (sdata)\n" );
68         if ( rc < 0 )
69             error( 1, errno, "recv failure" );
70         /* process data */
71     }
72     heartbeats = 0;
```

```

73     tv.tv_sec = T1;
74 }
75 }
```

hb_client2.c

图 2.50 客户端消息处理

处理数据和 heartbeat 消息

40~43 程序调用 select 并处理返回的任何错误。

44~54 除了程序在 shb 套接字上发送 heartbeat 消息之外，超时处理和图 2.47 里的相同。

55~62 如果数据在 shb 套接字上已经准备好，程序就读取 heartbeat 字节，但是不对它进行任何操作。

63~71 如果数据在 sdata 套接字上已经准备好，程序就读取尽可能多的数据然后处理它。注意因为程序不再处理固定长度的消息，所以程序读取所有可获得的数据直到缓冲区满。如果数据少于缓冲区可以保存的数量，recv 函数就返回可获得的数据，但是它不会阻塞。如果数据多于缓冲区可以保存的数量，那么套接字仍然可以读出数据，下一个 select 函数立即返回，然后我们可以处理下一个缓冲的数据。

72~73 因为程序已经从对等方接收到了消息，程序重置 heartbeats 变量和计时器。

最后，检查一下本例子的服务器代码（如图 2.51）。跟客户端一样，除了连接和处理多个套接字的代码之外，它几乎和原来的服务器（如图 2.48）一模一样。

hb_sever2.c

```

1 #include "etcp.h"
2 #include "heartbeat.h"

3 int main( int argc, char **argv )
4 {
5     fd_set allfd;
6     fd_set readfd;
7     char msg[ 1024 ];
8     struct sockaddr_in peer;
9     struct timeval tv;
10    SOCKET s;
11    SOCKET sdata;
12    SOCKET shb;
13    int rc;
14    int maxfd1;
```

```

15 int missed_heartbeats = 0;
16 int peerlen = sizeof( peer );
17 char hbmsg[ 1 ];

18 INIT();
19 s = tcp_server( NULL, argv[ 1 ] );
20 sdata = accept( s, ( struct sockaddr * )&peer,
21     &peerlen );
22 if ( !isValidsock( sdata ) )
23     error( 1, errno, "accept failed" );
24 rc = readn( sdata, ( char * )&peer.sin_port,
25     sizeof( peer.sin_port ) );
26 if ( rc < 0 )
27     error( 1, errno, "error reading port number" );
28 shb = socket( PF_INET, SOCK_STREAM, 0 );
29 if ( !isValidsock( shb ) )
30     error( 1, errno, "shb socket failure" );
31 rc = connect( shb, ( struct sockaddr * )&peer, peerlen );
32 if ( rc )
33     error( 1, errno, "shb connect error" );
34 tv.tv_sec = T1 + T2;
35 tv.tv_usec = 0;
36 FD_ZERO( &allfd );
37 FD_SET( sdata, &allfd );
38 FD_SET( shb, &allfd );
38 maxfd1 = ( sdata > shb ? sdata : shb ) + 1;

```

hb_sever2.c

图 2.51 服务器初始化和连接

➤ 初始化和连接

19~23 服务器监听并接收来自客户端的数据连接。注意服务器也获取 `peer` 中客户端的地址，以建立 heartbeat 连接。

24~27 接下来，服务器读取客户端正在监听 heartbeat 连接的端口号。服务器直接读取端口号到 `peer` 结构中。因为端口已经是网络字节顺序，所以没有必要担心调用 `hton`s 或 `ntoh`s，这也是服务器必须把它存储在 `peer` 中的原因。

28~33 在获得 shb 套接字之后，服务器建立 heartbeat 连接。

34~39 服务器初始化计时器和 select 掩码。

下面以检查图 2.52 中的服务器代码的方式结束本例子。

hb_sever2.c

```

40  for ( ; ; )
41  {
42      readfd = allfd;
43      rc = select( maxfd1, &readfd, NULL, NULL, &tv );
44      if ( rc < 0 )
45          error( 1, errno, "select failure" );
46      if ( rc == 0 ) /* timed out */
47      {
48          if ( ++missed_heartbeats > 3 )
49              error( 1, 0, "connection dead\n" );
50          error( 0, 0, "missed heartbeat #%d\n",
51                  missed_heartbeats );
52          tv.tv_sec = T2;
53          continue;
54      }
55      if ( FD_ISSET( shb, &readfd ) )
56      {
57          rc = recv( shb, hbmsg, 1, 0 );
58          if ( rc == 0 )
59              error( 1, 0, "client terminated\n" );
60          if ( rc < 0 )
61              error( 1, errno, "shb recv failure" );
62          rc = send( shb, hbmsg, 1, 0 );
63          if ( rc < 0 )
64              error( 1, errno, "shb send failure" );
65      }
66      if ( FD_ISSET( sdata, &readfd ) )
67      {
68          rc = recv( sdata, msg, sizeof( msg ), 0 );
69          if ( rc == 0 )
70              error( 1, 0, "client terminated\n" );
71          if ( rc < 0 )

```

```

72         error( 1, errno, "recv failure" );
73
74     /* process data */
75
76     missed_heartbeats = 0;
77     tv.tv_sec = T1 + T2;
78 }
79 }
```

hb_sever2.c

图 2.52 服务器消息处理

- 42~45 和客户端一样，服务器调用 select 并处理错误。
- 46~53 超时处理照搬图 2.48 中原来的服务器。
- 55~65 如果 shb 套接字可读取数据，服务器就从套接字中读取单字节的 heartbeat 消息，并返回给客户端。
- 66~74 如果数据连接中有数据，服务器就读取并处理它，检查错误和 EOF 字符。
- 75~76 因为服务器已经从对等方读取数据，它知道连接仍然是活动的，所以服务器重置丢失的 heartbeat 消息计数和计时器。

当运行这个客户端和服务器，然后通过从网络中断开主机连接来模拟网络紊乱时，就会获得跟运行 *hb_server* 和 *hb_client*一样的结果。

小结

在本节中，我们已经知道了虽然 TCP 不是以一种及时的方式提供通知连接中断的方法，但是应用程序自己加入这个功能是很容易的。本节提供了 heartbeat 函数，并探讨了该函数的两种不同的模式。虽然显得有些过度重视这个问题，但是两个模式在各种情况下都不是最好的。

第一个例子考虑 heartbeat 消息，应用程序交换的消息都包含标识消息类型的域。该方法的优点是简单：只需要为 heartbeat 增加另一种类型的消息。

第二个例子讨论了应用程序交换的是没有内部消息边界和关联类型域的数据流。这种类型的数据传输的一个例子就是一系列的按键。在这种情况下，本节使用独立的连接来发送和接收 heartbeat 消息。当然这个方法也可以用于交换消息的应用程序，但是它要比仅仅增加一个 heartbeat 消息类型要复杂得多。

《UNIX 网络编程》[Stevens 1998]第 21.5 小节开发了使用 TCP 紧急数据的方法来提供 heartbeat 的另一个方法。再次指出，这说明了应用程序编程人员实现 heartbeat 函数可以使

用的方法具有灵活性。

最后，我们应当认识到尽管本节的讨论主要集中在 TCP 上，相应的原则一样适用于 UDP。考虑一个广播消息给 LAN 上的几个客户端或者广播数据给 WAN 上几个客户端的服务器，因为没有连接，所以客户端就不能意识到服务器崩溃、服务器主机崩溃或者网络紊乱。前面讨论的为 TCP 提供 heartbeat 的两种方法都可以稍作修改就用于 UDP。

如果数据报包含一个类型域，那么服务器仅需要定义一个 heartbeat 数据报类型并在一段时间间隔内没有其他消息时发送它。同样，可以广播 heartbeat 数据报给一个独立的端口，客户端也在该端口上监听。

技巧 11 为来自对等方的不合要求的行为做准备

在编写网络应用程序时，通常因为对等方是在远程机器上而认为不值得在应用程序中处理某个特定的事件。在这点上，我们应当牢记下面摘自主机需求 RFC (RFC 1122[BrADEN 1989]，第 12 页) 的一段话：

软件应当编写成能够处理任何想像的到的错误，不管该错误可能发生的概率是如何的小：数据包迟早会以错误和属性的特定组合到达，除非软件已经为错误做好了准备，否则难免会发生混乱。通常，最好假定网络充满着恶意的实体，它们会发送特意设计的具有最坏影响的数据包。该假设将导致相应的保护设计，尽管在 Internet 上发生的最严重的问题是由小概率事件触发的不可预料机制导致的；仅仅是人类的恶意永远也不会导致如此曲折的路径！

今天，这个建议显得比当初写它的时候更加重要。现在有很多 TCP 的具体实现，其中一些已经很严重地崩溃了。相应地，编写网络应用程序的工程师的数量正在增加，很多工程师缺乏前辈网络程序员的经验。

然而，最大的因素有可能是连接到 Internet 的一般 PC 数量的增加。过去，用户可能会指望有一定数量的技术专家，理解诸如没有退出网络应用程序就关闭计算机之类操作的后果，现在远不是这样的情形了。

结果是，不管这种操作发生的可能性有多小，程序自我保护并试图预测对等方会采取的每一个操作都是十分重要的。在讨论 TCP 失败模式时，技巧 9 已经涉及了这方面的一些东西；在讨论探测连接中断时，技巧 10 也涉及了一些。本技巧将讨论对等方可能做的导致故障的一些事情。应当牢记的规则是，不能假设对等方会严格遵守应用程序协议，甚至是在我们实现双方协议的时候也应当这样。

检查客户端终止

例如假定客户端通过在一行数据里给服务器发送字符串 “quit” 说明它已经结束请求了，进一步假设服务器使用 `readline` 函数（如图 2.55 所示）读取输入行，该函数在技巧 9 中首次出现。如果客户端崩溃了或者在发送 `quit` 命令之前终止了那将会发生什么呢？客

客户端 TCP 发送 FIN 消息给对等方 TCP，服务器读取并返回 EOF 字符。当然，检测到这点很容易，但是服务器必须检测它。假设客户端将会做它应当做的事情是很容易的事，代码如下：

```
for( ; ; )
{
    if (readline(s, buf, sizeof(buf)) < 0)
        error(1, errno, "readline failure" );
    if (strcmp( buf, "quit\n" ) == 0)
        /* do client termination functions*/
    else
        /*process request*/
}
```

虽然这段代码表面上看起来是正确的，但是如果客户端没有发送 quit 命令就终止，它就有可能在重复处理最后的请求时失败。

下面，假定我们发现前面代码段中出现的问题并改变代码明确地检查 EOF 字符：

```
for( ; ; )
{
    rc = readline(s, buf, sizeof(buf));
    if (rc < 0 )
        error(1, errno, "readline failure" );
    if (rc= = 0 || strcmp( buf, "quit\n" ) = = 0)
        /* do client termination functions*/
    else
        /*process request*/
}
```

因为没有考虑在发送 quit 命令或中断之前客户端主机崩溃将会发生什么，该段代码仍然是不正确的。甚至在我们发现了问题时，也很容易做出不好的决策。检查客户端主机的崩溃需要在 readline 函数附近设置计时器，而且如果希望进行某些客户端终端处理的话需要大约两倍多的代码。面对这么多需要增加的代码，程序员会想：“唔，客户端崩溃的可能性到底有多大？”

问题是并不需要客户端主机崩溃就会导致以上问题。如果主机是一个 PC，用户没有终止应用程序就可能简单地关闭计算机——例如，客户端可能运行在一个最小化或被其他窗

窗口覆盖的窗口中，就很有可能忘记客户端仍在运行。同时也存在另一种可能性，如果应用程序之间的网络连接经过客户端主机的 modem（今天大多数的 Internet 连接都是这么建立的），用户可能简单地关闭 modem，或者电话线上出现噪音导致 modem 关闭连接。所有的这些情况对服务器来说，都和客户端主机崩溃没有区别。

在某些情况下，可能通过重新拨号来恢复由 modem 引起的问题（回忆一下前面讲到的 TCP 可以从暂时的网络紊乱恢复），但是通过电话线连接的双方的 IP 地址通常是在连接建立时动态地由 ISP 分配的。在这些情况下，不可能分配到相同的地址，所以客户端就不可能恢复连接。

正如我们在技巧 10 里讨论的那样，实现 heartbeat 来检测客户端终端并不是必须的。程序只需要在读操作附近放一个计时器，如果客户端在一段时间内没有任何请求，服务器就可以断定客户端已经断开了。大多数 FTP 服务器采用这种策略：如果客户端在一定时间间隔内没有任何请求，服务器就退出连接。这可以通过计时器或使用 select 计时器很容易达到，如 heartbeat 例子中所做的那样。

如果我们关心的仅仅是服务器不会永远挂起，那么程序就可以使用 keep-alive，在 keep-alive 计时器时间到头时退出连接。图 2.53 显示了一个简单的 TCP 服务器，该服务器从客户端接收连接，从套接字读取数据，把结果写入标准输出。为了防止服务器很长时间挂起，可以通过调用 setsockopt 启用 keep-alive。setsockopt 的第四个参数必须指向一个非零的整数就能启用 keep-alive，如果它为零就禁用 keep-alive。

keep.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     SOCKET s1;
6     int on = 1;
7     int rc;
8     char buf[ 128 ];
9
10    INIT();
11    s = tcp_server( NULL, argv[ 1 ] );
12    s1 = accept( s, NULL, NULL );
13    if ( !isValidsock( s1 ) )
14        error( 1, errno, "accept failure\n" );
15    if ( setsockopt( s1, SOL_SOCKET, SO_KEEPALIVE,
16                    ( char * )&on, sizeof( on ) ) )

```

```

16     error( 1, errno, "setsockopt failure" );
17     for ( ;; )
18     {
19         rc = readline( sl, buf, sizeof( buf ) );
20         if ( rc == 0 )
21             error( 1, 0, "peer disconnected\n" );
22         if ( rc < 0 )
23             error( 1, errno, "recv failure" );
24         write( 1, buf, rc );
25     }
26 }
```

keep.c

图 2.53 使用 keep-alive 的服务器

在 bsd 上启动该服务器，使用另一个系统上的 telnet 连接该服务器。在连接完成以及发送“hello”确认连接已经建立之后，断开另一个系统的网络连接。服务器的输出如下所示：

```

bsd: $ keep 9000
hello
Other system disconnected from network
...
2 hours, 11 minutes, 15 seconds later
keep: recv failure: Operation timed out (60)
bsd: $
```

正如所料，bsd 系统上的 TCP 撤消连接并返回 ETIMEDOUT 错误给服务器。这时，服务器终止并释放所有的资源。

» 检查输入的有效性

在编写任何类型的程序时有一个共同的原则，那就是不能假定应用程序将只会接收它计划处理的那些数据。忽略这个原则是程序失去自我保护的又一个原因，我们希望专业程序员编写的商业软件应当遵循该原则。令人惊奇的是，该原则经常被忽略。在[Miller et al. 1995]中，研究人员把随机产生的输入供给来自多个供应商的标准 UNIX 实用程序。6-43%（测试不同供应商的软件，结果也不同）的实用程序接收这些输入后崩溃（成为转储内存映像）或挂起（陷入死循环）。测试的 7 个商业系统平均失败率为 23%。

应用程序崩溃的两个主要原因是：缓冲区溢出和指针失控。实际上，在前面提到的研究中，这两个错误导致大部分的失败。有的人也许会认为网络程序中缓冲区溢出的情况很少见，这是因为程序在调用读操作函数（read、recv、recvfrom、readv 和 readmsg）时总是要指定缓冲区的大小。但是正如我们将要看到的那样，犯这种错误是很容易的。实际上，可以在技巧 16 的 shutdownc.c 程序中的第 42 行的注释中看到一个实际的例子。

为了弄清楚这将会发生什么，下面开发 readline 函数，它在技巧 9 中首次使用。该函数的目标是从套接字中读取一行数据到 bufptr 指向的缓冲区中，并以空值结束它。readline 的定义如下：

```
#include "etcp.h"

int readline( SOCKET s, char *buf, size_t len );
>Returns: number of bytes read or -1 on error
```

可以马上放弃的方法是下面的代码：

```
while ( recv ( fd, &c, 1, 0 ) == 1 )
{
    *bufptr++ = c;
    if ( c == '\n' )
        break;
}
/* check for errors, null terminate, etc. */
```

该段代码开始重复调用 recv，因为它每调用一次都需要两个上下文切换，所以效率十分低。

另一方面，程序员不得不编写类似的代码——请参阅图 3.13 中的 readclf 函数。

然而，更为重要的是代码中没有检查缓冲区溢出。

为了弄清楚在一个更为合理的实现中也可能发生同样的错误，考虑一下下面的代码段：

```
static char *bp;
static int cnt = 0;
static char b[1500];
char c;
```

```

for ( ; ; )
{
    if ( cnt -- <= 0 )
    {
        cnt = recv (fd, b, sizeof( b ), 0);
        if ( cnt < 0 )
            return -1;
        if ( cnt == 0)
            return 0;
        bp = b;
    }
    c = *bp++;
    *bufptr++ = c;
    if (c == '\n' )
    {
        *bufptr = '\0';
        break;
    }
}

```

本段代码通过读取一大块数据到临时缓冲区然后在检查新行时把它们一个字节一个字节地移动到最终缓冲区中，这样避免了前面的代码段的低效率。注意，尽管因为 `bufptr` 指向的缓冲区没有做溢出检查，它和第一段代码一样会遭遇相同的错误：尽管不能编写类似的通用读行函数，但还是可以想像该段代码嵌入到大函数中时会发生错误。

现在请看看一个真实的具体实现（图 2.54）。

~~readline.c~~

```
1 int readline( SOCKET fd, char *bufptr, size_t len )
2 {
3     char *bufx = bufptr;
4     static char *bp;
5     static int cnt = 0;
6     static char b[ 1500 ];
7     char c;
8
9     while ( --len > 0 )
10    {
11         if ( (c = (char)recv( fd, b + cnt, 1, 0 )) == '\n' )
12             break;
13         if ( c == '\r' )
14             continue;
15         if ( c != '\n' )
16             ++cnt;
17         if ( cnt >= sizeof(b) - 1 )
18             break;
19     }
20
21     if ( cnt < len )
22         bufx[ len ] = '\0';
23     else
24         bufx[ len ] = '\n';
25
26     return cnt;
27 }
```

```

10     if ( --cnt <= 0 )
11     {
12         cnt = recv( fd, b, sizeof( b ), 0 );
13         if ( cnt < 0 )
14             return -1;
15         if ( cnt == 0 )
16             return 0;
17         bp = b;
18     }
19     c = *bp++;
20     *bufptr++ = c;
21     if ( c == '\n' )
22     {
23         *bufptr = '\0';
24         return bufptr - bufx;
25     }
26 }
27 set_errno( EMSGSIZE );
28 return -1;
29 }
```

readline.c

图 2.54 readline 函数的错误实现

乍一看，该函数的实现似乎很正确，程序传递缓冲区的大小给 `readline` 函数，外层循环确认程序没有超出缓冲区大小。如果超出了范围，程序设置 `errno` 为 `EMSGSIZE` 并返回-1。

为了弄明白该段代码的问题，假定以下面的方式调用它：

```
rc = readline( s, buffer, 10 );
```

程序从套接字中读取一行数据：

```
123456789<n1>
```

当加载新行字符到 `c` 时，`len` 就为零，说明这是可以接受的最后一个字节。第 20 行存储新行字符到缓冲区中，并向前推进 `bufptr` 指针到缓冲区尾。问题出现在第 23 行，在这一行上程序存储字节 0 到缓冲区外面。

注意在内层循环中存在着同样的问题。为了找出这个错误，假设一 `cnt` 等于 0，输入

readline 函数, recv 函数返回 1 个字节。下面将会发生什么呢? 我们应当把这种情况描述为缓冲区下溢错误。

本例子显示了即使是在我们认为已经检查了的时候缓冲区溢出错误也是很容易产生的。图 2.55 显示了 readline 的最后、正确版本。

readline.c

```

1 int readline( SOCKET fd, char *bufptr, size_t len )
2 {
3     char *bufx = bufptr;
4     static char *bp;
5     static int cnt = 0;
6     static char b[ 1500 ];
7     char c;
8
9     while ( --len > 0 )
10    {
11        if ( --cnt <= 0 )
12        {
13            cnt = recv( fd, b, sizeof( b ), 0 );
14            if ( cnt < 0 )
15            {
16                if ( errno == EINTR )
17                {
18                    len++; /* the while will decrement */
19                    continue;
20                }
21                return -1;
22            }
23            if ( cnt == 0 )
24                return 0;
25            bp = b;
26        }
27        c = *bp++;
28        *bufptr++ = c;
29        if ( c == '\n' )
30        {
31            *bufptr = '\0';
32        }
33    }
34    return len;
35 }
```

```

31         return bufptr - bufix;
32     }
33 }
34 set_errno( EMSGSIZE );
35 return -1;
36 }

```

readline.c

图 2.55 readline 的最后版本

本版本和上一个版本的唯一区别就是程序预先减少计数器 len 和 cnt，而且程序检查并在调用 recv 时忽略 EINTR 错误。通过预先减少 len，可以保证给字节 0 预留空间。同样地，通过预先减少 cnt，可以保证程序不会从空缓冲区中读取数据。

小结

程序必须准备接收来自用户和对等方的意外操作和数据。本节讨论了对等方不合理行为的两个实例。首先看到的例子是，程序不能完全依赖对等方告知什么时候结束数据发送，然后第二个例子说明了检查非法输入的重要性，最后还按照这个原则开发了一个健壮的 readline 函数。

技巧 12 不要认为成功的 LAN 策略

一定可以移植到 WAN 上

许多网络应用程序都是在 LAN 甚至是单机上开发并测试的。这种方法有简单、方便以及开销不大的优势，但是它也许会掩盖问题。

尽管技巧 7 中介绍了很多很容易发生的网络崩断，LAN 还是提供了一个很好的环境，IP 数据报在这个环境中几乎不丢失和延时，实际上也永远不会乱序到达。因为 LAN 提供了如此理想的环境，所以不能想当然地认为在 LAN 上成功运行的应用程序移植到 WAN 或 Internet 上也一定能运行得很好。移植之后，有可能发生两种类型的问题：

- (1) 由于 WAN 引起的额外延时，WAN 上的应用程序的性能可能不令人满意。
- (2) LAN 上成功运行的不正确代码可能在 WAN 上运行失败。

第一种类型的问题通常意味着应用程序的设计必须重新进行。

性能问题的例子

作为这种类型问题的例子，改变 `hb_server`（图 2.48）和 `hb_client`（图 2.47）程序中的 `T1` 为 2 秒，`T2` 为 1 秒（见图 2.46）。这个改变导致 `heartbeat` 消息每隔 2 秒发送一次，如果 3 秒中之内没有接收到应答，应用程序就退出。

首先，在 LAN 上运行这些程序。运行近 7 个小时之后，服务器 36 次报告丢失一个 heartbeat 消息，一次连续丢失 2 个 heartbeat 消息。客户端报告在所有的 12139 次发送消息中，只有 11 次要发送第二个 heartbeat 消息。在客户端手动中断之前，客户端和服务器运行正常。这些结果在 LAN 中司空见惯，除了偶尔的小延时之外，消息都很及时地到达。

下面，在 Internet 上运行这两个程序。12 分钟以后，客户端发送三个 heartbeat 消息都没有应答，因此不得不结束。下面给出部分客户端的输出，它能帮助我们看清问题：

```
sparc: $ hb_client 205.184.151.171 9000
hb_client: sending heartbeat #1
hb_client: sending heartbeat #2
hb_client: sending heartbeat #3
hb_client: sending heartbeat #1
hb_client: sending heartbeat #2
hb_client: sending heartbeat #1
hb_client: sending heartbeat #1
hb_client: sending heartbeat #1
hb_client: sending heartbeat #2
hb_client: sending heartbeat #1
hb_client: sending heartbeat #2
hb_client: sending heartbeat #3
hb_client: connection dead
Terminates 1 second after last heartbeat
sparc: $
```

这次，客户端 251 次发送第一个 heartbeat 消息，但是不得不 247 次发送第二遍。也就是说客户端几乎从来没有及时收到第一个 heartbeat 消息的回复。客户端需要 10 次发送第三个 heartbeat 消息。

服务器端也显示了较大的性能下降。服务器 247 次在等待第一个 heartbeat 消息时超时，5 次等待第二个消息时超时，一次第三个消息超时。

这个例子说明了 LAN 上运行良好的应用程序可能在 WAN 上有显著的性能问题。

» 屏蔽错误的例子

作为第二种类型问题的例子，考虑一个基于 TCP 的遥测应用程序，服务器每隔一秒从远程感应设备接收测量数据包。这些数据包包含两或三个整型值。服务器的不成熟的实现可能有如下的事件循环：

```
int pkt [ 3 ]
for ( ; ; )
{
    rc = recv ( s, ( char* ) pkt, sizeof ( pkt ), 0 );
    if ( rc != sizeof ( int ) * 2 && rc != sizeof ( int ) * 3 )
        /* log error and quit */
    else
        /* process rc / siezeof ( int ) values */
}
```

根据技巧 6 中的知识，可以知道这段代码是不正确的，但是在此还是简单模拟一下。编写一个服务器程序，如图 2.56 所示，它实现了如前所述的循环。

telemetry.c

```
1 #include "etcp.h"
2 #define TWOINTS      ( sizeof( int ) * 2 )
3 #define THREEINTS( sizeof( int ) * 3 )
4 int main( int argc, char **argv )
5 {
6     SOCKET s;
7     SOCKET s1;
8     int rc;
9     int i = 1;
10    int pkt[ 3 ];

11    INIT();
12    s = tcp_server( NULL, argv[ 1 ] );
13    s1 = accept( s, NULL, NULL );
14    if ( !isValidsock( s1 ) )
15        error( 1, errno, "accept failure" );
16    for ( ; ; )
```

```

17  {
18      rc = recv( s1, ( char * )pkt, sizeof( pkt ), 0 );
19      if ( rc != TWOINTS && rc != THREEINTS )
20          error( 1, 0, "recv returned %d\n", rc );
21      printf( "Packet %d has %d values in %d bytes\n",
22              i++, ntohl( pkt[ 0 ] ), rc );
23  }
24 }
```

telemetrys.c

图 2.56 遥测服务器模拟程序

11~15 这些代码实现了标准初始化并接收数据序列。

16~23 该循环重复地从客户端接收数据。如果每次读操作没有获得完整的 sizeof(int) * 2 或 sizeof(int) * 3 个字节，程序就记录错误并退出。否则，程序转换第一个整数为主机字节顺序（技巧 28）并向标准输出打印这些数据以及接收到数据的数量。后面将会（图 2.57）看到客户端把它要发送的值的数量放在第一个整数里。作这些说明的目的是为了帮助我们明白模拟中发生的事情，而不是技巧 6 里讨论的“消息大小”数据包头。

为了测试这个服务器，下面编写一个每隔一秒发送一个整数数据包的客户端，模拟远程感应设备。客户端程序如图 2.57 所示。

telemetryc.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     int rc;
6     int i;
7     int pkt[ 3 ];
8
8     INIT();
9     s = tcp_client( argv[ 1 ], argv[ 2 ] );
10    for ( i = 2;; i = 5 - i )
11    {
12        pkt[ 0 ] = htonl( i );
13        rc = send( s, ( char * )pkt, i * sizeof( int ), 0 );
14        if ( rc < 0 )
15            error( 1, errno, "send failure" );
```

```

16     sleep( 1 );
17 }
18 }
```

telemetry.c

图 2.57 遥测客户端模拟程序

8-9 程序初始化并连接到服务器。

10~17 程序每隔一秒发送包含两个或三个整数的数据包给服务器。正如前面讨论的那样，程序把发送整数的个数（转换为网络字节顺序）作为数据包里的第一个值。

为了运行模拟程序，在 bsd 上启动服务器，而在 sparc 上启动客户端。服务器的输出如下所示：

```

bsd: $ telemetrys 9000
Packet 1 has 2 values in 8 bytes
Packet 2 has 3 values in 12 bytes
                                         Many lines deleted
Packet 22104 has 3 values in 12 bytes
Packet 22105 has 2 values in 8 bytes
                                         Client terminated after
                                         6 hrs., 8 mins., 25 secs.
telemetrys: recv returned 0
bsd: $
```

虽然服务器有一个明显的错误，但它可以在 LAN 上连续运行 6 个多小时，只有手动停止客户端才终止模拟。

模拟程序的输出结果使用 awk 脚本来检查每个读操作是否都返回正确的字节数。

然而，当这个服务器运行在 Internet 上时，结果就完全不一样了。再次在 sparc 上运行客户端，而在 bsd 上运行服务器，但是通过指定客户端的地址为连接到 Internet 的一个地址，数据传输都通过 Internet 路由。可以从服务器打印出来的最后几行看出，15 分钟后服务器发生一个致命错误。

```

Packet 893 has 2 values in 8 bytes
Packet 894 has 3 values in 12 bytes
Packet 895 has 2 values in 12 bytes
Packet 896 has -268436204 values in 8 bytes
Packet 897 has 2 values in 12 bytes
```

```

Packet 898 has -268436204 values in 8 bytes
Packet 899 has 2 values in 12 bytes
Packet 900 has -26846204 values in 12 bytes
Telemetrys: recv returned 4
bsd: $

```

服务器处理第 895 个数据包时发生错误，程序应当在这里读取 8 个字节，但是实际上读了 12 个字节。图 2.58 显示了所发生的问题。

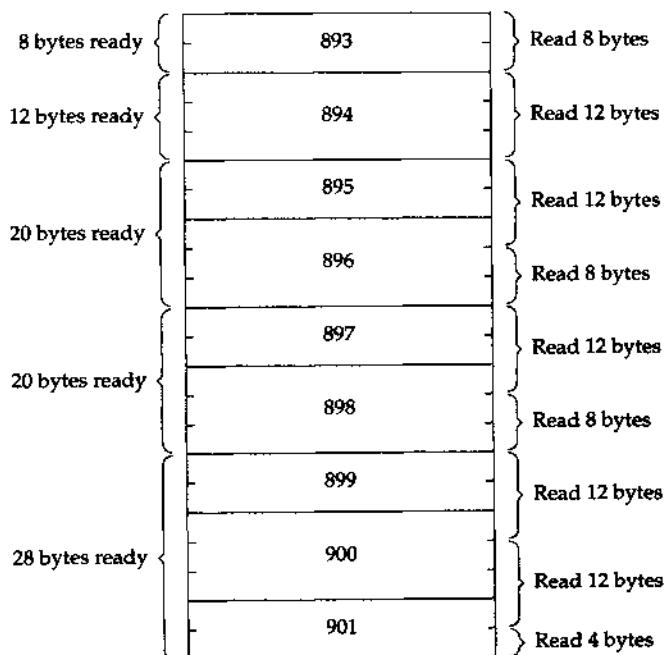


图 2.58 致命错误

左边的数字显示了服务器 TCP 可以读取的字节数。右边的数字显示了实际读取的字节数。从中可以看到第 893 和 894 个数据包如期传送和读取了。当 telemetrys 调用 recv 读取第 895 个数据包时，可以读取 20 个字节。

tcpdump（请参阅技巧 34）进行的网络跟踪显示双方主机的 TCP 段在这个时间上同时丢失。这可能是因为网络拥挤导致中间路由器丢弃数据包。在第 895 个数据包可以传递之前，telemetrys 已经准备好第 896 个数据包，它们是同时传递的。

第 895 个数据包有 8 个字节，但是因为第 896 个数据包同样可以读取，所以服务器读取第 895 个数据包和第 896 个数据包中的第一个整数。这是 telemetrys 显示读取了 12 个字节而第 895 个数据包仅有两个整数的原因。下一个读操作返回第 896 个数据包中的

后两个整数，因为 `telemetryc` 没有初始化第二个值，所以 `telemetrys` 输出数据值个数错误。

如图 2.58 所示，同样的情况再次发生在第 897 和 898 个数据包上，然后下一个读操作可以读取 28 个字节。现在 `telemetrys` 依次读取第 899 个数据包，第 900 个数据包的第一个值，第 900 个数据包的区域部分第 901 个数据包的第一个值，以及第 901 个数据包里的最后一个值。最后一个读操作仅返回 4 个字节，这导致测试在 `telemetrys` 的第 19 行失败，结束模拟。

不幸的是，该模拟在前面发生了更坏的事情：

```
Packet 31 has 2 values in 8 bytes
Packet 32 has 3 values in 12 bytes
Packet 33 has 2 values in 12 bytes
Packet 34 has -268436204 vlaues in 8 bytes
Packet 35 has 2 values in 8 bytees
Packet 36 has 3 values in 12 bytes
```

模拟仅进行了 33 秒就发生了一个没有检测出来的错误。如图 2.59 显示的那样，当 `telemetrys` 从第 33 个数据包里读取数据时，可以获得 20 个字节，因此读取的是 12 个字节而不是 8 个。这意味着只有两个值的数据包被误解为有三个数据，然后有三个值的数据包被误解为只有两个值。注意服务器接收第 35 个包时，`telemetrys` 重新同步，因此错误没有检测出来。

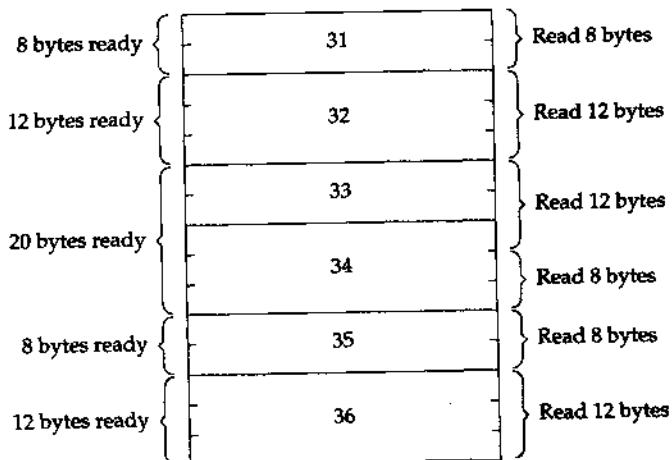


图 2.59 一个没有检测出来的错误

小结

LAN 提供的接近理想的环境可能会掩盖网络应用程序的很多性能问题，它甚至掩盖了某些错误。我们永远也不能假定 LAN 上成功运行的应用程序在 WAN 上一样也会成功运行。

WAN 环境中增加的时间延时可能导致在 LAN 上有不错性能的应用程序在 WAN 上表现出不那么令人满意的性能。由于这个原因，有时需要应用程序改变设计。

在繁忙的 WAN 上特别是 Internet 上的拥塞，可能导致数据以不可预测的质量和不可预测的次数传递。这就需要程序员特别注意不要做出在任意给定的时间内有多少数据到达以及数据是以何种频率到达的任何假设。

本节中我们集中讨论了 TCP 应用程序，虽然应用这些到 UDP 上甚至可以获得更好的效果，但是 UDP 在减轻 WAN 环境的苛刻性方面没有可靠性优势。

技巧 13 学习协议是如何工作的

Stevens 在[Stevens 1998]中评述了大多数的网络编程问题是和编程以及 API 无关的，而是由于对下层网络协议的不正确理解造成的。这个观点被网络新闻组经常问到的问题所证实（请参阅技巧 44）。例如，任何人都可以通过参考 UNIX 或 Windows 机器上的在线文档来找出如何禁用 Nagle 算法（技巧 24），但是只有已经理解了 TCP 的流量控制和 Nagle 算法在流量控制中扮演的角色，程序员才知道什么时候禁用以及什么时候不禁用。

同样地，技巧 10 里讨论的 TCP 缺乏对网络中断的即时通知，直到理解了不使用通知的原因才会认为这种做法不是荒谬的。一旦我们确实理解了这些原因，在应用程序里以一定的时间间隔发送 heartbeat 探询消息就会变得容易理解了。

获得对协议的理解有几个方法，这些方法很多将在第 4 章中讨论，但是在这一章所涉及的部分方法也是很有意义的。关于 TCP/IP 协议组的主要信息源是 RFC。尽管 RFC 以不同程度的严肃性覆盖了大量的主题，但所有的 TCP/IP 协议组规范都是以 RFC 的形式出版的。RFC 文档是 TCP/IP 协议组应当如何工作的官方的权威文档。RFC 以及它们的索引可以在 RFC 编辑的主页上获取：

[<http://www.rfc-editor.org>](http://www.rfc-editor.org)

请参阅技巧 43，了解获得 RFC 拷贝的其他方法。

因为 RFC 是许多作者共同劳动的成果，所以在清楚性方面差异很大。另外，一些主题在一系列的 RFC 中讨论，有时获得完整的描述是很困难的。

幸运的是，初学者可以使用的有关协议的信息可以从其他途径获得。下面将给出两个这样的途径，其他途径将在第四章中给出。

[Comer 1995]以 RFC 的角度讨论了 TCP/IP 协议组。也就是说，Comer 给出了主要的协议并讲述了这些协议应当如何以 RFC 指定的方式工作。因此，有些人把他的书描述为采用理论的方法来接近 TCP/IP 协议，和 Stevens[Stevens 1994, Stevens 1996]采用的实际方法相反。

Comer 的书是一本优秀的、写得十分清晰的有关 TCP/IP 协议的介绍，其中包括了经常使用的 RFC 线索。这些线索便利了进一步的学习，给出了 RFC 有条理的看法。

[Stevens 1994, Stevens 1996]从具体实现的角度讨论了 TCP/IP 协议组。也就是说，该书讲述了主要的 TCP/IP 具体实现是如何在实际中工作的。通过使用 tcpdump（请参阅技巧 34）捕捉实时协议数据以及使用 tcpdump 和从 tcpdump 所得的数据画出事件时间顺序表（如图 2.32 所示）来查看正在运转的协议，可以了解 TCP/IP 的工作情况。这种方法，加上详细的数据包格式图表以及设计为揭示所讨论协议的某些方面的小应用程序，使协议以活动的方式显示出来，这种效果是单纯的描述不能达到的。

虽然这些书从不同的侧面讨论 TCP/IP 协议，但是我们不能认定其中一种方法比另一种方法好以及其中一本书比另一本书好。某一时间应当使用什么书依赖于当时的需要，实际上，这些书互为补充，任何认真的网络程序员应当每样都有一本。

» 小结

本节讨论了理解协议是如何工作的重要性，其中提到了 RFC 包含了 TCP/IP 的官方规范，并建议使用 Comer 和 Stevens 的著作进一步研究协议以及协议的功能。

技巧 14 不要把 OSI 七层参考模型看得太重要了

因为设计和实现网络协议是一项任务繁重而且十分困难的工作，所以通常把它们简化并把它们划分为较小的片段。通常的方法是把工作划分成多层，每层都为上面各层提供服务，并使用下面各层提供的服务。

例如，图 2.1 是简化的 TCP/IP 协议栈，从中可以看出，IP 层提供了一个称为数据报传递的服务给 TCP 和 UDP 层。为了提供该项服务，IP 利用接口层的服务，把数据报传递给物理介质。

» OSI 模型

网络分层的著名例子可能是国际标准化组织（ISO）的开放系统互联参考模型。

很多人误认为是 OSI 模型首先提出分层、虚拟以及其他基本网络概念的。实际上，在 OSI 模型出笼之前，这些思想在早期的开发 TCP/IP 协议组的 ARPANET 研究者中已经广泛熟悉和使用。

请参阅 RFC 871，了解有关这方面的有趣的介绍。



因为该模型有 7 层，如图 2.60 所示，通常被人们称为 OSI 7 层模型。

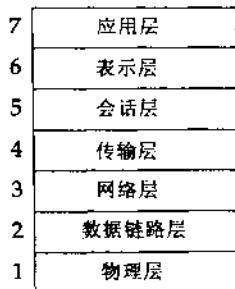


图 2.60 OSI 7 层参考模型

正如我们前面提到的那样，层次 N 为层次 N+1 服务，并使用 N-1 层的服务。另外，每一层只能和它最接近的邻居通信。这种通信是通过预先定义好了的相邻层次间的接口进行的，因此原则上一个层次可以被提供相同服务的其他层次代替而根本不会影响其他各层。通信栈中的对应层通过协议交换数据（经过网络）。

因为 OSI 中的各层经常在网络书籍中提到，所以在这里提供一个简短的有关各层提供的服务的描述是值得的。

(1) 物理层——该层和硬件打交道。它指定了接口的电气和时间特性、位流该怎样放在介质上、物理组帧以及连接头的大小和形状。

(2) 数据链路层——数据链路层是物理层的软件接口。它的工作是为网络层提供“可靠的线路”。它包含了网络层用来和物理设备会话的“设备驱动程序”，同时它也处理诸如链路数据分帧、为检测数据讹误的校验以及调节对下面的物理介质的访问等事情。网络层和数据链路层之间的接口通常使用允许设备独立的机制。

(3) 网络层——网络层负责在接点之间传递数据包。它负责诸如寻址和路由计算、分组和重组以及拥塞和流量控制等事情。

(4) 传输层——传输层处理可靠的双方终端对终端的通信。它通过对由讹误、丢失数据包和乱序包的处理来补偿下面各层的不可靠性。它也可以提供流量控制以及防止拥塞。

(5) 会话层——传输层提供给通信双方一个全双工的可靠的通信流。会话层建立在传输层之上，给应用程序提供的额外服务，包括会话的建立和释放（例如登录、退出登录）、对话控制（例如模拟半双工终端）、同步（例如为大文件的传输设立检查点）以及给简单可靠的第 4 层数据流增加结构。

(6) 表示层——表示层处理数据翻译，该层处理数据表示之间的会话（例如，ASCII 到 EBCDIC 之间的转化）和压缩。

(7) 应用层——应用层包括使用其他六层传输数据的用户应用程序。TCP/IP 世界里熟悉的例子是 telnet、ftp、邮件客户端和 Web 浏览器。

OSI 7 层模型的官方描述在[International Standards Organization 1984]中给出，但是它仅仅是笼统地描述了各层应当做的事情。各层提供的服务以及使用协议的详细描述在其他的

ISO 文档中给出。该模型和它的各层相对详细的解释以及有关 ISO 文档的指示可以在[Jain and Agrawala 1993]中找到。

虽然 OSI 模型作为讨论网络体系结构和具体实现的框架是很有用的，但是它不能作为实现网络体系结构的蓝本。而且也不能认为该模型在第 N 层放置了某些功能，就意味着第 N 层是唯一的或者是最好的放置这个功能的层次。实际上，人们对该模型持有的一个批评就是层次里功能的安排充满着随意性并总是不很明显。

事实是 OSI 模型有很多缺陷，虽然最后产生了一些具体实现，但是 OSI 协议实质上目前已经消亡了。实际上，该模型的一个问题就是它在设计（该协议是由一个委员会设计的）时没有考虑到具体实现的利益。而 TCP/IP 协议组的开发是基于实验性质的具体实现并且充分考虑了具体实现的利益。

OSI 模型的另一个问题就是它太复杂了并且先天效率不高。一些功能在几个层次中都有。例如，大多数层次都有错误检测和校正功能。

正如 Tanenbaum 指出的那样[Tanenbaum 1996]，该模型的一个最大的缺陷就是遭受着“communications mentality”的局限性。该局限性是反应在术语以及层次之间使用的接口原语的规范上的，OSI 模型使用的术语和一般网络术语不一样，使用的原语更适合于电话系统而不是计算机。

» TCP/IP 模型

下面用 TCP/IP 模型来跟 OSI 模型做比较。理解 TCP/IP 模型仅仅说明了 TCP/IP 协议组的设计，而且它不是以 OSI 模型的方式作为参考模型来服务的，这是很重要的。因为这个原因，人们看不到 TCP/IP 是作为新的网络协议的基础。尽管如此，用它跟 OSI 模型做比较来看看 TCP/IP 的层次是如何映射到 OSI 模型上去的还是十分有益的。至少，它可以提醒我们 OSI 模型并不是“唯一正确的道路”。

正如图 2.61 里看到的那样，TCP/IP 协议栈由四层组成。应用层完成 OSI 模型中应用层、

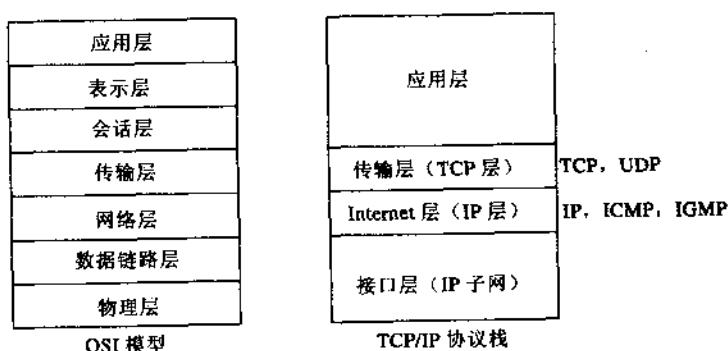


图 2.61 OSI 模型和 TCP/IP 协议栈的比较

表示层和会话层的所有功能。传输层和 OSI 模型中对应的传输层功能一样，都是处理终端到终端的通信。传输层定义了 TCP 和 UDP 协议。Internet 层定义了 IP、ICMP 和 IGMP 协议，它对应的是 OSI 模型中的网络层。

我们已经对 IP 很熟悉了。ICMP 是 Internet Control Message Protocol 的缩写，它用于系统间的通信控制和错误通知。例如，“主机不可到达”消息是以 ICMP 消息的方式传送的，和 echo 请求及 ping 程序使用的返回消息是一样的。IGMP 是 Internet Group Management Protocol 的缩写，主机用它保持广播路由器通知本地广播组成员。虽然 ICMP 和 IGMP 消息传递 IP 数据报，但是它们都看成是 IP 的一部分而不是高层次的协议。

最后，接口层处理计算机和下面的网络硬件之间的接口。可以把它看成对应于 OSI 模型中的数据链路层和物理层。在 TCP/IP 体系结构中，除了简单地说名它以系统相关的方式处理网络硬件的接口，结构层其实没有做更多的描述。

在离开 TCP/IP 协议栈中的各层之前，先分析一下终端对终端网络中对应层之间的通信。图 2.62 显示两个 TCP/IP 协议栈通过几个中间路由器通信。

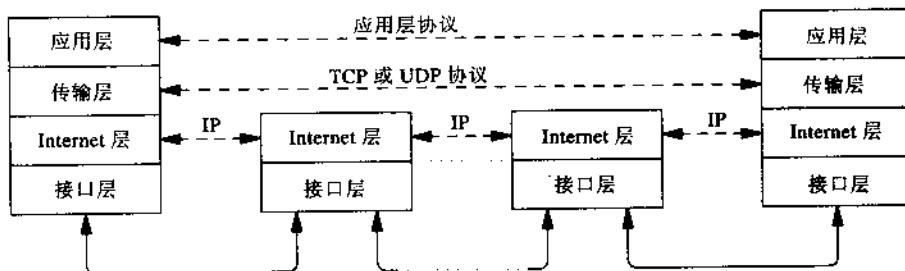


图 2.62 对等层次通信

虽然我们已经知道了一个应用程序中的数据在堆栈中沿着堆栈下行，通过网络，并且沿着对等方的协议栈上行到达对等方应用程序，堆栈中的每一层看起来好像和对等方相应层次直接进行会话。例如，如果应用程序是 FTP，FTP 客户端直接和 FTP 服务器对话，而不必担心 TCP、IP 和所涉及的物理网络究竟干了些什么。

其他各层也是一样的。例如，如果传输层是 TCP 协议，它就和对等方的 TCP 进行会话，而不必担心会话使用的是什么协议或网络。理想的情况是当第 N 层发送一个消息时，对等方的第 N 层接收完全一样的消息，低层次对数据进行的任何操作在消息到达对等方对应层次前都解开了。

最后一点需要再做一点解释。看看图 2.62，就可以发现传输层时最低的终端对终端的层次，它是两个终点之间直接进行会话的最低层次。另一方面，Internet 层和“下一个节点”上的路由器或系统进行会话。

在本讨论中，假设中间有路由器，因此对于终点之一的系统来说，“下一个节点”并不是最终的目的主机。

因为中间系统可能会改变某些域，如 IP 报头的生存时间（TTL）域，所以目的终端点的 Internet 层看到的消息和源终端点 Internet 层发送的消息不一样。

这个事实加重了 Internet 层和传输层之间的区别。Internet 层考虑的是发送消息给下一个节点，而且它是和下一个节点的 Internet 层而不是终端点的 Internet 层进行会话的。然而传输层却是直接和终端点的传输层进行会话，而不会注意到中间系统的存在。

小结

本节对比了 OSI 参考模型和 TCP/IP 模型。从中了解了 OSI 7 层模型是一个有用的描述性工具，但是基于它的具体实现却很少成功。

第三章

创建高效和健壮的网络程序

技巧 15 理解 TCP 写操作

本节以及下节讨论 TCP/IP 编程的写操作和读操作的某些方面。讨论的焦点不是特定的 API，也不是系统调用的细节本身，而是关系到写操作和读操作的某些语意问题。

技巧 6 中已经讲述了写操作调用和 TCP 发送的段之间并没有一一对应的关系。本节将进一步讨论这个主题。写操作调用和 TCP/IP 之间的交互是系统相关的，通过考虑特定的具体实现可以得出协议规范是很具体的结论。因此，为了能具体地进行讨论，在此描述传统的 BSD 具体实现。该具体实现被广泛使用，而且该实现的源代码可以很容易地获得。

原始的 4.4 BSD-lite2 具体实现的源代码可以在 Walnut Creek CDROM (<<http://www.cdrom.com>>) 上获得。[Wright and Stevens 1995] 中不仅给出了源代码，还有丰富的注释。

从应用程序看写操作

当用户程序在 TCP 连接上调用一个写操作时，首先发生的事情是数据从用户的缓冲区拷贝到内核。从这一点来说，所发生的事情依赖于连接所处的状态。TCP 可能决定发送所有数据、或者部分数据，或者什么数据也不发送。后面还将会对 TCP 的这个决定做进一步的讨论，但是首先看看从用户应用程序的观点看到的写操作。

程序员很容易认为当写 n 个字节返回值 n 时，这 n 个字节实际上已经发送到对等方，甚至是被确认了。不幸的是，这是不正确的看法。TCP 尽可能多地发送它能发送的数据（有可能什么也不发送），然后立刻返回值 n 。应用程序不能断定究竟发送了多少数据，也不能断定对等方是否确认发送的数据。

通常，除非 TCP 发送缓冲区已满，否则写操作是不会阻塞的。这意味着写操作几乎总是立即返回，而且如果它们返回了，也不能保证所写数据的处置。正如技巧 9 里讨论的那样，这里隐含着可靠数据的传输。

从应用程序方面来看，数据已经写入 TCP 了，因此 TCP 的“保证递交”保证了它会到达对等方。实际上，当写操作返回时，因为写入的部分或全部数据可能仍然在排队等待传输，所以如果对等方主机或应用程序这时崩溃的话，数据将会丢失。

如果发送方应用程序崩溃的话，TCP 就会继续设法传送数据给对等方。

程序员应当注意的另一个重要的东西是对调用写操作时发生错误的处理。如果程序往磁盘写数据时，`write` 无错返回，那么程序知道该写操作成功了，程序也不再需要考虑所写数据的处理了。

当然，事实并非如此。通常，数据放在内核缓冲区中等待写入到磁盘中，如果在数据写之前系统崩溃的话，数据就很有可能丢失。然而，问题的关键是一旦 `write` 返回，它就是程序获得的关于是否发生问题的最后指示。程序员通常易于认为数据在写入之前丢失是一个不可避免的但又不太可能的事件，如磁盘本身崩溃引起的。

然而，对于 TCP 来说，通常会返回给写操作一个错误。因为写操作可能在数据实际传送之前返回，所以错误经常是通过下一个操作返回，如技巧 9 里讨论的那样。因为下一个操作通常是读操作，所以可以这样说写操作错误通过读操作返回。写操作返回的错误仅仅是那些在调用写操作时就已经发生的错误，这些错误包括：

- 套接字描述符非法。
- 没有指向套接字的文件描述符（在调用 `send` 及其相应函数的情况下）。
- 调用里指定的套接字不存在或套接字没有连接。
- 缓冲区地址参数指定了一个非法的地址。

这些错误大多数是由不正确的代码引起的，在程序的开发阶段之后就很少遇到这些问题了。唯一的例外是 EPIPE 错误（或 SIGPIPE 信号），该错误在对等方重置连接时发生。发生这种错误的条件在技巧 9 里讨论对等方应用程序崩溃时已经讨论了。

从以上的各种考虑中可以得出结论，当在一个 TCP 连接上应用写操作时，该写操作最好理解为拷贝数据到发送队列中并通知 TCP 队列中有了新数据。下面继续讨论 TCP 在接收到通知后会做些什么事情，但是应当注意的是 TCP 所做的事情和一些操作本身是异步的。

从 TCP 看写操作

正如前面讨论的那样，写操作负责从应用程序的写缓冲区中移动数据到内核缓冲区中，并且负责通知 TCP 应用程序新数据已经到达，要求 TCP 处理。现在来看看 TCP 用来决定它是否可以立即发送新到达数据以及可以发送多少的条件。我们的目的不是提供 TCP 发送逻辑如何工作的完整解释，而是评价影响该逻辑的一些因素。通过这种方法可以更好地理解应用程序所做的操作。

TCP 发送机制的一个基本目标是尽可能高效地利用可获得的带宽。为了达到这个目的，TCP 选择以 MMS 大小发送数据块。

在连接建立期间，TCP 连接的每一方都可以指定可以接收的 MMS 大小。对等方不能通过发送大于 MMS 大小的段来遵循该约定。MMS 派生于 MTU，如技巧 7 里讨论的那样。

同时，发送的数据也不能超出对等方的缓冲区。正如技巧 1 里看到的那样，它由 TCP 的发送窗口来控制。

如果仅仅需要考虑以上问题，那么发送机制就很简单：以 MMS 大小立即发送 TCP 发送窗口允许发送的最大的可获得的段。不幸的是，还有其他问题需要考虑。

首先需要考虑的是特别重要的拥塞控制问题。如果 TCP 将要突然地输入大量的段到网络中，路由器缓冲区空间可能会耗尽，导致路由器丢弃数据报。而这又会导致重传，进一步使网络拥挤。在极端的情况下，网络拥挤会导致很少甚至没有数据报被传递，这时的网络称作拥塞崩溃。为了预防拥塞，TCP 不能在一个空闲连接上突然发送几个段。TCP 应当一开始时发送一个段，然后增加网络中没有确认的段的数量，直到达到一个稳定的状态。

这个问题可以形象地想像为房间里有几个人，其中一个大喊：“着火了！”所有的人都同时冲向门口，拥挤导致很少甚至没有人出去。相反，如果人们以排成一队通过门口，就不会有拥挤，所有的人都可以出去。

TCP 使用两个算法来预防拥塞。两种方法都使用额外的窗口，称作拥塞窗口，来控制拥塞。任何时候 TCP 发送数据的最大数量是发送窗口和拥塞窗口的最小值。注意这两个窗口负责流量控制的不同方面。发送窗口由对等方 TCP 使用，以预防应用程序发送的数据超过对等方 TCP 的缓冲区。拥塞窗口由本方 TCP 使用，以预防应用程序超过网络所能承受的能力。通过限制传输的数据为两个窗口的最小值，可以保证遵循两种类型的流量控制。

第一个流量控制算法，称作慢启动，“缓慢地”增加输入到网络中段的频率，直到达到阀值。

“缓慢地”是比喻，因为增长实际上是指数级别的。在慢启动期间，每一个段的 ACK 消息都会打开拥塞窗口。开始是 1 个段，然后以拥塞窗口大小依次产生 1、2、4、8、…个段。

当拥塞窗口大小达到阀值，称作慢启动阀值（slow start threshold），慢启动过程就结束了，另一个拥塞预防算法接管过来。在拥塞预防阶段，连接假定为到达了稳定的状态，同时 TCP 还连续地探询网络有没有可获得的额外带宽。在拥塞预防阶段，拥塞窗口以线性的方式打开——每个来回至多一个段。

对于 TCP 发送机制来说，拥塞窗口可以潜在地预防本来有可能发送出去的数据。如果发生了拥塞（通过丢失段可以知道），或者如果网络已经空闲了一段时间，拥塞窗口就会减小，也许减小到只有 1 个段。根据队列中排队数据的多少以及应用程序试图发送数据的多少，这种机制可以避免传输部分或全部的数据。

有关拥塞预防算法的权威性的信息源可以在 [Jacobson 1998] 中得到，该论文首次提出拥塞预防算法。该论文可读性很强，还给出了使用该算法获得显著网络性能提高的实验的结果。[Stevens 1994] 中有该算法的详细解释，以及使用该算法时 LAN 跟踪的输出。今天，这

些算法成了兼容 TCP 具体实现的必备部分 (RFC 1222; [Braden 1989])。

尽管这些算法功能很强大，它们的具体实现其实是很简单的——两个状态变量加上几行代码。

详细信息可以参阅[Wright and Stevens 1995]。

影响 TCP 发送机制的另一个因素是 Nagle 算法。该算法是在 RFC 896 [Nagle 1984] 中首先给出的，指出在任何给定的时间不能出现多于一个没有确认的“小段 (Small Segment)”，“小段”的意思是大小小于 MSS 的段。Nagle 算法的目的就是避免 TCP 发送一系列的小段给网络造成数据泛滥。TCP 在接收前一个小段的 ACK 消息之前，一直保存数量很小的数据。正如将要在技巧 24 里看到的那样，如果不使用 Nagle 算法，就会对应用程序的性能有很大的影响。

如果应用程序以小数据块的方式写数据，Nagle 算法的效果就是明显的。例如，假设有一个空闲的连接，它的发送和拥塞窗口都很大，而且程序执行两个连续的写操作。因为窗口允许它发送而且 Nagle 算法因没有未确认的数据（连接空闲）也不会反对发送数据，所以第一个写操作写入的数据立即被发送出去。然而，当第二个写操作的数据到达 TCP 时，即使这时发送和拥塞窗口还有空间，它也不会被发送，这是因为已经有一个未确认的小段，所以 Nagle 算法要求数据排队等待另一个段的 ACK 消息的到达。

Nagle 算法通常在实现时要求做到，如果存在任何未确认的数据就不能发送小段。这是 RFC 1122 建议的过程。和其他具体实现一样，BSD 派生的具体实现对这个原则进行了折中，允许在空闲连接上发送大的写操作遗留下来的小段。例如，如果空闲连接的 MSS 是 1460 字节，应用程序写了 1600 字节，那么——发送和拥塞窗口允许——TCP 将连续发送一个大小为 1460 字节的段和一个 140 字节的段而不需要等待任何 ACK 消息。按照 Nagle 算法的严格要求，TCP 需要保存该小段直到前一个 1460 的段被确认或者应用程序写入了足够的新数据来填充完整大小的段。

Nagle 算法实际上是一个称作 silly window syndrome (SWS) 预防的算法的半集。SWS 预防发送少量的数据。SWS 以及它是如何显著地降低性能的讨论在 RFC 813[Clark 1982] 中给出。正如我们已经看到的那样，Nagle 算法是 SWS 预防在发送方的表现形式。SWS 预防也需要接收方的配合，也就是说接收方不要通知小窗口。

发送窗口是对等方可获得的缓冲区空间数量的估计。对等方 TCP 在它发送的每一个段中包含一个 window update 值来通知目前可获得的缓冲区空间的数量。SWS 预防的接收方不通知缓冲区空间的很小增长。

举一个例子可以使这个问题更加清楚。假设对等方有 14600 字节的缓冲区空间可获得，而 MSS 的大小为 1460 字节。进一步假设对等方应用程序十分缓慢，每次只能读取 100 字节。在应用程序发送 10 个段给对等方后，发送窗口满了，应用程序不得不停止发送数据。现在对等方应用程序读取了 100 个字节，所以接收缓冲区中有 100 个字节的额外空间。因为 TCP 在一定的时间内不能发送小段的话就会不顾 Nagle 算法，所以如果对等方通知了这可获得的 100 字节，应用程序就会发送一个 100 字节的小段。而且，显而易见应用程序会一直发送 100 字节的数据包，这是因为对等方每次都读取 100 个字节，每次都会发布可获

得 100 字节缓冲区空间的 update window 通知。

除非缓冲区空间有了“显著的增长”，否则 SWS 预防的接收方是不会发送 window update 消息的。RFC 1122 定义“显著”为完全大小的段或增长到大于最大窗口的一半大小。BSD 派生的具体实现要求增长为两个完全大小的段或一个半最大窗口的大小。

接收方 SWS 预防看起来似乎没有必要（因为发送方 SWS 预防了小段的传输），但是它为没有实现 Nagle 算法的 TCP/IP 栈和禁用了 Nagle 算法的连接提供了保护措施（请参阅技巧 24）。RFC 1122 要求兼容的 TCP 具体实现双方都要实现 SWS 预防。

有了所有这些信息，现在可以概括一下 BSD 派生的 TCP 的发送机制了。其他具体实现的机制在细节上可能有些不同，但是多数东西是相同的。

任何时候调用 TCP 输出过程，它就会以发送缓冲区、发送窗口大小、拥塞窗口大小以及 MSS 的最小值来计算可以发送数据的数量。如果满足以下条件，数据就会发送：

- (1) 应用程序发送的是完全 MSS 大小的段。
- (2) 连接是空闲的，而且可以清空发送缓冲区。
- (3) Nagle 算法被禁用了，而且可以清空发送缓冲区。
- (4) 有紧急的数据要发送。
- (5) 应用程序有小段要发送，该段已经“有一段时间”不能发送了。

当然 TCP 有一个不能发送的小段时，它就启动一个计时器，等待和重传前等待 ACK 消息的到来前相同的时间量（该值受到种种限制的影响，一般为 5 到 60 秒）。也就是说，计时器设置为 RTO。如果该计时器，称作持续计时器，终止了的话，那么 TCP 就只会受限于发送窗口和拥塞窗口来传输段。甚至在对等方只通知了可获得一个 0 字节的窗口时，TCP 仍然试图发送 1 个字节的数据。这是为了防止丢失的 window update 消息导致死锁。

- (6) 对等方的接收窗口至少一半是打开的。
- (7) TCP 需要重传段。
- (8) TCP 需要为对等方数据发送 ACK 消息。
- (9) TCP 需要发布一个 window update 消息。

小结

本节详细地讨论了 TCP 的一些操作。从中可以看到，以应用程序的观点来看，理解写操作的最好方法是把它看做是一个从用户空间拷贝数据到操作系统内核发送空间然后返回的操作，TCP 什么时候实际传输该数据以及当时能传输多少数据依赖于连接的状态，通常情况下，应用程序是不可预测这些东西的。

本节还讨论了 BSD TCP 发送机制，从中了解了对等方可获得的缓冲区空间的大小（由发送窗口指示）、网络拥塞的评价（由拥塞窗口指示）、可发送的数据的数量、SWS 预防以及在决定数据是否可以发送时 TCP 重传策略是如何共同发挥作用的。

技巧 16 理解 TCP 顺序释放操作

正如我们已经看到的那样，一个 TCP 连接有三个阶段：

- (1) 连接建立阶段。
- (2) 数据传输阶段。
- (3) 连接撤消阶段。

本节主要介绍数据传输和连接撤消阶段之间的过渡。特别是，本节的主要兴趣在于了解对等方是如何完成数据传输阶段并准备撤消连接的，以及应用程序如何才能和对等方进行类似的通信。

正如后面将要看到的那样，通常情况下一方已经结束发送数据并通知对等方结束发送数据，但是仍然接收数据。这种情况是很有可能的，这是因为 TCP 连接是全双工的，一个方向上数据的流动和另一个方向的数据流动是不相关的。

例如，客户端连接到服务器，进行一系列的请求，然后关闭连接中属于它的一半，通知服务器它已经结束请求了。服务器可能不得不做大量的处理，甚至是联系其他的服务器来回答客户端，因此尽管客户端已经停止发送数据给它，它可能仍然有数据发给客户端。而且，服务器可能响应任意数量的数据，客户端可能不能用“计算字节数”的方法来获悉服务器什么时候结束。因此，服务器可能使用和客户端相同的策略，关闭连接中输入它的一半，通知客户端它已经结束响应了。

当回答完客户端的最后一个请求而且服务器已经关闭己方的连接告诉客户端它已经结束回答时，连接进入完全撤消阶段。请注意这时候发生了什么：客户端和服务器都已经使用了关闭己方连接作为通知对等方自己已经结束发送数据的方法。实际上，它们已经发送了一个 EOF 字符。

» shutdown 调用

应用程序是如何关闭自己一方的连接呢？因为仍然有数据来自对等方，所以它不能只是终止或关闭套接字。因此套接字 API 提供了 `shutdown` 接口。除了有一个额外的参数指示应当关闭哪一方的连接之外，`shutdown` 调用和 `close` 调用是基本一样的：

```
#include <sys/socket.h>      /* UNIX */
#include <winsock.h>          /* Windows */

int shutdown( int s, int how )  /* UNIX */
int shutdown( SOCKET s, int how ) /* Windows */

>Returns: 0 on success, -1 (UNIX) or SOCKET_ERROR (Windows) on error
```

不幸的是，在 UNIX 和 Winsock 的 `shutdown` 的具体实现之间存在着语意和 API 上的差

异。传统的 shutdown 具体实现的 how 参数只是作为个数。POSIX 和 Winsock 却给这个参数都赋予了符号名称，但是这些名称是不同的。图 3.1 给出了参数 how 的值、名称和意义。

how			操作
数字	POSIX	Winsock	
0	SHUT_RD	SD_RECEIVE	关闭连接的接收方
1	SHUT_WR	SD_SEND	关闭连接的发送方
2	SHUT_RDWR	SD_BOTH	关闭双方

图 3.1 shutdown 调用参数 how 的值

参数 how 的值在名称上的差异可以通过在头文件中定义另一个函数或仅使用数字值来弥补。然而，更为严重的事情是具体实现之间的语意差异。下面来看看它的三个操作究竟做些什么。

how = 0 关闭连接的接收方。两个具体实现都标记套接字为不能接受任何数据，如果应用程序已经发出了任何读操作，就返回 EOF。但是对那些在停止时已经在应用程序中排队的数据的响应，或者对那些对等方新数据的接收的响应，二者是明显不同的。UNIX 刷新输入队列，因此丢弃应用程序没有读取的任何数据。如果有新数据到达，TCP 确认了它们但是悄悄地把它们丢弃，这是因为应用程序不能接收任何数据了。另一方面，如果队列中有数据或者新的数据已经到达了，Winsock 就重置连接。因为这个原因，一些作者（请参阅[Quinn and Shute 1996]），认为在 Winsock 下使用 shutdown(s, 0) 很不安全。

how = 1 关闭发送方的连接。套接字标记为不能发送任何额外的数据，以后任何试图在套接字上发布的写操作将导致错误。发送缓冲区中的数据发送出去之后，TCP 发送一个 FIN 消息给对等方，告诉它不会有数据再传送过去了。这称作半关闭（half close）。这是使用 shutdown 最通常的情况，两种具体实现有相同的语意。

how = 2 关闭双方的连接。该操作如同 shutdown 调用了两次，一次设置参数 how 为零，另一次设置参数 how 为 1。虽然有人认为

```
shutdown( s, 2 );
```

等效于 close 或 closesocket 调用，但事实却并非如此，这在后面将做详细解释。通常没有什么原因通过设置参数 how 为 2 来调用 shutdown，但是[Quinn and Shute 1996]声称在一些 Winsock 的具体实现中，除非 shutdown 首先以 how=2 的方式调用否则 closesocket 是不会正常工作的。在 Winsock 环境下，以 how 设置为 2 的方式调用 shutdown 和以 how 设置为 0 的方式调用 shutdown 有着相同的问题——连接可能被重置。

关闭套接字和调用 `shutdown` 之间有很大的区别。首先, `shutdown` 实际上并没有“关闭”套接字, 即使是 `how` 设置为 2 时也是如此。也就是说, 套接字和它的资源(也许当参数 `how` 设置为 0 或 2 时接收缓冲区除外)都没有释放。同时, 注意当调用 `shutdown` 时, 它影响了所有打开套接字的进程。例如, 以 `how` 设置为 1 的方式调用使套接字的所有拥有者都不能往里面写数据。如果调用了 `close` 或 `closesocket`, 就像什么事情也没有发生一样, 套接字的其他拥有者仍然可以使用它。

最后的那个事实可以看作使用 `shutdown` 的一个优势。只要通过以 `how=1` 的方式调用 `shutdown`, 就可以保证对等方接收到一个 EOF 字符, 而不管其他的进程是否已经打开了套接字。而调用 `close` 或 `closesocket` 就不能保证, 因为直到套接字的引用记数减为 0 时才会发送 FIN 消息给对等方。也就是说, 直到所有的进程都已经关闭了套接字。

最后, 有必要提醒一下的是, 尽管本节主要讨论的是 TCP, `shutdown` 也可以用于 UDP。因为 UDP 不需要关闭连接, 所以以参数 `how` 设置为 1 或 2 的方式调用 `shutdown` 的用法是有问题的, 但是以参数 `how` 设置为 0 的方式调用它来防止在一个特定的 UDP 端口接收数据报是很有用的。

顺序释放

到目前为止已经讨论了 `shutdown` 调用, 下面看看它是如何用于连接的顺序释放的。顺序释放的目的是为了在连接撤消之前保证双方接收来自对等方的所有数据。

术语“顺序释放”和 XTI (请参阅技巧 5) `t_sndrel` 命令有关, 但是不要和它混淆, 通常称 `t_sndrel` 命令为顺序释放是为了和中途释放 (abortive release) `t_snndis` 区分。`t_sndrel` 命令和套接字接口的 `shutdown` 命令执行相同的功能。两个命令都用于引起连接的顺序释放。

因为来自对等方的数据有可能丢失, 所以在一些情况下仅仅关闭连接是不够的。回忆一下当应用程序关闭连接时, 没有传递的数据都会丢弃掉。

为了对顺序释放过程进行实验有所帮助, 下面编写了客户端, 该客户端发送数据给服务器, 然后读取并打印来自服务器的应答。客户端显示在图 3.2 中。客户端从标准输入中读取发送给服务器的数据, `fgets` 返回 NULL 说明接收到了 EOF 字符, 客户端开始撤消连接。命令行-c 是切换客户端如何撤消连接的控制。如果没有指定-c, `shutdownnc` 就调用 `shutdown` 来关闭连接的写入方。如果指定了-c, `shutdownnc` 就调用 CLOSE, 睡眠 5 秒钟, 然后退出。

shutdownnc.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     fd_set readmask;
6     fd_set allreads;
```

```
7 int rc;
8 int len;
9 int c;
10 int closeit = FALSE;
11 int err = FALSE;
12 char lin[ 1024 ];
13 char lout[ 1024 ];

14 INIT();
15 opterr = FALSE;
16 while ( ( c = getopt( argc, argv, "c" ) ) != EOF )
17 {
18     switch( c )
19     {
20         case 'c' :
21             closeit = TRUE;
22             break;

23         case '?' :
24             err = TRUE;
25     }
26 }

27 if ( err || argc - optind != 2 )
28     error( 1, 0, "usage: %s [-c] host port\n",
29            program_name );
30 s = tcp_client( argv[ optind ], argv[ optind + 1 ] );
31 FD_ZERO( &allreads );
32 FD_SET( 0, &allreads );
33 FD_SET( s, &allreads );
34 for ( ; )
35 {
36     readmask = allreads;
37     rc = select( s + 1, &readmask, NULL, NULL, NULL );
38     if ( rc <= 0 )
39         error( 1, errno, "bad select return (%d)", rc );
40     if ( FD_ISSET( s, &readmask ) )
```

```

41      {
42          rc = recv( s, lin, sizeof( lin ) - 1, 0 );
43          if ( rc < 0 )
44              error( 1, errno, "recv error" );
45          if ( rc == 0 )
46              error( 1, 0, "server disconnected\n" );
47          lin[ rc ] = '\0';
48          if ( fputs( lin, stdout ) )
49              error( 1, errno, "fputs failed" );
50      }
51      if ( FD_ISSET( 0, &readmask ) )
52      {
53          if ( fgets( lout, sizeof( lout ), stdin ) == NULL )
54          {
55              FD_CLR( 0, &allreads );
56              if ( closeit )
57              {
58                  CLOSE( s );
59                  sleep( 5 );
60                  EXIT( 0 );
61              }
62              else if ( shutdown( s, 1 ) )
63                  error( 1, errno, "shutdown failed" );
64          }
65          else
66          {
67              len = strlen( lout );
68              rc = send( s, lout, len, 0 );
69              if ( rc < 0 )
70                  error( 1, errno, "send error" );
71          }
72      }
73  }
74 }
```

shutdownc.c

图 3.2 进行顺序释放实验的客户端

初始化

14~30 程序执行标准的 TCP 客户端初始化，检查命令行中的-c 开关。

处理数据

40~50 如果 TCP 套接字可读取数据，程序就试图读取尽可能多（一直到缓冲区的大小）的数据。如果程序得到一个 EOF 字符或错误，就终止；否则，程序把接收到的任何数据写入到标准输出。

请注意第 42 行 recv 调用中的 sizeof(lin) - 1。虽然技巧 11 中的避免缓冲区溢出的教训仍历历在目，该代码的原始版本指定的是 sizeof(lin)，这行代码在第 47 行代码

```
lin[ rc ] = '\0';
```

溢出缓冲区时会导致一个失败。

53~64 如果程序从标准输出中获得一个 EOF 字符，就会根据是否设置了-c 开关来调用 shutdown 或 CLOSE。

65~71 否则，程序把来自标准输入的数据写往服务器。

程序本来可以使用系统提供的 echo 服务来对客户端进行实验，但是为了搞清楚发生了什么错误以及引入一个可选的延迟选项，程序提供了自己版本的 TCP echo 服务器。程序 tcpecho.c 除了采用了一个可选的命令行参数外，没有什么值得注意的地方，如果提供了该参数，就会导致程序读取和响应每一个数据块之间睡眠指定的秒数（如图 3.3 所示）。

首先，以-c 开关运行 shutdownc，这样程序可以在从标准输入获得 EOF 字符时关闭套接字。程序设置 tcpecho 在响应读取的数据之前延迟 4 秒：

```

bsd: $ tcpecho 9000 4 &
[1] 3836
bsd: $ shutdownc -c localhost 9000
data1
data2
^D
tcpecho: send failed: Broken pipe (32)      These three lines were
                                                entered one after another
                                                as quickly as possible.
                                                4 seconds after "data1" sent

```

tcpecho.c

```

1 #include <etcp.h>

2 int main( int argc, char **argv )
3 {
4     SOCKET s;

```

```

5   SOCKET s1;
6   char buf[ 1024 ];
7   int rc;
8   int nap = 0;

9   INIT();
10  if ( argc == 3 )
11      nap = atoi( argv[ 2 ] );
12  s = tcp_server( NULL, argv[ 1 ] );
13  s1 = accept( s, NULL, NULL );
14  if ( !isvalidsock( s1 ) )
15      error( 1, errno, "accept failed" );
16  signal( SIGPIPE, SIG_IGN ); /* report sigpipe to us */
17  for ( ; ; )
18  {
19      rc = recv( s1, buf, sizeof( buf ), 0 );
20      if ( rc == 0 )
21          error( 1, 0, "client disconnected\n" );
22      if ( rc < 0 )
23          error( 1, errno, "recv failed" );
24      if ( nap )
25          sleep( nap );
26      rc = send( s1, buf, rc, 0 );
27      if ( rc < 0 )
28          error( 1, errno, "send failed" );
29  }
30 }
```

tcpecho.c

图 3.3 TCP echo 服务器

接下来，键入两行数据“data1”和“data2”，后面紧跟着<CNTRL-D>，发送一个 EOF 字符给 shutdownc，导致程序关闭套接字。注意两行数据都不会返回。来自 tcpecho 的错误消息告诉我们所发生的事情：当服务器从 sleep 调用返回并试图返回“data1”数据行时，它获得一个返回的 RST 消息，这是因为客户端已经关闭了连接。

如技巧 9 中指出的，事实上写入第二行（“data2”）返回一个错误。也注意到有一种情况是通过写入调用替代读取调用返回一个出错信息，参见技巧 15。

关键的问题是，尽管客户端给服务器发送已经结束发送数据的信号，客户端还是在服务器完成处理之前就撤消连接了，因此数据丢失了。图 3.4 的剩余部分显示了 TCP 段的交换。

下面重复进行该实验，但是不用-c 开关启动 shutdownc。

```
bsd: $ tcpecho 9000 4 &
[1] 3845
bsd: $ shutdownc localhost 9000
data1
data2
^D
data1                                4 seconds after "data1" sent
data2                                4 seconds after "data1" received
tcpecho: client disconnected
shutdownc: server disconnected
```

这次，程序正常运行。当 shutdownc 发现标准输出上的 EOF 时，它调用 shutdown，通知 tcpecho 它已经结束发送数据。但是它继续从连接中读取数据。当 tcpecho 看到来自客户端的 EOF 字符时，它就关闭连接，导致 TCP 发送队列中残留的任何数据，并在最后发送 FIN 消息。当客户端接收 EOF 时，它知道自己已经接收到了所有来自服务器的消息，因此程序终止。

注意直到服务器试图写数据并接收到错误或 EOF 字符时，才能区分客户端所做的操作（shutdown 或 close）。如图 3.4 所示，直到 shutdownc 的 TCP 响应包含“data1”的数据，两个实验的数据交换却是相同的。

因为它是一个经常导致困惑的地方，所以最后还有一点值得注意。我们已经在示例代码中多次看到当 TCP 从对等方接收到一个 FIN 消息时，它就通过返回值 0 给读操作来通知应用程序。从图 3.2 的第 45 行和图 3.3 的第 20 行可以看到这些例子，这些例子是通过询问 recv 是否返回 0 来显式地检查 EOF 条件。如果涉及到了一个 select 调用，那么在和图 3.2 类似的例子的情况下，就经常会发生混淆。当应用程序的对等方通过关闭或停止或终止来关闭连接的写入方时，select 返回套接字一个读操作事件。如果应用程序没有检查 EOF 条件，它可能就会试图处理一个零长度的段或在读操作和 select 之间连续地循环。

网络新闻组中经常遇到的问题是抱怨“虽然 select 显示有数据可以读取，当我执行读操作时，却没有数据可获得。”当然，这里所发生的问题是对等方已经关闭了（至少）连接的写入方，而“读操作事件”其实是 TCP 把对等方的 EOF 发送给应用程序。

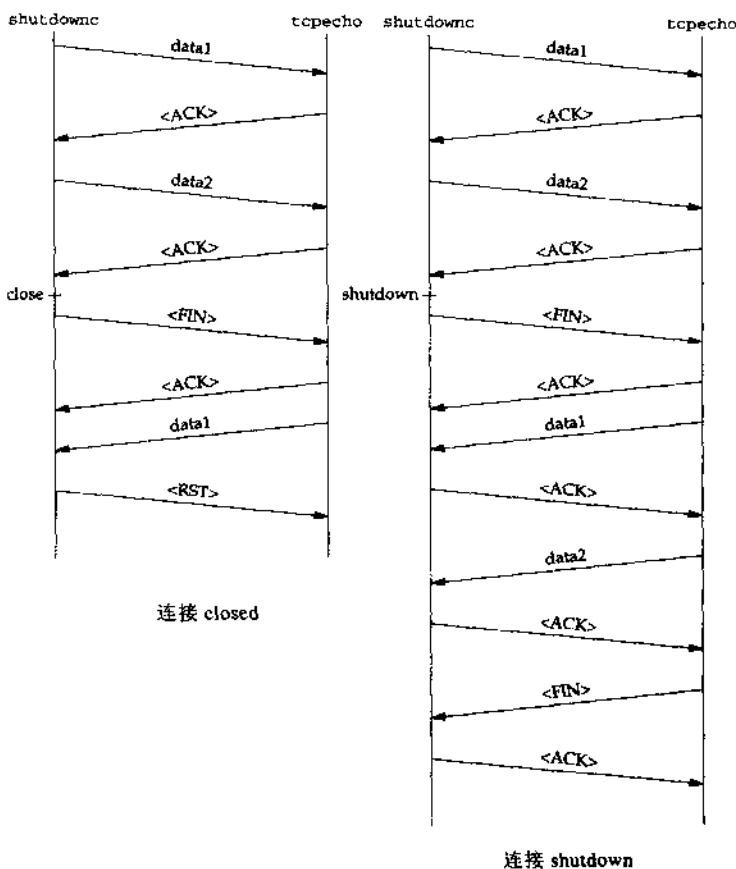


图 3.4 close 和 shutdown 的比较

小结

本节讨论了 TCP 的 shutdown 系统调用并拿它和 close 调用做比较。从中了解了可以用 shutdown 调用来关闭连接的读方、写方或者读写双方，而且 shutdown 调用和 close 调用影响套接字的引用记数方式不同。

接着本节显示了如何使用 shutdown 来调用连接的顺序释放的。顺序释放可以确保不会丢失数据的连接释放过程。

技巧 17 考虑让 inetd 启动应用程序

当程序运行在 UNIX 或其他操作系统上时，Internet 超级服务器 inetd 提供了一种简单的方法只需要少量的工作就使应用程序具有网络功能。同时，让单一进程监听进来的连接

或 UDP 数据报有助于节省内存。

通常情况下，inetd 至少支持 TCP 和 UDP 协议，同时也有可能支持其他的协议。本章考虑的仅仅是 inetd 在 TCP 和 UDP 协议上的使用。根据 inetd 监听的是 TCP 连接还是 UDP 连接的不同，inetd 的操作有很大的不同。

» TCP 服务器

对于 TCP 服务器来说，inetd 监听在应用程序已知的端口上监听连接请求；接受连接；映射连接到标准输入、标准输出和标准错误输出；启动适当的服务器。当服务器运行时，从标准输入获取输入，把输出写入到标准输出或标准错误输出。也就是说，跟对等方的连接可以在文件描述符 0、1 和 2 上获得。只有在 inetd 配置文件（/etc/inetd.conf）中特别指定，inetd 才会在已知端口监听其他连接。如果这样的一个连接到达了，就启动服务器的另一个实例，而不管第一个实例是否已经终止。图 3.5 举例说明了这种情况。注意服务器没有必要关心它自己是否需要服务于多个客户端。它仅仅是简单地对另一方的单一客户端执行自己的功能然后终止。其他客户端由其他的服务器实例来服务。

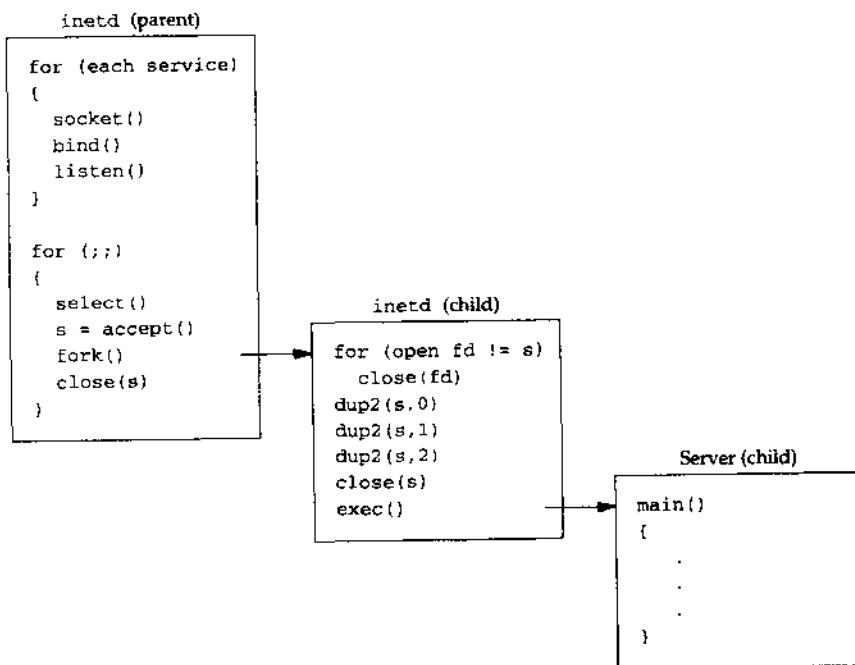


图 3.5 启动 TCP 服务器时 inetd 所做的操作

使用 inetd 通常是一个十分吸引人的策略，因为它把我们从管理 TCP 连接或 UDP 建立的琐事中解脱出来，使我们能够把应用程序编写为好像是一个普通的过滤器。图 3.6 中给出的例子虽然不是特别有趣，却很简单明了。

rlnumd.c

```

1 #include <stdio.h>

2 void main( void )
3 {
4     int cnt = 0;
5     char line[ 1024 ];

6     /*
7      * We have to explicitly set line buffering since the stdio
8      * routines will not see the socket as a terminal device.
9      */
10    setvbuf( stdout, NULL, _IOLBF, 0 );
11    while ( fgets( line, sizeof( line ), stdin ) != NULL )
12        printf( "%3i: %s", ++cnt, line );
13 }

```

rlnumd.c

图 3.6 rlnumd——行记数程序

该程序许多地方需要详细解释：

(1) 程序的代码中没有涉及到任何 TCP 或网络的引用。这并不意味着程序不能使用有关套接字的调用 (`getpeername`, `[gs]etsockopt` 等等)，而是程序并不总是需要这么做。程序也不会在使用 `read` 或 `write` 时受到限制。程序就像没有使用 `inted` 一样，可以使用 `send`、`recv`、`sendto` 和 `recvfrom`。

(2) 程序不得不显式地设置行缓冲区。这是因为标准 I/O 库仅仅当它认为是给终端写数据时才自动地设置行缓冲区。标准 I/O 库通过这个设置为交互应用程序提供即时的反馈。

(3) 如技巧 6 里讨论的那样，标准 I/O 库处理输入流，把它们解析为记录。

(4) 程序默认一行数据不会长于 1023 个字节。多于 1023 个字节的行将在每 1023 个字节后面加入序号。

[Oliver 2000] 中指出的事实是另一个程序很容易不处理缓冲区错误的例子。技巧 11 中以一定的篇幅讨论了这个问题。

(5) 尽管这是一个微不足道的程序，但是许多“真实的”TCP/IP 应用程序如 `rlogin`、`telnet` 和 `ftp` 使用的就是这个技术。

图 3.6 中的程序可以作为“普通的”过滤器或作为远程的行记数服务程序。为了建立远程行记数服务，仅需要选择一个端口号，并在 `/etc/services` 中增加一个带有服务名称和端

口号的条目以及在/etc/inetd.conf 中增加一个描述服务和寻找服务可执行程序位置的条目。例如，如果要调用服务 rlnum，并用服务器 rlnum 实现它并指定其端口为 8000，那么就需要加入下面一行：

```
rlnum      8000/tcp #remote line-numbering service
```

到/etc/services 文件中，以及增加下面一行：

```
rlnum stream tcp nowait jcs /usr/home/jcs/rlnumd rlnumd
```

到/etc/inetd.conf。/etc/services 文件中的那行内容仅仅说明了服务 rlnum 在端口 8000 使用 TCP。/etc/inetd.conf 文件中的域的意义如下：

(1) 服务名称如/etc/services 文件中列出的一样。这是客户端在使用该服务器时所连接的已知端口的“名称”。在这个例子中，服务名称是 rlnum。

(2) 服务器使用的套接字类型。TCP 服务器为 stream，UDP 服务器为 dgram。因为这个例子是 TCP 服务器，所以指定为 stream。

(3) 服务器使用的协议。该域为 tcp 或 udp。这个例子中，指定为 tcp。

(4) wait/nowait 标志。UDP 服务器总是为 wait，而 TCP 服务器总是为 nowait。当指定为 nowait 时，inetd 在接收到连接后立即继续在服务器的已知端口监听其他的连接。如果指定为 wait，inetd 在服务器终止前不会对套接字做进一步的处理。之后，它继续监听连接请求（基于流的服务器）或数据报的到来（基于数据报的服务器）。如果基于流的服务器指定为 wait，inetd 就不为连接调用 accept，但是传递监听套接字给服务器，该服务器在终止前必须接收至少一个连接。[Kacker 1998]中指出，指定 TCP 应用程序为 wait 是一个强大但是难于理解的选择。下面是一些可能在 TCP 连接上使用 wait 选项的应用程序：

- 作为不可靠网络守候进程的重启动驱动。只要守候进程正确地运行，它就接受来自客户端的连接，但是如果守候进程因为某些原因崩溃了，下一个连接请求将导致 inetd 重新启动。
- 作为一种保证某一时刻只有一个客户端使用该服务器的方法。
- 作为一种控制负载驱动的多线程/多进程应用程序的方法。使用这种模式，初始进程由 inetd 启动，然后按照需求由产生的子进程或线程动态地控制负载。当负载减少时，子进程或线程就会运行，最后当父进程空闲时，就停止运行，释放所有的资源并将返回监听套接字的控制权给 inetd。

在这个例子中，和通常的 TCP 服务器一样，指定的是 nowait。

(5) 运行服务器的用户名。该名称应当在/etc/passwd 文件中。/etc/inetd.conf 文件中的许多标准服务器都指定 root 为用户名，但是服务器可以以任何用户的名义运行。本例子选择用户 jcs 运行 rlnum 服务器。

(6) 服务器可执行文件的完整路径。因为 rlnumd 是在用户 jcs 的主目录，所以指定该域为 /usr/home/jcs/rlnumd。

(7) 可以传递给服务器的参数（以 argv[0] 开始），最多 5 个。

为了运行新的服务，就迫使 inetd 重新读取它的配置文件（在大多数的具体实现中通过发送 inetd 一个 SIGHUP 信号来完成），并用 telnet 连接它。

```
bsd: $ telnet localhost rlnum
Trying 127.0.0.1...
Connected to localhost
Escape character is '^]'.

hello
1: hello
word
2: world
^]
telnet> quit
Connection closed.

Bsd: $
```

➤ UDP 服务器

因为 UDP 是一个无连接协议（请参阅技巧 1），所以 inetd 没有连接可以监听。当 UDP 服务器的已知端口上有数据可读时，inetd 就要求操作系统（通过 select 系统调用）通知它。当通知到达时，inetd 映射套接字到标准输入、标准输出以及标准错误输出，并启动 UDP 服务器。和 TCP 服务器一般设置为不等待不同的是，直到 inetd 所启动的服务器终止时，inetd 才会在已知端口上进行进一步的操作。这时，如果已知端口上有数据报到达，它将再次要求操作系统通知它。服务器在退出之前必须从套接字读取至少一条消息，因此 inetd 不会看到相同的消息以及派生并且不会再次执行它，以避免死循环。

图 3.7 中的例子显示了 inetd 启动 UDP 服务器的一个例子，该 UDP 服务器返回带进程 ID 的数据报。

udpecho1.c

```
1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     struct sockaddr_in peer;
5     int rc;
6     int len;
```

```

7   int pidsz;
8   char buf[ 120 ];

9   pidsz = sprintf( buf, "%d: ", getpid() );
10  len = sizeof( peer );
11  rc = recvfrom( 0, buf + pidsz, sizeof( buf ) - pidsz, 0,
12      ( struct sockaddr * )&peer, &len );
13  if ( rc <= 0 )
14      exit( 1 );
15  sendto( 1, buf, rc + pidsz, 0,
16      ( struct sockaddr * )&peer, len );
17  exit( 0 );
18 }

```

udpecho1.c

图 3.7 简单的请求/应答 UDP 服务器

udpecho1

9 程序从操作系统获得服务器的进程 ID (PID)，把它转换为 ASCII 码，并把它放在 I/O 缓冲区的开头。

10~14 程序从客户端读取数据报到服务器 PID 之后的缓冲区。

15~17 程序返回带服务器进程 ID 的数据报，然后服务器退出。

为了实验该服务器，必须使用一个简单 UDP 客户端，如图 3.8 所示，该客户端从标准输入读取请求，把它们发送到服务器，并把应答写入到标准输出。

udpclient.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     struct sockaddr_in peer;
5     SOCKET s;
6     int rc = 0;
7     int len;
8     char buf[ 120 ];
9     INIT();
10    s = udp_client( argv[ 1 ], argv[ 2 ], &peer );
11    while ( fgets( buf, sizeof( buf ), stdin ) != NULL )

```

```

12  {
13      rc = sendto( s, buf, strlen( buf ), 0,
14          ( struct sockaddr * )&peer, sizeof( peer ) );
15      if ( rc < 0 )
16          error( 1, errno, "sendto failed" );
17      len = sizeof( peer );
18      rc = recvfrom( s, buf, sizeof( buf ) - 1, 0,
19          ( struct sockaddr * )&peer, &len );
20      if ( rc < 0 )
21          error( 1, errno, "recvfrom failed" );
22      buf[ rc ] = '\0';
23      fputs( buf, stdout );
24  }
25  EXIT( 0 );
26 }

```

udpclient.c

图 3.8 简单 UDP 客户端

udpclient

10 程序调用 `udp_client`, 用对等方的地址填充 `peer` 变量, 获得一个 UDP 套接字。

11~16 程序从标准输入读取一行, 并把它作为一个 UDP 数据报发送给命令行中指定的主机和端口。

17~21 程序调用 `recvfrom` 读取服务器的应答, 如果出现错误就退出。

22~23 程序以 `null` 结束应答并把它写入到标准输出。

对于 `udpclient`, 有两点需要指出:

(1) 客户端假定它总是可以从服务器获得应答。如技巧 1 中讨论的那样, 网络不能保证服务器的数据报将会到达。因为 `udpclient` 是一个交互程序, 所以如果调用 `recvfrom` 时挂起, 可以退出并重新启动它。然而, 在一个非交互的客户端中, 应当在 `recvfrom` 周围设置一个计数器来防止数据报丢失的情况发生。

我们并不需要在 `udpecho1` 中考虑这些情况, 因为我们直到有一个数据报可以读取 (否则 `inetd` 就不会启动服务器)。因为下一个服务器 (请参阅图 3.9), 将会考虑数据报丢失, 所以在 `recvfrom` 附近放置了一个计时器。

(2) 使用 `udpecho1` 时, 不需要接受发送方的地址和端口号, 因为这些数据都是已知的。所以第 18 行和第 19 行可以替换为:

```
rc = recvfrom ( s, buf, sizeof( buf ) - 1, 0, NULL, NULL);
```

然而，在下一个例子中，客户端有时需要知道服务器发送应答的地址，所以 UDP 客户端总是取该地址。

为了测试该服务器，在/etc/inetd.conf 文件中加入下列一行：

```
udpecho dgram udp wait jcs /usr/home/jcs/udpechod udpechod
```

在 bsd 的/etc/services 文件中加入下列一行：

```
udpecho 8001/udp
```

然后重命名 udpecho1 为 udpechod，并通知 inetd 重新读取它的配置文件。当 udpclient 运行在 sparc 上时，得到以下结果：

```
sparc: $ udpclient bsd udpecho
one
28685: one
two
28686: two
three
28687: three
^C
sparc: $
```

这个结果说明了典型的 UDP 服务器的一个重要事实：它们通常不和客户端进行对话。也就是说，UDP 服务器仅接收一个请求并响应单一的应答。在 UDP 服务器由 inetd 启动的情况下，典型的操作是接收请求，响应，然后退出。UDP 服务器不得不尽快退出，这是因为 inetd 不在服务器的已知端口上监听其他请求，它知道服务器已经终止才重新监听。

在前面的结果中，可以知道即使 udpclient 看起来好像是和 udpecho1 进行一个对话，其实它是为服务器发送的每条消息调用一个服务器的实例。这明显效率不高，更为重要的是它意味着服务器不在来自客户端的消息之间保留状态信息。这对于 udpecho1 来说没什么问题，因为 udpecho1 的每一条消息实际上都是一个独立的事物，但是并不是所有的服务器都这样。解决这个问题的一个办法就是，当服务器要接受一条来自客户端的消息时（为了防止无限循环），它就连接到客户端，这样就会获得一个新的（暂时的）端口，并产生一个子进程，然后服务器退出。以后的会话由子进程来进行。

存在另一种可能。例如，服务器可以自己保持几个单独的客户端的信息。通过接受来自多个客户端的数据报，服务器可以把它的启动耗费分摊到几个客户端上，还可以一直运行直到已经空闲了一段时间为止。这种方法有使客户端变简单的优势，但是服务器的耗费要多一些。

—udpecho2.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     struct sockaddr_in peer;
5     int s;
6     int rc;
7     int len;
8     int pidsz;
9     char buf[ 120 ];
10
11    pidsz = sprintf( buf, "%d: ", getpid() );
12    len = sizeof( peer );
13    rc = recvfrom( 0, buf + pidsz, sizeof( buf ) - pidsz,
14                   0, ( struct sockaddr * )&peer, &len );
15    if ( rc < 0 )
16        exit( 1 );
17
18    s = socket( AF_INET, SOCK_DGRAM, 0 );
19    if ( s < 0 )
20        exit( 1 );
21    if ( connect( s, ( struct sockaddr * )&peer, len ) < 0 )
22        exit( 1 );
23    if ( fork() != 0 )           /* error or parent? */
24        exit( 0 );
25
26    /* child process */
27
28    while ( strncmp( buf + pidsz, "done", 4 ) != 0 )
29    {
30        if ( write( s, buf, rc + pidsz ) < 0 )
31            break;
32        pidsz = sprintf( buf, "%d: ", getpid() );
33    }
34 }
```

```

29     alarm( 30 );
30     rc = read( s, buf + pidsz, sizeof( buf ) - pidsz );
31     alarm( 0 );
32     if ( rc < 0 )
33         break;
34 }
35 exit( 0 );
36 }

```

udpecho2.c

图 3.9 udpechod 的另一个版本

udpecho2

10~15 程序获得进程的 PID，把它放在缓冲区中，然后接收第一个消息，如 *udpecho1* 所做的一样。

16~20 程序获得一个新套接字，并使用 *peer* 中的地址把它连接到客户端，*peer* 中的地址是由 *recvfrom* 调用用来填充的。

21~22 父进程派生子进程然后退出。这时 *inetd* 是空闲的，它可以在服务器的已知端口上监听其他消息。这里需要指出的很重要的一点是，子进程使用的是一个新端口号，它由 *connect* 调用绑定到套接字上。

24~35 程序接着附加父进程的 PID 到第一条消息上，并把它返回给客户端。程序继续接收来自客户端的消息，附加上子进程的 PID，返回消息给客户端，直到获得一个以“done”开头的消息为止，这时服务器终止运行。第 30 行周围的 *alarm* 语句是为了防止客户端在没有发送“done”消息之前就终止运行，导致 *udpecho2* 一直挂起。因为程序没有设置 *SIGALRM* 的信号处理程序，所以如果 *alarm* 终止，*UNIX* 就停止运行。

当重命名新版本的服务器为 *udpechod*，并运行时，就会获得以下结果：

```

sparc: $ udpclient bsd udpecho
one
28743: one
two
28744: two
three
28744: three
done
^C
sparc: $

```

这次可以看到第一条消息有父进程的 PID（服务器由 inetd 启动），而其他的消息有相同的 PID——子进程的 PID。从中也可以了解 udpclient 为什么总是取对等方的地址：它需要找出后面消息的新端口号（在多宿主服务器的情况下，还有可能是新的 IP 地址）。当然，只需要第一个 recvfrom 调用做这件事情，但是为了使程序简单，没有为第一个调用使用特殊的处理。

» 小结

本节介绍了如何不必费太大的力气使应用程序增加网络功能。inetd 守候程序负责监听连接或到达的数据报；映射套接字到标准输入、标准输出和标准错误输出，然后启动应用程序。之后，应用程序仅可以从标准输入、标准输出和标准错误输出中读取或写入数据，而不必担心甚至是知道它正在和网络进行会话。本节分析了一个根本没有网络代码的过滤器的例子，这个例子由 inetd 启动后作为远程服务工作得很好。

本节同时分析了一个 UDP 服务器，该服务器可以和它的客户端进行扩展的会话。这需要服务器获得一个新套接字和端口号，然后在退出之前派生一个子进程。

技巧 18 考虑让 tcpmux “指定”

服务器的已知端口

设计者面对的一个问题是已知端口号的选择。Internet Assigned Numbers Authority (IANA) 划分可获得的端口为三类：“官方”已知端口、已注册的端口以及动态或私有端口。

前面我们已经使用“已知端口”作为一般服务器的存取端口号。严格来说，已知端口是那些由 IANA 控制的端口。

已知端口号是那些介于 0 和 1023 之间的端口号。它们由 IANA 控制。已注册的端口号是那些介于 1024 和 49151 之间的端口号。IANA 不控制这些端口号，但是他们注册这些端口号并作为服务列出这些端口号给网络社区。动态或私有端口号是那些介于 49152 到 65535 之间的端口号。这些端口号应该用于暂时的端口，但是许多系统并不遵循这个约定。例如，BSD 派生的系统通常使用介于 1024 和 5000 之间的端口号作为暂时端口号。可以在 <http://www.isi.edu/in-notes/iana/assignments/port-numbers> 上获得 IANA 分配的最新已知端口和已注册端口的完整列表。

服务器程序设计者可以向 IANA 获取一个注册端口号。

可以在 <http://www.isi.edu/cgi-bin/iana/port-number.pl> 上在线申请分配已知端口或注册端口。

当然，这并不能阻止其他人使用分配的端口，而且使用同一端口的服务器迟早会运行在同一台机器上。处理这个问题的一般方法是使用一个默认的端口号，但是可以使用命令行选项来改变它。

另一个很少使用但灵活度更高的解决方案是使用 inetc 的 TCP Port Service Multiplexor (TCPMUX) 功能（请参阅技巧 17）。RFC 1078 中详细地描述了 TCPMUX 功能。TCPMUX 在端口 1 上监听 TCP 连接。客户端连接到 TCPMUX 然后发送它要启动的服务名称给 TCPMUX，该服务名称应当以回车换行符 (<CR><LF>) 结束。服务器，或者是 TCPMUX，发送一个字符给客户端，指示一个正面 (+) 或负面 (-) 的确认，之后跟着一条可选的以 <CR><LF> 结尾的解释性消息。服务名称是放在 inetc.conf 文件中，大小写不敏感，但是它们以字符串 tcpmux/开头以区分一般的服务名称。如果服务名称以+字符开头，TCPMUX 代替服务器发送一条正面确认消息，这就允许那些没有使用 TCPMUX 协议的服务器，如 rlnumd（参阅图 3.6），照样使用 TCPMUX。

例如，如果希望以 TCPMUX 服务器的方式运行技巧 17 中的远程行记数服务，就需要在 /etc/inetc.conf 文件中加入下面一行：

```
tcpmux/+rlnumd stream tcp nowait jcs /usr/home/jcs/rlnumd rlnumd
```

通过通知 inetc 读取它的配置文件然后使用 telnet，指定 TCPMUX 服务，来进行测试。

```
bsd: $ telnet localhost tcpmux
Trying 127.0.0.1...
Connected to localhost
Escape character is '^}'.

rlnumd
+Go
hello
 1: hello
world
 2: world
^]
telnet> quit
Connection closed.

bsd: $
```

TCPMUX 的一个问题是它不是被所有的操作系统支持，它甚至并不是被所有的 UNIX 操作系统支持。幸运的是，TCPMUX 协议十分简单，甚至可以轻松地编写自己的版本。因

为 TCPMUX 必须做和 inetd 同样多的工作（除了监听几个端口之外），下面编写的自己的版本 inetd 可以帮助加深对 inetd 是如何工作的理解。下面以定义、全局变量和 main 函数来分析该程序（如图 3.10 所示）。

tcpmux.c

```

1 #include "etcp.h"

2 #define MAXARGS      10      /* maximum arguments to server */
3 #define MAXLINE       256     /* maximum line size in tcpmux.conf */
4 #define NSERVTAB    10      /* number of service_table entries */
5 #define CONFIG        "tcpmux.conf"

6 typedef struct
7 {
8     int flag;
9     char *service;
10    char *path;
11    char *args[ MAXARGS + 1 ];
12 } servtab_t;

13 int ls;                      /* socket to listen on */
14 servtab_t service_table[ NSERVTAB + 1 ];

15 int main( int argc, char **argv )
16 {
17     struct sockaddr_in peer;
18     int s;
19     int peerlen;

20 /* Initialize and start the tcpmux server */

21     INIT();
22     parsetab();
23     switch ( argc )
24     {
25         case 1:      /* default everything */
26             ls = tcp_server( NULL, "tcpmux" );

```

```

27         break;

28     case 2: /* specify interface, default port */
29         ls = tcp_server( argv[ 1 ], "tcpmux" );
30         break;

31     case 3: /* specify everything */
32         ls = tcp_server( argv[ 1 ], argv[ 2 ] );
33         break;

34     default:
35         error( 1, 0, "usage: %s [ interface [ port ] ]\n",
36               program_name );
37     }
38     daemon( 0, 0 );
39     signal( SIGCHLD, reaper );

40 /* Accept connections to tcpmux port */

41 for ( ; )
42 {
43     peerlen = sizeof( peer );
44     s = accept( ls, ( struct sockaddr * )&peer, &peerlen );
45     if ( s < 0 )
46         continue;
47     start_server( s );
48     CLOSE( s );
49 }
50 }
```

tcpmux.c

图 3.10 tcpmux——定义、全局变量和 main 函数

✍ main

6-12 servtab_t 对象定义了 service_table 中的条目。如果 tcpmux 代替服务器发送正面确认消息的话，flag 域就被设置为 TRUE。

22 程序以调用 parsetab 开始，读取并遍历 tcpmux.conf 文件，并建立 service_table。

parsetab 函数以后将在图 3.12 中介绍。

23~37 我们自己版本的 `tcpmux` 允许用户指定它将要监听的接口或端口。程序代码用用户指定的任何参数初始化服务器，而为其他服务器使用默认的端口。

38 程序调用 `daemon` 函数把 `tcpmux` 放在后台并断开它和终端的联系。

39 程序为 `SIGCHLD` 信号安装一个信号处理程序。这是为了防止程序启动的服务器在终止时发生异常（并占用系统资源）。

在一些系统中，`signal` 函数为老版本的“不可靠信号”提供了一个接口。对于这些系统来说，`sigaction` 函数应当用来提供可靠的信号语意。一个通常的策略是按照 `sigaction` 的方式来提供自己的 `signal` 函数。附录 A 中给出了该函数的具体实现。

41~49 程序中的循环接收到 `tcpmux` 的连接，并调用 `start_server`，派生并执行被请求的服务器。`start_server` 函数如图 3.11 所示。当 `start_server` 返回时，因为父进程不再需要使用套接字，所以关闭它。

下面分析以下 `start_server` 函数。这个函数做了 `tcpmux` 中的大部分工作。

tcpmux.c

```

1 static void start_server( int s )
2 {
3     char line[ MAXLINE ];
4     servtab_t *stp;
5     int rc;
6     static char err1[] = "-unable to read service name\r\n";
7     static char err2[] = "-unknown service\r\n";
8     static char err3[] = "-unable to start service\r\n";
9     static char ok[] = "+OK\r\n";
10
11    rc = fork();
12    if ( rc < 0 )          /* fork error */
13    {
14        write( s, err3, sizeof( err3 ) - 1 );
15        return;
16    }
17    if ( rc != 0 )          /* parent */
18    {
19        CLOSE( ls );          /* close listening socket */

```

```

20 alarm( 10 );
21 rc = readcrlf( s, line, sizeof( line ) );
22 alarm( 0 );
23 if ( rc <= 0 )
24 {
25     write( s, err1, sizeof( err1 ) - 1 );
26     EXIT( 1 );
27 }

28 for ( stp = service_table; stp->service; stp++ )
29     if ( strcasecmp( line, stp->service ) == 0 )
30         break;
31 if ( !stp->service )
32 {
33     write( s, err2, sizeof( err2 ) - 1 );
34     EXIT( 1 );
35 }

36 if ( stp->flag )
37     if ( write( s, ok, sizeof( ok ) - 1 ) < 0 )
38         EXIT( 1 );
39 dup2( s, 0 );
40 dup2( s, 1 );
41 dup2( s, 2 );
42 CLOSE( s );
43 execv( stp->path, stp->args );
44 write( 1, err3, sizeof( err3 ) - 1 );
45 EXIT( 1 );
46 }

```

-tcpmux.c

图 3.11 start_server 函数

↙ start_server

10~17 函数首先派生并创建一个子进程，该子进程和父进程相同，如果派生失败，函数写一个错误消息给客户端并返回（因为派生失败，所以没有子进程——函数返回到父

进程里的 main 函数）。如果派生成功，而且派生的进程就是父进程，函数就返回。

19~27 在子进程中，函数关闭监听套接字并读取客户端希望从连接套接字上启动的服务。函数在读操作周围放置一个警报信号，以便如果客户端再也不发送服务名称时函数可以自动终止。如果 readcrlf 返回错误，函数发送一个错误消息给客户端并退出。函数 readcrlf 在后面的图 3.13 中接收。

28~35 函数在 service_table 中搜寻被请求服务的名称，如果没有找到它，函数就写一个错误消息给客户端并退出。

36~38 如果服务名称以+开头，函数发送一个正面的确认给客户端。否则，函数让服务器发送它。

39~45 函数 dup 套接字到标准输入、标准输出和标准错误输出，并关闭初始的套接字。最后函数调用 execv 用服务器的进程映像代替本进程映像。调用 execv 之后，子进程就是客户端请求的服务器。如果 execv 返回，函数通知客户端不能启动请求的服务并退出。

函数 parsetab 如图 3.12 所示。它执行对 tcpmux.conf 文件简单但冗长的遍历。tcpmux.conf 文件的格式为：

service_name path arguments ...

tcpmux.c

```

1 static void parsetab( void )
2 {
3     FILE *fp;
4     servtab_t *stp = service_table;
5     char *cp;
6     int i;
7     int lineno;
8     char line[ MAXLINE ];

9     fp = fopen( CONFIG, "r" );
10    if ( fp == NULL )
11        error( 1, errno, "unable to open %s", CONFIG );
12    lineno = 0;
13    while ( fgets( line, sizeof( line ), fp ) != NULL )
14    {
15        lineno++;
16        if ( line[ strlen( line ) - 1 ] != '\n' )
17            error( 1, 0, "line %d is too long\n", lineno );
18        if ( stp >= service_table + NSERVTAB )

```

```

19         error( 1, 0, "too many entries in tcpmux.conf\n" );
20         cp = strchr( line, '#' );
21         if ( cp != NULL )
22             *cp = '\0';
23         cp = strtok( line, "\t\n" );
24         if ( cp == NULL )
25             continue;
26         if ( *cp == '+' )
27         {
28             stp->flag = TRUE;
29             cp++;
30             if ( *cp == '\0' || strchr( "\t\n", *cp ) != NULL )
31                 error( 1, 0, "line %d: white space after '+'\n",
32                         lineno );
33         }
34         stp->service = strdup( cp );
35         if ( stp->service == NULL )
36             error( 1, 0, "out of memory\n" );
37         cp = strtok( NULL, "\t\n" );
38         if ( cp == NULL )
39             error( 1, 0, "line %d: missing path name (%s)\n",
40                   lineno, stp->service );
41         stp->path = strdup( cp );
42         if ( stp->path == NULL )
43             error( 1, 0, "out of memory\n" );
44         for ( i = 0; i < MAXARGS; i++ )
45         {
46             cp = strtok( NULL, "\t\n" );
47             if ( cp == NULL )
48                 break;
49             stp->args[ i ] = strdup( cp );
50             if ( stp->args[ i ] == NULL )
51                 error( 1, 0, "out of memory\n" );
52         }
53         if ( i >= MAXARGS && strtok( NULL, "\t\n" ) != NULL )
54             error( 1, 0, "line %d: too many arguments (%s)\n",

```

```

55         lineno, stp->service );
56     stp->args[ i ] = NULL;
57     stp++;
58 }
59 stp->service = NULL;
60 fclose ( fp );
61 }
```

*tcpmux.c*图 3.12 **parsetab** 函数

函数 `readcrlf`, 如图 3.13 所示, 每次从它的输入中读取一个字节。尽管这样做的效率不高, 但是它保证了程序读取的仅仅是客户端数据的第一行。第一行之后的所有数据是由程序正在启动的服务器来处理的, 如果程序执行的缓冲区读而客户端发送了多行数据, 就有可能导致一些服务器数据由 `tcpmux` 读取并丢失。

注意 `readcrlf` 仅接受由新行字符结束的行。这和 Robustness Principle[Postel 1981a]是一致的, 它是这样描述的: “对接收的数据要尽量开明, 对发送的数据要尽量保守。”在两种情况下, <CR><LF>或<LF>可以省略。

函数 `readcrlf` 和 `read`、`readline`、`readn` 以及 `readvrec` 有相同的定义:

```

# include "etc.h"

int readcrlf( SOCKET s, char *buf, size_t len );

>Returns:number of bytes read or -1 on error
```

library/readcrlf.c

```

1 int readcrlf( SOCKET s, char *buf, size_t len )
2 {
3     char *bufx = buf;
4     int rc;
5     char c;
6     char lastc = 0;
7
8     while ( len > 0 )
9     {
10         if ( ( rc = recv( s, &c, 1, 0 ) ) != 1 )
11             {
```

```

11      /*
12       * If we were interrupted, keep going,
13       * otherwise, return EOF or the error.
14      */
15
16      if ( rc < 0 && errno == EINTR )
17          continue;
18      return rc;
19
20      if ( c == '\n' )
21      {
22          if ( lastc == '\r' )
23              buf--;
24          *buf = '\0';           /* don't include <CR><LF> */
25          return buf - bufx;
26      }
27
28      *buf++ = c;
29      lastc = c;
30      len--;
31
32  }

```

library/readcrlf.c

图 3.13 readcrlf 函数

最后看一看 reaper 函数（如图 3.14 所示）。当 tcpmux 启动的服务器终止时，UNIX 发送一个 SIGCHLD 信号给它的父进程（tcpmux）。这导致调用 reaper 函数以及依次调用 waitpid 获取已经终止的子进程的状态。这在 UNIX 系统中是必须的，因为子进程可以在终止时返回状态给父进程（例如，exit 的参数）。

一些 UNIX 系统也可以返回其他信息。例如，BSD 派生的系统可以返回正在停止运行的进程以及它的子进程的资源的概括。所有的 UNIX 系统至少返回一个指示，说明进程是否因为调用 exit 而停止运行以及退出时的状态如何，或者说明进程是否由信号终止以及是什么信号。

直到父进程通过调用 wait 或 waitpid 收集这些状态信息，UNIX 才应当一直保留子进程中包含终止状态信息的资源部分。已经终止但其父进程仍然没有收集状态信息的子进程称

为死 (defunct) 或僵 (zombie) 进程。

```
tcpmux.c
1 void reaper( int sig )
2 {
3     int waitstatus;
4
5     while ( waitpid( -1, &waitstatus, WNOHANG ) > 0 )
6         {};
7 }
```

tcpmux.c

图 3.14 reaper 函数

通过在 `tcpmux.conf` 文件中创建下面一行：

```
+rlnum /usr/home/jcs/rlnumd rlnumd
```

可以对 `tcpmux` 进行测试。然后在不支持 `tcpmux` 的 `sparc` 机器上启动 `tcpmux`，并从 `bsd` 机器上 `telnet` 到 `tcpmux`：

```
sparc: # tcpmux
bsd: $ telnet sparc tcpmux
Trying 10.9.200.201...
Connected to sparc.
Escape character is '^]'.
xlnum
+OK
hello
1: hello
world
2: world
^]
telnet> quit
Connection closed.
bsd: $
```

小结

在许多系统中可获得的 TCPMUX 服务可以帮助解决为服务器分配已知端口的问题。本节实现了自己版本的 `tcpmux` 守候程序，没有该程序的系统可以通过使用它来使用它提供的服务。

技巧 19 考虑使用两个 TCP 连接

在许多应用程序中，让不同的进程或线程处理 TCP 连接的读和写是十分方便的。这在 UNIX 社区中特别实用，在 UNIX 中通常是派生一个子进程处理 TTY 连接的写（说），而父进程是处理读的。

典型的情况如图 3.15 所示，它描述了一个终端模拟。父进程花费大多数时间在 TTY 连接的读操作阻塞上。如果数据在 TTY 连接上可获得，父进程就读取数据并把它输出到屏幕上，有时执行一些重新格式化操作。子进程花费大多数时间在读取键盘输入阻塞上。当键盘上的数据可获得时，子进程处理键盘映射并把数据写入到 TTY 连接中。

这是一个十分吸引人的策略，因为它自动地处理键盘和 TTY 数据两种情况，它把键盘映射和屏幕重新格式化逻辑划分为不同的模块，而且它提供了比混合代码概念上更简单的程序。实际上，在引入 `select` 机制之前，该技术是处理多输入的唯一实用方法。

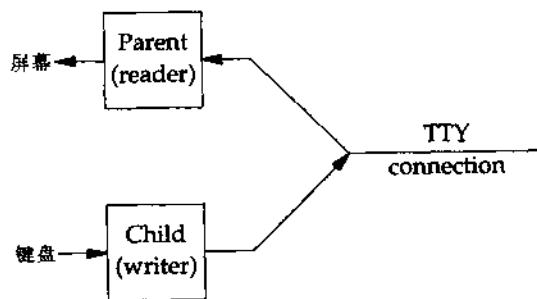


图 3.15 服务于一个 TTY 连接的两个进程

单一连接的体系结构

注意在图 3.15 中，如果用“TCP 连接”代替“TTY 连接”的话，什么也不会改变。因此相同的技术可以而且也经常是用于处理网络连接。同时也应当注意使用两个线程而不是两个进程同样也不会改变图中显示的状况，因此同样的方法也可以用于使用线程环境中。

然而，这里有一个隐含条件。使用一个 TTY 连接，任何发生在写操作上的错误都从 `write` 调用本身返回，反之使用 TCP，错误通常是由读操作返回。在多进程的

情况下，让读进程来通知写进程错误是十分困难的。特别是，如果对等方应用程序已经终止运行，那么就只有读进程受到通知，而读进程一定会接着通知写进程。

让我们改变一下我们的视角，假设有一个应用程序通过 TCP 连接来接收和发送消息到一个外部系统。消息不同步发送和接收——也就是说，并不是所有进来的消息都会产生一个响应，而且并不是所有出去的消息都是对进来消息的响应。如果进一步假设消息不得不在进入和离开应用程序时重新格式化，就很有必要使用如图 3.16 所示的多进程应用程序。

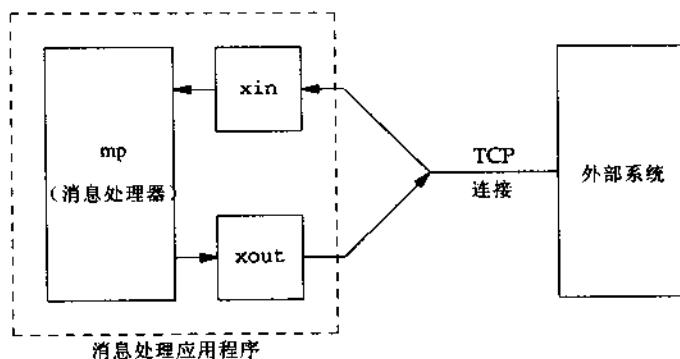


图 3.16 带一个 TCP 连接的传递消息的应用程序

图中 `xin` 进程从外部系统读取数据，把它积累到消息上，并在必要时重新格式化，并传递重新格式化消息给主消息处理进程处理。同样地，`xout` 进程重格式化出去的消息为外部系统所需的格式并把数据写入到 TCP 连接中。主进程 `mp` 处理重新格式化的进来的消息并产生出去的消息。我们让组成应用程序的三个进程中的 IPC 机制为非指定的，但是它可以是管道、共享内存、消息队列或任何几个其他的类型。请参阅 [Stevens 1999]，了解这些情况的完全处理。这类应用程序的一个现实世界中的例子是网关，它在利用 TCP 通信的系统和使用其他协议通信的系统之间传输消息。

如果扩展这个例子为包含其他需要不同格式消息的外部系统，就可以看出这个方法的功能是多么的强大。每一个外部主机都有它自己的通信进程组来处理它自己的消息，这使系统在概念上更简单，允许在一台外部主机上所做的改变不影响其他的主机，而且仅仅通过为每一台主机启动所需的通信进程，也允许很轻松地为给定的外部主机集配置系统。

然而，确实存在前面间接提及的问题：写进程不能从它的写操作中接收错误消息。而且，有可能应用程序一定要确认外部系统确实接收到了消息，因此必须实现某种正面的确认，如技巧 9 中所述的那样。这意味着必须在 `xin` 和 `xout` 之间建立一个独立的通信频道，或者 `xin` 必须发送错误指示以及对 `mp` 的确认，接着 `mp` 必须把它们发送到 `xout`。这两个选项使涉及交换的所有处理变得复杂。

当然，我们可以抛弃多进程体系结构并把所有的事情都归入单一的进程，也许可以使用 `select` 来处理多路消息。然而，这意味着必须牺牲前面讨论的灵活性和概念上的简单性。

本节中的其余部分分析了保持图 3.16 中灵活性的另一种体系结构，但是允许每一个通

信进程完全处理它自己的 TCP 连接。

» 双连接体系结构

尽管 `xin` 和 `xout` 和图 3.16 中的外部连接共享 TCP 连接，但是它们之间共享有关连接的状态信息是十分困难的。而且 `xin` 和 `xout` 对待连接就像它就是单向的一——也就是说，数据好像是单方向流动的。它们必须做一些处理来防止 `xout` “偷窃” `xin` 中的输入数据以及防止 `xin` 破坏 `xout` 发送的数据。这些事实会导致前面所讨论的问题。

这些问题的解决方案是对外部系统维护两个连接，一个为 `xin` 服务另一个为 `xout` 服务。通过这个改变，系统体系结构成为图 3.17 所描述的那样：

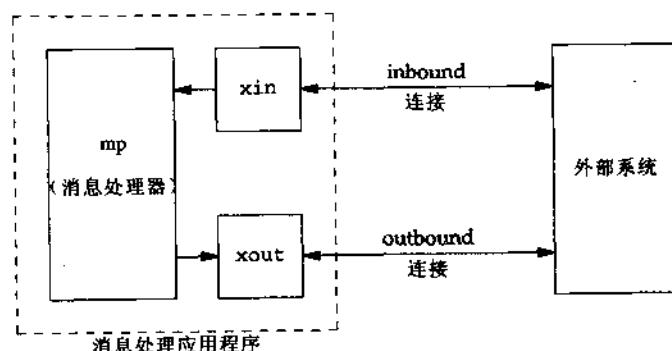


图 3.17 使用两个 TCP 连接的传递消息的应用程序

如果系统不需要应用程序级别的 ACK，该体系结构中获得最大好处的是 `xout`，现在它可以对等方直接获得错误消息以及 EOF 字符了。另一方面，`xout` 现在必须稍微复杂一点，这是因为它必须有一个读操作的挂起来收集这些事件。幸运的是，这可以通过使用 `select` 系统调用轻松做到。

为了试验这种体系结构，下面编写了一个简单的 `xout` 进程，它从标准输入中读取数据并把数据写入 TCP 连接。该程序显示在图 3.18 中，使用 `select` 来保持连接上的一个读操作挂起，而它所希望的输入仅仅是 EOF 或错误指示。

xout1.c

```

1 #include "etcp.h"

2 int main( int argc, char **argv )
3 {
4     fd_set allreads;
5     fd_set readmask;
6     SOCKET s;
  
```

```
7  int rc;
8  char buf[ 128 ];

9  INIT();
10 s = tcp_client( argv[ 1 ], argv[ 2 ] );
11 FD_ZERO( &allreads );
12 FD_SET( s, &allreads );
13 FD_SET( 0, &allreads );
14 for ( ;; )
15 {
16     readmask = allreads;
17     rc = select( s + 1, &readmask, NULL, NULL, NULL );
18     if ( rc <= 0 )
19         error( 1, rc ? errno : 0, "select returned %d", rc );

20     if ( FD_ISSET( 0, &readmask ) )
21     {
22         rc = read( 0, buf, sizeof( buf ) - 1 );
23         if ( rc < 0 )
24             error( 1, errno, "read failure" );
25         if ( send( s, buf, rc, 0 ) < 0 )
26             error( 1, errno, "send failure" );
27     }

28     if ( FD_ISSET( s, &readmask ) )
29     {
30         rc = recv( s, buf, sizeof( buf ) - 1, 0 );
31         if ( rc == 0 )
32             error( 1, 0, "server disconnected\n" );
33         else if ( rc < 0 )
34             error( 1, errno, "recv failure" );
35         else
36         {
37             buf[ rc ] = '\0';
38             error( 1, 0, "unexpected input [%s]\n", buf );
39         }

```

```

40      }
41  }
42 }
```

xout1.c

图 3.18 仅仅读取错误指示和 EOF 字符的“写入者”

✓ 初始化

9~13 程序执行通常的初始化，调用 `tcp_client` 来建立一个连接，并设置为在标准输入或刚建立的 TCP 连接上进行选择。

✓ 处理标准输入事件

20~27 如果程序在标准输入上获得输入，就通过 TCP 连接发送给对等方。

✓ 处理套接字事件

28~40 如果程序在套接字上获得输入事件，就检查 EOF 字符或错误消息。因为程序不会从连接获得任何数据，所以如果检查到了 EOF 字符或错误消息，程序就输出一个诊断消息并退出。

通过使用图 2.53 中的 `keep` 程序作为外部系统而把简单的 shell 脚本（图 3.17 中的 `mp`）作为消息处理程序，就可以演示程序 `xout1`。shell 脚本仅简单地响应消息“Message”和每秒钟在标准输出上的记数：

```

MSGNO=1
while true
do
    echo message $MSGNO
    sleep 1
    MSGNO=`expr $MSGNO + 1`
done
```

注意在这个例子中，`xout1` 使用管道来作为 IPC 机制。这意味着 `xout1` 不能移植到 Windows 环境中，这是因为 `select` 调用仅在 Windows 环境下正常工作。本来可以为 IPC 使用 TCP 或 UDP，但是这需要消息处理程序更为精细。

为了试验 `xout1`，首先在一个窗口中启动外部系统 `keep`，而在另一个窗口中启动消息处理程序：

```
bsd: $ keep 9000
message 1
```

```
bsd: $ mp | xout1 localhost 9000
```

```

message 2
message 3
message 4
^C      "External System"terminated
bsd: $                                         xout1: server disconnected
                                                Broken pipe
                                                bsd: $

```

“Broken pipe”消息来自 mp 脚本。当 xout1 终止时，它和脚本之间的管道就会关闭。当脚本试图写下一行时，写操作失败并且脚本以“Broken pipe”错误终止。

如果外部系统和消息处理应用程序之间需要 ACK，就会发生一个有趣的情形。在这个例子中，xin 和 dout 都需要改变（假设两个方向都要 ACK——如果程序仅仅关心发送到外部系统的消息，就没有必要改变 xin）。下面开发了一个写操作（xout）进程的例子。读操作进程和前面的一样。

新的写操作进程必须解决和技巧 10 中 heartbeat 函数相同类型的问题。程序发送一个消息之后，必须在计时器终止之前从对等方接收一个 ACK 消息。如果计时器终止了，程序必须发起某种类型的错误恢复。在这个例子中，程序仅仅是退出。

新写操作程序 xout2 的策略是不从标准输入接受任何其他消息，直到外部系统确认通过 TCP 连接发送的最后消息为止。使用技巧 20 中描述的一般分时机制的更复杂的方法也可以，后面将分析这个具体实现，但是对于多数应用程序来说，这里采用的简单方法已经足够了。程序 xout2 如图 3.19 所示。

xout2.c

```

1 #include "etcpc.h"

2 #define ACK      0x6      /* an ACK character */

3 int main( int argc, char **argv )
4 {
5     fd_set allreads;
6     fd_set readmask;
7     fd_set sockonly;
8     struct timeval tv;
9     struct timeval *tvp = NULL;
10    SOCKET s;
11    int rc;
12    char buf[ 128 ];
13    const static struct timeval T0 = { 2, 0 };

```

```

14 INIT();
15 s = tcp_client( argv[ 1 ], argv[ 2 ] );
16 FD_ZERO( &allreads );
17 FD_SET( s, &allreads );
18 sockonly = allreads;
19 FD_SET( 0, &allreads );
20 readmask = allreads;
21 for ( ; ; )
22 {
23     rc = select( s + 1, &readmask, NULL, NULL, &tv );
24     if ( rc < 0 )
25         error( 1, errno, "select failure" );
26     if ( rc == 0 )
27         error( 1, 0, "message timed out\n" );
28
29     if ( FD_ISSET( s, &readmask ) )
30     {
31         rc = recv( s, buf, sizeof( buf ), 0 );
32         if ( rc == 0 )
33             error( 1, 0, "server disconnected\n" );
34         else if ( rc < 0 )
35             error( 1, errno, "recv failure" );
36         else if ( rc != 1 || buf[ 0 ] != ACK )
37             error( 1, 0, "unexpected input [%c]\n", buf[ 0 ] );
38         tvp = NULL;           /* turn timer off */
39         readmask = allreads; /* and stdin on */
40     }
41
42     if ( FD_ISSET( 0, &readmask ) )
43     {
44         rc = read( 0, buf, sizeof( buf ) );
45         if ( rc < 0 )
46             error( 1, errno, "read failure" );
47         if ( send( s, buf, rc, 0 ) < 0 )
48             error( 1, errno, "send failure" );
49         tv = T0;           /* reset timer */

```

```

48         tvp = &tv;           /* turn timer on */
49         readmask = sockonly; /* and stdin off */
50     }
51 }
52 }
```

xout2.c

图 3.19 处理 ACK 消息的“写入者”

／ 初始化

14~15 这是 TCP 客户端的标准初始化。

16~20 程序设置两个 select 掩码——一个接收来自标准输入和 TCP 套接字的事件，另一个仅接收来自套接字的事件。程序在发送数据给 TCP 连接后使用第二个掩码 `sockonly`，所以程序在接收外部系统的确认之前不会读取任何来自标准输入的其他数据。

／ 处理计时器事件

26~27 如果 select 调用超时了，程序就不会及时获得确认。因此程序输出诊断消息并退出。

／ 处理套接字事件

28~39 当程序接收到来自套接字的读操作事件时，就检查该消息是否为错误或 EOF 字符，如果是的话就退出，如图 3.18 中所述。如果程序接收到数据，就检查它是否是一个单字符以及该字符是否为 ACK。如果程序忽略这个测试，最后的消息就被确认，所以程序通过设置 `tvp` 为 NULL 来关闭计时器，通过设置 `readmask` 检查标准输入和套接字程序再次允许从标准输入接收数据。

／ 处理标准输入事件

40~46 当程序获得一个标准输入事件时，跟通常一样检查错误消息或 EOF 字符。如果 `read` 成功返回，程序把数据写入到 TCP 连接中。

47~50 因为程序已经写入数据到外部主机，所以程序希望获得一个返回的 ACK 消息。程序通过设置 `tv` 和由 `tvp` 指向它来设置和启动计时器。最后，程序通过设置 `readmask` 为 `sockonly` 禁止标准输入事件，以使 `select` 仅认可来自 TCP 连接的事件。

通过加入下面两行可以试验 `xout2`:

```

if ( send( s1, "\006", 1, 0 ) < 0 )          /* \006 == ACK */
    error ( 1, errno, "send failure" );
```

在 `keep.c` (如图 2.53 所示) 中第 24 行的写操作之前，如果做 `xout1` 中一样的测试的话，

除了 `xout2` 在没有接收到来自对等方的 ACK 消息就退出之外，获得的结果和前面的一模一样。

技巧 21 中还会对这个例子重新进行分析，技巧 21 中使用了一个通用的分时机制，允许在任何时候处理标准输入请求，而不是在等待来自对等方的 ACK 消息时禁止它们。

» 小结

本节探讨了在对等应用程序之间使用两个连接的思想。从中可以看到甚至在连接的读操作和写操作在不同的进程中执行时都允许监测连接的状态。

技巧 20 考虑使应用程序事件驱动（1）

本节和下节将分析为 TCP/IP 编程使用事件驱动技术。作为这个分析的一部分，本节开发了一个通用的分时机制，允许在一定的时间间隔后指定一个必须发生的事情，而且使该事件在指定的时间上异步发生。本节将研究计时器机制的具体实现，然后在技巧 21 中应用它，在技巧 21 中还会重新分析技巧 19 中的双连接体系结构。

事件驱动的应用程序和非事件驱动的应用程序之间的区别可以由两个应用程序 `hb_client2`（如图 2.49，图 2.50 所示）和 `tcprw`（如图 2.42 所示）很好地说明。在 `tcprw` 中，控制的流动是顺序的：首先从标准输入读取一行并把它发送到对等方，然后从对等方接收到应答并写入到标准输出。注意当程序正在等待来自标准输入的输入时程序不能接收对等方的数据。正如前面所看到的那样，这意味着可能错过对等方已经终止并发送了一个 EOF 字符的情况。同时，当程序正在等待对等方的应答时，也不能从标准输入读取任何数据。这意味着从用户的观点来看应用程序是没有响应的，如果对等方主机在对等方应用程序崩溃之前响应的话程序就会挂起。

用 `tcprw` 和 `hb_client2` 做比较，`hb_client2` 可以在任何时候接收两个连接的数据，或者是发生了超时。这些“事件”都没有必要等待其他事件——这是称它为事件驱动的原因。

注意扩展 `hb_client2` 为处理更多的连接或输入的程序是很容易的，`select` 机制使这个成为可能。它允许程序在任何连接或输入已经准备好时立即在几个事件上阻塞并返回。在 UNIX 系统下，这种机制或它的 SysV 同类 `poll` 是在非线程环境中做这种事情的唯一的有效方法。

直到目前为止，传统的看法是移植方面的考虑要求我们宁愿选用 `select` 而不是 `poll`，因为 Winsock 和几乎所有的现代 UNIX 系统都支持 `select`，而 `poll` 仅在大多数的 SysV 具体实现中可以找到。

然而，因为它可以很好地扩展到大数量的描述符，一些支持多个并发连接的大服务器应用程序（如 Web 服务器）使用的是 `poll` 机制。这是因为 `select` 受限于固定数量的描述符。这个限制通常是 1024，而且有可能更小。例如，在基于 FreeBSD 的 bsd 系统上，默认为 256，改变这个默认值需要重新编译系统，虽然可以这样做但是十分麻烦。即使是仅仅增加这个限制而重新编译系统，也没有消除这个限制。另一方面，`poll` 机制没有对它可以处理的描述符的数据进行限制。

另一个需要考虑的问题是效率。当有大量的描述符时，典型的 select 具体实现会变得效率十分低下。请参阅[Banga and Mogul 1998]，了解有关这方面的更多知识。（该论文同时提供了一个用 LAN 上获得的结果外推至 WAN 上以期望获得希望的性能的危险的一个例子，如技巧 12 中讨论的那样。）当程序正在等待大量描述符上的几个事件时，这个性能问题就变得特别尖锐——也就是说，第一个参数 maxfd 很大，但仅有几个描述符用 FD_SET 注册。这是因为内核必须检验每一个可能的描述符（0...maxfd）来确定应用程序是否希望在一个事件上等待它。poll 调用使用描述符数组来告诉内核应用程序感兴趣的事件是什么，因此不会发生这个问题。

虽然使用 select 和 poll 使程序可以轻松地处理多路的多 I/O 事件，但是处理多计时器也很困难，因为这些调用仅提供一个单一的超时值。为了解决这个问题，为了事件驱动的程序提供一个更灵活的环境，我们开发了 select 过程的一个变种称作 tselect。虽然和 tselect 关联的 timeout 和 untimeout 调用以相同名称的 UNIX 内核计时器过程模拟，但是它们是在用户空间实现的，并且依靠 select 提供 I/O 多路复用和计时器。

正如前面说明的那样，和 tselect 函数关联的有三个调用。第一个是 tselect 本身，它和使用 select 多路复用 I/O 事件有相同的用法，唯一的区别是 tselect 没有 timeout 参数（select 的第五个参数）。相应地，所有的计时器事件用 timeout 调用指定，通过该调用用户可以指定计时器应当运行多长时间以及当它终止时应当采取什么操作。untimeout 调用用于在计时器超时之前取消它。

三个调用总结如下：

```
#include "etcp.h"

int tselect( int maxfd, fd_set *rdmask, fd_set *wmask,
             fd_set *exmask );
    Returns:number of ready events, 0 if remaining events, -1 error

unsigned int timeout( void ( *handler )( void * ), void *arg, int ms );
    Returns:timer ID to be used in the untimeout call

void untimeout( unsigned int timerid );
```

如果关联到 timeout 调用的计时器终止了，就用 arg 参数中指定的参数来调用由 handler 参数指定的函数。这样，为了让 retransmit 函数在 1.5 秒之后用整型参数 sock 调用，应当首先调用：

```
timeout ( retransmit, (void *)sock, 1500 );
```

之后跟着一个 tselect 调用。超时值 ms 以毫秒指定，但是应当意识到系统时钟的间隔可能不太精确，UNIX 系统中的典型值是 10ms，因此不能期望计时器比该值更精确。

后面将会分析使用 tselect 的几个例子，但是首先分析一下它的具体实现。tevent_t 结构的定义和全局声明在图 3.20 中给出。

lib/tselect.c

```

1 #include "etcp.h"

2 #define NTIMERS 25

3 typedef struct tevent_t tevent_t;
4 struct tevent_t
5 {
6     tevent_t *next;
7     struct timeval tv;
8     void ( *func )( void * );
9     void *arg;
10    unsigned int id;
11};

12 static tevent_t *active = NULL;      /* active timers */
13 static tevent_t *free_list = NULL;   /* inactive timers */

```

lib/tselect.c

图 3.20 tselect 的全局数据

✍ 声明

2 NTIMERS 定义指定了程序一次应当分配多少计时器。因为初始时没有计时器，所以 timeout 的第一次调用将导致分配 NTIMERS 个计时器。如果这些计时器都处于使用之中而进行了第二次 timeout 调用，就分配另外的 NTIMERS 个计时器。

3-11 每一个活动的计时器由 tevent_t 结构表示，这些结构由 next 域连接在一起。程序使用 tv 域来保存计时器终止之时的时间。func 和 arg 域是计时器处理函数和参数，当计时器终止时就调用这个函数。最后，程序把每一个活动的计时器的 ID 放置到 id 域中。

12 活动的计时器以终止时间的顺序在活动计时器列表中连接在一起。（模块）全局变量 active 指向列表中的第一个计时器。

13 当前不活动的计时器在一个自由列表中连接在一起。当 timeout 过程需要一个新的计时器时，它就从自由列表中弹出一个。free_list（模块）全局指针指向自由列表的顶端。

下面将分析 timeout 和计时器分配过程（图 3.21）。

lib/select.c

```
1 static tevent_t *allocate_timer( void )
2 {
3     tevent_t *tp;
4
5     if ( free_list == NULL ) /* need new block of timers? */
6     {
7         free_list = malloc( NTIMERS * sizeof( tevent_t ) );
8         if ( free_list == NULL )
9             error( 1, 0, "couldn't allocate timers\n" );
10        for ( tp = free_list;
11              tp < free_list + NTIMERS - 1; tp++ )
12            tp->next = tp + 1;
13        tp->next = NULL;
14    }
15    tp = free_list;           /* allocate first free */
16    free_list = tp->next;    /* and pop it off list */
17    return tp;
18
19 unsigned int timeout( void ( *func )( void * ), void *arg, int ms )
20 {
21     tevent_t *tp;
22     tevent_t *tcur;
23     tevent_t **tprev;
24     static unsigned int id = 1;           /* timer ID */
25
26     tp = allocate_timer();
27     tp->func = func;
28     tp->arg = arg;
29     if ( gettimeofday( &tp->tv, NULL ) < 0 )
30         error( 1, errno, "timeout: gettimeofday failure" );
31     tp->tv.tv_usec += ms * 1000;
32     if ( tp->tv.tv_usec > 1000000 )
33     {
34         tp->tv.tv_sec += tp->tv.tv_usec / 1000000;
```

```

33     tp->tv.tv_usec *= 1000000;
34 }
35 for ( tprev = &active, tcur = active;
36     tcur && !timercmp( &tp->tv, &tcur->tv, < ); /* XXX */
37     tprev = &tcur->next, tcur = tcur->next )
38 { ;
39 *tprev = tp;
40 tp->next = tcur;
41 tp->id = id++;           /* set ID for this timer */
42 return tp->id;
43 }

```

lib/select.c

图 3.21 timeout 和 allocate_timer 过程

allocate_timer

4~13 allocate_time 过程被 timeout 调用来分配一个自由计时器。如果自由列表是空的，程序就会在内存堆里为 NTIMERS 个 tevent_t 结构分配足够的内存，并把它们连接在一起。

14~16 程序弹出自由列表中的第一个计时器并把它返回给调用者。

timeout

23~26 程序分配一个计时器并以传递进来的参数填充 func 和 arg 域。

37~34 程序通过增加 ms 参数到当前时间来计算计时器终止的时间。程序存储结果到 tv 域。

35~38 程序向下搜索活动计时器列表，直到找到该计时器的正确位置。“正确位置”是指它前面的所有计时器的终止时间都比它小或等于它的终止时间，而它后面的所有计时器都比它的终止时间大。图 3.22 列出了这个位置并显示了扫描期间 tcur 和 tprev 变量的用法。图 3.22 往列表中插入一个 t_{new} 新计时器， t_{new} 满足条件 $t_0 \leq t_i \leq t_{new} \leq t_2$ 。虚线框住的以 t_{new} 为标签的结构显示了将要插入新计时器的位置。第 36 行中使用有点难于理解的 timercmp 宏是因为定义在 winsock2.h 中的版本没有用并且不支持 “ \geq ” 操作符。

39~42 程序在适当的位置插入新计时器，分配给它一个计时器 ID，并把该 ID 返回给调用者。程序返回一个 ID 而不是 tevent_t 结构是为了避免竞争条件。当一个计时器终止时，tevent_t 结构返回到自由列表的开头。如果分配了一个新的计时器，它就使用该结构。如果应用程序现在试图取消第一个计时器而且程序使用了结构的地址，取消的就是第二个计时器了。通过使用 ID，可以避免这个问题。

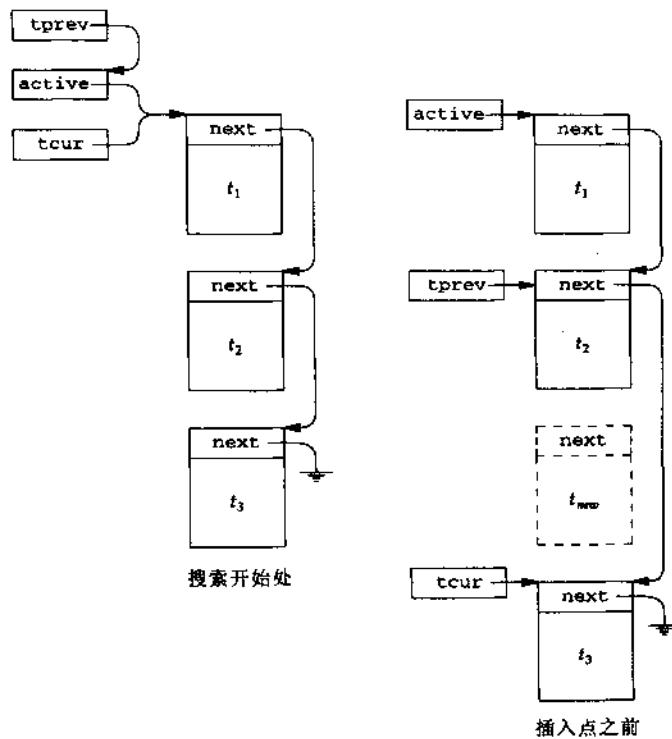


图 3.22 搜索插入点之前以及之后的活动计时器列表

图 3.21 中结尾返回的计时器 ID 由图 3.23 列出的 untimout 函数使用。

lib/tselect.c

```

1 void untimout( unsigned int id )
2 {
3     tevent_t **tprev;
4     tevent_t *tcur;
5
6     for ( tprev = &active, tcur = active;
7           tcur && id != tcur->id;
8           tprev = &tcur->next, tcur = tcur->next )
9     { ; }
10    if ( tcur == NULL )
11        error( 0, 0,
12                "untimout called for non-existent timer (%d)\n", id );
13    return;

```

```

14 }
15 *tprev = tcur->next;
16 tcur->next = free_list;
17 free_list = tcur;
18 }

```

lib/tselect.c

图 3.23 untimout 函数

✍ 查找计时器

5~8 程序搜索活动的计时器列表来查找具有 id 值的 ID 的计时器。该循环和 timeout (图 3.21) 中的相似。

9~14 如果程序试图取消的计时器不在列表中，程序输出诊断信息并返回。

✍ 取消计时器

15~17 程序通过从活动列表中断开 tevent_t 结构来取消计时器，并把该计时器返回到自由列表中。

计时器过程集最后的一个函数是 tselect 本身 (图 3.24)。

lib/tselect.c

```

1 int tselect( int maxp1, fd_set *re, fd_set *we, fd_set *ee )
2 {
3     fd_set rmask;
4     fd_set wmask;
5     fd_set emask;
6     struct timeval now;
7     struct timeval tv;
8     struct timeval *tvp;
9     tevent_t *tp;
10    int n;
11
12    if ( re )
13        rmask = *re;
14    if ( we )
15        wmask = *we;
16    if ( ee )
17        emask = *ee;
18    for ( ; ; )

```

```
18  {
19      if ( gettimeofday( &now, NULL ) < 0 )
20          error( 1, errno, "tselect: gettimeofday failure" );
21      while ( active && !timercmp( &now, &active->tv, < ) )
22      {
23          active->func( active->arg );
24          tp = active;
25          active = active->next;
26          tp->next = free_list;
27          free_list = tp;
28      }
29      if ( active )
30      {
31          tv.tv_sec -= active->tv.tv_sec - now.tv_sec;;
32          tv.tv_usec = active->tv.tv_usec - now.tv_usec;
33          if ( -tv.tv_usec < 0 )
34          {
35              tv.tv_usec += 1000000;
36              tv.tv_sec--;
37          }
38          tvp = &tv;
39      }

40      else if ( re == NULL && we == NULL && ee == NULL )
41          return 0;
42      else
43          tvp = NULL;
44      n = select( maxp1, re, we, ee, tvp );
45      if ( n < 0 )
46          return -1;
47      if ( n > 0 )
48          return n;
49      if ( re )
50          *re = rmask;
51      if ( we )
52          *we = wmask;
```

```

53     if ( ee )
54         *ee = emask;
55 }
56 }

```

lib/tselect.c

图 3.24 tselect 函数

✓ 保存事件屏蔽

11~16 因为单一的 tselect 调用可能产生几个 select 调用，程序保存传递给 select 的事件屏蔽。

✓ 分派计时器事件

19~28 虽然活动列表中的第一个 tevent_t 结构具有比当前时间小或相等的终止时间，但是程序还是调用该计时器的处理程序，从活动列表中弹出该结构，并把它返回给自由列表。正如图 3.21 显示的那样，在一些系统中的不好的 timercmp 版本需要特殊的 timercmp 调用。

✓ 计算下一个事件的时间

29~39 如果活动列表中仍然有计时器，程序就计算当前时间和计时器终止时间之差，程序将该值传递给 select。

40~41 如果没有其他的计时器而且没有需要等待的其他 I/O 事件，tselect 就返回。注意程序返回 0，表明没有事件需要等待，它和 select 的返回值 0 的意思不一样。

42~43 如果没有更多的计时器事件，但是系统存在 I/O 事件，那么程序设置 tvp 为 NULL 告诉 select 不能超时终止。

✓ 调用 select

44~48 程序调用 select 来等待一个事件。如果 select 返回错误，那么程序就把它返回给应用程序。如果 select 返回正值，就说明一个或更多的 I/O 事件已经准备好了，程序返回事件的个数给应用程序。因为程序用应用程序的事件屏蔽来调用 select，所以他们已经设置好了。

49~54 如果 select 返回 0，那么至少一个事件已经准备好了。因为 select 将清空应用程序事件屏蔽中的事件，所以程序在继续之前在循环的开始处（程序在这个地方分派计时器）存储它们。

程序使用直接的线性搜索来寻找计时器的插入和删除点。虽然这种方法对小规模或中等规模数量的计时器还可以，但是它不能扩展到数量很多的计时器上，因为插入和删除的时间复杂度是 $O(n)$ (n 个计时器) 个操作（分派计时器事件是一个 $O(1)$ 操作）。其他可能的实现是堆[Sedgewick 1998]，它的插入、删除和分派的事件复杂度是 $O(\log n)$ ，以及哈希

定时轮 (timing wheel) [Verghese and Lauck 1997]，它的时间复杂度对三种操作的效率可以为 O(1)。

注意因为 tselect 中不需要程序中有一个等待的 I/O 事件，所以可以使用它作为严格的定时机制，它比使用 sleep 有两个优势：

(1) 在 UNIX 系统中，sleep 机制的粒度不太精确。sleep 时间间隔必须是以秒计的整数。Windows 版本的 sleep 没有这个限制，而且许多 UNIX 具体实现有许多具有更精确间隔的计时器机制，但是这些机制并不是在所有的具体实现里都有。我们宁愿采用可以在大多数平台上运行的具有较好间隔的时间机制。因为这个原因，通常在 UNIX 里使用 select 调用作为具有较好间隔的时间机制。

(2) 因为需要记录，所以为多计时器使用 sleep 或简单的 select 在管理时很困难。tselect 的优势是它已经为我们做好了记录。

不幸的是，使用 tselect 作为计时器在 Windows 平台下不能正常工作。Winsock API 规范[WinSock Group 1997]称为计时器使用 select 为“无法原谅的缺陷”。也许你会感到诧异，当一个系统调用在已经发行的规范上不能执行时缺陷究竟在哪里？但是程序员仍然应当遵循这个建议。在 Windows 平台下，我们将继续使用 tselect 函数，只是在使用它的时候指定一个 I/O 事件。

小结

本节讨论了使应用程序事件驱动的好处，同时本节开发了一个通用的计时器实用程序，该程序可以提供数量不受限制的计时器。

技巧 21 考虑使应用程序事件驱动（2）

本节将继续技巧 20 的讨论。本节将演示在一个应用程序中使用 tselect 函数，并分析事件驱动应用程序的其他一些方面。本演示将继续使用技巧 19 中的两个连接的体系结构。

如果我们再次看看 xout2 的话（图 3.19），就可以看出它不是事件驱动的。一旦给对方发送一条消息，直到已经接收到消息的 ACK 才能从标准输入接收其他输入，这是为了防止新消息重置计时器。如果在第一个消息确认之前为后面的消息重新启动计时器，就不知道第一个消息是否被确认。

当然，问题的存在是因为 xout2 仅有一个计时器，所以不能在同一时刻为多个消息计时。通过使用 tselect，就可以多路复用 select 提供的计时器。

为了设置将要遵循的规则，假设技巧 19 的外部系统是某种类型的网关，它就会转发消息给使用不可靠协议的第三方系统。例如，进一步假设网关本身不提供消息是否成功递交的指示，它就可能发送数据报给一个无线网络。它仅仅转发消息并把从第三方系统接收到的 ACK 消息返回给我们。

为了提供一些可靠性，新的写模块 xout3 在给定的时间间隔内没有接收 ACK 消息的话

就重新发送消息一次。如果在重新发送之后消息仍没有得到确认，xout3 就会记录这个情况并丢弃消息。为了让 ACK 消息和 ACK 确认的消息关联起来，xout3 在每一个消息中放置一个 cookie。消息的最后接收者在 ACK 消息中返回该 cookie。下面以图 3.25 中的 xout3 的声明部分开始分析。

xout3.c

```

1 #define ACK          0x6      /* an ACK character */
2 #define MRSZ         128      /* max unacknowledged messages */
3 #define T1           3000     /* wait 3 secs for first ACK */
4 #define T2           5000     /* and 5 seconds for second ACK */
5 #define ACKSZ        ( sizeof( u_int32_t ) + 1 )

6 typedef struct          /* data packet */
7 {
8     u_int32_t len;       /* length of cookie and data */
9     u_int32_t cookie;   /* message ID */
10    char buf[ 128 ];    /* message */
11 } packet_t;

12 typedef struct          /* message record */
13 {
14     packet_t pkt;      /* pointer to saved msg */
15     int id;            /* timer id */
16 } msgrec_t;

17 static msgrec_t mr[ MRSZ ];
18 static SOCKET s;

```

xout3.c

图 3.25 xout3 声明

声明

5 每个消息中附加的 cookie 实际上是一个 32 位的消息号，对等方的 ACK 消息定义为一个 ASCII ACK 字符，之后跟着被确认消息的 cookie。因此，程序定义 ACKSZ 作为 cookie 的大小加 1。

6~11 packet_t 结构定义了程序要发给对等方的数据包。因为消息可以是可变长度的，程序在数据包中包含了每一个消息的长度，所以对等方可以使用该域把数据流拆分为

几个记录，如技巧 6 里讨论的那样。`len` 域是消息本身和附加的 cookie 的大小。参阅图 2.31 之后的讨论，了解有关结构中成员包装的警告信息。

12~16 `msgrec_t` 结构包含了发送给对等方的 `packet_t` 结构。程序在 `msgrec_t` 结构中保存数据包，以备程序在需要时重新发送它。`id` 域是计时器的 ID，该计时器用作消息的 RTO 计时器。

17 每一个没有确认的消息都有一个相关的 `msgrec_t` 结构。这些结构在 `mr` 数组中保存。

下面分析一下 `xout3` 的 `main` 函数（图 3.26）。

初始化

11~15 该初始化操作和 `xout2` 中连接对等方并初始化标准输入的 `tselect` 事件屏蔽和 `tcp_client` 返回的套接字是一样的。

16~17 程序通过设置数据包的长度为 -1 来标记每一个可获得 `msgrec_t` 条目。

18~25 除了没有计时器参数之外，调用 `tselect` 和调用 `select` 的方法是一样的。如果 `tselect` 返回错误 0，程序输出就会诊断消息并退出。和 `select` 不一样的是，`tselect` 并不希望返回零值，这是因为超时事件的处理是由内部来完成的。

处理套接字输入

26~32 当程序得到套接字的读事件，就需要一个 ACK 消息。正如技巧 6 中讨论的那样，程序不能仅仅调用 `recv` 来获取 `ACKSZ` 个字节，这是因为这时还不能获得这么多的数据。程序也不能调用诸如直到接收指定数目字节的数据才返回的 `readn` 函数，这是因为和应用程序的事件驱动的初衷是相违背的——直到 `readn` 函数返回才能处理其他事件。因此，程序为发布的读操作必须读取为完成正在读取的 ACK 消息所需的所有数据。因为变量 `cnt` 包含了程序已经读取的字节数，所以 `ACKSZ-cnt` 就是完成 ACK 所需的数据的数目。

33~35 如果程序已经读取的数据少于 `ACKSZ`，程序就会返回至 `tselect` 等待更多的数据或其他事件。如果当前 `recv` 已经完全读取了完整的 ACK 消息，程序就为下一个 ACK 消息重置 `cnt` 为零（还没有读取下一个 ACK 消息的任何字节）。

36~40 下面，程序执行和技巧 11 中对应的数据完整性检测。如果消息不是有效的 ACK 消息，程序就输出诊断消息并继续运行。因为该错误指示对等方发送的数据并不是期望接收的数据，所以程序运行到这里最好中断。

41~42 最后，程序通过从 ACK 消息中拷贝出 cookie，调用 `findmsgrec` 来获得和该消息关联的 `msgrec_t` 结构，并使用它来取消计时器以及释放 `msgrec_t` 结构。`Findmsgrec` 和 `freemsgrec` 函数在如图 3.27 显示。

处理来自标准输入的输入

51~57 当 `tselect` 在标准输入上返回一个读事件时，程序会分配 `msgrec_t` 条目并读取消息到数据包中。程序从连续的消息计数器 `msgid` 中分配消息 ID，并把它保存在数据包的

cookie 域中。注意程序没有必要调用 htonl 函数，这是因为对等方并不检测 cookie，仅仅是不作任何改变返回。最后，程序在消息数据包中设置消息和 cookie 的总长度。这时，程序调用了 htonl，这是因为对等方使用该域来读取消息的区域部分（请参阅技巧 28）。

58~61 程序发送完全数据包给对等方，并调用 timeout 函数启动 RTO 计时器。

-xout3.c

```
1 int main( int argc, char **argv )
2 {
3     fd_set allreads;
4     fd_set readmask;
5     msgrec_t *mp;
6     int rc;
7     int mid;
8     int cnt = 0;
9     u_int32_t msgid = 0;
10    char ack[ ACKSZ ];
11
12    INIT();
13    s = tcp_client( argv[ 1 ], argv[ 2 ] );
14    FD_ZERO( &allreads );
15    FD_SET( s, &allreads );
16    for ( mp = mr; mp < mr + MRSZ; mp++ )
17        mp->pkt.len = -1;
18    for ( ; ; )
19    {
20        readmask = allreads;
21        rc = tselect( s + 1, &readmask, NULL, NULL );
22        if ( rc < 0 )
23            error( 1, errno, "tselect failure" );
24        if ( rc == 0 )
25            error( 1, 0, "tselect returned with no events\n" );
26
27        if ( FD_ISSET( s, &readmask ) )
28        {
29            rc = recv( s, ack + cnt, ACKSZ - cnt, 0 );
30            if ( rc == 0 )
31                error( 1, 0, "server disconnected\n" );
32        }
33    }
34}
```

```

31     else if ( rc < 0 )
32         error( 1, errno, "recv failure" );
33     if ( ( cnt += rc ) < ACKSZ )/* have whole msg? */
34         continue;           /* no, wait for more */
35     cnt = 0;              /* new msg next time */
36     if ( ack[ 0 ] != ACK )
37     {
38         error( 0, 0, "warning: illegal ACK msg\n" );
39         continue;
40     }
41     memcpy( &mid, ack + 1, sizeof( u_int32_t ) );
42     mp = findmsgrec( mid );
43     if ( mp != NULL )
44     {
45         untimeout( mp->id );    /* cancel timer */
46         freemsgrec( mp );      /* delete saved msg */
47     }
48 }

49     if ( FD_ISSET( 0, &readmask ) )
50     {
51         mp = getfreerec();
52         rc = read( 0, mp->pkt.buf, sizeof( mp->pkt.buf ) );
53         if ( rc < 0 )
54             error( 1, errno, "read failure" );
55         mp->pkt.buf[ rc ] = '\0';
56         mp->pkt.cookie = msgid++;
57         mp->pkt.len = htonl( sizeof( u_int32_t ) + rc );
58         if ( send( s, &mp->pkt,
59                     2 * sizeof( u_int32_t ) + rc, 0 ) < 0 )
60             error( 1, errno, "send failure" );
61         mp->id = timeout( ( tofunc_t )lost_ACK, mp, T1 );
62     }
63 }
64 }
```

xout3.c

图 3.26 xout3 的 main 函数

xout3 的其余部分在图 3.27 中显示。

xout3.c

```

1 msgrec_t *getfreerec( void )
2 {
3     msgrec_t *mp;
4
5     for ( mp = mr; mp < mr + MRSZ; mp++ )
6         if ( mp->pkt.len == -1 ) /* record free? */
7             return mp;
8     error( 1, 0, "getfreerec: out of message records\n" );
9     return NULL;           /* quiet compiler warnings */
10 }
11
12 msgrec_t *findmsgrec( u_int32_t mid )
13 {
14     msgrec_t *mp;
15
16     for ( mp = mr; mp < mr + MRSZ; mp++ )
17         if ( mp->pkt.len != -1 && mp->pkt.cookie == mid )
18             return mp;
19     error( 0, 0, "findmsgrec: no message for ACK %d\n", mid );
20     return NULL;
21 }
22
23 void freemsgrec( msgrec_t *mp )
24 {
25     if ( mp->pkt.len == -1 )
26         error( 1, 0, "freemsgrec: message record already free\n" );
27     mp->pkt.len = -1;
28 }
29
30 static void drop( msgrec_t *mp )
31 {
32     error( 0, 0, "Dropping msg: %s", mp->pkt.buf );
33     freemsgrec( mp );

```

```

29 }

30 static void lost_ACK( msgrec_t *mp )
31 {
32   error( 0, 0, "Retrying msg: %s", mp->pkt.buf );
33   if ( send( s, &mp->pkt,
34             sizeof( u_int32_t ) + ntohl( mp->pkt.len ), 0 ) < 0 )
35     error( 1, errno, "lost_ACK: send failure" );
36   mp->id = timeout( ( tofunc_t )drop, mp, T2 );
37 }

```

xout3.c

图 3.27 xout3 支持函数

getfreerec

1~9 该函数在 mr 数组中查询一个没有使用的条目。程序执行一个简单的线性数组查找，直到找到数据包长度为-1 的空条目为止。如果 mr 数据很大，程序就可以维护如图 3.21 中为 tevent_t 记录使用的自由列表。

findmsgrec

10~18 除了用指定的消息 ID 搜索一条记录之外，该函数几乎和 getfreerec 函数一模一样。

freemsgrec

19~24 在确认该条目不再空闲之后，程序设置消息数据包的长度为-1，并标记条目为空闲。

drop

25~29 当第二次发送的消息仍没有被确认时，就调用该函数（请参阅 lost_ACK）。程序记录诊断信息并通过调用 freemsgrec 来丢弃该消息。

lost_ACK

30~37 当第一次发送的消息没有被确认时，就调用该函数。程序重新发送该消息，并启动一个新的 ROT 计时器。在指定计时器终止时调用的函数为 drop。

为了测试 xout3，我们编写了一个随机丢弃消息的服务器应用程序，我们称服务器为 extsys（为外部系统创建的服务器）并在图 3.28 中显示它。

extsys.c

```

1 #include "etcp.h"

2 #define COOKIESZ 4 /* set by our peer */

3 int main( int argc, char **argv )
4 {
5     SOCKET s;
6     SOCKET s1;
7     int rc;
8     char buf[ 128 ];
9     INIT();
10    s = tcp_server( NULL, argv[ 1 ] );
11    s1 = accept( s, NULL, NULL );
12    if ( !isValidsock( s1 ) )
13        error( 1, errno, "accept failure" );
14    srand( 127 );
15    for ( ; ; )
16    {
17        rc = readvrec( s1, buf, sizeof( buf ) );
18        if ( rc == 0 )
19            error( 1, 0, "peer disconnected\n" );
20        if ( rc < 0 )
21            error( 1, errno, "recv failure" );
22        if ( rand() % 100 < 33 )
23            continue;
24        write( 1, buf + COOKIESZ, rc - COOKIESZ );
25        memmove( buf + 1, buf, COOKIESZ );
26        buf[ 0 ] = '\006';
27        if ( send( s1, buf, 1 + COOKIESZ, 0 ) < 0 )
28            error( 1, errno, "send failure" );
29    }
30 }
```

extsys.c

图 3.28 “外部系统”

✍ 初始化

9~14 程序执行通常的服务器初始化事物，并调用 `srand` 函数作为随机数产生程序的种子。

虽然标准 C 运行库函数 `rand` 速度很快而且很简单，但是它有几个不尽如人意的属性。尽管它对于 `xout3` 的演示还是很好的，但是严格的模拟应当使用更成熟的随机数产生程序。请参阅 [Knuth 1998]，了解更多的细节。

17~21 程序使用 `readvrec` 函数从 `xout3` 读取可变长度记录。

22~23 程序随机丢弃大约三分之一接收到的消息。

24~28 如果程序没有丢弃消息，就把它写入到标准输出，在输入缓冲区中将 `cookie` 向下移动一个位置，加入 `ACK` 字符，并把 `ACK` 消息返回给对等方。

在一个窗口中启动 `extsys` 并在另一个窗口中使用技巧 20 中的管道来测试 `xout3`（图 3-29）。

<pre>bsd \$ mp xout3 localhost 9000 xout3: Retrying msg: message 3 xout3: Retrying msg: message 4 xout3: Retrying msg: message 5 xout3: Dropping msg: message 4 xout3: Dropping msg: message 5 xout3: Retrying msg: message 11 xout3: Retrying msg: message 14 xout3: Dropping msg: message 11 xout3: Retrying msg: message 16 xout3: Retrying msg: message 17 xout3: Dropping msg: message 14 xout3: Retrying msg: message 19 xout3: Retrying msg: message 20 xout3: Retrying msg: message 16 xout3: server disconnected Broken pipe bsd \$</pre>	<pre>bsd \$ extsys 9000 message 1 message 2 message 3 message 6 message 7 message 8 message 9 message 10 message 12 message 13 message 15 message 18 message 17 message 21 message 20 message 23 ^C bsd \$</pre> <p style="text-align: right;"><i>Server terminated</i></p>
--	---

图 3.29 `xout3` 演示

通过观察，可以得到有关 `xout3` 的一些结论：

(1) 不能保证消息按次序递交。实际上，正如在图 3.29 中看到的消息 17 和消息 20 那样，重新发送的消息可能导致乱序到达。

(2) 我们可以通过给 `msgrec_t` 结构增加一次重发来使消息可以多次重发，让 `lost_ACK` 在耗尽重发次数之前一直发送消息。

(3) 可以很容易地通过修改 `xout3` 来使用 UDP 而不是 TCP。这是提供可靠 UDP 的第一步（请参阅技巧 8）。

(4) 对于处理 `tselect` 中多个套接字的应用程序来说，抽取内嵌 `readn` 代码到一个独立的函数更有意义。这个函数可以接收包含 `cnt`、指向输入缓冲区（或缓冲区本身）的指针以及已经读取完整消息后要调用的函数的地址的结构的输入。

(5) 作为一个例子，`xout3` 似乎有点像被迫的，特别是在技巧 19 中的上下文方面中，但是它确实说明了一种类型的解决方法，这些问题在现实事件中是经常发生的。

小结

本节和上节分析了事件驱动编程以及如何使用 `select` 来响应所发生的事件。在技巧 20 中，我们开发了 `tselect` 函数，它允许我们多路复用多个计时器到单一的 `select` 计时器上。该函数以及 `timeout` 和 `untimeout` 支持函数允许我们用少量的管理实现多事件的分时处理。

本节中，我们使用了 `tselect` 函数改进技巧 19 中的例子。通过使用 `tselect`，可以对通过网关服务器 `xout3` 发向不可靠终点的消息提供单独重传计时器。

技巧 22 不要使用 TIME-WAIT

ASSASSINATION 关闭连接

本节讨论的是 TCP TIME-WAIT 状态，它在 TCP 中服务的函数以及我们为什么不能击败它。因为 TIME-WAIT 状态被 TCP 状态机器的细节所隐藏，所以许多网络程序员只是朦胧地意识到它的存在，而没有真正理解它的用途和重要性。实际上，即使从来都没有听说过 TIME-WAIT 状态，也可以编写 TCP/IP 应用程序。但是对该状态的工作方式的了解对我们理解应用程序中看起来很怪异的行为以及避免那些将导致不可预测结果的操作是很有必要的（例如，请参阅技巧 23）。

本节以分析什么是 TIME-WAIT 状态以及它是如何适合 TCP 连接的发展开始，接着分析 TIME-WAIT 状态的用途以及它为什么很重要。本节最后分析为什么一些程序员试图击败它以及如何击败它，我们还指出可以另一个技巧显示完成相同事情的正确方法。

什么是 TIME-WAIT 状态

TIME-WAIT 状态在连接撤消阶段起作用。回忆一下技巧 7 中的 TCP 连接的撤消过程，它通常需要交换四个段，如图 3.30 所示。

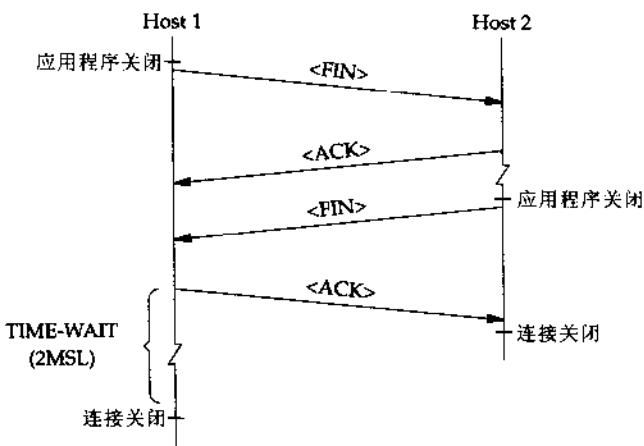


图 3.30 连接撤销

图 3.30 显示了主机 1 和主机 2 上的应用程序之间的连接。主机 1 上的应用程序关闭己方的连接并引发 TCP 发送一个 FIN 消息给主机 2。主机 2 确认该 FIN 消息，并把 FIN 作为一个 EOF 递交给应用程序（假设应用程序有读等待——请参阅技巧 16）。一段时间过后，主机 2 上的应用程序关闭它那边的连接，引发一个 FIN 消息发送给主机 1，主机 1 返回一个 ACK 消息。

这时，主机 2 关闭连接并释放连接的资源。以主机 2 的观点来看，连接已经不再存在了。然而，主机 1 还没有关闭连接，而是进入 TIME-WAIT 状态，并为两个最大段生存时间（2MSL）保留在此状态。

最大段生存时间（MSL）是段在丢弃之前能在网络中存在的最长时间。每一个 IP 数据报都有一个 TTL 域，数据报每次由路由器转发时都对 TTL 减 1。当 TTL 域变为 0 时，就丢弃该数据报。虽然 TTL 域的官方单位时秒，该域通常当作路由器计算的简单中继。RFC 1812[Baker 1995]讨论了这个问题以及使用它的原因。

等待 2MSL 时间之后，主机 1 也关闭连接并释放它的资源。

我们应当注意 TIME-WAIT 状态的三个方面：

(1) 通常情况下，仅有一方——执行主动关闭操作的一方——进入 TIME-WAIT 状态。

“主动关闭”的意思是指首先发送 FIN 消息的一方，另一方执行被动关闭。也有可能出现同时关闭的情况，这时双方同时关闭，它们的 FIN 消息在网络中相遇。在这种情况下，可以认为双方应用程序都执行了主动关闭操作，而且双方都进入 TIME-WAIT 状态。

(2) RFC 793[Postel 1981b]定义 MSL 为 2 分钟。在这个定义下，连接在 TIME-WAIT 状态下保持 4 分钟。然而，该值在实际中被广泛地忽略掉了。例如，因为 BSD 派生的系统的 MSL 是 30 秒，所以 TIME-WAIT 状态持续一分钟。其他介于 30 秒和 2 分钟的值也很常见。

(3) 如果连接处于 TIME-WAIT 状态期间有段到达，就重新启动一个 2MSL 计时器。我们将在下面讨论使用 TIME-WAIT 的原因时看到为什么会发生这种情况。

➤ 为什么需要 TIME-WAIT

设立 TIME-WAIT 状态有两个目的：

(1) 当由主动关闭方发送最后的 ACK 消息丢失并导致另一方重新发送 FIN 消息时，TIME-WAIT 维护连接状态。

(2) TIME-WAIT 为连接中“离群的段”提供从网络中消失的时间。

下面依次对这两个目的进行分析。当执行主动关闭的一方准备确认另一方的 FIN 消息时，就可以知道到目前为止双方传输的所有数据都已经被对等方接收到了。然而，这个最后的 ACK 消息丢失的可能还存在。这种情况如果发生了，执行被动关闭的一方就会运行超时并重新传输 FIN 消息（因为它没有接收最后序列号的 ACK 消息）。

现在考虑一下如果执行主动关闭的一方不进入 TIME-WAIT 状态就关闭连接那会发生什么呢？当重传的 FIN 消息达到时，因为 TCP 已经不再有连接的信息了，所以它就用 RST（重新启动）消息应答，导致对等方进入错误状态而不是有序终止状态。但是如果发送最后 ACK 消息的一方处于 TIME-WAIT 状态并仍然记录着连接的信息，它就可以正确地响应来自对等方的 FIN 消息。

上面的分析说明了当连接处于 TIME-WAIT 状态而段到达时为什么要重新启动终止时间为 2MSL 的计时器。如果最后的 ACK 消息丢失了而对等方重传了 FIN 消息，处于 TIME-WAIT 状态的一方就会再次确认 FIN 消息。重新启动计时器是为了防止该 ACK 再次丢失。

TIME-WAIT 状态的另一个目的更为重要。因为 IP 数据报在 WAN 中可能丢失或者延迟，所以 TCP 使用正面确认机制重传一定时间内没有被对等方确认的段（技巧 1）。如果数据报仅仅是延时而不是丢失，或者是数据报的 ACK 消息丢失了，重传的数据就可能在原先的数据接收之后到达。TCP 注意到延迟数据的序列是在当前接收窗口之外，处理并丢弃延迟的数据。

现在考虑一下如果延迟或重传段在连接关闭之后到达时将会发生什么。通常情况下，因为 TCP 仅仅丢弃该数据并响应 RST 消息，所以这不会造成任何问题。当 RST 消息到达发出延时段的主机时，因为该主机也没有记录该连接的任何信息，所以它也丢弃该段。然而，如果两个相同主机之间又建立了一个具有相同端口号的新连接，那么离群的段就可能被看成是属于新连接的。如果离群的段中数据的任何序列号恰好处在新连接的当前接收窗口中，数据就会被新连接接收，其结果是破坏新连接。

TIME-WAIT 状态通过确保旧套接字对（两个 IP 地址以及它们相应的端口号）在旧连接的段在网络上消失之前不会被重用来防止这种情况发生的。这样，我们可以看到 TIME-WAIT 状态在提供 TCP 的可靠性方面扮演着十分重要的角色。如果没有 TIME-WAIT 状态，TCP 就不能保证“以顺序的方式不被破坏地”递交数据。

➤ TIME-WAIT Assassination

不幸的是，确实存在过早关闭 TIME-OUT 状态的可能性，这种情况称作 TIME-OUT assassination，它可以是“意外”发生也可以是故意为之。

首先，让我们看看它是如何意外发生的。RFC 793 指出当连接处于 TIME-WAIT 状态并接收到一个 RST 消息时，TCP 应当立即关闭连接。假定有一个连接处于 TIME-WAIT 状态，这时就会有一个 TCP 不可接受的旧的重复段到达（也就是说，序列号处于当前接受窗口的外面）。TCP 响应一个 ACK 消息，指示它能接受的序列号（对等方 FIN 消息之后的序列号）。然而，对等方因为不要有连接的任何信息，所以以 RST 消息响应该 ACK。当该 RST 到达连接处于 TIME-WAIT 状态的主机时，它就会导致立即关闭连接——TIME-OUT 状态自杀了。

RFC 1337[Braden 1992b] 中描述了这种可能性，该文档还讨论了 TIME-WAIT assassination 的危害。这些危害影响了旧连接（也就是具有相同套接字对的连接）的再生，并且有可能导致旧数据的错误接收，导致发生无限 ACK 消息循环的连接不同步，以及新连接的错误终止。

幸运的是，可以通过改变 TCP 处于 TIME-WAIT 状态时忽略 RST 消息来很容易地防止这种情况发生。虽然 RFC 1337 指出该改变没有被官方采用，但它还是在一些 TCP/IP 协议栈里实现了。

发生 TIME-WAIT assassination 的另一种途径是人为的。通过使用后面讨论的 SO_LINGER 套接字选项，程序员可以强制立即关闭连接，甚至在应用程序正在执行主动关闭时也可以这样。这种不可靠的方法有时被建议用作让服务器从 TIME-WAIT 状态恢复以使它能在崩溃或中断之后重新启动的一种方法。技巧 23 中讨论了该问题的细节并给出了更好的解决方法。健壮的应用程序永远不应该干涉 TIME-WAIT 状态——它是 TCP 可靠性机制的一个重要部分。

通常情况下，当应用程序关闭连接时，close 或 closesocket 调用会立即返回，即使是发送缓冲区中仍有等待传输的数据时也同样如此。当然，TCP 仍然会试图递交没有发送的数据，但是应用程序并不知道递交是否成功。为了防止这个问题发生，可以设置 SO_LINGER 套接字选项。为了达到这个目的，必须填充 linger 结构并以 SO_LINGER 调用 setsockopt。

在大多数的 UNIX 系统中，linger 结构在 /usr/include/sys/socket.h 头文件中定义。Windows 系统在 winsock.h 或 winsock2.h 中定义它。在两种情况下，它都有如下格式：

```
struct linger {
    int l_onoff;           /* option on/off */
    int l_linger;          /* linger time */
}
```



如果 `l_onoff` 成员为 0，就关闭 `linger` 选项，所发生的操作等同于默认的操作——`close` 或 `closesocket` 调用立即返回，操作系统内核继续试图递交任何没有发生出去的数据。如果 `l_onoff` 为非零值，它就为操作系统内核停留的时间间隔，等待任何将要发送和确认的挂起数据。也就是说，`close` 或 `closesocket` 直到所有数据被递交或时间间隔到时才会返回。

如果当停留时间终止时仍有数据没有发送出去，`close` 或 `closesocket` 返回 `EWOULDBLOCK`，没有递交的数据就有可能丢失。如果数据已经发送完毕，两个调用就都返回 0。

不幸的是，`l_linger` 成员的意义是与具体实现相关的。在 Windows 和一些 UNIX 具体实现中，它的意思是关闭套接字时停留时间以秒计的数量。在 BSD 派生的系统中，它是停留的计时器节拍数（而文档却说它是以秒计的数量）。

我们应当意识到当以这种方法使用 `SO_LINGER` 时，只能保证数据可以传递到对等方 TCP，而它不能保证数据被对等方应用程序读取到。到达这个目标的更好方法是使用技巧 16 中讲述的顺序关闭过程。

最后，如果 `l_linger` 成员的值为零，连接就异常中断。也就是说，发送一个 RST 消息给对等方，连接不经过 TIME-WAIT 状态就立刻被关闭。这就是前面提及的人为 TIME-WAIT assassination。前面我们还提到，这是一个危险的方法，应当永远不要在正常程序中使用。

小结

本节分析了经常导致误解的 TIME-WAIT 状态。从中知道了 TIME-WAIT 状态是 TCP 可靠性中很重要的一部分，程序员不应该试图绕过它。从中我们知道 TIME-OUT 状态可以通过某些“正常的”网络事件提前终止它，它也可以在程序的控制下通过使用 `SO_LINGER` 套接字选项来终止。

技巧 23 服务器应设置 `SO_REUSEADDR` 选项

网络新闻组中最频繁问到的问题是：“当我的服务器崩溃或终止时，我试图重新启动它，发生‘地址已经使用的错误’。几分钟后，我们就可以重新启动服务器。如何才能使我的服务器立即启动？”为了说明这个问题，下面编写一个不完善的 echo 服务器，该服务器也有这个问题。

badserver.c

```

1 #include "etcp.h"

2 int main( int argc, char **argv )
3 {

```

```

4   struct sockaddr_in local;
5   SOCKET s;
6   SOCKET s1;
7   int rc;
8   char buf[ 1024 ];

9   INIT();
10  s = socket( PF_INET, SOCK_STREAM, 0 );
11  if ( !isValidsock( s ) )
12      error( 1, errno, "Could not allocate socket" );
13  bzero( &local, sizeof( local ) );
14  local.sin_family = AF_INET;
15  local.sin_port = htons( 9000 );
16  local.sin_addr.s_addr = htonl( INADDR_ANY );
17  if ( bind( s, ( struct sockaddr * )&local,
18          sizeof( local ) ) < 0 )
19      error( 1, errno, "Could not bind socket" );
20  if ( listen( s, NLISTEN ) < 0 )
21      error( 1, errno, "listen failed" );
22  s1 = accept( s, NULL, NULL );
23  if ( !isValidsock( s1 ) )
24      error( 1, errno, "accept failed" );
25  for ( ; )
26  {
27      rc = recv( s1, buf, sizeof( buf ), 0 );
28      if ( rc < 0 )
29          error( 1, errno, "recv failed" );
30      if ( rc == 0 )
31          error( 1, 0, "Client disconnected\n" );
32      rc = send( s1, buf, rc, 0 );
33      if ( rc < 0 )
34          error( 1, errno, "send failed" );
35  }
36 }
```

badserver.c

图 3.31 不完善的 echo 服务器

乍一看，除了硬编码（hard-coded）的端口号之外，该服务器似乎没有问题。实际上，如果在一个窗口中运行它，在另一个窗口中用 telnet 连接该服务器，确实可以得到想要的结果（图 3.32；本节中省去了 telnet 连接消息）。

<pre>bsd \$ badserver badserver: Client disconnected bsd \$ badserver badserver: Client disconnected bsd \$</pre>	<pre>bsd \$ telnet localhost 9000 hello hello ^] telnet> quit Client terminated Connection closed. Server restarted bsd \$ telnet localhost 9000 world world ^] telnet> quit Client terminated Connection closed. bsd \$</pre>
---	--

图 3.32 客户端终止

在确认服务器正在运行之后，通过转入 telnet 命令行模式并退出来终止客户端。注意到当我们立即重复该实验时，获得的是相同的结果。也就是说，badserver 重新启动时没有任何错误。

现在，让我们重新做这个实验，但是这次中断服务器。当我们试图重新启动服务器时，得到“地址已经使用”的错误消息（错误诊断消息显示在下一行）。两个实验的区别是，我们在第二个实验中中断的是服务器而不是（telnet）客户端（图 3.33）。

<pre>bsd \$ badserver ^C Server terminated bsd \$ badserver badserver: Could not bind socket: Address already in use (48) bsd \$</pre>	<pre>bsd \$ telnet localhost 9000 hello again hello again Connection closed by foreign host. bsd \$</pre>
---	---

图 3.33 中断服务器

为了弄清楚这两个例子中所发生的事情，必须理解两件事：

(1) TCP TIME-WAIT 状态。

(2) TCP 连接完全由 4-tuple（本地地址、本地端口号、外部地址、外部端口号）指定。

回忆一下技巧 22，执行主动关闭（发送第一个 FIN 消息）的 TCP 连接的一方进入 TIME-WAIT 状态并持续这种状态 2MSL 的时间。这个事实给我们在例子中看到的行为提供了第一个线索：当客户端执行主动关闭时，可以不会发生意外就重新启动连接的双方，但是当服务器执行主动关闭操作时，就不能重新启动。TCP 不允许服务器重新启动，这是因为前面的连接仍然处于 TIME-WAIT 状态。

然而，如果服务器重新启动了，而一个客户端连接进来，这时就需要一个新的连接，该连接甚至不是连接到同一台主机上。正如前面提到的那样，TCP 连接的是由本地地址、

远程地址以及端口号完全指定的，因此即使是同一个远程主机的客户端连接服务器，也不会发生任何问题，除非它使用前一个连接相同的端口号。

即使客户端是来自相同远程主机并且使用和前一个连接相同的端口号，也可能不会发生任何问题。传统的 BSD 具体实现允许接收对等方的消息，只要客户端的 SYN 消息的序列号比处于 TIME-WAIT 状态的连接最后一个序列号大就可以。

知道这些知识后，我们就可能对当重新启动服务器时 TCP 仍返回错误感到很诧异。其实问题不在 TCP 中，TCP 只需要 4-tuple 地址是唯一的，但是对于套接字 API 来说，却需要调用两个函数来完全指定连接的地址。在调用 bind 函数时，并不知道是否将会调用 connect；即使调用了，也不知道它是指定唯一的连接还是使用已经存在的那个连接。Torek[Torek 1994]以及其他的一些文档，它们用具有 bind、connect 和 listen 三个函数功能的一个函数来代替这三个函数。这样的话，就可以让 TCP 在不需要拒绝对已经终止或崩溃了的重起服务器的连接就可以检测到指定 4-tuple 地址的正在使用的那些连接。不幸的是，Torek 的解决方法并没有被人们采用。

幸运的是，该问题还有一个更容易的解决方法。程序员可以通过设置 SO_REUSEADDR 套接字选项来绑定一个已经使用的端口。为了实验这种方法，修改 badserver 应用程序（图 3.31），在第 7 和第 8 行之间加入下列一行：

```
const int on = 1;
```

在第 12 和第 13 行之间加入系列几行：

```
if ( setsockopt ( s, SOL_SOCKET, SO_REUSEADDR, &on,
    sizeof ( on ) ) )
    error ( 1, errno, "setsockopt failed" );
```

注意必须在调用 bind 之前调用 setsockopt。如果把该服务器称作 goodserver 并重新做图 3.33 中的实验，将会获得图 3.34 中的结果。

这次我们不用等待前一个连接的 TIME-WAIT 状态终止就可以重新启动服务器。因此，程序员应当总是在服务器里设置 SO_REUSEADDR 选项。注意前面的模板程序和库函数 tcp_server 已经自动地为我们设置了这个选项。

<pre>bsd \$ goodserver ^C bsd \$ goodserver</pre>	<pre>Server terminated</pre>	<pre>bsd \$ telnet localhost 9000 hello once again hello once again Connection closed by foreign host. Server restarted</pre>
		<pre>bsd \$ telnet localhost 9000 hello one last time hello one last time</pre>

图 3.34 使用 SO_REUSEADDR 终止的服务器

不幸的是，包括编写数据在内的某些人却认为设置 `SO_REUSEADDR` 是很危险的，这是因为它允许创建相同的 TCP 4-tuple 连接，这样会导致失败。这种观点是错误的。例如，如果我们试图创建两个相同的监听套接字，TCP 会拒绝绑定，即使是我们已经指定了 `SO_REUSEADDR` 也是这样。

```
bsd $ goodserver &
[1] 1883
bsd $ goodserver
goodserver: Could not bind socket: Address already in use (48)
bsd $
```

同样地，如果绑定相同的本地地址和端口号到使用 `SO_REUSEADDR` 的两个不同的客户端上，第二个客户端的绑定就会成功返回。但是如果连接第二个客户端到和第一个连接相同的远程主机的话，TCP 就会拒绝该连接。

在此我们重申，没有任何理由可以在服务器中不设置 `SO_REUSEADDR`。设置了该选项可以避免在服务器因不是执行主动关闭的一方而失败时不能立即重新启动服务器的问题。

Stevens[Stevens 1998]指出使用 `SO_REUSEADDR` 有一个很小的风险。如果服务器绑定通配符地址 `INADDR_ANY`（很多情况下），另一个服务器就可能设置了 `SO_REUSEADDR` 地址，绑定了同一个端口号但是指定的是一个具体的地址，这样就导致它可以“窃取”前一个服务器的连接。特别是对于网络文件系统（NFS）甚至是 UNIX 系统来说，它是一个特殊问题，这是因为 NFS 绑定一个非限制的端口 2049，在该端口上监听。注意，尽管这个风险不是因为 NFS 设置 `SO_REUSEADDR` 选项造成的，但是其他服务器可以设置该选项。也就是说，不管我们是否在服务器中设置了 `SO_REUSEADDR` 选项，该风险都可能存在，因此这并不能成为不设置它的原因。

应当指出的是 `SO_REUSEADDR` 还有其他的用途。例如，假设服务器运行在一台多地址的主机上，而且服务器需要知道客户端指定的目的地址是哪一个接口。对于 TCP 来说，到达这个目的是很容易的，因为服务器只需要在每个连接建立的时候调用 `getsockname`。但是除非 TCP/IP 具体实现支持 `IP_RECVSTADDR` 套接字选项，UDP 服务器没有其他方法找出具体的接口。UDP 服务器可以通过指定 `SO_REUSEADDR` 选项并绑定它的已知端口到它所关心的特定端口或者将那些它不关心的地址绑定到通配符地址 `INADDR_ANY` 上来解决这个问题，然后服务器就通过了解数据报到达哪一个套接字上来知道客户端指定的是哪一个地址。

希望提供根据客户端指定的不同地址而提供不同服务的 TCP（或 UDP）服务器有时采用相同的模式。例如，假设我们希望使用 `tcpmux`（技巧 18）的 `home-grown` 版本，当客户端连接到一个特定的接口如 198.200.200.1 时提供一套服务，而当客户端连接到其他接口时提供另一套服务。为了达到这个目的，可以在接口 198.200.200.1 上启动一个 `tcpmux` 的实

例提供特定的服务集，而在通配地址 INADDR_ANY 上启动另一个 `tcpmux` 实例，提供标准的服务集。因为 `tcpmux` 设置了 `SO_REUSEADDR` 选项，所以 TCP 允许我们第二次绑定端口 1，即使第二次绑定指定的是通配地址。

最后，`SO_REUSEADDR` 可以用于支持广播、允许多个应用程序同时监听进来的广播数据报的系统上。有关这个用途的详细信息可以在[Stevens 1998]中找到。

小结

本节分析了 `SO_REUSEADDR` 选项，了解了如何通过设置它来允许重新启动一个在 TIME-WAIT 状态中还存在连接的服务器。从中了解了服务器应当总是设置该选项，而且设置该选项时没有任何风险。

技巧 24 尽量使用大型写操作

代替多个小规模写操作

使用本节标题中的建议有两个原因。第一个是显而易见的，而且我们也在前面讨论过：每一次对写函数（`write`、`send` 等等）的调用至少需要两个上下文切换，这种切换是很耗费资源的操作。另一方面，除了对资源的滥用之外，如每次只写一个字节，多个写操作也许并不对应用程序增加额外的开销。这样，避免使用过多的系统调用是一个好的工程实践问题而不是一个紧迫的性能问题。

然而，有一个更为紧迫的原因要求避免使用过多的小规模写操作：Nagle 算法的影响。技巧 15 中简单地讨论了 Nagle 算法，本节将详细地分析它，以及它是如何跟应用程序交互的。我们将会看到不考虑 Nagle 算法因素可能会造成对应用程序极大的负面影响。

不幸的是，Nagle 算法，跟 TIME-WAIT 状态一样，很多网络程序员都不了解它。和 TIME-WAIT 状态一样，对 Nagel 算法了解的缺乏经常导致在解决有关该算法的问题时得到不正确的解决方案。我们首先分析为什么要使用该算法，然后看看什么时候以及如何禁用该算法，最后将了解如何以一种有效的方法来处理它，以提供良好的应用性能而对网络没有负面影响。

Nagle 算法由 John Nagle (RFC 896[Nagle 1984]) 提出，目的是为了适应 telnet 以及类似程序的性能问题。这些程序的问题是，它们通常在每一个独立的段中发送一个按键而导致网络中存在一系列的“小数据报（tiny gram）”。看来这是很简单的一个问题。首先，因为最小的 TCP 数据段（没有数据）是 40 字节，所以每个段发送一个字节将导致百分之 4000 的开销。但是更为重要的是，网络中数据包数量的增加会引起网络拥塞，发生拥塞后的重传会导致更多的拥塞。在极端的情况下，网络中每个段都存在几个拷贝，吞吐量减小为正

常情况下的几分之一。

如果网络中不存在未确认的数据（对等方已经确认我们发送出去的所有数据），我们就称该连接为空闲。正如最初想像的那样，除非连接处于空闲状态，否则 TCP 是不会发送新数据的，Nagle 算法通过这种方法防止了前面描述的问题。这可以防止一个连接一次出现多个小段。

RFC1122[Braden 1989]文档中给出的过程放松了这个限制，它可以允许数据在积累到足够一个完整的段时发送出去。也就是说，如果 TCP 可以发送至少 MSS 字节的数据，那么即使连接不处于空闲状态也可以发送数据。注意 Nagle 算法维持的条件：每个连接每次至多只能有一个小段。

许多具体实现对该规则进行了一些折衷，在每个发送而不是每个段的基础上实现 Nagle 算法。为了讲清楚这个差别，假定 MSS 是 1460 字节，一个应用程序执行 1600 字节的写操作，而发送和拥塞窗口至少为 2000 字节，而且连接处于空闲状态。以每个段为基础实现 Nagle 算法的具体实现发送 1460 字节，然后在发送其余的 140 字节之前等待 ACK 消息的到来——Nagle 算法应用于每次段的发送上。在每个发送基础上实现 Nagle 算法的具体实现现在发送 1460 字节之后立即发送余下的 140 字节段——Nagle 算法仅应用于应用程序递交新的数据给 TCP 发送时。

Nagle 算法本身运行得很好。它防止了应用程序以小段造成网络拥塞，而且在大多数情况下，TCP 运行起来的效果跟没有 Nagle 算法是一样的。

例如，假定每次以 200ms 递交到 TCP 一个字节。如果连接的 RTT 为 1 秒，不带 Nagle 算法的 TCP 具体实现将以百分之 4000 的开销每分钟发送 5 个段。而对于带 Nagle 算法的 TCP 具体实现来说，第一个字节立即被发送，用户输入的后面 4 个字节一直保存到第一个段的 ACK 消息到达。这时四个字节一起发送。这样只发送了 2 个段而不是 5 个，开销减少为百分之 1600，而数据传输率维持为同样的每秒 5 个字节。

不幸的是，Nagle 算法可能很糟糕地影响后面的 TCP 的另一个特性——延时的 ACK 消息。

当对等方的段到达时，TCP 延迟发送 ACK 消息，希望应用程序对刚接收到的数据做出响应，以便 ACK 消息可以在新数据中捎带出去。该延时值通常为 200 毫秒（BSD 具体实现）。

RFC 1122 对延时值的大小没有作任何规定，只是说明了它不能大于 500 毫秒，该文档同时也建议至少每隔一次确认一下。

和 Nagle 算法一样，延迟确认机制的目的是为了减少网络中传输的段的数量。

下面看看这两个机制在典型的请求/响应会话中是如何相互影响的。在图 3.35 中，客户端发送短请求给服务器，等待响应，然后做出其他请求。

注意 Nagle 算法在这里并没有起作用，这是因为客户端只有在知道响应到达之后才会发送另一个段，响应中还捎带了第一个请求的 ACK 消息。在服务器一方，延迟的 ACK 消息使服务器有时间做出响应，结果是每个请求/响应只花了两个段。如果 RTT 是段的来回时

间，而 T_p 是服务器处理请求并做出响应所需的时间（以毫秒计），那么每个请求/响应花费 $RTT + T_p$ 毫秒。

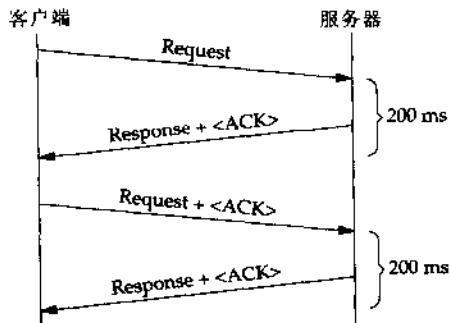


图 3.35 单段响应/请求数据流

现在假定客户端以两个分开的写操作来发送请求，发生这种情况的一个经常的原因是请求有一个报头后面跟着一些数据。例如，发送可变长度请求给服务器的客户端可能首先发送请求的长度，之后跟着实际的请求数据。

我们可以在图 2.31 中看到这种情况的例子，但是在那个图里我们考虑的是在一个段里发送长度和数据。

图 3.36 中显示了数据流。

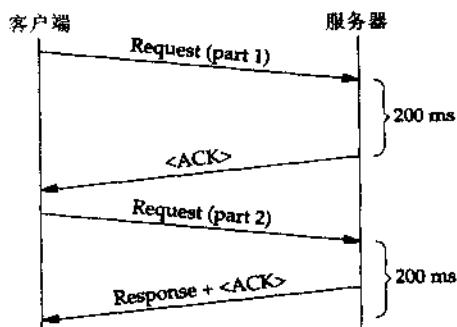


图 3.36 Nagle 算法和延迟 ACK 算法互相影响

这次，两个算法互相影响的结果是每个请求/响应对要发送的段加倍了，而且带来了很大的延迟。

请求的第一部分的数据立即被发送出去，但是 Nagle 算法防止第二部分发送。当服务器应用程序接收到请求的第一部分时，它不能响应，这是因为它还没有接收到全部请求。这意味着延迟 ACK 的计时器不得不在第一部分段的 ACK 消息产生之前发生超时。实际上，Nagle 和延迟 ACK 算法互相阻塞：Nagle 算法防止第一部分数据确认之前发送请求的第二部分，而延迟 ACK 算法要求等到计时器终止才发送 ACK 消息，这是因为服务器正在等待

请求的第二部分。每个请求/应答对现在花费四个段和 $2 \times RTT + T_p + 200$ 毫秒。结果是，即使我们不计算服务器处理和消息来回的时间，每秒也至多能处理 5 个请求/应答对。

对于很多系统来说，这个讨论都简单化了。例如，BSD 派生的系统每 200 毫秒检测所有连接的延迟 ACK，然后不管实际延迟了多长时间立即发送延迟的 ACK 消息。这意味着实际的延迟可能在 0 到 200 毫秒之间，所以可以估算平均延时为 100 毫秒。然而，延时值因为“phase effects”在大多数情况下仍然为 200 毫秒，“phase effects”中的延迟被下一个 200 毫秒中断。实际上，第一个响应是和时钟 tick 的响应是同步的。请参阅[Minshall et al. 1999]，了解这种情况的一个很好的例子。

如果我们分析一下最后的例子，就可以发现问题是因为客户端执行了一系列的写、写、读操作。任何这样的操作序列都会触发 Nagle 和延迟 ACK 算法之间的互相影响，因此应当尽量避免这种情况发生。更为普遍的是，执行小规模写操作的应用程序将会在对等方不立即响应的时候经历这个问题。

例如，假定一个数据收集应用程序每隔 50 毫秒给服务器发送一个整数。如果服务器没有回答这些消息，而只是记录它们作以后分析用，就可以看到同样的互相影响。客户端发送一个整数，然后被 Nagle 和延迟 ACK 算法延迟了（比方说）200 毫秒，之后每 200 毫秒发送 4 个整数。

☒ 禁用 Nagle 算法

因为最后的例子中的服务器仅仅记录接收的数据，Nagle 和延迟 ACK 算法之间的互相影响没有什么危害，而且实际上它会减少数据包的数量为原来的四分之一。然而，假定客户端给服务器发送的是一系列的温度值，而如果在这些温度值超出了关键区域之后服务器就必须在 100 毫秒之内做出反应。由两个算法互相影响而引入的 200 毫秒延迟就不能接受了，因此我们希望消除它。

幸运的是，RFC 1122 明确规定应当有禁用 Nagle 算法的方法。温度监控客户端的例子就是需要使用这种禁用的实例。一个影响比较小但更为现实的例子是运行在 UNIX 上的 X Window 系统。因为 X 使用 TCP 在显示（服务器）和系统的应用程序（客户端）部分之间进行通信，X 服务器必须不受 Nagle 算法的影响传递输入（入鼠标移动）给 X 客户端。

对于套接字 API，Nagle 算法可以由 TCP_NODELAY 套接字选项来禁用。设置该选项可以禁用 Nagle 算法：

```
const int on = 1;
setsockopt ( s, IPPROTO_TCP, TCP_NODELAY, &on, sizeof ( on ) );
```

不能仅仅因为我们可以关闭 Nagle 算法，就意味着我们应当关闭它。具有合理的理由禁用 Nagle 算法的应用程序远远少于不能禁用的应用程序。程序员之所以陷入经典的 Nagle/延迟 ACK 互相影响的困境之中是因为执行了一系列的小规模写操作，但是他们不想用一个

大型写操作代替它们来传递消息给对等方应用程序的方法来解决。然后，程序员注意到应用程序的性能比想象中的低得多，于是寻求帮助。有些人总是说：“哦，这是 Nagle 算法的原因，关闭它吧。”可以确信的是，当关闭 Nagle 算法时，性能问题确实得到解决。不幸的是，该性能是以网络中存在大量的小数据报为代价获得的。如果很多应用程序都这么做，或者干脆如某些人建议的那样设置默认值为禁用，将会增加网络拥塞的可能，而且甚至有可能在极端情况下导致崩溃。

聚集写操作

正如我们看到的那样，有些应用程序必须禁用 Nagle 算法，但是大多数的应用程序禁用 Nagle 算法是因为以多个小规模的写操作发送逻辑上有关联的数据引发了性能问题。因为合并数据的方法有多种，所以它们可以一起写。在极端的情况下，我们可以在发送之前拷贝不同的段到一个缓冲区中，但是正如技巧 26 中解释的那样，这是最后迫不得已才采用的方法。有时，事先做好规划，就可以安排数据以图 2.31 中的方式存储在一起。然而，数据通常驻留在两个或更多的不连接的缓冲区中，我们希望有一种有效的方法一次把它们写出去。

幸运的是，UNIX 和 Winsock 环境都提供了这么做的方法。不幸的是，它们是以稍微不同的方法提供的。在 UNIX 系统下，有 writev 以及对应的 readv 调用。使用 writev，可以指定一系列的缓冲区，收集要写的数据。该调用巧妙地解决我们的问题：我们可以安排数据保存在多个缓冲区中，然后同时写出去，避免出现 Nagle 和延迟 ACK 算法的互相影响。

```
# include <sys/uio.h>
ssize_t writev( int fd, const struct iovec *iov, int cnt );
ssize_t readv( int fd, const struct iovec *iov, int cnt );
>Returns:bytes transferred or -1 on error
```

参数 iov 是一组 iovec 结构的指针，iovec 结构包含指向缓冲区的指针以及指定缓冲区长度的变量。

```
struct iovec {
    char *iov_base; /* base address. */
    size_t iov_len; /* Length. */
};
```

前面的定义取自 FreeBSD 系统，许多系统现在定义基本地址指针为

```
void *iov_base; /* Base address. */
```

第三个参数 cnt 是数组中 iovec 结构的个数（也就是说，分开缓冲区的个数）。

`writev` 和 `readv` 调用有相同的接口。它们不仅可以用于套接字，还可以用在任何类型的文件描述符上。

为了弄清楚这两个函数是如何工作的，下面重新编写了图 2.31 中的可变长度记录客户端，该客户端使用了 `writev` 调用（图 3.37）。

```

1 #include "etcp.h"
2 #include <sys/uio.h>

3 int main( int argc, char **argv )
4 {
5     SOCKET s;
6     int n;
7     char buf[ 128 ];
8     struct iovec iov[ 2 ];

9     INIT();
10    s = tcp_client( argv[ 1 ], argv[ 2 ] );
11    iov[ 0 ].iov_base = ( char * )&n;
12    iov[ 0 ].iov_len = sizeof( n );
13    iov[ 1 ].iov_base = buf;
14    while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
15    {
16        iov[ 1 ].iov_len = strlen( buf );
17        n = htonl( iov[ 1 ].iov_len );
18        if ( writev( s, iov, 2 ) < 0 )
19            error( 1, errno, "writev failure" );
20    }
21    EXIT( 0 );
22 }
```

vrcv.c

vrcv.c

图 3.37 使用 `writev` 调用发送可变长度消息的客户端

✍ 初始化

9~13 执行一般的客户端初始化后，程序设置 `iov` 数组。因为 `writev` 调用指定 `iov` 参数指向的结构为 `const` 变量，就保证 `iov` 数组不被 `write` 调用改变，所以程序就可以在 `while`

循环的之外设置结构的大多数域。

事件循环

14~20 程序调用 fgets 读取一行输入，计算它的长度，并在 iov 数组中设置该长度。同时程序将转换长度值为网络字节顺序并把它放在 n 中。

如果我们启动技巧 6 中的 vrs 服务器，然后运行 rcv，就将获得和前面相同的结果。在 Winsock 中，函数有不同但是相似的接口。

```
# include <winsock2.h>

int WSAAPI WSAsend( SOCKET s, LPWSABUF buf, DWORD cnt,
    LPDWORD sent, DWORD flags, LPWSAOVERLAPPED ov,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE func );

>Returns:0 if successful, SOCKET_ERROR otherwise
```

最后两个参数用于重叠 I/O，实现本节中的目标可以把它忽略掉，程序应当把它们都设置为 NULL。参数 buf 指向 WSABUF 结构数据，它的作用和 writev 中的 iovec 结构相似：

```
typedef struct _WSAVUF {
    u_long      len;          /* the length of the buffer */
    char FAR*   buf;          /* the pointer to the buffer */
} WSABUF, FAR* LPWSABUF;
```

如果调用成功返回，参数 sent 就指向包含所发送字节数目的一个 DWORD 变量。参数 flag 和 send 调用的 flag 域是类似的。

我们可以使用 WSASend 创建可变长度消息客户端的 Winsock 版本（图 3.38）。

vrcvw.c

```
1 #include "etcp.h"

2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     int n;
6     char buf[ 128 ];
7     WSABUF wbuf[ 2 ];
8     DWORD sent;
```

```

9  INIT();
10 s = tcp_client( argv[ 1 ], argv[ 2 ] );
11 wbuf[ 0 ].buf = ( char * )&n;
12 wbuf[ 0 ].len = sizeof( n );
13 wbuf[ 1 ].buf = buf;
14 while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
15 {
16     wbuf[ 1 ].len = strlen( buf );
17     n = htonl( wbuf[ 1 ].len );
18     if ( WSASend( s, wbuf, 2, &sent, 0, NULL, NULL ) < 0 )
19         error( 1, errno, "WSASend failure" );
20 }
21 EXIT( 0 );
22 }

```

vrcvw.c

图 3.38 vrcv 的 Winsock 版本

可以看出，除了聚集写操作调用有些不同之外，Winsock 版本和 UNIX 版本基本一样。

小结

本节分析了 Nagle 算法以及它跟延迟 ACK 算法之间的相互影响。从中可以看出，执行几个小规模写操作而不是用一个大型写操作代替的应用程序易于遭受很大的性能下降。

因为 Nagle 算法可以预防一个普遍存在的问题——网络中充满了小数据报，所以不能通过禁用它来解决执行多个小规模写操作的应用程序的性能问题。解决的方法是，应用程序应当整理好逻辑上有关联的数据然后一次发送出去。本节的最后分析了达到这个目的的一个方便的方法：UNIX 系统下的 writev 调用和 Winsock 下的 WSASend 调用。

技巧 25 理解如何使 connect 调用具有超时机制

正如我们在技巧 7 中的讨论的那样，通常的 TCP 连接建立过程包括三个段的交换（称作三阶段握手）。如图 3.39 所示，该过程由客户端的 connect 调用发起，并以服务器收到 SYN 的 ACK 消息结束。

其他的交换，如同时发生的连接的初始化 SYN 段在网络中相遇，当然也是可能的，但是绝大多数的连接建立以图 3.39 中的方式进行。

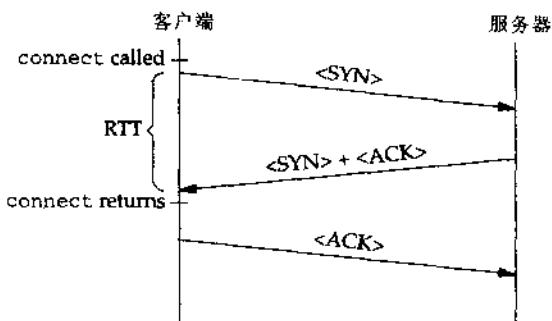


图 3.39 通常的三阶段握手

在阻塞套接字的一般情况下，`connect` 直到客户端对 SYN 消息的 ACK 消息到达之前才会返回。因为这至少要花费一个 RTT 的时间，而且如果发生了网络拥塞或所连接的主机没有打开的话也可能更长，所以能够自行中断 `connect` 是很有用的。当然 TCP 最后可以自己退出 `connect` 连接，但是默认值（通常是 75 秒）相对我们可以等待的时间来说可能太长了。一些具体实现如 Solaris 提供了套接字选项来控制连接中断超时时间，但是不幸的是这些选项并不是在所有的系统中都有。

使用报警信号

使 `connect` 调用超时有两种方法，最容易的方法是简单地在调用周围设置报警信号。例如，假定我们希望 `connect` 不能超过 5 秒钟完成。为了达到这个目的，我们修改 `tcpclient.skel`（图 2.19），增加一个简单的信号处理程序并修改 `main` 函数如下：

```
void alarm_hdlr ( int sig )
{
    return;
}

int main ( int argc, char **argv )
{
    ...
    signal ( SIGALRM, alarm_hdlr );
    alarm ( 5 );
    rc = connect ( s, ( struct sockaddr * )&peer, sizeof ( peer ) );
    alarm ( 0 );
    if ( rc < 0 )
    {
        ...
    }
}
```

```

if (errno == EINTR)
    error (1, 0, "connect timed out\n");
...
}

```

如果重命名模板程序为 connectto，并用它连接一个繁忙的 Web 站点如 Yahoo，就可能得到预想的结果：

```

bsd: $ connectto yahoo.com daytime
connectto: connect timed out                                5 seconds later
vsd: $

```

虽然该解决方法很简单，但是它还存在几个潜在的问题。下面讨论一下这些问题，并研究另一个方法，该方法虽然复杂一点，但是可以避免这些问题。

首先，我们的样本代码假定警告信号计时器并没有由程序的其他部分使用，而且没有其他的信号处理程序。如果计时器已经运行了，我们的程序就重置它，并取消老的计时器。为了达到健壮的目的，程序不得不保存并恢复当前计时器中剩下的时间量（由 `alarm` 调用返回），而且同时也保存并恢复 `SIGALRM` 信号的当前处理程序（由 `signal` 调用返回）。为了正确地完成这个操作，程序应当在等待 `connect` 时获取已经花费的时间，并从原来的计时器中剩下的时间里减去它。

其次，为了达到简单的目的，如果 `connect` 超时我们就仅终止客户端，但是我们应当采取其他的一些操作。然而，我们应当意识到我们不能重新启动 `connect`，这是因为套接字作为第一个 `connect` 的结果仍然绑定了，如果试图重新启动 `connect` 连接的话将会导致“地址已经使用”的错误。也许在一段延时之后，如果我们希望重试 `connect`，那么我们应当首先调用 `close`（或 `closesocket`）关闭套接字，然后使用 `socket` 来重新打开它。

该方法的另一个可能的问题是一些 UNIX 系统可能在信号处理程序返回后自动地重新启动 `connect` 调用。在这种情况下，调用只有到默认 TCP 超时间隔已经终止时才会返回。所有的现代 UNIX 系统都支持 `sigaction` 调用，它可以用于代替 `signal` 调用。使用 `sigaction` 调用，我们可以指定 `connect` 调用是否需要重新启动。然而，因为一些老的系统不支持该调用，所以对它们来说使用 `alarm` 来使 `connect` 调用具有超时机制是很困难的。

如果我们希望做的事情是输出诊断信息然后终止，就可以在信号处理程序里执行这些操作。因为它在 `connect` 调用重新启动之前发生，所以并不存在系统是否支持 `sigaction` 的问题。然而，如果我们希望采取其他的一些操作，就可能不得不使用 `longjmp` 跳出信号处理程序，而这总是引入了 race condition。

注意在终止的简单情况下也可能存在 race condition。假定连接完成并且 `connect` 调用返回了，但是在程序调用 `alarm` 关闭计时器之前，计时器终止，导致信号处理程序的调用，结果程序被终止。

```

alarm ( 5 );
rc = connect ( s, NULL, NULL );
/* timer fires here */
alarm ( 0 );

```

在这种情况下，即使连接实际上是成功的，我们也已经终止了程序。原来的代码没有这个 race condition，这是因为即使 connect 调用返回和调用 alarm 之间计时器到时了，信号处理程序也仅仅是返回而没有采取任何操作。

因为这些原因，许多专家认为使 connect 调用具有超时机制的正确方法是使用 select 调用，我们将在下面讨论它。

使用 select

使 connect 调用具有超时机制的第二个更通用的方法是让套接字成为非阻塞的套接字，然后用 select 来等待它完成。该方法避免了许多前面通过使用 alarm 来达到这个目的所遇到的许多问题，但是该方法仍然可移植性的问题，即使是在 UNIX 具体实现中也是这样，我们必须适应它。

首先看看 connect 代码本身。我们以采用 `tcpclient.skel` 代码并修改 `main` 函数开始，如图 3.40 所示：

connectto1.c

```

1 int main( int argc, char **argv )
2 {
3     fd_set rdevents;
4     fd_set wrevents;
5     fd_set exevents;
6     struct sockaddr_in peer;
7     struct timeval tv;
8     SOCKET s;
9     int flags;
10    int rc;
11
12    INIT();
13
14    set_address( argv[ 1 ], argv[ 2 ], &peer, "tcp" );

```

```
13 s = socket( AF_INET, SOCK_STREAM, 0 );
14 if ( !isValidsock( s ) )
15     error( 1, errno, "socket call failed" );

16 if( ( flags = fcntl( s, F_GETFL, 0 ) ) < 0 )
17     error( 1, errno, "fcntl (F_GETFL) failed" );
18 if ( fcntl( s, F_SETFL, flags | O_NONBLOCK ) < 0 )
19     error( 1, errno, "fcntl (F_SETFL) failed" );

20 if ( ( rc = connect( s, ( struct sockaddr * )&peer,
21         sizeof( peer ) ) ) && errno != EINPROGRESS )
22     error( 1, errno, "connect failed" );
23 if ( rc == 0 )           /* already connected? */
24 {
25     if ( fcntl( s, F_SETFL, flags ) < 0 )
26         error( 1, errno, "fcntl (restore flags) failed" );
27     client( s, &peer );
28     EXIT( 0 );
29 }

30 FD_ZERO( &rdevents );
31 FD_SET( s, &rdevents );
32 wrevents = rdevents;
33 exevents = rdevents;
34 tv.tv_sec = 5;
35 tv.tv_usec = 0;
36 rc = select( s + 1, &rdevents, &wrevents, &exevents, &tv );
37 if ( rc < 0 )
38     error( 1, errno, "select failed" );
39 else if ( rc == 0 )
40     error( 1, 0, "connect timed out\n" );
41 else if ( isconnected( s, &rdevents, &wrevents, &exevents ) )
42 {
43     if ( fcntl( s, F_SETFL, flags ) < 0 )
44         error( 1, errno, "fcntl (restore flags) failed" );
45     client( s, &peer );
```

```

46  }
47 else
48     error( 1, errno, "connect failed" );
49 EXIT( 0 );
50 }

```

connection.c

图 3.40 使用 select 使 connect 超时

设置套接字为非阻塞

16~19 程序获取套接字的标志，把它和 O_NONBLOCK 标志相 OR，然后设置新的标志。

初始化连接

20~29 程序调用 `connect` 启动连接序列。因为程序已经标记套接字为非阻塞，所以 `connect` 调用立即返回。如果连接像连接自己一样很快建立起来了，`connect` 返回 0，程序就重新把套接字设置回到阻塞状态并调用 `client` 函数。通常情况下，连接在 `connect` 返回之前是不会建立的，因此它会返回 EINPROGRESS 错误。如果它返回任何其他的错误，程序就输出诊断信息并终止。

调用 select

30~36 程序调用 `select` 的通常设置，包括设置 5 秒的超时。同时程序也因后面解释的原因记录异常事件。

处理 select 的返回

37~40 如果 `select` 返回错误或发生了超时事件，程序就输出诊断信息并终止。当然，程序也可以在超时事件中采取一些其他的操作。

41~46 程序调用 `isconnected` 函数检查连接是否成功，如果成功了，就把套接字设置回原来的阻塞状态并调用 `client` 函数。函数 `isconnected` 在图 3.41 和图 3.42 中给出。

47~48 如果连接没有成功，程序就输出诊断信息并终止。

不幸的是，UNIX 和 Winsock 使用不同的方法来指示连接是否成功。这是我们抽取检测函数到一个独立函数的原因。下面我们首先给出 `isconnected` 的 UNIX 版本。

在 UNIX 中，一旦连接建立，套接字就可以执行写操作了。如果发生了错误，套接字就既可读也可写了。然而，我们不能依靠这个来判断连接是否成功，这是因为 `connect` 调用可能在程序调用 `select` 之前就成功返回并有数据可读了。在这种情况下，套接字就既可读又可写了，和发生错误的现象一样。在这里，我们调用 `getsockopt` 来获取套接字的错误状态。

connecttol.c

```

1 int isconnected( SOCKET s, fd_set *rd, fd_set *wr, fd_set *ex )
2 {
3     int err;
4     int len;
5     errno = 0;           /* assume no error */
6     if ( !FD_ISSET( s, rd ) && !FD_ISSET( s, wr ) )
7         return 0;
8     if ( getsockopt( s, SOL_SOCKET, SO_ERROR, &err, &len ) < 0 )
9         return 0;
10    errno = err;        /* in case we're not connected */
11    return err == 0;
12 }

```

connecttol.c

图 3.41 isconnected 的 UNIX 版本

5~7 如果套接字既不可读也不可写，那么连接就没有返回，而程序返回 0。程序预先设置 `errno` 为零，所以调用者可以决定套接字是否没有准备好（本例子）或者发生了错误（后面将讨论）。

8~11 程序调用 `getsockopt` 来获取套接字的错误。在 UNIX 的某些版本中，如果套接字发生了错误 `getsockopt` 就返回 -1。在本例子中，`errno` 设置为错误代码。其他版本的 UNIX 仅返回套接字的错误状态而让调用者来检查。本代码对两种情况都进行处理，该思想来自于 Stevens[Stevens 1998]。

对于 Winsock 来说，当使用 `select` 时，在非阻塞的套接字上调用 `connect` 返回的错误由异常事件来指示。注意这和 UNIX 系统不一样，UNIX 的异常事件总是指示紧急数据的到来。`isconnected` 的 Windows 版本在图 3.42 中给出。

connecttol.c

```

1 int isconnected( SOCKET s, fd_set *rd, fd_set *wr, fd_set *ex )
2 {
3     WSAClearLastError();
4     if ( !FD_ISSET( s, rd ) && !FD_ISSET( s, wr ) )
5         return 0;
6     if ( FD_ISSET( s, ex ) )
7         return 0;
8     return 1;
9 }

```

connecttol.c

图 3.42 isconnected 的 Windows 版本

3~5 和 UNIX 版本一样，程序检查套接字是否已经连接。如果没有，程序就设置最后的错误并返回 0。

6~8 如果套接字存在一个异常事件，程序就返回 0，否则返回 1。

小结

正如我们所看到的那样，使 `connect` 调用具有超时机制，这带来了比通常情况多得多的可移植性问题。因为这个原因，当我们需要使用这样的超时机制时，应当特别注意所使用的平台。

最后，应当意识到虽然可以为 `connect` 调用缩短超时时间间隔，但是却没有任何方法来加长它。我们分析的方法都是在 TCP 执行退出之前退出 `connect` 的。在每个套接字基础上没有通用的机制来改变 TCP 的超时时间值。

技巧 26 避免数据拷贝

许多网络应用程序，特别是那些主要考虑机器之间移动数据的那些应用程序，从一个缓冲区拷贝数据到另一个缓冲区的操作占用了大多数的处理时间。本节我们将考虑减少数据拷贝量的方法，提高程序性能，使它们更“空闲”。实际上，如果我们考虑一下标准过程：不是从一个函数到另一个函数传递数组中的数据，而是传递初始数组的指针，那么我们应当在内存中避免拷贝大量数据的想法似乎变得并不重要的。

当然，程序通常并不在同一个进程的函数之间拷贝数据缓冲区，但是在多进程应用程序中，不管应用程序使用什么 IPC 机制，应用程序通常都要从一个进程到另一个进程之间传输大量的数据。甚至是在同一个进程之间，当消息由两个或更多的部分组成时也是经常拷贝数据的。我们希望能够组合这些数据块然后传输到另一个进程或机器。技巧 24 中讨论了这种情况的一个普通的例子，预先附加消息头到消息中。首先拷贝消息头到一个缓冲区中，然后拷贝数据进去，整个消息就拷贝完了。

在同一个进程中避免这种拷贝通常是一个好的程序设计习惯问题。如果我们知道程序将要预先给读入到缓冲区的数据附加消息头，那么我们应当在执行读操作时就为它们预留空间。也就是说，当我们将要预先附加一个包含在 `struct hdr` 的消息头，那么我们可以这样执行读操作：

```
rc = read ( fd, buf + sizeof ( struct hdr ),
            sizeof ( buf ) - sizeof ( struct hdr ) );
```

我们将会在图 3.9 中看到使用这个记数的一个例子。

另一种技术就是定义消息数据包为一个结构，数据缓冲区为该结构的一个成员。就这

样我们就可以仅读取数据到适当的结构域：

```
struct
{
    struct hdr header;      /* defined elsewhere */
    char data [ DATASZ ];
} packet;
rc = read ( fd, packet.data, sizeof ( packet.data ) );
```

我们在图 2.31 中已经看到了这样的一个例子，在那里我们已经讨论过了，在定义这样的结构时应当十分小心。

第三种十分灵活的方法是使用图 3.37 (UNIX) 和图 3.38 (Winsock) 中讨论的聚集写操作。该技术允许当消息的大小不固定时连接消息的各部分。

当涉及到多个进程时，避免数据拷贝是一件很困难的事情。在多进程应用程序十分普遍的 UNIX 系统中（例如，图 3.16），经常遇到这个问题。在这种情况下，这个问题通常更尖锐。因为大多数的 IPC 函数涉及到从发送进程拷贝数据到内核，然后从内核拷贝数据到接收进程，所以数据拷贝了两次。因此，我们应当至少使用前面讨论的方法之一来避免过多的数据。

共享内存缓冲区

实际上通过利用空想内存，可以做到避免所有数据拷贝，甚至是在进程之间的数据拷贝。共享内存是由两个或更多进程之间共享的一块内存。每个进程映射共享内存块到它自己地址空间上的一个（可能是不同的）地址上，之后就像它是普通的用户空间内存一样来使用它。

具体的思想是在共享内存中分配一个缓冲区数组，在一个缓冲区中创建消息，然后传递缓冲区的索引到下一个使用方便的 IPC 函数的进程。使用这种方法，实际“移动”的数据表示缓冲区数组索引的整数。例如，我们在图 3.43 中使用 TCP 作为我们的 IPC 机制在进程 1 和进程 2 之间传递整数‘3’。当进程 2 接收到该整数，它知道在 smbarray[3]中有数据在等待它读取。

在图 3.43 中，以虚线框起的两个方框分别表示进程 1 和进程 2 的地址空间，它们相交的部分表示已经映射到它们自己地址空间的共享内存块。缓冲区数组位于共享内存块中，它可以被两个进程存取。进程 1 使用独立的 IPC 频道（在本例子中是 TCP）来通知进程 2 数据已经准备好了，同时也通知进程 2 数据在哪一个数组元素中。

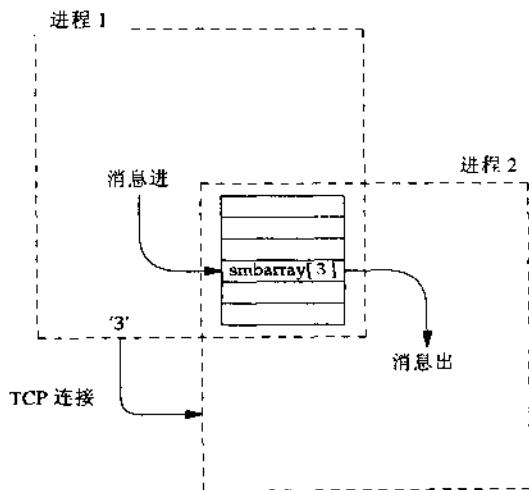


图 3.43 用共享内存缓冲区传递消息

尽管我们只实现了两个进程，该技术对多个进程同样适用。当然，进程 2 也可以通过获取一个共享内存缓冲区，在缓冲区里面创建它的消息并传递索引给进程 1，来传递消息给进程 1。

唯一遗忘的是对缓冲区的同时访问问题——也就是说，避免两个进程同时获得同一个缓冲区的使用权。如我们将在后面演示的那样，这可以很容易地通过互斥信号灯来达到。

» 共享内存缓冲区系统

实现刚才描述的共享缓冲区系统是很容易做到的。涉及到的复杂性主要来自获取并映射共享内存，以及安排缓冲区访问的同步。当然，这些细节是与系统有关的，因此我们给出了 UNIX 和 Windows 的具体实现。

在过问系统有关的部分之前，首先看看 API 和它的具体实现。在用户级别，系统由 5 个函数组成：

```
# include "etcph.h"
void init_smb( int init_freelist );
void *smballoc( void );
>Returns:pointer to a shared memory buffer

void smbfree( void *smbptr );
void smbsend( SOCKET s, void *smbptr );
void *smbrecv( SOCKET s );
>Returns:pointer to a shared memory buffer
```

在使用该系统之前，每个进程调用 `smb_init` 获得共享内存和同步互斥信号灯并执行初始化。其中一个进程必须以 `init_freelist` 设置为 `TRUE` 来调用 `init_smb`。

通过调用 `smballoc` 可以获得一个共享内存缓冲区，该调用返回一个指向新分配的缓冲区的指针。当进程完成了对缓冲区的操作时，可以通过调用 `smbfree` 来释放它并把它返回给系统。

在进程已经在共享内存缓冲区中创建消息之后，它就可以通过调用 `smbsend` 把它传递给另一个进程。正如我们前面讨论的那样，该调用仅仅传递缓冲区的索引。进程通过调用 `smbrecv` 接收对等方的缓冲区，该调用返回刚刚到达的缓冲区的指针。

虽然系统使用 TCP 作为 IPC 方法来传递缓冲区索引，但是 TCP 并不是唯一的方法，甚至也不是最好的方法。TCP 对我们来说很方便，因为它在 UNIX 和 Windows 系统都支持它，而且因为我们可以使用已经有的东西，所以就不去考虑使用其他的 IPC 方法了。在 UNIX 环境中，UNIX 域套接字是另一个可以使用的方法，命名管道也一样可以。`SendMessage`、`QueueUserAPC` 和命名管道是 Windows 中可以使用的方法。

下面我们以 `smballoc` 和 `smbfree` 函数开始分析系统的具体实现，如图 3-44。

头信息

2~8 可以分配的缓冲区保存在自由列表中。在该列表中，每个缓冲区在它的第一个 `sizeof(int)` 字节中包含了列表中下一个缓冲区的索引。这种布置由 `smb_t union` 获取。在缓冲区数组的最后是一个整数，它的值为自由列表中的第一个缓冲区的索引，或者如果自由列表为空的话它就为-1。通过引用为 `smbarry[NSMB].nexti` 以及使用 `FREE_LIST` 为它定义一个方便的别名可以访问该整数。`smbarry` 指针指向缓冲区数组本身。该指针设置为共享内存块，每个进程映射它到自己的地址空间上。我们使用缓冲区的索引而不是地址是因为每个进程可能映射共享内存不同的地址。

smb.c

```

1 #include "etc.h"

2 #define FREE_LIST     smbarry[ NSMB ].nexti

3 typedef union
4 {
5     int nexti;
6     char buf[ SMBUFSZ ];
7 } smb_t;
8 smb_t *smbarry;

9 void *smballoc( void )

```

```

10 {
11   smb_t *bp;

12   lock_buf();
13   if ( FREE_LIST < 0 )
14     error( 1, 0, "out of shared memory buffers\n" );
15   bp = smbarray + FREE_LIST;
16   FREE_LIST = bp->nexti;
17   unlock_buf();
18   return bp;
19 }

20 void smbfree( void *b )
21 {
22   smb_t *bp;

23   bp = b;
24   lock_buf();
25   bp->nexti = FREE_LIST;
26   FREE_LIST = bp - smbarray;
27   unlock_buf();
28 }

```

—smb.c

图 3.44 smballoc 和 smbfree 函数

✓ smballoc

12 函数调用 `lock_buf` 防止其他进程访问自由列表。该函数是实现相关的。系统将在 UNIX 具体实现中使用 SysV 信号灯，而在 Windows 使用互斥机制。

13~16 函数从自由列表中弹出一个缓冲区。如果不能获得缓冲区，函数输出诊断消息并终止。这里也可以返回一个 NULL 指针。

17~18 函数释放自由列表的访问权并返回缓冲区的指针。

✓ smbfree

23~27 锁上自由列表之后，函数通过把它的索引号放在自由缓冲区链的开头把指定的缓冲区放在列表中。然后函数释放自由列表并返回。

下面看看 `smbsend` 和 `smbrecv` 函数。这两个函数只简单地发送和接收缓冲区的整数索

引，该缓冲区将要从一个进程传递到另一个进程。如果愿意的话，它们可以很容易地修改为使用 IPC 的其他形式（图 3.45）。

smb.c

```

1 void smbsend( SOCKET s, void *b )
2 {
3     int index;
4
5     index = ( smb_t * )b - smbarray;
6     if ( send( s, ( char * )&index, sizeof( index ), 0 ) < 0 )
7         error( 1, errno, "smbsend: send failure" );
8
9
10    int index;
11    int rc;
12
13    rc = readn( s, ( char * )&index, sizeof( index ) );
14    if ( rc == 0 )
15        error( 1, 0, "smbrecv: peer disconnected\n" );
16    else if ( rc != sizeof( index ) )
17        error( 1, errno, "smbrecv: readn failure" );
18    return smbarray + index;
19 }
```

smb.c

图 3.45 smbsend 和 smbrecv 函数

✓ smbsend

4~6 函数计算由 b 指向的缓冲区的索引，并使用 send 把它发送到对等方进程。

✓ smbrecv

12~16 函数调用 readn 读取传递过来的缓冲区的索引。如果读取时发生了错误或者没有得到希望获得的字节数，函数就输出错误并终止。

17 否则，函数转换缓冲区索引为指针并把它返回给调用者。

UNIX 具体实现

为了完成共享内存缓冲区的具体实现，我们需要两个组成部分：分配和映射共享内存块的方法，以及防止同时访问自由列表的同步机制。我们使用 SysV 共享内存机制来处理共享内存的分配和映射。而在 Windows 具体中，我们应当使用内存映射文件来代替。对于那些支持 POSIX 共享内存的系统来说，POSIX 共享内存也是一种选择。

我们仅使用两个 SysV 共享内存系统调用：

```
# include <etcp.h>

int shmget( key_t key, size_t size, int flags );
>Returns: shared memory segment ID on success, -1 on error

void *shmat( int segid, const void *baseaddr, int flags );
>Returns: base address of segment if OK, -1 on error
```

`shmget` 系统调用用来分配一块共享内存段。第一个参数 `key` 是一个系统范围内唯一的整数，它命名了该内存段。我们使用的整数具有 ASCII 表示 `SMBM`。

命名空间的使用和文件系统的使用不同，这被认为是 SysV IPC 机制失败的主要原因之一。函数 `ftok` 可以用语映射文件名到一个 IPC 键，但是这种映射并不是唯一的。实际上，正如 [Stevens 1999] 中介绍的那样，标准的 SVR4 版本的 `ftok` 会发生冲突（也就是说，两个文件名称映射到同一个键）的概率大约为 75%。

参数 `size` 指定共享内存段的字节大小，很多 UNIX 系统内核把 `size` 取整为系统内存页的倍数。参数 `flag` 用于指定权限和其他标志，值 `SHM_R` 和 `SHM_W` 分别指定拥有者的读权限和写权限，组权限和其他权限通过向右移动 3 字节（组）或 6 字节（其他）来指定。这样，组写权限指定为 `(SHM_W >> 3)`，而其他读权限指定为 `(SHM_R >> 3)`。如果 `IPC_CREATE` 是 `flags` 参数的组合之一，该段就会在不存在时创建。另外，如果 `IPC_EXCL` 是 `flags` 参数的组合之一，如果内存段已经存在了，`shmget` 将返回 `EEXIST` 错误。

`shmget` 调用仅创建或访问共享内存段。为了映射它到一个特定的进程地址空间，我们使用了 `shmat` 调用。参数 `segid` 是 `shmget` 返回的段描述符。我们可以指定地址，希望内核映射内存段到 `baseaddr`，但是该参数通常总是指定为 `NULL`，告诉内核选择地址。当 `baseaddr` 为非 `NULL` 时，参数 `flags` 用于帮助控制对内存段映射地址的取整。

前面我们已经提到了，使用 SysV 信号灯来创建系统的互斥机制。虽然使用 SysV 信号灯时有些困难，包括我们已经讨论的有关共享内存的命名空间问题，但是因为它们在当今的 UNIX 系统中普遍存在，所以我们使用它就是因为这个原因。对于共享内存，我们必须在开始使用它之前首先分配并初始化信号灯。

`semget` 调用和 `shmget` 调用相似：它分配一个信号灯并返回它的 ID。参数 `key` 的作用和 `shmget` 调用的对应参数相似：命名信号灯。在 SysV IPC 中，信号灯以集合的方式分配，参数 `nsems` 告诉分配集合中有多少个信号灯。参数 `flags` 的作用和 `shmget` 中对应的参数类似。

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

int shmget( key_t key, int nsems, int flags );
>Returns: semaphore ID if OK, -1 on error

int semctl( int semid, int semnum, int cmd, ... );
>Returns: non-negative number if OK, -1 on error

Int semop( int semid, struct sembuf *oparray, size_t nops );
>Returns: 0 if OK, -1 on error
```

我们使用 `semctl` 来设置信号灯的初始值。它也可以用于设置并获得和信号灯关联的不同控制值。参数 `semid` 是 `semget` 返回的信号灯 ID。参数 `semnum` 指示该命令引用集合中的哪一个信号灯。因为我们只需要分配一个信号灯，所以该值对于我们来说总为零。参数 `cmd` 是我们希望执行的实际控制操作。该函数可能还有更多的参数，在函数原型中用 ‘...’ 代替。

`semop` 调用用于增加或减少信号灯。当进程试图减少信号灯为负值时，进程就会进入睡眠状态，知道其他进程增加信号灯的值到等于或大于该进程试图减少的数量。因为我们使用信号灯作为互斥信号，所以锁定自由列表时减 1，释放自由列表时加 1。因为我们初始化信号灯的值为 1，所以它的效果就是：当一个进程试图锁定一个已经锁定了自由列表，该进程就会被挂起。

参数 `semid` 是由 `semget` 返回的信号灯 ID。参数 `oparray` 指向一个 `sembuf` 结构的数组，为集合中的一个或多个信号灯指定操作。参数 `nops` 指定 `oparray` 指向的数组中的条目数。

这里给出的 `sembuf` 结构，指定哪一个信号灯将起作用 (`sem_num`)，增加或减少多少 (`sem_op`)，以及两个特殊操作标志 (`sem_flg`)：

```
struct sembuf {
    u_short sem_num;           /* semaphore # */
    short sem_op;              /* semaphore operation */
    short sem_flg;             /* operation flags */
};
```

可以给 `sem_flg` 指定的两个标志位是：

- IPC_NOWAIT——如果操作导致信号灯的值变为负值的话，指示 semop 返回 EAGAIN 错误而不是让进程进入睡眠状态。
- SEM_UNDO——如果进程终止，指示 semop 撤消任何信号灯操作的影响。它具有释放互斥信号灯的效果。

现在我们可以分析以 UNIX 为中心的共享内存缓冲区系统的代码。所有的系统有关的代码都放置在图 3.46 中给出的 init_smb 函数中。

smb.c

```

1 #include <sys/shm.h>
2 #include <sys/sem.h>
3 #define MUTEX_KEY      0x534d4253 /* SMBS */
4 #define SM_KEY          0x534d424d /* SMBM */
5 #define lock_buf()      if ( semop( mutex, &lkbbuf, 1 ) < 0 ) \
6                           error( 1, errno, "semop failed" )
7 #define unlock_buf()    if ( semop( mutex, &unlkbbuf, 1 ) < 0 ) \
8                           error( 1, errno, "semop failed" )
9 int mutex;
10 struct sembuf lkbbuf;
11 struct sembuf unlkbbuf;

12 void init_smb( int init_freelist )
13 {
14     union semun arg;
15     int smid;
16     int i;
17     int rc;

18     lkbbuf.sem_op = -1;
19     lkbbuf.sem_flg = SEM_UNDO;
20     unlkbbuf.sem_op = 1;
21     unlkbbuf.sem_flg = SEM_UNDO;
22     mutex = semget( MUTEX_KEY, 1,
23                     IPC_EXCL | IPC_CREAT | SEM_R | SEM_A );
24     if ( mutex >= 0 )
25     {
26         arg.val = 1;
27         rc = semctl( mutex, 0, SETVAL, arg );

```

```

28     if ( rc < 0 )
29         error( 1, errno, "semctl failed" );
30 }
31 else if ( errno == EEXIST )
32 {
33     mutex = semget( MUXKEY, 1, SEM_R | SEM_A );
34     if ( mutex < 0 )
35         error( 1, errno, "semctl failed" );
36 }
37 else
38     error( 1, errno, "semctl failed" );

39 smid = shmget( SMKEY, NSMB * sizeof( smb_t ) + sizeof( int ),
40                 SHM_R | SHM_W | IPC_CREAT );
41 if ( smid < 0 )
42     error( 1, errno, "shmget failed" );
43 smbarray = ( smb_t * )shmat( smid, NULL, 0 );
44 if ( smbarray == ( void * )-1 )
45     error( 1, errno, "shmat failed" );

46 if ( init_freelist )
47 {
48     for ( i = 0; i < NSMB - 1; i++ )
49         smbarray[ i ].nexti = i + 1;
50     smbarray[ NSMB - 1 ].nexti = -1;
51     FREE_LIST = 0;
52 }
53 }

```

smb.c

图 3.46 (UNIX) init_smb 函数

／ 定义和全局变量

- 3~4 函数定义了共享内存 (SMBM) 和信号灯 (SMBS) 键。
- 5~8 函数也根据信号灯操作定义了加锁和释放原语。
- 9~11 函数为用作互斥信号的信号灯声明了一些变量。

✍ 分配和初始化信号灯

18~21 函数初始化用于加锁和释放自由列表的信号灯操作。

22~38 这些代码创建并初始化信号灯。函数在参数 `flags` 中设置 `IPC_EXCL` 和 `IPC_CREATE` 来调用 `semget`。这将导致 `semget` 创建信号灯，除非它已经存在了。如果信号灯不存在，`semget` 返回一个信号灯 ID。在这种情况下，函数初始化信号灯为 1（非锁定状态）。如果信号灯已经存在，函数以不指定 `IPC_EXCL` 和 `IPC_CREATE` 的方式再次调用 `semget`，获得信号灯 ID。[Stevens 1999]中指出，这里存在竞争条件（race condition），但是它并不影响我们，因为服务器在调用 `listen` 之前调用了 `init_smb`，而且客户端知道 `connect` 返回才会调用它。

[Stevens 1999]讨论了该竞争条件以及一般情况下如何避免它。

✍ 分配、选用和初始化共享内存缓冲区

39~45 函数分配并选用共享内存段。如果该内存段已经存在，`shmget` 仅仅返回它的 ID。

46~53 如果 `init_smb` 以 `init_freelist` 设置为 TRUE 的方式调用，函数将新分配的缓冲区压入到自由列表中并返回。

✖ Windows 具体实现

在演示系统之前，先看看 Window 版本的具体实现。前面已经提到了，所有的系统有关的代码都在 `init_smb` 函数中，在 Windows 系统中创建一个互斥信号很简单，只需要调用 `CreateMutex` 函数：

```
#include <windows.h>

HANDLE CreateMutex( LPSECURITY_ATTRIBUTES lpsa,
                    BOOL fInitialOwner, LPTSTR lpszMutexName );

>Returns: A handle to the mutex if OK, NULL otherwise
```

参数 `lpsa` 是一个指向安全结构的指针。因为我们将不使用该特性，所以指定它为 `NULL`。参数 `fInitialOwner` 指示互斥信号的创建者是否为初始的拥有者，也就是说，该互斥信号是否初始为锁定的。参数 `lpszMutexName` 命名信号灯，以便其他进程也可以访问它。如果互斥信号已经存在，`CreateMutex` 仅返回互斥信号的句柄。

锁定和释放互斥信号由 `WaitForSingleObject` 和 `ReleaseMutex` 调用来处理：

其他的调用，如 `WaitForMultipleObjects` 也可以用于锁定互斥信号。如果互斥已经发出了信号（非锁定状态），所有的这些调用就都以互斥信号锁定的结果立即返回。否则线程进入睡眠状态，直到互斥信号再次变为非锁定状态。

```
#include <windows.h>

DWORD WaitForSingleObject( HANDLE hObject, DWORD dwTimeout );
>Returns: WAIT_OBJECT_0 (0) if OK, nonzero value otherwise

BOOL ReleaseMutex( HANDLE hMutex );
>Returns: TRUE on success, FALSE otherwise
```

`WaitForSingleObject` 调用的参数 `hObject` 是对象的句柄（在这个例子中是互斥信号），它是等待的对象。如果由 `hObject` 指定的对象已经发出了信号，`WaitForSingleObject` 就把它置为非发送信号状态并返回。如果对象没有发送信号，它就让调用线程睡眠，直到对象成为发送信号状态为止。这时，`WaitForSingleObject` 置对象为非发送信号状态并唤醒睡眠的线程。参数 `dwTimeout` 指定线程愿意等待对象变发送信号状态的以毫秒为单位的时间数，如果该计时器在对象变发送信号状态之前终止，`WaitForSingleObject` 返回 `WAIT_TIMEOUT`。可以通过指定 `dwTimeout` 参数为 `INFINITE` 来禁用超时函数。

当线程在由互斥信号保护的临界区结束时，通过调用 `ReleaseMutex` 释放由参数 `hMutex` 指定的互斥信号句柄。

在 Windows 系统下，通过内存映射文件到每个需要访问共享内存的进程获得共享内存段。这和使用 UNIX 系统下的 `mmap` 系统调用相似。为了达到这个目的，首先创建通过调用 `CreateFile` 以通常的方法创建一个文件。一旦获得了该文件，我们就用 `CreateFileMapping` 创建它的映射，然后用 `MapViewOfFile` 把它映射到我们的地址空间。

```
#include <windows.h>

HANDLE CreateFileMapping( HANDLE hFile, LPSECURITY_ATTRIBUTES lpsa,
    DWORD fdwProtect, DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow, LPSTR lpszMapName );
>Returns: Handle to file mapping if OK, NULL on error

LPVOID MapViewOfFile( HANDLE hFileMapObj, DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
    DWORD dwBytesToMap );
>Returns: Address at which memory mapped if OK, NULL on error
```

`CreateFileMapping` 的 `hFile` 参数是将要映射的文件的句柄。参数 `lpsa` 是安全信息，我们不使用它。参数 `fdwProtect` 指定内存对象的访问权限，可以为 `PAGE_READONLY`、`PAGE_READWRITE` 或 `PAGE_WRITECOPY`，如果选择了最后一个值，就导致进程在写数据给它时内核创建数据的私有拷贝。我们使用 `PAGE_READWRITE`，这是因为我们将既要

读也要写该内存。还有其他的标志可以组合到该域来控制内存页的存储。因此我们不需要这些，所以在这里不讨论它们。dwMaximumSizeHigh 和 dwMaximumSizeLow 一起组成 64 位的内存对象。最后，lpszMapName 是内存对象的名称，提供它是为了其他进程可以用它的名称访问它。

一旦创建了，内存对象就可以用 MapViewOfFile 映射到每个进程的地址空间。hFileMapObj 是由 CreateFileMapping 返回的句柄。我们在 dwDesiredAccess 中再次指定希望获得的访问权限。该参数可以取值为 FILE_MAP_WRITE（写和读访问权限），FILE_MAP_READ（只读访问权限）、FILE_MAP_ALL_ACCESS（和 FILE_MAP_WRITE 一样）和 FILE_MAP_COPY。如果指定了最后一个值，进程在写数据时就创建数据的一个私有拷贝。参数 dwFileOffsetHigh 和 dwFileOffsetLow 指定映射在文件中开始的位置。我们指定这两个参数都为零，这是因为我们希望映射整个文件。将要映射的内存的数量由 dwBytesToMap 指定。

请参阅[Richter 1997]，了解有关 Windows 版本的 init_smb 的详细信息。

现在我们可以给出 Windows 版本的 init_smb 了。从图 3.47 可以看出，它本质上和 UNIX 版本上一样的。

—smb.c

```

1 #define FILENAME "./smbfile"
2 #define lock_buf()    if ( WaitForSingleObject( mutex, INFINITE ) \
3                                != WAIT_OBJECT_0 ) \
4                                error( 1, errno, "lock_buf failed" )
5 #define unlock_buf() if ( !ReleaseMutex( mutex ) ) \
6                                error( 1, errno, "unlock_buf failed" )
7 HANDLE mutex;

8 void init_smb( int init_freelist )
9 {
10    HANDLE hfile;
11    HANDLE hmap;
12    int i;

13    mutex = CreateMutex( NULL, FALSE, "smbmutex" );
14    if ( mutex == NULL )
15        error( 1, errno, "CreateMutex failed" );
16    hfile = CreateFile( FILENAME,
17                        GENERIC_READ | GENERIC_WRITE,
18                        FILE_SHARE_READ | FILE_SHARE_WRITE,

```

```

19     NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL );
20 if ( hfile == INVALID_HANDLE_VALUE )
21     error( 1, errno, "CreateFile failed" );
22 hmap = CreateFileMapping( hfile, NULL, PAGE_READWRITE,
23     0, NSMB * sizeof( smb_t ) + sizeof( int ), "smbarray" );
24 smbarray = MapViewOfFile( hmap, FILE_MAP_WRITE, 0, 0, 0 );
25 if ( smbarray == NULL )
26     error( 1, errno, "MapViewOfFile failure" );

27 if ( init_freelist )
28 {
29     for ( i = 0; i < NSMB - 1; i++ )
30         smbarray[ i ].nexti = i + 1;
31     smbarray[ NSMB - 1 ].nexti = -1;
32     FREE_LIST = 0;
33 }
34 }
```

smb.c

图 3.47 (Windows) init_smb 函数

为了试验我们的共享内存缓冲区系统，我们编写了简短的客户端（图 3.48）和服务器（图 3.49）程序。

smbc.c

```

1 #include "etcps.h"

2 int main( int argc, char **argv )
3 {
4     char *bp;
5     SOCKET s;

6     INIT();
7     s = tcp_client( argv[ 1 ], argv[ 2 ] );
8     init_smb( FALSE );
9     bp = smballoc();
10    while ( fgets( bp, SMBUFSZ, stdin ) != NULL )
11    {
```

```

12     smb_send( s, bp );
13     bp = smb_alloc();
14 }
15 EXIT( 0 );
16 }
```

smbc.c

图 3.48 使用共享内存缓冲区系统的客户端

```

1 #include "etcp.h"

2 int main( int argc, char **argv )
3 {
4     char *bp;
5     SOCKET s;
6     SOCKET s1;

7     INIT();
8     init_smb( TRUE );
9     s = tcp_server( NULL, argv[ 1 ] );
10    s1 = accept( s, NULL, NULL );
11    if ( !isValidSock( s1 ) )
12        error( 1, errno, "accept failure" );
13    for ( ; ; )
14    {
15        bp = smbrecv( s1 );
16        fputs( bp, stdout );
17        smbfree( bp );
18    }
19    EXIT( 0 );
20 }
```

smbs.c

图 3.49 使用共享内存缓冲区系统的服务器

当运行这些客户端和服务器时，我们将获得预期的结果：

```

bsd: $ smbc localhost 9000
Hello
World!
^C
bsd: $

bsd: $ smbs 9000
Hello
World!
smbs: smbrecv: peer disconnected
bsd: $

```

注意，因为 smbc 直接从标准输入读取每一行到共享内存缓冲区，而且因为 smbs 直接从共享内存缓冲区中写每一行到标准输出，所以没有必要进行数据拷贝。

小结

本节分析了避免不必要的数据拷贝的方法。在许多网络应用程序中，从缓冲区中拷贝数据到另一个缓冲区占用了 CPU 使用的绝大部分。

为了进行进程间的通信，本节开发了一个共享内存缓冲区系统，它允许我们在一个进程到另一个进程之间传递数据拷贝，开发了包括 UNIX 和 Windows 的版本。

技巧 27 在使用之前置 sockaddr_in 为零

虽然我们通常只使用 sockaddr_in 结构中的三个域 sin_family、sin_port 和 sin_addr，但大多数的具体实现中还有其他的域。例如，许多具体实现有一个 sin_len 域包含结构的长度。例如，该域在 BSD 具体实现 4.3BSD 版本 Reno 派生的系统以及更后的系统中就存在。另一方面，Winsock 具体实现却没有该域。

如果拿 BSD 派生的系统 FreeBSD 中的 sockaddr_in 结构：

```

struct sockaddr_in{
    u_char    sin_len;
    u_char    sin_family;
    u_short   sin_port;
    struct in_addr sin_addr;
    char     sin_zero [8];
};

```

和 Winsock 具体实现中的：

```

struct sockaddr_in {
    short    sin_family;

```

```

    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};


```

作比较，就可以看出它们和其他大多数具体实现都共享着一个额外的域 `sin_zero`。虽然该域没有被使用（它的用途是为了使 `sockaddr_in` 结构为 16 字节长），它必须设置为零。

这么做是因为一些具体实现在绑定地址到套接字时，用地址结构和关联到每个接口上的地址进行二进制比较。这些代码要求 `sin_zero` 域必须为零。请参阅[Wright and Stevens 1995]，了解详细信息。

因为无论如何我们必须置 `sin_zero` 为零，所以在使用它之前置整个结构为零不失为一个很好的习惯。通过这种方法，所有的域都清空了，避免了没有写入文档中的域以及它们用法的影响。回忆一下图 2.16 中的程序，函数 `set_address` 所做的第一件事情就是调用 `bzero` 清空整个 `sockaddr_in` 结构。

技巧 28 不要忘记字节的性别

根据体系结构的不同，现代计算机以不同的方式存储整数。例如，考虑一下 32 位的整数 305419896 (0x12345678)。有些机器以最重要字节优先的方式存储这个 4 字节的整数（所谓的 big endian 格式）：

12	34	56	78
----	----	----	----

而有的机器以最不重要字节优先的方式存储（little endian 格式）：

78	56	34	12
----	----	----	----

术语 little endian 和 big endian 来自[Cohen 1981]，该文档把正在进行的哪一种格式是“最好”的争论比作 Jonathan Swift 的 Gulliver's Travels 中的 Lilliputian 派系斗争，Lilliputian 对鸡蛋应当在大头还是小头打开进行了无休止的争斗。虽然一些计算机也使用其他的存储格式，但是实际上今天所有的机器都使用 big endian 或 little endian。

通过编写一个测试程序看看整数 0x12345678 是如何存储的（图 3.50）就可以很轻松地判断给定机器所使用的格式。

endian.c

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

```

2 #include <sys/types.h>
3 #include "etcp.h"
4 int main( void )
5 {
6     u_int32_t x = 0x12345678; /* 305419896 */
7     unsigned char *xp = ( char * )&x;
8
9     printf( "%0x %0x %0x %0x\n",
10            xp[ 0 ], xp[ 1 ], xp[ 2 ], xp[ 3 ] );
11 }

```

endian.c

图 3.50 测试整数存取格式的程序

当在基于 Intel 的 bsd 系统上运行该程序时，获得如下结果：

```

bsd: $endian
78 56 34 12
bsd: $

```

说明该机器实际上是一台 little endian 机器。

存储格式的特殊选择通常生动地称作字节性别（byte sex）。这对我们是很重要的，因为 big endian 和 little endian 的机器（以及使用其他格式的机器）经常使用 TCP/IP 协议进行会话。因为诸如源地址和目的地址、端口号、数据报长度、窗口大小等信息都是以整数的方式进行交换，所以双方必须对整数的表示进行统一。

为了避免互操作的问题，所有整数的协议数据都转换成网络字节顺序（network byte order），它是定义位 big endian 的。大多数的事情都由下面的协议帮我们处理，但是我们需要指定网络地址、端口号，而且有时还要指定其他头信息数据到 TCP/IP 中。当我们做这些事情的时候，必须总是以网络字节顺序的方式指定它们。

为了帮助我们解决这个问题，有两个函数将数据从主机格式转换成网络字节格式，还有两个函数将数据从网络字节格式转换成主机格式。下页的定义是 POSIX 的，一些版本的 UNIX 不是在 netinet/in.h 头文件中定义这些函数。`uint32_t` 和 `uint16_t` 分别是无符号 32 位和 16 位整数的 POSIX 定义，它们不一定在所有的具体实现中都定义了。而且 `htonl` 和 `ntohl` 总是采用并返回无符号 32 位整数，不管是在 UNIX 还是 Winsock 具体实现中都是这样。类似地，`htons` 和 `ntohs` 总是采用并返回无符号 16 位整数，这和具体实现无关。

```

#include <netinet/in.h>          /* UNIX */
#include <winsock2.h>           /* Winsock */

uint32_t htonl( uint32_t host32 );
uint16_t htons( uint16_t host16 );

Both return: an integer in network byte order

uint32_t ntohl( uint32_t network32 );
uint16_t ntohs( uint16_t network16 );

Both return: an integer in host format

```

这些名称中的 ‘l’ 和 ‘s’ 代表 “long” 和 “short”。只有当这些函数最初在 32 位机器上的 4.2BSD 系统中实现时，它们才有意义，这是因为长整数是 32 位的，而短整数是 16 位的。随着 64 位机器的来临，这种理解又不对了，因此我们应当小心地记住，‘l’ 函数是在 32 位整数上工作的但并非是长整数，而 ‘s’ 函数是在 16 位整数上工作的但并非是短整数。一种方便的记忆方式是把 ‘l’ 记作工作在协议头中出现的比较长的整数域，而 ‘s’ 函数工作在比较短的整数域上。

主机到网络函数（htonl 和 htons）转换主机格式的整数位网络字节顺序，而网络到主机函数（ntohl 和 ntohs）的功能相反。注意在 big endian 机器上，这些函数不做任何事情，通常定义为如下的宏：

```
#define htonl(x) (x)
```

little endian（和其他体系结构）的机器以系统相关的方式定义它们。问题是不需要知道或关心我们的机器是 big endian 还是 little endian，这是因为这些函数总是替我们做了“正确的事情”。

我们同时应当理解这些函数仅仅需要由协议来进行域检验。也就是说，因为 IP、UDP 和 TCP 都把用户数组看作没有结构的字节集合，所以不关心用户数据中整数是否为网络字节顺序。但是我们应当在我们要发送的数据上使用 ntohs 和 htons 函数，作为可交互性的一种帮助。即使应用程序的目的是只运行在一个平台上，但是终究有一天，我们将不得不把它移植到另一个平台上，我们的额外努力终将得到回报。

不同体系结构的机器之间的数据转换问题通常是一个很困难的问题。许多工程师仅通过把所有的数据转换为 ASCII（或者如果涉及到 IBM 大型机的话，就转换为 EBCDIC）来简单地处理它。处理这个问题的另一个方法是 Sun 的远程过程调用（RPC）工具中的外部数据表示（External Data Representation, XDR）组件。XDR 在 RFC 1832[Srinivasan 1995]中定义，它是将不同类型的数据进行编码的一套规则，也是描述数据编码的一种语言。虽然 XDR 目的是作为 RPC 的一部分使用的，但是我们也可以在不使用 RPC 的情况下直接使用它。[Stevens 1999]讨论了 XDR 以及如何在不使用 RPC 的情况下使用它。

最后，我们应当注意解析函数如 `gethostbyname` 和 `getservbyname`（请参阅技巧 29）都是以网络字节顺序的方式返回它们的值。下面的代码：

```
struct servent *sp;
struct sockaddr_in *sap;

sp = getservbyname ( name, protocol );
sap->sin_port = htons ( sp->s_port );
```

是错误的，除非它是运行在 big endian 机器上，否则将会导致错误。

小结

本节分析了 TCP/IP 是如何使用整数的标准表示方式称作网络字节顺序在协议头信息中处理整数的。从中知道了函数 `htonl`、`htons`、`ntohl` 和 `ntohs` 可以用于转换整数为网络字节顺序，或者从网络字节顺序转换回整数，最后我们提到 XDR 对于机器之间的一般数据转换通常是很常用的。

技巧 29 不要在应用程序中 对 IP 地址和端口号硬编码

程序仅有两种方法来获得 IP 地址和端口号：

- (1) 以命令行参数，或在 GUI 界面的情况下，从对话框或类似的方式中获得。
- (2) 从一个或多个解析函数如 `gethostbyname` 或 `getservbyname` 中获得。

`getservbyname` 其实并不是一个解析函数（也就是说，它不是映射名称到 IP 地址或者相反的 DNS 客户端的一部分），但是把它列出来是因为它们执行相似的服务，而且经常使用它们。

我们应当永远不要在程序中硬编码这些参数，也不要把它们放在自己的（私有）配置文件中。UNIX 和 Windows 都提供了标准的方法来获取这些信息，我们应当很好地利用它们。

现在的许多 IP 地址都是动态地使用动态主机配置协议（dynamic host configuration protocol, DHCP）来分配，这就为避免给应用程序指定固定的 IP 地址提供了一个硬性规定。有些人甚至认为 DHCP 的流行以及 IPv6 地址的长度和复杂性意味着我们根本不应该给应用程序提供一个数字的 IP 地址，而是应当提供一个唯一的主机名称，让应用程序通过

`gethostbyname` 或其他类似函数把它们解析为 IP 地址。即使没有使用 DHCP，维护和网络管理的要求也强烈反对硬编码这些信息或者把它放置非标准的位置。例如，如果网络重新分配地址，任何硬编码的应用程序都将崩溃。

在一个快速但不规范的程序里硬编码这些值是很诱人的，这是因为我们不用去劳烦 `getXbyY` 或相关的函数。不幸的是，一旦创建了，这些快速但不规范的程序就会流行起来，而且有时甚至还会成为产品。我们的模板程序以及从它们派生出来的库函数（请参阅技巧 4）的一个巨大的优势是这些代码已经为我们编写好了，因此没有必要去抄近路。

在我们继续之前，让我们考虑一下一些解析函数以及如何使用它们。我们已经多次见到 `gethostbyname`：

```
#include <netdb.h>          /* UNIX */
#include <winsock2.h>         /* WinSock */

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

Returns: Pointer to a hostent structure if OK, NULL with error in h_errno otherwise

我们传递主机名称给 `gethostbyname` 函数，该函数返回 `hostent` 结构的一个指针：

```
struct hostent {
    char *h_name;           /* official name of host */
    char **h_aliases;       /* alias list */
    int h_addrtype;         /* host address type */
    int h_length;            /* length of address */
    char **h_addr_list;     /* list of addresses from DNS */
#define h_addr h_addr_list[0] /* first address*/
}
```

`h_name` 域指向主机的“正式”名称，而 `h_aliases` 域指向正式名称的别名列表。根据地址是 IPv4 地址还是 IPv6 地址，`h_addrtype` 域取值为 `AF_INET` 或 `AF_INET6`。同样地，根据地址的类型 `h_length` 为 4 或者 16。主机的类型 `h_addrtype` 的所有地址返回在由 `h_addr_list` 指向的列表中。`h_addr` 定义用作列表中第一个（有可能是唯一的）地址的别名。因为 `gethostbyname` 返回地址的列表，所以应用程序可以尝试列表中的每一个地址，直到成功连接所希望的地址为止。

使用 `gethostbyname` 必须记住 4 件事情：

(1) 如果主机既支持 IPv4 也支持 IPv6，它也只返回一种类型的地址。在 UNIX 系统中，地址的类型根据解析选项 `RES_USE_INET6` 的不同返回不同的值，`RES_USE_INET6` 可以由 `res_init` 调用、环境变量或 DNS 配置文件中一个选项来设置。Winsock 系统中，它总是返回一个 IPv4 地址。

(2) hostent 结构在静态内存中分配，这意味着 gethostbyname 是不可再进入的。

(3) 静态 hostent 结构中的指针指向其他静态或分配的内存，因此如果我们希望拷贝它的话，就对结构进行深度拷贝（deep copy）。也就是说，我们必须为它分配内存，然后从 hostent 结构本身分开拷贝到由域指向的数据。

(4) 正如技巧 28 中讨论的那样，因为 h_addr_list 域指向的地址已经是网络字节顺序了，所以不能对它们使用 htonl 函数。

也可以使用 gethostbyaddr 函数映射 IP 地址到主机名称：

```
#include <netdb.h>      /* UNIX */
#include <winsock2.h>    /* WinSock */

struct hostent *gethostbyaddr( const char *addr, int len, int type );
```

Returns: Pointer to a hostent structure if OK, NULL with error in h_errno otherwise

虽然参数 addr 的类型为 `char*`，但是它却指向一个 `in_addr`（或者在 IPv6 中为 `in6_addr`）结构。该结构的长度由 `len` 参数指定，其类型（`AF_INET` 或 `AF_INET6`）由 `type` 指定。前面对 `gethostbyname` 的注释对 `gethostbyaddr` 也一样适用。

对于支持 IPv6 的主机来说，因为 `gethostbyname` 不能指定希望返回的地址类型，所以该函数并不能令人满意。为了支持 IPv6（以及其他）地址类型，函数 `gethostbyname` 的功能由 `gethostbyname2` 扩展了，它允许获取特定的地址类型：

```
#include <netdb.h>

struct host *gethostbyname2( const char *name, int af );
```

Returns: Pointer to a hostent structure if OK, NULL with error in h_errno otherwise

参数 `af` 是地址类型，目前来说它的值为 `AF_INET` 或者为 `AF_INET6`。Winsock 没有 `gethostbyname2`，但是它使用功能更强的（但是更复杂）的 `WSALookupServiceNext` 接口来代替。

`IPv4/IPv6` 的互操作性是处理这两种不同类型的地址的一个很大的问题，`gethostbyname2` 只提供了一种方法来达到这个目标，这个问题在 [Stevens 1998] 中有深入的讨论，该文档还开发了 POSIX 中 `getaddrinfo` 函数的具体实现，该函数以协议无关的方式为两种类型的地址提供了一种方便的方法。通过使用 `getaddrinfo` 函数，可以编写无缝地工作在 IPv4 和 IPv6 下的应用程序。

正如让系统（或 DNS）为我们转换主机名称为 IP 地址是很明智的一样，让系统自己管理端口号也是很明智的。我们已经在技巧 18 里讨论了做到这一点的一个方法，本节将介绍另一种方法。和 `gethostbyname` 和 `gethostbyaddr` 在主机名称和 IP 地址之间转换一样，`getservbyname` 和 `getservbyport` 在端口号和它们的符号名称之间进行转换。例如，日期时间服务在端口 13 监听连接（TCP）或数据报（UDP），我们可以通过 telnet 使用该服务：

```
telnet bsd 13
```

然而，这需要我们记住日期时间服务的端口号为 13。幸运的是，telnet 也接收服务端口的符号名称：

```
telnet bsd daytime
```

Telnet 通过调用 `getservbyname` 执行端口和符号名称之间的映射，我们也可以在应用程序中这么做。回忆一下图 2.16，将会看到模板代码实际上已经这么做了。函数 `set_address` 首先假定端口参数是一个 ASCII 码整数并试图把它转换为二进制数，如果该转换失败了，那么函数就调用 `getservbyname` 来查找符号端口名称并把它转换成端口号。

`getsvrbyname` 调用和 `gethostbyname` 类似：

```
#include <netdb.h>      /* UNIX */
#include <winsock2.h>    /* WinSock */

struct servent *getservbyname( const char *name, const char *proto );
```

Returns: Pointer to a servent structure if OK, NULL otherwise

参数 `name` 是要查找的服务的名称——例如为“`daytime`”。如果参数 `proto` 不为 `NULL`，`getservbyname` 返回跟名称和协议都匹配的一个服务；否则，它返回它找出的第一个具有名称 `name` 的服务。结构 `servent` 包含有关服务的信息：

```
struct servent {
    char *s_name;          /* official service name */
    char **s_aliases;      /* alias list */
    int   s_port;           /* port # */
    char *s_proto;          /* protocol to use */
};
```

`s_name` 和 `s_aliases` 域包含服务的正式名称和别名的指针。服务的端口号放置在 `s_port` 域，通常该数字已经为网络字节地址。服务使用的协议（TCP 或 UDP）由 `s_proto` 域指定。我们也可以通过使用 `getservbyport` 函数映射端口号为服务名称：

```
#include <netdb.h>      /* UNIX */
#include <winsock2.h>    /* WinSock */

struct servent *getservbyport( int *port, const char *proto );
```

Returns: Pointer to a servent structure if OK, NULL otherwise

要查找的端口号在参数 `port` 中以网络字节顺序传递进来。参数 `proto` 和前面的函数相应参数一样。

从程序员的观点来看，前面的模板程序和相关的库解决了主机和服务名称相互转换的问题。它们调用适当的函数，我们不需要担心什么。然而，我们确实需要理解如何给系统提供必要的信息，以便模板代码可以提取它。

提供这些数据的最通常的方法是：

- DNS
- Network Information System (NIS) 或 NIS+
- 主机和服务文件

DNS 是一个用于在主机名称和 IP 地址之间相互转换的分布式数据库。

DNS 也用于路由邮件，当我们发送邮件到：

`Jsmith@somecompany.com`

时，DNS 就用于为 `somecompany.com` 定位邮件处理器（或多个处理器），请参阅[Albits and Liu 1998]，了解更多的信息。

数据库中条目的责任由区域（粗略地对应为域）和子区域分担。例如，`bigcompany.com` 可能是一个区域，它可以分解为对应于部门或区域办公室的子区域。每个区域和子区域运行一个或多个 DNS 服务器，这些服务器包含那个（子）区域中所有主机的信息。其他 DNS 服务器可以查询 `bigcompany.com` 服务器解析 Bigcompany 中的主机名称。

DNS 为 UDP 应用程序提供了一个很好的例子。许多 DNS 服务器的通信是通过短 UDP 事务进行的。客户端（通常是解析函数之一）发送包含 DNS 服务器查询的 UDP 数据报。如果在一定时间内没有接收到应答，就尝试另一个可获得的服务器；否则，请求就重新发布给原来的服务器，增加超时值。

现在，绝大多数主机/IP 地址转换都是由 DNS 进行的。甚至没有连接大规模 WAN 的网络也通常使用 DNS，这是因为 DNS 简化了 IP 地址的管理。也就是说，如果网络中增加了新的主机或者已存在主机的 IP 地址改变了，就仅需要更新 DNS 数据库，而不需要更新每台机器上的主机文件。

NIS 和它的后继者 NIS+ 提供了包含网络的多种类型的信息的集中数据库。除了主机名称和 IP 地址之外，NIS 可以管理服务、密码、组和其他网络有关的数据。前面讨论的标准的解析函数可以查询 NIS 数据库。在一些系统中，如果 NIS 不知道询问的主机名称，它就自动地查询 DNS，解析函数可以处理这种情况。

NIS 的优点是它集中了和网络有关数据的管理，这样就简化了大规模网络的管理。一些权威认为不应鼓励使用 NIS，因为密码文件存在潜在的安全危险。NIS+ 引入了一些安全机制，但是很多人仍然认为它不够安全。NIS 在[Brown 1994]中有详细的讨论。

放置主机/IP 转换信息的最后也是最不希望最不标准的地方是每台机器上的主机文件。该文件通常位于 `/etc/hosts` 中，包含网络中主机的名称、别名和 IP 地址。标准的解析函数查

询主机文件。大多数的系统提供了一个系统范围内的配置条目，指定是否应当在 DNS 查询之前或之后查询主机文件。

另一个文件，通常为 /etc/services，列出了服务/端口转换。除非使用了 NIS，否则通常是每台机器维护自己的该文件的拷贝。因为该文件很少改变，所以它没有主机文件出现的管理问题。我们已经在技巧 17 中讨论了服务文件以及它的结构。

使用主机文件的主要缺点是显而易见的管理问题。对于具有多个主机的网络来说，这个问题将会变得很难处理。结果是多数权威人士完全反对使用它们。例如，[Lehey 1996] 中这样说，“网络中之所以不使用 DNS 只有一个原因：那就是你不连接到网络。”

小结

本节我们强烈地建议不要在应用程序中硬编码地址或端口号。本节分析了我们可以在应用程序中使用几个标准方法来获得地址和端口号，然后分析了它们的优缺点。

技巧 30 理解已连接 UDP 套接字

本节我们将讨论使用 UDP 的 connect。从技巧 1 中我们知道 UDP 是一个无连接协议——也就是说它仅仅传输独立的带地址的数据报——因此“连接”的概念似乎放错了地方。然而，回忆一下我们已经在图 3.9 中看到了这样的一个例子，该例子在 inetd 发起的 UDP 服务器中使用 connect 获取一个新的（暂时的）服务器端口，以便 inetd 可以在原来的已知端口上监听数据报。

在我们讨论为什么要在 UDP 套接字上使用 connect 之前，应当清楚这里的“连接”具体是什么意思。对于 TCP 来说，调用 connect 将导致双方之间以三阶段握手（图 3.39）中的初始状态交换信息。部分这些状态信息是每一方的地址和端口号，因此 TCP 的 connect 函数的一个功能可以理解为绑定远程主机的地址和端口号到本地套接字上。

这和调用 bind 有区别，bind 函数的功能是绑定本地地址和端口号到套接字。

对于 DUP 来说，因为双方没有共享状态要交换，所以调用 connect 不会导致任何网络拥塞。而且，它的操作完全是本地的，它仅仅绑定远程主机地址和端口到本地套接字上。

虽然对 UDP 套接字调用 connect 似乎没有很大的用途，但是我们会获得一些额外的效率，它通常用于获得一些本不可以获得的功能。让我们首先从发送者的观点看看连接 UDP 套接字的原因，然后从接收者的观点看看使用它的原因。

我们注意到已连接 UDP 套接字的第一个问题是它从不使用 sendto，而是使用 send 或 write（UNIX）来代替。

我们仍然在一个已连接的 UDP 套接字上使用 sendto，但是指定对等方地址的指针为 NULL，并置其长度为零。我们当然也可以使用 sendmsg，但是我们必须再次设置结构 msghdr 的 msg_name

域为 NULL，并把 msg_namelen 域置零。

当然，这不会有什么收获，但是使用 connect 确实可以获得很大的性能增加。

在 BSD 的具体实现中，sendto 实际就是调用 connect 的特殊情况。当用 sendto 发送数据报时，内核短暂连接套接字，发送数据报，然后断开套接字。在对 4.3BSD 和与其有密切关系的 SunOS4.1.1 的研究中，[Partridge and Pine 1993]发现以这种方式连接和断开连接套接字几乎消耗了传输 UDP 数据报三分之一的处理时间。除了改进用来定位和套接字相关的协议控制块（protocol control block, PCB）之外，他们研究的 UDP 代码在 4.4BSD 及其后代如 FreeBSD 中仍然没有改变。特别是这些堆栈仍然执行短暂的连接和断开连接，因此如果我们要发送一系列的 UDP 数据报给同一个接收者，就有可能通过一开始调用 connect 来提高性能。

虽然前面所描述性能的获得在一些具体实现中已经实现了，但是 UDP 数据报的发送者连接对等方的主要原因是接收不同步错误的通知。为了理解这个问题，假定我们发送一个 UDP 数据报给对等方，但是没有进程在目的端口上监听。对等方的 UDP 返回一个 ICMP 端口不可到达消息，通知本方 TCP/IP 这个事实，但是除非我们连接了套接字，否则应用程序是不能接收到这个通知的。让我们分析一下这是为什么。当我们调用 sendto 时，消息就附加了一个 UDP 消息头，并传递到 IP 层，数据在 IP 层被封装在 IP 数据报中，然后放在接口输出队列上。在数据报已经放置在队列上之后（或者如果没有其他要求等待的事情就发送出去），sendto 返回给应用程序一个成功状态。这之后一段时间（这是为什么使用术语异步的原因），来自对等方的 ICMP 消息到达了，虽然 ICMP 消息包含了 UDP 消息头的拷贝，但是本方堆栈没有那个应用程序发送数据报的记录（回忆一下技巧 1，因为 UDP 是无状态的，所以数据报一旦发送过去，系统就会丢弃数据报的信息）。然而，当连接套接字到对等方时，这些信息就记录在和套接字相关的 PCB 中，本方 TCP/IP 栈就可以用 PCB 和 UDP 消息头的拷贝进行匹配，区分 ICMP 消息是由哪个套接字发送出去的。

为了说明这些问题，从技巧 17 中抽取 udpclient 程序（图 3.8），用它来发送数据报到没有进程监听的端口上：

```
bsd: $ udpclient bsd 9000
Hello,world!
^C
client hangs, terminated manually
bsd: $
```

下面，修改 udpclient，在调用 udp_client 之后加入下列几行：

```
if ( connect ( s, ( struct sockaddr * ) &peer, siezeof ( peer ) ) )
    error ( 1, errno, "connect failed" );
```

如果称该程序为 `udpconn1`, 然后运行它, 将获得以下结果:

```
bsd: $ udpConn1 bsd 9000
Hello, World!
udpconn1: sendto failed: socket is already connected (56)
bsd: $
```

错误是因为在已连接的套接字上使用了 `sendto`。所发生的事情是当 `sendto` 要求 UDP 短暂地连接套接字, UDP 发现该套接字已经连接了, 因此返回 `EISCONN` 错误。

通过改变 `udpconn1` 中的 `sendto` 为:

```
rc = send ( s, buf, strlen ( buf ), 0 );
```

可以改正这个错误。

如果称新程序为 `updconn2` 并运行它, 将获得以下结果:

```
bsd: $ updclient2 bsd 9000
Hello, World!
updconn2: recvfrom failed: Connection refused (61)
bsd: $
```

这次, 我们从 `recvfrom` 获得一个 `ECONNREFUSED` 错误。该错误是 ICP 端口不可到达错误已经传递到本方应用程序的结果。

通常没有任何原因要求接收者连接到对等方 (当然, 除非它也要成为发送者)。然而, 有一种情况下它是很有用的。回忆以下技巧 1 中的电话/邮件比喻。因为 TCP 是一个基于连接的协议, 每一方都知道对等方是谁, 所以可以确信它获得的每一个字节都是来自对等方的。TCP 连接就像一个私有电话线——线上没有其他任何一方。

另一方面, 接收 UDP 数据报的应用程序就像邮箱一样。跟任何人可以发送信件到一个给定的邮箱一样, 主机上的任何应用程序可以发送给接收程序一个数据报, 只要数据报具有合适的地址和端口号。

有时只希望从一个应用程序接收数据报。接收应用程序可以通过连接对等方达到这个目的。之后, 只有来自对等方的数据报才能传递到该应用程序。为了明白这是如何工作的, 我们编写了一个 UDP echo 服务器, 它连接到第一个客户端并给它发送一个数据报 (图 3.51)。

udpconnser.c

```
1 #include "etcp.h"
2 int main( int argc, char **argv )
```

```

3  {
4      struct sockaddr_in peer;
5      SOCKET s;
6      int rc;
7      int len;
8      char buf[ 120 ];
9
10     INIT();
11     s = udp_server( NULL, argv[ 1 ] );
12     len = sizeof( peer );
13     rc = recvfrom( s, buf, sizeof( buf ), 0,
14                     ( struct sockaddr * )&peer, &len );
15     if ( rc < 0 )
16         error( 1, errno, "recvfrom failed" );
17     if ( connect( s, ( struct sockaddr * )&peer, len ) )
18         error( 1, errno, "connect failed" );
19
20     while ( strncmp( buf, "done", 4 ) != 0 )
21     {
22         if ( send( s, buf, rc, 0 ) < 0 )
23             error( 1, errno, "send failed" );
24         rc = recv( s, buf, sizeof( buf ), 0 );
25         if ( rc < 0 )
26             error( 1, errno, "recv failed" );
27     }
28     EXIT( 0 );
29 }
```

udpconnser.c

图 3.51 连接的 UDP echo 服务器

9~15 程序执行标准的 UDP 初始化并接收第一个数据报，在 *peer* 中保存发送者的地址和端口号。

16~17 程序连接对等方。

18~25 程序循环遍历响应数据报，直到接收包含唯一单词“done”的数据报为止。

我们可以使用 *udpconn2* 和 *udpconnsev* 做实验。首先，我们启动服务器在端口 9000 上监听数据报。

```
udpconnser v 9000
```

然后在不同的窗口启动 udpconn2 的两个拷贝。

<pre>bsd: \$ udpconn2 bsd 9000 one one three three done ^C bsd: \$</pre>	<pre>bsd: \$ udpconn2 bsd 9000 two udpconn2: recvfrom failed: Connection refused (61) bsd: \$</pre>
--	---

在第一个窗口中输入“one”，udpconnser v 适时地回送回来。然后在第二个窗口中输入“two”，但是 recvfrom 返回 ECONNREFUSED 错误。这是因为服务器已经连接到 udpconn2 的第一个拷贝而且没有接收来自任何其他地址/端口组合的数据报，所以 UDP 返回 ICMP 端口不可到达消息。

当然，udpconn2 的两个拷贝都有相同的源地址，但是 TCP/IP 堆栈分配的暂时地址是不同的。

在第一个窗口中输入“three”，可以验证 udpconnser v 仍然正确操作，然后输入“done”终止服务器。最后手工中断 udpconn2 的第一个拷贝。

正如我们看到的那样，不仅 udpconnser v 拒绝接收来自非连接的任何地址/端口的数据报，而且它还通过 ICMP 端口不可到达消息通知其他应用程序这个事实。当然，客户端也必须连接到服务器来接收 ICMP 消息。如果使用 udpclient 的初始版本代替 udpconn2 重新运行最后的测试，就可以看到当我们输入“tow”时 udpclient 的第二个拷贝仅仅挂起。

小结

本节分析了 UDP 中 connect 的使用。虽然无连接协议中的 connect 的使用看起来似乎没有意义，但是我们可以看到它将使程序的效率更高。而且如果我们希望接收某些有关 UDP 数据报的错误消息就必须使用它。本节我们也学会了如何使用 connect 只接收来自一方的数据报。

技巧 31 记住这个世界并不全是 C 语言

整本书中，我们一直都是使用 C 语言作为例子和解释的，当然 C 语言不是唯一的选择。许多人更喜欢用 C++、Java 甚至 Pascal 编写应用程序。本节我们将探讨使用脚本语言进行

网络编程的思想，并以 Perl 给出一些例子。

我们已经看到几个使用小程序来测试或驱动大程序的例子。例如，在技巧 30 中我们使用 `udpclient`、`udpconn1` 和 `udpconn2` 来测试已连接 UDP 套接字的行为。这三个程序都很简单，而且基本上是一样的。在这种情况下，使用脚本语言是很有用的。以脚本语言编写的程序易于组合和修改，这是因为我们没有必要编译它们。在一个特殊的库上连接它们，或者担心 `makefiles`——我们只需要简单地编写脚本并运行它们。

例如，图 3.52 给出了一个实现 `udpclient` 功能的最小 Perl 脚本。

```
pudclient
```

```

1  #! /usr/bin/perl5
2  use Socket;
3  $host = shift || 'localhost';
4  $port = shift || 'echo';
5  $port = getservbyname( $port, 'udp' ) if $port =~ /\D/;
6  $peer = sockaddr_in( $port, inet_aton( $host ) );
7  socket( S, PF_INET, SOCK_DGRAM, 0 ) || die "socket failed $!";
8  while ( $line = <STDIN> )
9  {
10    defined( send( S, $line, 0, $peer ) ) || die "send failed $!";
11    defined( recv( S, $line, 120, 0 ) ) || die "recv failed $!";
12    print $line;
13 }
```

```
pudclient
```

图 3.52 `udpclient` 的 Perl 版本

尽管我们的目的不是给出一个 Perl 教程，但比较详细地分析该脚本程序是值得的。

标准的 Perl 参考书是[Wall et al. 1996]。第六章讨论了 Perl5 的网络和 IPC 功能。有关 Perl 的更详细的信息，包括哪里可以得到它们，可以在<http://www.perl.com> 上找到。

初始化

2 该行让 Perl 定义可以让脚本获得的某些常量（如 `PF_INET`）。

获得命令行参数

3~4 程序从命令行读取主机和端口号。注意该脚本实际上比对应的 C 版本具有更大的功能，这是因为如果忽略了一个参数或两个都忽略的话，它默认的主机参数是“`localhost`”，端口为“`echo`”。

填充 sockaddr_in 结构并分配套接字

5~6 该代码执行技巧 4 中图 2.16 的 set_address 函数相同的功能。注意该函数十分简单。这两行接收数字 IP 地址或主机名称，以及数字或字符的端口名称。

7 程序分配一个 UDP 套接字。

客户端循环

8~13 跟 udpclient 一样，程序从标准输入中循环读取数据行，把它们写给对等方，读取对等方的响应，并把响应输出到标准输出。

虽然相同的网络函数有时具有稍微不同的参数，而且有可能以一种不可预料的方式返回结果，但是图 3.52 的总体流程和感觉还是很熟悉和顺眼的。任何具备网络编程基本知识人只要获得一点 Perl 知识就可以在短期内变得十分内行。

为了比较，图 3.53 给出了一个 TCP echo 服务器。我们可以使用 telnet 或任何其他能够充当 echo 客户端的 TCP 应用程序连接该服务器。

我们再次看到熟悉的套接字 API 的调用序列，即使是根本不知道 Perl，我们也可以跟随程序的流程。我们特别指出两个 Perl 语言的两个特别之处：

(1) 第 11 行的 accept，如果该调用成功了就返回 TRUE，并在它的第二个参数 (\$1) 中返回新套接字。这使 for 循环语句处理第 11 行连续的连接。

(2) 因为 recv 返回发送者（或“未定义”）的地址，而不是读取字节的数量，所以我们必须通过显式地获取 \$line（第 16 行）的长度来检查 EOF 字符。last 操作符实现同 C 语言中 break一样的功能。

pechos

```

1  #! /usr/bin/perl5
2  use Socket;
3  $port = shift;
4  $port = getservbyname( $port, 'tcp' ) if $port =~ /\D/;
5  die "Invalid port" unless $port;
6  socket( S, PF_INET, SOCK_STREAM, 0 ) || die "socket: $!";
7  setsockopt( S, SOL_SOCKET, SO_REUSEADDR, pack( 'l', 1 ) ) ||
8  die "setsockopt: $!";
9  bind( S, sockaddr_in( $port, INADDR_ANY ) ) || die "bind: $!";
10 listen( S, SOMAXCONN );
11 for( ; accept( $1, S ); close( $1 ) )
12 {
13     while ( TRUE )
14     {
15         defined( recv( $1, $line, 120, 0 ) ) || die "recv: $!";

```

```

16     last if length( $line ) == 0;
17     defined( send( $1, $line, 0 ) ) || die "send: $!";
18 }
19 }

```

pechos

图 3.53 echo 服务器的 Perl 版本

我们可以从这两个程序中看到，Perl 和其他脚本语言对于创建测试程序、大规模系统的原型以及小实用程序来说是十分优秀的。实际上，Perl 和其他脚本语言在开发 Web 服务器和专用 Web 客户端中使用十分广泛（例如，请参阅[Castro 1998]和[Patchet and Wright 1998]）。

除了简单和可以迅速创建原型外，还有两个使用脚本语言的原因。其中一个原因是利用脚本语言的特殊性能。例如，Perl 在数据操作和规则表达方面做得很出色，在这些方面使用 Perl 比通常的语言比如 C 更方便。

作为一个具体的例子，让我们假定每天早晨我们都想检查 comp.protocols.tcp-ip 新闻组有关 TCP 或 UDP 的消息。图 3.54 是 Perl 脚本自动执行该任务的框架。正如它表现出来的那样，该脚本用处不大，这是因为它显示新闻服务器上的每条消息，甚至不是新消息也显示出来，而且消息的选择是十分原始的。我们可以很容易地修改该脚本更好地列出文章，但是为了简单起见，我们还是让它保留为一个框架，而不是陷入某些 Perl 细节的困境之中。网络新闻传输协议（NNTP）的细节在 RFC 977[Kantor and Lapsley 1986]中有详细的讨论。

tcpnews

```

1  #! /usr/bin/perl5
2  use Socket;
3  $host = inet_aton( 'nntp.ix.netcom.com' ) || die "host: $!";
4  $port = getservbyname( 'nntp', 'tcp' ) || die "bad port";
5  socket( S, PF_INET, SOCK_STREAM, 0 ) || die "socket: $!";
6  connect( S, sockaddr_in( $port, $host ) ) || die "connect: $!";
7  select( S );
8  $| = 1;
9  select( STDOUT );
10 print S "group comp.protocols.tcp-ip\r\n";
11 while ( $line = <S> )
12 {
13   last if $line =~ /^211/;
14 }
15 ($rc, $total, $start, $end) = split( /\s/, $line );

```

```

16 print S "xover $start-$end\nquit\r\n";
17 while ( $line = <S> )
18 {
19   ( $no, $sub, $auth, $date ) = split( /\t/, $line );
20   print "$no, $sub, $date\n" if $sub =~ /TCP|UDP/;
21 }
22 close( S );

```

tcpnews

图 3.54 收集新闻文章的 Perl 脚本概览

✓ 初始化并连接到新闻服务器

2~6 这几行是标准 TCP 连接逻辑的 Perl 版本。

✓ 设置非缓冲的 I/O

7~9 在 Perl 中，函数 `print` 调用 `stdio` 库，回忆一下技巧 17，它将导致输出到将要缓冲的套接字。这三行关闭缓冲。虽然 `select` 操作符和我们前面讨论的 `select` 系统调用看起来有些相似，但是它仅仅用于选择默认的文件描述符。一旦选择成功，我们就可以设置输出到套接字 `S` 上，通过设置特殊 Perl 变量 `$|` 为非零可以使套接字为非缓冲。

这并不正确。实际发生的是每次调用 `write` 或 `print` 后都会调用 `fflush`。该函数的效果是该套接字的输出是非缓冲的。

第 9 行恢复标准输出为默认的文件描述符。

✓ 选择 comp.protocols.tcp-ip 新闻组

10~14 程序发送给新闻服务器一个组命令，设置当前组为 `comp.protocols.tcp-ip` 新闻组。服务器以下列格式响应：

```
211 total_articles first_article# last_article# group_name
```

通过查找来自服务器中响应代码为 211 的行，程序在第 13 行查询该响应。注意行输入操作符 `<...>` 负责为我们分割 TCP 输入为行。

15~16 一旦程序找到组命令的响应，程序发送下面几行：

```
xover first_article# - last_article#
quit
```

给服务器。`xover` 命令告诉新闻服务器发送给定区域中每篇文章的“概览”。概览是 tab 分

隔的行，包括文章号、主题、作者、日期和时间、消息 ID、该文章引用的文章的消息 ID、字节数以及行数。quit 命令告诉服务器我们没有请求了，并断开连接。

获取文章概览

程序读取每个概览，把它们分隔到我们感兴趣的域中，根据字符串“TCP”或“UDP”是否出现在主题中过滤输出。

当我们运行 `tcpnews` 时，获得以下结果：

```
bsd: $ tcpnews
74179, Re: UDP multicast, Thu, 22 Jul 1999 21:06:47 GMT
74181, Re: UDP multicast, 22 Jul 1999 16:10:45 -0500
74187, Re: UDP multicast, Thu, 22 Jul 1999 23:23:00 +0200
74202, Re: NT 4.0 Server and TCP/IP, Fri, 23 Jul 1999 11:56:07 GMT
74227, New Seiko TCP/IP Chip, Thu, 22 Jul 1999 08:39:09 -0500
74267, WATTCP problems, Mon, 26 Jul 1999 13:18:14 -0500
74277, Re: New Seiko TCP/IP Chip, 26 Jul 1999 23:33:42 GMT
74305, TCP Petri Net model, Wed, 28 Jul 1999 02:27:20 +0200
bsd: $
```

本节我们集中精力在 Perl 语言上，但是还有其他优秀的脚本语言可以用于网络编程。包括：

- TCL/Expect
- Python
- JavaScript
- VB (Windows)

所有这些语言在自动化简单网络任务、构造原型以及创建快速而简单实用程序或测试例子方面都做得很出色。正如我们已经看到的那样，脚本语言通常比常规编程语言更容易使用，因为它们为我们处理了很多单调的网络杂务（当然，这是以性能为代价的）。精通至少一门这些语言在开发效率上会获得巨大的收获。

小结

本节探讨了网络编程中脚本语言的使用。从中知道使用这些语言编写小的实用程序和测试程序是很有用的。

技巧 32 理解缓冲区大小的影响

本节给出设置 TCP 发送和接收缓冲区大小的一些经验。我们已经在技巧 7 中了解如何用 `setsockopt` 设置缓冲区大小。现在我们应当设置它们为多大。

首先发现的事情是适当的缓冲区大小依赖于应用程序。对于一个交互式的应用程序如 `telnet` 来说，小的缓冲区就足够了。这么做有两个原因：

- (1) 通常客户端发送少量的数据给对等方然后等待响应。所以，分配很大的缓冲区空

间给这种类型的连接是对系统资源的浪费。

(2) 如果分配了大的缓冲区，需要立即响应的用户输入就可能停留在大量的数据后面。例如，如果用户列出一个大文件并按下中断键（也就是`<CTRL-C>`）终止列出操作，缓冲区中数据仍然输出到屏幕上。如果缓冲区太大，将导致列出中断之前大量的延时。

通常当我们关心缓冲区大小时，希望获得最大的数据吞吐量。也就是说，应用程序涉及从一个主机到另一个主机大量的数据传输，而且几乎所有的数据流量都是一个方向的。在本节的其余部分，我们将讨论这种类型的大量数据传输应用程序。

一般的建议是，为了获得最大的吞吐量，发送和接收缓冲区应当至少为带宽延迟乘积。我们将看到，这是一个正确但是不是特别有用的原则。在讨论原因之前，首先看看带宽延迟乘积以及为什么它是“正确”的大小。

我们已经好几次遇到往返时间 RTT。RTT 是数据包从一个主机到另一个主机然后返回所花费的时间。它占用带宽延迟乘积的“延迟”部分，这是因为 RTT 是衡量数据包发送和它的 ACK 被发送者接收之间的延迟时间。

带宽延迟乘积的另外部分是带宽。这是在某些物理介质上每单位时间可以传输的数据量。

当然，从技术上讲这并不正确，但是该术语已经使用了很长一段时间，可以理解为它的技术意思的外延。

带宽通常以位每秒计算。例如，Ethernet 具有每秒 10M 的带宽。

带宽延迟乘积，BWD，由

$$\text{BWD} = \text{bandwidth} \times \text{RTT}$$

给出。如果以秒计算 RTT，BWD 的单位为

$$\begin{aligned}\text{BWD} &= (\text{bits/second}) \times \text{seconds} \\ &= \text{bits}\end{aligned}$$

如果我们把两台主机之间的通信频道看作“管道”（实际上，我们经常这么称呼它们），那么带宽延迟乘积是管道的以位计的容量（图 3.55）。它是在给定时间内可以在网络上传输的数据量。



图 3.55 流量为 BWD 位的管道

现在让我们假定该管道在大量数据传输使用所有可获得的网络带宽期间的稳定状态下是怎么样的（也就是说，在缓慢启动之后）。图 3.56 中的发送方用 TCP 段将管道塞满了，

在发送另一个之前必须等待第 n 个段离开网络。因为管道中具有和段一样多的 ACK 消息，所以当它接收第 $n-8$ 个段的 ACK 消息时，发送方可以推断第 n 个段已经离开网络。

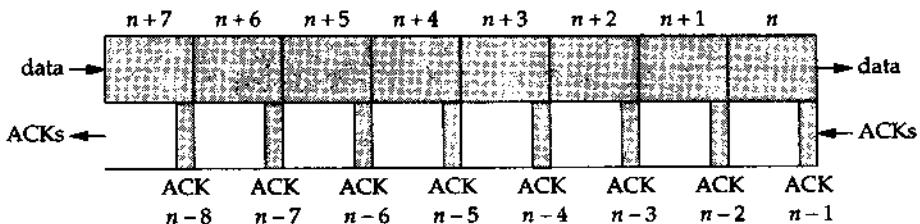


图 3.56 处于稳定状态的网络

这说明了处于稳定状态的 TCP 连接的自计时特性(self-clocking property) [Jacobson 1988] 返回给发送方的 ACK 作为 TCP 发送另一个段的信号时钟。

这种机制通常称作 ACK 时钟。

注意如果该自计时正在工作并保持管道满时，发送窗口必须足够大允许 16 个 ($n-8 \dots n+7$) 没有确认的段。反过来这意味着发送方缓冲区和对等方的接收缓冲区必须大到足够保存 16 个段。通常情况下，这些缓冲区必须保存“满管道的”数据。也就是说，它们必须至少和带宽延时乘积一样大。

前面我们已经提到的这个规则并不十分有用。原因是通常情况下很难知道带宽延时乘积为多少。例如，假定我们要编写一个 FTP 类型的应用程序。我们应当设置发送和接收缓冲区为多大？首先，我们在编写应用程序的时候不知道将要使用网络的类型（所以也不知道它的带宽）。即使在运行时查询接口来决定这个值，我们也还要确定延迟值。原则上讲，应当使用某种类似 ping 的机制来预测延迟值，但是即使在不可靠的事件中我们获得了好的预测值，延迟值也很可能在连接的生存周期中变化。

[Semke et al. 1998] 中给出了这个问题的一个解决方法，就是在连接的生存周期内动态地改变缓冲区的大小。作者发现观察拥塞窗口的一个方法是把它作为带宽延时乘积的估计。通过匹配缓冲区大小为拥塞窗口的大小（带有某些误差和公平性的限制），使它们能够为变化带宽延时乘积的并发连接获得不错的吞吐量。不幸的是，该解决方法需要改变内核，因此对应用程序来说是不可获得的。

通常的解决办法是使用默认的缓冲区大小，或者设置缓冲区为一个很大的值。这两个选择都不够吸引人。第一个解决办法可能导致吞吐量的减少，而第二个解决办法在 [Semke et al. 1998] 中讨论过了，可能导致缓冲区耗竭和操作系统失败。

如果对应用程序将要运行的特定环境缺乏的认识，那么最好的操作解决办法可能是为交互性比较强的应用程序使用小的缓冲区而对大量数据传输的应用程序设置 32K 到 64K 之间的缓冲区。然而，应当认识到高速网络需要很大的缓冲区来利用带宽。[Mahdavi 1997] 为调制 TCP 堆栈适应高性能传输给出了一些建议。

有一个更容易应用的规则，它可以避免许多具体实现中的低性能。[Comer and Lin 1995] 描述了由传统的 10Mb/s Ethernet LAN 和 100-Mb/s ATM LAN 连接的两台主机之间的实验。在默认的缓冲区大小 16K 下，Ethernet LAN 中达到 1.313Mb/s 吞吐量的相同的 FTP 会在更快的 ATM LAN 中降为 0.322Mb/s。

在进一步的研究中，作者发现缓冲区大小、ATM 接口的 MTU、TCP MSS 和套接字层传输数据给 TCP 的方法对设置 Nagle/延迟 ACK 交互（请参阅技巧 24）都有影响。

MTU，是 maximum transmission unit 的缩写，是网络可以携带的最大数据帧。例如 Ethernet 的 MTU 是 1500 字节。对于[Comer and Lin 1995]中描述的 ATM 网络来说，它是 9188 字节。

虽然这些结果是在特定 TCP 具体实现（SunOS 4.1.1）的 ATM LAN 中获得的，它们对很多网络和具体实现都适用。关键的参数是网络 MTU 和套接字层传递数据给 TCP 的方法，这是由大多数 BSD 派生的具体实现所共享的。

该文作者发现了该问题的一个很好的解决方法。说它很好是因为它仅涉及到调整发送缓冲区的大小——接收缓冲区的大小并非问题的本质。如果发送缓冲区至少 3 倍于 MSS，[Comer and Lin 1995]中描述的互操作就可能发生。

它的工作原理是强迫接收者发送一个更新窗口，然后发送一个 ACK 消息，这样就避免了延迟 ACK 以及它对发送者 Nagle 算法的相互影响。究竟要不要发送更新窗口依赖于接收者缓冲区的大小是否小于或至少三倍于 MSS，但是在两种情况下都会发送更新窗口。

因此，非交互的应用程序应当总是设置它们的发送缓冲区至少为 $3 \times \text{MSS}$ 。回忆一下技巧 7，这必须在调用 listen 或 connect 之前执行。

小结

TCP 的性能和它的发送和接收缓冲区的大小有很大的关系（请参阅技巧 36，了解关于此的令人吃惊的例子）。本节我们知道高效的大量数据传输的应用程序的最优缓冲区大小由 bandwidth-delay product 决定，但是这个发现在实际的应用程序中受到了限制。

虽然应用 bandwidth-delay 规则很困难，有一个规则很容易应用而且我们应当总是注意使用它，我们应当让发送缓冲区总是至少 3 倍于 MSS 的大小。

第四章

工具和资源

技巧 33 熟悉 ping 实用程序

调试网络和运行在网络上的应用程序的最基本的和有效的工具之一就是 ping 实用程序。它的主要功能是验证两个主机之间的连接。它也是调试网络问题的一个极具价值的工具。

在我们继续之前，应当澄清两个有关 ping 的误解。首先，按照 ping 程序编写者 Mike Muuss 的说法，“ping”不是 packet internet groper 的缩写，而是以潜艇声纳产生的声音命名的。ping 程序的历史以及它的开发过程在 Muuss 的“*The Story of the Ping Program*”中有描述，该文章在<<http://ftp.arl.mil/~mike/ping.html>>上有链接，上面还有源代码的拷贝。

其次，因为 ping 不使用 TCP 或 UDP，所以它没有和任何已知端口关联。ping 使用 ICMP echo 函数来探询对等方的连接。回忆一下技巧 14 的内容，虽然 ICMP 消息是在 IP 数据报中携带的，但是它们并没有看作是在 IP 之上的独立协议，而是作为 IP 协议的一部分的。

RFC 792 [Postel 1981]在第一页写道：“ICMP 使用基本的 IP 支持，就像它是 IP 之上的更高层次，然而 ICMP 实际上是 IP 整体中的一部分，而且它必须在每个 IP 模块中实现。”

ping 数据包具有图 4.1 的格式。消息的 ICMP 部分地显示在图 4.2 中，由 8 字节的 ICMP 报头和 n 字节的可选数据部分组成。

包含在 ping 数据包中的附加数据数目 n 的通常值为 56 (UNIX) 和 32 (Windows)，但是该数字可以用-s (UNIX) 或-l (Windows) 选项来改变。

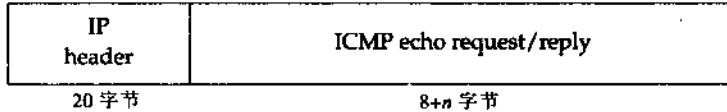


图 4.1 ping 数据包的格式

ping 实用程序的一些版本允许用户指定附加数据的值，甚至可以指定它们应当为伪随机的。默认情况下，大多数版本的 ping 实用程序仅使用轮换数据作为附加的数据。

在解决数据有关的问题时，指定特定数据模式有时是十分有用的。

UNIX 版本的 ping 在附加数据的开始 8 个字节放置一个时间戳（timeval 结构）（当然，假定附加数据至少有 8 个字节）。当 ping 接收到反射回来的响应时，它就使用时间戳计算 RTT。Windows 版本的 ping 似乎没有这么做（从 tcpdump 输出的结果来看），但是 Visual C++ 附带的样本程序 ping 采用了该方法。

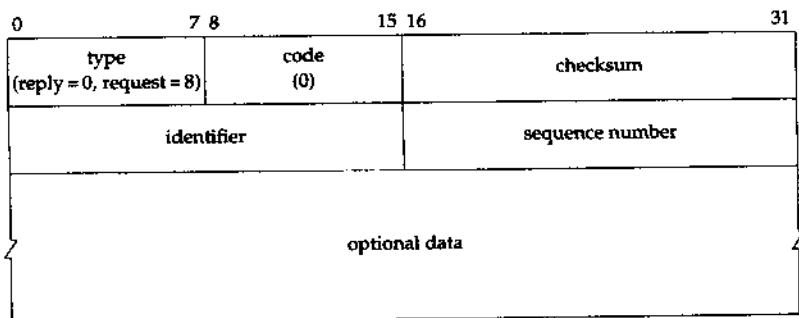


图 4.2 ICMP 反射回来的请求/响应数据包

因为 identifier 和序列号（sequence number）域没有被 ICMP 反射回来的请求/应答消息使用，所以 ping 使用它们 ICMP 响应已经返回的标识。因为没有任何端口关联到 ICMP 数据报，所以这些数据报就会被传递到所有具有指定 ICMP 协议的开放套接字的进程上（技巧 40）。因此，ping 在 identifier 域放置它的进程 ID，以便每个 ping 实例可以区分数据报返回时是否是它自己的响应。我们可以把 identifier 域的这种用法作为端口号的替代者。

同样地，ping 在序列号域中放一个不断增加的计数器来将每个反射的应答结合到一个反射的请求上。该数字是 ping 输出的 icmp_seq。

通常，当我们不能连接对等方时，要做的第一件事情就是 ping 向要连接的主机。例如，假定我们正在尝试 telnet 到主机 A，但是连接超时了。这有可能是因为我们和主机 A 之间的网络存在问题，有可能是因为主机 A 关闭了，也有可能是因为 telnet 服务器的 TCP 栈存在问题。

通过 ping 对等方，我们首先可以确认是否可以到达主机 A。如果 ping 成功了，我们知道网络没有问题，而是主机 A 本身的问题。如果不能 ping 通主机 A，可以试着 ping 最近的路由器看看是否可以到达局域段的边界，如果成功了，我们就可以启动 traceroute（技巧 35）看看究竟可以到达从我们的主机到主机 A 之间的路径的什么地方。使用这种方法通常可以帮助找出有问题的路由器，或者至少可以知道问题出在什么地方。

因为 ping 在 IP 层上进行操作，所以它不依靠 TCP 或 UDP 是否正常配置，所以有时 ping 自己对验证网络软件是否正确安装很有用。在最低的层次上，我们可以 ping 我们的 loopback 地址 localhost (127.0.0.1)，验证机器是否具备网络功能。如果成功，我们可以 ping 一个

或多个网络接口来确认它们是否配置正确。

下面尝试 ping 主机 netcom4.netcom.com，该主机离我们有 10 个节点远，结果如图 4.3 所示：

```
bsd: $ ping netcom4.netcom.com
PING netcom4.netcom.com (199.183.9.104): 56 data bytes
64 bytes from 199.183.9.104: icmp_seq=0 ttl=245 time=598.554 ms
64 bytes from 199.183.9.104: icmp_seq=1 ttl=245 time=550.081 ms
64 bytes from 199.183.9.104: icmp_seq=2 ttl=245 time=590.079 ms
64 bytes from 199.183.9.104: icmp_seq=3 ttl=245 time=530.114 ms
64 bytes from 199.183.9.104: icmp_seq=4 ttl=245 time=480.137 ms
64 bytes from 199.183.9.104: icmp_seq=5 ttl=245 time=540.081 ms
64 bytes from 199.183.9.104: icmp_seq=6 ttl=245 time=580.084 ms
64 bytes from 199.183.9.104: icmp_seq=7 ttl=245 time=580.078 ms
64 bytes from 199.183.9.104: icmp_seq=8 ttl=245 time=490.078 ms
64 bytes from 199.183.9.104: icmp_seq=9 ttl=245 time=560.090 ms
64 bytes from 199.183.9.104: icmp_seq=10 ttl=245 time=490.090 ms
64 bytes from 199.183.9.104: icmp_seq=11 ttl=245 time=490.090 ms
^C
ping terminated manually
--- netcom4.netcom.com ping statistics ---
12 packets transmitted, 10 packets received, 16% packet loss
round-trip min/avg/max/stddev = 480.137/540.939/598.554/40.871 ms
bsd: $
```

图 4.3 运行时间短的 ping 程序

这里需要注意几个问题。首先，ping 的 RTT 一般为 500ms。实际上，从最后一行可以看出 RTT 在 480.137ms 到 598.554 之间变化，标准的偏差是 40.871ms。该程序运行时间很短，不能得出任何结论，但是大约两分钟的长时间地运行以得到类似的结果，因此我们可以得出网络负载很稳定的结论。RTT 的波动通常是网络负载变化无常的信号。在网络具有比较大的负载的时候，路由器的排队时间变长，因此 RTT 值变大。当网络负载较小时，队列长度变短，RTT 值也随之变短。

图 4.3 中需要注意的另一件事情就是序列号为 4 的 ICMP 反射回来的请求没有返回。这说明要么是请求要么是应答被中间的路由器之一丢弃了。在总结行里可以看出程序发送了 12 个请求（0~11），但是只接收到 10 个。丢失的响应的序列号为 4，另一个序列号为 11，这有可能是因为在它返回之前我们就终止了 ping 的运行。

小结

ping 实用程序是最基本的网络连接测试程序之一。因为它只需要操作低层次的网络服务，所以在验证连接是否正常时很有用。它也可以验证高层次的服务如 TCP 或应用程序层服务如 telnet 是否正常工作。

使用 ping，通过观察应答的 RTT 变化值以及丢失的响应就可以推断网络环境。

技巧 34 学会使用 tcpdump

或一个类似的工具

在处理或调试网络应用程序或解决网络问题时最有用的工具之一是 sniffer 或网络分析器。

通常，sniffer 是昂贵的硬件设备，但是当代工作站可以以一个进程的方式执行它们的功能。

今天，大多数支持网络的操作系统上都有 sniffer 程序。有时操作系统提供了一个私有的 sniffer（例如，Solaris 的 snoop 或 AIX 的 iptrace/ireport 程序），有时可以使用第三方的应用程序如 tcpdump。

而且 sniffer 已经从严格的诊断工具发展为更通用的角色如研究和教育工具。例如，研究者通常使用 sniffer 来研究网络动态和交互作用。在[Stevens 1994, Stevens 1996]中，Stevens 给出了如何使用 tcpdump 来研究和理解 TCP/IP 协议。通过观察协议发送的实际数据，可以获得对协议在实际中是如何工作的更深的理解，也可以发现特定的具体实现是否按照规范来实现。

本节我们集中在 tcpdump 上。正如前面已经提到的那样，还有其他基于软件的网络 sniffer，其中一些可以提供更好的输出结果，但是 tcpdump 有可以在任何 UNIX 系统和 Windows 机器上运行的优势。因为 tcpdump 可以获得源代码，所以它可以改编成特殊要求的应用程序或在需要时移植到新的环境中。

tcpdump 的源代码可以在：

<http://www-nrg.ee.lbl.gov/nrg.html> 上获得。

Windows 版本的源代码和可执行程序 Windump 可以在：

<http://netgroup-serv.polito.it/windump> 上获得。

» tcpdump 是如何工作的

在我们继续其他问题之前，先看看 tcpdump 是如何工作的，以及它是如何在协议栈中拦截数据包的。因为它是经常导致困惑的原因，所以花点时间来理解它是很值得的。和大多数的 sniffer 一样，tcpdump 有两个组成部分：从网络中捕获以及过滤数据包的核心部分以及处理用户界面、格式化和过滤（如果核心部分没有做的话）的用户空间部分。

tcpdump 的用户空间部分使用 libpcap（捕获数据包的库）和核心部分进行通信。该库在它自己的一方很有用，它抽取和链路层通信的系统有关的细节。例如，在基于 BSD 的系统上，libpcap 和 BSD packet filter (BPF) 进行会话[McCanne and Jacobson 1993]。BPF 检查通过链路层的每个数据包并拿它和用户指定的过滤器进行匹配。如果过滤器选择了数据包，就放置数据包的一份拷贝到内核缓冲区中，供关联到该过滤器的应用程序使用。当缓冲区填满了，或者当用户指定的计时器到时了，缓冲区的内容就通过 libpcap 传递到应用程序。

图 4.4 说明了这个过程，该图给出了 tcpdump 和另一个（未指定的）从 BPF 读取原始网络数据包应用程序以及第三个照常从 TCP/IP 堆栈读取数据的一般 TCP/IP 应用程序。

虽然我们给出了 tcpdump 和另一个使用 libpcap 的程序，但是直接和 BPF 或任何其他接口进行会话还是有可能的，我们将在后面讨论它。libpcap 的优势是它为我们访问原始数据包提供了一种系统无关的方法。目前的库支持 BPF、数据链路提供商接口 (DLPI)、SunOS NIT、基于流的 NIT、Linux SOCK_PACKET 套接字、snoop 接口 (IRIX) 和 Stanford enet 接口。Windows 版本的 libpcap 可以和 WinDump 的发布一起获得。

注意 BPF 在设备驱动程序层连接网络数据包——也就是说，它们从线缆上一下来就被拦截了。这和从原始套接字读取数据不同。在读取原始套接字的情况下，IP 数据报由 IP 层处理，然后直接传递给应用程序，不首先通过传输层（TCP/IP）。

版本 2.0 之后，WinDump 体系结构和 BSD 系统使用的体系结构十分相似。WinDump 使用一个特殊的网络设备接口规范（NDIS）驱动程序，提供了一个 BPF 兼容的过滤器和接口。在 WinDump 体系结构中，NDIS 驱动程序实际上是协议栈的一部分，但是从功能上看，它和图 4.4 是一样的，只是用“NDIS Packet Driver”代替了“BPF”。

其他的操作系统稍微有些不同。SVR4 派生的系统使用 DLPI[UNIX International 1991] 提供对原始数据包的访问。DLPI 是链路层的协议无关的基于流的[Ritchie 1984]的接口。我们可以通过 DLPI 直接访问链路层，但是考虑到效率我们通常把 pfmod 和 bufmod 的 STREAMS 模块推入到流中。模块 bufmod 提供了消息缓冲并通过限制传递数据时所需的上下文切换来提高性能。

这和从套接字中读取满缓冲的数据而不是每次一个字节相类似。

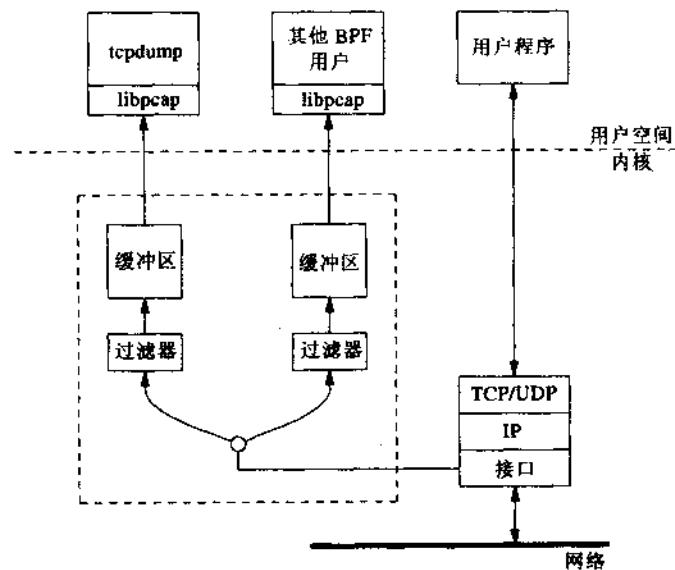


图 4.4 通过 BPF 捕获数据包

模块 pfmod 是一个过滤器，它和 BPF 提供的很相似。因为该过滤器和 BPF 过滤器不兼容，tcpdump 没有把这个模块引入到流中，而是在用户空间部分执行过滤操作。它比 BPF 体系结构的效率低，这是因为不管 tcpdump 是否对它很感兴趣，所有数据包都会传递给用户空间部分。

图 4.5 说明了这个原理，给出了没有 pfmod 和另一个也接收原始数据包的应用程序的 tcpdump，但是它使用了内核过滤器。

我们再次给出使用 libpcap 的应用程序，但是对于 BPF 来说这并不是必须的。我们可

以使用 `getmsg` 和 `putmsg` 函数直接从流中发送或接收数据。[\[Rago 1993\]](#)是一本有关 STREAMS 编程、DLPI 和 `getmsg` 和 `putmsg` 系统调用的十分优秀的书。在[\[Stevens 1998\]](#)的第 33 章中可以找到有关此的简要讨论。

最后，还有 Linux 体系结构。在 Linux 中，对原始网络数据包的访问是通过 `SOCK_PACKET` 套接字接口进行的。我们只需要打开一个 `SOCK_PACKET` 套接字、绑定所需的接口、启用乱模式，然后从里面读取数据，就可以使用这个简单而且优雅的接口。

从 Linux 版本 2.2 的内核开始，就可以获得一个稍微有些不同的接口，该接口也是被推荐使用的。

但是 `libpcap` 的当前版本仍然使用这里描述的接口。

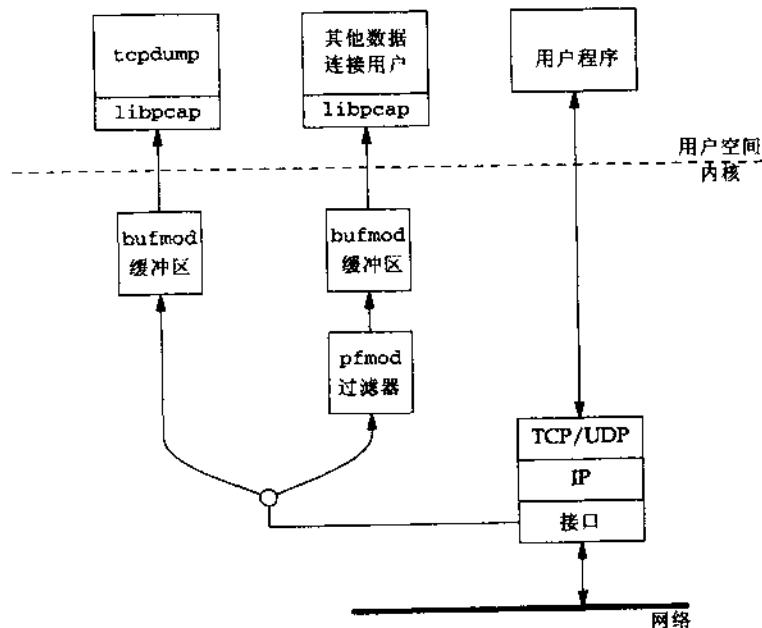


图 4.5 使用 DLPI 截获数据包

例如，

```
s = socket ( AF_INET, SOCK_PACKET, htons (ETH_P_ALL) );
```

打开一个可以访问所有以太数据包的套接字。我们也可以指定第三个参数为 `ETH_P_IP` (IP 数据包)、`ETH_P_IPV6` (IPv6 数据包) 或 `ETH_P_ARP` (ARP 数据包)。除了该接口提供的是链路层的访问而不是网络 (IP) 层之外，我们可以把它看作和原始套接字 (`SOCK_RAW`) 相似的接口。

不幸的是，虽然该接口简单而且方便，但是它的效率不高。因为不存在有别于普通套接字缓冲区的内核缓冲机制，所以只要可以获得一个数据包，它就会传递给应用程序。因为它也没有内核过滤机制（除了 `ETH_P_*` 参数之外），所以必须在应用程序级别实现过滤机制，这意味着每一个数据包都必须传递给应用程序。

使用 tcpdump

使用 `tcpdump` 的第一个步骤是获得授权。因为 `sniffer` 会带来显而易见的安全危险，`tcpdump` 的默认配置需要 `root` 权限。

这不适用于 Windows 环境。一旦安装了 NDIS 数据包捕获驱动器，任何人都可以使用 WinDump。

在很多情况下，不需要给它们 `root` 权限就可以让所有用户使用 `tcpdump`，这是十分方便的。达到这个目的的方法根据 UNIX 的版本变化而变化，它在每个系统的 `tcpdump` 手册中给出。在大多数情况下，它涉及到使网络接口对任何人可读或者以 `setuid` 应用程序安装 `tcpdump` 程序。

`tcpdump` 的最简单的调用是不使用任何命令行参数。这将导致 `tcpdump` 捕获并列出网络上所有的数据包。然而，指定一个过滤器是很有用的。这是因为我们可以只获得我们感兴趣的数据包而不被其他的数据包干扰。例如，如果我们希望只查看来自主机 `bsd` 的数据包，就应当这样调用 `tcpdump`：

```
tcpdump host bsd
```

如果我们只对主机 `bsd` 和 `sparc` 之间的数据包感兴趣，那么可以使用如下的过滤器：

```
host bsd and host sparc
```

或者它的缩写：

```
host bsd and sparc
```

过滤器语言十分丰富，可以用于过滤下面各类属性：

- 协议
- 源和/或目的主机
- 源和/或目的网络
- 源和/或目的以太地址
- 源和/或目的端口
- 数据包的大小
- 数据包是否为广播或多路传播（IP 或以太地址）
- 数据包是否使用特定的主机作为网关

另外，还可以测试协议报头中的特定位或字节。例如，为了只截获设置了具有紧急位的 TCP 段，可以使用如下的过滤器：

```
tcp[ 13 ] & 16
```

为了很好地理解最后的例子，必须知道 TCP 报头的第 14 个字节的第 4 位是紧急位。

我们可以使用布尔操作符 `and`（或 ‘`&&`’）、`or`（或 ‘`||`’）和 `not`（或 ‘`!`’）来连

接前面给出的简单谓词，这样可以指定任意复杂的过滤器。作为最后一个例子，下面的过滤器捕获来自外部网络的 ICMP 数据包：

```
icmp and not src net localnet
```

请参阅 *tcpdump* 的手册页，了解复杂过滤器的其他例子。

» tcpdump 输出

tcpdump 的输出是协议相关的。看一些例子来获得最普通协议的输出的感觉。*tcpdump* 文档详细地涉及了不同的输出。

第一个例子是简单邮件传输协议 (SMTP) 会话的网络跟踪——这是发送电子邮件的协议。除了我们在每行的开始增加了以斜体字输出的行号、已经从主机 *bsd* 中删除了域名以及为了符合页面的要求将长行折行外，图 4.6 中的输出看起来似乎和 *tcpdump* 产生的一样。

为了产生这个跟踪，我们发送电子邮件给 *gte.net* 上的用户。也就是说电子邮件的地址的格式为 *user@gte.net*。

```
1 12:54:32.920881 bsd.1067 > ns1.ix.netcom.com.domain:
 45801+ MX? gte.net. (25)
2 12:54:33.254981 ns1.ix.netcom.com.domain > bsd.1067:
 45801 5/4/9 (371) (DF)
3 12:54:33.256127 bsd.1068 > ns1.ix.netcom.com.domain:
 45802+ A? mta.pop2.gte.net. (33)
4 12:54:33.534962 ns1.ix.netcom.com.domain > bsd.1068:
 45802 1/4/4 (202) (DF)
5 12:54:33.535737 bsd.1059 > mta.pop2.gte.net.smtp:
  S 585494507:585494507(0) win 16384
    <mss 1460,nop,wscale 0,nop,nop,
    timestamp 6112 0> (DF)
6 12:54:33.784963 mta.pop2.gte.net.smtp > bsd.1059:
  S 1257159392:1257159392(0) ack 585494508 win 49152
    <mss 1460,nop,wscale 0,nop,nop,
    timestamp 7853753 6112> (DF)
7 12:54:33.785012 bsd.1059 > mta.pop2.gte.net.smtp:
  . ack 1 win 17376 <nop,nop,
  timestamp 6112 7853753> (DF)
8 12:54:34.235066 mta.pop2.gte.net.smtp > bsd.1059:
  P 1:109(108) ack 1 win 49152
    <nop,nop,timestamp 7853754 6112> (DF)
9 12:54:34.235277 bsd.1059 > mta.pop2.gte.net.smtp:
  P 1:19(18) ack 109 win 17376
    <nop,nop,timestamp 6113 7853754> (DF)

  14 lines deleted

24 12:54:36.675105 bsd.1059 > mta.pop2.gte.net.smtp:
  F 663:663(0) ack 486 win 17376
    <nop,nop,timestamp 6118 7853758> (DF)
25 12:54:36.685080 mta.pop2.gte.net.smtp > bsd.1059:
  F 486:486(0) ack 663 win 49152
    <nop,nop,timestamp 7853758 6117> (DF)
26 12:54:36.685126 bsd.1059 > mta.pop2.gte.net.smtp:
  . ack 487 win 17376
    <nop,nop,timestamp 6118 7853758> (DF)
27 12:54:36.934985 mta.pop2.gte.net.smtp > bsd.1059:
  F 486:486(0) ack 664 win 49152
    <nop,nop,timestamp 7853759 6118> (DF)
28 12:54:36.935020 bsd.1059 > mta.pop2.gte.net.smtp:
  . ack 487 win 17376
    <nop,nop,timestamp 6118 7853759> (DF)
```

图 4-6 SMTP 跟踪显示 DNS 与 TCP 输出

第 1~4 行与找出 SMTP 服务器 gte.net 的地址有关，它们是 tcpdump 为 DNS 查询和响应产生的输出的例子。第 1 行，bsd 查询它的 ISP 的名称服务器（ns1.ix.netcom.com）获取 gte.net 上的邮件服务器的地址。第一个域给出数据包的时间戳（12:54:32.920881）。因为 bsd 提供了以微秒为单位的计时器，所以显示了小数点后 6 位数字。接下来，可以看到该数据包是从 bsd 的 1067 端口到主机 ns1 的 53 端口（域）的。最后，可以看到数据包中数据信息。第一个域（45801）是 bsd 上的解析函数用来匹配查询响应的。‘+’说明如果没有结果的话，解析者就希望 DNS 服务器查询其他的服务器。‘MX?’ 说明这是为在下一个域（gte.net）中命名的网络获取邮件交换记录的请求。‘(25)’的意思是请求为 25 个字节长。

第 2 行是第 1 行中查询的响应。数字 45801 是所响应查询的号码。接下来的以斜杠分开的域是回答记录的个数、名称服务器（授权）记录的个数和其他记录的个数。‘(371)’告诉我们响应有 371 个字节。最后，‘(DF)’表明该响应在 IP 报头中设置了“Don't Fragment”位。开头的两行说明了技巧 29 中提及的使用 DNS 定位邮件处理程序。

和开始两行找出 gte.net 的邮件处理程序相对应的是，接下来的两行用来查询它的 IP 地址。第 3 行中的 ‘A?’ 说明这是一个对 mtapop2.gte.net（GTE 的一个邮件服务器）的 IP 地址请求。

第 5~28 行给出了实际 SMTP 传输的细节。bsd 和 mtapop2 之间的三阶段握手从第 5 行开始到第 7 行结束。时间戳和主机之后的第一个域是 flags。第 5 行中的 ‘S’ 说明设置了 SYN 标志。其他可能的标志是 ‘F’（FIN 标志）、‘U’（URG 标志）、‘P’（PUSH 标志）、‘R’（rst 标志）和 ‘.’（无标志）。接下来是第一个也是“最后的”序列号，之后在括号中跟着数据字节的个数。这些域可能会导致一点误解，这是因为除了在没有数据时“最后的”序列号通常是第一个没有使用的序列号。理解这些域的最好的方法是段（SYN 或数据）中的第一个序列号是作为第一个数字给出的，第二个数字第一个序列号加上段中数据的字节个数。注意默认的操作是显示 SYN 段中的实际序列号，以及显示（更容易跟踪的）后面的段偏移量。该操作可以用 -S 命令行选项改变。

除了开始的之外，所有 SYN 将包含 ACK 域，指示发送者希望从对等方获得的下一个序列号。默认地，该域之后的域（以 ack nnn 的格式）又是 SYN 段的偏移量。

ACK 域的之后是 window 域。这是对等方应用程序将要接收数据的数量。它通常用作反映连接对等方可获得的缓冲区空间。

最后，任何在段中给出的 TCP 选项在尖括号中都给出（<和>）。主要的 TCP 选项在 RFC 793[Postel 1981b]和 RFC 1323[Jacobson et al. 1992]中有详细的讨论。[Stevens 1994]也有这些选项的讨论，在下面的主页上可以获得完整列表：

```
<http://www.isi.edu/in-notes/iana/assignments/tcp-parameters>
```

第 8~23 行显示了 bsd 上的 sendmail 和 mtapop2 上的 SMTP 服务器之间的会话。中间省略了多行。第 24~28 行是连接撤消。首先 bsd 在第 24 行发送一个 FIN 消息，之后第 25 行

紧跟着 mtapop2 的 FIN。注意 mtapop2 在第 27 行重新发送 FIN 消息，说明它没有接收到来自 bsd 的第一个 FIN 的确认消息。这又说明了技巧 22 中讨论的 TIME-WAIT 状态的重要性。

下面看看 UDP 数据报产生的输出。为了达到这个目的，我们使用 udphelloc（技巧 4）发送一个 NULL 字节到 netcom.com 域中服务器的 daytime 端口：

```
bsd: $ udphelloc netcom4.netcom.com daytime
Thu sep 16 15:11:49 1999
Bsd: $
```

主机 netcom4 在另一个 UDP 数据报中返回日期和时间。tcpdump 的输出为：

```
18:12:23.130009 bsd.1127 > netcom4.netcom.com.daytime: udp 1
18:12:23.389284 netcom4.netcom.com.daytime > bsd.1127: udp 26
```

这两行告诉我们 bsd 发送了一个 1 字节的 UDP 数据报给 netcom4，netcom4 响应一个 26 字节的数据报。

ICMP 数据包的输出相似。下面是一个从 bsd 到 netcom4 之间 ping 的跟踪：

```
1 06:21:28.690390 bsd > netcom4.netcom.com: icmp: echo request
2 06:21:29.400433 netcom4.netcom.com > bsd: icmp: echo reply
```

icmp：告诉我们这是一个 ICMP 数据报，后面的文本告诉我们它是什么类型的 ICMP 数据报。

tcpdump 的一个缺点就是它对数据显示的支持。通常在调试网络应用程序时，知道实际发送了什么数据是很有用的。我们可以使用-s 和-x 命令行选项获取这些数据，但是数据的显示却只有 16 进制。-x 选项告诉 tcpdump 以 16 进制的方式输出数据包的内容。-s 选项指出要截获数据包中的多少数据。默认地，tcpdump 仅截获开始的 68 个（SunOS 的 NIT 为 96 个）字节——对于大多数的协议来说获取报头信息已经是足够了。重复前面 UDP 的例子，但是这次同时也截获数据：

```
tcpdump -x -s 100 1
```

在删除 DNS 流量和来自 bsd 的域之后，获得如下结果：

```
1 12:57:53.299924 bsd.1053 > netcom4.netcom.com.daytime: udp 1
    4500 001d 03d4 0000 4011 17a1 c7b7 c684
    c7b7 0968 041d 000d 0009 9c56 00
2 12:57:53.558921 netcom4.netcom.com.daytime > bsd.1053: udp 26
    4500 0036 f0c8 0000 3611 3493 c7b7 0968
```

```
c7b7 c684 000d 041d 0022 765a 5375 6e20
5365 7020 3139 2030 393a 3537 3a34 3220
3139 3939 0a0d
```

第一个数据包的最后一个字节是 `duphello` 发送给 `netcom4` 的 NULL 字节。第二个数据包的最后 26 个字节是应答。如果我们正在查找特定的数据，该输出翻译起来很费劲。

由于历史原因，`tcpdump` 的作者不愿意提供数据的 ASCII 翻译，因为它使技术上不成熟的不法分子监听密码变得很容易。许多人认为监听密码工具的大量存在造成这种阻力不切实际，种种迹象表明 `tcpdump` 的未来版本将提供该功能。

同时，为 `tcpdump` 编写过滤器来提供 ASCII 输出是网络程序员的一个极其普通的练习，而且在 Internet 上也可以下载这个程序。本着 Internet 精神，这里给出了一个。图 4.7 中的 Perl 脚本称作 `tcpdump`，输送它的输出给脚本，并以 ASCII 的方式格式化数据。

```
tcpd
1 #! /usr/bin/perl5
2 $tcpdump = "/usr/sbin/tcpdump";
3 open( TCPD, "$tcpdump @ARGV |" ) |||
4 die "couldn't start tcpdump: \"$!\\n\"";
5 $| = 1;
6 while ( <TCPD> )
7 {
8   if ( /^t/ )
9   {
10     chop;
11     $str = $_;
12     $str =~ tr / \t//d;
13     $str = pack "H*", $str;
14     $str =~ tr/\x0-\x1f\x7f-\xff//;
15     printf "\t%-40s\t%s\n", substr( $_, 4 ), $str;
16   }
17 else
18 {
19   print;
20 }
21 }
```

图 4.7 过滤 `tcpdump` 输出的 Perl 脚本

如果我们使用 `tcpd` 而不是 `tcpdump` 重复最后的例子，将获得以下结果：

```

1 12:58:56.428052 bsd.1056 > netcom4.netcom.com.daytime: udp 1
    4500 001d 03d7 0000 4011 179e c7b7 c684    E.....@.....
    c7b7 0968 0420 000d 0009 9c53 00          ...h. ....S.
2 12:58:56.717128 netcom4.netcom.com.daytime > bsd.1056: udp 26
    4500 0036 10f1 0000 3611 146b c7b7 0968    E..6...6..K...h
    c7b7 c684 000d 0420 0022 7256 5375 6e20    ..... .rVSun
    5365 7020 3139 2030 393a 3538 3a34 3620    Sep 19 09:58:46
    3139 3939 0a0d                           1999..

```

小结

`tcpdump` 是理解网络上发生了什么事情的一个不可缺少的工具。了解“线缆上”实际发送或接收了什么数据经常使应用程序中细小的故障得到标识和修改。它同时也作为一个学习网络动态的研究工具和教学工具充当重要的角色。作为 Stevens 的 TCP/IP Illustrated 系列的例证，它对理解网络协议的操作也很有帮助。

技巧 35 学会如何使用 traceroute

`traceroute` 实用程序是一个十分重要和有用的工具，它可以用来调试网络路由问题、研究 Internet 上的流量模式，也可以用来探索网络拓扑结构。和其他许多通常的网络工具一样，`traceroute` 是由加里弗尼亚大学的 Lawrence Berkeley 实验室开发的。

`traceroute` 的作者 Van Jacobson 在源代码的注释里写道：“我为了发现路由问题，连续 48 小时不睡眠编写出这个程序。”

`traceroute` 名字后面包含的思想很简单：通过要求每个中间的路由器发送一个 ICMP 错误消息给源主机来决定两个主机之间的网络路径。后面我们将看看它确切的实现机制，但是首先我们看看它运行的结果以及运行后提供的信息。首先我们跟踪 bsd 和南佛罗里达大学的一台计算机之间的路径（图 4.8）。我们对一些很长的输出结果行进行折行处理，以便它们能够显示在页面上。

```

bsd: $ traceroute ziggy.usf.edu
traceroute to ziggy.usf.edu (131.247.1.40), 30 hops max, 40 byte packets
1 tam- f1-pm8. netcom, net (163.179.44.15)
                                128.960 ms  139.230 ms  129.483 ms
2 tam- f1-gw1. netcom, net (163.179.44.254)

```

```

        139. 436 ms  129.226 ms  129. 570 ms
3  hl-0.mig-f1-gw1.netcom.net (165.236.144.110)
                279.582 ms  199.325 ms  289.611 ms
4  a5-0-0-7.was-dc-gwl.netcom.net (163.179.235.121)
                179. 505 ms  229. 543 ms  179.422 ms
5  hl-0.mae-east.netcom.net (163.179.220.182)
                189.258 ms  179.211 ms  169.605 ms
6  si-mae-e-f0-0.sprintlink.net (192.41.177.241)
                189.999 ms  179.399 ms  189.472 ms
7  sl-bb4-dc-1-0-0.sprintlink.net (144.228.10.41)
                180.048 ms  179.388 ms  179.562 ms
8  sl-bb10-rly-2-3.sprintlink.net (144.232.7.153)
                199.433 ms  179.390 ms  179.468 ms
9  sl-bb11-rly-9-0.sprintlink.net (144.232.0.46)
                199.259 ms  189.315 ms  179.459 ms
10 sl-bb10-orl-1-0.sprintlink.net (144.232.9.62)
                189.987 ms  199.508 ms  219.252 ms
11 sl-gw3-orl-4-0-0.sprintlink.net (144.232.2.154)
                219.307 ms  209.382 ms  209.502 ms
12 sl-usf-1-0-0.sprintlink.net (144.232.154.14)
                209.518 ms  199.288 ms  219.495 ms
13 131.247.254.36 (131.247.254.36)  209.318 ms  199.281 ms  219.588 ms
14 ziggy.usf.edu (131.247.1.40)  209.591 ms *  210.159 ms

```

图 4.8 到 ziggy.usf.edu 的 traceroute

图 4.8 中每行最左边的数字是中间节点号，之后是该中间节点主机或路由器的名称，后面跟着它的 IP 地址。如果该中间节点没有名称，traceroute 仅显示它的 IP 地址，第 13 个中间节点就是一个例子。我们将看到，默认情况下每个中间节点上的主机或路由器都会探询三次，IP 地址后面的三个数字就是这些探询的 RTT。如果一个探询没有应答，或者如果响应丢失了，就用 '*' 代替时间。

虽然 ziggy.usf.edu 和 bsd 处在同一个城市，但是在 Internet 上它们之间有 14 个中间节点。如果分析一下它们之间的路径，就可以发现它通过了 Tampa 上两个路由器，它们属于 netcom.net 网络（bsd 用来连接 Internet 的 ISP），之后又通过了两个 Netcom 路由器，然后到达华盛地区 MAE-EAST 上的 netcom.net 路由器（中间节点 5）。MAE-EAST 是一个主要的网络交换点，很多 ISP 都在上面交换流量。可以看到该路径在位于 sprintlink.net 网络的第 6 个中间节点上离开 MAE-EAST。从 Sprintlink MAE-EAST 路由器，返回东海岸，直到到达第 13 个中间节点上的 usf.edu 域。最后在第 14 个节点上到达 ziggy。

作为一个有趣的比较。看看从 bsd 到 UCLA 有多远。当然，从地理位置上来看，UCLA 位于加里弗尼亚的洛杉矶。当我们 traceroute 位于 UCLA 的主机 panther 时，我们获得图 4.9 中的结果。

```
bsd: $ traceroute panther, cs .ucla. edu
traceroute to panther.cs.ucla.edu (131.179.128.25),
          30 hops max, 40 byte packets
 1  tam- fl-pm8. netcom. net (163.179.44.15)
                  148.957 ms   129.049 ms   129.585 ms
 2  tam- fl-gwl. netcom. net (163.179.44.254)
                  139.435 ms   139.258 ms   139.434 ms
 3  hl- 0.mig- fl-gwl. netcom. net (165.236.144.110)
                  139.538 ms   149.202 ms   139.488 ms
 4  a5-0-0-7.was-dc-gwl.netcom.net (163.179.235.121)
                  189.535 ms   179.496 ms   168.699 ms
 5  h2- 0. mae-east, netcom, net (163.179.136.10)
                  180.040 ms   189.308 ms   169.479 ms
 6  cpe3- fddi- 0. washington. cw. net (192.41.177.180)
                  179.186 ms   179.368 ms   179.631 ms
 7  core5-hssi6-0-0.Washington.cw.net (204.70.1.21)
                  199.268 ms   179.537 ms   189.694 ms
 8  corerouter2.Bloomington.cw.net (204.70.9.148)
                  239.441 ms   239.560 ms   239.417 ms
 9  bordercore3.Bloomington.cw.net (166.48.180.1)
                  239.322 ms   239.348 ms   249.302 ms
10 ucla-internet-t-3.Bloomington.cw.net (166.48.181.254)
                  249.989 ms   249.384 ms   249.662 ms
11 cbn5-t3-1.cbn.ucla.edu (169.232.1.34)
                  258.756 ms   259.370 ms   249.487 ms
12 131.179.9.6 (131.179.9.6)  249.457 ms   259.238 ms   249.666 ms
13 Panther. CS.UCLA.EDU (131.179.128.25)  259.256 ms   259.184 ms *
bsd: $
```

图 4.9 到 panther.cs.ucla.edu 的 traceroute

这次我们看到该路径只有 13 个中间节点，而且它在第 11 个节点上到达 ucla.edu 域。所以 bsd 在拓扑结构上到 UCLA 的距离比到南佛罗里达大学的距离更近。

Chapman 大学也在洛杉矶附近，但是它离 bsd 仅有 9 个中间节点。这是因为 `champman.edu` 和 `bsd` 一样通过 `netcom.net` 网络连接 Internet，其流量完全在 `netcom.net` 的主干网上传输。

➤ tracerout 是如何工作的

现在让我们看看 `traceroute` 是如何工作的。回忆一下技巧 22 中的内容，每个 IP 数据报都有一个 TTL 域，它在通过每个路由器时减 1。当 TTL 值为 1（或 0）的数据报到达一个路由器时，就会被抛弃，并返回一个 ICMP “*time exceeded in transit*” 错误消息给发送者。

`traceroute` 实用程序利用了这个特性，首先发送一个 TTL 值设置为 1 的 UDP 数据报给目的主机。当 UDP 数据报到达第一个节点时，路由器注意到 TTL 设置为 1，就丢弃该数据报，然后发送一个 ICMP 消息给发起者，于是就知道了第一个节点的 IP 地址（从 ICMP 消息的源地址可以得知），`traceroute` 用 `gethostbyaddr` 查找该地址的名称。为了获得下一个节点的标识，`traceroute` 重复这个过程，但是设置 TTL 为 2，当数据报到达第一个节点，它的 TTL 减为 1 并转发到下一个节点，该节点发现数据报的 TTL 为 1，就丢弃它并返回一个 ICMP 消息。通过增加 TTL 的值重复这个过程，`traceroute` 可以获得源主机和目的主机之间的路径信息。

当数据报的初始 TTL 域足够到达目的主机时，它的 TTL 域将会变为 1，但是因为目的主机不会转发它，TCP/IP 栈将它交给一个等待程序。然而，因为 `traceroute` 发送的 UDP 数据报的目的端口是一个不会被应用程序使用的端口，所以目的主机返回一个 ICMP “*port unreachable*” 的错误消息。当 `traceroute` 接收到这消息，它就知道已经找到了目的主机，路径跟踪过程随之终止。

因为 UDP 是一个不可靠的协议（技巧 1），所以存在数据报丢失的可能性。因此 `traceroute` 对每个路由器或主机探询多次。也就是说，每个 TTL 的 UDP 数据报要发送多次。默认情况下，`traceroute` 探询每个节点 3 次，但是可以通过`-q` 标志改变该值。

同样地，`traceroute` 必须决定用多长时间来等待每个探询返回的 ICMP 消息。默认情况下，等待时间为 5 秒，但是它可以由`-w` 标志改变。如果 ICMP 在该时间间隔内没有接收到，就输出 '*' 代替 RTT 值。

这个过程可能导致几个问题。显然，`traceroute` 依赖于路由器正确地丢弃 TTL 值为 1 的 IP 数据报并且要求路由器发送“*time exceeded in transit*” ICMP 消息。不幸的是，一些路由器不发送超时消息，而是返回进来数据报中剩下的 TTL 值的一个 ICMP 消息。因为对路由器来说该值为 0，所以该消息就会被返回路径上的下一个节点抛弃（当然，第一个节点除外）。这么做的结果就跟路由器根本没有发送 ICMP 消息一样，因此这种情况也会导致输出星号。

一些路由器错误地转发 TTL 值为 0 的数据报。当发生了这种情况，下一路程段的路由器比如路由器 N+1，丢弃数据报并返回 ICMP “*time exceeded in transit*” 消息。当下一个探询消息到达时，路由器 N+1 接收 TTL 值为 1 的数据报，和通常一样返回 ICMP 错误消息。这将导致路由器 N+1 看起来像两个节点——一次是作为前一个节点的路由器发生错误的结

果，另一次是合法 TTL 丢弃的结果。图 4.10 中给出了这种情况，图 4.11 给出了一个节点 5 和节点 6 出现这种情况的一个例子。

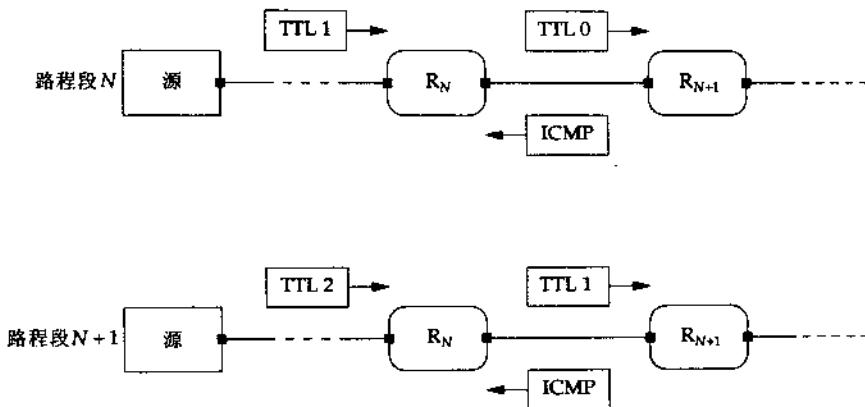


图 4.10 路由器 N 错误地发送 TTL 值为 0 的数据报

```

bsd: $ traceroute syrup.hill.com
traceroute to syrup.hill.com (208.162.106.3),
          30 hops max, 40 byte packets
 1  tam-f1-pm5.netcom.net (163.179.44.11)
                  129.120 ms  139.263 ms  129.603 ms
 2  tam-f1-gwl.netcom.net (163.179.44.254)
                  129.584 ms  129.328 ms  149.578 ms
 3  hl-0.mig-f1-gwl.netcom.net (165.236.144.110)
                  219.595 ms  229.306 ms  209.602 ms
 4  a5-0-0-7.was-dc-gwl.netcom.net (163.179.235.121)
                  179.248 ms  179.521 ms  179.694 ms
 5  h2-0.mae-east.netcom.net (163.179.136.10)
                  179.274 ms  179.325 ms  179.623 ms
 6  h2-0.mae-east.netcom.net (163.179.136.10)
                  169.443 ms  199.318 ms  179.601 ms
 7  cpe3-fddi-0.washington.cw.net (192.41.177.180)  189.529 ms
                  core6-serial5-1-0.Washington.cw.net (204.70.1.221)
                  209.496 ms  209.247 ms
 8  bordercore2.Boston.cw.net (166.48.64.1)
                  209.486 ms  209.332 ms  209.598 ms
 9  hill-associatesinc-internet.Boston.cw.net (166.48.67.54)
                  229.602 ms  219.510 ms *
10  syrup.hill.com (208.162.106.3)  239.744 ms  239.348 ms  219.607 ms

```

图 4.11 具有重复节点的 traceroute

图 4.11 还存在另一个有趣的现象。在第 7 节点，我们看到在第一次探询后路由改变了。这可能是因为第 6 个节点上的路由器执行了某种类型的负载平衡操作，或者是因为 cpe3-fddi-0.washington.cw.net 在第 7 个节点的第一次探询之后断线了，core6-serial5-1-0.washington.cw.net 取而代之。

`traceroute` 不幸遇到的另一个逐渐普遍的问题是路由器不加区别地阻止 ICMP 消息。一些机构错误地认为任何 ICMP 消息都很危险，因此阻止它们通过。这样的行为将导致 `traceroute` 毫无用处。当遇到这样的一个路由器时，对 `traceroute` 来说，就像遇到了黑洞，该路由器之外的任何信息都不会返回，这是因为它将丢弃“time exceeded in transit” 和 “port unreachable” ICMP 消息。

`traceroute` 发生的另一个值得注意的问题是路径的不对称性。当我们运行 `traceroute` 时，获得的是从源主机到目的主机的路径，但是它没有告诉我们数据报从源主机到目的主机可以采用的路径。虽然有人可能认为它们其实几乎是一样的，但是[Paxson 1997]发现所研究的路径中 49% 都存在不对称性，源主机访问的来和去的路径至少经过一个不同的节点。

通过使用-s 选项指定松散源路由通过目的主机返回到源主机。从理论上讲是可以获得源主机来和去的路径的，但是 Jacobson 在 `traceroute` 源代码注释中指出，很多路由器不能正确地处理源路由，因此该技术实际上是毫无价值的。请参阅[Stevens 1994]的第 8 章，了解该方法的解释和一个成功应用程序的例子。

Paxson 也指出我们可以预料到因为 hot potato routing 而发生的非对称路径的增加[Paxson 1995]。

在 hot potato routing 算法中，位于美国东海岸的主机 A（假定）发送一个数据报给位于西海岸的主机 B（假定）。主机 A 通过 ISP1 连接到 Internet，而主机 B 通过 ISP2 连接到 Internet。假设 ISP1 和 ISP2 都有覆盖全国的主干网。因为主干网带宽是一种稀缺的资源，ISP1 将数据报在东海岸（MAE-EAST，假定）递交给 ISP2，让它在 ISP2 的主干网上传输。同样地，当主机 B 应答时，ISP2 将应答在西海岸递交给 ISP1，于是导致不对称的路径。

Windows TRACERT

我们已经讲述了 UNIX 版本的 `traceroute`。不同版本的 Windows 操作系统中有一个相同的工具 `TRACERT`。除了不是以 UDP 数据报而是以 ICMP echo 请求（ping）探询之外，`TRACERT` 和 `traceroute` 的工作原理是相似的。目的主机的结果是返回 ICMP echo 响应而不是端口不可到达消息。当然，中间路由器仍然返回“time exceeded in transit” 消息。

`traceroute` 的最近版本有一个选项 [-I] 模仿这个操作，<ftp://ftp.ee.lbl.gov/traceroute.tar.Z> 上有 `traceroute` 的最新版本。

这个变化大概是考虑到 UDP 数据报经常被路由器过滤掉，而 ping 使用的 ICMP echo 请求/应答却很少被过滤掉。具有讽刺意义的是，初始的 `traceroute` 也使用 echo 请求来探询

路由器，但是因为许多路由器严格地遵循 RFC792[Postel 1981]的 ICMP 不应当用另一个 ICMP 消息响应一个 ICMP 消息[Jacobson 1999]的原则，所以改为使用 UDP 数据报。RFC1122[Braden 1989]认为 ICMP 消息不应当用于响应 ICMP 错误消息，但是 TRACERT 在旧路由器上仍有问题。

RFC1393[Malkin 1993]建议增加一个 IP 选项和独立的 ICMP 消息来提供可靠的 traceroute 服务（增加其他的有用信息），但是因为它需要改变路由器和主机，该方法没有被采用。

小结

traceroute 实用程序在诊断网络问题、研究网络路径或探索网络拓扑结构上是十分有用的工具。我们已经看到，Internet 的“形状”经常十分惊人，而且这也可能导致在应用程序中引入令人吃惊的影响。通过使用 traceroute，我们可以发现导致应用程序出现问题的网络异常。

traceroute 和 TRACERT 实用程序的工作原理都是以不断增长的 TTL 值发送数据报给目的主机，然后从中间路由器观察传输消息时超过的 ICMP 时间。二者之间的差别是 traceroute 发送 UDP 数据报，而 TRACERT 发送 ICMP echo 请求消息。

技巧 36 学会使用 ttcp

通常一个使用 TCP 或 UDP 发送任意数量的数据给另一台机器（甚至是同一台机器）并收集数据传输的统计信息的实用程序是很有用的。我们已经创建了几个这样的小程序，但是现在我多么希望有一个具有该功能的现成工具。我们可以使用这样的工具来驱动我们自己的应用程序或获得有关特定 TCP/IP 栈或网络的性能。这些信息可能在项目的原型确定和设计阶段是很重要的。

我们将要研究的工具是 ttcp，它来自美国 Army Ballistics Research Laboratory (BRL)，是由 Mike Muuss (ping 程序的作者) 和 Terry Slattery 开发的。该实用程序可以在 Internet 上广泛获得。我们使用的版本，由 John Lin 修改为可以报告一些额外的统计信息，可以通过匿名 FTP 从 gwen.cs.purdue.edu 下的 /pub/lin 目录下获得。没有经 Lin 修改的版本可以通过匿名 FTP 从 ftp.sgi.com 下的 /sgi/src/ttcp 目录下获得。后一个版本附带一个手册页。

我们将会看到，ttcp 有几个选项允许我们控制发送数据的数量、每个读操作和写操作的大小、套接字发送和接收缓冲区的大小、是否应当禁用 Nagle 算法以及内存的缓冲区对齐方式。图 4.12 给出了 ttcp 的用法总结。

```
Usage: ttcp-t [-options] host [ < in ]
               ttcp -r [-options > out]
Common options:
```

```

-t ## length of bufs read from or written to
    network (default 8192)
-u use UDP instead of TCP
-p ## port number to send to or listen at (default 5001)
-s -t: source a pattern to network
    -r: sink (discard) all data from network
-A align the start of buffers to this modulus (default 16384)
-O start buffers at this offset from the modulus (default 0)
-v verbose: print more statistics
-d set SO_DEBUG socket option
-b ## set socket buffer size (if supported)
-f X format for rate: k,K = kilo(bit,byte); m,M = mega;
    g,G = giga

Options specific to -t:
-n## number of source bufs written to network (default 2048)
-D don't buffer TCP writes (sets TCP_NODELAY socket option)

Options specific to -r:
-B for -s, only output full blocks as specified by -t (for TAR)
-T "touch": access each byte as it's read

```

图 4.12 ttcp 用法总结

作为一个例子，让我们实验一下套接字发送缓冲区大小。首先，我们用默认大小运行测试获得一个基线。在一个窗口中，我们启动 ttcp 的一个实例充当数据接收器。

```
bsd: $ ttcp -rsv
```

然后在另一个窗口中启动数据源：

```

bsd: $ ttcp -tsv hsd
ttcp-t: buflen=8192, nbuf=2048, align=16384/0, port=5013  tcp -> bsd
ttcp-t: socket
ttcp-t: connect
ttcp-t: 16777216 bytes in 1.341030 real seconds
        = 12217.474628 KB/sec (95.449021 Mb/sec)
ttcp-t: 16777216 bytes in 0.00 CPU seconds
        = 16384000.000000 KB/cpu sec
ttcp-t: 2048 I/O calls, msec/call = 0.67, calls/sec = 1527.18

```

```
ttcp-t: buffer address 0x8050000
bsd: $
```

我们看到，ttcp 提供给了我们传输的性能图，大约花了 1.3 秒传输 16MB 的数据。

接收进程也输出相同的统计信息，但是因为它们提供基本一样的信息，我们在这里没有把它们显示出来。

我们同时也用 tcpdump 监测数据传输。输出的典型行为：

```
13:05:44.084576 bsd.1061 > bsd.5013: . 1:1449(1448)
    ack 1 win 17376 <nop,nop,timestamp 11306 11306> (DF)
```

从中我们可以看到 TCP 发送了 1448 字节的段。

现在让我们设置数据源的输出缓冲区大小为 1448 字节重新做该实验，数据接收器跟前面的一样：

```
bsd: $ ttcp -tsvb 1448 had
ttcp-t: socket
ttcp-t: sndbuf
ttcp-t: connect
ttcp-t: buflen=8192, nbuf=2048, align=16384/0, port=5013,
      sockbufsize=1448  tcp -> bsd
ttcp-t: 16777216 bytes in 2457.246699 real seconds
      = 6.667625 KB/sec (0.052091 Mb/sec)
ttcp-t: 16777216 bytes in 0.00 CPU seconds
      = 16384000.000000 KB/cpu sec
ttcp-t: 2048 I/O calls, msec/call = 1228.62, calls/sec = 0.83
ttcp-t: buffer address 0x8050000
```

这次几乎花了 41 分钟来完成整个传输。发生了什么问题呢？我们注意到的第一件事情是虽然它花了 40 多分钟，实际的 CPU 时间却是小得不能测量。这中间不管发生了什么事情，都没有在耗费 CPU 的操作上花太多时间。

下面看看该实验的 tcpdump 输出。图 4.13 给出了典型的 4 行。

```
16:03:57.168093 bsd.1187 > bsd.5013: P 8193:9641(1448)
    ack 1 win 17376 <nop,nop,timestamp 44802 44802> (DF)
16:03:57.368034 bsd.5013 > bsd.1187: . ack 9641 win 17376
```

```

<nop,nop,timestamp 44802 44802> (DF)
16:03:57.368071 bsd.1187 > bsd.5013: P 9641:11089(1448)
    ack 1 win 17376 <nop,nop,timestamp 44802 44802> (DF)
16:03:57.568038 bsd.5013 > bsd.1187: . ack 11089 win 17376
    <nop,nop,timestamp 44802 44802> (DF)

```

图 4.13 ttcp -tsvB 1448 bsd 的典型输出

我们立刻注意到段之间的传输几乎是整整 200 毫秒，于是马上怀疑受到了 Nagle 算法/延迟 ACK 的相互影响。实际上，我们知道 ACK 延迟了。

我们可以通过-D 选项关闭 Nagle 算法来测试我们的假说。当我们禁用 Nagle 算法来重新运行我们的实验时，

```

bsd: $ ttcp -tsvDb 1448 bsd
ttcp-t: buflen=8192, nbuf=2048, align=16384/0, port=5013,
        sockbufsize=1448 tcp -> bsd
ttcp-t: socket
ttcp-t: sndbuf
ttcp-t: connect
ttcp-t: nodelay
ttcp-t: 16777216 bytes in 2457.396882 real seconds
        = 6.667218 KB/sec (0.052088 Mb/sec)
ttcp-t: 16777216 bytes in 0.00 CPU seconds
        = 16384000.000000 KB/cpu sec
ttcp-t: 2048 I/O calls, msec/call = 1228.70, calls/sec = 0.83
ttcp-t: buffer address 0x8050000

```

发现和前面没有区别——结果完全一样。

我们可以把这个当作盲目得出结论会有危险的插曲。对该问题稍微做一下考虑就会清楚 Nagle 算法与此无关，因为我们发送的是完整大小的段。实际上，这是我们基线运行时要做的事情：找出 MMS。

在技巧 39 中，我们将研究允许通过内核跟踪系统调用的工具。在那个技巧中，当我们重新研究这个例子时，将会发现 ttcp 的写操作直到大约 1.2 秒后才返回。我们也看到 ttcp 的输出中显示了每个 I/O 调用大约花费 1228 毫秒。因为技巧 15 中讨论的 TCP 写操作除非发送缓冲区已满通常不会阻塞，所以现在我们可以理解发生了什么事情。当 ttcp 写 8192 字节时，内核拷贝开始的 1448 字节到套接字缓冲区然后因没有空间而阻塞进程。TCP 在一个段中发送这些字节但是因发送缓冲区中没有更多的未发送数据而不能继续发送。

我们可以看到每个发送的段都设置了 PSH 标志，而且 BSD 派生的堆栈仅在当前传输清空发送缓冲区时才设置 PSH 标志。

因为数据接收器没有对接收的数据应答，延迟的 ACK 机制就起作用，ACK 直到 200 毫秒的计时器到时才返回。

在基线运行时，TCP 能够持续发送完全大小的段，这是因为套接字缓冲区足够大（bsd 为 16K），可以保存几个段。基线运行时的内核跟踪显示在那种情况下，写操作大约 0.3 秒后返回。

我们在这里要说明的是保持发送缓冲区大小至少为对等方接收缓冲区大小的重要性。虽然接收方希望接收更多的数据，但是发送方的输出缓冲区让最后发送的段填满了，发送方在接收到对等方已经接收数据的 ACK 到来之前不能释放数据。因为单一段和 16K 接收缓冲区比较起来显得很小，所以段的接收不会触发窗口更新（技巧 15），因此 ACK 延迟了 200 毫秒。请参阅技巧 32 了解缓冲区大小的更多信息。

虽然这个实验的主要目的不是演示有关发送和接收缓冲区的相对大小，但是它可以显示我们如何使用 ttcp 来测试设置不同 TCP 连接参数带来的影响。我们也可以看到如何从 ttcp 统计、tcpdump 和内核系统调用跟踪中收集数据理解所观察到的行为。

在离开 ttcp 的讨论之前，我们应当提一下，它也可以用于提供主机之间的“网络管道”。例如，它有一种方法从主机 A 到主机 B 拷贝目录结构。在主机 B 上输入命令

```
ttcp -rB | tar -xpf -
```

在主机 A 上输入命令

```
tar -cf - directory | ttcp -t A
```

我们可以通过在中间机器上使用下列命令扩展网络管道到多个机器

```
ttcp -r | ttcp -t next_hop
```

小结

本节我们了解了如何使用 ttcp 实验不同的 TCP 连接参数。正如我们提到的那样，ttcp 也可以通过提供一个 TCP 或 UDP 数据源或接收器来测试我们自己的应用程序。最后，我们看到可以使用 ttcp 在两个或更多的机器之间提供网络管道。

技巧 37 学会使用 lsof

网络编程（不涉及网络的编程也存在这个问题）中经常遇到的问题是哪个进程打开文件或套接字。这在网络编程中尤其重要。这是因为如同在技巧 16 里看到的，当实际使用套

接字的进程退出或关闭它的时候，其他打开套接字的进程将阻止发送 FIN 消息。

虽然其他进程似乎不太可能打开套接字，但是这很容易发生，特别是在 UNIX 环境中。经常发生的情况是一个进程创建连接并启动另一个进程来处理连接。举个例子来说，inetd（请参阅技巧 17）的工作原理就是这样。如果建立连接的进程在启动子进程之后不能关闭套接字，套接字的引用数将为 2，因此当子进程关闭套接字时，连接仍然保持开状态，于是不会发送 FIN 消息。

该问题在另一个情况下也可以发生。假定和子进程会话的客户端主机崩溃了，子进程挂起，我们在技巧 10 里讨论了这个问题。如果建立连接的进程终止了，因为本地端口已经绑定到子进程，所以它不能重新启动（除非它已经指定 SO_REUSEADDR——请参阅技巧 23）。

在这些情况和其他情况下，必须知道哪个进程或哪几个进程打开了套接字。netstat 实用程序（请参阅技巧 38）告诉我们一些进程正在使用某个特定的端口或地址，但是它没有告诉我们到底是哪一个进程。一些 UNIX 系统让 fstat 实用程序做这个工作，但是并不是所有的系统都有它。幸运的是，Victor Abell 的公共域 lsof 程序可以在很多 UNIX 系统上获得。

lsof 程序可以通过匿名 FTP 从 vic.cc.purdue.edu 的 /pub/tools/unix/lsof 目录下获得。

lsof 特别灵活——它的手册页有 36 页——而且它可以提供大量的有关打开文件的信息。和 tcpdump 一样，为多个 UNIX 系统提供单一公共接口是一个很大的进步。

让我们看看它在网络中很有用的特性。它的手册页包含大量的有关不同设置的用法信息。

假定我们执行 netstat -af inet（请参阅技巧 38），而且注意到某个进程正在端口 6000 上监听，部分输出显示如下：

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address      Foreign Address        (state)
Tcp      0      0  *.6000                *.*                  LISTEN
```

端口 6000 不输入“已知端口”的范围（请参阅技巧 18），因此我们想知道哪个服务器正在监听端口上面连接。我们前面已经提到了，netstat 输出对此毫无所知——它仅仅告诉我们某个进程正在那个端口上监听。幸运的是，该问题可以通过 lsof 轻松地解答：

```
bsd# lsof -i TCP:6000
COMMAND   PID USER FD  TYPE      DEVICE SIZE/OFF NODE NAME
XF86_Mach 253 root  0u  inet 0xf5d98840      0t0    TCP *:6000 (LISTEN)
```

需要注意的第一件事情是，我们是作为 root 运行 lsof 的。这是因为我们使用的 lsof 的版本只列出属于用户 root 的文件。这是一个可以在编译时禁用的安全特性。接下来我们看

到进程由 root 通过命令 XF86_Mach 启动，它是运行在主机上的 X-server。

-i TCP: 6000 告诉 lsof 查询绑定到端口 6000 上的打开的“inet”TCP 套接字。我们可以使用 -i TCP 查找所有的 TCP 套接字，而使用 -i 查找所有的 TCP/UDP 套接字。

下面假定我们运行另一个 netstat 并发现有人建立了到 vic.cc.purdue.edu 的 FTP 连接。

```
Active Internet connections
Proto Recv-Q Send-Q Local Address Foreign Address      (state)
Tcp      0      0 bsd.1124      vic.cc.purdue.edu.ftp ESTABLISHED
```

我们可以用 lsof 找出这是哪个用户：

```
bsd# lsof -i @vic.cc.purdue.edu
COMMAND PID USER   FD   TYPE      DEVICE SIZE/OFF NODE NAME
ftp     450 jcs    3u  inet 0xf5d99f00  0t0  TCP bsd:1124->
       vic.cc.purdue.edu:ftp (ESTABLISHED)

bsd#
```

和以往一样，我们从 bsd 删除了域名，而且因为格式原因对输出进行了换行处理，这样可以从输出看出用户 jcs 打开了 FTP 连接。

我们应当强调，lsof 的名字的意思毕竟为 list open files，它只可以提供已经打开的信息。这意味着它不能给我们任何处于 TIME-WAIT 状态（技巧 22）的 TCP 连接的信息，这是因为没有打开的套接字或文件关联到这个连接上。

小结

本节我们知道如何使用 lsof 实用程序来回答有关打开文件的很多问题。虽然主要在网络上讨论这个应用程序，但是 lsof 可以提供有关不同打开文件类型的信息。不幸的是，lsof 只可以在 UNIX 系统上获得，它没有 Windows 版本。

技巧 38 学会使用 netstat

内核维持着很多有关网络对象有用统计信息，我们可以用 netstat 查询它们。可以获得四种类型的信息。

活动套接字

首先，我们获取有关活动套接字的信息。虽然 netstat 可以提供集中类型的套接字的信

息，但是我们只对那些有关 inet 域和 UNIX 域套接字感兴趣；也就是说，那些 AF_INET 和 AF_LOCAL（或 AF_UNIX）的域。我们可以列出所有类型的套接字，也可以用-f 选项指定地址组来选择要列出的套接字类型。

在默认情况下，套接字绑定到 INADDR_ANY 的那些服务器没有列出来。但是这可以通过指定-a 选项来改变。例如，如果我们只对 TCP/UDP 套接字感兴趣，我们可以以下面的方式调用 netstat：

```
bsd: $ netstat -f inet
Active Internet connections
Proto Recv-Q Send-Q Local Address          Foreign Address      (state)
tcp      0      0  localhost.domain        *.*                LISTEN
tcp      0      0  bsd.domain              *.*                LISTEN
udp      0      0  localhost.domain        *.*                LISTEN
udp      0      0  bsd.domain              *.*                LISTEN
bsd: $
```

输出只显示运行在 bsd 上的域名服务器（有名称的）。如果我们要要求查找所有的服务器，我们将获得：

```
bsd: $ netstat -af inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address          Foreign Address      (state)
tcp      0      0  *.6000                 *.*                LISTEN
tcp      0      0  *.smtp                 *.*                LISTEN
tcp      0      0  *.printer              *.*                LISTEN
tcp      0      0  *.rlnum                *.*                LISTEN
tcp      0      0  *.tcpmux               *.*                LISTEN
tcp      0      0  *.chargen              *.*                LISTEN
tcp      0      0  *.discard              *.*                LISTEN
tcp      0      0  *.echo                 *.*                LISTEN
tcp      0      0  *.time                 *.*                LISTEN
tcp      0      0  *.daytime              *.*                LISTEN
tcp      0      0  *.finger               *.*                LISTEN
tcp      0      0  *.login                *.*                LISTEN
tcp      0      0  *.cmd                  *.*                LISTEN
tcp      0      0  *.telnet               *.*                LISTEN
```

```

tcp      0      0  *.ftp          *.*        LISTEN
tcp      0      0  *.1022         *.*        LISTEN
tcp      0      0  *.2049         *.*        LISTEN
tcp      0      0  *.1023         *.*        LISTEN
tcp      0      0  *.sunrpc       *.*        LISTEN
tcp      0      0  localhost.domain *.*        LISTEN
tcp      0      0  bsd.domain     *.*        LISTEN
udp      0      0  *.udpecho      *.*        LISTEN
udp      0      0  *.chargen      *.*        LISTEN
udp      0      0  *.discard      *.*        LISTEN
udp      0      0  *.echo         *.*        LISTEN
udp      0      0  *.time         *.*        LISTEN
udp      0      0  *.daytime       *.*        LISTEN
udp      0      0  *.ntalk         *.*        LISTEN
udp      0      0  *.biff          *.*        LISTEN
udp      0      0  *.1011         *.*        LISTEN
udp      0      0  *.nfsd          *.*        LISTEN
udp      0      0  *.1023         *.*        LISTEN
udp      0      0  *.sunrpc       *.*        LISTEN
udp      0      0  *.1024         *.*        LISTEN
udp      0      0  localhost.domain *.*        LISTEN
udp      0      0  bsd.domain     *.*        LISTEN
udp      0      0  *.syslog        *.*        LISTEN

bsd: $

```

如果我们要运行 lsof (技巧 37)，我们将看到实际上大多数的这些“服务器”是 inetd (请参阅技巧 17) 监听的标准服务的连接或数据报。TCP 连接的状态栏下的“LISTEN”意思是服务器正在等待客户端连接。

如果我们 telnet 到 echo 服务器

```
bsd: $ telnet bsd echo
```

我们看到有一个 ESTABLISHED 连接：

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp	0	0	bsd.echo	bsd.1035	ESTABLISHED

```

tcp      0      0  bsd.1035          bsd.echo          ESTABLISHED
tcp      0      0  *. echo           *.*              LISTEN

```

我们已经从输出中删除了那些和 echo 服务器无关的行。注意，因为连接的是同一个机器，所以连接两次出现在 netstat 列表中——一次为客户端，另一次为服务器。还要注意 inetd 仍然监听到 echo 服务器的额外连接。

最后一点值得讨论一下。虽然我们的 telnet 客户端已经连接到端口 7 (echo 端口)，而且它实际上使用端口 7 作为它的目的端口，主机仍然在端口 7 上监听连接。这不会有任何问题，因为 TCP 通过技巧 23 中的由本地 IP 地址和端口号以及远程 IP 地址和端口号的 4-tuple 地址来标识连接。正如我们看到的那样，inetd 在星号表示的通配地址 (INADDR_ANY) 上监听，而连接的 IP 地址为“bsd”。如果我们想启动跟 echo 服务器的另一个 telnet 连接，除了客户端的源端口号变为 1025 之外，我们还要有和前面两个看起来一样的条目。

如果我们现在终止客户端并再次运行 netstat，将获得：

```

Proto Recv-Q Send-Q Local Address      Foreign Address      (state)
tcp      0      0  bsd. 1035          bsd. echo          TIME_WAIT

```

这显示了连接的客户端方处于 TIME_WAIT 状态，这在技巧 22 中我们讨论过了。状态栏中也有可能出现其他可能的状态。请参阅 RFC 793[Postel 1981b]，了解有关 TCP 连接状态的知识。

接口

我们可以从 netstat 获得的第二种类型的数据是有关接口的信息。我们已经在技巧 7 中看到了这个例子。我们通过使用-I 选项获得基本的接口信息。

```

bsd: $ netstat -i
      Name  Mtu   Network     Address      Ipkts  Ierrs    Opkts  Oerrs  Coll
      ed0   1500 <Link>  00.00.c0.54.53.73  40841    0  5793    0    0
      ed0   1500   172.30      bsd          40841    0  5793    0    0
      tun0*  1500 <Link>                  397     0   451    0    0
      tun0*  1500  205.184.142 205.184.142.171397  0     0   451    0    0
      s10*   552   <Link>                  0     0     0    0    0
      lo0   16384 <Link>                  353     0   353    0    0
      lo0   16384  127      localhost      353     0   353    0    0

```

从这个输出中可以看出 bsd 配置了四个接口。第一个 ed0 是 Ethernet LAN 适配器。我们可以看出它位于私有 (RFC 1918[Rekhter, Moskowitz et al. 1996]) 网络 172.30.0.0 上。第一个条目中的 00.00.c0.54.73 是 Ethernet 网卡的介质访问控制 (MAC) 地址。那里有 40841 个输入数据包、5793 个输出数据包，没有错误，没有冲突。这在技巧 7 中讨论过了。MTU 是 1500，多数 Ethernet 可以接受这个设置。

tun0 接口是一个拨号 Point-to-Point Protocol (PPP) 链接。它位于 205.184.142.0 网络上。起 MTU 为 1500 字节。

sl0 接口是串行线路网际接口协议 (Serial Line Internet Protocol, SLIP; RFC 1055[Romkey 1988])，这是另一个更老的点对点协议。该接口在 bsd 上没有使用。

我们也可以通过指定 -b 和/或 -d 选项联合 -I 选项来列出接口进来和出去的字节数以及丢弃的数据包的数量。

路由表

从 netstat 可以获得的下一个信息是路由表。我们使用 -n 选项来请求 IP 地址的数字而不是名称，这样我们就可以看到实际涉及的网络。

```
bsd: $ netstat -rn
Routing tables
Internet:
Destination      Gateway          Flags   Refs      Use     Netif Expire
default          163.179.44.41    UGSc        2          0     tun0
127.0.0.1        127.0.0.1      UH          1         34     lo0
163.179.44.41    205.184.142.171 UH          3          0     tun0
172.30           link#1         UC          0          0     ed0
172.30.0.1       0:0:c0:54:53:73 UHLW         0        132     lo0
bsd: $
```

图 4.14 netstat 输出的路由表

图 4.14 中报告的接口和连接显示在图 4.15 中。接口 lo0 没有显示出来是因为它完全包含在 bsd 主机中。

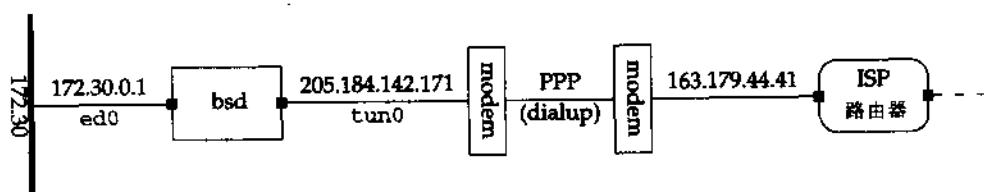


图 4.15 中报告的接口和主机

在我们看该输出的各个条目之前，先讨论一下个列的含义。第一列表示路由的目的地。它可以是一个特定的主机、网络或者是默认的路由。

Flags 域里可以有几个标志，大多数是具体实现有关的。我们只考虑：

U——路由是“UP”。

H——这是一个到主机的路由。如果没有给出这个标志，路由是到网络的（如果没有使用 CIDR（请参阅技巧 2）就可能为子网）。

G——路由是间接的。也就是说，目的地址不是直接连接的，而是必须通过中间路由器或网关（G）才能到达。

很容易误认为 H 和 G 标志是同一件事情的相反方面：路由要么是到主机（H），要么是到中间网关（G）。这种认识是不正确的。H 标志告诉我们第一栏中的地址是一个主机的完全地址。没有 H 说明第一栏中的地址不包含主机 ID 部分——它是网络的地址。标志 G 告诉第一栏中的地址是否可以从这个主机直接到达，或者我们必须通过中间的网关。

完全可以让一个路由具有 G 和 H 标志。例如，考虑一下图 4.16 中显示的两个网络，主机 H1 和 H2 连接到 Ethernet 网络 172.20，H3 通过 PPP 连接网络 198.163.2。

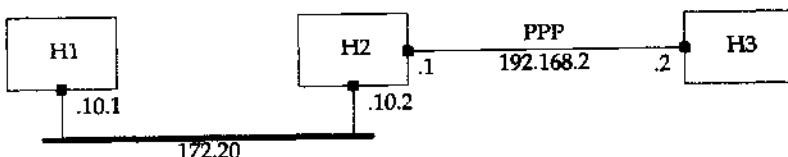


图 4.16 H2 充当 H3 的网关

到 H3 的路由可能在 H1 的路由表中显示为

Destination	Gateway	Flags	Refs	Use	Netif	Expire
192.168.2.2.	172.20.10.2	UGH	0	0	ed0	

设置了 H 标志是因为 192.168.2.2 是一个完全的主机地址。设置了 G 标志是因为 H1 没有直接连接到 H3，但是必须通过 H2 (172.20.10.2) 来到达它。注意图 4.14，到 163.179.44.41 没有设置 G 标志，这是因为 163.179.44.41 直接连接到 bsd 的 tun0 接口 (205.184.142.171)。

在图 2.9 中，可能在 H1 的路由表中没有 H3 的条目。而且，可能存在 190.50.2 子网的条目——这是使用子网的目的：为了减少路由表的大小，该子网中 H1 的路由条目可能为：

Destination	Gateway	Flags	Refs	Use	Netif	Expire
190.50.2	190.50.1.4	UG	0	0	ed0	

没有设置 H 标志是因为 190.50.2 是子网的地址，不是一个主机。设置了 G 标志是因为 H3 没有直接连接到 H1。从 H1 到 H3 的数据报必须通过路由器 R1 (190.50.1.4)。

Gateway 域的意义依赖于是否设置了 G 标志。当路由为间接时(设置了 G 标志), **Gateway** 域包含了如何到达直接连接目的地址的信息。在许多具体实现中, 这总是目的地址直接连接的接口的 IP 地址。在 BSD 派生的具体实现中, 它可能是图 4.14 中最后一行显示的链路层 MAC 地址。在这种情况下, 设置了 L 标志。

Refs 域告诉我们该路由的引用次数。也就是说, 路由有多少个活动的使用。

Use 域告诉我们使用路由发送了多少数据包, **Netif** 域是关联接口的名称。该接口和用 -i 选项获得的信息是同一个对象。

现在我们看看 netstat -rn 输出域的意思, 最后我们可以检查一下图 4.14 中的条目。

图 4.14 中的第一行是默认路由。这是当路由表不包含特定路由时 IP 数据报发送到的地方。例如, 如果我们 ping netcom4.netcom.com, 将获得:

```
bsd: $ ping netcom4.netcom.com
PING netcom4.netcom.com (199.183.9.104): 56 data bytes
64 bytes from 199.183.9.104: icmp_seq=0 ttl=248 time=268.604 ms
...
...
```

因为没有到 199.183.9.104 或任何包含该主机的路由, 所以 ICMP echo 请求(请参阅技巧 33) 就发送到默认的路由上。根据 netstat 输出的第一行, 因为该路由的网关是 163.179.44.41, 所以数据报应当发送到那里。图 4.14 中的第 3 行有一个到 163.179.44.41 的直接路由, 因此发送到它那里的数据报应当发送到 IP 地址为 205.184.142.171 的接口上。

输出的第 2 行是 loopback 地址(127.0.0.1) 的路由。因为这是一个主机地址, 所以没有设置 H 标志。因为它是直接连接, 所以设置了 G 标志, **Gateway** 域告诉我们接口(lo0)的 IP 地址。

图 4.14 中的第 4 行是到本地 Ethernet 的路由。因为 bsd 是 BSD 派生的系统, **Gateway** 域指定为 Link#1。其他系统可能简单地把接口的 IP 地址(172.30.0.1) 连接到 LAN。

协议统计

从 netstat 可以获得的最后一类信息是协议统计。如果我们指定了-s 选项, netstat 就提供有关 IP、ICMP、IGMP、UDP 和 TCP 协议的信息。如果我们只对其中一个协议感兴趣, 可以用-p 选项指定它。例如, 为了获得有关 UDP 的统计, 我们可以使用:

```
bsd: $ netstat -sp udp
udp:
 82 datagrams received
 0 with incomplete header
 0 with bad data length field
```

```

0 with bad checksum
1 dropped due to no socket
0 broadcast/multicast datagrams dropped due to no socket
0 dropped due to full socket buffers
0 not for hashed pcb
81 delivered
82 datagrams output
bsd: $

```

我们可以通过重复 s 选项来抑制没有发生的统计。

```

bsd: $ netstat -ssp udp
udp:
82 datagrams received
1 dropped due to no socket
81 delivered
82 datagrams output
bsd: $

```

偶尔检查一下 TCP 统计信息是一个明智而且值得的做法。在 bsd 系统上，netstat 报告了 45 个不同的 TCP 统计。下面是用-ssp tcp 选项调用 netstat 收集到的非零统计信息。

```

tcp:
446 packets sent
    190 data packets (40474 bytes)
    213 ack-only packets (166 delayed)
    18 window update packets
    32 control packets
405 packets received
    193 acks (for 40488 bytes)
    12 duplicate acks
    302 packets (211353 bytes) received in-sequence
    10 completely duplicate packets (4380 bytes)
    22 out-of-order packets (16114 bytes)
    2 window update packets
20 connection requests

```

```

2 connection accepts
13 connections established (including accepts)
22 connections closed (including 0 drops)
    3 connections updated cached RTT on close
    3 connections updated cached RTT variance on close
2 embryonic connections dropped
193 segments updated rtt (of 201 attempts)
31 correct ACK header predictions
180 correct data packet header predictions

```

这些统计信息是在 bsd 重新启动后发送和接收几个邮件消息并读取一些新闻组收集到的。如果我们认为乱序或重复段是不太可能发生的时间的话，那么这些统计信息就会驱散这些错误的想法。例如，在接收到的 405 个数据包中，有 10 个是重复的 22 个是乱序的。

[Bennett et al. 1999]说明了数据包的重新排序为什么不是必不可少的不正常行为，以及我们可能在以后会看到更多的这些东西。

» Windows netstat

我们已经讨论了 UNIX 家族的 netstat 实用程序。Windows 环境也有一个 netstat，它提供了许多相同的参数和数据。虽然输出大体上相似，但是不是很全面，尽管如此它仍然提供了很多有用的信息。

» 小结

我们已经分析了 netstat 实用程序以及它提供的一些有关网络对象的信息。我们已经看到 netstat 可以告诉我们有关活动套接字、配置在系统上的网络接口、路由表以及协议统计信息。

虽然本节很长而且包含了很多信息，但是它反映了该实用程序本身是很重要的。netstat 提供了网络子系统多个方面的信息，因此有多种不同类型的输出。

技巧 39 学会使用系统调用跟踪工具

当调试网络应用程序时，能够通过系统内核跟踪系统调用有时很有用。我们已经在技巧 36 中看到了这样的一个例子，我们将再次简要地讨论这个例子。

许多操作系统提供一些跟踪系统调用的方法。BSD 世界里有一个工具 ktrace，SVR4 世界里有 truss，Linux 系统里有 strace。

因为这些实用程序基本相似，所以我们通过 `ktrace` 演示它们的用法。看看 `truss` 或 `strace` 的手册页可以帮助我们移植相同的技术到其他环境中。

» 不成熟的终止

第一个例子技巧 16 中开发的基于 `shutdown` 程序的早期版本（图 3.2）。`badclient` 后面的思想和 `shutdownnc` 后面的思想是一样的：我们从标准输入接受输入直到获得 EOF 为止。这时，我们调用 `shutdown` 发送给对等方 FIN 消息，然后继续从对等方读取输入直到在连接上接收了说明对等方已经结束发送数据的 EOF 字符。`badclieng` 程序在图 4.17 中显示。

badclient.c

```

1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4     SOCKET s;
5     fd_set readmask;
6     fd_set allreads;
7     int rc;
8     int len;
9     char lin[ 1024 ];
10    char lout[ 1024 ];

11    INIT();
12    s = tcp_client( argv[ optind ], argv[ optind + 1 ] );
13    FD_ZERO( &allreads );
14    FD_SET( 0, &allreads );
15    FD_SET( s, &allreads );
16    for ( ; )
17    {
18        readmask = allreads;
19        rc = select( s + 1, &readmask, NULL, NULL, NULL );
20        if ( rc <= 0 )
21            error( 1, errno, "bad select return (%d)", rc );
22        if ( FD_ISSET( s, &readmask ) )
23        {
24            rc = recv( s, lin, sizeof( lin ) - 1, 0 );
25            if ( rc < 0 )

```

```

26             error( 1, errno, "recv error" );
27         if ( rc == 0 )
28             error( 1, 0, "server disconnected\n" );
29         lin[ rc ] = '\0';
30         if ( fputs( lin, stdout ) )
31             error( 1, errno, "fputs failed" );
32     }
33     if ( FD_ISSET( 0, &readmask ) )
34     {
35         if ( fgets( lout, sizeof( lout ), stdin ) == NULL )
36         {
37             if ( shutdown( s, 1 ) )
38                 error( 1, errno, "shutdown failed" );
39         }
40     else
41     {
42         len = strlen( lout );
43         rc = send( s, lout, len, 0 );
44         if ( rc < 0 )
45             error( 1, errno, "send error" );
46     }
47 }
48 }
49 }
```

badclient.c

图 4.17 不正确的 echo 客户端

22~32 如果 select 指示在连接上有读事件，程序就试图读取数据。如果程序获得了一个 EOF 字符，就说明对等方已经结束发送数据，因此程序终止。否则，把刚才接收到的数据写入到标准输出。

33~47 如果 select 指示在标准输入上有读事件，程序调用 fgets 读取数据。如果 fgets 返回 NULL，这说明发生了错误或者读到了 EOF 字符，程序调用 shutdown 通知对等方我们结束数据发送。否则，我们发送数据给对等方。

现在让我们看看运行 *badclient* 时发生了什么事情。我们使用 *tcpecho*（图 3.3）作为该实验的服务器。回忆一下技巧 16，我们可以指定 *tcpedho* 在反射它的响应之前应当延迟的秒数。程序设置该延迟为 30 秒。在启动 *badclient* 之后，我们输入“hello”，之后立即跟着<CNTRL-D>作为 fgets 的 EOF 字符。

```

bsd: $ tcpecho 9000 30
                                30 seconds later
tcpecho: recv failed:
      Connection reset by peer (54)
bsd: $

```

bsd: \$ tcpecho 9000 30 30 seconds later tcpecho: recv failed: Connection reset by peer (54) bsd: \$	bsd: \$ badclient bsd 9000 hello ^D badclient: server disconnected bsd: \$
--	--

我们看到 `badclieng` 立即终止，报出 `tcpecho` 断开连接的错误。但是 `tcpecho` 仍然运行而且直到 30 秒的延迟到时了还保持睡眠。在同一时刻，它从读操作那里获得一个“`Connection reset by peer`”。

这是令人感到吃惊的结果，我们希望在 30 秒之后看到 `tcpecho` 返回消息（`badclient` 打印）然后在下一个读操作上获得 EOF 字符之后终止。相反的是，`badclient` 立即终止而 `tcpecho` 得到读错误。

诊断这个问题的第一步是使用 `tcpdump`（请参阅技巧 34）看看这两个程序发送和接收了什么东西，图 4.18 给出了 `tcpdump` 的输出，输出结果已经删除了连接建立阶段并进行了折行处理。

```

1 18:39:48.535212 bsd.2027 > bsd.9000:
    P 1:7(6) ack 1 win 17376 <nop,nop, timestamp 742414 742400> (DF)
2 18:39:48.546773 bsd.9000 > bsd.2027:
    . ack 7 win 17376 <nop,nop,timestamp 742414 742414> (DF)
3 18:39:49.413285 bsd.2027 > bsd.9000:
    F 7:7(0) ack 1 win 17376 <nop,nop, timestamp 742415 742414> (DF)
4 18:39:49.413311 bsd.9000 > bsd.2027:
    .ack 8 win 17376 <nop,nop,timestamp 742415 742415> (DF)
5 18:40:18.537119 bsd.9000 > bsd.2027:
    P 1:7(6) ack 8 win 17376 <nop,nop, timestamp 742474 742415> (DF)
6 18:40:18.537180 bsd.2027 > bsd.9000:
    R 2059690956:2059690956(0) win 0

```

图 4.18 `badclient` 的 `tcpdump` 输出

奇怪的是，除了最后一行外所有的事情都很正常。我们看到 `badclient` 在第一行发送“`hello`”给 `tcpecho`，而且我们看到在第 3 行 FIN 在 1 秒后触发 shutdown。`tcpecho` 对两个消息在第 2 和第 4 行进行了期望的 ACK 响应。`badclient` 发送“`hello`”之后的 30 秒，我们看到 `tcpecho` 在第 5 行把它发送回去，但是没有响应 ACK 消息，连接的 `badclient` 方返回一个 RST（第 6 行）导致 `tcpecho` 的“`Connection reset by peer`”错误。RST 消息是因为 `badclient` 已经终止而发出的，但是它为什么要这么做呢？`tcpdump` 的输出没有任何线索。

因为 `tcpdump` 输出确实告诉我们 `tcpecho` 没有做任何事情导致对等方提前终止，所以问题一定完全出在 `badclient` 中。现在必须看看 `badclient` 里面究竟有些什么了，可以采用的一个方法就是看它调用了什么系统调用。

最后，我们重新运行实验，但是以下面的方式调用 `badclient`:

```
bsd: $ ktrace badclient bsd 9000
```

这跟以前一样运行 `badclient`，但是对它调用的系统调用进行了跟踪。默认地，跟踪结果写入到文件 `ktrace.out` 中。为了输出 `trace` 的结果，我们运行 `kdump`。图 4.19 显示了 `kdump` 的结果，其中删除了几个启动应用程序和连接建立的初步的调用。

```
1 4692 badclient CALL  read(0,0x804e000,0x10000)
2 4692 badclient GIO    fd 0 read 6 bytes
  "hello
  "
3 4692 badclient RET    read 6
4 4692 badclient CALL  sendto(0x3,0xefbfce68,0x6,0,0,0)
S 4692 badclient GIO    fd 3 wrote 6 bytes
  "hello
  "
6 4692 badclient RET    sendto 6
7 4692 badclient CALL  select(0x4,0xefbfd6f0,0,0,0)
8 4692 badclient RET    select 1
9 4692 badclient CALL  read(0,0x804e000,0x10000)
10 4692 badclient GIO   fd 0 read 0 bytes
  ""
11 4692 badclient RET   read 0
12 4692 badclient CALL  shutdown(0x3,0x1)
13 4692 badclient RET   shutdown 0
14 4692 badclient CALL  select(0x4,0xefbfd6f0,0,0,0)
15 4692 badclient RET   select 1
16 4692 badclient CALL  shutdown(0x3,0x1)
17 4692 badclient RET   shutdown 0
18 4692 badclient CALL  select(0x4,0xefbfd6f0,0,0,0)
19 4692 badclient RET   select 2
20 4692 badclient CALL  recvfrom(0x3,0xefbfd268,0x3ff,0,0,0)
```

```

21 4692 badclient GIO    fd 3 read 0 bytes
  "hello\n"

22 4692 badclient RET    recvfrom 0
23 4692 badclient CALL  write(0x2,0xefbfcc6f4,0xb)
24 4692 badclient GIO    fd 2 wrote 11 bytes
  "badclient: "
25 4692 badclient RET    write 11/0xb
26 4692 badclient CALL  write (0x2,0xefbfcc700,0x14)
27 4692 badclient GIO    fd 2 wrote 20 bytes
  "server disconnected"
  "
28 4692 badclient RET    write 20/0x14
29 4692 badc 1 lnt CALL exit ( 0x1 )

```

图 4.19 在 badclient 上运行 ktrace 的结果

开始的两个条目是进程 ID 和执行程序的名称。在第 1 行，我们看到以 fd 设置为 0（标准输入）调用 read。第 2 行，读取了 6 个字节（“GIO”代表 general I/O），还读取了“hello\n”。第 3 行显示 read 调用返回 6，读取的字节数。类似地，第 4-6 行显示 badclient 把这些写入到 fd 3 上，套接字连接到 tcpecho。下面的第 7 和第 8 行显示 select 返回 1，意思是一个事件已经准备好了。第 9-11 行我们看到 badclient 在标准输入上读取 EOF 字符，并在第 12 行和第 13 行调用 shutdown。

到目前为止，所有的事情都是意料之中的，但是第 14-17 行出现了意外：select 返回单一事件而 shutdown 再次调用，检查一下图 4.17，我们直到当描述符 0 已经准备好了时才会发生这种情况。这里没有调用我们所期望的 read。因为当我们键入<CTRL-D>时，fgets 在 EOF 处标记了流，所以它没有调用 read 就返回了。

例如，我们可以在[Kernighan and Ritchie 1988]fgets 的具体实现（通过 getc）样本程序中使用这种方法。

第 18 和 19 行 select 返回在标准输入和套接字中都有的事件。第 20~22 行显示 recvfrom 返回 0（EOF 字符），trace 的其余部分显示 badclient 写它的错误消息并终止。

现在很清楚发生了什么：select 在第 15 行返回准备好读操作的标准输入，这是因为我们在第一个 shutdown 之后为标准输入调用 FD_CLR 没有成功。随后的 shutdown 调用（第二个）导致 TCP 关闭连接。

我们可以在[Wright and Stevens 1995]的第 1014 页上看到这种情况，其中 tcp_usrclosed 是作为 shutdown 的结果调用的。如果已经调用了 shutdown，那么连接就处于 FIN-WAIT-2 状态，tcp_usrclosed 在第 1021 的第 444 行调用 oisdisconnected。这个调用将完全关闭套接字并导致 select 返回一个 EOF 的读事件。

因为连接已经关闭了，所以 `recvfrom` 返回 0，说明这是一个 EOF 字符，`badclient` 输出“server disconnected”诊断信息并终止。

理解本例子中发生了什么事情的关键是对 `shutdown` 的第二个调用。一旦我们看到它，就只需要问自己它是如何发生的，于是从那里可以很容易地发现遗漏的 FD_CLR。

» 不好的 ttcp 性能

下一个例子是技巧 36 中开始的例子的继续。回忆一下在那个例子中，当我们设置套接字发送缓冲区为连接 MSS 的大小时，传输 16MB 的数据所花费的时间从默认缓冲区大小的 1.3 秒到几乎 41 秒。

我们从实验中取出一些 `ktrace` 具有代表性的结果显示出来。以下的方法调用 `kdump`：

```
kdump -R -m -l
```

来在条目之间输出相对时间并禁止输出和每个写操作有关的 8KB 的数据。

```
12512 ttcp      0.000023 CALL  write(0x3,0x8050000,0x2000)
12512 ttcp      1.199605 GIO   fd 3 wrote 8192 bytes
    ""
12512 ttcp      0.000446 RET    write 8192/0x2000
12512 ttcp      0.000022 CALL  write(0x3,0x8050000,0x2000)
12512 ttcp      1.199574 GIO   fd 3 wrote 8192 bytes
    ""
12512 ttcp      0.000442 RET    write 8192/0x2000
12512 ttcp      0.000023 CALL  write(0x3,0x8050000,0x2000)
12512 ttcp      1.199514 GIO   fd 3 wrote 8192 bytes
    ""
12512 ttcp      0.000432 RET    write 8192/0x2000
```

图 4.20 ttcp-tsvb 1448 bsd 的具有代表性的 ktrace 结果

注意每个写操作的时间都一致为 1.2 秒。为了对比的需要，基线（baseline）运行的结果显示在图 4.21 中。在那里，我们看到尽管有多个变量，但是平均值仍低于 0.5 毫秒。

```
12601 ttcp      0.000033 CALL  write(0x3,0x8050000,0x2000)
12601 ttcp      0.000279 GIO   fd 3 wrote 8192 bytes
    ""
12601 ttcp      0.000360 RET    write 8192/0x2000
```

```

12601 ttcp      0.000033 CALL  write(0x3,0x8050000,0x2000)
12601 ttcp      0.000527 GIO    fd 3 wrote 8192 bytes
"""

12601 ttcp      0.000499 RET   write 8192/0x2000
12601 ttcp      0.000032 CALL  write(0x3,0x8050000,0x2000)
12601 ttcp      0.000282 GIO    fd 3 wrote 8192 bytes
"""

12601 ttcp      0.000403 RET   write 8192/0x2000

```

图 4.21 ttcp -tsv bsd 的具有代表性的 ktrace 结果

图 4.20 中 GIO 条目长时间和图 4.21 中短时间的比较给了我们技巧 36 中的结论，我们的写操作在内核阻塞了，从那个地方开始我们就能够理解导致大传输时间的准确机制。

小结

我们已经看到了如何以两种方法使用系统调用跟踪。在第一个例子中我们可以通过查看应用程序调用的系统调用来发现应用程序中的错误。在第二个例子中，我们感兴趣的不是调用本身，而是调用完成时花费的时间。

正如我们在以前讨论其他工具时看到的那样，经常是多个工具收集的信息才能使我们理解应用程序中发生的异常。系统调用工具如 ktrace、truss 和 strace 正是网络程序员在理解应用程序操作时可以采用的多种武器。

技巧 40 创建和使用捕获 ICMP 消息的工具

有时能够监测 ICMP 消息是很有用的。我们可以使用 tcpdump 或一些其他的行监测程序达到这个目的，但是有时有一个比较小的程序也许会更好一些。使用 tcpdump 有性能和安全方面的隐患，而只监测 ICMP 消息的程序却没有这方面的问题。

首先，使用行监测程序如 tcpdump 通常意味着让网络接口处于乱模式。这将增加 CPU 的负担，这是因为不管该数据包是否发向运行行监测程序的主机，线缆上的每个 Ethernet 数据包将导致一个中断。

其次，许多单位限制甚至禁止使用行监测程序，这是因为有可能用它来窃取密码。因为读取 ICMP 消息不会承担这个安全风险，所以这对这些单位来说是可以接受的。

本节我们开发了一个工具，它允许我们监测 ICMP 消息，而且不存在使用行监测程序的种种缺点。该工具也为我们研究 raw 套接字提供了机会，我们在前面没有涉及到这方面的东西。

回忆一下技巧 33 就可以知道，ICMP 消息是在 IP 数据报中携带的。虽然 ICMP 消息的

内容根据它的类型而改变，但是我们关心的仅仅是图 4.22 中显示的 `icmp_type` 和 `icmp_code` 域，当然处理 ICMP 不可到达消息时除外。

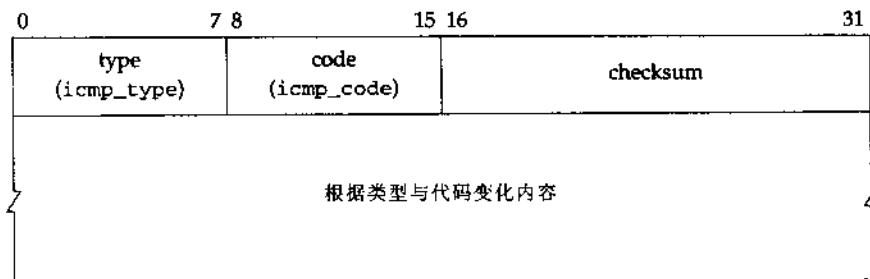


图 4.22 通常的 ICMP 消息

人们对 raw 套接字和它们的功能通常感到很困惑。raw 套接字不能用于截获 TCP 段或 UDP 数据报，这是因为它们从来不会传递给 raw 套接字。而且 raw 套接字也不接收每种类型的 ICMP 消息。例如，在 BSD 派生的系统中，因为 ICMP echo 消息、时间戳请求和地址掩码请求是完全由内核来处理的，所以 raw 套接字也不能获得它们。通常情况下，raw 套接字接收任何内核不知道协议类型的 IP 数据报，大多数 ICMP 消息和所有的 IGMP 消息属于此类。

读取 ICMP 消息

我们从工具的包含文件和 `main` 函数开始（图 4.23）。

-icmp.c

```

1 #include <sys/types.h>
2 #include <netinet/in_systm.h>
3 #include <netinet/in.h>
4 #include <netinet/ip.h>
5 #include <netinet/ip_icmp.h>
6 #include <netinet/udp.h>
7 #include "etcpc.h"

8 int main( int argc, char **argv )
9 {
10     SOCKET s;
11     struct protoent *pp;
12     int rc;
13     char icmpdg[ 1024 ];

```

```

14 INIT();
15 pp = getprotobyname( "icmp" );
16 if ( pp == NULL )
17     error( 1, errno, "getprotobyname failed" );
18 s = socket( AF_INET, SOCK_RAW, pp->p_proto );
19 if ( !isvalidsock( s ) )
20     error( 1, errno, "socket failed" );

21 for ( ; ; )
22 {
23     rc = recvfrom( s, icmpdg, sizeof( icmpdg ), 0,
24                 NULL, NULL );
25     if ( rc < 0 )
26         error( 1, errno, "recvfrom failed" );
27     print_dg( icmpdg, rc );
28 }
29 }
```

-icmp.c

图 4.23 icmp 的 main 函数

✓ 打开 raw 套接字

15~20 因为我们使用的 raw 套接字，所以必须指定感兴趣的协议。getprotobyname 调用返回一个包含 ICMP 协议号的结构。注意程序指定的是 SOCK_RAW 而不是以前的 SOCK_STREAM 或 SOCK_DGRAM。

✓ 事件循环

21~28 程序使用 recvfrom 读取每个 IP 数据报。程序调用 print_dg 输出接收到的 ICMP 消息。

☒ 输出 ICMP 消息

下面我们考虑一下格式化和输出 ICMP 消息。这由图 4.25 中的 print_dg 函数来完成。传递给 print_dg 的缓冲区具有图 4.24 显示的结构。

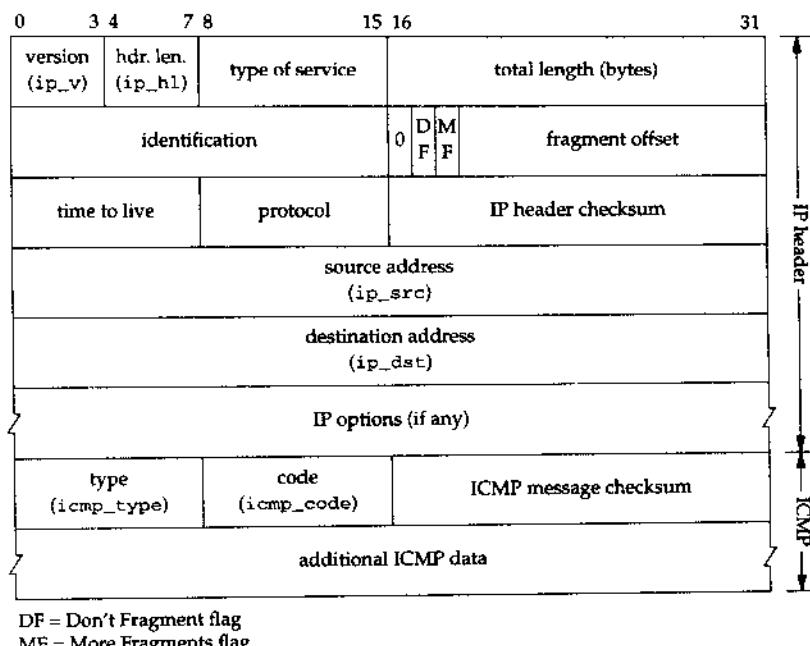


图 4.24 传递给 print_dg 的 ICMP 消息

如图 4.24 显示的那样，缓冲区包含了一个 IP 头，之后跟着 ICMP 消息本身。代码引用的域名称的字体为 Courier。

icmp.c

```

1 static void print_dg( char *dg, int len )
2 {
3     struct ip *ip;
4     struct icmp *icmp;
5     struct hostent *hp;
6     char *hname;
7     int hl;
8     static char *redirect_code[] =
9     {
10         "network", "host",
11         "type-of-service and network", "type-of-service and host"
12     };
13     static char *timexceed_code[] =
14     {
15         "transit", "reassembly"
16     };

```

```

17 static char *param_code[] =
18 {
19     "IP header bad", "Required option missing"
20 };

21 ip = ( struct ip * )dg;
22 if ( ip->ip_v != 4 )
23 {
24     error( 0, 0, "IP datagram not version 4\n" );
25     return;
26 }
27 hl = ip->ip_hl << 2;           /* IP header length in bytes */
28 if ( len < hl + ICMP_MINLEN )
29 {
30     error( 0, 0, "short datagram (%d bytes) from %s\n",
31            len, inet_ntoa( ip->ip_src ) );
32     return;
33 }
34 hp = gethostbyaddr( ( char * )&ip->ip_src, 4, AF_INET );
35 if ( hp == NULL )
36     hname = "";
37 else
38     hname = hp->h_name;
39 icmp = ( struct icmp * )( dg + hl ); /* ICMP packet */
40 printf( "ICMP %s (%d) from %s (%s)\n",
41         get_type( icmp->icmp_type ),
42         icmp->icmp_type, hname, inet_ntoa( ip->ip_src ) );
43 if ( icmp->icmp_type == ICMP_UNREACH )
44     print_unreachable( icmp );
45 else if ( icmp->icmp_type == ICMP_REDIRECT )
46     printf( "\tRedirect for %s\n", icmp->icmp_code <= 3 ?
47             redirect_code[ icmp->icmp_code ] : "Illegal code" );
48 else if ( icmp->icmp_type == ICMP_TIMXCEED )
49     printf( "\tTTL == 0 during %s\n", icmp->icmp_code <= 1 ?
50             timexceed_code[ icmp->icmp_code ] : "Illegal code" );
51 else if ( icmp->icmp_type == ICMP_PARAMPROB )

```

```

52     printf( "\t%s\n", icmp->icmp_code <= 1 ?
53         param_code[ icmp->icmp_code ] : "Illegal code" );
54 }

```

———icmp.c

图 4.25 print_dg 函数

获得 IP 头的指针并检查数据包的有效性

- 21 程序首先设置 ip (一个 struct ip 指针) 到刚才读取的数据报。
- 22~26 ip_v 域是该 IP 数据报的版本。如果这不是一个 IPv4 数据报，程序就输出错误消息并返回。
- 27~33 ip_hl 域在 32 位字中包含 IP 报头的长度，程序将该数值乘以 4 获得字节的长度并保存结果到 hl 中。接下来程序检查数据报确认 ICMP 消息至少为最小合法长度。

获得发送者的主机名称

- 34~38 程序使用 ICMP 消息的源地址查找发送者的主机名称，如果 gethostbyaddr 返回 NULL，就设置 hname 为空字符串；否则就设置它为主机名。

越过 IP 头并输出发送者和类型

- 39~42 程序设置 struct icmp 指针 icmp 为 IP 头之后的第一个字节。程序使用该指针获得 ICMP 消息类型 (icmp_type) 并输出类型和发送者的地址和主机名称。程序调用 get_type 来获得 ICMP 类型的 ASCII 表示。get_type 函数在图 4.26 中显示。

输出类型有关的信息

- 43~44 如果这是 ICMP 不可到达消息，就调用 print_unreachable 函数输出更多的信息，该函数在后面的图 4.29 中显示。

- 45~47 如果这是重定向消息，程序就从 icmp_code 域获得重定向类型并打印它。
- 48~50 如果这是超时消息，程序通过检查 icmp_code 域来决定消息是否在传输或重组时超时，并输出结果。

- 51~53 如果该 ICMP 消息报告参数错误，程序从 icmp_code 域获得问题的类型并输出它。

get_type 函数很直接。程序仅仅检测类型代码是否合法并返回对应字符串的指针（图 4.26）。

———icmp.c

```

1 static char *get_type( unsigned icmptype )
2 {
3     static char *type[] =
4     {

```

```

5     "Echo Reply",           /* 0 */
6     "ICMP Type 1",         /* 1 */
7     "ICMP Type 2",         /* 2 */
8     "Destination Unreachable", /* 3 */
9     "Source Quench",       /* 4 */
10    "Redirect",            /* 5 */
11    "ICMP Type 6",          /* 6 */
12    "ICMP Type 7",          /* 7 */
13    "Echo Request",        /* 8 */
14    "Router Advertisement", /* 9 */
15    "Router Solicitation", /* 10 */
16    "Time Exceeded",        /* 11 */
17    "Parameter Problem",   /* 12 */
18    "Timestamp Request",   /* 13 */
19    "Timestamp Reply",      /* 14 */
20    "Information Request",  /* 15 */
21    "Information Reply",    /* 16 */
22    "Address Mask Request", /* 17 */
23    "Address Mask Reply"    /* 18 */
24  );
25 if ( icmp_type < ( sizeof( type ) / sizeof( type[ 0 ] ) ) )
26   return type[ icmp_type ];
27 return "UNKNOWN";
28 }

```

icmp.c

图 4.26 get_type 函数

最后一个函数是 `print_unreachable`, ICMP 不可到达消息返回 IP 报头和导致产生不可到达错误的消息的 IP 数据报的开始 8 个字节, 这可以使我们找出发送者和不可递交消息接收者的地址和端口号。

从 raw 套接字读取的 IP 数据报的 ICMP 不可到达消息的结构如图 4.27 显示。`pring_dg` 已经处理的部分用阴影部分覆盖, 它不会传递给 `print_unreachable` 函数。`print_unreachable` 输入变量 `icmp` 和本地变量 `ip` 和 `udp` 也显示在图中。



图 4.27 ICMP 不可到达消息

`print_unreachable` 函数从报头和包含的 IP 数据报的开始 8 个字节中抽取信息。虽然我们把这 8 个字节标注为 UDP 报头，但是它们也可以是 TCP 报头的开始 8 个字节；端口号在两个报头中处于相同的位置。UDP 报头的格式在图 4.28 中显示。

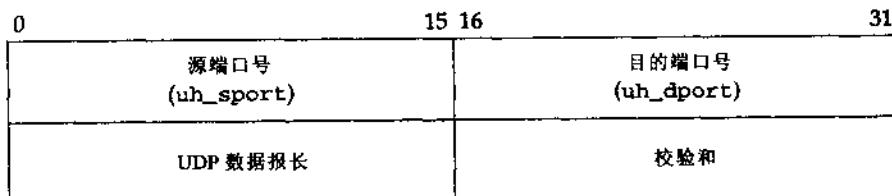


图 4.28 UDP 报头

代码在图 4.29 中显示

icmp.c

```

1 static void print_unreachable( struct icmp *icmp )
2 {
3     struct ip *ip;
4     struct udphdr *udp;
5     char laddr[ 15 + 1 ];
6     static char *unreach[] =
7     {
8         "Network unreachable",           /* 0 */
9         "Host unreachable",            /* 1 */
10        "Protocol unreachable",       /* 2 */
11        "Port unreachable",          /* 3 */
12        "Fragmentation needed, DF bit set", /* 4 */
13        "Source route failed",        /* 5 */
14        "Destination network unknown", /* 6 */
15        "Destination host unknown",   /* 7 */
16        "Source host isolated",      /* 8 */
17        "Dest. network admin. prohibited", /* 9 */
18        "Dest. host admin. prohibited", /* 10 */
19        "Network unreachable for TOS", /* 11 */
20        "Host unreachable for TOS",   /* 12 */
21        "Communication admin. prohibited", /* 13 */
22        "Host precedence violation",  /* 14 */
23        "Precedence cutoff in effect" /* 15 */
24    };

```

```

25 ip = ( struct ip * )( ( char * )icmp + 8 );
26 udp = ( struct udphdr * )( ( char * )ip + ( ip->ip_hl << 2 ) );
27 strcpy( laddr, inet_ntoa( ip->ip_src ) );
28 printf( "\t%s\n\tSrc: %s.%d, Dest: %s.%d\n",
29         icmp->icmp_code < ( sizeof( unreach ) /
30             sizeof( unreach[ 0 ] ) )?
31             unreach[ icmp->icmp_code ] : "Illegal code",
32             laddr, ntohs( udp->uh_sport ),
33             inet_ntoa( ip->ip_dst ), ntohs( udp->uh_dport ) );
34 }

```

icmp.c

图 4.29 print_unreachable 函数

设置指针并获得源地址

25~26 程序以分别设置 ip 和 udp 指针到 IP 报头和 IP 数据报有效负载的开始 8 个字节开头。

27 程序从 IP 报头中拷贝源地址到本地变量 laddr。

输出地址、端口号和类型

28~33 程序输出源地址和目的地址以及端口号，也输出不可到达消息的类型。

作为 icmp 用法的一个例子，下面 traceroute（请参阅技巧 35）产生的最后几个 ICMP 消息：

```
traceroute -q 1 netcom4.netcom.com
```

-q 1 选项告诉 traceroute 只发送一个探询而不是默认的三次。

```

ICMP Time Exceeded (11) from h1-0.mig-f1-gw1.icg.net (165.236.144.110)
    TTL == 0 during transit
ICMP Time Exceeded (11) from s10-0-0.dfw-tx-gw1.icg.net (165.236.32.74)
    TTL == 0 during transit
ICMP Time Exceeded (11) from dfw-tx-gw2.icg.net (163.179.1.133)
    TTL == 0 during transit
ICMP Destination Unreachable (3) from netcom4.netcom.com (199.183.9.104)
    Port unreachable
    Src: 205.184.142.71.45935, Dest: 199.183.9.104.33441

```

当然，我们一般不会运行 `icmp` 来监测 `traceroute` 的运行，但是它在解决连接问题时是很有用的。

» 小结

本节我们开发了一个捕获并应答 ICMP 消息的工具。这个工具可以帮助我们诊断网络和路由问题。

在开发 `icmp` 时，我们探讨了 `raw` 套接字的用法并分析了 IP 和 UDP 数据报的报头格式以及 ICMP 消息的报头格式。

技巧 41 读 Stevens 的书

网络编程有关的新闻组中经常碰到的问题是：“我们应当读什么书来学习 TCP/IP 的知识？”绝大多数的回答都提到了 Rich Stevens 的书籍。

我们已经在本书中多次参考了 Stevens 的书，但是现在我们希望更进一步地看看他们。对于网络程序员来说，Stevens 编写了两本书：`TCP/IP Illustrated` 系列共三卷和 `UNIX Network Programming` 系列共两卷。这两个系列书的焦点和目的是不同的，所以我们分别讨论它。

» 《TCP/IP Illustrated》系列

正如书名显示的那样，《`TCP/IP Illustrated`》对普通的 TCP/IP 协议和程序是如何工作进行了直接的阐述。我们在技巧 13 中已经提到了，使用的基本工具是 `tcpdump`。通过编写和运行小测试程序，然后用 `tcpdump` 观察网络流量的结果，Stevens 引导我们加深了对协议实际是如何工作的理解和评估。

通过使用不同的系统，Stevens 说明了同一协议的不同具体实现导致细微的不同操作。更为重要的是，我们也可以学会如何设计、运行和解释我们自己的实验，可以回答在自己工作中遇到的问题。

因为该系列的每一卷以不同的方式与 TCP/IP 协议匹配，所以对各卷作简要的介绍是很有用的。

✓ 第一卷：协议

该卷讨论了经典 TCP/IP 的各个协议和它们相互之间的关系。该书以链路层协议如 Ethernet、SLIP 和 PPP 开始。之后的讨论转移到 ARP 和 Reverse Address Resolution Protocol (RARP) 以及它们是如何跟链路层和 IP 层紧密联系的。

本卷有几章主要讲述 IP 协议以及它跟 ICMP 和路由的关系，也讨论了在 IP 层操作的应用程序如 `ping` 和 `traceroute`。

接下来讨论了 UDP 和相关的问题如广播和 IGMP。主要基于 UDP 的协议如 DNS、Trivial

File Transfer Protocol (TFTP) 和 Bootstrap Protocol (BOOTP) 也在本卷涉及到了。

TCP 和它的操作花了八章的篇幅。之后各章介绍了通常的 TCP 应用程序如 telnet、rlogin、FTP、SMTP (email) 和 NFS。

读完本卷的人对经典的 TCP/IP 协议和基于它们的协议有一个很好的了解。

第二卷：具体实现

第二卷是由 Gary Wright 编写的，几乎是 4.4BSD 网络代码各行的注释。因为 BSD 网络代码广泛地考虑被当作参考具体实现，该卷对于那些想知道和理解基本的 TCP/IP 协议是如何实现的人来说是必不可少的。

本书涉及了几个链路层协议 (Ethernet、SLIP 和 loopback 接口)、IP 协议、路由、ICMP、IGMP 和广播、套接字层、UDP、TCP 和其他相关领域的具体实现。因为该书中给出了实际的代码，所以读者可以获得实现重大网络系统必然涉及到的权衡和问题。

第三卷：实现事务、HTTP、NNTP 和 UNIX 域名协议的 TCP

第三卷是第一卷和第二卷的继续。它以 T/TCP 的描述以及它是如何操作的开始。描述的风格和第一卷一模一样。接下来给出了 T/TCP 的具体实现，其风格和第二卷相像。

第二部分，讲述了两个十分流行的应用层协议 Hypertext Transfer Protocol (HTTP) 和 Network News Transfer Protocol (NNTP)。这两个协议各自是 World Wide Web 和 Usenet News Groups 的基础。

最后，Stevens 讲述了 UNIX 域套接字和它们的具体实现。这其实是第二卷的继续，但是因为篇幅的限制在第二卷中省略掉了。

《UNIX Network Programming》系列

《UNIX Network Programming》从应用程序员的角度接近 TCP/IP。本书的焦点不是 TCP/IP 本身，而是如何使用它来编写网络应用程序。

第一卷：网络 API：套接字和 XTI

每个要实际编程的网络程序员都应当拥有这本书。该卷详细地讲述了使用套接字和 XTI API 进行 TCP/IP 编程。除了通常的有关客户端-服务器编程部分外，Stevens 也涉及了广播、路由套接字、非阻塞 I/O、IPv6 和它跟 IPv4 应用程序的秒度交互、raw 套接字、链路层的编程以及 UNIX 域套接字。

本书包括一个特别有用的章节，它比较了各种客户端-服务器设计的模式。后面的附录也涉及了实际的网络和调试技术。

第二卷：进程间通信

该卷详尽地介绍了 IPC。除了传统的 UNIX 管道和 FIFO、SysV 消息队列、信号灯和共享内存之外，该书更为现代的 POSIX IPC 方法。

本书很好地介绍了 POSIX 线程以及它是如何用作如互斥信号灯、条件变量和读写锁定等同步对象的。对于那些想要知道这些东西是如何工作的人来说，Stevens 给出了几个同步原语和 POSIX 消息队列的具体实现。

该书以 RPC 和 Solaris Doors 结束。

原来计划要出版讲述应用程序的第三卷，但是 Stevens 不幸在完成之前逝世了。计划该卷的材料可以在第一版的《UNIX Network Programming》[Stevens 1990]中获得。

技巧 42 阅读代码

网络编程新手经常问有经验的同事如何才能学会他们的技术和为什么老手们都知道似乎是多得令人吃惊的材料。当然，我们可以以多种方法了解和理解这些东西，但是其中一个最有价值而且被大多数人采用的方法，就是阅读熟练程序员的代码。

我们很幸运，处于萌芽的开源运动使阅读代码比以往变得更容易。阅读和研究一些可获得的高质量代码有两个不同的好处。

(1) 很明显，我们可以知道熟练程序员们如何靠近和解决特定的问题。我们可以采用它们的技术并把它们应用到我们自己的问题上，修改或改编它们。当我们这么做的时候，该技术发展了并成为我们自己的技术。终究有一天，当其他的人阅读我们的代码时，会对我们发明这么聪明的技术感到吃惊。

(2) 不是很明显但是在某些方面更重要的是我们知道没有魔术在里面。网络编程新手有时认为操作系统代码、或者实现网络协议的代码很深而且是不可了解的——它是巫师施魔法产生的，常人不可触及它。通过阅读这些代码，可以知道它们仅仅是优秀的（而且通常也是标准的）工程，我们也可以做到。

总之，通过阅读代码我们可以学会很深和神秘的东西只不过仅仅是标准技术的问题，我们知道这些技术是什么东西以及如何应用它们。阅读代码不容易，它需要集中精力，但是获得的好处比起付出的代价来说是很值得的。

有几个很好的源代码可供阅读，但是如果有讲述形式的帮助时会变得更容易一些，所以先介绍一些书籍。Lions 的《A Commentary on the UNIX Operating System》[Lions 1977]，不久就成了人们的喜爱的书籍。最近，因为一些人包括 Dennis Ritchie 的努力以及目前 UNIX 基本代码的拥有者 SCO 的慷慨，该书向公众公开。

该书最初只对 UNIX 源代码许可证拥有者开放，但是复印的拷贝在秘密传播，对许多 UNIX 程序员来说这简直就是奖品。

该书给出了 UNIX 操作系统早期版本（版本 6）的源代码，除了支持分时 TTY 终端外，没有网络组件。但是研究这些代码仍然有价值，即使我们对 UNIX 没有兴趣，也可以把它当作编程大师编写的优秀工程软件来研究。

包含源代码的有关操作系统的另一本相当不错的书是《Operating Systems: Design and

Implementation》[Tanenbaum and Woodhull 1997]。这本书讲述了 MINIX 操作系统。虽然这本书中没有讲述网络代码，但是在随书 CDROM 中可以获得它。

对于那些对网络有兴趣的人来说，《TCP/IP Illustrated, Volume 2》[Wright and Stevens 1995]是一本很不错的书。我们已经在技巧 41 中讨论了这本书，并且提到了对 TCP/IP 具体实现和总体网络技术有兴趣的人应当很好地研究它。

因为《TCP/IP Illustrated, Volume 2》讨论了当前几个开放源代码系统（FreeBSD, OpenBSD, NetBSD）使用的 BSD 网络代码，所以它为那些想实验这些代码的人提供了极好的实验手册。最初的 4.4BSD Lite 代码可以从 Walnut Creek CDROM FTP 服务器获得（<ftp://ftp.cdrom.com/pub/4.4BSD-Lite>）。

《Internetworking with TCP/IP Volume II》[Comer and Stevens 1999]给出了另一个 TCP/IP 栈。对比《TCP/IP Illustrated, Volume 2》来说，该书提供了代码是如何工作的完整解释。代码本身可以下载并使用。

还有许多好的源代码可以阅读，其中大多数没有附带手册来减轻阅读负担。开放代码 UNIX/Linux 项目是不错的地方。所有这些工程都可以从 CDROM 或通过 FTP 获得源代码。

Free Software Foundations 的 GNU 项目有大多数的标准 UNIX 实用程序的重新实现，这也是值得研究的另一个有些代码。

有关这些项目的信息可以在下面的主页上获得：

- FreeBSD 的主页

[<http://www.freebsd.org>](http://www.freebsd.org)

- GNU 项目

[<http://www.gnu.org>](http://www.gnu.org)

- Linux Kernel Archives

[<http://www.kernel.org>](http://www.kernel.org)

- NetBSD 的主页

[<http://www.netbsd.org>](http://www.netbsd.org)

- OpenBSD 的主页

[<http://www.openbsd.org>](http://www.openbsd.org)

所有这些源代码都有丰富的与网络有关的代码，即使对 UNIX 不感兴趣的人也值得一读。

小结

学习如何进行网络编程（或其他类型编程）的一个最好也是最令人愉快的方法是阅读那些已经掌握了这些材料的人编写的代码。到目前为止，获得操作系统的源代码仍然是十分困难的事情。随着开源运动的出现，这种情况很快会改变。几个 TCP/IP 栈以及相关实用程序（telnet、FTP、inetd 等等）的源代码可以从 BSD 和 Linux 操作系统项目那里获得。我们已经讨论了一些源代码，而且越来越多的源代码可以在 Web 上获得。

我们提到的四本书的特别值得一看，因为它们对代码提供了注释。

技巧 43 访问 RFC Editor 主页

我们前面已经提到 TCP/IP 协议集的“规范”和相关 Internet 体系结构问题包含在一系列的称作 Requests for Comments (RFC) 的文档中。实际上，RFC 可以追溯到 1969 年，不仅仅是一套协议规范。它们也许应当最好看作是一套讨论计算机通信和网络不同方面的工
作论文。并不是所有的 RFC 都处理技术问题，其中一些只是奇异的观察、拙劣的模仿、或者
是诗歌，还有的是另类观点。到 1999 年底，已经分配了 2700 多个 RFC 号，虽然其中一
些 RFC 从来都没有发布过。

虽然并不是每个 RFC 都指定了一个 Internet 标准，但是每个“Internet 标准”都是以 RFC 的形式发布的。RFC 子系列的成员都给了一个额外的格式为“STDxxxx”的标签。那些将要成为 Internet 标准的 RFC 的当前列表和状态都列在 STD0001 中。

然而，我们不能说不在 STD0001 中的 RFC 就没有技术价值。一些 RFC 描述的思想或建议仍在发展和研究之中。其他的一些 RFC 仅仅是情报，或者报告了 IETF 支持的工作组的思想。

获得 RFC

获得一份 RFC 的拷贝有多种方法，最简单的方法是访问 RFC Editor 的主页<<http://www.rfc-editor.org>>。该 Web 页提供了基于表格的下载功能，它使检索变得很容易。如果不知道 RFC 号，可以通过关键字检索来定位到适当的 RFC。该主页也提供相同形式 RFC 的 STD、FYI (For Your Information) 和 BCP (Best Current Practices) 子系列。

通过 FTP 站点 <ftp://ftp.isi.edu/in-notes/> 下也可以获得 RFC。从其他几个 FTP 存档站点也可以获得 RFC。

不能访问 Web 或 FTP 的读者可以通过电子邮件获取 RFC。通过发送消息体为

```
help: ways_to_get_rfcs
```

的电子邮件可以获得邮件订阅 RFC 的完整指令和 FTP 站点的列表。

对于任何一种方法来说，要获取的第一个东西就是 RFC 的当前索引（rfc-index.txt）。RFC 一旦发表了，就永远不能改变，因此 RFC 中的信息只可以通过发布一个替代 RFC 来修改。对于每个 RFC，索引说明了是否有一个替代 RFC，如果有的话，是哪一个。索引也在一个条目中说明了更新但没有取代该 RFC 的任何其他 RFC。

最后，从不同的发布商的 CDROM 中也可以获得 RFC。Walnut Creek CDROM (<http://www.cdrom.com/>) 和 InfoMagic (<http://www.infomagic.com/>) 都提供了包含 RFC 和其他有关 Internet 的文档的 CDROM。当然，不能说这些 CDROM 包含所有的 RFC 集合，但是因为 RFC 本身不能改变，所以它们可能过时，也就不会包括在最新的 RFC 中。

技巧 44 经常访问新闻组

Internet 上获得网络编程的建议和信息的最有价值的地方是 Usenet 新闻组。网上很多形形色色的新闻组，它们覆盖从 cabling (comp.dcom.cabling) 到 Network Time Protocol (comp.protocols.time.ntp) 的各个方面的网络知识。

涉及 TCP/IP 问题和编程的一个优秀的新闻组是 comp.protocols.tcp-ip。每天花些时间阅读新闻组中的文章可以获得各种各样的有用信息、技巧和技术。这些主题从如何连接 Windows 机器到 Internet 到有关 TCP/IP 协议、它们的具体实现和操作的最好技术应有尽有。

一开始的时候，网络新闻组的数目可能会吓倒你，所以最好从 comp.protocols.tcp-ip 开始，或者从操作系统有关的组如 comp.os.linux.networking 或 comp.os.ms-windows.programmer.tools.winsock 开始。我们在这些新闻组里看到的文章也会推荐其他可能引起我们兴趣或有用的更专业的新闻组。

令人感到荒谬的是，从新闻组中学东西的一个最好的方法是回答问题。通过利用我们在某一个方面的专门知识，把它们组织好解释给其他人，可以获得对知识本身更深的理解。写一个 50 到 100 字的回答文章可以为我们赢得更多的知识。

Usenet Info Center (<<http://matblab.unc.edu/usenet-l/>>) 上有 Usenet 新闻组的详细介绍。该站点包含 Usenet 的历史、使用和实践的许多文章，也有大多数新闻组的总结，包括每天的平均消息数、平均读者、管理员（如果有的话）、它在什么地方存档（如果有的话）以及联系每个组的 FAQ 指针。

Usenet Info Center 也有搜索功能帮助查找特殊主题的新闻组。这是另一个有价值的资源，我们可以使用它来定位我们感兴趣的讨论主题。

前面已经建议了，许多新闻组都有 FAQ，发表问题之前阅读它是很有帮助的。如果我们问了一个 FAQ 中已经存在的问题，就有可能引用到上面去，而且还可能因为在发表文章之前像因为懒惰而没有做家庭作业一样受到别人的斥责。

其他有关新闻组的资源

在离开新闻组主题之前，我们应当提一下两个其他的和它们有关的十分有价值的资源。第一个是 Deja News (<<http://www.deja.com>>)。

1999 年 5 月，Deja 新闻组从 Deja 新闻改为 Deja.com。主办者说，这是因为他们已经熟悉称它为“Deja”，而且也因为他们希望扩展他们的焦点和服务范围。在本节中，我们仅提及原来的 Usenet 新闻组的文档和检索服务。

该站点把将近 45000 个讨论论坛的消息归档，内容包括 Usenet 新闻组和他们自己的 Deja 社区讨论组。Deja.com 报道几乎三分之二的消息都是来自 Usenet 新闻组。到 1999 年底，Deja.com 已经收入了 1995 年 5 月之后的所有消息。

Deja 的 Power Search 功能允许用户搜索一个特定的新闻组、新闻组的分类，甚至通过关键字、主题、作者和时间范围进行搜索。当我们需要解决一个特定的问题或获得特定问题的答案时，它提供强有力的搜索工具。

第二个有价值的资源是 Uri Raz 的 TCP/IP Resource List，每隔两周发布 comp.protocols.tcp-ip 和专门新闻组上的内容。当我们希望找出特定的信息或更广阔、更通常的 TCP/IP 以及它的编程 API 的概览时，该列表是一个不错的起始点。

该列表提供的主题包括有关 TCP/IP 和网络的书籍、在线资源如 IETF 页面和哪里找到 FAQ、在线书籍和杂志、TCP/IP 教程、有关 IPv6 的信息源、许多流行网络书籍的主页、出版商的主页、GNU 和开放资源 OS 主页、搜索引擎和它们的用法以及有关网络的新闻组。

从下面的 Web 可以获得最新版本的列表：

```
<http://www.private.org.il/tcpip\_r1.html>
<http://www.best.com/~mphunter/tcpip\_resource.html>
```

或者通过 FTP 获得：

```
<ftp://rtfm.mit.edu/pub/usenet-by-group/news.answers/internet/tcp-ip/
resource-list>
<ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/comp/protocol/tcp-ip/TCP\_IP\_Resources\_List.html>
```

TCP/IP 资源列表的作者经常更新，这是使它特别有价值的原因之一。更新很重要是因为 Web 链接特别容易变化并且很快变陈旧。

附录 A

各种 UNIX 代码

etcp.h 头文件

几乎所有的示例程序都是以包含 `etcp.h` 头文件开始的（图 A.1）。该文件包含其他所需要的头文件，包括 `skel.h`（图 2.14）和其他一些定义、类型定义和原型。

etcp.h

```
1 #ifndef __ETCP_H__
2 #define __ETCP_H__

3 /* Include standard headers */

4 #include <errno.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <stdio.h>
8 #include <stdarg.h>
9 #include <string.h>
10 #include <netdb.h>
11 #include <signal.h>
12 #include <fcntl.h>
13 #include <sys/socket.h>
14 #include <sys/wait.h>
15 #include <sys/time.h>
16 #include <sys/resource.h>
17 #include <sys/stat.h>
```

```

18 #include <netinet/in.h>
19 #include <arpa/inet.h>
20 #include "skel.h"

21 #define TRUE          1
22 #define FALSE         0
23 #define NLISTEN        5      /* max waiting connections */
24 #define NSMB           5      /* number shared memory bufs */
25 #define SMBUFSZ        256    /* size of shared memory buf */

26 extern char *program_name;    /* for error */

27 #ifdef __SVR4
28 #define bzero(b,n)  memset( (b), 0, (n) )
29#endif

30 typedef void ( *tofunc_t )( void * );

31 void error( int, int, char*, ... );
32 int readn( int, char *, size_t );
33 int readvrec( int, char *, size_t );
34 int readcrlf( int, char *, size_t );
35 int readline( int, char *, size_t );
36 int tcp_server( char *, char * );
37 int tcp_client( char *, char * );
38 int udp_server( char *, char * );
39 int udp_client( char *, char *, struct sockaddr_in * );
40 int tselect( int, fd_set *, fd_set *, fd_set * );
41 unsigned int timeout( tofunc_t, void *, int );
42 void untimeout( unsigned int );
43 void init_smb( int );
44 void *smballoc( void );
45 void smbfree( void * );
46 void smbsend( SOCKET, void * );
47 void *smbrecv( SOCKET );
48#endif /* __ETCP_H__ */

```

etcp.h

附图 A.1 etcp.h 头文件

daemon 函数

我们在 `tcpmux` 中使用的 `daemon` 函数是一个标准的 BSD 库函数。对于 SVR4 系统，我们提供附图 A.2 中的版本。

library/daemon.c

```

1 int daemon( int nocd, int noclose )
2 {
3     struct rlimit rlim;
4     pid_t pid;
5     int i;
6
6     umask( 0 );      /* clear file creation mask */
7
7     /* Get max files that can be open */
8
8     if ( getrlimit( RLIMIT_NOFILE, &rlim ) < 0 )
9         error( 1, errno, "getrlimit failed" );
10
10    /* Become session leader, losing controlling terminal */
11
11    pid = fork();
12    if ( pid < 0 )
13        return -1;
14    if ( pid != 0 )
15        exit( 0 );
16    setsid();
17
17    /* ... and make sure we don't acquire another */
18
18    signal( SIGHUP, SIG_IGN );
19    pid = fork();
20    if ( pid < 0 )
21        return -1;
22    if ( pid != 0 )
23        exit( 0 );

```

```

24 /* Change to root directory unless asked not to */
25 if ( !nocd )
26     chdir( "/" );
27 /*
28 * Unless asked not to, close all files.
29 * Then dup stdin, stdout, and stderr
30 * onto /dev/null.
31 */
32 if ( !noclose )
33 {
34 #if 0 /* change to 1 to close all files */
35     if ( rlim.rlim_max == RLIM_INFINITY )
36         rlim.rlim_max = 1024;
37     for ( i = 0; i < rlim.rlim_max; i++ )
38         close( i );
39 #endif
40     i = open( "/dev/null", O_RDWR );
41     if ( i < 0 )
42         return -1;
43     dup2( i, 0 );
44     dup2( i, 1 );
45     dup2( i, 2 );
46     if ( i > 2 )
47         close( i );
48 }
49 return 0;
50 }

```

library/daemon.c

附图 A.2 daemon 函数

signal 函数

我们在书中已经讨论过，一些版本的 UNIX 通过使用不可靠的信号语义来实现 signal

函数。在这些系统上，我们应当使用 `sigaction` 函数来获得可靠的信号语义。为了便于移植，我们用 `sigaction` 重新实现了 `signal`（附图 A.3）。

```
library	signal.c
1  typedef void sighndlr_t( int );
2  sighndlr_t *signal( int sig, sighndlr_t *hndlr )
3  {
4      struct sigaction act;
5      struct sigaction xact;
6
7      act.sa_handler = hndlr;
8      act.sa_flags = 0;
9      sigemptyset( &act.sa_mask );
10     if ( sigaction( sig, &act, &xact ) < 0 )
11         return SIG_ERR;
12     return xact.sa_handler;
13 }
```

library signal.c

附图 A.3 `signal` 函数

附录 B

各种 Windows 代码

当在 Windows 下编译示例程序时，我们可以和 UNIX 一样使用头文件 `etc.h`（请参阅附图 A.1）。所有的与系统有关的信息都在 `skel.h` 头文件中。Windows 版本的头文件如附图 B.1 所示。

skel.h

```
1 #ifndef __SKEL_H__
2 #define __SKEL_H__

3 /* Winsock version */

4 #include <windows.h>
5 #include <winsock2.h>

6 struct timezone
7 {
8     long tz_minuteswest;
9     long tz_dsttime;
10};

11 typedef unsigned int u_int32_t;

12 #define EMSGSIZE      WSAEMSGSIZE
13 #define INIT()         init( argv );
14 #define EXIT(s)        do { WSACleanup(); exit( ( s ) ); } \
15                           while ( 0 )
16 #define CLOSE(s)      if ( closesocket( s ) ) \
17                           error( 1, errno, "close failed" )
18 #define errno         ( GetLastError() )
```

```

19 #define set_errno(e)      SetLastError( ( e ) )
20 #define isvalidsock(s)   ( ( s ) != SOCKET_ERROR )
21 #define bzero(b,n)        memset( ( b ), 0, ( n ) )
22 #define sleep(t)         Sleep( ( t ) * 1000 )
23 #define WINDOWS

24 #endif /* __SKEL_H__ */

```

skel.h

附图 B.1 Windows 版本的 skel.h

Window 兼容性函数

附图 B.2 显示了示例代码中使用的不同函数，这些函数在 Windows 中没有提供。

library/wincompat.c

```

1 #include <sys/timeb.h>
2 #include "etcp.h"
3 #include <winsock2.h>

4 #define MINBSDSOCKERR          ( WSAEWOULDBLOCK )
5 #define MAXBSDSOCKERR          ( MINBSDSOCKERR + \
6                               ( sizeof( bsdsocketerrs ) / \
7                                 sizeof( bsdsocketerrs[ 0 ] ) ) )

8 extern int sys_nerr;
9 extern char *sys_errlist[];
10 extern char *program_name;
11 static char *bsdsocketerrs[] =
12 {
13     "Resource temporarily unavailable",
14     "Operation now in progress",
15     "Operation already in progress",
16     "Socket operation on non-socket",
17     "Destination address required",
18     "Message too long",

```

```
19 "Protocol wrong type for socket",
20 "Bad protocol option",
21 "Protocol not supported",
22 "Socket type not supported",
23 "Operation not supported",
24 "Protocol family not supported",
25 "Address family not supported by protocol family",
26 "Address already in use",
27 "Can't assign requested address",
28 "Network is down",
29 "Network is unreachable",
30 "Network dropped connection on reset",
31 "Software caused connection abort",
32 "Connection reset by peer",
33 "No buffer space available",
34 "Socket is already connected",
35 "Socket is not connected",
36 "Cannot send after socket shutdown",
37 "Too many references: can't splice",
38 "Connection timed out",
39 "Connection refused",
40 "Too many levels of symbolic links",
41 "File name too long",
42 "Host is down",
43 "No route to host"
44 };

45 void init( char **argv )
46 {
47     WSADATA wsadata;

48     if( program_name = strrchr( argv[ 0 ], '\\' ) ) ?
49         program_name++ : ( program_name = argv[ 0 ] );
50     WSAStartup( MAKEWORD( 2, 2 ), &wsadata );
51 }
```

```
52 /* inet_aton - version of inet_aton for SVr4 and Windows */
53 int inet_aton( char *cp, struct in_addr *pin )
54 {
55     int rc;

56     rc = inet_addr( cp );
57     if ( rc == -1 && strcmp( cp, "255.255.255.255" ) )
58         return 0;
59     pin->s_addr = rc;
60     return 1;
61 }

62 /* gettimeofday - for tselect */
63 int gettimeofday( struct timeval *tvp, struct timezone *tzp )
64 {
65     struct _timeb tb;

66     _ftime( &tb );
67     if ( tvp )
68     {
69         tvp->tv_sec = tb.time;
70         tvp->tv_usec = tb.millitm * 1000;
71     }
72     if ( tzp )
73     {
74         tzp->tz_minuteswest = tb.timezone;
75         tzp->tz_dsttime = tb.dstflag;
76     }
77 }

78 /* strerror - version to include Winsock errors */
79 char *strerror( int err )
80 {
81     if ( err >= 0 && err < sys_nerr )
82         return sys_errlist[ err ];
83     else if ( err >= MINBSDSOCKERR && err < MAXBSDSOCKERR )
```

```
84     return bsdsocketerrs[ err - MINBSDSOCKERR ];
85 else if ( err == WSASYNSNOTREADY )
86     return "Network subsystem is unusable";
87 else if ( err == WSAVERNOTSUPPORTED )
88     return "This version of Winsock not supported";
89 else if ( err == WSANOTINITIALISED )
90     return "Winsock not initialized";
91 else
92     return "Unknown error";
93 }
```

library/wincompat.c

附图 B.2 Windows 兼容性函数

读者信息反馈表

亲爱的读者：

首先感谢您对本社电脑图书的关爱，为了更好地完善本社的服务体系，希望您能在百忙之中抽出一点时间，填妥以下表格。我们将审慎考虑您的宝贵意见并加以改进。同时，您也将有机会免费获得我社提供的精品图书。再次向您表示深深的谢意。

请寄往以下地址：100044 北京西城三里河路6号 中国电力出版社计算机编辑室

网 址：www.infopower.com.cn

E-mail：cepp01@mx.cei.gov.cn

姓名：_____ 性别：1.男 0.女 生日：_____年_____月_____日

文化程度：1.小学 2.中学 3.高中/中专 4.大学 5.硕士及以上

计算机应用水平：1.初学者 2.一般爱好者 3.电脑发烧友

4.计算机专业人员 5.计算机高级技术人员

职 业：1.系统/网络管理员 2.采购 3.销售 4.行政

5.财务/文秘 6.设计/研究 7.维修/质量控制/测试

8.生产/制造 9.教育培训 10.其他_____

工作单位：_____

地 址：_____

电 话：_____ 传 真：_____

邮 编：_____ E-mail：_____

书名：_____ 购买书店：_____

本书品质：5.非常好 4.很好 3.普通 2.不好 1.很差

·封面设计：5 4 3 2 1

·印刷：5 4 3 2 1

·作者功力：5 4 3 2 1

·编排：5 4 3 2 1

本书的内容，对您而言：

太难了 有点难 恰到好处 有点简单 太简单

如果您要买电脑图书，您最常到：

一般书店 计算机图书专卖店 电脑门市 新华书店 电脑图书展

邮购 网上订购 其他_____

您选择购买本书的原因为：(可复选)

品质优良 朋友建议 价格合理 书店推荐 报刊杂志推荐 作者 出版社

广告 促销优惠 附加价值（光盘或磁盘） 其他_____

您希望中国电力出版社将来为您出版哪一类型电脑图书？(可复选)

操作系统 数据库 网络/通信 程序设计 图像处理/多媒体设计

电脑硬件/接口开发 开源软件 办公自动化软件 儿童电脑 其他_____

您是否愿意收到中国电力出版社的计算机图书目录？ 是 否

其他建议：_____



系统网络管理



DNS—Windows NT 版

作者: Paul Albitz
书号: ISBN 7-5083-0455-1
定价: 39.00 元



Windows NT TCP/IP

网络管理
作者: Paul Albitz
书号: ISBN 7-5083-0439-x
定价: 49.00 元



Cisco 路由器管理

作者: Scott M.Ballew
书号: ISBN 7-5083-0196-x
定价: 40.00 元



UNIX 系统管理

作者: Frisch
书号: ISBN 7-5083-0458-6
定价: 89.00 元



Windows 2000 活动目录

作者: Alistair G.Lowe-Norris
书号: ISBN 7-5083-0480-2
定价: 79.00 元

本书作者是微软活动目录和核心开发小组成员

程序语言与编程



C++ 语言核心
作者: Gregory Satir
书号: ISBN 7-5083-0473-x
定价: 29.00 元



Visual Basic
Win32 API 编程
作者: Steven Roman
书号: ISBN 7-5083-0308-3
定价: 65.00 元



实用 C 语言编程
作者: Steve Oualline
书号: ISBN 7-5083-0308-3
定价: 49.00 元



Delphi 技术手册
作者: Ray Lischner
书号: ISBN 7-5083-0264-8
定价: 25.00 元



UNIX 操作系统
作者: Jerry Peek
书号: ISBN 7-5083-0264-8
定价: 25.00 元

网络设计与开发



优化 Web 性能
作者: Patrick Killelea
书号: ISBN 7-5083-0234-6
定价: 49.00 元



CSS 权威指南
作者: Eric A.Meyer
书号: ISBN 7-5083-0484-5
定价: 59.00 元



JavaScript 权威指南
作者: David Flanagan



ASP 技术手册
作者: A.Keyton Weissinger



Web 设计技术手册
作者: Jennifer Niederst
书号: ISBN 7-5083-0484-5
定价: 59.00 元

全球销量已逾 15 万册