



(第二版)

TCP/IP Sockets in C Second Edition

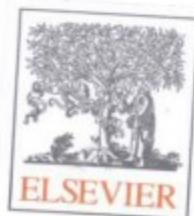
# TCP/IP Sockets 编程 (C语言实现)

Michael J. Donahoo Kenneth L. Calvert 著  
陈宗斌 等 译



计算机  
科学  
基础  
教程  
PDG

清华大学出版社



# TCP/IP Sockets in C Second Edition

## 内容简介

本书为开发成熟且功能强大的Web应用程序提供所需的知识和技巧。本书以教学指南的方式，帮助读者掌握在C语言环境下，用套接字实现客户-服务器项目开发的任务和技术。本书的本版次增加了对最新技术的介绍，如对IPv6的支持，以及更详细的编程策略等内容。

本书内容简明扼要、示例丰富，既可作为高等学校网络编程课程的教学参考书，也是网络开发人员进行应用程序开发的常备参考书。

## 作者介绍

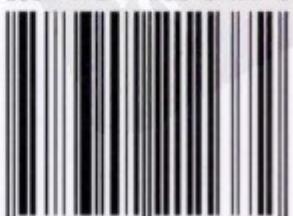
Michael J. Donahoo

Baylor大学副教授，为本科生和研究生讲授网络技术课程。他编写了多本关于各种语言的套接字编程图书，以及一本SQL方面的图书。

Kenneth L. Calvert

Kentucky大学教授，在该大学里，他主要讲授计算机网络系统课程，具有20多年的TCP/IP套接字编程经验。

ISBN 978-7-302-21137-2



9 787302 211372 >

定价：29.00元

第二版



TCP/IP Sockets in C Second Edition

# TCP/IP Sockets 编程 (C语言实现)

Michael J. Donahoo Kenneth L. Calvert 著  
陈宗斌 等 译



清华大学出版社  
北京

TCP/IP Sockets in C, Second Edition  
Michael J. Donahoo, Kenneth L. Calvert  
ISBN: 9780123745408

Copyright © 2009 by Elsevier. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.  
ISBN: 9789812724137

Copyright © 2009 by Elsevier(Singapore) Pet Ltd. All rights reserved.

Printed in China by Tsinghua University Press under special arrangement with Elsevier (singapore) Pte Ltd.. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由 Elsevier(Singapore) Pet Ltd. 授予清华大学出版社在中国大陆地区(不包括香港、澳门特别行政区以及台湾地区)发行与销售。未经许可之出口,视为违反著作权法,将受法律之制裁。

北京市版权局著作合同登记号 图字: 01-2009-4884 号

本书封面贴有 Elsevier 防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

#### 图书在版编目(CIP) 数据

TCP/IP Sockets 编程:C 语言实现: 第 2 版/(美)多纳霍(Donahoo M. J.), (美)卡尔弗特(Calvert K. L.)著;陈宗斌等译.—北京:清华大学出版社,2009.11

书名原文: TCP/IP Sockets in C, Second Edition

ISBN 978-7-302-21137-2

I. T… II. ①多… ②卡… ③陈… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2009)第 175766 号

责任编辑: 龙啟铭

责任校对: 徐俊伟

责任印制: 杨 艳

出版发行: 清华大学出版社

<http://www.tup.com.cn>

社 总 机: 010-62770175

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

地 址: 北京清华大学学研大厦 A 座

邮 编: 100084

邮 购: 010-62786544

印 刷 者: 清华大学印刷厂

装 订 者: 三河市李旗庄少明装订厂

经 销: 全国新华书店

开 本: 185×230 印 张: 12.25

字 数: 294 千字

版 次: 2009 年 11 月第 1 版

印 次: 2009 年 11 月第 1 次印刷

印 数: 1~3000

定 价: 29.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。  
联系电话: 010-62770177 转 3103 产品编号: 033748-01

# 译者序

目前，Internet 已经广泛渗透到人们的日常生活中，并且变得越来越重要。尽管现在有其他语言提供了对 Internet 的访问，人们对原始的基于 C 的 Berkeley Sockets 仍然有很浓厚的兴趣。对于那些想要设计和构建使用 Internet（即使用 TCP/IP）的分布式应用程序的人来说，Sockets API 仍然很重要。并且此接口已经证明非常健壮，足以支持网际协议的新版本（IPv6），现在几乎所有的公共计算平台都支持 IPv6。

在编写本书的第 2 版时，与第 1 版相比，考虑到了两个重要的方面。第一，有一些主题需要更深入地介绍，另外一些主题则需要扩展。第二个考虑是人们日益接受并使用 IPv6，它现在实质上受到所有当前的最终系统平台的支持。本书这一版在这两个方面都做了有针对性的介绍。

本书内容简明扼要、示例丰富，非常适合于学习涉及编程的计算机网络初级课程的学生（研究生和大学生）以及希望编写他们自己的程序以便通过 Internet 通信的程序员阅读。本书要求读者具有 C 和 UNIX 方面的基本编程技能和经验，熟悉一些 C 语言的概念，比如指针和类型强制转换，并且应该基本了解数据的二进制表示。

参加本书翻译的人员有：陈宗斌、王馨、陈红霞、张景友、易小丽、陈婷、管学岗、王新彦、金惠敏、张海峰、徐晔、戴锋、张德福等。

由于时间紧迫，加之译者水平有限，错误在所难免，恳请广大读者批评指正。

# 第 2 版的前言

在我们编写本书的第 1 版时，在关于联网的大学课程中包括编程方面的内容不是很常见。现在看起来这似乎令人难以置信，目前，Internet 变得对我们的世界如此重要，并且讲授实用编程和真实协议示例的好处被如此广泛地接受。尽管现在有其他语言提供了对 Internet 的访问，人们对原始的基于 C 的 Berkeley Sockets 仍然有很浓厚的兴趣。用于联网的 Sockets API（应用程序编程接口）是 20 世纪 80 年代在加州大学伯克利分校为 UNIX 的 BSD 版本开发的——这种 UNIX 版本是现在称为开源项目的最初几个示例之一。

Sockets API 和 Internet 在许多竞争性协议族——IPX、Appletalk、DECNet、OSI、SNA 以及 TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）——的世界中逐渐成长起来，并且 Sockets 被设计成支持所有这些协议。在我们编写本书第 1 版时，只有很少的几个协议族被广泛应用，今天这个数量甚至更少。然而，就像我们在本书第 1 版中所预计的，对于那些想要设计和构建使用 Internet（即使用 TCP/IP）的分布式应用程序的人来说，Sockets API 仍然很重要。并且此接口已经证明非常健壮，足以支持网际协议的新版本（IPv6），现在几乎所有的公共计算平台都支持 IPv6。

两个主要的考虑促使我们编写了本书的第 2 版。第一，依据我们自己的经验和其他人的反馈意见，我们发现一些主题需要更深入地介绍，另外一些主题则需要扩展。第二个考虑是人们日益接受并使用 IPv6，它现在实质上受到所有当前的最终系统平台的支持。在编写本书时，不可能使用 IPv6 与 Internet 上的大部分主机交换信息，但是有可能给其中的许多主机分配一个 IPv6 地址。尽管称 IPv6 将接管整个世界仍言之过早，但是开始编写应用程序以为之做好准备却正当其时。

## 相比第 1 版的变化

更新并相当大地扩展了大部分材料，添加了两章内容。相比第 1 版的主要变化如下。

- 涵盖了 IPv6。现在包括了三种代码：特定于 IPv4 的代码、特定于 IPv6 的代码和通

用代码。后面几章中的代码设计成用于双栈机器上的任何一种协议版本。

- 新增了关于用 C++ 进行套接字编程的额外一章内容（由 David B. Sturgill 撰稿）。PracticalSocket 库提供了基本套接字功能的包装器。这些允许教师给没有 C 编程背景的学生讲授套接字编程，方法是给他们提供一个库，然后逐一剖析各层知识。学生在理解了地址/端口和客户/服务器之后，可以立即开始进行开发。以后可以通过剖析包装器代码内部的工作原理，向他们展示套接字编程的细节。那些讲授涉及联网的主题（例如，操作系统）的教师可以使用库，并且只是有选择地剖析其内部的工作原理。
- 更深入地介绍了用于组织发送和接收消息的代码的数据表示问题和策略。根据我们的教学经验，发现学生极少理解在内存中如何实际地存储数据<sup>1</sup>，因此我们尝试对这个问题进行更多的讨论以作为补偿。与此同时，国际化的重要性也与日俱增，因此本书包括了对广泛字符和编码的基本介绍。
- 省略了参考一节的内容。组成 Sockets API 的大多数函数的描述被集合进前面几章中。不过，由于有如此之多的在线参考信息资源（包括“参考手册（man page）”）可用，我们选择省略了 API 的完整列表，以便于进行更多的代码演示。
- 突出了重要而细微的事实和警告。排版设备突出了重要的概念和信息，在第 1 版中可能遗漏了这些信息。

尽管本书涵盖的范围有所扩展，但是我们没有把所有的一切都纳入本书中（甚至那些要求包括的内容也是如此）；有些主题需要参阅更全面的教材（或者将在下一版中介绍），这样一些主题的例子有原始套接字以及利用 WinSock 编程。

## 本书读者对象

我们最初编写本书的目的是，当我们希望学生学习套接字编程时，可以给他们分发一些资料，使得我们不必占用宝贵的课堂时间来讲授它。自从本书第 1 版起的几年间，我们了解到对于一些主题学生需要许多帮助，而有些主题则不需要太多手把手的指导。我们还发现我们的图书至少会被那些希望了解主题的简要介绍的从业者赏识。因此，本书同时针对两类普通读者：学习涉及编程的计算机网络初级课程的学生（研究生和大学生），以及希望编写他们自己的程序以便通过 Internet 通信的从业者。对于学生，本书打算作为一种补充材料，而不是关于网络的主要教材。尽管本书第 2 版的篇幅比第 1 版长很多并且涵盖的范围也大得多，我们还是希望本书仍然被人们认为作为补充材料具有良好的价值。对于只想编写一些有用代码的从业者，应该把本书作为独立的导论——但是应该告诫这类读者的是，

---

<sup>1</sup> 我们推测这是由于在大学课程里 C++ 和 Java 的广泛应用，它们对程序员隐藏了这类细节。



本书将不会使他们成为专家。我们通过实践学习知识的哲学没有改变，我们采用的方法也没有改变，即我们只提供足以帮助读者开始自学的简明教程，而把全面的细节留给其他作者去完成。对于这两类读者对象，我们的目标是教给读者足够多的知识，以便他们可以开始试验并自学相关知识。

## 假定的背景

我们假定读者具有 C 和 UNIX 方面的基本编程技能和经验。期望你熟悉一些 C 语言的概念，比如指针和类型强制转换，并且应该基本了解数据的二进制表示。一些示例将被分成应该单独编译的文件；我们假定你可以处理这项任务。

下面是一个小测试：如果你可以设法想出下面的代码段用于做什么，那么在理解本书中的代码方面应该没有什么问题：

```
typedef struct {
    int a;
    short s[2];
} MSG;

MSG *mp, m = {4, 1, 0};
char *fp, *tp;
mp = (MSG *) malloc(sizeof(MSG));
for (fp = (char *)m.s, tp = (char *)mp->s; tp < (char *) (mp+1);)
    *tp++ = *fp++;
```

如果不理解这个代码段，不要绝望（我们的代码中没有如此费解的内容），但是你可能想查阅自己最喜爱的 C 编程书籍，以便搞清楚这里将发生什么事情。

你还应该熟悉进程/地址空间、命令行参数、程序终止以及常规文件输入和输出等 UNIX 概念。第 4 章和第 6 章中的材料假定读者掌握了 UNIX 的更高级一些的知识。事先掌握一些网络方面的概念（比如协议、地址、客户和服务器）将是有帮助的。

## 平台需求与可移植性

我们的介绍是基于 UNIX 的。在开发本书时，好几个人强烈要求同时包括进针对 Windows 以及 UNIX 的代码。由于多方面的原因我们不可能这样做，包括为本书设定的目标篇幅长度（和定价）。

对于那些只能访问 Windows 平台的人，请注意本书前面几章中的示例需要进行最低限度的修改以便使用 WinSock（必须在程序开头更改包括文件并添加一个设置调用，并在程

序末尾添加一个清理调用）。其他大多数示例还需要额外一些非常少的修改。不过，有些示例是如此依赖于 UNIX 编程模型，以至于把它们移植到 WinSock 是没有意义的。可以从本书的 Web 站点 ([www.elsevierdirect.com/companions/9780123745408](http://www.elsevierdirect.com/companions/9780123745408)) 上获取其他示例的为 WinSock 准备的版本，以及所需的代码修改的详细说明。另请注意：通过在用于 Windows 的 Cygwin UNIX 库程序包（可以在线获取它）下执行最少量的修改，就可以使几乎所有的示例代码工作。

对于本书第 2 版，我们采纳了 C99 语言标准。这个语言版本受到大多数编译器的支持，并且提供了如此之多的可以改进可读性的优点——包括行定界的注释、固定大小的整数类型，以及出现在块中任意位置的声明——以至于我们不能调整它，而只是使用它。

我们的代码利用了“Basic Socket Interface Extensions for IPv6”[6]。这些扩展当中有一个新的、不同的接口连接到名称系统。由于我们完全依赖于这个新的接口 (`getaddrinfo()`)，我们的通用代码可能不会在一些较老的平台上运行。不过，我们期望大多数现代系统将非常好地运行我们的代码。

本书中包括的示例程序全都在\*NIX 和 MacOS 中进行了测试（应该可以不加修改地编译和运行）。头 (.h) 文件的位置和依赖性不是非常标准，可能需要在你的系统上进行一些修改。套接字选项支持也因系统的不同而有广泛的不同；我们尽量关注那些受到最普遍支持的套接字选项。参考 API 文档，了解系统的特定细节（API 文档的含义是指你的系统的“参考手册”。要了解这方面的情况，可以输入“man man”，或者使用你最喜爱的 Web 搜索工具）。

要知道的是，尽管我们努力实现基本级别的健壮性，但是我们的代码示例的主要目标是用于教学，并且代码不具有生产质量。出于简洁性和清晰性考虑，我们牺牲了一些健壮性，尤其是在通用的服务器代码中（事实证明：编写可以在 IPv4 和 IPv6 协议配置的各种组合下工作的服务器同时最大化各种环境下成功的客户连接的可能性是很重要的）。

## 本书将不会使你成为一名专家

我们希望本书第 2 版将会被用作一种参考资源，甚至对那些已经相当了解套接字的读者也是如此。与第 1 版一样，我们在编写这一版时也学到了一些知识。但是要成为一名专家需要多年的经验，以及学习其他更全面的资源[3、16]。

本书第 1 章旨在给你提供刚好够用的总体知识，以便让你准备好编写代码。第 2 章说明了如何使用 IPv4 或 IPv6 编写 TCP 客户和服务器。第 3 章说明如何使客户和服务器使用网络的名称服务，还描述了如何使它们独立于 IP 版本。第 4 章介绍了 UDP (User Datagram Protocol, 用户数据报协议)。第 5 章和第 6 章提供了编写更多程序所需的背景知识，而第 7 章则把 Sockets 实现中的一些内容与 API 调用关联起来；这 3 章内容实质上是独立的，可

以任意顺序介绍它们。最后，第 8 章展示了一个 C++类库，它提供了对套接字功能的简化访问。

在全书中，某些语句将会高亮显示，比如：本书将不会使你成为一名专家！我们的目标是：使你注意那些细微而重要的事实和思想，在第一次阅读时可能会遗漏它们。页边距中的标记告诉你“注意”以粗体显示的任何内容。

## 致谢

正是由于许多人的贡献，本书才得以问世。除了在编写本书第 1 版时帮助过我们的那些人（Michel Barbeau、Steve Bernier、Arian Durresi、Gary Harkin、Ted Herman、Lee Hollaar、David Hutchison、Shunge Li、Paul Linton、Ivan Marsic、Willis Marti、Kihong Park、Dan Schmitt、Michael Scott、Robert Strader、Ben Wah 和 Ellen Zegura）之外，我们还要特别感谢 David B. Sturgill 和 Bobby Krupczak，David B. Sturgill 撰写了第 8 章中的代码和正文内容，Bobby Krupczak 则帮助审阅了本书第 2 版的草稿。最后，要感谢 Morgan Kaufmann/Elsevier 的各位工作人员——我们的编辑 Rick Adams、助理编辑 Maria Alonso 和项目经理 Melinda Ritchie，感谢你们的耐心、帮助，以及对我们的图书质量的关心。

## 反馈

我们非常有兴趣清除错误，这样可以改进将来版本/印刷的质量，因此如果你发现任何错误，请给我们中的任何一个人发送电子邮件。我们将在本书的 Web 页面上维护一份勘误表。

我们的电子邮件地址如下：

M.J.D.: jeff\_donahoo@baylor.edu

K.L.C.: calvert@netlab.uky.edu



# 目录

<b>第1章 简介 .....</b>	<b>1</b>
1.1 网络、分组和协议 .....	1
1.2 关于地址.....	3
1.2.1 记下 IP 地址.....	4
1.2.2 处理两个版本 .....	4
1.2.3 端口号 .....	5
1.2.4 特殊地址 .....	5
1.3 关于名称.....	6
1.4 客户与服务器 .....	7
1.5 什么是套接字 .....	8
练习题.....	9
<b>第2章 基本的 TCP 套接字.....</b>	<b>10</b>
2.1 IPv4 TCP 客户.....	10
2.2 IPv4 TCP 服务器.....	15
2.3 创建和销毁套接字 .....	20
2.4 指定地址.....	21
2.4.1 通用地址 .....	22
2.4.2 IPv4 地址.....	22
2.4.3 IPv6 地址.....	23
2.4.4 通用地址存储器 .....	23
2.4.5 二进制/字符串地址转换 .....	24
2.4.6 获取套接字的关联地址 .....	25
2.5 连接套接字.....	25

2.6 绑定到地址 .....	26
2.7 处理进入的连接 .....	27
2.8 通信 .....	28
2.9 使用 IPv6.....	29
练习题 .....	31
<b>第 3 章 关于名称和地址族 .....</b>	<b>32</b>
3.1 将名称映射到数字 .....	32
3.1.1 访问名称服务 .....	33
3.1.2 详细信息 .....	37
3.2 编写地址通用的代码 .....	38
3.2.1 通用的 TCP 客户 .....	39
3.2.2 通用的 TCP 服务器.....	42
3.2.3 IPv4 与 IPv6 之间互操作 .....	45
3.3 从数字获取名称 .....	46
练习题 .....	47
<b>第 4 章 使用 UDP 套接字.....</b>	<b>48</b>
4.1 UDP 客户 .....	48
4.2 UDP 服务器 .....	52
4.3 利用 UDP 套接字进行发送和接收 .....	54
4.4 连接 UDP 套接字 .....	56
练习题 .....	56
<b>第 5 章 发送和接收数据.....</b>	<b>58</b>
5.1 编码整数 .....	59
5.1.1 整数的大小 .....	59
5.1.2 字节排序 .....	61
5.1.3 符号性与符号扩展 .....	62
5.1.4 手工编码整数 .....	63
5.1.5 在流中包装 TCP 套接字.....	66
5.1.6 结构覆盖：对齐与填充 .....	68
5.1.7 字符串和文本 .....	71
5.1.8 位操作：编码布尔值 .....	73

5.2 构造、成帧和解析消息 .....	74
5.2.1 成帧 .....	80
5.2.2 基于文本的消息编码 .....	86
5.2.3 二进制消息编码 .....	88
5.2.4 综合应用 .....	91
5.3 小结 .....	91
练习题 .....	91
<b>第 6 章 超越基本的套接字编程 .....</b>	<b>93</b>
6.1 套接字选项 .....	93
6.2 信号 .....	95
6.3 非阻塞 I/O .....	100
6.3.1 非阻塞套接字 .....	100
6.3.2 异步 I/O .....	101
6.3.3 超时 .....	105
6.4 多任务处理 .....	109
6.4.1 每个客户一个进程 .....	110
6.4.2 每个客户一个线程 .....	115
6.4.3 受限的多任务处理 .....	119
6.5 多路复用 .....	120
6.6 多个接收者 .....	125
6.6.1 广播 .....	126
6.6.2 多播 .....	129
6.6.3 广播与多播 .....	133
练习题 .....	134
<b>第 7 章 揭密 .....</b>	<b>135</b>
7.1 缓冲和 TCP .....	137
7.2 死锁风险 .....	139
7.3 关于性能 .....	140
7.4 TCP 套接字的生存期 .....	141
7.4.1 连接 .....	141
7.4.2 关闭 TCP 连接 .....	145
7.5 解多路复用揭密 .....	149

练习题 .....	150
<b>第 8 章 用 C++ 进行套接字编程 .....</b>	<b>151</b>
8.1 PracticalSocket 库概述 .....	152
8.2 加 1 服务 .....	154
8.2.1 加 1 服务器 .....	154
8.2.2 加 1 客户 .....	156
8.2.3 运行服务器和客户 .....	157
练习题 .....	158
8.3 调查服务 .....	158
8.3.1 调查的支持函数 .....	159
8.3.2 调查服务器 .....	161
8.3.3 调查客户 .....	166
8.3.4 运行服务器和客户 .....	167
8.4 第二种样式的调查服务 .....	168
8.4.1 套接字地址支持 .....	168
8.4.2 套接字的 iostream 接口 .....	169
8.4.3 增强的调查服务器 .....	170
8.4.4 增强的调查客户 .....	175
8.4.5 管理客户 .....	176
8.4.6 运行服务器和客户 .....	177
练习题 .....	177
<b>参考文献 .....</b>	<b>179</b>

# 第 | 章

## 简介

今天，人们可以使用计算机打电话、看电视、给他们的朋友发送即时消息、与其他人玩游戏，以及购买可以想到的大多数物品——从歌曲到汽车。程序通过 Internet 通信的能力使得所有这一切都成为可能。很难说现在可以通过 Internet 访问多少台个人计算机，但是我们可以安全地说这个数量在快速增长；用不了多长时间就会达到数十亿。而且，每天都在开发新的应用程序。在日益增长的带宽和访问量的推动下，在可预见的将来 Internet 的影响将继续升级。

一个程序怎样通过网络与另一个程序通信？本书的目标是：在 C 编程语言的环境下，开始让你理解这个问题的答案。长时间以来，C 都是实现网络通信软件所选的语言。的确，称为套接字的 API（application programming interface，应用程序编程接口）最初就是用 C 开发的。

不过，在深入研究套接字的细节之前，值得从总体上简要探讨一下网络和协议，看看我们的代码将适合于应用在什么位置。我们在本书中的目标不是讲授网络和 TCP/IP 如何工作——有许多针对此目的的优秀教材可以阅读[1、3、10、15、17]，而是介绍一些基本的概念和术语。

### 1.1 网络、分组和协议

计算机网络由通过通信信道互连的机器组成。我们把这些机器称为主机（host）和路由器（router）。主机是运行应用程序（比如 Web 浏览器、IM 代理或文件共享程序）的计算机。主机上运行的应用程序是网络的真正“用户”。路由器也称为网关（gateway），这种机器的职责是把信息从一条通信信道中继或转发（forward）到另一条通信信道。它们可能会运行程序，但是通常不会运行应用程序。出于我们的目的，通信信道（communication channel）是把字节序列从一台主机传送到另一台主机的工具；它可能是有线（例如，以太网）、无线（例如，WiFi）或其他连接。

路由器很重要，这只是由于它并不实际用于把每一台主机直接连接到所有其他的主机，而是代之以将少数几台主机连接到一个路由器，再把该路由器连接到其他路由器，如此下去，从而构成网络。这种安排允许利用数量相对较少的通信信道连接每台机器；大多数主机只需要一条通信信道。不过，通过网络交换信息的程序不会直接与路由器交互，它们基本上感觉不到路由器的存在。

信息（information）的含义是指由程序构造和解释的字节序列。在计算机网络的环境中，这些字节序列一般称为分组（packet）。分组包含网络用于执行其任务的控制信息，有时也包括用户数据。一个例子是用于标识分组目的地的信息。路由器使用这种控制信息来查明如何转发每个分组。

协议（protocol）是关于由通信程序交换的分组及其含义的协定。协议说明如何构造分组——例如，目的地信息位于分组中的什么位置以及分组有多大——以及如何解释信息。协议通常设计成使用给定的能力解决特定的问题。例如，HTTP（HyperText Transfer Protocol，超文本传输协议）解决的是在存储或生成超文本对象的服务器之间传输这些对象的问题，Web 浏览器则使它们对用户可见并且有用。即时消息传递协议解决的是允许两个或更多的用户交换简短文本消息的问题。

实现一个有用的网络需要解决大量不同的问题。为了保持事情易于管理并且是模块化的，设计了不同的协议来解决不同的问题集。TCP/IP 就是这样一个解决方案的集合，有时称之为协议族（protocol suite）。它碰巧就是 Internet 中使用的协议族，但是它也可以在独立的专用网络中使用。今后，当我们谈论网络（network）时，指的就是使用 TCP/IP 协议族的任何网络。TCP/IP 协议族中的主要协议是 IP（Internet Protocol，网际协议）、TCP（Transmission Control Protocol，传输控制协议）和 UDP（User Datagram Protocol，用户数据报协议）。

事实证明：分层组织协议是有用的；TCP/IP 以及几乎所有其他的协议族都是这样组织的。图 1.1 显示了主机和路由器中的协议、应用程序与 Sockets API 之间的关系，以及数据从一个应用程序（使用 TCP）到另一个应用程序的流动。标记为 TCP 和 IP 的方框表示这些协议的实现。这样的实现通常驻留在主机的操作系统中。应用程序通过 Sockets API（以

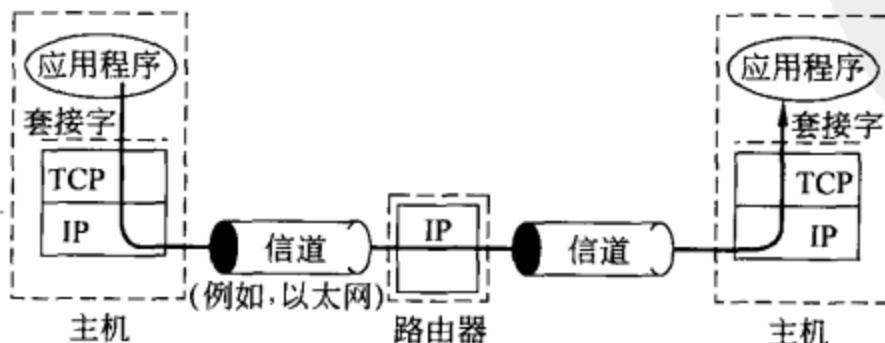


图 1.1 TCP/IP 网络

虚线表示) 访问 UDP 和 TCP 提供的服务。箭头描绘了数据流动的方向: 从应用程序开始, 通过 TCP 和 IP 的实现, 通过网络, 并在另一端通过 IP 和 TCP 的实现进行备份。

在 TCP/IP 中, 底下一层由底层通信信道(例如, 以太网或拨号调制解调器连接)组成。这些信道由网络层(network layer)使用, 这一层处理把分组转发到其目的地的问题(即路由器所做的工作)。TCP/IP 协议族中唯一的网络层协议是网际协议 IP; 它解决的问题是使任意两台主机之间的信道和路由器序列看起来像是主机到主机之间的单独一条信道。

网际协议提供了数据报(datagram)服务: 通过网络独立地处理并递送每个分组, 就像通过邮政系统发送信件或包裹一样。为了使之工作, 每个 IP 分组都必须包含其目的地的地址, 就像把每个邮件包裹发送到某个人的地址一样(稍后将讨论关于地址的更多知识)。尽管大多数投递公司都会保证包裹的递送, 但是 IP 是唯一一种“尽力而为”的协议: 它尝试递送每个分组, 但是在通过网络传输过程中它可能(并且偶尔会)丢失、重新排序或复制分组。

IP 上面一层称为传输层(transport layer)。它允许在两种协议中选择其一: TCP 和 UDP。每种协议都构建于 IP 提供的服务之上, 但是它们采用不同的方式提供不同类型的传输, 它们被具有不同需求的应用程序协议(application protocol)使用。TCP 和 UDP 具有一个共同的功能: 寻址。回忆可知 IP 把分组递送到主机; 显然, 需要进行更细粒度的寻址以便把分组送到特定的应用程序, 也许是同一台主机上使用网络的许多应用程序之一。TCP 和 UDP 都使用地址(称为端口号(port number))来确定主机内的应用程序。TCP 和 UDP 被称为端到端传输协议(end-to-end transport protocol), 因为它们自始至终把数据从一个程序运送到另一个程序(而 IP 只把数据从一台主机运送到另一台主机)。

TCP 被设计成在 IP 提供的主机到主机信道中检测可能发生的丢失、复制分组及其他错误, 并从中恢复过来。TCP 提供了可靠的字节流(reliable byte-stream)信道, 因此应用程序不必处理这些问题。它是面向连接(connection-oriented)的协议: 在把它用于通信之前, 两个程序必须先建立一条 TCP 连接, 这涉及在两台通信的计算机上的 TCP 实现之间完成握手消息(handshake message)的交换。使用 TCP 在许多方面也类似于文件输入/输出(I/O)。事实上, 由一个程序写并由另一个程序读的文件就是通过 TCP 连接进行通信的合理模型。另一方面, UDP 不会尝试从 IP 经历的错误中恢复; 它只是扩展 IP “尽力而为”的数据报服务, 使得它在应用程序之间(而不是主机之间)工作。因此, 使用 UDP 的应用程序必须准备好处理分组丢失、重新排序等问题。

## 1.2 关于地址

在邮寄信件时, 在邮政业务可以理解的表格中提供收信人的地址。在可以通过电话与某个人交谈之前, 必须给电话系统提供一个电话号码。与之类似, 在一个程序可以与另一

个程序通信之前，它必须告诉网络某些信息以标识另一个程序。在 TCP/IP 中，它采用两份信息来标识一个特定的程序：IP 使用的 Internet 地址（Internet address）以及端口号（port number），后者是由传输协议（TCP 或 UDP）解释的额外地址。

Internet 地址是二进制数字。它们具有两种类型，对应于已经标准化的网际协议的两个版本。最常见的是版本 4（IPv4，[12]）；另一种是版本 6（IPv6，[5]），它刚刚开始部署。IPv4 地址的长度为 32 位，因为这正好足够用于标识 40 亿个不同的目的地，对于今天的 Internet 来说，它们确实不够大（这看起来似乎有很多地址，但是由于分配它们的方式，许多地址被浪费掉了。在全部 IPv4 地址空间中，目前已经分配了超过一半的地址）。因此，引入了 IPv6。IPv6 地址的长度为 128 位。

### 1.2.1 记下 IP 地址

在表示 Internet 地址以便于人类使用（与在程序内使用它们相对）方面，为 IP 的两个版本采用了不同的约定。按照惯例，将 IPv4 地址写为一组 4 个用句点隔开的十进制数字（例如，10.1.2.3），这称为点分四组（dotted-quad）表示法。点分四组字符串中的 4 个数字表示 Internet 地址的 4 个字节的内容——因此，每个部分都是一个 0~255 之间的数字。

另一方面，根据约定，16 字节的 IPv6 被表示为用冒号隔开的十六进制数字的组合（例如，2000:fdb8:0000:0000:0001:00ab:853c:39a1）。每一组数字表示 2 字节的地址；可以省略前导 0，因此上述示例中的第 5 组和第 6 组可以只呈现为:1:ab:。此外，只包含 0 的一个组序列可以完全省略（而在地址的其余部分中留下用于分隔它们的冒号）。因此，上述示例可以写为 2000:fdb8::1:00ab:853c:39a1。

从技术上讲，每个 Internet 地址都指的是主机与底层通信信道之间的连接——换句话说，就是网络接口。一台主机可能具有多个接口，例如，主机同时具有通往有线网络（以太网）和无线网络（WiFi）的连接是比较常见的。由于每条这样的网络连接都属于单个主机，Internet 地址就标识主机及其通往网络的连接。不过，反过来则不然，因为单个主机可能具有多个接口，并且每个接口可能具有多个地址（事实上，同一个接口可以同时具有 IPv4 和 IPv6 地址）。

### 1.2.2 处理两个版本

在编写本书第一版时，IPv6 尚未受到广泛的支持。今天，大多数系统都能够“开包即用”地支持 IPv6。为了顺利地从 IPv4 过渡到 IPv6，大多数系统都是双栈（dual-stack）系统，同时支持 IPv4 和 IPv6。在这样的系统中，每个网络接口（信道连接）可能至少具有一个 IPv4 地址和一个 IPv6 地址。

两个 IP 版本的存在使套接字程序员的生活变得复杂。一般来讲，在创建要通信的套接字时，将需要选择 IPv4 或 IPv6 作为底层协议。那么，怎样编写可以同时使用这两个版本

的应用程序呢？幸运的是，双栈系统通过支持这两个协议版本并且允许 IPv6 套接字与 IPv4 或 IPv6 应用程序通信，来处理互操作性。当然，IPv4 和 IPv6 地址区别很大；不过，可以使用 IPv4 映射的地址（IPv4 mapped address）将 IPv4 地址映射为 IPv6 地址。IPv4 映射的地址是通过在 IPv4 地址前添加 4 个字节的前缀::ffff 而构成的。例如，132.3.23.7 的 IPv4 映射的地址是::ffff:132.3.23.7。为了便于人类阅读，通常用点分四组表示法书写最后 4 个字节。在第 3 章中将更详细地讨论协议互操作性。

不幸的是，具有 IPv6 Internet 地址并不足以让你能够通过 Internet 与所有其他支持 IPv6 的主机通信。为此，还必须与 ISP（Internet Service Provider，Internet 服务提供商）协商，要求提供 IPv6 转发服务。

### 1.2.3 端口号

我们以前提过，在把消息传送到程序时需要采用两段地址。TCP 或 UDP 中的端口号总是相对于 Internet 地址解释的。回到我们前面的类比上来，端口号对应于给定街道地址（比如大型建筑物）的房间号。邮政业务使用街道地址把信件投递到邮箱；然后，任何清空邮箱的人负责把信件送到建筑物内正确的房间。或者考虑一家具有内部电话系统的公司：要与该公司内的某个人通话，首先要拨打该公司的主机电话号码以连接到内部电话系统，然后拨打你希望与之通话的某个人的特定电话的分机号。在这些类比中，Internet 地址就是街道地址或公司的主机电话号码，而端口则对应于房间号或电话分机号。端口号在 IPv4 和 IPv6 中是相同的：即 16 位无符号的二进制数字。因此，每个端口号的范围在 1~65 535 之间（0 是预留的）。

### 1.2.4 特殊地址

在 IP 的每个版本中，都定义了某些特殊用途的地址。其中值得知道的地址是回送地址（loopback address），它总是被分配给特殊的回送接口（loopback interface），回送接口是简单地把传输的分组正确地回送给发送者的虚拟设备。回送接口对于测试非常有用，因为发给该地址的分组将立即返回到目的地。而且，它存在于每一台主机上，甚至当计算机没有其他的接口（即未连接到网络）时也可以使用它。IPv4 的回送地址是 127.0.0.1；<sup>1</sup>，对于 IPv6，它是 0:0:0:0:0:0:1（或者只是::1）。

另一组 IPv4 地址是为特殊用途预留的，包括那些为“专用”预留的地址。这组地址包括以 10 或 192.168 开头的所有 IPv4 地址，以及那些第一个数字是 172 并且第二个数字在 16~31 之间的地址（对于 IPv6 没有对应的类别）。这些地址最初被指定在不属于全球 Internet 一部分的专用网络中使用。今天，它们通常用在通过 NAT（network address translation，网

<sup>1</sup> 从技术上讲，以 127 开头的任何 IPv4 地址都应该是回送地址。

络地址转换) [7] 设备连接到 Internet 的家庭和小型办公室网络中。这种设备充当路由器，用于在转发分组中转换(重写)其中的地址和端口。更确切地讲，它把其接口之一上的分组中的(专用地址，端口)对映射到其他接口上的(公共地址，端口)对。这允许一小组主机(例如，家庭网络上的那些主机)有效地“共享”单个 IP 地址。这些地址的重要性在于：不能从全球 Internet 到达它们。如果在一台具有专用类别中的地址(例如，在家庭网络上)的机器上试验本书中的代码，并且尝试与不具有这些地址之一的另一台主机通信，一般不会成功，除非具有专用地址的主机发起通信——即使这样也可能会失败。

相关类别包含链路本地(link-local)或“自动配置”地址。对于 IPv4，这类地址以 169.254 开头。对于 IPv6，其前 16 位块是 FE80、FE90、FEA0 或 FEB0 的任何地址都是链路本地地址。这些地址只能用于连接到同一个网络的主机之间的通信；路由器将不会转发把这类地址作为其目的地的分组。

最后，另一个类别包含多播(multicast)地址。常规的 IP(有时称为“单播”)地址指单一目的地，而多播地址潜在地指任意数量的目的地。多播是一个高级主题，将在第 6 章中简要介绍它。在 IPv4 中，点分四组格式的多播地址的第一个数字在 224~239 之间。在 IPv6 中，多播地址以 FF 开头。

## 1.3 关于名称

你最有可能习惯于按名称(name)引用主机(例如，host.example.com)。不过，Internet 协议处理的是地址(二进制数字)，而不是名称。你应该理解使用名称而不是地址是一种方便的特性，它独立于 TCP/IP 提供的基本服务——无需使用名称即可编写并使用 TCP/IP 应用程序。在使用名称标识通信端点时，系统将做一些额外的工作把名称解析为地址。由于两个原因，这个额外的步骤通常是值得做的。第一，与点分四组表示法(就 IPv6 而言，是十六进制的数字串)相比，人类显然更容易记住名称。第二，名称提供了某种级别的间接性，它把用户与 IP 地址变化隔离开。在编写本书第 1 版期间，Web 服务器 www.mkp.com 的地址改变了。由于我们总是按名称引用该 Web 服务器，www.mkp.com 将解析成当前 Internet 地址，而不是 208.164.121.48。IP 地址中的变化对于使用名称访问 Web 服务器的程序是透明的。

名称解析服务可以访问广泛来源的信息。两种主要的来源是 DNS(Domain Name System，域名系统)和本地配置数据库。DNS [8] 是一种分布式数据库，它把像 www.mkp.com 这样的域名映射到 Internet 地址及其他信息；DNS 协议[9]允许连接到 Internet 的主机使用 TCP 或 UDP 从该数据库中检索信息。本地配置数据库一般是用于本地名称-Internet 地址映射的特定于操作系统的机制。

## 1.4 客户与服务器

在我们的邮政和电话系统的类比中，每次通信都是由一方发起的，发起方发送信件或者打电话，而另一方则通过发送回复信件或者拿起电话并交谈来响应发起者的联系。Internet 通信是类似的。客户（client）和服务器（server）就是指这些角色：客户程序发起通信，而服务器程序则被动地等待，然后响应联系它的客户。客户与服务器一起组成了应用程序（application）。客户与服务器描述了一种典型的情况，其中服务器使一种特定的能力（例如，数据库服务）可供能够与之通信的任何客户使用。

程序是充当客户还是充当服务器决定了它使用 Sockets API 与其对应方（peer）建立通信的一般形式（客户是服务器的对应方，反之亦然）。此外，客户-服务器的区别很重要，因为客户最初需要知道服务器的地址和端口，但是服务器则不然。利用 Sockets API，当服务器接收到来自客户的初始通信时，如果需要，它可以获悉客户的地址信息。这类似于打电话——为了被打电话，一个人不需要知道打电话的人的电话号码。与打电话一样，一旦建立了连接，服务器与客户之间的区别就消失了。

客户怎样查明服务器的 IP 地址和端口号呢？通常，客户知道它想要与之通信的服务器的名称——例如，通过像 <http://www.mkp.com> 这样的 URL（Universal Resource Locator，统一资源定位器），并使用名称解析服务获悉相应的 Internet 地址。

查找服务器的端口号则是一个不同的故事。从原理上讲，服务器可以使用任何端口，但是客户必须能够获悉它是什么。在 Internet 中，具有把众所周知的端口号分配给某些应用程序的惯例。IANA（Internet Assigned Number Authority，Internet 编号授权委员会）监督这种分配方式。例如，端口号 80 被分配给 HTTP（HyperText Transfer Protocol，超文本传输协议）。在运行 HTTP 客户浏览器时，它默认会尝试联系那个端口上的 Web 服务器。所有分配的端口号列表是由 Internet 编号授权委员会维护的（参见 <http://www.iana.org/assignments/port-numbers>）。

你可能听过客户-服务器的一种替代方案，即 P2P（peer-to-peer，点对点）方案。在 P2P 中，应用程序同时消费和提供服务，这与传统的客户-服务器体系结构不同，在这种体系结构中，服务器提供服务，客户则消费服务。事实上，P2P 节点有时也称为“servent”，它结合了 server 和 client 这两个单词。那么，你需要学习一组不同的技术给 P2P（而不是客户-服务器）编程吗？不是这样的。在 Sockets 中，客户与服务器之间的区别只在于谁建立初始连接，以及谁等待连接。P2P 应用程序通常会发起连接（到达现有的 P2P 节点）并接受连接（来自其他 P2P 节点）。在阅读本书后，你将能够像客户-服务器那样很好地编写 P2P 应用程序。

## 1.5 什么是套接字

套接字（socket）是一个抽象层，应用程序可以通过它发送和接收数据，其方式与打开文件句柄允许应用程序读、写数据到稳定的存储器非常相似。套接字允许应用程序插入到网络中，并与插入到同一个网络中的其他应用程序通信。由一台机器上的应用程序写到套接字中的信息可以被不同机器上的应用程序读取，反之亦然。

不同类型的套接字对应于不同的底层协议族以及协议族内的不同协议栈。本书只论及 TCP/IP 协议族。今天，TCP/IP 中的主要类型的套接字是流套接字（stream socket）和数据报套接字（datagram socket）。流套接字使用端到端协议（其下一层是 IP），从而提供可靠的字节流服务。TCP/IP 流套接字代表 TCP 连接的一端。数据报套接字使用 UDP（同样，其下一层是 IP），从而提供“尽力而为”的数据报服务，应用程序可以使用它来发送各个最大长度大约为 65 500 个字节的消息。流套接字和数据报套接字还受到其他协议族的支持，但是本书只论及 TCP 流套接字和 UDP 数据报套接字。TCP/IP 套接字是由 Internet 地址、端到端协议（TCP 或 UDP）和端口号唯一标识的。在学习本书的过程中，你将遇到将套接字绑定到地址的多种方式。

图 1.2 描绘了单个主机内的应用程序、套接字抽象层、协议和端口号之间的逻辑关系。关于这些关系要注意几点。第一，一个程序可以同时使用多个套接字。第二，多个程序可以同时使用同一个套接字抽象层，尽管这种情况不太常见。图 1.2 显示每个套接字都有一个关联的本地 TCP 或 UDP 端口，它用于把传入的分组指引到应该接收它们的应用程序。以前我们说过端口标识主机上的应用程序。实际上，端口标识主机上的套接字。不过，如图 1.2 所示，由于多个套接字可以与一个本地端口相关联，不能仅仅使用端口来标识套接字。对于 TCP 套接字，这种情况最常见；幸运的是，不需要理解这种细节即可编写使用 TCP 套接字的客户-服务器程序。第 7 章中将介绍关于这方面的完整知识。

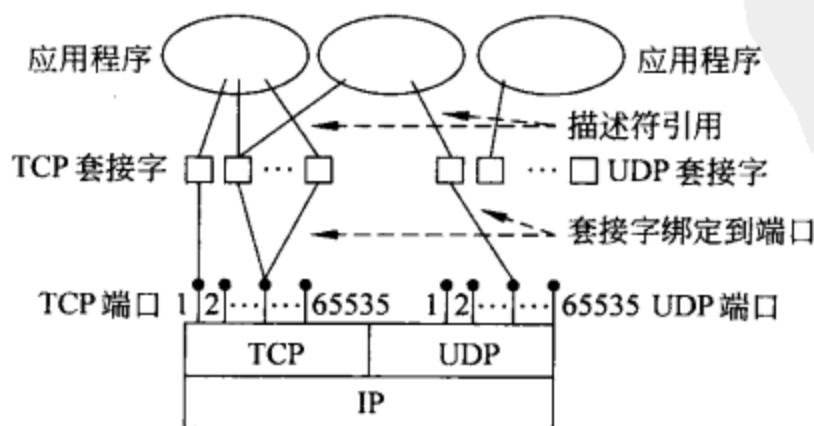


图 1.2 套接字、协议与端口

## 练习题

1. 在\*NIX 中使用 ifconfig 命令或者在 Windows 中使用 ipconfig 命令报告你的 IP 地址。确定这些地址都是 IPv6 格式的地址。
2. 使用 hostname 命令报告你正在使用的计算机的名称。
3. 你可以查明任何直接连接的路由器的 IP 地址吗？
4. 使用 Internet 搜索试着了解关于 IPv5 所发生的事情。
5. 使用尽可能少的字符书写以下 IPv6 地址：2345:0000:0000:A432:0000:0000:0000:0023
6. 你能够想到不适合客户-服务器模型的真实的通信示例吗？
7. 你的家庭连接到了多少种不同类型的网络？有多少种网络支持双向传输？
8. IP 是一种“尽力而为”的协议，它要求把信息分解为数据报，它们可能会丢失、复制或重新排序。TCP 隐藏了所有这些细节，提供了一种可靠的服务，用于接收和递送完整的字节流。关于在 IP 之上提供 TCP 服务你可能会做什么？在 TCP 可用时，为什么任何人都会使用 UDP？

## 第 2 章

### 基本的 TCP 套接字

**现** 在该学习编写你自己的套接字应用程序了。我们将从 TCP 开始。此时，你可能准备好动手编写一些实际的代码，因此我们首先将完成一个 TCP 客户和服务器的工作示例，然后将介绍基本 TCP 中使用的套接字 API 的详细信息。为了保持事情更简单，我们最初将介绍适用于一种特定的 IP 版本（即 IPv4）的代码，在编写本书时，它仍然是网际协议占主导地位的版本，这样就有了广泛的回旋余地。本章末尾介绍了编写客户和服务器的 IPv6 版本所需要的（少量）修改。第 3 章将演示独立于协议的应用程序的创建方法。

我们的示例客户和服务器实现了应答（echo）协议。它的工作方式如下：客户连接到服务器并发送它的数据；服务器简单地把它接收到的任何内容发送回客户并断开连接。在应用程序中，客户发送的数据是作为命令行参数提供的字符串。客户将打印它从服务器接收到的数据，因此可以看到返回的内容。许多系统出于调试和测试的目的都包括应答服务。

#### 2.1 IPv4 TCP 客户

客户与服务器之间的区别很重要，因为在通信过程中的某些步骤中它们以不同的方式使用套接字接口。我们首先重点关注客户。它的职责是发起与被动等待联系的服务器之间的通信。

典型的 TCP 客户的通信涉及如下 4 个基本步骤。

- (1) 使用 `socket()` 创建 TCP 套接字。
- (2) 使用 `connect()` 建立到达服务器的连接。
- (3) 使用 `send()` 和 `recv()` 通信。
- (4) 使用 `close()` 关闭连接。

`TCPEchoClient4.c` 是用于 IPv4 的 TCP 应答客户的实现。

**TCPEchoClient4.c**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>
9 #include "Practical.h"
10
11 int main(int argc, char *argv[]) {
12
13 if (argc < 3 || argc > 4) //Test for correct number of arguments
14     DieWithUserMessage("Parameter(s)",
15         "<Server Address> <Echo Word> [<Server Port>]");
16
17 char *servIP = argv[1]; //First arg; server IP address (dotted quad)
18 char *echoString = argv[2]; //Second arg: string to echo
19
20 //Third arg (optional): server port (numeric). 7 is well-known echo port
21 in_port_t servPort = (argc == 4) ? atoi(argv[3]) : 7;
22
23 //Create a reliable, stream socket using TCP
24 int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
25 if (sock < 0)
26     DieWithSystemMessage("socket() failed");
27
28 //Construct the server address structure
29 struct sockaddr_in servAddr; //Server address
30 memset(&servAddr, 0, sizeof(servAddr)); //Zero out structure
31 servAddr.sin_family = AF_INET; //IPv4 address family
32 //Convert address
33 int rtnVal = inet_pton(AF_INET, servIP, &servAddr.sin_addr.s_addr);
34 if (rtnVal == 0)
35     DieWithUserMessage("inet_pton() failed", "invalid address string");
36 else if (rtnVal < 0)
37     DieWithSystemMessage("inet_pton() failed");
38 servAddr.sin_port = htons(servPort); //Server port
39
40 //Establish the connection to the echo server
41 if (connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
42     DieWithSystemMessage("connect() failed");
```

```

43
44 size_t echoStringLen = strlen(echoString); //Determine input length
45
46 //Send the string to the server
47 ssize_t numBytes = send(sock, echoString, echoStringLen, 0);
48 if (numBytes < 0)
49     DieWithSystemMessage("send() failed");
50 else if (numBytes != echoStringLen)
51     DieWithUserMessage("send()", "sent unexpected number of bytes");
52
53 //Receive the same string back from the server
54 unsigned int totalBytesRcvd = 0; //Count of total bytes received
55 fputs("Received: ", stdout); //Setup to print the echoed string
56 while (totalBytesRcvd < echoStringLen) {
57     char buffer[BUFSIZE]; //I/O buffer
58     /* Receive up to the buffer size (minus 1 to leave space for
59      a null terminator) bytes from the sender */
60     numBytes = recv(sock, buffer, BUFSIZE - 1, 0);
61     if (numBytes < 0)
62         DieWithSystemMessage("recv() failed");
63     else if (numBytes == 0)
64         DieWithUserMessage("recv()", "connection closed prematurely");
65     totalBytesRcvd += numBytes; //Keep tally of total bytes
66     buffer[numBytes] = '\0'; //Terminate the string!
67     fputs(buffer, stdout); //Print the echo buffer
68 }
69
70 fputc('\n', stdout); //Print a final linefeed
71
72 close(sock);
73 exit(0);
74 }

```

TCPEchoClient4.c 做了以下事情。

(1) 应用程序建立和参数解析：第 1~21 行。

- 包括文件：第 1~9 行。

这些头文件声明了 API 的标准函数和常量。参考文档（例如，参考手册），了解用于系统上的套接字函数和数据结构的合适的包括文件。利用了自己的包括文件 Practical.h，它带有用自己的函数的原型，将在下面描述它们。

- 典型的参数解析和可靠性检查：第 13~21 行。

作为前两个参数传入要发回的 IPv4 地址和字符串。客户可以选择接受服务器端口

作为第三个参数。如果没有提供端口，客户将使用众所周知的应答协议端口7。

(2) TCP套接字的创建：第23~26行。

使用socket()函数创建套接字。该套接字用于IPv4(`af_inet`)，使用称为TCP(`ipproto_tcp`)的基于流的协议(`sock_stream`)。如果成功，socket()就会为套接字返回一个整数值的描述符或者“句柄”。如果套接字失败，它就会返回-1，并且会调用错误处理函数`DieWithSystemMessage()`(以后将描述它)，打印一个提供信息的提示并退出。

(3) 准备地址并建立连接：第28~42行。

- 准备`sockaddr_in`结构用于存放服务器地址：第29~30行。

为了建立套接字，必须指定要连接到的地址和端口。`sockaddr_in`结构被定义为这种信息的“容器”。调用`memset()`确保未显式设置的结构的任何部分都包含0。

- 填充`sockaddr_in`：第31~38行。

必须设置地址族(`AF_INET`)、Internet地址和端口号。函数`inet_pton()`把服务器的Internet地址的字符串表示(以点分四组表示法作为命令行参数传入)转换为32位的二进制表示。以前把服务器的端口号从命令行字符串转换为二进制形式；调用`hton()`(意指“host to network short”)确保根据API的需要对二进制值进行格式化(第5章中描述了这样做的原因)。

- 连接：第40~42行。

`connect()`函数用于建立给定套接字与由`sockaddr_in`结构中的地址和端口标识的套接字之间的连接。由于Sockets API是通用的，需要把指向`sockaddr_in`地址结构(它特定于IPv4地址)的指针强制转换(`cast`)为泛型类型(`sockaddr*`)，并且必须提供地址数据结构的实际大小。

(4) 把应答字符串发送给服务器：第44~51行。

查明参数字符串的长度并保存它，以便以后使用。把指向应答字符串的指针传递给`send()`调用；在应用程序启动时，把字符串本身存储在某个位置(像所有的命令行参数一样)。我们确实不关心它位于何处，而只需知道第一个字节的地址以及要发送多少字节(注意：我们不会发送位于参数字符串末尾(以及C语言中的所有字符串)的字符串末尾标记符(0))。如果成功，`send()`就会返回发送的字节数；否则，它就会返回-1。如果`send()`失败或者发送了错误的字节数，我们必须处理错误。注意：这里将不会发生发送错误的字节数这种情况。然而，包括进针对这种情况的测试是一个好主意，因为在某些环境中可能发生这种错误。

(5) 接收应答服务器的答复：第53~70行。

TCP是一种字节流协议。这类协议的一种实现是不会保持`send()`的边界。通过在连接的一端调用`send()`发送的字节可能不会通过在另一端单独调用一次`recv()`而全都返回(在第7章中将更详细地讨论这个问题)。因此需要反复接收字节，直至接收到的字节数与发送的

字节数相等为止。这个循环十有八九只会执行一次，因为来自服务器的数据事实上将一次性全部返回；不过，不保证这会发生，因此必须允许需要多次读取字节的可能性。编写使用套接字的应用程序的基本原则是：对于另一端的网络和程序将要做什么事情，永远都不能做出假设。

- 接收字节块：第 57~65 行。

`recv()`会阻塞到数据可用为止，并且返回复制到缓冲区中的字节数，如果失败，则会返回 -1。如果返回值为 0，则指示另一端的应用程序关闭了 TCP 连接。注意：传递给 `recv()` 的大小参数将为添加 null 终止字符预留空间。

- 打印缓冲区：第 66~67 行。

在接收到服务器发送的数据时将打印它。在接收到的每个数据块末尾添加 null 终止字符（0），以便 `fputs()` 可以把它作为字符串处理。我们没有检查接收到的字节数是否与发送的字节数相等。服务器可能发送完全不同的内容（这取决于发送的字符串的长度），并将把它写到标准输出。

- 打印换行符：第 70 行。

当接收到与发送的一样多的字节时，就会退出循环，并打印一个换行符。

- (6) 终止连接并退出：第 72~73 行。

`close()` 函数通知远程套接字通信结束，然后释放分配给套接字的本地资源。

客户应用程序（实际上包括本书中的所有程序）利用了如下两个错误处理函数：

```
DieWithUserMessage(const char *msg, const char *detail)
DieWithSystemMessage(const char *msg)
```

这两个函数都把用户提供的消息字符串（`msg`）打印到 `stderr`，其后接着是详细的消息字符串；然后，它们利用一个错误返回代码调用 `exit()`，导致应用程序终止。唯一的区别是详细消息的来源。对于 `DieWithUserMessage()`，详细消息是用户提供的；对于 `DieWithSystemMessage()`，详细消息是由系统基于特殊变量 `errno` 的值提供的（该变量描述了系统调用的最近失败（如果有的话）的原因）。仅当错误情况是由设置 `errno` 的系统调用产生的时，才会调用 `DieWithSystemMessage()`（为了使程序保持简单，示例没有包含专门用于从错误中恢复的大量代码——它们只是简单地终止程序并退出。生产代码一般不应该如此轻易地放弃）。

偶尔，我们需要在不退出的情况下给用户提供信息；如果需要格式化能力，就使用 `printf()`；否则，就使用 `fputs()`。特别是，应该尽量避免使用 `printf()` 输出固定的、预先格式化的字符串。你永远也不应该做的一件事是：把从网络接收到的文本作为第一个参数传递给 `printf()`。它会引起严重的安全性问题，要代之以使用 `fputs()`。

注意：`DieWith...()` 函数是在头文件 “`Practical.h`” 中声明的。不过，这些函数的实际实

现包含在 DieWithMessage.c 文件中，应该编译该文件并把它与本书中的所有示例应用程序相连接。

### DieWithMessage.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void DieWithUserMessage(const char *msg, const char *detail) {
5     fputs(msg, stderr);
6     fputs(": ", stderr);
7     fputs(detail, stderr);
8     fputc('\n', stderr);
9     exit(1);
10 }
11
12 void DieWithSystemMessage(const char *msg) {
13     perror(msg);
14     exit(1);
15 }
```

如果编译 TCPEchoClient4.c 和 DieWithMessage.c 创建程序 TCPEchoClient4，就可以与具有 Internet 地址 169.1.1.1 的应答服务器通信，如下所示：

```
% TCPEchoClient4 169.1.1.1 "Echo this!"
Received: Echo this!
```

为了让客户工作，就需要服务器。许多系统出于调试和测试的目的，包括了一个应答服务器；不过，出于安全性原因，这样的服务器最初通常是禁用的。如果不能访问应答服务器，也没有问题，因为我们将编写一个应答服务器。

## 2.2 IPv4 TCP 服务器

我们现在把注意力转向构造 TCP 服务器。服务器的职责是建立通信端点，并被动等待来自客户的连接。对于基本的 TCP 服务器通信，要执行如下 4 个常规的步骤。

- (1) 使用 `socket()` 创建 TCP 套接字。
- (2) 利用 `bind()` 给套接字分配端口号。
- (3) 使用 `listen()` 告诉系统允许对该端口建立连接。
- (4) 反复执行以下操作：
  - 调用 `accept()` 为每个客户连接获取新的套接字。

- 使用 `send()` 和 `recv()` 通过新的套接字与客户通信。
- 使用 `close()` 关闭客户连接。

创建套接字以及发送、接收和关闭这些操作与客户中相同。不同之处在于：服务器中的套接字的使用必须涉及将一个地址绑定到套接字，然后把该套接字用作获得连接到客户机的其他套接字的方式（在代码后面的注释中详细说明这一点）。服务器与每个客户机的通信都尽可能简单：它简单地接收客户连接上的数据，并把相同的数据发送回客户机；它会重复这个过程，直至客户机关闭它那一端的连接，此时，将不会传入更多的数据。

### TCPEchoServer4.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>
8 #include "Practical.h"
9
10 static const int MAXPENDING = 5; //Maximum outstanding connection requests
11
12 int main(int argc, char *argv[]) {
13
14 if (argc != 2) //Test for correct number of arguments
15   DieWithUserMessage("Parameter(s)", "<Server Port>");
16
17 in_port_t servPort = atoi(argv[1]); //First arg: local port
18
19 //Create socket for incoming connections
20 int servSock; //Socket descriptor for server
21 if ((servSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
22   DieWithSystemMessage("socket() failed");
23
24 //Construct local address structure
25 struct sockaddr_in servAddr; //Local address
26 memset(&servAddr, 0, sizeof(servAddr)); //Zero out structure
27 servAddr.sin_family = AF_INET; //IPv4 address family
28 servAddr.sin_addr.s_addr=htonl(INADDR_ANY); //Any incoming interface
29 servAddr.sin_port = htons(servPort); //Local port
30
31 //Bind to the local address
32 if (bind(servSock, (struct sockaddr*) &servAddr, sizeof(servAddr)) < 0)

```

```

33 DieWithSystemMessage("bind() failed");
34
35 //Mark the socket so it will listen for incoming connections
36 if (listen(servSock, MAXPENDING) < 0)
37 DieWithSystemMessage("listen() failed");
38
39 for (;;) {                                //Run forever
40     struct sockaddr_in clntAddr;    //Client address
41     //Set length of client address structure (in-out parameter)
42     socklen_t clntAddrLen = sizeof(clntAddr);
43
44     //Wait for a client to connect
45     int clntSock=accept(servSock, (struct sockaddr*)&clntAddr, &clntAddrLen);
46     if (clntSock < 0)
47         DieWithSystemMessage("accept() failed");
48
49     //clntSock is connected to a client!
50
51     char clntName[INET_ADDRSTRLEN]; //String to contain client address
52     if (inet_ntop(AF_INET, &clntAddr.sin_addr.s_addr, clntName,
53         sizeof(clntName)) != NULL)
54         printf("Handling client %s/%d\n", clntName, ntohs(clntAddr.sin_port));
55     else
56         puts("Unable to get client address");
57
58     HandleTCPClient(clntSock);
59 }
60 //NOT REACHED
61 }

```

(1) 程序建立和参数解析：第1~17行。

使用 atoi() 函数把端口号从字符串转换为数值；如果第一个参数不是一个数字，atoi() 将返回 0，以后当调用 bind() 时将引发一个错误。

(2) 套接字创建和设置：第19~37行。

- 创建 TCP 套接字：第20~22行。

我们将创建一个流套接字，就像在客户中所做的那样。

- 填充想要的端点地址：第25~29行。

在服务器上，需要把服务器套接字与地址和端口号相关联，使得客户连接可以到达正确的位置。既然是为 IPv4 编写程序，所以应使用 sockaddr\_in 结构。由于我们不是非常关心所在的地址（分配给运行服务器的机器的任何地址都可以），因此允许系统通过指定通配符地址 inaddr\_any 作为我们想要的 Internet 地址来选择它（对于

服务器来说，这通常是要做的正确事情，并且它使服务器不必查明任何实际的 Internet 地址）。在 `sockaddr_in` 中设置地址和端口号之前，分别使用 `htonl()` 和 `htons()` 把它们转换为网络字节顺序（byte order）（参见 5.1.2 节，了解详细信息）。

- 把套接字绑定到指定的地址和端口：第 32~33 行。

如上所述，服务器的套接字需要与本地地址和端口相关联；`bind()` 函数用于完成这个任务。注意：**客户必须给 `connect()` 提供服务器的地址，而服务器必须给 `bind()` 指定它自己的地址**。它们必须对这份信息（即服务器的地址和端口）达成协议以便进行通信；它们实际上都不需要知道客户的地址。注意：`bind()` 可能由于多种原因而失败；最重要的原因之一是：另外某个套接字已经绑定到指定的端口（参见 7.5 节）。此外，在一些系统上，需要特殊的权利以绑定到某些端口（通常是那些编号小于 1024 的端口）。

- 设置要侦听的套接字：第 36~37 行。

`listen()` 调用告诉 TCP 实现允许来自客户的进入的连接。在调用 `listen()` 之前，对套接字地址的任何进入的连接请求都将被无声地拒绝——也就是说，`connect()` 在客户端将会失败。

### (3) 反复处理进入的连接：第 39~59 行。

- 接受进入的连接：第 40~47 行。

如以前所讨论的，在其上调用 `listen()` 的 TCP 套接字的使用方式不同于在客户应用程序中看到的套接字。服务器应用程序不是在套接字上执行发送和接收，而是调用 `accept()`，它会阻塞，直至建立了连接以侦听套接字的端口号为止。此时，`accept()` 为新套接字返回一个描述符，它已经连接到发起连接的远程套接字。第二个参数指向 `sockaddr_in` 结构，第三个参数是一个指针，指向此结构的长度。一旦成功，`sockaddr_in` 就会包含返回的套接字连接到的客户的 Internet 地址和端口，并将这个地址的长度写入第三个参数指向的整数中。注意：由返回的描述符引用的套接字已经被连接，这意味着它准备好进行发送和接收（关于底层实现中所发生的事情的详细信息，参见 7.4.1 节）。

- 报告被连接客户：第 51~56 行。

此时，`clntAddr` 包含连接客户的地址和端口号；我们提供了一个“呼叫者 ID”函数，并打印出客户的信息。如你可能期望的，`inet_ntop()` 执行与 `inet_pton()`（我们在客户中使用过它）相反的操作。它接受客户地址的二进制表示，并把它转换为点分四组格式的字符串。由于实现以所谓的网络字节顺序处理端口和地址（5.1.2 节），在把端口号传递给 `printf()` 之前必须转换它（显然，`inet_pton()` 负责为地址执行这种转换）。

- 处理应答客户：第 58 行。

`HandleTCPClient()` 负责“应用程序协议”。下面将讨论它。因此，我们分离出了服

务器的特定于“应答”的部分。

我们分离出了实现应答服务器的“应答”部分的函数。尽管这个应用程序协议(application protocol)只用少数几行代码即可实现，但它是一种良好的设计实践，将其细节与服务器代码的其余部分隔离开。这促进了代码重用。

`HandleTCPClient()`在给定的套接字上接收数据，并在相同的套接字上发回它，只要 `recv()` 返回一个正值（指示接收到数据），这个过程就会反复进行。`recv()`会阻塞到接收到数据或者客户关闭连接为止。当客户正常地关闭连接时，`recv()`返回 0。可以在 `TCPServerUtility.c` 文件中找到 `HandleTCPClient()`。

## HandleTCPClient()

---

```

1 void HandleTCPClient(int clntSocket) {
2     char buffer[BUFSIZE]; //Buffer for echo string
3
4     //Receive message from client
5     ssize_t numBytesRcvd = recv(clntSocket, buffer, BUFSIZE, 0);
6     if (numBytesRcvd < 0)
7         DieWithSystemMessage("recv() failed");
8
9     //Send received string and receive again until end of stream
10    while (numBytesRcvd > 0) { //0 indicates end of stream
11        //Echo message back to client
12        ssize_t numBytesSent = send(clntSocket, buffer, numBytesRcvd, 0);
13        if (numBytesSent < 0)
14            DieWithSystemMessage("send() failed");
15        else if (numBytesSent != numBytesRcvd)
16            DieWithUserMessage("send()", "sent unexpected number of bytes");
17
18        //See if there is more data to receive
19        numBytesRcvd = recv(clntSocket, buffer, BUFSIZE, 0);
20        if (numBytesRcvd < 0)
21            DieWithSystemMessage("recv() failed");
22    }
23
24    close(clntSocket); //Close client socket
25 }
```

假设把 `TCPEchoServer4.c`、`DieWithMessage.c`、`TCPServerUtility.c` 和 `AddressUtility.c` 编译成可执行程序 `TCPEchoServer4`，并在 Internet (IPv4) 地址为 169.1.1.1、端口为 5000 的主机上运行该程序。另外假设在 Internet 地址为 169.1.1.2 的主机上运行客户并把它连接到服务器上。服务器的输出应该如下所示：

```
% TCPEchoServer4 5000
Handling client 169.1.1.2
```

而客户的输出应该如下所示：

```
% TCPEchoClient4 169.1.1.1 "Echo this!" 5000
Received: Echo this!
```

服务器把它的套接字绑定到端口 5000，并等待来自客户的连接请求。客户建立连接，发送消息“Echo this!”给服务器，并接收应答的响应。在这个命令中，必须在命令行上给 TCPEchoClient 提供端口号，因为它与应答服务器通信，而该应答服务器位于端口 5000 上，而不是众所周知的端口 7 上。

我们提过编码使用套接字的网络应用程序的关键原则是防御性编程：你的代码绝对不能对通过网络接收到的任何信息做出假定。如果你想试验 TCP 服务器，看看它如何响应多种错误的客户行为，则该怎么办？可以编写一个 TCP 客户程序，用于发送伪造的消息并打印结果；不过，这可能单调乏味并且耗时。更快的替代方案是使用大多数系统上可用的 telnet 程序。这是一个命令行工具，用于连接到服务器，发送输入的任何文本，并打印出响应。telnet 接受两个参数：服务器和端口。例如，要 telnet 到上述的示例应答服务器，可以试试以下命令：

```
% telnet 169.1.1.1 5000
```

现在输入要应答的字符串，telnet 将打印出服务器响应。telnet 的行为在各种实现之间有所不同，因此可能需要研究在系统上使用它的特定细节。

既然已经看到了完整的客户和服务器，下面就更详细地探讨一下组成 Sockets API 的各个函数。

## 2.3 创建和销毁套接字

要使用 TCP 或 UDP 通信，程序首先要求操作系统创建套接字抽象层的实例。完成这个任务的函数是 `socket()`；它的参数指定了程序所需的套接字类型。

---

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

---

第一个参数确定套接字的通信领域（domain）。回忆可知：Sockets API 为大量的通信领域提供了一个泛型接口。不过，我们感兴趣的只是 IPv4（`AF_INET`）和 IPv6（`AF_INET6`）。注意：你可能在本书中看到一些程序使用 `PF_XXX`，而不是 `AF_XXX`。通常，这些值是等

价的，在这种情况下可以互换使用它们，但是不保证这一点。<sup>1</sup>

第二个参数指定套接字的类型 (type)。类型决定了利用套接字进行的数据传输的语义——例如，不管传输是否可靠，也不管是否保留了消息边界，等等。常量 SOCK\_STREAM 利用可靠的字节流语义指定一个套接字，而 SOCK\_DGRAM 则指定一种“尽力而为”的数据报套接字。

第三个参数指定要使用的特定的端到端协议 (end-to-end protocol)。对于 IPv4 和 IPv6，我们希望把 TCP (由常量 IPPROTO\_TCP 标识) 用于流套接字，或者把 UDP (由 IPPROTO\_UDP 标识) 用于数据报套接字。如果把常量 0 作为第三个参数，则会导致系统为指定的协议族和类型选择默认的端到端协议。由于在 TCP/IP 协议族中目前对于流套接字只有一种选择，我们可以指定 0，而不是显式给出协议编号。不过，将来某一天可能会在 Internet 协议族中出现其他的端到端协议实现相同的语义。在这种情况下，指定 0 可能导致使用不同的协议，这可能是或者可能不是人们想要的。主要的任务是确保通信程序使用相同的端到端协议。

我们以前说过 `socket()` 返回通信实例的句柄 (handle)。在源于 UNIX 的系统上，它是一个整数：非负值表示成功，-1 表示失败。非失败的值应该被视作不透明的句柄 (opaque handle)，就像文件描述符一样 (事实上，它就是一个文件描述符，取自与 `open()` 返回的数字相同的空间)。这种句柄 (我们称之为套接字描述符 (socket descriptor)) 被传递给其他 API 函数，以标识要在其上执行操作的套接字抽象层。

当利用套接字完成应用程序时，它会调用 `close()`，提供不再需要的套接字的描述符。

---

```
int close(int socket)
```

---

`close()` 告诉底层协议栈发起关闭通信以及释放与套接字关联的任何资源所需的任何动作。如果成功，`close()` 就会返回 0；如果失败，则会返回 -1。一旦调用了 `close()`，就会对导致错误的套接字调用其他操作 (例如，`send()` 和 `recv()`)。

## 2.4 指定地址

使用套接字的应用程序需要能够标识它们将用于通信的远程端点。我们已经见过客户必须指定它需要与之通信的服务器的地址和端口号。此外，套接字层有时需要把地址传递给应用程序。例如，一种类似于电话网络中的“呼叫者 ID”的特性可以让服务器知道与之通信的每个客户的地址和端口号。

---

<sup>1</sup> 要知道的事实是，这是 Sockets 接口的一个丑陋的部分，并且文档简直毫无用处。

在本节中，我们描述了由 Sockets API 用作这种信息的容器的数据结构。

### 2.4.1 通用地址

Sockets API 定义了一种泛型数据类型——`sockaddr` 结构，用于指定与套接字关联的地址：

```
struct sockaddr {
    sa_family_t sa_family; //Address family (e.g., AF_INET)
    char sa_data[14];      //Family-specific address information
};
```

这个地址结构的第一部分定义了地址族——地址属于的空间。出于我们的目的，我们将总是使用系统定义的常量：`AF_INET` 和 `AF_INET6`，它们分别指定了 IPv4 和 IPv6 的 Internet 地址族。第二部分是一个小位块，其精确形式依赖于地址族（这是在操作系统和网络中处理异构性的典型方式）。如我们在 1.2 节中所讨论的，用于 Internet 协议族的套接字地址具有两个部分：32 位（IPv4）或 128 位（IPv6）Internet 地址和 16 位端口号。<sup>1</sup>

### 2.4.2 IPv4 地址

用于 TCP/IP 套接字地址的 `sockaddr` 结构的特定形式依赖于 IP 版本。对于 IPv4，使用 `sockaddr_in` 结构。

```
struct in_addr {
    uint32_t s_addr;           //Internet address (32 bits)
};

struct sockaddr_in {
    sa_family_t sin_family;   //Internet protocol (AF_INET)
    in_port_t sin_port;       //Address port (16 bits)
    struct in_addr sin_addr;  //IPv4 address (32 bits)
    char sin_zero[8];         //Not used
};
```

可以看到，`sockaddr_in` 结构具有用于端口号和 Internet 地址以及地址族的字段。`sockaddr_in` 只是 `sockaddr` 结构中的数据的更详细的视图，它是为使用 IPv4 的套接字而定制的，理解这一点很重要。因此，我们可以填写 `sockaddr_in` 的字段，然后把它强制转换为 `sockaddr`（事实上是把指向 `sockaddr_in` 的指针强制转换为指向 `sockaddr` 的指针），并把它传

---

<sup>1</sup> 机敏的读者可能注意到泛型 `sockaddr` 结构并不是大得足以存放 16 字节的 IPv6 地址和 2 字节的端口号。稍后我们将处理这个难点。

递给套接字函数，它们将会检查 `sa_family` 字段以获悉实际的类型，然后强制转换回合适的类型。

### 2.4.3 IPv6 地址

对于 IPv6，则使用 `sockaddr_in6` 结构。

```
struct in_addr {
    uint32_t s_addr[16];           //Internet address (128 bits)
};

struct sockaddr_in6 {
    sa_family_t sin6_family;      //Internet protocol (AF_INET6)
    in_port_t sin6_port;          //Address port (16 bits)
    uint32_t sin6_flowinfo;        //Flow information
    struct in6_addr sin6_addr;    //IPv6 address (128 bits)
    uint32_t sin6_scope_id;       //Scope identifier
};
```

除了 `sockaddr_in` 的那些字段之外，`sockaddr_in6` 结构还具有一些额外的字段。这些字段打算用于 IPv6 协议的一些不太常用的能力。本书中（通常）将忽略它们。

与 `sockaddr_in` 一样，必须把 `sockaddr_in6` 强制转换为 `sockaddr`（事实上是把指向 `sockaddr_in6` 的指针强制转换为指向 `sockaddr` 的指针），以便把它传递给多个不同的套接字函数。此外，其代码实现使用地址族字段来确定参数的实际类型。

### 2.4.4 通用地址存储器

如果你知道关于在 C 中如何为数据结构分配资源的任何事情，可能已经注意到 `sockaddr` 并不是大得足以存放 `sockaddr_in6`（如果你不知道关于它的任何事情，也不要感到恐惧：第 5 章中将介绍你需要知道的大部分知识）。特别是，如果想为一个地址结构分配资源，但是不知道实际的地址类型（例如，IPv4 或 IPv6），则该怎么办？泛型 `sockaddr` 将不会工作，因为它对于某些地址结构显得太小了。<sup>1</sup>为了解决这个问题，套接字设计者创建了 `sockaddr_storage` 结构，它被保证与任何支持的地址类型一样大。

```
struct sockaddr_storage {
    sa_family_t
    ...
}
```

---

<sup>1</sup> 你可能想知道为什么会是这样。其原因显然必须涉及向后兼容性：Sockets API 最初是在很长时间以前指定的，它是在 IPv6 之前出现的，当时资源短缺，没有理由具有较大的结构。如果现在改变它以使得它变得更大，显然会破坏与某些应用程序之间的二进制兼容性。

```
//Padding and fields to get correct length and alignment
...
};
```

与 sockaddr一样，我们仍然具有前导地址族字段，以确定地址的实际类型；不过，利用 sockaddr\_storage，就有足够的空间可用于任何地址类型（关于如何实现这一点的提示，请参阅 5.1.6 节中有关 C 编译器如何在内存中布置结构的讨论）。

关于地址还要注意最后一点。在一些平台上，地址结构包含一个额外的字段，用于存储地址结构的长度（以字节为单位）。对于 sockaddr、sockaddr\_in、sockaddr\_in6 和 sockaddr\_storage，这个额外的字段分别被命名为 sa\_len、sin\_len、sin6\_len 和 ss\_len。由于长度字段并非在所有系统中都是可用的，因此要避免使用它。通常，使用这种形式的结构的平台都会定义一个值（例如，sin6\_len），可以在编译时测试它，以便查看长度字段是否存在。

#### 2.4.5 二进制/字符串地址转换

为了使套接字函数理解地址，它们必须具有“数字”（即二进制）形式；不过，人类使用的地址一般是“可打印的”字符串（例如，192.168.1.1 或 1::1）。可以使用 inet\_pton() 函数（pton= printable to numeric）把地址从可打印的字符串转换为数字：

---

```
int inet_pton(int addressFamily, const char *src, void *dst)
```

---

第一个参数 addressFamily 指定要转换的地址的地址族。回忆可知：Sockets API 为大量通信领域提供了通用接口。不过，我们在本书中只对 IPv4 (AF\_INET) 和 IPv6 (AF\_INET6) 感兴趣。src 参数引用一个 null 终止的字符串，其中包含要转换的地址。dst 参数指向调用者的空间中的内存块以存放结果；它的长度必须足以存放结果（至少 4 个字节用于 IPv4 以及 16 个字节用于 IPv6）。如果转换成功，inet\_pton() 就会返回 1，并在网络字节顺序中返回 dst 引用的地址；如果 src 指向的字符串未被格式化为有效的地址，就会返回 0；如果指定的地址族未知，则会返回 -1。

我们可以采取另一种方式，使用 inet\_ntop() (ntop = numeric to printable) 把地址从数字转换为可打印的形式：

---

```
const char *inet_ntop(int addressFamily, const void *src, char *dst,
socklen_t dstBytes)
```

---

第一个参数 addressFamily 指定要转换的地址的类型。第二个参数 src 指向包含要转换的数字地址的内存块的第一个字节。块的大小由地址族确定。dst 参数指向在调用者的空间中分配的缓冲区（内存块），将把得到的字符串复制到其中；它的大小由 dstBytes 给定。怎

样知道使内存块的大小为多大呢？系统定义的常量 INET\_ADDRSTRLEN（用于 IPv4）和 INET6\_ADDRSTRLEN（用于 IPv6）指示可能最长的结果字符串（以字节为单位）。inet\_ntop() 返回一个指向字符串的指针，如果转换成功，该字符串中就包含可打印的地址（即第三个参数）；否则，就包含 NULL。

#### 2.4.6 获取套接字的关联地址

系统把本地和外部地址与每个连接的套接字（TCP 或 UDP）相关联。以后我们将讨论如何分配这些值的详细信息。可以为本地地址使用 getsockname() 以及为外部地址使用 getpeername() 来查明这些地址。这两个方法都返回一个 sockaddr 结构，其中包含 Internet 地址和端口信息。

---

```
int getpeername(int socket, struct sockaddr *remoteAddress,
                socklen_t *addressLength)
int getsockname(int socket, struct sockaddr *localAddress,
                socklen_t *addressLength)
```

---

socket 参数是我们想要获得其地址信息的套接字的描述符。remoteAddress 和 localAddress 参数指向将由实现把地址信息存放在其中的地址结构；它们总会被调用者强制转换为 sockaddr \*。如果我们事先不知道 IP 协议版本，就应该传入一个 sockaddr\_storage（事实上是指向它的指针）以接收结果。与其他使用 sockaddr 的套接字调用一样，addressLength 是一个输入/输出型参数，它以字节为单位指定了缓冲区的长度（输入）和返回的地址结构（输出）。

### 2.5 连接套接字

在可以通过一个 TCP 套接字发送任何数据之前，必须把它连接到另一个套接字。在这种意义上，使用 TCP 套接字类似于使用电话网络。在可以通话前，必须指定要想通话的电话号码，并且必须建立连接；如果不能建立连接，以后必须再次尝试。连接建立过程在客户与服务器之间有着重大的区别：客户发起连接，而服务器则被动地等待客户连接它（关于连接过程以及它如何与 API 函数相关联的额外详细信息，参见 7.4 节）。要建立与服务器的连接，可以在套接字上调用 connect()。

---

```
int connect(int socket, const struct sockaddr *foreignAddress,
            socklen_t addressLength)
```

---

第一个参数 socket 是由 socket() 创建的描述符。foreignAddress 被声明为一个指向

`sockaddr` 的指针，因为 Sockets API 是通用的；出于我们的目的，它将总是一个指针，指向包含服务器的 Internet 地址和端口的 `sockaddr_in` 或 `sockaddr_in6`。`addressLength` 指定地址结构的长度，通常被给出为 `sizeof(struct sockaddr_in)` 或 `sizeof(struct sockaddr_in6)`。当 `connect()` 返回时，就会连接套接字，并且可以通过调用 `send()` 和 `recv()` 继续进行通信。

## 2.6 绑定到地址

如前所述，客户和服务器在服务器的地址和端口处“聚会”。为了使之工作，服务器必须首先与该地址和端口相关联。这是使用 `bind()` 完成的。另请注意：客户把服务器的地址提供给 `connect()`，但是服务器必须把它自己的地址指定给 `bind()`。客户和服务器应用程序都不需要知道客户的地址即可使它们通信（当然，出于日志或其他目的，服务器可能希望知道客户的地址）。

---

```
int bind(int socket, struct sockaddr *localAddress, socklen_t addressSize)
```

---

第一个参数是以前调用 `socket()` 返回的描述符。与 `connect()` 一样，地址参数被声明为一个指向 `sockaddr` 的指针，但是对于 TCP/IP 应用程序，它将总是指向 `sockaddr_in`（用于 IPv4）或 `sockaddr_in6`（用于 IPv6），其中包含本地接口的 Internet 地址和要侦听的端口。`addressSize` 参数是地址结构的大小。如果成功，`bind()` 就返回 0；如果失败，则返回 -1。

认识到程序不可能把套接字绑定到任意 Internet 地址很重要——如果给定了（任何一种类型的）特定的 Internet 地址，仅当把地址分配给运行程序的主机时，调用才会成功。具有多个 Internet 地址的主机上的服务器可能绑定到一个特定的地址，因为它只希望接受到达该地址的连接。不过，通常服务器希望接受发送到任何主机地址的连接，并且希望把 `sockaddr` 的地址部分设置为“通配符”地址 `INADDR_ANY`（用于 IPv4）或 `IN6ADDR_ANY`（用于 IPv6）。通配符地址的语义是：它匹配任何特定的地址。对于服务器，这意味着它将接收寻址到（指定类型的）任何主机地址的连接。

虽然 `bind()` 通常由服务器使用，但是客户也可以使用 `bind()` 来指定它的本地地址/端口。对于那些没有利用 `bind()` 选择它们自己的本地地址/端口的 TCP 客户，在调用 `connect()` 期间确定本地 Internet 地址和端口。因此，如果客户将要使用 `connect()`，则必须在调用它之前先调用 `bind()`。

可以利用 `IN6ADDR_ANY_INIT` 把 `in6_addr` 结构初始化为通配符地址；不过，这个特殊的常量只能被用作声明中的“初始化器”。一定要注意：虽然 `inaddr_any` 被定义成位于主机字节顺序中，因此在把它用作 `bind()` 的参数之前必须先转换为网络字节顺序，但是 `in6addr_any` 和 `in6addr_any_init` 已经位于网络字节顺序中。

最后，如果把端口号 0 提供给 bind()，系统将为你选择未使用的本地端口。

## 2.7 处理进入的连接

在绑定后，服务器套接字就具有一个地址（或者至少具有一个端口）。需要执行的另一个步骤是指示底层协议实现侦听来自客户的连接；这是通过在套接字上调用 listen() 完成的。

---

```
int listen(int socket, int queueLimit)
```

---

listen() 函数导致内部状态改变为给定的套接字，使得将会处理进入的 TCP 连接请求，然后对它们进行排队，以便程序可以接受它们（7.4 节介绍了关于 TCP 连接的生命周期的更多详细信息）。queueLimit 参数指定了可以在任意时间等待的进入连接数量的上限。queueLimit 的精确效果与系统非常相关，因此要查阅本地系统的技术规范。<sup>1</sup> 如果成功，listen() 就会返回 0；如果失败，则会返回 -1。

一旦把套接字配置为侦听，程序就可以开始接受其上的客户连接。首先，服务器现在似乎应该等待它设置的套接字上的连接，通过该套接字进行发送和接收，关闭它，然后重复这个过程。不过，它的工作方式并不是这样。已经被绑定到端口并且标记为“侦听”的套接字实际上从来不会用于发送和接收。它被代之以用作获取新套接字的方式，其中每个新套接字用于一条客户连接；服务器然后在新套接字上执行发送和接收。服务器通过调用 accept() 获取用于进入的客户连接的套接字。

---

```
int accept(int socket, struct sockaddr *clientAddress,
           socklen_t *addressLength)
```

---

这个函数为套接字使队列中的下一条连接出队。如果队列为空，accept() 就会阻塞，直至一个连接请求到达。当成功时，accept() 就会利用连接的另一端的客户的地址和端口填充由 clientAddress 指向的 sockaddr 结构。一旦调用，addressLength 参数应该指定由 clientAddress 指向的结构的大小（即可用的空间）；一旦返回，它就会包含返回的实际地址的大小。初学者常犯的一个错误是：无法初始化 addressLength 指向的整数，因此它将包含 clientAddress 指向的结构的长度。下面显示了正确的方式：

```
struct sockaddr_storage address;
socklen_t addrLength = sizeof(address);
int newConnection = accept(sock, &address, &addrLength);
```

---

<sup>1</sup> 关于使用“参考手册”的更多信息，参见“前言”。

如果成功，`accept()`就会返回连接到客户的新套接字的描述符。作为第一个参数传递给 `accept()` 的套接字没有改变（未连接到客户），并且会继续侦听新的连接请求。一旦失败，`accept()` 就会返回 -1。在大多数系统上，仅当传递了一个错误的套接字描述符时，`accept()` 才会失败。不过，在一些平台上，如果新套接字在创建后并在被接受前经历了网络级错误，那么它可能会返回一个错误。

## 2.8 通信

一旦“连接”套接字，就可以开始发送和接收数据。如我们所看到的，客户通过调用 `connect()` 创建一个连接的套接字，并由服务器上的 `accept()` 返回一个连接的套接字。在连接后，客户与服务器之间的区别实际上会消失，至少就 Sockets API 而言是这样。通过连接的 TCP 套接字，可以使用 `send()` 和 `recv()` 进行通信。

---

```
ssize_t send(int socket, const void *msg, size_t msgLength, int flags)
ssize_t recv(int socket, void *rcvBuffer, size_t bufferLength, int flags)
```

---

这些函数具有非常相似的参数。第一个参数是通过其发送或接收数据的连接的套接字的描述符。对于 `send()`，`msg` 指向要发送的字节序列，`msgLength` 是要发送的字节数。`send()` 的默认行为是阻塞到发送了所有数据为止（在 6.3 节和第 7 章中将再次讨论这种行为）。对于 `recv()`，`rcvBuffer` 指向缓冲区——即内存中像字符数组这样的区域——其中将用于存放接收到的数据，并且 `bufferLength` 给出了缓冲区的长度，它是可以一次接收的最大字节数。`recv()` 的默认行为是阻塞到至少传输了一些字节为止（在大多数系统上，将导致 `recv()` 的调用者解除阻塞的最小数据量是 1 字节）。

`send()` 和 `recv()` 中的 `flags` 参数提供了一种方式，用于改变套接字调用的默认行为的某些方面。把 `flags` 设置为 0 用于指定默认的行为。`send()` 和 `recv()` 用于返回发送或接收的字节数；如果失败，则返回 -1（另请参见 6.3 节）。

记住：TCP 是一种字节流协议，因此不会保留 `send()` 边界。在接收者上调用 `recv()` 一次所读取的字节数不一定由调用 `send()` 一次所写入的字节数确定。如果利用 3000 字节调用 `send()`，可能需要多次调用 `recv()` 来获取全部 3000 字节，即使给每个 `recv()` 调用传递 5000 字节的缓冲区也是如此。如果利用 100 字节调用 `send()` 四次，可能通过调用 `recv()` 一次来接收全部 400 字节。在编写 TCP 套接字应用程序时一个常见的错误涉及如下假定：如果利用一个 `send()` 写入所有的数据，则可以利用一个 `recv()` 读取所有这些数据。第 7 章中演示了各种可能的情况。

## 2.9 使用IPv6

迄今为止,我们见过了只适用于IPv4的客户和服务器。如果想使用IPv6,则该怎么办?其中的变化相对很小,实质上涉及使用对应的IPv6地址结构和常量。让我们看看我们的TCP应答服务器的IPv6版本。

### TCPEchoServer6.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>
8 #include "Practical.h"
9
10 static const int MAXPENDING = 5; //Maximum outstanding connection requests
11
12 int main(int argc, char *argv[]) {
13
14     if (argc != 2) //Test for correct number of arguments
15         DieWithUserMessage("Parameter(s)", "<Server Port>");
16
17     in_port_t servPort = atoi(argv[1]); //First arg: local port
18
19     //Create socket for incoming connections
20     int servSock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
21     if (servSock < 0)
22         DieWithSystemMessage("socket() failed");
23
24     //Construct local address structure
25     struct sockaddr_in6 servAddr; //Local address
26     memset(&servAddr, 0, sizeof(servAddr)); //Zero out structure
27     servAddr.sin6_family = AF_INET6; //IPv6 address family
28     servAddr.sin6_addr = in6addr_any; //Any incoming interface
29     servAddr.sin6_port = htons(servPort); //Local port
30
31     //Bind to the local address
32     if(bind(servSock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
33         DieWithSystemMessage("bind() failed");
34
```

```

35 //Mark the socket so it will listen for incoming connections
36 if (listen(servSock, MAXPENDING) < 0)
37     DieWithSystemMessage("listen() failed");
38
39 for (;;) {                                //Run forever
40     struct sockaddr_in6 clntAddr; //Client address
41     //Set length of client address structure (in-out parameter)
42     socklen_t clntAddrLen = sizeof(clntAddr);
43
44 //Wait for a client to connect
45 int clntSock=accept(servSock, (struct sockaddr*)&clntAddr, &clntAddrLen);
46 if (clntSock < 0)
47     DieWithSystemMessage("accept() failed");
48
49 //clntSock is connected to a client!
50
51 char clntName[INET6_ADDRSTRLEN];//Array to contain client address string
52 if (inet_ntop(AF_INET6, &clntAddr.sin6_addr.s6_addr, clntName,
53     sizeof(clntName)) != NULL)
54     printf("Handling client %s/%d\n", clntName, ntohs(clntAddr.sin6_port));
55 else
56     puts("Unable to get client address");
57
58 HandleTCPClient(clntSock);
59 }
60 //NOT REACHED
61 }

```

(1) 套接字创建: 第 19~22 行。

通过将通信领域指定为 AF\_INET6, 构造一个 IPv6 套接字。

(2) 填充本地地址: 第 24~29 行。

对于本地地址, 使用 IPv6 (struct sockaddr\_in6) 地址结构和常量 (AF\_INET6 和 IN6ADDR\_ANY)。一个细微的区别是: 不必像对 INADDR\_ANY 所做的那样, 把 IN6ADDR\_ANY 转换为网络字节顺序。

(3) 报告被连接客户: 第 51~56 行。

clntAddr (它包含连接客户的地址) 被声明为 IPv6 套接字地址结构。当把数字地址表示转换为字符串时, 最大字符串长度现在是 INET6\_ADDRSTRLEN。最后, 对 inet\_ntop() 的调用使用一个 IPv6 地址。

你现在已经见过了特定于 IPv4 和 IPv6 的客户和服务器。在第 3 章中, 将会看到如何建立它们, 以便适合于处理任何一类地址。

## 练习题

1. 使用 telnet 试验本书的 TCP 应答服务器。你使用的操作系统是什么？服务器看上去是在你输入时（逐字符）应答，还是仅当你输入完一行后才应答？
2. 使用 telnet 在端口 80 上联系你最喜爱的 Web 服务器，并获取默认的页面。通常可以通过向 Web 服务器发送字符串“GET /”来执行此任务。报告服务器地址/名称，以及默认页面中的文本。
3. 对于 TCPEchoServer.c，我们使用 bind()给套接字显式提供了一个地址。我们说过，套接字必须具有一个地址以进行通信，而我们还没有在 TCPEchoClient.c 中执行 bind()。如何给应答客户的套接字提供一个本地地址？
4. 修改客户和服务器，使得服务器首先“讲话”，发送一条问候消息，并且客户在发送任何信息之前会等待，直至它接收到问候消息为止。在客户与服务器之间需要就什么达成一致意见？
5. 服务器被期望不停机地运行一段较长的时间。因此，它们必须被设计成提供良好的服务，而不管客户做什么。检查示例 TCPEchoServer.c，并列出你可能想到的任何事情，客户可能通过做这些事情导致服务器给其他客户提供低劣的服务。建议一些改进方法，以修正你发现的问题。
6. 使用 getsockname()和 getpeername()，修改 TCPEchoClient4.c，紧接在 connect()之后打印本地和外部地址。
7. 当在未连接的 TCP 套接字上调用 getpeername()时会发生什么事情？
8. 使用 getsockname()和 getpeername()，修改 TCPEchoServer4.c，在 bind()之前和之后打印服务器套接字的本地和外部地址，以及在通过 accept()返回客户套接字之后打印其本地和外部地址。
9. 修改 TCPEchoClient4.c 以使用 bind()，使得系统同时选择地址和端口。
10. 修改 TCPEchoClient4.c，使得新版本绑定到特定的本地地址和系统选择的端口上。如果本地地址改变或者把程序移到具有不同本地地址的主机上，你认为会发生什么事情？
11. 当你尝试在调用 connect()之后进行绑定时会发生什么事情？
12. 为什么套接字接口使用特殊的套接字接受连接？换句话说，如果让服务器创建套接字，使用 bind()和 listen()设置它，建立连接，通过该套接字发送和接收数据，关闭它，然后重复这个过程，那么会发生什么错误（提示：考虑当服务器关闭了上一个连接之后在到达的连接请求上所发生的事情）？

# 第 3 章

## 关于名称和地址族

此时，你已经知道了用于构建可以工作的 TCP 客户和服务器的足够多的知识。不过，迄今为止，我们的示例虽然足够有用，但是还有两个特性可以改进。第一，指定目的地的唯一方式是利用 IP 地址，比如 169.1.1.1 或 FE80:1034::2A97:1001:1。这对于大多数人来说有点痛苦，他们并不擅长处理较长的必须适当格式化的数字串（让我们面对这个问题）。这就是为什么除了 Internet 地址之外，大多数应用程序还允许使用像 www.mkp.com 和 server.example.com 这样的名称指定目的地的原因。但是如我们所看到的，Sockets API 只接受数字参数，因此应用程序需要采用一种方式把名称转换为需要的数字形式。

关于迄今为止我们的示例的另一个问题是：是选择使用 IPv4 还是使用 IPv6 绑定到代码中——我们见过的每个程序都只处理 IP 协议的一个版本。这些程序是有意这样做的——以便保持事情简单。但是如果我们可以对余下的代码隐藏这种选择，从而让参数决定是创建用于 IPv4 还是用于 IPv6 的套接字，岂不是更好吗？

事实证明：API 对这两个问题（以及更多的问题）都提供了解决方案！在本章中，我们将看到：(1) 如何访问名称服务以在名称与数值之间转换；(2) 如何编写代码以便在运行时在 IPv4 和 IPv6 之间进行选择。

### 3.1 将名称映射到数字

利用点或冒号分隔的数字串标识端点对用户不是非常友好，但是这不是喜欢名称胜过地址的唯一原因。另一个原因是：主机的 Internet 地址被绑定到它连接到的网络的某个部分。这是不灵活性的根源：如果主机移到另一个网络或者改变 Internet 服务提供商 (ISP)，它的 Internet 地址一般不得不改变。这样，必须向通过该地址引用主机的每个人通知所做的改变，否则他们将不能访问主机！在编写本书时，本书出版商 (Morgan Kaufmann) 的 Web 服务器的 Internet 地址是 129.35.69.7。不过，我们总是把该 Web 服务器称为 www.mkp.com。显然，与 129.35.69.7 相比，www.mkp.com 更容易记住。事实上，你通常最有可能考虑通

过名称指定 Internet 上的主机。此外，如果 Morgan Kaufmann 的 Web 服务器由于某种原因改变了它的 Internet 地址（例如，新的 ISP、服务器移到另一台机器上），只需简单地把 www.mkp.com 的映射从 129.35.69.7 更改为新的 Internet 地址，就允许对使用名称标识 Web 服务器的所有程序透明地执行改变。<sup>1</sup>

为了解决这些问题，Sockets API 的大多数实现允许访问把名称映射到其他信息（包括 Internet 地址）的名称服务（name service）。你已经看见过映射到 Internet 地址的名称（www.mkp.com）。用于服务（例如，应答）的名称也可映射到端口号。把名称映射到数值（地址或端口号）的过程被称为解析（resolution）。可以用许多方式把名称解析为二进制数值；你的系统可能允许访问其中的多种方式。其中一些方式涉及在后台与其他系统交互；另外一些方式则严格地是在本地进行的。

名称服务并不是使 TCP/IP 工作所必需的，记住这一点是至关重要的。由于上述原因，名称只是提供了一个间接层次。主机命名服务可以访问广泛来源的信息。两个主要的来源是 DNS（Domain Name System，域名系统）和本地配置数据库。DNS [8] 是一种分布式数据库，用于把像 www.mkp.com 这样的域名（domain name）映射到 Internet 地址及其他信息；DNS 协议[9] 允许连接到 Internet 的主机使用 TCP 或 UDP 从该数据库中检索信息。本地配置数据库一般是用于名称-Internet 地址映射的特定于操作系统的机制。幸运的是，对于程序员来说，名称服务的实现细节隐藏在 API 后面，因此我们需要知道的唯一一件事是：如何请求它解析（resolve）名称。

### 3.1.1 访问名称服务

用于名称服务的首选接口是通过 getaddrinfo() 函数<sup>2</sup>：

---

```
int getaddrinfo (const char *hostStr, const char *serviceStr,
                 const struct addrinfo *hints, struct addrinfo **results)
```

---

getaddrinfo() 的前两个参数指向 null 终止的字符串，它们分别表示主机名称或地址以及服务名称或端口号。第三个参数描述了要返回的信息种类；我们将在下面讨论它。最后一个参数是 struct addrinfo 指针的位置，其中将存储一个指向包含结果的链表的指针。getaddrinfo() 的返回值指示解析是成功（0）还是不成功（非 0 错误代码）。

使用 getaddrinfo() 还需要使用另外两个辅助函数：

---

1 在我们编写本书第 1 版时，Morgan Kaufmann 的 Web 服务器的地址实际上是 208.164.121.48。它们大概改变了它们的地址，以帮助我们说明这一点。

2 从历史上讲，其他函数也可用于此目的，并且许多应用程序仍在使用它们。不过，它们具有几个缺点，并且自从 POSIX 2001 标准起就逐步停止使用它们。

```
void freeaddrinfo(struct addrinfo *addrList)
const char *gai_strerror(int errorCode)
```

getaddrinfo()创建结果的动态分配的链表，必须在调用者完成链表后释放它。给定指向结果链表头部的指针，freeaddrinfo()将会释放为该链表分配的所有存储空间。如果不能调用这个方法，则可能会导致有害的内存泄漏。**仅当利用返回的信息完成了程序时，才应该调用这个方法；在这个函数返回之后，结果链表中不包含任何信息是可靠的。**万一 getaddrinfo()返回一个非 0（错误）值，把它传递给 gai\_strerror()将生成一个字符串，它描述了出错的是什么。

一般来讲，getaddrinfo()接受主机/服务对的名称作为输入，并返回一个结构的链表，其中包含创建套接字以连接到命名的主机/服务所需的所有信息，包括：地址/协议族（v4 或 v6）、套接字类型（例如，流或数据报）、协议（用于 Internet 协议族的 TCP 或 UDP），以及数字型套接字地址。链表中的每个条目都存放在 addrinfo 结构中，如下所示：

```
struct addrinfo {
    int ai_flags;           //Flags to control info resolution
    int ai_family;          //Family: AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;        //Socket type: SOCK_STREAM, SOCK_DGRAM
    int ai_protocol;        //Protocol: 0 (default) or IPPROTO_XXX
    socklen_t ai_addrlen;   //Length of socket address ai_addr
    struct sockaddr *ai_addr; //Socket address for socket
    char *ai_canonname;     //Canonical name
    struct addrinfo *ai_next; //Next addrinfo in linked list
};
```

ai\_sockaddr 字段包含一个合适类型的 sockaddr，并利用（数字）地址和端口信息填充它。应该可以明显看出哪个字段包含地址族、套接字类型和协议信息（结果中未使用标志字段；我们稍后将讨论它的使用）。实际上，结果返回在指向 addrinfo 结构的链表的指针中；ai\_next 字段包含用于该链表的指针。

为什么使用一个链表呢？这有两个原因。第一，对于主机和服务的每种组合，可能有地址族（v4 或 v6）和套接字类型/协议（流/TCP 或 数据报/UDP）的多种不同的组合表示可能的端点。例如，主机“server.example.net”可能具有在 IPv4/TCP 和 IPv6/UDP 上的端口 1001 上进行“垃圾邮件”服务侦听的多个实例。getaddrinfo()函数可以返回这二者。第二个原因是主机名可以映射到多个 IP 地址；getaddrinfo()有助于返回所有这些地址。

因此，getaddrinfo()会为给定的主机名/服务对返回所有切实可行的组合。但是请等一等，如果你不需要选择并且事先确切知道想要什么，则该怎么办？你不希望必须编写代码为特定的组合（比如，IPv4/TCP）搜索返回的列表。此时，getaddrinfo()的第三个参数就派上用场了！它允许你告诉系统为你过滤结果。在示例程序 GetAddrInfo.c 中将看到它的使用。

## GetAddrInfo.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <netdb.h>
5 #include "Practical.h"
6
7 int main(int argc, char *argv[]) {
8
9     if (argc != 3)           //Test for correct number of arguments
10        DieWithUserMessage("Parameter(s)", "<Address/Name> <Port/Service>");
11
12    char *addrString = argv[1];    //Server address/name
13    char *portString = argv[2];    //Server port/service
14
15    //Tell the system what kind(s) of address info we want
16    struct addrinfo addrCriteria; //Criteria for address match
17    memset(&addrCriteria, 0, sizeof(addrCriteria));      //Zero out structure
18    addrCriteria.ai_family = AF_UNSPEC;                  //Any address family
19    addrCriteria.ai_socktype = SOCK_STREAM;              //Only stream sockets
20    addrCriteria.ai_protocol = IPPROTO_TCP;             //Only TCP protocol
21
22    //Get address(es) associated with the specified name/service
23    struct addrinfo *addrList;   //Holder for list of addresses returned
24    //Modify servAddr contents to reference linked list of addresses
25    int rtnVal=getaddrinfo(addrString,portString,&addrCriteria,&addrList);
26    if (rtnVal != 0)
27        DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
28
29    //Display returned addresses
30    for (struct addrinfo *addr = addrList; addr != NULL; addr = addr->ai_next){
31        PrintSocketAddress(addr->ai_addr, stdout);
32        fputc('\n', stdout);
33    }
34
35    freeaddrinfo(addrList); //Free addrinfo allocated in getaddrinfo()
36
37    exit(0);
38 }
```

- (1) 应用程序建立和参数解析：第9~13行。  
(2) 构造地址规范：第15~20行。

`addrCriteria` 结构将指示我们对哪种结果感兴趣。

- 声明和初始化 `addrinfo` 结构：第 16~17 行。
- 设置地址族：第 18 行。

我们把地址族设置为 `AF_UNSPEC`，它允许返回的地址来自于任何地址族（包括 `AF_INET` 和 `AF_INET6`）。

- 设置套接字类型：第 19 行。

我们想要流/TCP 端点，因此把它设置为 `SOCK_STREAM`。系统将过滤掉使用不同协议的结果。

- 设置协议：第 20 行

我们想要 TCP 套接字，因此把它设置为 `IPPROTO_TCP`。由于 TCP 是流套接字的默认协议，因此保持这个字段为 0 将具有相同的结果。

(3) 获取地址信息：第 22~27 行。

- 声明用于结果链表头部的指针：第 23 行。
- 调用 `getaddrinfo()`：第 25 行。

我们传递想要的主机名、端口以及 `addrCriteria` 结构中编码的约束条件。

- 检查返回值：第 26~27 行。

如果成功，`getaddrinfo()` 就返回 0。否则，返回值将指示具体的错误。辅助函数 `gai_strerror()` 返回一条字符串错误消息，解释给定的错误返回值。注意：这些消息不同于正常的基于 `errno` 的消息。

(4) 打印地址：第 29~33 行。

迭代地址的链表，并把每个地址打印到控制台。函数 `PrintSocketAddress()` 接受一个要打印的地址以及要在其上打印的流。我们展示了它的代码，它在本章后面的 `AddressUtility.c` 中。

(5) 释放地址链表：第 35 行。

系统为它返回的 `addrinfo` 结构的链表分配存储空间。在完成它时，我们必须调用辅助函数 `freeaddrinfo()` 来释放该内存。

程序 `GetAddrInfo.c` 接受两个命令行参数：主机名称（或地址）和服务名称（或端口号），并打印 `getaddrinfo()` 返回的地址信息。假设你想查找名为“localhost”的主机（即你正在运行的主机）上的名为“whois”的服务的地址，下面给出了其使用方法：

```
% GetAddrInfo localhost whois
127.0.0.1-43
```

要查找主机“pine.netlab.uky.edu”上的服务“whois”，方法如下：

```
% GetAddrInfo pine.uky.edu whois
```

128.163.170.219-43

程序可以处理名称和数字参数的任意组合：

```
% GetAddrInfo 169.1.1.100 time
169.1.1.100-37
% GetAddrInfo FE80:0000:0000:0000:0000:ABCD:0001:0002:0003 12345
fe80::abcd:1:2:3-12345
```

这些示例全都返回一个答案。但是如上所述，一些名称具有多个与它们关联的数字地址。例如，“google.com”通常与众多 Internet 地址相关联。这允许将服务（例如，搜索引擎）放在多台主机上。为什么这样做呢？一个原因是健壮性。如果任何一台主机失败，服务可以继续运行，因为客户可以使用任何一台提供服务的主机。另一个优点是可伸缩性。如果客户随机地选择要使用的数字地址（以及相应的主机），就可以把负载分散在多个服务器上。好消息是 `getaddrinfo()` 可以返回一个名称映射到的全部地址。可以通过利用流行的 Web 站点的名称执行程序来试验它（注意：为第二个参数提供 0 将导致只会打印地址信息）。

### 3.1.2 详细信息

如前所述，`getaddrinfo()` 是一个“瑞士军刀”类型的函数。我们将在这里介绍它的能力中的一些难以理解的地方。初学者可能希望跳过本节内容，并在以后再回过头来学习它。

第三个参数（`addrinfo` 结构）告诉系统调用者对哪一类端点感兴趣。在 `GetAddrInfo.c` 中，我们把这个参数设置成指示任何地址族都是可接受的，并且我们想要一个流/TCP 套接字。我们可以代之以指定数据报/UDP 或特定的地址族（比如，`AF_INET6`），或者可以把 `ai_socktype` 和 `ai_protocol` 字段设置为 0，指示我们希望接收所有可能的数据。甚至可能为第三个参数传递一个 `NULL` 指针；系统被期望会处理这种情况，就好像在把 `ai_family` 设置为 `AF_UNSPEC` 并且把其他所有的字段都设置为 0 的情况下传递了 `addrinfo` 结构。

第三个参数中的 `ai_flags` 字段提供了对 `getaddrinfo()` 的行为的额外控制。它是一个整数，其中的各个位都会被系统解释为布尔变量。下面给出了每个标志的含义；可以使用位“或”运算符“|”组合标志（参见 5.1.8 节，了解如何执行该操作）。

- `AI_PASSIVE`: 如果在设置这个标志时 `hostStr` 是 `NULL`，任何返回的 `addrinfos` 都将把它们的地址设置为合适的“任意”地址常量——`INADDR_ANY`（IPv4）或 `IN6ADDR_ANY_INIT`（IPv6）。
- `AI_CANONNAME`: 就像一个名称可以解析成许多数字地址一样，也可以把多个名称解析成同一个 IP 地址。不过，一个名称通常被定义为正式（“规范”）名称。通过在 `ai_flags` 中设置这个标志，我们指示 `getaddrinfo()` 在链表的第一个 `struct addrinfo` 的 `ai_canonname` 字段中返回一个指向规范名称（如果它存在的话）的指针。

- AI\_NUMERICHOST：如果 hostStr 没有指向具有有效的数字地址格式的字符串，这个标志将导致返回一个错误。在不使用这个标志的情况下，如果 hostStr 参数指向数字地址的无效的字符串表示，就会尝试通过名称系统解析它；这可能在名称服务的无用查询上浪费时间和带宽。如果设置了这个标志，就会利用 inet\_pton()简单地转换并返回给定的有效地址字符串。
- AI\_ADDRCONFIG：如果设置了这个标志，仅当系统具有为特定的地址族配置的接口时，getaddrinfo()才会返回该地址族的地址。因此，仅当系统具有带有 IPv4 地址的接口时，才会返回 IPv4 地址，对于 IPv6 同样如此。
- AI\_V4MAPPED：如果 ai\_family 字段包含 AF\_INET6，并且没有找到匹配的 IPv6 地址，那么 getaddrinfo()就会返回 IPv4 映射的 IPv6 地址。这种技术可用于在仅支持 IPv4 的主机和 IPv6 主机之间提供有限的互操作性。

## 3.2 编写地址通用的代码

一位著名的诗人曾经写下过这样一句话：“To v6 or not to v6, that is the question（转换或者不转换到 v6，这是一个问题）。”幸运的是，Socket 接口允许我们延迟到执行时再回答这个问题。在我们以前的 TCP 客户和服务器示例中，我们使用 AF\_INET 或 AF\_INET6 给套接字创建和地址字符串转换函数指定了特定的 IP 协议版本。不过，getaddrinfo()允许我们编写处理地址族的代码，而不必为每个版本重复执行这些步骤。在本节中，我们将使用它的能力修改特定于版本的客户和服务器代码，以使它们成为通用的。

在执行该任务之前，介绍了一个方便的小方法，它用于打印（任何一种类型的）套接字地址。给定一个包含 IPv4 或 IPv6 地址的 sockaddr 结构，它把地址打印到给定的输出流，并为其地址族使用正确的格式。给定任何其他类型的地址，它将打印一个错误字符串。这个函数接受一个泛型 struct sockaddr 指针，并把地址打印到指定的流。可以在 AddressUtility.c 中找到 PrintSocketAddress() 的实现，Practical.h 中包括有函数原型。

### PrintSocketAddress()

```

1 void PrintSocketAddress(const struct sockaddr *address, FILE *stream) {
2     //Test for address and stream
3     if (address == NULL || stream == NULL)
4         return;
5
6     void *numericAddress;    //Pointer to binary address
7     //Buffer to contain result (IPv6 sufficient to hold IPv4)
8     char addrBuffer[INET6_ADDRSTRLEN];
9     in_port_t port;          //Port to print

```

```

10 //Set pointer to address based on address family
11 switch (address->sa_family) {
12 case AF_INET:
13     numericAddress = &((struct sockaddr_in *) address)->sin_addr;
14     port = ntohs(((struct sockaddr_in *) address)->sin_port);
15     break;
16 case AF_INET6:
17     numericAddress = &((struct sockaddr_in6 *) address)->sin6_addr;
18     port = ntohs(((struct sockaddr_in6 *) address)->sin6_port);
19     break;
20 default:
21     fputs("[unknown type]", stream);      //Unhandled type
22     return;
23 }
24 //Convert binary to printable address
25 if (inet_ntop(address->sa_family, numericAddress, addrBuffer,
26     sizeof(addrBuffer)) == NULL)
27     fputs("[invalid address]", stream); //Unable to convert
28 else {
29     fprintf(stream, "%s", addrBuffer);
30     if (port != 0) //Zero not valid in any socket addr
31         fprintf(stream, "-%u", port);
32 }
33 }

```

### 3.2.1 通用的 TCP 客户

使用 getaddrinfo()，我们可以编写不特定于某个 IP 版本的客户和服务器。让我们首先转换 TCP 客户，使之与版本无关；我们将删除版本号并把它命名为 TCPEchoClient.c。一般策略是设置 getaddrinfo()的参数，使之返回 IPv4 和 IPv6 地址，并使用第一个工作的地址。由于我们的地址搜索功能可能在别的位置是有用的，我们分离出了负责创建和连接客户套接字的代码，并把它放在单独的函数 SetupTCPClientSocket()中，它位于 TCPClientUtility.c 中。这个建立函数接受在字符串中指定的主机和服务，并返回被连接的套接字（如果失败，则返回-1）。主机或服务可能被指定为 NULL。

#### TCPClientUtility.c

```

1 #include <string.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netdb.h>

```

```

6 #include "Practical.h"
7
8 int SetupTCPClientSocket(const char *host, const char *service) {
9     //Tell the system what kind(s) of address info we want
10    struct addrinfo addrCriteria;           //Criteria for address match
11    memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
12    addrCriteria.ai_family = AF_UNSPEC;      //v4 or v6 is OK
13    addrCriteria.ai_socktype = SOCK_STREAM; //Only streaming sockets
14    addrCriteria.ai_protocol = IPPROTO_TCP; //Only TCP protocol
15
16    //Get address(es)
17    struct addrinfo *servAddr; //Holder for returned list of server addrs
18    int rtnVal = getaddrinfo(host, service, &addrCriteria, &servAddr);
19    if (rtnVal != 0)
20        DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
21
22    int sock = -1;
23    for(struct addrinfo *addr = servAddr; addr != NULL; addr = addr->ai_next){
24        //Create a reliable, stream socket using TCP
25        sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
26        if (sock < 0)
27            continue; //Socket creation failed; try next address
28
29        //Establish the connection to the echo server
30        if (connect(sock, addr->ai_addr, addr->ai_addrlen) == 0)
31            break; //Socket connection succeeded; break and return socket
32
33        close(sock); //Socket connection failed; try next address
34        sock = -1;
35    }
36
37    freeaddrinfo(servAddr); //Free addrinfo allocated in getaddrinfo()
38    return sock;
39 }

```

(1) 解析主机和服务: 第 10~20 行。

我们传递给 `getaddrinfo()` 的条件说明我们不关心使用的是哪种协议 (`AF_UNSPEC`), 但是套接字地址用于 TCP (`SOCK_STREAM/IPPROTO_TCP`)。

(2) 尝试从地址列表中创建并连接套接字: 第 22~35 行。

- 创建合适的套接字类型: 第 25~27 行。

`getaddrinfo()` 返回匹配的域 (`AF_INET` 或 `AF_INET6`) 以及套接字类型/协议。在创建新套接字时, 把该信息传递给 `socket()`。如果系统不能创建指定类型的套接字,

就移到下一个地址。

- 连接到指定的服务器：第30~34行。

我们使用从getaddrinfo()获得的地址，尝试连接到服务器。如果连接成功，就退出地址查找循环。如果连接失败，就关闭套接字，并试试下一个地址。

- (3) 释放地址列表：第37行。

为了避免内存泄漏，需要释放由getaddrinfo()创建的地址链表。

- (4) 返回得到的套接字描述符：第38行。

如果我们成功地创建并连接一个套接字，就返回套接字描述符。如果没有地址成功，就返回-1。

现在，我们准备好查看通用的客户。

## TCPEchoClient.c

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netdb.h>
8 #include "Practical.h"
9
10 int main(int argc, char *argv[]) {
11
12     if (argc < 3 || argc > 4)      //Test for correct number of arguments
13         DieWithUserMessage("Parameter(s)",
14             "<Server Address/Name> <Echo Word> [<Server Port/Service>]");
15
16     char *server = argv[1];        //First arg: server address/name
17     char *echoString = argv[2];    //Second arg: string to echo
18     //Third arg (optional): server port/service
19     char *service = (argc == 4) ? argv[3] : "echo";
20
21     //Create a connected TCP socket
22     int sock = SetupTCPClientSocket(server, service);
23     if (sock < 0)
24         DieWithUserMessage("SetupTCPClientSocket() failed", "unable to connect");
25
26     size_t echoStringLen = strlen(echoString); //Determine input length
27
28     //Send the string to the server

```

```

29     ssize_t numBytes = send(sock, echoString, echoStringLen, 0);
30     if (numBytes < 0)
31         DieWithSystemMessage("send() failed");
32     else if (numBytes != echoStringLen)
33         DieWithUserMessage("send()", "sent unexpected number of bytes");
34
35     //Receive the same string back from the server
36     unsigned int totalBytesRcvd = 0;    //Count of total bytes received
37     fputs("Received: ", stdout);        //Setup to print the echoed string
38     while (totalBytesRcvd < echoStringLen) {
39         char buffer[BUFSIZE];           //I/O buffer
40         //Receive up to the buffer size (minus 1 to leave space for
41         //a null terminator) bytes from the sender
42         numBytes = recv(sock, buffer, BUFSIZE - 1, 0);
43         if (numBytes < 0)
44             DieWithSystemMessage("recv() failed");
45         else if (numBytes == 0)
46             DieWithUserMessage("recv()", "connection closed prematurely");
47         totalBytesRcvd += numBytes;       //Keep tally of total bytes
48         buffer[numBytes] = '\0';         //Terminate the string!
49         fputs(buffer, stdout);          //Print the buffer
50     }
51
52     fputc('\n', stdout);              //Print a final linefeed
53
54     close(sock);
55     exit(0);
56 }
```

在套接字创建之后，TCPEchoClient.c 中的余下部分与特定于版本的客户完全相同。关于这段代码必须提及一个警告。在 SetupTCPClientSocket() 的第 25 行中，我们把返回的 addrinfo 结构的 ai\_family 字段作为第一个参数传递给 socket()。严格地讲，这个值标识了一个地址族 (AF\_XXX，而 socket 的第一个参数则指示想要的套接字的协议族 (PF\_XXX))。在我们经历过的所有实现中，这两个族是可互换的——特别是 AF\_INET 和 PF\_INET 被定义为具有相同的值，就像 PF\_INET6 和 AF\_INET6 一样。我们的通用代码依赖于这个事实。作者坚持认为这些定义将不会改变，但是感觉到对这个假设（它允许更简洁的代码）进行彻底的解密很重要。可以非常直观地消除这个假设，我们把它留作练习。

### 3.2.2 通用的 TCP 服务器

我们的与协议无关的 TCP 应答服务器使用了与客户中类似的修改。回忆可知：典型的

服务器绑定到任何可用的本地地址。为了完成这项任务，我们（1）指定了 AI\_PASSIVE 标志并且（2）为主机名指定了 NULL。实际上，这会获取一个适合于传递给 bind() 的地址，包括用于本地 IP 地址的通配符——用于 IPv4 的 INADDR\_ANY 或者用于 IPv6 的 IN6ADDR\_ANY\_INIT。对于同时支持 IPv4 和 IPv6 的系统，getaddrinfo() 一般将先返回 IPv6，因为它为互操作性提供了更多的选项。不过，注意：应该选择哪些选项以最大化连通性的问题依赖于运行服务器的特定环境——从它的名称服务到它的 Internet 服务提供商。我们在里展示的方法实质上尽可能简单，并且不太可能适用于需要跨广泛平台运行的生产服务器。参见下一节内容，了解更多的信息。

就像在我们的与协议无关的客户中一样，我们分离出了在单独的函数 SetupTCPServerSocket() 中建立套接字所涉及的步骤，该函数位于 TCPServerUtility.c 中。这个建立函数迭代从 getaddrinfo() 返回的地址，当它可以成功地绑定并侦听或者当它遍历完地址时，就会停止迭代。

## SetupTCPServerSocket()

```

1 static const int MAXPENDING = 5; //Maximum outstanding connection requests
2
3 int SetupTCPServerSocket(const char *service) {
4     //Construct the server address structure
5     struct addrinfo addrCriteria;           //Criteria for address match
6     memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
7     addrCriteria.ai_family = AF_UNSPEC;      //Any address family
8     addrCriteria.ai_flags = AI_PASSIVE;       //Accept on any address/port
9     addrCriteria.ai_socktype = SOCK_STREAM;   //Only stream sockets
10    addrCriteria.ai_protocol = IPPROTO_TCP;  //Only TCP protocol
11
12    struct addrinfo *servAddr;               //List of server addresses
13    int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
14    if (rtnVal != 0)
15        DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
16
17    int servSock = -1;
18    for(struct addrinfo *addr=servAddr; addr != NULL;addr=addr->ai_next) {
19        //Create a TCP socket
20        servSock = socket(servAddr->ai_family, servAddr->ai_socktype,
21                          servAddr->ai_protocol);
22        if (servSock < 0)
23            continue; //Socket creation failed; try next address
24
25        //Bind to the local address and set socket to list
26        if((bind(servSock, servAddr->ai_addr, servAddr->ai_addrlen) == 0) &&
```

```

27     (listen(servSock, MAXPENDING) == 0)) {
28     //Print local address of socket
29     struct sockaddr_storage localAddr;
30     socklen_t addrSize = sizeof(localAddr);
31     if(getsockname(servSock, (struct sockaddr *)&localAddr, &addrSize)< 0)
32         DieWithSystemMessage("getsockname() failed");
33     fputs("Binding to ", stdout);
34     PrintSocketAddress((struct sockaddr *) &localAddr, stdout);
35     fputc('\n', stdout);
36     break;           //Bind and list successful
37 }
38
39     close(servSock); //Close and try again
40     servSock = -1;
41 }
42
43 //Free address list allocated by getaddrinfo()
44 freeaddrinfo(servAddr);
45
46 return servSock;
47 }

```

我们还分离出了接受客户连接的代码，并把它放在单独的函数 AcceptTCPConnection() 中，该函数位于 TCPServerUtility.c 中。

### **AcceptTCPConnection()**

```

1 int AcceptTCPConnection(int servSock) {
2     struct sockaddr_storage clntAddr; //Client address
3     //Set length of client address structure (in-out parameter)
4     socklen_t clntAddrLen = sizeof(clntAddr);
5
6     //Wait for a client to connect
7     int clntSock=accept(servSock, (struct sockaddr *)&clntAddr,&clntAddrLen);
8     if (clntSock < 0)
9         DieWithSystemMessage("accept() failed");
10
11 //clntSock is connected to a client!
12
13 fputs("Handling client ", stdout);
14 PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
15 fputc('\n', stdout);
16
17 return clntSock;

```

18 }

注意：我们使用 `getsockname()` 打印本地套接字地址。在执行 `TCPEchoServer.c` 时，它将打印通配符本地网络地址。最后，我们在与协议无关的应答服务器中使用了我们的新函数。

## TCPEchoServer.c

```
1 #include <stdio.h>
2 #include "Practical.h"
3 #include <unistd.h>
4
5 int main(int argc, char *argv[]) {
6
7     if (argc != 2) //Test for correct number of arguments
8         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
9
10    char *service = argv[1]; //First arg: local port
11
12    //Create socket for incoming connections
13    int servSock = SetupTCPSServerSocket(service);
14    if (servSock < 0)
15        DieWithUserMessage("SetupTCPSServerSocket() failed", service);
16
17    for (;;) { //Run forever
18        //New connection creates a connected client socket
19        int clntSock = AcceptTCPConnection(servSock);
20
21        HandleTCPClient(clntSock); //Process client
22        close(clntSock);
23    }
24    //NOT REACHED
25 }
```

### 3.2.3 IPv4 与 IPv6 之间互操作

我们的通用客户和服务器忘记了它们是使用 IPv4 还是 IPv6 套接字。一个显而易见的问题是：“如果一方使用 IPv4 而另一方使用 IPv6，则会如何？”答案是：如果（并且仅当）使用 IPv6 的程序是双栈（dual-stack）系统时——即同时支持版本 4 和版本 6，它们应该能够互操作。特殊的“v4-v6 映射”地址类使之成为可能。这种机制允许 IPv6 套接字连接到 IPv4 套接字。关于其隐含意义及其工作方式的详细讨论超出了本书的范围，但是基本思想

是：双栈系统中的 IPv6 实现可以识别 IPv4 地址与 IPv6 套接字之间的通信是想要的，并会把 IPv4 地址转换为“v4-v6 映射”地址。因此，每个套接字都会以它自己的格式处理地址。

例如，如果客户是一个地址为 1.2.3.4 的 v4 套接字，服务器正在侦听双栈平台中的一个 v6 套接字，当连接请求进入时，服务器端的实现将自动执行转换，并告诉服务器它被连接到一个 v6 套接字，它具有 v4 映射的地址::ffff:1.2.3.4（注意：实际情况比这里描述的要复杂一点；特别是，服务器端的实现首先将尝试匹配绑定到 v4 地址的套接字，仅当它无法找到一个匹配时才会执行转换；参见第 7 章，了解更多详细信息）。

如果服务器正在侦听 v4 套接字，客户就会尝试从双栈平台上的 v6 套接字执行连接，并且客户不会在调用 `connect()` 之前把套接字绑定到特定的地址，客户端的实现将会识别它正连接到一个 IPv4 地址，并在调用 `connect()` 时给套接字分配一个 v4 映射的 IPv6 地址。在连接请求发出时，栈将“魔术般地”把分配的地址转换为一个 IPv4 地址。注意：在两种情况下，通过网络传递的消息实际上都是 IPv4 消息。

虽然 v4 映射的地址提供了良好的互操作性措施，但是实际情况是：当考虑到只支持 v4 的主机、只支持 v6 的主机、支持 IPv6 但未配置 IPv6 地址的主机，以及支持 IPv6 并在局域网上使用它但是没有广域 IPv6 传输可用（即它们的提供商不支持 IPv6）的主机这些情况时，可能涉及的范围非常大。尽管我们的示例代码——客户会尝试 `getaddrinfo()` 返回的各种可能的情况，服务器会设置 `AI_PASSIVE` 并绑定到 `getaddrinfo()` 返回的第一个地址——涵盖了最有可能发生的情况，但是需要非常小心地设计生产代码，以最大化客户和服务器在各种条件下彼此能够找到对方的可能性。实现这一点的细节超出了本书的范围；读者应该参阅 RFC 4038，了解更多详细信息。

### 3.3 从数字获取名称

因此我们能够从主机名称获取 Internet 地址，但是我们能够在另一个方向上（从 Internet 地址获取主机名称）执行映射吗？答案是“通常可以”。有一个名为 `getnameinfo()` 的“逆”函数，它接受一个 `sockaddr` 地址结构（实际上是用于 IPv4 的 `struct sockaddr_in` 和用于 IPv6 的 `struct sockaddr_in6`）与地址长度。该函数以 `null` 终止的字符串形式返回一个对应的节点和服务名称——如果在名称系统中存储了数据与名称之间的映射的话。`getnameinfo()` 的调用者必须为节点和服务名称预先分配空间，并传入用于该空间的指针和长度。节点或服务名称的最大长度分别由常量 `NI_MAXHOST` 和 `NI_MAXSERV` 给出。如果调用者为节点和/or 服务指定长度 0，`getnameinfo()` 将不会返回对应的值。如果成功，该函数将返回 0；如果失败，则会返回一个非 0 值。可以再次把非 0 的失败返回代码传递给 `gai_strerror()`，以获取对应的错误文本字符串。

---

```
int getnameinfo (const struct sockaddr *address, socklen_t addressLength,
                 char *node, socklen_t nodeLength, char * service,
                 socklen_t serviceLength, int flags)
```

---

与 `getaddrinfo()`一样，多个标志控制着 `getnameinfo()` 的行为。下面描述了它们；与以前一样，可以使用位“或”运算符（“|”）组合其中一些标志。

- `NI_NOFQDN`: 只返回本地主机的主机名称，而不是 FQDN (Fully Qualified Domain Name, 完全限定的域名) (FQDN 包含所有部分，例如，`protocols.example.com`，而主机名称则只是第一部分，例如“`protocols`”。
- `NI_NUMERICHOST`: 返回地址而不是名称的数字形式。如果你只想使用这种服务来代替 `inet_ntop()`，这潜在地避免了代价高昂的名称服务查找。
- `NI_NUMERICSERV`: 返回服务而不是名称的数字形式。
- `NI_NAMEREQD`: 如果不能为给定的地址找到一个名称，就返回一个错误。如果不使用这个选项，就会返回地址的数字形式。
- `NI_DGRAM`: 指定数据报服务；默认行为假定一种流服务。在某些情况下，服务为 TCP 和 UDP 使用不同的端口号。

如果程序需要它自己的主机名称，则该怎么办？`gethostname()` 接受一个缓冲区和缓冲区长度，并把运行调用程序的主机的名称复制到给定的缓冲区中。

---

```
int gethostname(char *nameBuffer, size_t bufferLength)
```

---

## 练习题

1. `GetAddrInfo.c` 需要两个参数。如果你不知道任何主机名称，怎样使之解析服务名称？
2. 修改 `GetAddrInfo.c`，使之接受可选的第三个参数，其中包含在 `addrinfo` 参数的 `ai_flags` 字段中传递给 `getaddrinfo()` 的“标志”。例如，传递“-n”作为第三个参数应该会导致 `ai_numerichost` 标志被设置。
3. `getnameinfo()` 适用于 IPv6 地址以及 IPv4 吗？当给定地址::1 时，它会返回什么？
4. 修改通用的 `TCPEchoClient` 和 `TCPEchoServer`，消除 3.2.1 节末尾提到的假设。

# 第 4 章

## 使用 UDP 套接字

UDP (User Datagram Protocol, 用户数据报协议) 提供了比 TCP 更简单的端到端服务。事实上, UDP 只执行两种功能: (1) 它向 IP 层添加了另一个寻址 (端口) 层; (2) 它会检测传输中可能发生的数据损坏, 并丢弃任何损坏的数据报。由于这种简单性, UDP (数据报) 套接字具有一些与我们以前见过的 TCP (流) 套接字不同的特征。

例如, UDP 套接字在使用前不必进行连接。TCP 类似于电话通信, 而 UDP 则类似于通过邮件通信: 在发送包裹或信件前不必“连接”, 但是必须为每个包裹或信件指定目的地址。在接收时, UDP 套接字就像是一个邮箱, 可以把来自许多不同来源的信件或包裹放入其中。

UDP 套接字与 TCP 套接字之间的另一个区别在于它们处理消息边界的方式: UDP 套接字会保留它们。与 TCP 套接字相比, 在某些方面这使得接收应用程序消息更简单。在 4.3 节中将进一步讨论它。最后一个区别是 UDP 提供的端到端传输服务是一种“尽力而为”的服务: 不保证通过 UDP 套接字发送的消息将会到达其目的地。这意味着使用 UDP 套接字的程序必须准备处理消息的丢失和重新排序; 我们将在后面看到这方面的一个示例。

同样, 我们通过简单的客户和服务器程序介绍 Sockets API 的 UDP 部分。与以前一样, 它们实现了一个普通的应答协议。然后, 我们将在 4.3 节和 4.4 节中更详细地描述 API 功能。

### 4.1 UDP 客户

我们的 UDP 应答客户在设置服务器地址以及与服务器通信方面看上去类似于我们的与地址族无关的 TCPEchoClient.c。不过, 它不会调用 `connect()`; 它使用 `sendto()` 和 `recvfrom()`, 而不是 `send()` 和 `recv()`; 并且它只需要执行一次接收, 因为 UDP 套接字会保留消息边界, 这与 TCP 的字节流服务不同。当然, UDP 客户只与 UDP 服务器通信。许多系统出于调试和测试的目的而包括了 UDP 应答服务器; 服务器将简单地把它接收到的任何消息发送回它们的任何源发地。在建立后, 我们的应答客户将执行以下步骤: (1) 它把应答字符串发送

给服务器；(2) 它接收应答；(3) 它关闭程序。

## UDPEchoClient.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/socket.h>
6 #include <netdb.h>
7 #include "Practical.h"
8
9 int main(int argc, char *argv[]) {
10
11    if (argc < 3 || argc > 4)      //Test for correct number of arguments
12        DieWithUserMessage("Parameter(s)",
13                           "<Server Address/Name> <Echo Word> [<Server Port/Service>]");
14
15    char *server = argv[1];          //First arg: server address/name
16    char *echoString = argv[2];      //Second arg: word to echo
17
18    size_t echoStringLen = strlen(echoString);
19    if (echoStringLen > MAXSTRINGLENGTH) //Check input length
20        DieWithUserMessage(echoString, "string too long");
21
22    //Third arg (optional): server port/service
23    char *servPort = (argc == 4) ? argv[3] : "echo";
24
25    //Tell the system what kind(s) of address info we want
26    struct addrinfo addrCriteria;      //Criteria for address match
27    memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
28    addrCriteria.ai_family = AF_UNSPEC; //Any address family
29    //For the following fields, a zero value means "don't care"
30    addrCriteria.ai_socktype = SOCK_DGRAM; //Only datagram sockets
31    addrCriteria.ai_protocol = IPPROTO_UDP; //Only UDP protocol
32
33    //Get address(es)
34    struct addrinfo *servAddr;          //List of server addresses
35    int rtnVal = getaddrinfo(server, servPort, &addrCriteria, &servAddr);
36    if (rtnVal != 0)
37        DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
38
39    //Create a datagram/UDP socket
40    int sock = socket(servAddr->ai_family, servAddr->ai_socktype,
```

```

41     servAddr->ai_protocol);           //Socket descriptor for client
42 if (sock < 0)
43     DieWithSystemMessage("socket() failed");
44
45 //Send the string to the server
46 ssize_t numBytes = sendto(sock, echoString, echoStringLen, 0,
47     servAddr->ai_addr, servAddr->ai_addrlen);
48 if (numBytes < 0)
49     DieWithSystemMessage("sendto() failed");
50 else if (numBytes != echoStringLen)
51     DieWithUserMessage("sendto() error", "sent unexpected number of bytes");
52
53 //Receive a response
54
55 struct sockaddr_storage fromAddr; //Source address of server
56 //Set length of from address structure (in-out parameter)
57 socklen_t fromAddrLen = sizeof(fromAddr);
58 char buffer[MAXSTRINGLENGTH + 1]; //I/O buffer
59 numBytes = recvfrom(sock, buffer, MAXSTRINGLENGTH, 0,
60     (struct sockaddr *) &fromAddr, &fromAddrLen);
61 if (numBytes < 0)
62     DieWithSystemMessage("recvfrom() failed");
63 else if (numBytes != echoStringLen)
64     DieWithUserMessage("recvfrom() error", "received unexpected number of bytes");
65
66 //Verify reception from expected source
67 if (!SockAddrsEqual(servAddr->ai_addr, (struct sockaddr *) &fromAddr))
68     DieWithUserMessage("recvfrom()", "received a packet from unknown source");
69
70 freeaddrinfo(servAddr);
71
72 buffer[echoStringLen] = '\0';      //Null-terminate received data
73 printf("Received: %s\n", buffer); //Print the echoed string
74
75 close(sock);
76 exit(0);
77 }

```

(1) 程序建立和参数解析：第 1~23 行。

把要应答的服务器地址/名称作为前两个参数传入。我们限制了应答消息的大小；因此，我们必须验证给定的字符串是否满足这种限制。客户可以选择接受服务器端口或服务器名称作为第三个参数。如果没有提供端口，客户就会使用众所周知的应答协议服务名称

“echo”。

(2) 获取服务器的外部地址: 第 25~37 行。

对于服务器, 可能提供 IPv4 地址、IPv6 地址或者要解析的名称。对于可选的端口, 可能提供端口号或服务名称。我们使用 `getaddrinfo()` 来确定相应的地址信息(即地址族、地址和端口号)。注意: 我们将为 UDP (SOCK\_DGRAM 和 IPPROTO\_UDP) 接受任何地址族 (AF\_UNSPEC) 的一个地址; 指定后两个参数实际上就把返回的地址族限制为 IPv4 和 IPv6。另请注意: `getaddrinfo()` 可能返回多个地址; 当使用列表中后面的某个地址可以进行通信时, 通过简单地使用第一个地址可能无法与服务器通信。**生产客户应该准备试验所有返回的地址。**

(3) 套接字创建和设置: 第 39~43 行。

这几乎与 TCP 应答客户完全相同, 只不过我们是使用 UDP 创建数据报套接字。注意: 在与服务器通信之前不需要调用 `connect()`。

(4) 发送单个应答数据报: 第 45~51 行。

利用 UDP, 简单地告诉 `sendto()` 数据报的目的地。如果我们需要, 可以调用 `sendto()` 多次, 并在每次调用时改变目的地, 从而通过同一个套接字与多个服务器通信。第一次调用 `sendto()` 还会把未被其他任何套接字使用的一个任意选择的本地端口号分配给由 `sock` 标识的套接字, 因为我们以前没有把套接字绑定到端口号。我们不知道(或者不关心)所选的端口号是什么, 但是服务器将使用它来把应答消息发回给我们。

(5) 获取并打印应答消息: 第 55~73 行。

- 接收消息: 第 55~64 行。

我们把 `fromAddrLen` 初始化为包含地址缓冲区 (`fromAddr`) 的大小, 然后作为最后一个参数传递其地址。`recvfrom()` 会阻塞, 直至寻址到这个套接字的端口的 UDP 数据报到达。然后, 它会从第一个到达的数据报中把数据复制进缓冲区中, 并从分组的头部将其来源的 Internet 地址和 (UDP) 端口号复制进结构 `fromAddr` 中。注意: 数据缓冲区实际上比 `MAXSTRINGLENGTH` 大 1 字节, 这允许我们添加一个 null 字节来终止字符串。

- 检查消息来源: 第 67~70 行。

由于没有连接, 接收的消息可能来自任何来源。输出参数 `fromAddr` 告知我们数据报的来源, 我们对其进行检查以确保它与服务器的 Internet 地址匹配。我们使用我们自己的函数 `SockAddrsEqual()` 对套接字地址执行与协议无关的比较。尽管极不可能出现分组是从任何其他来源到达的情况, 我们还是包括了这项检查, 以强调它是可能的。还有另外一种复杂的情况。对于具有多个或重复请求的应用程序, 我们必须记住 UDP 消息可能重新排序并且随意延迟, 因此只检查源地址和端口可能是不够的(例如, UDP 之上的 DNS 协议使用标识符字段来链接请求和响应, 并删除重

复的分组）。这是我们最后一次使用从 `getaddrinfo()` 返回的地址，因此我们可以释放关联的存储空间。

- 打印接收到的字符串：第 72~73 行。

在把接收到的数据作为字符串进行打印之前，首先要确保它是 null 终止的。

- (6) 结尾：第 75~76 行。

对于介绍 UDP 套接字调用，这个示例客户很好；在大多数时间它都会正确地工作。不过，它不适合于生产应用，因为如果发送到服务器或者从服务器发出的消息丢失，对 `recvfrom()` 的调用将会永远阻塞，并且程序不会终止。客户一般通过使用超时（`timeout`）来处理这个问题，我们将在本书后面的 6.3.3 节中介绍这个主题。

## 4.2 UDP 服务器

下一个示例程序 `UDPEchoServer.c` 实现了应答服务器的 UDP 版本。服务器非常简单：它会永远循环，接收一条消息，然后把相同的消息发送回它的任何源发地。实际上，服务器只会接收并发送回消息的前 255 个字符；任何多余的字符都会被套接字实现悄悄地丢弃（参见 4.3 节，了解相关的解释）。

### **UDPEchoServer.c**

```

1 #include <stdlib.h>
2 #include <string.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netdb.h>
6 #include "Practical.h"
7
8 int main(int argc, char *argv[]) {
9
10    if (argc != 2)           //Test for correct number of arguments
11        DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
12
13    char *service = argv[1]; //First arg: local port/service
14
15    //Construct the server address structure
16    struct addrinfo addrCriteria;           //Criteria for address
17    memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
18    addrCriteria.ai_family = AF_UNSPEC;      //Any address family
19    addrCriteria.ai_flags = AI_PASSIVE;       //Accept on any address/port
20    addrCriteria.ai_socktype = SOCK_DGRAM;   //Only datagram socket
21    addrCriteria.ai_protocol = IPPROTO_UDP; //Only UDP socket

```

```
22
23 struct addrinfo *servAddr;           //List of server addresses
24 int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
25 if (rtnVal != 0)
26     DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
27
28 //Create socket for incoming connections
29 int sock = socket(servAddr->ai_family, servAddr->ai_socktype,
30     servAddr->ai_protocol);
31 if (sock < 0)
32     DieWithSystemMessage("socket() failed");
33
34 //Bind to the local address
35 if (bind(sock, servAddr->ai_addr, servAddr->ai_addrlen) < 0)
36     DieWithSystemMessage("bind() failed");
37
38 //Free address list allocated by getaddrinfo()
39 freeaddrinfo(servAddr);
40
41 for (;;) {                         //Run forever
42     struct sockaddr_storage clntAddr; //Client address
43     //Set Length of client address structure (in-out parameter)
44     socklen_t clntAddrLen = sizeof(clntAddr);
45
46     //Block until receive message from a client
47     char buffer[MAXSTRINGLENGTH];    //I/O buffer
48     //Size of received message
49     ssize_t numBytesRcvd = recvfrom(sock, buffer, MAXSTRINGLENGTH, 0,
50         (struct sockaddr *) &clntAddr, &clntAddrLen);
51     if (numBytesRcvd < 0)
52         DieWithSystemMessage("recvfrom() failed");
53
54     fputs("Handling client ", stdout);
55     PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
56     fputc('\n', stdout);
57
58     //Send received datagram back to the client
59     ssize_t numBytesSent = sendto(sock, buffer, numBytesRcvd, 0,
60         (struct sockaddr *) &clntAddr, sizeof(clntAddr));
61     if (numBytesSent < 0)
62         DieWithSystemMessage("sendto() failed");
63     else if (numBytesSent != numBytesRcvd)
64         DieWithUserMessage("sendto()", "sent unexpected number of bytes");
65 }
```

```

66 //NOT REACHED
67 }

```

- (1) 程序建立和参数解析：第 1~13 行。  
(2) 分析/解析地址/端口参数：第 15~26 行。

可能在命令行上将端口指定为端口号或服务名称。我们使用 `getaddrinfo()` 来确定实际的本地端口号。与我们的 UDP 客户一样，我们将接受用于 UDP（`SOCK_DGRAM` 和 `IPPROTO_UDP`）的任何地址族（`AF_UNSPEC`）的地址。我们希望 UDP 服务器接受来自它的任何接口的应答请求。设置 `AI_PASSIVE` 标志使得 `getaddrinfo()` 返回通配符 Internet 地址（用于 IPv4 的 `INADDR_ANY` 或者用于 IPv6 的 `IN6ADDR_ANY`）。`getaddrinfo()` 可能返回多个地址；我们只使用第一个地址。

- (3) 套接字创建和设置：第 28~39 行。

这几乎与 TCP 应答服务器完全相同，只不过我们会创建一个使用 UDP 的数据报套接字。此外，我们不需要调用 `listen()`，因为没有连接建立这个步骤——一旦套接字具有地址，它就准备好接收消息。

- (4) 以迭代方式处理进入的应答请求：第 41~65 行。

UDP 服务器与 TCP 服务器之间的几个关键区别体现在它们与客户进行通信的方式上。在 TCP 服务器中，在等待来自客户的连接时，会阻塞对 `accept()` 的调用。由于 UDP 服务器不会建立连接，因此不需要为每个客户获取一个新的套接字。我们可以代之以利用绑定到想要端口号的相同套接字立即调用 `recvfrom()`。

- 接受应答请求：第 42~52 行。

`recvfrom()` 会阻塞，直至接收到来自客户的数据报。由于没有连接，每个数据报可能来自于不同的发送者，我们会在接收到数据报的同时获悉其来源。`recvfrom()` 把源的地址放在 `clntAddr` 中。这个地址缓冲区的长度由 `cliAddrLen` 指定。

- 发送应答消息：第 59~64 行。

`sendto()` 把缓冲区中的数据传输回通过 `clntAddr` 指定的地址。每个接收到的数据报都会被视作一个客户应答请求，因此我们只需要一次发送和接收——这与 TCP 应答服务器不同，其中我们需要一直接收到客户关闭了连接为止。

## 4.3 利用 UDP 套接字进行发送和接收

一旦创建了 UDP 套接字，就可以使用它来向任何地址发送消息/接收来自任何地址的消息，以及接连向许多不同的地址发送消息/接收来自许多不同地址的消息。为了允许为每条消息指定目的地址，Sockets API 提供了一个不同的发送例程 `sendto()`，它一般用于 UDP

套接字。类似地，除了消息本身之外，recvfrom()例程还会返回每个接收到的消息的源地址。

---

```
ssize_t sendto(int socket, const void *msg, size_t msgLength, int flags,
               const struct sockaddr *destAddr, socklen_t addrLen)
ssize_t recvfrom(int socket, void *msg, size_t msgLength, int flags,
                  struct sockaddr *srcAddr, socklen_t *addrLen)
```

---

sendto()的前 4 个参数与 send() 的那些参数相同。另外两个参数指定消息的目的地。同样，它们始终是分别指向 struct sockaddr\_in 及其大小的指针，或者分别是指向 struct sockaddr\_in6 及其大小的指针。类似地，recvfrom()接受与 recv() 相同的参数，但是，除此之外，它还具有另外两个参数，用于告知调用者所接收到的数据报的来源。要注意的一件事情是：addrLen 是 recvfrom() 中的一个输入/输出型的参数：在输入时，它指定地址缓冲区 srcAddr 的大小，在与 IP 版本无关的代码中它通常是一个 struct sockaddr\_storage。在输出时，它指定实际地复制到缓冲区中的地址的大小。初学者常犯的两个错误是：(1) 给用于 addrLen 的整数传递一个整数值而不是一个指针；(2) 忘记初始化所指向的长度变量以包含合适的大小。

我们已经指出了 TCP 与 UDP 之间的一个细微而重要的区别，即 UDP 会保留消息边界。特别是，每次调用 recvfrom() 都会返回来自至多一个 sendto() 调用的数据。而且，不同的 recvfrom() 调用永远不会返回来自相同的 sendto() 调用的数据（除非把 MSG\_PEEK 标志用于 recvfrom()——参见本节中的最后一个段落）。

当 TCP 套接字上的 send() 调用返回时，调用者只知道数据已经复制进缓冲区中以进行传输；数据还可能会或者可能不会实际地进行传输（第 7 章中将更详细地解释这一点）。不过，UDP 不会缓冲数据以进行可能的重传，因为它不会从错误中恢复。这意味着当 UDP 套接字上的 sendto() 调用返回时，就已经把消息传递给底层信道以进行传输，并且已经（或者很快将要）发送出去。

在消息从网络到达的时间与通过 recv() 或 recvfrom() 返回其数据的时间之间，将把数据存储在一种先进先出（first-in, first-out, FIFO）接收缓冲区中。利用被连接的 TCP 套接字，所有已接收但尚未递送的字节都将被视作一个连续的序列（参见 7.1 节）。不过，对于 UDP 套接字，来自不同消息的字节可能来自于不同的发送者。因此，需要保留它们之间的边界，使得可以利用正确的地址返回来自每条消息的数据。缓冲区实际上包含数据“块”的一个 FIFO 序列，每个数据块都具有一个关联的源地址。调用 recvfrom() 永远不会返回多个数据块。不过，如果利用大小参数 n 调用 recvfrom()，并且接收 FIFO 中的第一个数据块的大小大于 n 时，则只会返回数据块的前 n 个字节。剩余的字节将被悄悄地丢弃，而不会向接收程序指示这一点。

由此，接收者在调用 recvfrom() 时应该总是提供较大的缓冲区，使之足以存放其应用程

序协议允许的最大消息。这种技术将保证不会有数据丢失。在 UDP 套接字上可以由 `recvfrom()` 返回的最大数据量是 65 507 字节——这是 UDP 数据报中可以携带的最大负载。

此外，接收者也可以结合使用 `MSG_PEEK` 标志与 `recvfrom()` “偷窥” 等待接收的第一个数据块。这个标志导致接收的数据保留在套接字的接收 FIFO 中，使得它可以被接收多次。如果内存不足，应用程序消息的大小差别很大，并且每条消息都在前几个字节中携带有关于其大小的信息，那么这种策略就很有用。接收者首先利用 `MSG_PEEK` 和较小的缓冲区调用 `recvfrom()`，检查消息的前几个字节以确定其大小，然后利用足以存放整个消息的较大缓冲区再次调用 `recvfrom()`（不使用 `MSG_PEEK`）。在通常内存充足的情况下，为可能最大的消息使用足够大的缓冲区是一种更简单的方法。

## 4.4 连接 UDP 套接字

在 UDP 套接字上调用 `connect()` 来固定通过套接字发送的将来数据报的目的地址是可能的。一旦连接，就可以使用 `send()` 代替 `sendto()` 传输数据报，因为不再需要指定目的地址。可以用类似的方式使用 `recv()` 代替 `recvfrom()`，因为连接的 UDP 套接字只能接收来自关联的外部地址和端口的数据报，因此在调用 `connect()` 之后，你就会知道任何进入的数据报的源地址。事实上，在连接后，就只能向指定给 `connect()` 的地址发送消息以及从该地址接收消息。注意：利用 UDP 连接然后使用 `send()` 和 `recv()` 不会改变 UDP 的行为方式。消息边界仍会保留，数据报可能会丢失，等等。可以通过利用 `AF_UNSPEC` 的地址族调用 `connect()` 来“断开连接”。

在 UDP 套接字上调用 `connect()` 的另一个细微的优点是：它允许接收由套接字上以前的动作产生的错误指示。规范的示例是给一个不存在的服务器或端口发送数据报。当发生这种情况时，最终导致错误的 `send()` 将不会返回错误指示。在以后某个时间，将把错误消息递送到你的主机，指示发送的数据报遇到了一个问题。由于这个数据报是一条控制（control）消息而不是常规的 UDP 数据报，如果套接字未连接，系统不能总是告知把它发送到哪里，因为未连接的套接字没有关联的外部地址和端口。不过，如果连接了套接字，系统就能够将错误数据报中的信息与套接字的关联的外部 IP 地址和端口相匹配（参见 7.5 节，了解关于这个过程的详细信息）。注意：将这样的控制错误消息递送到套接字将导致从后续的系统调用（例如，旨在获取应答的 `recv()`）而不是从令人厌恶的 `send()` 返回一个错误。

## 练习题

1. 修改 `UDPEchoClient.c`，以使用 `connect()`。在最终的 `recv()` 后面，说明如何断开连接 UDP 套接字。使用 `getsockname()` 和 `getpeername()`，打印调用 `connect()` 之前和之后以及断开连

接后的本地和外部地址。

2. 修改 UDPEchoServer.c，以使用 connect()。
3. 用实验方法验证你可以使用 UDP 套接字发送和接收的最大数据报的大小。对于 IPv4 和 IPv6，答案有什么不同吗？
4. 虽然 UDPEchoServer.c 使用 bind() 显式指定了它的本地端口号，但是我们在 UDPEchoClient.c 中没有调用 bind()。如何给 UDP 应答客户的套接字提供一个端口号？注意：对于 UDP 和 TCP，答案是不同的。我们可以使用 bind() 选择客户的本地端口。如果我们这样做，可能会遇到什么困难？
5. 修改 UDPEchoClient.c 和 UDPEchoServer.c，允许尽可能最大的应答字符串，其中应答请求被限制于单个数据报。
6. 修改 UDPEchoClient.c 和 UDPEchoServer.c，允许任意大小的应答字符串。可以忽略数据报丢失和重新排序（暂时是这样）。
7. 使用 getsockname() 和 getpeername()，修改 UDPEchoClient.c，紧接在调用 sendto() 之前和之后打印套接字的本地和外部地址。
8. 可以使用相同的 UDP 套接字将数据报发送到许多不同的目的地。修改 UDPEchoClient.c，向两个不同的 UDP 应答服务器发送应答数据报，以及从它们那里接收应答数据报。可以使用运行在多台主机上的用于本书的服务器，或者在同一台主机上利用不同的端口把它使用两次。

# 第 5 章

## 发送和接收数据

你通常会使用套接字，因为你的程序需要给另一个程序提供信息，或者需要使用另一个程序提供的信息。这没有什么魔术：任何交换信息的程序都必须就如何编码（encode）信息（表示为一个位序列）达成一致，对于哪个程序发送什么信息以及接收的信息如何影响程序的行为也是如此。这种关于通过通信信道交换的信息的形式和含义的协定称为协议（protocol）；在实现特定的应用程序中使用的协议就是应用程序协议（application protocol）。在前几章中的应答示例中，应用程序协议无足轻重：客户和服务器的行为都不受它们交换的消息的内容影响。由于在大多数真实的应用程序中客户和服务器的行为依赖于它们交换的信息，应用程序协议通常更复杂一点。

TCP/IP 协议在传输用户数据的字节时，将不会检查或修改它们。这使应用程序在编码它们的信息以进行传输方面具有巨大的灵活性。大多数应用程序协议是依据由字段（field）序列构成的具体消息（message）定义的。每个字段都包含一份特定的编码为位序列的信息。应用程序协议准确指定了发送者如何排列这些位序列，以及接收者如何解释或解析它们，从而可以提取每个字段的含义。由 TCP/IP 施加的唯一约束是：必须以块的形式发送和接收信息，这些块的长度（以位为单位）是 8 的倍数。因此，从现在起，我们将把消息视作字节（byte）的序列。由此，把传输的消息视作数字（每个数字在 0~255 之间）的序列或数组可能是有用的。这对应于可以用 8 位编码的二进制值的范围：00000000 用于 0, 00000001 用于 1, 00000010 用于 2，等等，直到 11111111 用于 255。

在构建一个程序通过套接字与其他程序交换信息时，通常会出现以下两种情况之一：一是在套接字两端设计/编写程序，在这种情况下可以自己自由地定义应用程序协议；二是实现一种别人已经详细说明的协议，也许是一个协议标准（standard）。在任何一种情况下，将不同类型的信息编码和解码为“导线上”的字节的基本原理是相同的（顺便说一下，如果“导线”是由一个程序编写然后由另一个程序读取的文件，那么本章中的所有内容也同样适用）。

## 5.1 编码整数

让我们首先考虑怎样通过套接字发送和接收整数——即可以表示整数的位组。在某种意义上，所有类型的信息最终都将被编码为固定大小的整数，因此发送和接收它们的能力是必不可少的。

### 5.1.1 整数的大小

我们见过 TCP 和 UDP 套接字传输字节的序列：8 位的组，它可以包含范围在 0~255 之间的整数值。有时，发送其值可能大于 255 的整数是必要的；必须使用多个字节编码这样的整数。为了交换固定大小的多字节整数，发送者和接收者必须提前就几件事情达成一致意见。第一件事情是要发送的每个整数的大小（以字节为单位）。

例如，`int` 可以被存储为 32 位的数量。除了 `int` 之外，C 语言还定义了另外几种整数类型：`short`、`char` 和 `long`；其思想是：这些整数可以具有不同的大小，并且程序员可以使用适合于应用程序的整数。不过，与一些语言不同，C 语言没有指定这些基本类型的准确大小，而是把它留给实现来完成。因此，`short` 整数的大小可能因平台而异。<sup>1</sup>C 语言规范确实指出：`char` 不大于 `short`，`short` 不大于 `int`，`int` 不大于 `long`，`long` 不大于 `long long`。不过，该规范并不要求这些类型实际上具有不同的大小——使 `char` 具有与 `long` 相同的大小从技术上讲是可能的！不过，在大多数平台上，它们的大小确实有区别，并且在你自己的平台上区分它们的大小也是一个安全的赌博。

那么，如何在你的平台上确定 `int`（或者 `char`、`long` 等）的准确大小呢？答案很简单：使用 `sizeof()` 运算符，它返回当前平台上由其参数（类型或变量）占据的内存空间（以“字节”为单位）。这里，关于 `sizeof()` 要注意两件事。第一，语言指定 `sizeof(char)` 是 1——总是如此。因此，在 C 语言中，“字节”是由 `char` 类型的变量占据的空间，并且 `sizeof()` 的单位实际上是 `sizeof(char)`。但是 C 语言的“字节”的准确大小是多少？这是要注意的第二件事：预先定义的常量 `CHAR_BIT` 指出表示一个 `char` 类型的值需要采用多少位——通常是 8 位，但是也可能是 10 位或者甚至是 32 位。

尽管总是可以为不同的基本整数类型编写一个简单的程序打印由 `sizeof()` 返回的值，从而消除关于你的平台上的整数大小的任何神秘性，但是如果你想为通过 Internet 发送特定大小的整数编写可移植的代码，C 语言没有确切说明其基本整数类型的大小这一事实就使得这有点困难。考虑通过 TCP 连接发送 32 位整数的问题。你是使用 `int`、`long`，还是别的类

<sup>1</sup> 对于本书中的“平台”，我们意指编译器、操作系统和硬件体系结构的组合。具有 Linux 操作系统并且运行在 Intel 的 IA-32 体系结构上的 gcc 编译器就是一个平台的示例。

型？在一些平台上，int 是 32 位，而在另外一些平台上，long 可能是 32 位。

C99 语言标准规范以一组可选类型的形式提供了一种解决方案：int8\_t、int16\_t、int32\_t 和 int64\_t（以及它们对应的无符号类型，比如 uint8\_t 等）全都具有它们的名称指示的大小（以位为单位）。在 CHAR\_BIT 的值是 8 的平台上<sup>1</sup>，这些类型分别是 1、2、4 和 8 字节的整数。尽管可能不会在每个平台上都实现这些类型，如果任何本地基本类型具有对应的大 小，则需要定义相应的类型（因此，如果某个平台上的 int 的大小是 32 位，就需要定义“可选”类型 int32\_t）。在本书余下部分中，我们将利用这些类型来指定我们想要的整数的大小。我们还将利用 C99 定义的类型 long long，它通常大于 long。程序 TestSizes.c 将打印 CHAR\_BIT 的值、所有基本整数类型的大小，以及 C99 标准定义的固定大小的类型的大小（如果在你的平台上未定义任何可选的类型，则该程序将不会编译）。

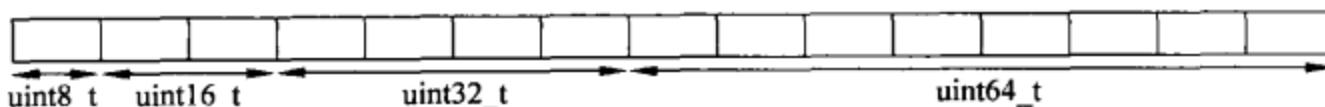
### TestSizes.c

```

1 #include <limits.h>
2 #include <stdint.h>
3 #include <stdio.h>
4
5 int main(int argc, char *argv[]) {
6     printf("CHAR_BIT is %d\n\n", CHAR_BIT); //Bits in a char (usually 8!)
7
8     printf("sizeof(char) is %d\n", sizeof(char)); //ALWAYS 1
9     printf("sizeof(short) is %d\n", sizeof(short));
10    printf("sizeof(int) is %d\n", sizeof(int));
11    printf("sizeof(long) is %d\n", sizeof(long));
12    printf("sizeof(long long) is %d\n\n", sizeof(long long));
13
14    printf("sizeof(int8_t) is %d\n", sizeof(int8_t));
15    printf("sizeof(int16_t) is %d\n", sizeof(int16_t));
16    printf("sizeof(int32_t) is %d\n", sizeof(int32_t));
17    printf("sizeof(int64_t) is %d\n\n", sizeof(int64_t));
18
19    printf("sizeof(uint8_t) is %d\n", sizeof(uint8_t));
20    printf("sizeof(uint16_t) is %d\n", sizeof(uint16_t));
21    printf("sizeof(uint32_t) is %d\n", sizeof(uint32_t));
22    printf("sizeof(uint64_t) is %d\n", sizeof(uint64_t));
23 }
```

<sup>1</sup> 我们不知道任何现代的通用计算平台上哪个平台上的 CHAR\_BIT 的值不是 8。在本书余下部分中，如果未作说明，则假定 CHAR\_BIT 的值是 8。

为了使事情更具体一点，在本节余下内容中我们将考虑编码不同大小的整数序列（确切地讲，依次是 1、2、4 和 8 字节的整数）的问题。因此，我们总共需要 15 字节，如下图所示。

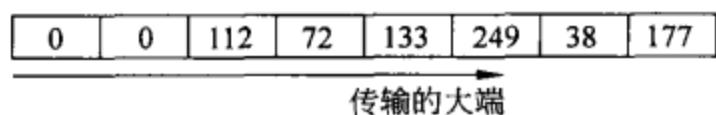


我们将考虑用于执行该任务的几种不同的方法，但是在各种情况下，我们都假定支持 C99 的固定大小的类型。

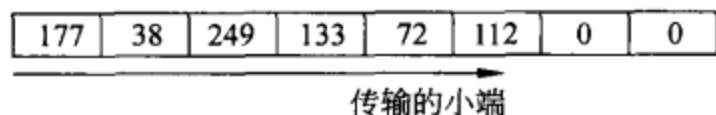
### 5.1.2 字节排序

一旦发送者和接收者指定了要传输的整数的大小，它们就需要在其他一些方面达成一致意见。对于需要多个字节进行编码的整数，它们必须回答以哪种顺序发送字节的问题。

有两种显而易见的选择：利用最低有效位（所谓的小端（little-endian）顺序）从数字的“右”端开始，或者利用最高有效位（大端（big-endian）顺序）从左端开始（注意：幸运的是，字节内位的排序是由实现以标准方式处理的）。考虑 long long 值 123456787654321L。它的 64 位表示（采用十六进制形式）是 0x0000704885F926B1。如果以大端顺序传输字节，（十进制）字节值的序列将如下所示：



如果以小端顺序传输它们，则序列将如下所示：



这里的要点是：对于多个字节的整数量，发送者和接收者需要协商是使用大端顺序还是小端顺序。<sup>1</sup>如果发送者使用小端顺序发送上面的整数，而接收者则期望使用大端顺序，那么接收者将不会接收到正确的值，而是将传输的 8 字节的序列解释为值 12765164544669515776L。

今天，用于在 Internet 中发送多个字节的数字的大多数协议都使用大端字节顺序；事实上，它有时也称为网络字节顺序（network byte order）。由硬件使用的字节顺序（无论它是大端顺序还是小端顺序）称为本机字节顺序（native byte order）。C 语言平台通常会提供一些函数，允许在本机字节顺序与网络字节顺序之间转换值；你可能回忆起我们已经遇到过 htons() 和 htonl()。这些例程以及 ntohs() 和 ntohl() 用于处理典型整数大小的转换。其名称以

<sup>1</sup> 对于大于 2 字节的整数也可能使用其他顺序，但是我们知道没有任何现代系统使用它们。

“l”（用于“long”）结尾的函数用于处理 32 位的数字，而以“s”（用于“short”）结尾的函数则用于处理 16 位的数字。“h”代表“host”（主机），“n”则代表“network”（网络）。因此，在第 2 章中使用 htons() 把 16 位的端口号从主机字节顺序转换为网络字节顺序，这是由于 Sockets API 例程只处理网络字节顺序中的地址和端口。这个事实值得重复提及，因为初级程序员通常由于忘记它而受挫：在 Sockets API 中，地址和端口总是使用网络字节顺序。稍后将充分使用这些顺序转换函数。

### 5.1.3 符号性与符号扩展

发送者与接收者必须协商的最后一个细节是：传输的数字是否带有符号。我们说过，字节包含 0~255 之间的值（十进制）。如果你不需要负数，事实就是这样，但是对于许多应用程序来说是需要负数的。幸运的是，可以把相同的 255 位的模式解释为 -128~127 之间的值。2 的补码（two's-complement）表示法是表示这样的有符号数字的常用方式。对于  $k$  位的数字，负整数  $-n$  ( $1 \leq n \leq 2^{k-1}$ ) 的 2 的补码表示法是任意值  $2^k - n$ 。非负整数  $p$  ( $0 \leq p \leq 2^{k-1} - 1$ ) 简单地通过  $k$  位的任意值  $p$  来进行编码。因此，给定  $k$  位，我们就可以使用 2 的补码表示  $-2^{k-1} \sim 2^{k-1} - 1$  之间的值。注意：最高有效位（most significant bit, msb）指示值为正（msb = 0）还是为负（msb = 1）。另一方面， $k$  位的无符号整数可以直接编码  $0 \sim 2^k - 1$  之间的值。因此，例如，32 位的值 0xFFFFFFFF（全都为 1 的值）在解释为有符号的数时，2 的补码数字表示 -1；在解释为无符号的整数时，它表示 4294967295。应该通过需要编码的值范围确定要传输的整数的符号性。

由于符号扩展（signextension），在处理具有不同符号性的整数时需要小心一些。当把有符号的值复制到任意更宽的类型时，将从符号位（即最高有效位）中复制额外的位。举一个例子，假设变量 smallInt 是 int8\_t 类型，即有符号的 8 位整数，而 widerInt 是 int16\_t 类型。另外假设 smallInt 包含（二进制）值 01001110（即十进制的 78）。如下赋值语句：

```
widerInt := smallInt;
```

将把二进制值 000000001001110 放入 widerInt 中。不过，如果在赋值语句之前 smallInt 具有值 11100010（十进制的 -30），那么其后的 widerInt 将包含二进制值 111111111100010。

现在假设变量 widerUInt 是 uint16\_t 类型，而 smallInt 同样具有值 -30，我们执行以下赋值：

```
widerUInt := smallInt;
```

你认为其后 widerUInt 的值是什么？答案同样是 111111111100010，因为在加宽 smallInt 的值以适应 widerUInt 时将会扩展它的符号，即使后一个变量是无符号的。如果把 widerUInt 的结果值打印为十进制数字，结果将是 65506。另一方面，如果我们具有 uint8\_t 类型的变

量 `smallUInt`, 其中包含相同的二进制值 11100010, 并且把它的值复制到更宽的无符号变量中:

```
widerUInt := smallUInt;
```

然后打印结果, 就会得到 226, 因为无符号整数类型的值 (足够合理地) 不是符号扩展的。

要记住的最后一点是: 在计算表达式的值时, 在任何计算发生之前, 将把变量的值加宽 (如果需要) 到“本机”(int) 大小。因此, 如果把两个 `char` 变量的值相加到一起, 结果的类型将是 `int`, 而不是 `char`:

```
char a,b;
printf("sizeof(a+b) is %d\n", sizeof(a+b));
```

在编写本书时使用的平台上, 这段代码将打印“`sizeof(a+b)` is 4”。`sizeof()`的参数 (表达式 `a + b`) 的类型是 `int`。这一般不是一个问题, 但是需要知道符号扩展也会发生在这种隐式加宽过程中。

### 5.1.4 手工编码整数

在对字节排序 (我们将使用大端顺序) 和符号性 (整数全都是无符号的) 达成一致意见之后, 我们就准备好构造消息。我们首先将说明如何使用移位和掩码操作“手工”执行该任务。程序 `BruteForceCoding.c` 具有一个方法 `EncodeIntBigEndian()`, 它使用大端表示法把任何给定的基本整数值作为指定字节数的序列存放在内存中的指定位置。该方法接受 4 个参数: 指向要存放值的起始位置的指针; 要编码的值 (表示为 64 位的无符号整数, 它足以保存任何其他的类型); 值的起始位置在数组中的偏移量; 以及要写入的值的大小 (以字节为单位)。当然, 在发送者处编码的任何数据都必须在接收者处是可解码的。`DecodeIntBigEndian()` 方法用于处理将给定长度的字节序列解码为 64 位的整数, 并将其解释为大端序列。

这些方法把所有的数字都视作无符号的; 参见练习, 了解其他的可能性。

#### **BruteForceCoding.c**

---

```

1 #include <stdint.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <limits.h>
5 #include "Practical.h"
6
7 const uint8_t val8 = 101;           //One hundred and one

```

```

8  const uint16_t val16 = 10001;           //Ten thousand and one
9  const uint32_t val32 = 100000001;        //One hundred million and one
10 const uint64_t val64 = 100000000001L;    //One trillion and one
11 const int MESSAGELENGTH = sizeof(uint8_t)+sizeof(uint16_t)+sizeof(uint32_t)
12     + sizeof(uint64_t);
13
14 static char stringBuf[BUFSIZE];
15 char *BytesToString(uint8_t *byteArray, int arrayLength) {
16     char *cp = stringBuf;
17     size_t bufSpaceLeft = BUFSIZE;
18     for (int i = 0; i < arrayLength && bufSpaceLeft > 0; i++) {
19         int strl = snprintf(cp, bufSpaceLeft, "%u ", byteArray[i]);
20         bufSpaceLeft -= strl;
21         cp += strl;
22     }
23     return stringBuf;
24 }
25
26 //Warning: Untested preconditions (e.g., 0 <= size <= 8)
27 int EncodeIntBigEndian(uint8_t dst[], uint64_t val, int offset, int size) {
28     for (int i = 0; i < size; i++) {
29         dst[offset++]=(uint8_t)(val >>((size -1)- i) * CHAR_BIT);
30     }
31     return offset;
32 }
33
34 //Warning: Untested preconditions (e.g., 0 <= size <= 8)
35 uint64_t DecodeIntBigEndian(uint8_t val[], int offset, int size){
36     uint64_t rtn = 0;
37     for (int i = 0; i < size; i++) {
38         rtn =(rtn << CHAR_BIT)| val[offset + i];
39     }
40     return rtn;
41 }
42
43 int main(int argc, char *argv[]) {
44     uint8_t message[MESSAGELENGTH]; //Big enough to hold all four values
45
46     //Encode the integers in sequence in the message buffer
47     int offset = 0;
48     offset = EncodeIntBigEndian(message, val8, offset, sizeof(uint8_t));
49     offset = EncodeIntBigEndian(message, val16, offset, sizeof(uint16_t));
50     offset = EncodeIntBigEndian(message, val32, offset, sizeof(uint32_t));
51     offset = EncodeIntBigEndian(message, val64, offset, sizeof(uint64_t));

```

```

52     printf("Encoded message:\n%s\n", BytesToDecString(message, MESSAGELENGTH));
53
54     uint64_t value =
55         DecodeIntBigEndian(message, sizeof(uint8_t), sizeof(uint16_t));
56     printf("Decoded 2-byte integer = %u\n", (unsigned int) value);
57     value = DecodeIntBigEndian(message, sizeof(uint8_t) + sizeof(uint16_t)
58         + sizeof(uint32_t), sizeof(uint64_t));
59     printf("Decoded 8-byte integer = %llu\n", value);
60
61     //Show signedness
62     offset = 4;
63     int iSize = sizeof(int32_t);
64     value = DecodeIntBigEndian(message, offset, iSize);
65     printf("Decoded value (offset %d, size %d) = %lld\n", offset, iSize, value);
66     int signedVal = DecodeIntBigEndian(message, offset, iSize);
67     printf("...same as signed value %d\n", signedVal);
68 }

```

(1) 声明和包括: 第 1~12 行。

- 库函数和常量: 第 1~5 行。
- 带有要编码的值的整型变量: 第 7~10 行。
- 消息长度计算: 第 11~12 行。

语言规范指示初始化器表达式求值为 15; 出于完整性考虑我们包括了它。

(2) BytesToDecString(): 第 14~24 行。

这个辅助例程接受一个字节数组及其长度, 并返回一个字符串, 其中包含每个字节的值, 作为一个 0~255 之间的十进制整数。

(3) EncodeIntBigEndian(): 第 27~32 行。

把给定的值迭代 size 次。在每次迭代过程中, 赋值语句的右边把要编码的值向右移位, 使得我们感兴趣的字节位于低 8 位中。然后把得到的值强制转换为 uint8\_t 类型, 这将丢弃除低 8 位之外的所有其他位, 并把它们放在数组中的合适位置。返回 offset 的最终值, 使得在编码一个整数序列时调用者不必重新计算它 (如我们所愿)。

(4) DecodeIntBigEndian(): 第 35~41 行。

我们在一个 64 位的整型变量中构造值。同样迭代 size 次, 每次都把累加的值向左移位, 并在下一个字节的值中进行按位“或”运算。

(5) 演示方法: 第 43~68 行。

- 声明缓冲区 (字节的数组), 以接收整数系列: 第 44 行。
- 编码数据项: 第 47~51 行。

将整数编码进以前所述的序列中的数组中。

- 打印编码数组的内容：第 52 行。
- 从编码的消息中提取并显示一些值：第 54~59 行。  
输出应该显示解码的值等于原始常量。
- 符号性效果：第 62~67 行。

在偏移量 4 处，字节值是 245（十进制）；由于它设置了其高阶位，如果它是有符号值的高阶字节，那么将把该值视为负值。我们显示这将被解码为从偏移量 4 处开始的 4 个字节，并把结果同时放在一个有符号整数和一个无符号整数中。

**注意：**我们可能考虑在 `EncodeIntBigEndian()` 和 `DecodeIntBigEndian()` 的开始处测试几个前提条件，比如： $0 \leq size \leq 8$  和 `dst != NULL`。你可以指出任何其他的前提条件吗？

运行程序产生输出，其中显示了以下（十进制）字节值：

101	39	17	5	245	225	1	0	0	0	232	212	165	16	1
byte	short	int			long									

可以看到，蛮力方法要求程序员做相当多的工作：计算和指定每个值的偏移量和大小，以及利用合适的参数调用编码例程。幸运的是，通常可以使用一些替代方式来构建消息。接下来将讨论它们。

### 5.1.5 在流中包装 TCP 套接字

编码多个字节的整数以便通过流（TCP）套接字进行传输的方式是：使用内置的 FILE 流工具——可以把相同的方法用于 `stdin`、`stdout` 等。要访问这些工具，需要通过 `fdopen()` 调用把一个或多个 FILE 流与套接字描述符相关联。

---

```
FILE *fdopen(int socketdes, const char *mode)
int fclose(FILE * stream)
int fflush(FILE * stream)
```

---

`fdopen()` 函数把套接字“包装”在流中并返回结果（如果发生错误，则返回 `NULL`）；就好像你可以在网络地址上调用 `fopen()` 一样。这允许通过像 `fgets()`、`fputs()`、`fread()` 和 `fwrite()` 这样的操作在套接字上执行缓冲的 I/O；`fdopen()` 的“模式”参数接受与 `fopen()` 相同的值。`fflush()` 把缓冲的数据推送到底层套接字。`fclose()` 用于关闭流以及底层套接字。`fflush()` 则导致通过底层套接字发送任何缓冲的数据。

---

```
size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream)
size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream)
```

---

`fwrite()` 方法把给定大小的指定数量的对象写到流中。`fread()` 方法则执行相反的操作，

从给定的流中读取给定大小的给定数量的对象，并按顺序把它们存放在 ptr 指向的位置。注意：大小是以 sizeof(char) 的单位给出的，而这些方法的返回值是读取/写入的对象数量，而不是字节数。特别是，fread() 永远不会从流中读取对象的一部分；与之类似，fwrite() 永远不会写入一部分对象。如果底层连接终止，这些方法将返回一个较短的数据项计数。

通过给 fwrite() 提供不同的大小，我们可以按顺序把消息输出到流。假定像 BruteForceCoding.c 中那样声明了变量 val8、val16 以及其余的变量，并且整型变量 sock 是我们想通过其写入消息的被连接的 TCP 套接字的描述符。最后，假定 htonl() 是一个把 64 位的整数从主机字节顺序转换为网络字节顺序（参见练习 2）。然后，我们可以一次一个整数地把消息写到套接字：

```
sock = socket(/*...*/);
/* ... connect socket ...*/
//wrap the socket in an output stream
FILE *outstream = fdopen(sock, "w");
//send message, converting each object to network byte order before sending
if (fwrite(&val8, sizeof(val8), 1, outstream) != 1) ...
    val16 = htons(val16);
if (fwrite(&val16, sizeof(val16), 1, outstream) != 1) ...
    val32 = htonl(val32);
if (fwrite(&val32, sizeof(val32), 1, outstream) != 1) ...
    val64 = htonl(val64);
if (fwrite(&val64, sizeof(val64), 1, outstream) != 1) ...
    fflush(outstream); //immediately flush stream buffer to socket
... //do other work...
fclose(outstream); //flushes stream and closes socket
```

发送端做了这么多的事。那么接收者如何恢复传输的值呢？如你可能期望的，接收端使用 fread() 采取了类似的步骤。现在假设变量 csock 包含用于 TCP 连接的套接字描述符，接收的值将存放在具有期望类型的变量 rcv8、rcv16、rcv32 和 rcv64 中，并且 ntohl() 是用于 64 位类型的从网络字节顺序到主机字节顺序的转换器。

```
/* ... csock is connected ...*/
//wrap the socket in an input stream
FILE *instream = fdopen(csock, "r");
//receive message, converting each received object to host byte order
if (fread(&rcv8, sizeof(rcv8), 1, instream) != 1) ...
if (fread(&rcv16, sizeof(rcv16), 1, instream) != 1) ...
rcv16 = ntohs(rcv16); //convert to host order
if (fread(&rcv32, sizeof(rcv32), 1, instream) != 1) ...
rcv32 = ntohl(rcv32);
if (fread(&rcv64, sizeof(rcv64), 1, instream) != 1) ...
```

```

recv64 = ntohll(recv64);
...
fclose(instream); //closes the socket connection!

```

利用套接字使用缓冲的 FILE 流的优点之一是：在从流中读取一个字节（通过 ungetc()）之后能够把它放回原处；在解析消息时，这有时可能是有用的。不过，我们必须强调 FILE 流只能用于 TCP 套接字。

### 5.1.6 结构覆盖：对齐与填充

构造包含二进制数据（即多字节整数）的消息的最常用的方法涉及把 C 结构覆盖在一块内存区域上，并直接分配给结构的字段。这是可能的，因为 C 语言规范显式定义了编译器如何在内存中布置结构。它具有这种特性，因为它被设计用于实现操作系统，其中效率是一个主要的考虑事项，并且确切知道如何表示数据结构通常是有必要的。结合使用我们以前见过的用于字节顺序转换的工具，这使得可以相当容易地构造和解析由特定大小的整数组成的消息（不过，我们稍后将遇到一个重要的告诫）。

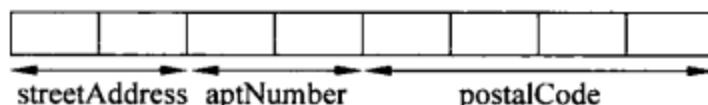
暂且假设我们正在处理地址的三个整数成分：街道上的编号，在 1 到 12000 左右之间；公寓编号，永远不会超过 8000（非公寓地址使用公寓编号 -1）；以及邮政编码，它是 10000~99999 之间的一个 5 位数字。前两个成分可以使用 16 位整数表示；第三个成分对于 16 位整数来说太大了，因此我们将为它使用 32 位的整数。如果我们想在程序内传递这些数字，可以声明一个结构并传递指向它的指针：

```

struct addressInfo {
    uint16_t streetAddress;
    int16_t aptNumber;
    uint32_t postalCode;
} addrInfo;

```

显然，这种结构对于在程序中传递数据是有用的，但是我们可以使用它通过 Internet 在程序之间传递信息吗？答案是可以。C 语言规范指出：可以像下面这样在内存中布置这种结构：



为了在程序之间交换信息，我们可以简单地使用结构的布局作为消息格式，然后直接从结构中获取内容并通过网络发送它们（在执行了任何所需的字节顺序转换之后）。如果上面声明的变量 `addrInfo` 被初始化成包含我们想发送的值，并且 `sock` 像通常的那样表示一个连接的套接字，就可以使用以下代码发送 8 字节的消息：

```
//... put values in addrInfo ...
//convert to network byte order
addrInfo.streetAddress = htons(addrInfo.streetAddress);
addrInfo.aptNumber = htons(addrInfo.aptNumber);
addrInfo.postalCode = htonl(addrInfo.postalCode);
if (send(sock, &addrInfo, sizeof(addrInfo), 0) != sizeof(addrInfo)) ...
```

在接收端，可以再次使用缓冲的输入流（参见上一节的内容）和 `fread()` 处理将正确数量的字节获取进结构中（否则，我们必须使用循环，因为如我们以前看到的，无法保证通过调用一次 `recv()` 来返回消息的所有字节）。一旦完成了这个任务，我们就只需关心字节排序：

```
struct addressInfo addrInfo;
//... sock is a connected socket descriptor ...
FILE *instream = fdopen(sock, "r");
if (fread(&addrInfo, sizeof(struct addressInfo), 1, instream) != 1) {
    //... handle error
}
//convert to host byte order
addrInfo.streetAddress = ntohs(addrInfo.streetAddress);
addrInfo.aptNumber = ntohs(addrInfo.aptNumber);
addrInfo.postalCode = ntohl(addrInfo.postalCode);
//use information from message...
```

现在，似乎可以使用如下所示的声明来构造 15 字节的消息：

```
struct integerMessage {
    uint8_t oneByte;
    uint16_t twoBytes;
    uint32_t fourBytes;
    uint64_t eightBytes;
}
```

唉，这不会工作，因为针对布置数据结构的 C 语言规则包括特定的对齐（alignment）要求，包括结构内的字段基于它们的类型开始于某些边界。可以把这些要求的要点总结如下：

- 数据结构是最大限度地对齐的。也就是说，一个结构的任何实例的地址（包括数组中的地址）都可以被其最大的本机整型字段除尽。
- 其类型是多字节整数类型的字段要与它们的大小（以字节为单位）对齐。因此，一个 `int32_t` 的整型字段的开始地址总是可以被 4 除尽，而一个 `uint16_t` 的整型字段的地址则保证可以被 2 除尽。

为了强制执行这些约束条件，编译器可能在结构的字段之间添加填充。由此，上面声

明的结构的大小不是 15，而是 16。为了理解其原因，让我们考虑应用的约束条件。第一，整个结构必须开始于可以被 8 除尽的地址。`twoBytes` 字段必须位于一个偶数地址上，`fourBytes` 必须位于一个可以被 4 除尽的地址上，而 `eightBytes` 的地址必须能够被 8 除尽。如果在 `oneByte` 字段与 `twoBytes` 字段之间插入一个字节的填充，那么所有这些约束条件都会得到满足，如下所示：



由编译器添加为填充的字节内容是未定义的。因此，如果使用上面的声明，并且接收者期望原始的未添加填充的布局，那么很可能出现不正确的行为。

最佳的解决方案是：通过布置消息使得不需要进行填充。不幸的是（并且有点令人惊奇的是），这并非总是可能的。就我们的 4 个整数的示例而言，如果利用相反顺序的字段排列消息：

```
struct backwardMessage {
    uint64_t eightBytes;
    uint32_t fourBytes;
    uint16_t twoBytes;
    uint8_t oneByte;
}
```

那么字段之间将不需要填充。不过，`sizeof(struct backwardMessage)` 将返回 16，而不是 15。这是由于在结构的实例之间（例如，在数组中）将需要一个字节的填充，以便满足第一个（最大限度的对齐）约束条件（必须保持不变的是，在结构的数组中，一个元素的地址加上结构的大小（利用 `sizeof()` 获得）将产生后续元素的地址）。因此，无法指定一个包含任意多字节整数并且 `sizeof()` 会为其返回一个奇数的结构。

如果我们不能消除编译器所需的填充，作为最后一种手段，我们可以把它包括在消息格式规范中。对于我们的示例，可以利用第一个字段后面的显式填充字节来定义消息格式：

```
struct integerMessage2 {
    uint8_t oneByte;
    uint8_t padding; //Required for alignment
    uint16_t twoBytes;
    uint32_t fourBytes;
    uint64_t eightBytes;
}
```

这个结构在内存中的布置方式与原来声明的 `integerMessage` 完全一样，只不过程序员现在可以控制和访问填充字节的内容。在 5.2.3 节中将看到使用结构来编码数据的更多

示例。

### 5.1.7 字符串和文本

旧式文本——可打印（可显示）的字符串——也许是表示信息的最常用的方式。我们已经在我们的应答客户和服务器中见过了发送和接收文本字符串的示例；你也可能习惯于使编写的程序产生文本输出。

文本很方便，因为我们时刻都在与它打交道：信息被表示为图书、报纸和计算机显示器上的字符串。因此，一旦我们知道如何编码文本以进行传输，发送大多数其他任何类型的数据就很直观：只需把它表示为文本，然后编码文本。你知道如何把数字和布尔值表示为文本字符串——例如，“123478962”、“6.02e23”、“true”、“false”。在本节中，我们将处理把这样的字符串编码为字节序列的问题。

为此，我们首先需要认可文本是由符号或字符序列组成的。C 语言包括了基本类型 `char` 用于表示字符，但是没有包括用于字符串的基本类型。传统上将 C 语言中的字符串表示为 `char` 的数组。在内部将 C 语言中的 `char` 值表示为整数。例如，字符 “a”（即用于字母 “a”的符号）对应于整数 97。字符 “X” 对应于 88，符号 “!”（感叹号）则对应于 33。

一组符号与一组整数之间的映射称为编码字符集（coded character set）。你可能听说过名为 ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）的编码字符集。ASCII 把英语字母表中的字母、数字、标点符号及一些其他的特殊（不可打印的）符号映射到 0~127 之间的整数。它从 20 世纪 60 年代起就用于数据传输，并且广泛用在像 HTTP（用于万维网的协议）这样的应用程序协议中，甚至到今天还在使用它。C 语言规范指定了一个基本字符集，它是 ASCII 的一个子集。ASCII（以及 C 语言的基本字符集）的重要性在于：对于只包含基本字符集中的字符的字符串，可以使用一个字符对应一个字节的方式进行编码（注意：在第 2 章中的应答客户和服务器中，编码是不相关的，因为服务器根本不会解释接收到的数据）。

如果说并且使用可以利用少量符号表示的语言（比如英语），这就很好。不过，考虑使用每个符号对应一个字节的方法可以编码不超过 256 个不同的符号，并且世界上的大部分人使用的语言都不超过 256 个符号，因此必须使用 8 个以上的位对每个字符进行编码。显然，使用 C 语言对于实现“可国际化的”代码提出了重大的挑战。中国的普通话就是一个著名的例子，这种语言需要数千个符号。

C99 扩展标准定义了一种 `wchar_t`（“wide character”（宽字符））类型，用于存储可能为每个符号使用多个字节的字符集中的字符。此外，还定义了多个库函数，用于支持在字节序列与 `wchar_t` 的数组之间相互进行转换（事实上，用于操作字符串的几乎所有的库函数都有一个宽字符串版本）。为了在宽字符串与适合于通过网络传输的编码的字符（字节）序列之间来回转换，我们将使用 `wcstombs()`（“wide character string to multibyte string”（宽

字符串到多字节字符串) 和 mbstowcs() 函数。

```
#include <stdlib.h>
size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs, size_t n);
size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);
```

其中第一个函数把 pwcs 指向的数组中的宽字符序列转换为多字节的字符序列，并把这些多字节的字符存储在 s 指向的数组中，如果下一个转换将超过总字节数 n 的限制或者如果存储了一个 null 字符，该函数将停止工作。第二个函数在相反方向上执行同样的操作。

如我们所看到的，对于需要更大整数值的编码字符集，可以用多种方式编码这些值以便通过网络进行传输。因此，发送者和接收者必须就如何将这些整数编码为字节序列达成一致意见。

例如，考虑在内部使用 16 位整数表示字符的平台，并且它的字符集包括 ASCII 作为一个子集——也就是说，字符“a”映射到 97，“X”映射到 88，等等。使用宽字符工具，可以声明一个宽字符串：

```
wchar_t testString[] = "Test!";
```

这个字符串的内部表示使用 16 位的整数（包括一个用于 null (0) 终止符的整数）。不过，可以使用多种方式把这些整数编码为 8 位字节的序列：

- 可以以大端顺序使用 2 字节表示每个字符。这样，对宽字符串“Test!”调用 wcstombs() 的结果将是以下序列：0, 84, 0, 101, 0, 115, 0, 116, 0, 33。
- 此外，也可以以小端顺序使用 2 字节，这将得到：84, 0, 101, 0, 115, 0, 116, 0, 33, 0。
- 我们可以使用将原始 ASCII 字符集中的符号映射到单个字节（与它们的 ASCII 编码相同）的编码，并为 ASCII 以外的符号使用 2 字节。如果设置了字节的高阶位，它就指示利用 2 字节编码下一个符号；否则，它就是利用单个字节编码的 ASCII 符号。

在这种情况下，调用 wcstombs() 之后的结果将是 5 字节的序列：84, 101, 115, 116, 33。

**注意：**在这些示例中，我们没有包括 null (0) 终止符。null 终止符是语言产生的，而不是字符串本身的一部分。因此，不应该与字符串一起传输它，除非协议显式指定采用这种方法标记字符串的末尾。

坏消息是 C99 的宽字符工具并未设计成给程序员提供对编码方案的显式控制。的确，它们假定依据平台的“区域”而定义的单一、固定的字符集。尽管一些工具支持多种字符集，但是它们甚至没有给程序员提供任何方式来获悉正在使用哪种字符集或编码。事实上，C99 标准指出：在多种情况下，在运行时更改区域的字符集的效果是不明确的。这意味着如果你想使用特定的字符集实现一种协议，将不得不自己实现编码。

### 5.1.8 位操作：编码布尔值

位图(bitmap)是编码布尔信息(在协议中经常使用它们)的非常简洁的方式。位图的思想是：整数类型的每一个位都可以编码一个布尔值——通常用0表示false(假)，用1表示true(真)。为了能够操纵位图，需要知道如何使用C语言的“位操作”运算符来设置和清除各个位。掩码(mask)是把一个或多个特定的位设置为1并且清除所有其他的位(即设置为0)的整数值。我们在这里主要讨论32位的映射和掩码，但是这里介绍的所有知识也适用于其他各种大小的类型。

让我们从0到31给整数的二进制表示中的位进行编号，其中位0是最有效位。一般来讲，`uint32_t`值是位位置*i*中具有1，并在所有其他位位置中具有0，正好就是 $2^i$ 。因此位5对应于数字32，位12对应于4096，等等。下面给出了一些示例掩码声明：

```
const int BIT5 = (1<<5);
const int BIT7 = 0x80;
const int BITS2AND3 = 12; //8+4
int bitmap = 128;
```

要设置int变量中的特定的位，可以使用按位“或”运算符(|)把它与该位的掩码结合起来：

```
bitmap |= BIT5;
//bit 5 is now one
```

要清除特定的位，可以把它与该位的掩码的按位求补(除了特定的位(它是0)之外，其他所有位置都具有1)之间进行按位“与”运算。C语言中的按位“与”运算符是&，而按位求补运算符是~。

```
bitmap &= ~BIT7;
//bit 7 is now zero
```

可以通过“或”运算符以及相应的掩码同时设置和清除多个位：

```
//clear bits 2, 3 and 5
bitmap &= ~(BITS2AND3|BIT5);
```

要测试是否设置了某个位，可以把掩码和值的按位“与”的结果同0作比较：

```
bool bit6Set = (bitmap & (1<<6)) != 0;
```

## 5.2 构造、成帧和解析消息

我们利用一个示例来结束本章，该示例演示了在实现其他人指定的协议时如何应用上述技术。这个示例是一个简单的“投票”协议，如图 5.1 所示。其中，客户发送一条请求（request）消息给服务器，该消息包含候选人 ID，它是一个 0~100 之间的整数。它支持两类请求。质询（inquiry）请求用于询问服务器有多少选票投给了给定的候选人，服务器发送回一条响应（response）消息，其中包含原始候选人 ID 以及在接收到针对该候选人的请求时的总票数。投票（voting）请求实际上用于给指定的候选人投票。服务器同样利用一条消息进行响应，该消息中包含候选人 ID 和总票数（它现在包括刚才投的票）。该协议运行在使用 TCP 的流套接字上。

通常，协议消息的“有线格式”将被精确地定义为协议的一部分。如我们所看到的，我们可能用多种方式编码它：我们可以使用文本字符串表示信息，或者将其表示为二进制数字。为了演示上述的所有技术，我们将指定“有线格式”的多个不同的版本。这也有助于我们重视协议的实现。

在实现任何协议时，对主程序逻辑隐藏编码消息的方式的细节是一个良好的实践。我们将在这里演示它，方法是：在主程序中使用一个泛型结构，与处理通过套接字发送/接收的消息的函数之间传递信息。这允许我们对“有线格式”处理的不同实现使用相同的客户和服务器代码。VoteProtocol.h 中定义的 VoteInfo 结构包含构造消息所需的一切信息：候选人 ID 号（一个整数）、针对该候选人的投票计数（一个 64 位的整数）、一个指示消息是否是“质询”的布尔值（质询不影响投票计数），以及另一个布尔值，指示消息是从客户发送到服务器的响应（true）还是请求（false）。这个文件中还定义了一些常量，用于所允许的最大候选 ID 号和线上的编码消息的最大长度（后者可以帮助程序确定其缓冲区的大小）。

```
struct VoteInfo {
    uint64_t count; //invariant: !isResponse => count==0
    int candidate; //invariant: 0 <= candidate <= MAX_CANDIDATE
    bool isInquiry;
    bool isResponse;
};

typedef struct VoteInfo VoteInfo;
```

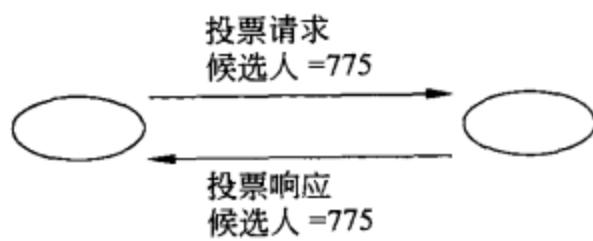


图 5.1 投票协议

```
enum {
    MAX_CANDIDATE = 1000,
    MAX_WIRE_SIZE = 500
};
```

注意：我们为请求消息和响应消息中包含的信息定义了单个 `VoteInfo` 结构。它与正确的控制结构一起允许为客户和服务器以及请求和响应重用相同的消息处理代码。

消息处理代码负责编码来自 `VoteInfo` 结构的信息并通过流套接字传输它，以及用于接收来自 TCP 套接字的数据，解析进入的投票协议消息（如果有的话），并且利用接收到的信息填充 `VoteInfo` 结构。

规则的设计进一步把这个过程分解为两个部分。第一部分关注的是成帧（framing），或者标记消息的边界，使得接收者可以在流中找到它。第二部分关注的是消息的实际编码，即是使用文本还是使用二进制数据表示它。注意：这两个部分可以彼此独立，并且在良好设计的协议中它们应该是分隔开的。换句话说，我们可以指定一种机制用于对消息进行成帧处理，并使之作为一个整体与其不同字段的编码分隔开。这就是我们将要做的。如果我们可以使用流处理函数，就可以使我们的工作更容易一点，因此我们将使客户和服务器把连接的套接字包装在 FILE 流中，一个用于输入，另一个用于输出。

成帧代码的接口在 `Framer.h` 中定义如下：

```
int GetNextMsg(FILE *in, uint8_t *buf, size_t bufSize);
int PutMsg(uint8_t buf[], size_t msgSize, FILE *out);
```

`GetNextMsg()`方法从给定的流中读取数据，并把它存放在给定的缓冲区中，直到它用光了所有的空间或者确定它接收到一条完整的消息为止。它会返回缓冲区中存放的字节数（所有的成帧信息都会被删除）。`PutMsg()`方法把成帧信息添加到给定缓冲区中包含的消息中，并把消息和成帧信息写到给定的流。注意：这两个方法都不需要知道关于消息内容的任何信息。

编码和解析代码的接口在 `VoteEncoding.h` 中定义如下：

```
bool Decode(uint8_t *inBuf, size_t mSize, VoteInfo *v);
size_t Encode(VoteInfo *v, uint8_t *outBuf, size_t bufSize);
```

`Encode()`方法接受一个 `VoteInfo` 结构作为输入，并依据特定的有线格式编码把它转换为一个字节序列；它会返回得到的字节序列的大小。`Decode()`方法接受一个指定大小的字节序列，并依据协议把它解析为消息，利用消息中的信息填充 `VoteInfo`。如果成功地解析了消息，它就会返回 `TRUE`；否则，就会返回 `FALSE`。

给定成帧和解析代码的这些接口，我们现在可以描述将使用这些方法的投票客户和服

务器程序。客户很直观：作为命令行参数提供候选人 ID，以及一个指示事务是质询（默认情况下，它是一个投票请求）的标志。一旦发送了请求，客户就等待响应，然后在它接收到响应时就关闭连接。

### **VoteClientTCP.c**

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <unistd.h>
6 #include <errno.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9 #include <arpa/inet.h>
10 #include <netdb.h>
11 #include "Practical.h"
12 #include "VoteProtocol.h"
13 #include "Framer.h"
14 #include "VoteEncoding.h"
15
16 int main(int argc, char *argv[]) {
17     if (argc < 4 || argc > 5) //Test for correct # of args
18         DieWithUserMessage("Parameter(s)", "<Server> <Port/Service> <Candidate> [I]");
19
20     char *server = argv[1];    //First arg: server address/name
21     char *service = argv[2];   //Second arg: string to echo
22     //Third arg: server port/service
23     int candi = atoi(argv[3]);
24     if (candi < 0 || candi > MAX_CANDIDATE)
25         DieWithUserMessage("Candidate # not valid", argv[3]);
26
27     bool inq = argc > 4 && strcmp(argv[4], "I") == 0;
28
29     //Create a connected TCP socket
30     int sock = SetupTCPClientSocket(server, service);
31     if (sock < 0)
32         DieWithUserMessage("SetupTCPClientSocket() failed", "unable to connect");
33
34     FILE *str = fdopen(sock, "r+"); //Wrap for stream I/O
35     if (str == NULL)
36         DieWithSystemMessage("fdopen() failed");
37

```

```
38 //Set up info for a request
39 VoteInfo vi;
40 memset(&vi, 0, sizeof(vi));
41
42 vi.isInquiry = inq;
43 vi.candidate = candi;
44
45 //Encode for transmission
46 uint8_t outbuf[MAX_WIRE_SIZE];
47 size_t reqSize = Encode(&vi, outbuf, MAX_WIRE_SIZE);
48
49 //Print info
50 printf("Sending %d-byte %s for candidate %d...\n", reqSize,
51        (inq ? "inquiry" : "vote"), candi);
52
53 //Frame and send
54 if (PutMsg(outbuf, reqSize, str) < 0)
55     DieWithSystemMessage("PutMsg() failed");
56
57 //Receive and print response
58 uint8_t inbuf[MAX_WIRE_SIZE];
59 size_t respSize = GetNextMsg(str, inbuf, MAX_WIRE_SIZE); //Get the message
60 if (Decode(inbuf, respSize, &vi)) { //Parse it
61     printf("Received:\n");
62     if (vi.isResponse)
63         printf(" Response to ");
64     if (vi.isInquiry)
65         printf("inquiry ");
66     else
67         printf("vote ");
68     printf("for candidate %d\n", vi.candidate);
69     if (vi.isResponse)
70         printf(" count = %llu\n", vi.count);
71 }
72
73 //Close up
74 fclose(str);
75
76 exit(0);
77 }
```

(1) 访问库函数和常量：第1~14行。

(2) 参数处理：第17~27行。

(3) 获取连接的套接字：第 30~32 行。

(4) 在流中包装套接字：第 34~36 行。

我们使用 `fdopen()` 打开流（模式“r+”用于打开可读和写的流）。

(5) 准备并发送请求消息：第 38~55 行。

- 准备一个带有候选人 ID 的 `VoteInfo` 结构：第 39~43 行。

- 编码为有线格式：第 47 行。

- 在成帧前打印编码的消息：第 50~51 行。

- 添加成帧信息并通过输出流发送它：第 54~55 行。

(6) 接收、解析和处理响应：第 58~71 行。

- 调用 `GetNextMsg()`：第 59 行。

该方法用于处理通过套接字接收足够多的数据来组成下一条消息的所有混乱的细节；像 `recv()` 一样，它可能会不确定地阻塞。

- 传递结果以进行解析：第 60 行。

- 如果响应是正确构成的就处理它：第 61~70 行。

无效的响应将被忽略。

(7) 关闭流：第 74 行。

现在转向服务器，它需要采用一种方式来跟踪所有候选人的投票计数。由于最多有 1001 位候选人，使用 64 位的整数数组就可以很好地满足它。服务器像我们见过的其他服务器一样准备好它的套接字，并等待进入的连接。当连接到达时，我们的程序就会通过该连接接收并处理消息，直至客户关闭它。注意：由于具有成帧/解析代码的非常基本的接口，在涉及处理接收到的消息中的错误时，我们的客户和服务器的头脑相当简单；服务器会简单地忽略任何错误地构成的消息，并立即关闭连接。

## VoteServerTCP.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdbool.h>
4 #include <stdint.h>
5 #include <unistd.h>
6 #include <errno.h>
7 #include <sys/socket.h>
8 #include <arpa/inet.h>
9 #include "Practical.h"
10 #include "VoteProtocol.h"
11 #include "VoteEncoding.h"
12 #include "Framer.h"
13
```

```
14 static uint64_t counts[MAX_CANDIDATE + 1];
15
16 int main(int argc, char *argv[]) {
17     if (argc != 2)      //Test for correct number of arguments
18         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
19
20     int servSock = SetupTCPServerSocket(argv[1]);
21     //servSock is now ready to use to accept connections
22
23     for (;;) {          //Run forever
24
25         //Wait for a client to connect
26         int clntSock = AcceptTCPConnection(servSock);
27
28         //Create an input stream from the socket
29         FILE *channel = fdopen(clntSock, "r+");
30         if (channel == NULL)
31             DieWithSystemMessage("fdopen() failed");
32
33         //Receive messages until connection closes
34         int mSize;
35         uint8_t inBuf[MAX_WIRE_SIZE];
36         VoteInfo v;
37         while ((mSize = GetNextMsg(channel, inBuf, MAX_WIRE_SIZE)) > 0) {
38             memset(&v, 0, sizeof(v));           //Clear vote information
39             printf("Received message (%d bytes)\n", mSize);
40             if (Decode(inBuf, mSize, &v)) {    //Parse to get VoteInfo
41                 if (!v.isResponse) {        //Ignore non-requests
42                     v.isResponse = true;
43                     if (v.candidate >= 0 && v.candidate <= MAX_CANDIDATE) {
44                         if (!v.isInquiry)
45                             counts[v.candidate] += 1;
46                         v.count = counts[v.candidate];
47                     } //Ignore invalid candidates
48                 }
49                 uint8_t outBuf[MAX_WIRE_SIZE];
50                 mSize = Encode(&v, outBuf, MAX_WIRE_SIZE);
51                 if (PutMsg(outBuf, mSize, channel) < 0) {
52                     fputs("Error framing/outputting message\n", stderr);
53                     break;
54                 } else {
55                     printf("Processed %s for candidate %d;current count is %llu.\n",
56                            (v.isInquiry ? "inquiry" : "vote"), v.candidate, v.count);
57                 }
58             }
59         }
60     }
61 }
```

```

58     fflush(channel);
59 } else {
60     fputs("Parse error, closing connection.\n", stderr);
61     break;
62 }
63 }
64 puts("Client finished");
65 fclose(channel);
66 } //Each client
67 //NOT REACHED
68 }

```

---

(1) 访问库函数和常量：第 1~12 行。

(2) 本地声明：第 14 行。

用于存储投票计数的数组。

(3) 声明和参数处理：第 16~18 行。

(4) 建立侦听套接字：第 20 行。

(5) 反复接受和处理客户：第 23~66 行。

- 等待连接：第 26 行。

AcceptTCPConnection() 打印客户信息。

- 在流中包装套接字：第 29~31 行。

- 接收并处理消息，直至连接关闭：第 34~63 行。

**注意：**服务器使用与客户相同的代码来解析、成帧和编码消息。

(6) 关闭客户连接：第 65 行。

### 5.2.1 成帧

应用程序协议通常处理具体的消息，它们被视作字段的集合。成帧指的是允许接收者定位消息（或其一部分）的边界的普通问题。无论信息是编码为文本、多字节二进制数字，还是这二者的组合，当消息的接收者接收到所有的消息时，应用程序协议必须指定它如何确定消息的编码形式。

当然，如果作为 UDP 数据报的有效载荷发送完整的消息，问题就很简单：数据报套接字上的每个发送/接收操作都只涉及一条消息，因此接收者可以准确知道消息结束于何处。不过，对于通过 TCP 套接字发送的消息，情况可能更复杂，因为 TCP 没有消息边界的概念。如果消息中的字段全都具有固定的大小并且消息由固定数量的字段组成，那么就可以提前知道消息的大小，并且接收者可以简单地把期望的字节数读入缓冲区中（TCPEchoClient.c 中使用了这种技术，其中我们知道服务器所期望的字节数）。不过，当消

息的长度可变时——例如，如果它包含一些长度可变的任意文本字符串——我们就不能提前知道要读取多少个字节。

如果接收者尝试从套接字接收的字节数多于消息中的字节数时，可能会发生两件事情中的其中一件事情。如果信道中没有其他消息，接收者将会阻塞，并会被阻止处理消息；如果发送者也被阻止等待应答，结果将出现死锁（deadlock）：连接的每一端都等待另一端发送更多的信息。另一方面，如果信道中已经有另外一条消息，接收者可能读取它的一些或全部内容作为第一条消息的一部分，从而导致其他类型的错误。因此，在使用 TCP 套接字时，成帧是一个重要的考虑事项。

注意，一些相同的考虑事项也适用于查找消息中的各个字段的边界：接收者需要知道一个字段的结束位置和另一个字段的开始位置。因此，我们在这里所说的关于对消息进行成帧处理的几乎所有的一切也适用于字段。不过，如上面所指出的，如果把定位消息末尾的问题与把它解析成字段的问题分开进行处理，就会得到最规则的代码。

两种常规技术可以让接收者明确地查找消息的末尾：

- 基于定界符：通过唯一标记（unique marker）指示消息的末尾，特别是发送者紧接着在数据后面传输的协商好的字节（或者字节序列）。
- 显式长度：在长度可变的字段或消息前附加一个长度（length）字段，指出它包含多少个字节。长度字段一般具有固定的大小；这会限制可以成帧的消息的最大大小。

基于定界符的方法的一个特例可以用于在 TCP 连接上发送的最后一条消息：发送者在发送消息后简单地关闭连接的发送端。在接收者读取消息的最后一个字节之后，它会接收到一个流结束的指示（即 `recv()` 返回 0 或者 `fread()` 返回 EOF），从而可以指示它已经到达了消息的末尾。

基于定界符的方法通常用于编码为文本的消息：定义特定的字符或字符序列来标记消息的末尾。接收者只需扫描输入（作为字符）以寻找定界符序列；它会返回定界符之前的字符串。其缺点是：消息自身绝对不能包含定界符；否则，接收者将会过早地找到消息的末尾。利用基于定界符的成帧方法，某个人必须负责确保这个前提条件得到满足。幸运的是，所谓的填充（stuffing）技术允许修改自然地出现在消息中的定界符，使得接收者不会将其识别为定界符。发送端对出现在文本中的定界符执行转换；接收者在扫描定界符时，也能识别出经过转换的定界符并恢复它们，使得输出消息与原始消息相匹配。这类技术的缺点是发送者和接收者双方都必须扫描消息的每个字节。

基于长度的方法更简单一些，但是需要对消息的大小设置一个已知的上限。发送者首先要确定消息的长度，将其编码为一个整数，并把结果作为消息的前缀。消息长度的上限确定了用来编码该长度所需要的字节数：如果消息所包含的字节数总是少于 256，则需要 1 个字节；如果它们总是少于 65536 字节，则需要 2 个字节，等等。

模块 `DelimFramer.c` 使用“换行”符（"\n"，字节值 10）作为定界符，实现了基于定界

符的成帧技术。我们的 PutMsg()方法没有进行填充，而是当要成帧的字节序列已经包含定界符时就简单地失败（灾难性地）。GetNextMsg()方法扫描流，把每个字节都复制到缓冲区中，直至它读取定界符或者用光了空间为止。它会返回存放在缓冲区中的字节数。如果消息被截短，也就是说该方法在没有遇到定界符的情况下返回一个填满的缓冲区，那么返回的计数将是负数。如果消息中的一些字节被累计并且在没有发现定界符时流终止，就认为出现错误并且会返回一个负值（即这个协议不接受将流末尾作为定界符）。因此，可以将空的但是正确地成帧的消息（返回长度 0）与流终止区分开。

### DelimFramer.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include "Practical.h"
5
6 static const char DELIMITER = '\n';
7
8 /* Read up to bufSize bytes or until delimiter, copying into the given
9  * buffer as we go.
10 * Encountering EOF after some data but before delimiter results in failure.
11 * (That is: EOF is not a valid delimiter.)
12 * Returns the number of bytes placed in buf (delimiter NOT transferred).
13 * If buffer fills without encountering delimiter, negative count is returned.
14 * If stream ends before first byte, -1 is returned.
15 * Precondition: buf has room for at least bufSize bytes.
16 */
17 int GetNextMsg(FILE *in, uint8_t *buf, size_t bufSize) {
18     int count = 0;
19     int nextChar;
20     while (count < bufSize) {
21         nextChar = getc(in);
22         if (nextChar == EOF) {
23             if (count > 0)
24                 DieWithUserMessage("GetNextMsg()", "Stream ended prematurely");
25             else
26                 return -1;
27         }
28         if (nextChar == DELIMITER)
29             break;
30         buf[count++] = nextChar;
31     }
32     if (nextChar != DELIMITER) { //Out of space: count==bufSize

```

```

33     return -count;
34 } else { //Found delimiter
35     return count;
36 }
37 }
38
39 /* Write the given message to the output stream, followed by
40 * the delimiter. Return number of bytes written, or -1 on failure.
41 */
42 int PutMsg(uint8_t buf[], size_t msgSize, FILE *out) {
43     //Check for delimiter in message
44     int i;
45     for (i = 0; i < msgSize; i++)
46         if (buf[i] == DELIMITER)
47             return -1;
48     if (fwrite(buf, 1, msgSize, out) != msgSize)
49         return -1;
50     fputc(DELIMITER, out);
51     fflush(out);
52     return msgSize;
53 }

```

(1) 声明常量定界符值: 第 6 行。

(2) 输入方法 GetNextMsg(): 第 17~37 行。

- 初始化字节计数: 第 18 行。
- 进行迭代, 直至缓冲区填满或者遇到 EOF: 第 20~37 行。

我们从输入流中获取下一个字节, 把它与 EOF 作比较, 然后与定界符作比较。对于 EOF, 如果不完整的消息位于缓冲区中, 我们就中止, 否则就返回 -1。对于定界符, 我们会跳出循环。如果缓冲区填满 (`count==bufSize`), 就返回读取的字节数的负值, 指示信道不是空的。

- 返回传输到缓冲区的字节计数: 第 35 行。

(3) 输出方法 PutMsg(): 第 42~53 行。

- 扫描输入消息, 寻找定界符: 第 43~47 行。

如果找到定界符, (我们将相当无助地) 终止程序。

- 把消息写到输出流: 第 48 行。
- 把定界符字节写到输出流: 第 50 行。
- 冲洗输出流: 第 47 行。

这确保通过底层套接字发送消息。

尽管我们的实现使得可以相当容易地更改用作定界符的单个字符, 但是一些协议利用

了多字符定界符。HTTP 协议（在万维网中使用它）使用由 4 字符序列\r\n\r\n 定界的文本编码的消息。可以扩展基于定界符的成帧模块以支持多字符定界符和处理填充，我们把它留作一个练习。

模块 LengthFramer.c 使用基于长度的成帧方法实现了成帧接口。它适用于其长度最多为 65535 ( $2^{16} - 1$ ) 个字节的消息。PutMsg()方法确定给定消息的长度，并把它作为 2 字节的大端整数写到输出流，其后接着完整的消息。在接收端，fread()方法用于把长度读作一个整数；在把它转换为主机字节顺序之后，将会从信道中读取那么多的字节。注意：利用这种成帧方法，发送者不必检查将要成帧的消息的内容；它只需检查消息没有超过长度限制。

### LengthFramer.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <netinet/in.h>
5 #include "Practical.h"
6
7 /* Read 2-byte length and place in big-endian order.
8  * Then read the indicated number of bytes.
9  * If the input buffer is too small for the data, truncate to fit and
10 * return the negation of the *indicated* length. Thus a negative return
11 * other than -1 indicates that the message was truncated.
12 * (Ambiguity is possible only if the caller passes an empty buffer.)
13 * Input stream is always left empty.
14 */
15 int GetNextMsg(FILE *in, uint8_t *buf, size_t bufSize) {
16     uint16_t mSize = 0;
17     uint16_t extra = 0;
18
19     if (fread(&mSize, sizeof(uint16_t), 1, in) != 1)
20         return -1;
21     mSize = ntohs(mSize);
22     if (mSize > bufSize) {
23         extra = mSize - bufSize;
24         mSize = bufSize; //Truncate
25     }
26     if (fread(buf, sizeof(uint8_t), mSize, in) != mSize) {
27         fprintf(stderr, "Framing error: expected %d, read less\n", mSize);
28         return -1;
29     }
30     if (extra > 0) { //Message was truncated
```

```

31     uint8_t waste[BUFSIZE];
32     fread(waste, sizeof(uint8_t), extra, in); //Try to flush the channel
33     return -(mSize + extra); //Negation of indicated size
34 } else
35     return mSize;
36 }
37
38 /* Write the given message to the output stream, followed by
39 * the delimiter. Precondition: buf[] is at least msgSize.
40 * Returns -1 on any error.
41 */
42 int PutMsg(uint8_t buf[], size_t msgSize, FILE *out) {
43     if (msgSize > UINT16_MAX)
44         return -1;
45     uint16_t payloadSize = htons(msgSize);
46     if ((fwrite(&payloadSize, sizeof(uint16_t), 1, out) != 1) || (fwrite(buf,
47         sizeof(uint8_t), msgSize, out) != msgSize))
48         return -1;
49     fflush(out);
50     return msgSize;
51 }

```

(1) 输入方法 GetNextMsg(): 第 15~36 行。

- 读取前缀长度: 第 19~20 行。  
fread()方法把 2 个字节读入无符号 16 位整数 mSize 中。
- 转换为主机字节顺序: 第 21 行。
- 根据需要截短消息: 第 22~25 行。

如果指示的大小大于调用者提供的缓冲区, 就截短消息使之能放得下, 并且记住它, 使得我们可以通过返回值来指示所做的事情。

- 读取消息: 第 26 行。
- 冲洗信道: 第 30~34 行。

如果由于缓冲区填满而过早返回, 就可以从信道中删除多余的字节, 并返回头部大小的负数。

- 返回大小: 第 35 行。

(2) 输出方法 PutMsg(): 第 42~51 行。

- 验证输入长度: 第 43~44 行。

由于我们使用 2 字节的无符号长度字段, 因此长度不能超过 65535 (UINT16\_MAX 的值)。

- 把长度转换为网络字节顺序: 第 45 行。

- 输出长度和消息：第 46~48 行。
- 冲洗以确保消息被发送：第 49 行。

### 5.2.2 基于文本的消息编码

现在，我们转向将投票消息表示为文本。由于我们只需表示数字和两个指示器，因此可以使用基本的 C 语言字符集 US-ASCII。消息包含由一个或多个 ASCII 空格字符（十进制值 32）隔开的文本字段。消息的开头是一个所谓的“魔术字符串”——即 ASCII 字符序列，它允许接收者快速将消息识别为投票协议的消息，并且与通过网络到达的随机垃圾消息区分开。投票/质询布尔值被编码成字符形式，其中字符“v”表示投票消息，“i”表示查询消息。由字符“R”指示响应的状态是响应。然后接着候选人 ID，再接着投票计数，它们都编码为十进制数字的字符串。模块 VoteEncodingText.c 实现了这种基于文本的编码。

#### **VoteEncodingText.c**

```

1  /* Routines for Text encoding of vote messages.
2   * Wire Format:
3   * "Voting <v|i> [R] <candidate ID> <count>"
4   */
5  #include <string.h>
6  #include <stdint.h>
7  #include <stdbool.h>
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include "Practical.h"
12 #include "VoteProtocol.h"
13
14 static const char *MAGIC = "Voting";
15 static const char *VOTESTR = "v";
16 static const char *INQSTR = "i";
17 static const char *RESPONSESTR = "R";
18 static const char *DELIMSTR = " ";
19 enum {
20     BASE = 10
21 };
22
23 /* Encode voting message info as a text string.
24  * WARNING: Message will be silently truncated if buffer is too small!
25  * Invariants (e.g. 0 <= candidate <= 1000) not checked.
26  */
27 size_t Encode(const VoteInfo *v, uint8_t *outBuf, const size_t bufSize) {

```

```
28 uint8_t *bufPtr = outBuf;
29 long size = (size_t) bufSize;
30 int rv = snprintf((char *) bufPtr, size, "%s %c %s %d", MAGIC,
31     (v->isInquiry ? 'i':'v'), (v->isResponse ? "R" : ""), v->candidate);
32 bufPtr += rv;
33 size -= rv;
34 if (v->isResponse) {
35     rv = snprintf((char *) bufPtr, size, " %llu", v->count);
36     bufPtr += rv;
37 }
38 return (size_t) (bufPtr - outBuf);
39 }
40
41 /* Extract message information from given buffer.
42  * Note: modifies input buffer.
43 */
44 bool Decode(uint8_t *inBuf, const size_t mSize, VoteInfo *v) {
45
46     char *token;
47     token = strtok((char *) inBuf, DELIMSTR);
48     //Check for magic
49     if (token == NULL || strcmp(token, MAGIC) != 0)
50         return false;
51
52     //Get vote/inquiry indicator
53     token = strtok(NULL, DELIMSTR);
54     if (token == NULL)
55         return false;
56
57     if (strcmp(token, VOTESTR) == 0)
58         v->isInquiry = false;
59     else if (strcmp(token, INQSTR) == 0)
60         v->isInquiry = true;
61     else
62         return false;
63
64     //Next token is either Response flag or candidate ID
65     token = strtok(NULL, DELIMSTR);
66     if (token == NULL)
67         return false; //Message too short
68
69     if (strcmp(token, RESPONSESTR) == 0) { //Response flag present
70         v->isResponse = true;
71         token = strtok(NULL, DELIMSTR); //Get candidate ID
```

```
72     if (token == NULL)
73         return false;
74     } else { //No response flag; token is candidate ID;
75         v->isResponse = false;
76     }
77     //Get candidate #
78     v->candidate = atoi(token);
79     if (v->isResponse) { //Response message should contain a count value
80         token = strtok(NULL, DELIMSTR);
81         if (token == NULL)
82             return false;
83         v->count = strtoll(token, NULL, BASE);
84     } else {
85         v->count = 0L;
86     }
87     return true;
88 }
```

Encode()方法使用 snprintf()构造一个包含消息中所有字段的字符串，并用空白隔开它们。仅当调用者提供的空间不足以存放字符串时，它才会失败。

Decode()方法使用 strtok()方法把接收到的消息分解为令牌（字段）。strtok()库函数接受一个指针和一个字符串，前者指向一个字符数组，后者包含要解释为定界符的字符。在第一次调用它时，它会返回最大的初始子串，它完全由不属于定界符字符串中的字符组成；用 null 字节替换该字符串的尾随定界符。在以后利用 NULL 作为第一个参数调用它时，将从原始字符串中把令牌从左边带到右边，直至没有更多的令牌为止，此时就返回 NULL。

Decode()首先寻找“魔术”字符串；如果它不在消息的开头，该方法就会失败并且返回 FALSE。注意，这阐释了关于实现协议的非常重要的一点：永远不要对来自网络的任何输入做出任何假定。你的程序必须总是为任何可能的输入做好准备，并优雅地处理它们。在这种情况下，如果一些期望的部分不存在或者不正确地格式化，Decode()方法将会简单地忽略消息中余下的内容并返回 FALSE。<sup>1</sup>否则，Decode()将逐令牌获取字段，并使用库函数 atoi()和 strtoll()把令牌转换为整数。

### 5.2.3 二进制消息编码

下面我们将展示另一种对投票协议消息进行编码的方法。与基于文本的格式相反，二进制格式使用固定大小的消息。每条消息都开始于一个 1 字节的字段，它在其高阶 6 位中

<sup>1</sup> 这也阐明了为什么最好在解析前先进行成帧处理的一个关键原因：当一条消息只被接收到一部分时从解析错误中恢复要复杂得多，因为接收者必须与发送者之间进行“回溯同步”。

包含“魔术”值 010101。与文本格式一样，这一点少量的冗余性为接收者接收到正确的投票消息提供了一定程度的保证。第一个字节的两个低阶位对两个布尔值进行了编码；注意以前在 5.1.8 节中说明了按位“或”操作的使用，它们用于设置标志。消息的第二个字节总是包含 0（它实际上是填充），第三个和第四个字节包含 candidateID 值。只有响应消息的最后 8 个字节才包含投票计数。

## **VoteEncodingBin.c**

```

1  /* Routines for binary encoding of vote messages
2   * Wire Format:
3   *
4   *      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
5   * +---+---+---+---+---+---+---+---+---+---+---+---+
6   * |           Magic    |Flags|           ZERO     |
7   * +---+---+---+---+---+---+---+---+---+---+---+---+
8   * |           Candidate ID     |
9   * +---+---+---+---+---+---+---+---+---+---+---+---+
10  * |
11  * |           Vote Count (only in response) |
12  * |
13  * |
14  * +---+---+---+---+---+---+---+---+---+---+---+---+
15  *
16 */
17
18 #include <string.h>
19 #include <stdbool.h>
20 #include <stdlib.h>
21 #include <stdint.h>
22 #include <netinet/in.h>
23 #include "Practical.h"
24 #include "VoteProtocol.h"
25
26 enum {
27     REQUEST_SIZE = 4,
28     RESPONSE_SIZE = 12,
29     COUNT_SHIFT = 32,
30     INQUIRE_FLAG = 0x0100,
31     RESPONSE_FLAG = 0x0200,
32     MAGIC = 0x5400,
33     MAGIC_MASK = 0xfc00
34 };
35

```

```
36 typedef struct voteMsgBin voteMsgBin;
37
38 struct voteMsgBin {
39     uint16_t header;
40     uint16_t candidateID;
41     uint32_t countHigh;
42     uint32_t countLow;
43 };
44
45 size_t Encode(VoteInfo *v, uint8_t *outBuf, size_t bufSize) {
46     if ((v->isResponse && bufSize < sizeof(voteMsgBin)) || bufSize < 2
47         * sizeof(uint16_t))
48         DieWithUserMessage("Output buffer too small", "");
49     voteMsgBin *vm = (voteMsgBin *) outBuf;
50     memset(outBuf, 0, sizeof(voteMsgBin)); //Be sure
51     vm->header = MAGIC;
52     if (v->isInquiry)
53         vm->header |= INQUIRE_FLAG;
54     if (v->isResponse)
55         vm->header |= RESPONSE_FLAG;
56     vm->header = htons(vm->header); //Byte order
57     vm->candidateID=htons(v->candidate); //Know it will fit, by invariants
58     if (v->isResponse) {
59         vm->countHigh = htonl(v->count >> COUNT_SHIFT);
60         vm->countLow = htonl((uint32_t) v->count);
61         return RESPONSE_SIZE;
62     } else {
63         return REQUEST_SIZE;
64     }
65 }
66
67 /* Extract message info from given buffer.
68 * Leave input unchanged.
69 */
70 bool Decode(uint8_t *inBuf, size_t mSize, VoteInfo *v) {
71
72     voteMsgBin *vm = (voteMsgBin *) inBuf;
73
74     //Attend to byte order; leave input unchanged
75     uint16_t header = ntohs(vm->header);
76     if ((mSize < REQUEST_SIZE) || ((header & MAGIC_MASK) != MAGIC))
77         return false;
78     /* message is big enough and includes correct magic number */
79     v->isResponse = ((header & RESPONSE_FLAG) != 0);
```

```

80     v->isInquiry = ((header & INQUIRE_FLAG) != 0);
81     v->candidate = ntohs(vm->candidateID);
82     if (v->isResponse && mSize >= RESPONSE_SIZE) {
83         v->count = ((uint64_t) ntohs(vm->countHigh) << COUNT_SHIFT)
84             | (uint64_t) ntohs(vm->countLow);
85     }
86     return true;
87 }

```

在这个版本中，Decode()方法的工作特别简单——它只是简单地把消息中的值复制到VoteInfo结构中，并自始至终进行字节顺序转换。

#### 5.2.4 综合应用

为了获得可以工作的投票服务器，我们只需把VoteServerTCP.c、两个成帧模块之一、两个编码模块之一以及辅助模块DieWithMessage.c、TCPClientUtility.c、TCPServerUtility.c和AddressUtility.c一起进行编译即可。例如：

```

$ gcc -std=gnu99 -o vs VoteServerTCP.c DelimFramer.c VoteEncodingBin.c \
DieWithMessage.c TCPServerUtility.c AddressUtility.c
$ gcc -std=gnu99 -o vc VoteClientTCP.c DelimFramer.c VoteEncodingBin.c \
DieWithMessage.c TCPClientUtility.c

```

成帧方法和编码的全部4种可能的组合都将会工作——假定客户和服务器使用相同的组合！

### 5.3 小结

我们已经看到如何将基本类型表示为字节序列以便进行“线上”传输。我们还考虑了编码文本字符串的一些微妙之处，以及一些成帧和解析消息的基本方法。我们还见到了面向文本和二进制编码协议的例子。

这里可能值得重申一下我们在前言中所说的：本章内容决不会使你成为专家！那需要大量的经验。但是本章中的代码能够作为你进一步探索的起点。

### 练习题

- 如果底层硬件平台采用的是小端顺序，而“网络”字节顺序是大端顺序，存在使htonl()和ntohl()的实现有所不同的任何原因吗？

2. 编辑函数 `uint64_t htonl (uint64_t val)`，它用于把 64 位整数从小端字节顺序转换为大端字节顺序。
3. 编写 `BruteForceEncoding.c` 中给出的方法的小端版本，即 `EncodeLittleEndian()` 和 `DecodeLittleEndian()`。
4. 编写方法 `EncodeBigEndianSigned()` 和 `DecodeBigEndianSigned()`，它们返回有符号的值（输入缓冲区仍然是无符号类型。提示：使用显式的类型强制转换）。
5. 仅当满足多个前提条件（比如  $0 \leq size \leq 8$ ）时，`BruteForceEncoding.c` 中的 `EncodeIntBigEndian()` 方法才会工作。修改该方法，测试这些前提条件，如果违反了其中的任何前提条件，就返回一个错误指示。与依靠调用者建立前提条件相比，使程序检查前提条件有什么优点和缺点？
6. 假定所有字节值都有同样的可能，那么包含随机位的消息通过投票协议的二进制编码中的“魔术测试”的概率有多大？假设给期望二进制编码的 `voteMsg` 的程序发送 ASCII 编码的文本消息，位于消息开头的哪些字符可以让消息通过“魔术测试”？
7. 假设我们在构建投票客户时使用了 `DelimFraming.c` 和 `VoteEncodingBin.c`。描述客户在哪些情况下将无法发送消息。
8. 扩展基于定界符的成帧实现以执行“字节填充”，使得可以传输包含定界符的消息，而无需 `PutMsg()` 的调用者关心它。也就是说，成帧模块会透明地处理包含定界符的消息（参见任何优秀的网络教材，以了解该算法）。
9. 扩展基于定界符的成帧实现，以处理任意的多字节定界符。确保你的实现是高效的（注意：这个问题比较重要！原始方法在最坏情况下的运行效率非常低）。
10. 如果调用者无法提供足够大的缓冲区，那么 `GetNextMsg()` 的两种实现都会截短接收到的消息。对于这两种实现，考虑在发生这种情况之后下一次调用 `GetNextMsg()` 时的行为。两种情况下的行为相同吗？如果不同，给出修改建议，使得两种实现在各种情况下都具有相同的行为方式。

# 第 6 章

## 超越基本的套接字编程

我们的客户和服务器示例演示了用于套接字编程的基本模式。下一步是把这些思想集成进像多任务处理、信令和广播这样的编程模型中。我们将在标准的 UNIX 编程的环境中演示这些原理；不过，大多数现代操作系统都支持类似的特性（例如，进程和线程）。

### 6.1 套接字选项

TCP/IP 协议的开发者花了大量的时间考虑可以满足大多数应用程序的默认行为（如果你怀疑这一点，可以阅读 RFC 1122 和 1123，它们基于多年的经验极其详细地描述了 TCP/IP 协议的实现的建议行为）。对于大多数应用程序，设计者做了一项非常好的工作；不过，很少有一种“万能式”的设计确实能够适合各种应用。例如，每个套接字都有一个关联的接收缓冲区。它应该是多大？每种实现都有默认的大小；不过，这个值可能并不总是适合于你的应用程序（另请参见 7.1 节）。套接字行为的这个特定的方面以及许多其他的方面与套接字选项（socket option）相关联：可以通过修改关联的套接字选项的值，更改套接字的接收缓冲区大小。函数 `getsockopt()` 和 `setsockopt()` 分别允许查询和设置套接字选项值。

---

```
int getsockopt(int socket, int level, int optName, void *optVal,
               socklen_t *optLen)
int setsockopt(int socket, int level, int optName, const void *optVal,
               socklen_t optLen)
```

---

对于这两个函数，`socket` 必须是由 `socket()` 分配的套接字描述符。可用的套接字选项被分成与协议栈的各层对应的层次；第二个参数指示正在处理的选项的层次。一些选项与协议无关，因此可以被套接字层自身处理（`SOL_SOCKET`），一些选项特定于传输协议（`IPPROTO_TCP`），还有一些选项由网络之间的协议处理（`IPPROTO_IP`）。选项本身由整数 `optName` 指定，它总是使用系统定义的常量指定的。参数 `optVal` 是一个指向缓冲区的指针。

对于 `getsockopt()`, 这个选项的当前值将由实现放入那个缓冲区中; 而对于 `setsockopt()`, 实现中的套接字选项将被设置为缓冲区中的值。在两个调用中, `optLen` 指定缓冲区的长度, 它对于正在处理的特定选项必须是正确的。注意: 在 `getsockopt()` 中, `optLen` 是一个输入/输出型参数, 最初指向一个包含缓冲区大小的整数; 在返回时, 所指向的整数则包含选项值的大小。下面的代码段演示了如何获取套接字的接收缓冲区的配置大小(以字节为单位), 然后把它增大一倍:

```

int recvBufferSize;
//Retrieve and print the default buffer size
int sockOptSize = sizeof(recvBufferSize);
if(getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &recvBufferSize, &sockOptSize)<0)
DieWithSystemMessage("getsockopt() failed");
printf("Initial Receive Buffer Size: %d\n", recvBufferSize);

//Double the buffer size
recvBufferSize *= 2;
if(setsockopt(sock, SOL_SOCKET, SO_RCVBUF,
    &recvBufferSize, sizeof(recvBufferSize)) < 0)
DieWithSystemMessage("setsockopt() failed");

```

注意: 不保证传递给 `setsockopt()` 的值是套接字缓冲区的新大小, 即使调用显然成功也是如此。相反, 最好把它视作用于“提示”系统用户所期望的值; 毕竟, 系统必须为所有用户管理资源, 并且可能会在调整缓冲区大小时考虑其他因素。

表 6.1 显示了每个层级的一些常用的选项, 包括 `optVal` 指向的缓冲区的说明和数据类型。

表 6.1 套接字选项

<b>SOL_SOCKET</b>			
选 项 名 称	类 型	值	说 明
SO_BROADCAST	int	0、1	允许广播
SO_KEEPALIVE	int	0、1	支持实时消息(如果被协议实现的话)
SO_LINGER	linger{}	时间	延迟 <code>close</code> 返回以等待确认的时间(参见 7.4 节)
SO_RCVBUF	int	字节数	套接字接收缓冲区中的字节数(参见上面的代码和 7.1 节)
SO_RCVLOWAT	int	字节数	将导致 <code>recv</code> 返回的最少可用字节数
SO_REUSEADDR	int	0、1	允许绑定(在某些条件下)到地址或端口
SO_SNDBUF	int	字节数	套接字发送缓冲区中的字节数(参见 7.1 节)
SO_SNDLOWAT	int	字节数	要发送的最少字节数

续表

**IPPROTO\_TCP**

选项名称	类型	值	说明
TCP_MAX	int	秒数	实时消息之间的秒数
TCP_NODELAY	int	0、1	禁止用于数据合并的 Nagle 算法中的延迟

**IPPROTO\_IP**

选项名称	类型	值	说明
IP_TTL	int	0~255	单播 IP 分组的生存期
IP_MULTICAST_TTL	unsigned char	0~255	多播 IP 分组的生存期（参见 6.6.2 节中的 MulticastSender.c）
IP_MULTICAST_LOOP	int	0、1	允许多播套接字接收它发送的分组
IP_ADD_MEMBERSHIP	ip_mreq{}	组地址	允许接收寻址到指定多播组的分组（参见 6.6.2 节中的 MulticastReceiver.c）——只能设置
IP_DROP_MEMBERSHIP	ip_mreq{}	组地址	禁止接收寻址到指定多播组的分组——只能设置

**IPPROTO\_IPv6**

选项名称	类型	值	说明
IPv6_V6ONLY	int	0、1	限制 IPv6 套接字只进行 IPv6 通信
IPv6_UNICAST_HOPS	int	-1~255	单播 IP 分组的生存期
IPv6_MULTICAST_HOPS	int	-1~255	多播 IP 分组的生存期（参见 6.6.2 节中的 MulticastSender.c）
IPv6_MULTICAST_LOOP	u_int	0、1	允许多播套接字接收它发送的分组
IPv6_JOIN_GROUP	ipv6_mreq{}	组地址	允许接收寻址到指定多播组的分组（参见 6.6.2 节中的 MulticastReceiver.c）——只能设置
IPv6_LEAVE_GROUP	ipv6_mreq{}	组地址	禁止接收寻址到指定多播组的分组——只能设置

## 6.2 信号

信号提供了一种机制，用于通知程序发生了某些事件——例如，用户输入了“中断”字符，或者计时器到期。一些事件（以及通知）可能异步地（asynchronously）发生，这意味着会将通知递送到程序，而不管它正在执行哪条语句。当把信号递送到程序时，将发生以下 4 件事情之一：

- (1) 忽略信号。进程永远不知道信号被递送。
- (2) 程序被操作系统强行终止。

(3) 程序执行被中断，并且执行程序(及其一部分)指定的信号处理例程(signal-handling routine)。这种执行发生在不同于程序的主线程的控制线程中，使得程序不必立即知道它。

(4) 信号被阻塞(block)，也就是说，阻止它起到任何作用，直至程序采取行动以允许递送它为止。每个进程都有一个掩码(mask)，指示目前在该线程中阻塞了哪些信号(实际上，程序中的每个线程都可以具有它自己的信号掩码)。

UNIX 具有许多不同的信号，每个信号都指示发生了不同类型的事情。每个信号都具有系统定义的默认行为(default behavior)，它是上面列出的两种可能的情况之一。例如，终止是 SIGINT 的默认行为，当通过该进程的控制终端接收到终止字符(通常是 Ctrl+C)时就递送它。

信号是一个复杂的主题，对它的详细讨论超出了本书的范围。不过，在套接字编程的环境中会频繁遇到一些信号。而且，在 TCP 套接字上发送数据的任何程序都必须显式处理 SIGPIPE 以便保证健壮性。因此，我们介绍了处理信号的基本知识，重点关注以下 5 种信号：

类 型	触 发 事 件	默 认 行 为
SIGALRM	报警计时器到期	终止
SIGCHLD	子进程退出	忽略
SIGINT	中断字符(Ctrl+C)输入	终止
SIGIO	套接字为 I/O 做好准备	忽略
SIGPIPE	尝试写到关闭的套接字	终止

应用程序可以使用 `sigaction()` 更改特定信号的默认行为<sup>1</sup>：

---

```
int sigaction (int whichSignal, const struct sigaction *newAction,
               struct sigaction *oldAction)
```

---

如果成功，`sigaction()` 就会返回 0，如果失败，则返回 -1；不过，有关其语义的细节要复杂一点。

每种信号都由一个整型常量标识；`whichSignal` 指定要更改其行为的信号。`newAction` 参数指向一个 `sigaction` 结构，它定义了给定信号类型的新行为；如果指针 `oldAction` 不是 `null` 指针，就会把描述给定信号的以前行为的 `sigaction` 结构复制给它，如下所示：

```
struct sigaction {
    void (*sa_handler)(int); //Signal handler
    sigset_t sa_mask; //Signals to be blocked during handler execution
```

---

<sup>1</sup> 对于一些信号，不能更改它们的默认行为，也不能阻塞信号；不过，对于我们考虑的 5 种信号，则不存在这种情况。

```
    int sa_flags; //Flags to modify default behavior  
};
```

字段 `sa_handler`(其类型为“指向函数的指针,该函数带有一个整型参数并且返回 `void`”)用于控制当递送信号(即未屏蔽它)时会发生前三种可能的情况中的哪一种。如果它的值是特殊常量 `SIG_IGN`,就会忽略信号。如果它的值是 `SIG_DFL`,就会使用该信号的默认行为。如果它的值是函数的地址(可保证它不同于两个常量),就会利用指示所递送信号的参数调用该函数(如果把相同的处理函数用于多个信号,参数就可用于确定哪个信号引发了调用)。

从以下意义上讲,信号可以“嵌套”:在处理一个信号时,可以递送另一个信号。如你可能想到的,这可能变得相当复杂。幸运的是,`sigaction()`机制允许在处理指定的信号时临时阻塞一些信号(除了那些已经被进程的信号掩码阻塞的信号之外)。字段 `sa_mask` 指定在处理 `whichSignal` 时要阻塞的信号;仅当 `sa_handler` 不是 `SIG_IGN` 或 `SIG_DFL` 时,它才是有意义的。默认情况下,`whichSignal` 总会被阻塞,而不管是否在 `sa_mask` 中反映了它(在一些系统上,在 `sa_flags` 中设置标志 `SA_NODEFER` 允许在处理指定的信号时递送它)。`sa_flags` 字段控制着处理 `whichSignal` 的方式的一些更多的细节;这些细节超出了本书讨论的范围。

`sa_mask` 被实现为一组布尔标志,其中每个标志用于一种信号。可以利用以下 4 个函数操纵这组标志:

---

```
int sigemptyset(sigset_t *set)  
int sigfillset(sigset_t *set)  
int sigaddset(sigset_t *set, int whichSignal)  
int sigdelset(sigset_t *set, int whichSignal)
```

---

`sigfillset()` 和 `sigemptyset()` 分别用于设置和清除给定集合中的所有标志。`sigaddset()` 和 `sigdelset()` 分别用于设置和清除给定集合中由信号编号指定的各个标志。当成功时,全部 4 个函数都会返回 0;如果失败,则返回 -1。

`SigAction.c` 显示了一个简单的 `sigaction()` 示例,它通过建立信号处理程序然后进入一个无限循环为 `SIGINT` 提供处理程序。当程序接收到中断信号时,处理函数(给 `sigaction()` 提供一个指向它的指针)就会执行并退出程序。

## SigAction.c

---

```
1 #include <stdio.h>  
2 #include <signal.h>  
3 #include <unistd.h>  
4 #include <stdlib.h>
```

```

5 #include "Practical.h"
6
7 void InterruptSignalHandler(int signalType); //Interrupt signal handling function
8
9 int main(int argc, char *argv[]) {
10    struct sigaction handler;           //Signal handler specification structure
11
12    //Set InterruptSignalHandler() as handler function
13    handler.sa_handler = InterruptSignalHandler;
14    //Create mask that blocks all signals
15    if (sigfillset(&handler.sa_mask) < 0)
16        DieWithSystemMessage("sigfillset() failed");
17    handler.sa_flags = 0;               //No flags
18
19    //Set signal handling for interrupt signal
20    if (sigaction(SIGINT, &handler, 0) < 0)
21        DieWithSystemMessage("sigaction() failed for SIGINT");
22
23    for (;;)
24        pause();                      //Suspend program until signal received
25
26    exit(0);
27 }
28
29 void InterruptSignalHandler(int signalType) {
30    puts("Interrupt Received. Exiting program.");
31    exit(1);
32 }

```

(1) 信号处理函数的原型：第 7 行。

(2) 建立信号处理程序：第 10~21 行。

- 指派函数以处理信号：第 13 行。
- 填充信号掩码：第 15~16 行。
- 为 SIGINT 设置信号处理程序：第 20~21 行。

(3) 持续不断地循环，直至接收到 SIGINT 信号：第 23~24 行。

`pause()` 用于挂起进程，直至接收到信号。

(4) 用于处理信号的函数：第 29~32 行。

`InterruptSignalHandler()` 打印一条消息并退出程序。

当要递送的信号被阻塞时，比如说，由于要处理另一个信号，那么会发生什么事情呢？递送将被延迟，直至处理程序执行完成。把这种信号称为挂起的（pending）信号。认识到

信号不会排队很重要——信号要么是挂起的，要么不是。如果在处理信号时把相同的信号递送多次，那么在处理程序完成原始执行之后它只会执行一次。考虑以下情况：其中当已经在执行用于 SIGINT 的信号处理程序时有三个 SIGINT 信号到达。其中第一个 SIGINT 信号会被阻塞；不过，后续两个信号将会丢失。当 SIGINT 信号处理函数完成执行时，系统将把该处理程序再执行一次。我们必须准备好在我们的应用程序中处理这种行为。要查看它的实际应用，可以修改 SigAction.c 中的 InterruptSignalHandler()，如下所示：

```
void InterruptSignalHandler(int ignored) {
    printf("Interrupt Received.\n");
    sleep(3);
}
```

用于 SIGINT 的信号处理程序会睡眠 3 秒钟并返回，而不会退出。现在，当你执行程序时，连续敲击中断键（Ctrl+C）若干次。如果连续敲击中断键两次以上，仍然只会看到两条“Interrupt Received”消息。第一个中断信号调用 InterruptSignalHandler()，它会睡眠 3 秒钟。第二个中断会被阻塞，因为 SIGINT 已经在被处理。第三个和第四个中断会丢失。这警告你将不再能够利用键盘中断键停止程序。你将需要使用 kill 命令，给进程显式发送另一个信号（比如 SIGTERM）。

信号的最重要的方面之一与套接字接口相关。如果在套接字调用（比如 recv() 或 connect()）中阻塞程序时递送信号，并且指定了用于该信号的处理程序，那么一旦处理程序执行完成，套接字调用将返回 -1，并且把 errno 设置为 EINTR。因此，用于捕获和处理信号的程序需要为从可能阻塞的系统调用返回的这些错误做好准备。

在本章后面，我们将遇到上面提到的 4 种信号中的第一种信号。这里，我们将简要描述 SIGPIPE 的语义。考虑以下情况：服务器（或客户）具有一个连接的 TCP 套接字，并且另一端可能突然地、出人意料地关闭连接，比如说，由于程序崩溃。服务器如何查明连接被断开呢？答案是：在它尝试在套接字上发送或接收数据之前，它不能查明这一点。如果它首先尝试接收数据，调用将返回 0。如果它首先尝试发送数据，此时，就会递送 SIGPIPE。因此，将同步（synchronously）而不是异步递送 SIGPIPE（为什么不是仅仅从 send() 返回 -1 呢？参见本章末尾的练习 5）。

这个事实对于服务器特别重要，因为 SIGPIPE 的默认行为是终止程序。因此，没有改变这种行为的服务器可能会被行为失常的客户所终止。服务器总是应该处理 SIGPIPE，使得它们可以检测到客户的消失，并收回用于给它提供服务的任何资源。

## 6.3 非阻塞 I/O

套接字调用的默认行为是：一直阻塞到请求动作完成为止。例如，直到至少接收到一条来自应答服务器的消息，`TCPEchoClient.c`（参见 3.2.1 节）中的 `recv()` 函数才会返回。当然，具有被阻塞函数的进程将会被操作系统挂起。

套接字调用可能由于多种原因而阻塞。如果数据不可用，数据接收函数（`recv()` 和 `recvfrom()`）就会阻塞。如果没有充足的空间用于缓冲传输的数据，TCP 套接字上的 `send()` 可能就会阻塞（参见 7.1 节）。在建立连接之前，用于 TCP 套接字的与连接相关的函数将会阻塞。例如，在客户利用 `connect()` 建立一条连接之前，`TCPEchoServer.c`（参见 3.2.2 节）中的 `accept()` 将会阻塞。较长的往返时间、较高错误率的连接或者缓慢（或停机）的服务器都可能会导致对 `connect()` 的调用阻塞很长的时间。在所有这些情况下，仅当满足了请求之后，函数才会返回。

如果程序在等待调用完成时还具有其他要执行的任务（例如，更新忙碌的光标或者响应用户请求），则会如何？这些程序可能没有时间等待阻塞的系统调用。丢失的 UDP 数据报则会如何处理？在 `UDPEchoClient.c`（参见 4.1 节）中，客户发送一个数据报给服务器，然后等待接收一个响应。如果从客户发送的数据报或者来自服务器的应答数据报丢失，我们的应答客户将会无限期地阻塞。在这种情况下，在一段时间后我们需要使用 `recvfrom()` 解除阻塞，以便允许客户处理数据报丢失。幸运的是，有多种机制可用于控制不想要的阻塞行为。我们在这里将讨论三种这样的解决方案：非阻塞套接字、异步 I/O 和超时。

### 6.3.1 非阻塞套接字

针对不想要的阻塞这个问题的一种显而易见的解决方案是：更改套接字的行为，使得所有的调用都是非阻塞的（nonblocking）。对于这样的套接字，如果请求的操作可以立即完成，调用的返回值就会指示成功，否则，它会指示失败（通常是 -1）。在任何一种情况下，调用都不会无限期地阻塞。就失败而言，我们需要能够区分由于阻塞而导致的失败与其他类型的失败。如果失败是由于调用被阻塞而发生的，系统就会把 `errno` 设置为 `EWOULDBLOCK`<sup>1</sup>，`connect()` 除外，它返回的 `errno` 为 `EINPROGRESS`。

我们可以通过调用 `fcntl()`（“file control”），更改默认的阻塞行为。

---

```
int fcntl(int socket, int command, ...)
```

---

<sup>1</sup> 一些套接字实现会返回 `EAGAIN`。在许多系统上，`EAGAIN` 和 `EWOULDBLOCK` 是相同的错误编号。

顾名思义，这个调用可以用于任何类型的文件：socket 必须是一个有效的文件（或套接字）描述符。要执行的操作由 command 给出，它总是一个系统定义的常量。通过与描述符关联的标志（与套接字选项不同）控制我们想要修改的行为，我们可以利用 F\_GETFL 和 F\_SETFL 命令获取和设置这些标志。在设置套接字标志时，必须在长度可变的参数列表中指定新的标志。控制非阻塞行为的标志是 O\_NONBLOCK。在获取套接字标志时，长度可变的参数列表为空。我们将在下一节中演示非阻塞套接字的使用，其中在 UDPEchoServer-SIGIO.c 中描述了异步 I/O。

非阻塞套接字的这种模型有几种例外情况。对于 UDP 套接字，没有发送缓冲区，因此 send() 和 sendto() 永远不会返回 EWOULDBLOCK。对于除 connect() 之外的所有其他的套接字调用，要么在返回前完成请求的操作，要么所有的操作都不执行。例如，recv() 要么接收来自套接字的数据，要么返回一个错误。非阻塞 connect() 则有所不同。对于 UDP，connect() 简单地为将来的数据传输分配目的地址，使得它永远不会阻塞。对于 TCP，connect() 用于发起 TCP 连接建立。如果在未阻塞的情况下不能完成连接，connect() 就会返回一个错误，把 errno 设置为 EINPROGRESS，指示套接字仍然会致力于建立 TCP 连接。当然，直到连接建立以后，后续的数据发送和接收才会发生。确定连接何时完成超出了本书讨论的范围<sup>1</sup>，因此我们建议应该在调用了 connect() 之后才把套接字设置为非阻塞的。

为了在各个发送和接收操作期间消除阻塞，在一些平台上可以使用一种替代方案。send()、recv()、sendto() 和 recvfrom() 的 flags 参数允许在特定的调用上修改行为的一些方面。一些实现支持 MSG\_DONTWAIT 标志，在 flags 中设置了它的任何调用中都会导致非阻塞的行为。

### 6.3.2 异步 I/O

非阻塞套接字调用的困难之处在于：无法知道何时调用将会成功，而只能通过周期性地试验它，直到它成功为止（一个称为“轮询”的进程）。为什么操作系统不会通知程序何时套接字调用将会成功呢？这样，程序就可以把它的时间花在做其他工作上，直到被通知套接字已为将要发生的事情做好准备。这称为异步 I/O（asynchronous I/O），其工作方式是：当套接字上发生某个与 I/O 相关的事件时，把 SIGIO 信号递送给进程。

安排 SIGIO 涉及三个步骤。首先，使用 sigaction() 通知系统想要如何布置信号。然后，使用 fcntl()，通过使某个进程成为套接字的所有者，确保与套接字相关的信号将被递送给这个进程（由于多个进程可以访问相同的套接字，对于哪个进程应该获取它可能存在歧义）。最后，再次通过 fcntl() 设置一个标志（FASYNC），把套接字标记为预先为异步 I/O 做好准备。

<sup>1</sup> 通常，可以使用 select() 调用检测连接完成，如 6.5 节中所述。

在我们的下一个示例中，我们修改了 UDPEchoServer.c（参见 4.2 节），结合使用异步 I/O 与非阻塞套接字调用。当没有客户需要应答时，修改过的服务器能够执行其他任务。在创建并绑定套接字之后，异步应答服务器无需调用 `recvfrom()` 并阻塞到数据报到达为止，而是为 SIGIO 建立一个信号处理程序并开始做其他工作。当数据报到达时，就把 SIGIO 信号递送给进程，触发处理函数的执行。处理函数调用 `recvfrom()`，发送回接收到的任何数据报作为应答，然后返回，主程序随之继续做它所做的任何工作。我们的描述只详细说明了与原始 UDP 应答服务器在代码上的区别。

## UDPEchoServer-SIGIO.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6 #include <sys/file.h>
7 #include <signal.h>
8 #include <errno.h>
9 #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include "Practical.h"
13
14 void UseIdleTime(); //Execution during idle time
15 void SIGIOHandler(int signalType); //Handle SIGIO
16
17 int servSock; //Socket--GLOBAL for signal handler
18
19 int main(int argc, char *argv[]) {
20
21     if (argc != 2) //Test for correct number of arguments
22         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
23
24     char *service = argv[1]; //First arg: local port
25
26     //Construct the server address structure
27     struct addrinfo addrCriteria; //Criteria for address
28     memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
29     addrCriteria.ai_family = AF_UNSPEC; //Any address family
30     addrCriteria.ai_flags = AI_PASSIVE; //Accept on any address/port
31     addrCriteria.ai_socktype = SOCK_DGRAM; //Only datagram sockets
32     addrCriteria.ai_protocol = IPPROTO_UDP; //Only UDP protocol
```

```
33 //List of server addresses
34 struct addrinfo *servAddr;
35 int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
36 if (rtnVal != 0)
37     DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
38
39 //Create socket for incoming connections
40 servSock = socket(servAddr->ai_family, servAddr->ai_socktype,
41     servAddr->ai_protocol);
42 if (servSock < 0)
43     DieWithSystemMessage("socket() failed");
44
45 //Bind to the local address
46 if (bind(servSock, servAddr->ai_addr, servAddr->ai_addrlen) < 0)
47     DieWithSystemMessage("bind() failed");
48
49 //Free address list allocated by getaddrinfo()
50 freeaddrinfo(servAddr);
51
52 struct sigaction handler;
53 handler.sa_handler = SIGIOHandler; //Set signal handler for SIGIO
54 //Create mask that mask all signals
55 if (sigfillset(&handler.sa_mask) < 0)
56     DieWithSystemMessage("sigfillset() failed");
57 handler.sa_flags = 0;           //No flags
58
59 if (sigaction(SIGIO, &handler, 0) < 0)
60     DieWithSystemMessage("sigaction() failed for SIGIO");
61
62 //We must own the socket to receive the SIGIO message
63 if (fcntl(servSock, F_SETOWN, getpid()) < 0)
64     DieWithSystemMessage("Unable to set process owner to us");
65
66 //Arrange for nonblocking I/O and SIGIO delivery
67 if (fcntl(servSock, F_SETFL, O_NONBLOCK | FASYNC) < 0)
68     DieWithSystemMessage(
69         "Unable to put client sock into non-blocking/async mode");
70
71 //Go off and do real work; echoing happens in the background
72
73 for (;;)
74     UseIdleTime();
75 //NOT REACHED
76 }
```

```

77
78 void UseIdleTime() {
79   puts(".");
80   sleep(3); //3 seconds of activity
81 }
82
83 void SIGIOHandler(int signalType) {
84   ssize_t numBytesRcvd;
85   do { //As long as there is input...
86     struct sockaddr_storage clntAddr; //Address of datagram source
87     size_t clntLen = sizeof(clntAddr); //Address length in-out parameter
88     char buffer[MAXSTRINGLENGTH]; //Datagram buffer
89
90     numBytesRcvd = recvfrom(servSock, buffer, MAXSTRINGLENGTH, 0,
91                           (struct sockaddr *) &clntAddr, &clntLen);
92     if (numBytesRcvd < 0) {
93       //Only acceptable error: recvfrom() would have blocked
94       if (errno != EWOULDBLOCK)
95         DieWithSystemMessage("recvfrom() failed");
96     } else {
97       fprintf(stdout, "Handling client ");
98       PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
99       fputc('\n', stdout);
100
101     ssize_t numBytesSent = sendto(servSock, buffer, numBytesRcvd, 0,
102                               (struct sockaddr *) &clntAddr, sizeof(clntAddr));
103     if (numBytesSent < 0)
104       DieWithSystemMessage("sendto() failed");
105     else if (numBytesSent != numBytesRcvd)
106       DieWithUserMessage("sendto()", "sent unexpected number of bytes");
107   }
108 } while (numBytesRcvd >= 0);
109 //Nothing left to receive
110 }

```

(1) 程序建立和参数解析: 第 1~24 行。

(2) 信号和空闲时间处理程序的原型: 第 14~15 行。

`UseIdleTime()` 模拟 UDP 应答服务器的其他任务。`SIGIOHandler()` 处理 SIGIO 信号。注意: `UseIdleTime()` 必须为任何“缓慢的”系统调用做好准备——比如从终端设备读取数据, 以便返回 -1 作为递送和处理 SIGIO 信号的结果 (在这种情况下, 它应该简单地验证 `errno` 是 EINTR, 并恢复它所做的任何事情)。

(3) 服务器套接字描述符: 第 17 行。

我们给套接字描述符提供了一个全局作用域，使得它可以被 SIGIO 处理函数访问。

#### (4) 建立信号处理：第 52~69 行。

`handler` 是 `sigaction` 结构，它描述了我们想要的信号处理行为。我们填充它，提供处理例程的地址，以及我们想阻塞的信号集合。

- 填充指向想要的处理程序的指针：第 53 行。
  - 指定在处理期间要阻塞的信号：第 55~56 行。
  - 指定如何处理 SIGIO 信号：第 59~60 行。
  - 安排 SIGIO 到达这个进程：第 63~64 行。
- `F_SETOWN` 命令标识要为这个套接字接收 SIGIO 的进程。
- 为非阻塞和异步 I/O 设置标志：第 67~69 行。

最后，我们标记套接字（利用 `fasync` 标志<sup>1</sup>），指示正在使用异步 I/O，因此将在分组到达时递交 SIGIO（至此，我们讨论的全都是如何处理 SIGIO）。由于我们不希望 `SIGIOHandler()` 在 `recvfrom()` 中阻塞，因此也会设置 `O_NONBLOCK` 标志。

#### (5) 当可用时使用空闲时间持续不断地运行：第 73~74 行。

#### (6) 执行非应答的服务器任务：第 78~81 行。

#### (7) 处理异步 I/O：第 83~110 行。

这段代码非常类似于前面的 `UDPEchoServer.c`（参见 4.2 节）中的循环。一个区别是：这里将把循环执行到没有更多要满足的挂起的应答请求时为止，然后返回；这种技术允许主程序线程继续做它所做的工作。

#### • 接收应答请求：第 90~99 行。

第一次调用 `recvfrom()` 将接收其到达会唤醒 SIGIO 信号的数据报。在执行处理程序期间可能会有额外的数据报到达，因此 `do/while` 循环会继续调用 `recvfrom()`，直到不会继续接收到更多的数据报为止。由于 `sock` 是一个非阻塞套接字，`recvfrom()` 随后会返回 -1，并把 `errno` 设置为 `EWOULDBLOCK`，从而终止循环和处理函数。

#### • 发送应答：第 101~106 行。

就像原始的 UDP 应答服务器中一样，`sendto()` 将反复把消息发送回客户。

### 6.3.3 超时

在上一小节中，我们依靠系统来通知程序所发生的与 I/O 相关的事件。不过，有时，我们实际上可能需要知道在某个时间段没有发生某个 I/O 事件。例如，我们已经提过 UDP 消息可能丢失；就这种丢失而言，我们的 UDP 应答客户（就此而言，或者任何其他使用 UDP 的客户）将永远不会接收到对其请求的响应。当然，客户不能直接辨别是否发生了丢

<sup>1</sup> 这个名称在一些平台上可能有所不同（例如，`O_ASYNC`）。

失，因此它会对其将会等待响应多长时间设置一个限制。例如，UDP 应答客户可能假定：如果服务器在两秒钟内没有响应它的请求，那么服务器将永远也不会响应。客户对这个两秒钟超时（timeout）的反应可能是放弃，或者通过重新发送请求再试一次。

实现超时的标准方法是在调用阻塞函数之前设置 alarm。

---

```
unsigned int alarm(unsigned int secs)
```

---

alarm()启动一个计时器，它在经过指定的秒数（secs）之后到期；alarm()为以前预定的任何报警返回剩余的秒数（或者如果没有预定报警，就返回 0）。当计时器到期时，就把 SIGALRM 信号发送给进程，并且执行用于 SIGALRM 的处理函数（如果有的话）。

如果应答请求或响应丢失，前面的 UDPEchoClient.c（参见 4.1 节）中所示的代码就有一个问题：客户将在 recvfrom()上无限期地阻塞，等待永远也不会到达的数据报。我们的下一个示例程序 UDPEchoClient-Timeout.c 修改了原始的 UDP 应答客户，如果没有在两秒钟的时间限制内接收到来自应答服务器的响应就重传请求消息。为了实现这一点，新客户为 SIGALRM 安装了一个处理程序，其位置正好在调用 recvfrom()之前，它为两秒钟设置了一个报警。在时间间隔的末尾，将递送 SIGALRM 信号，并调用处理程序。当处理程序返回时，阻塞的 recvfrom()将返回 -1，并把 errno 设置为 EINTR。客户然后把应答请求重新发送给服务器。应答请求的这种超时和重传动作最多会发生 5 次，之后客户就会放弃并报告失败。我们的程序描述只详细说明了与原始 UDP 应答客户在代码上的区别。

## UDPEchoClient-Timeout.c

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <signal.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9 #include <netdb.h>
10 #include "Practical.h"
11
12 static const unsigned int TIMEOUT_SECS = 2; //Seconds between retransmits
13 static const unsigned int MAXTRIES = 5;      //Tries before giving up
14
15 unsigned int tries = 0; //Count of times sent-GLOBAL for signal-handler access
16
17 void CatchAlarm(int ignored);           //Handler for SIGALRM
18

```

```
19 int main(int argc, char *argv[]) {
20
21     if ((argc < 3) || (argc > 4)) //Test for correct number of arguments
22         DieWithUserMessage("Parameter(s)",
23             "<Server Address/Name> <Echo Word> [<Server Port/Service>]\n");
24
25     char *server = argv[1];           //First arg: server address/name
26     char *echoString = argv[2];      //Second arg: word to echo
27
28     size_t echoStringLen = strlen(echoString);
29     if (echoStringLen > MAXSTRINGLENGTH)
30         DieWithUserMessage(echoString, "too long");
31
32     char *service = (argc == 4) ? argv[3] : "echo";
33
34     //Tell the system what kind(s) of address info we want
35     struct addrinfo addrCriteria;           //Criteria for address
36     memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
37     addrCriteria.ai_family = AF_UNSPEC;       //Any address family
38     addrCriteria.ai_socktype = SOCK_DGRAM;    //Only datagram sockets
39     addrCriteria.ai_protocol = IPPROTO_UDP;   //Only UDP protocol
40
41     //Get address(es)
42     struct addrinfo *servAddr; //Holder for returned list of server addrs
43     int rtnVal = getaddrinfo(server, service, &addrCriteria, &servAddr);
44     if (rtnVal != 0)
45         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
46
47     //Create a reliable, stream socket using TCP
48     int sock = socket(servAddr->ai_family, servAddr->ai_socktype,
49                     servAddr->ai_protocol);           //Socket descriptor for client
50     if (sock < 0)
51         DieWithSystemMessage("socket() failed");
52
53     //Set signal handler for alarm signal
54     struct sigaction handler;           //Signal handler
55     handler.sa_handler = CatchAlarm;
56     if (sigfillset(&handler.sa_mask) < 0) //Block everything in handler
57         DieWithSystemMessage("sigfillset() failed");
58     handler.sa_flags = 0;
59
60     if (sigaction(SIGALRM, &handler, 0) < 0)
61         DieWithSystemMessage("sigaction() failed for SIGALRM");
62
```

```

63 //Send the string to the server
64 ssize_t numBytes = sendto(sock, echoString, echoStringLen, 0,
65     servAddr->ai_addr, servAddr->ai_addrlen);
66 if (numBytes < 0)
67     DieWithSystemMessage("sendto() failed");
68 else if (numBytes != echoStringLen)
69     DieWithUserMessage("sendto() error", "sent unexpected number of bytes");
70
71 //Receive a response
72
73 struct sockaddr_storage fromAddr; //Source address of server
74 //Set length of from address structure (in-out parameter)
75 socklen_t fromAddrLen = sizeof(fromAddr);
76 alarm(TIMEOUT_SECS);           //Set the timeout
77 char buffer[MAXSTRINGLENGTH + 1]; //I/O buffer
78 while ((numBytes = recvfrom(sock, buffer, MAXSTRINGLENGTH, 0,
79     (struct sockaddr *) &fromAddr, &fromAddrLen)) < 0) {
80     if (errno == EINTR) {          //Alarm went off
81         if (tries < MAXTRIES) {    //Incremented by signal handler
82             numBytes = sendto(sock, echoString, echoStringLen, 0,
83                 (struct sockaddr *) servAddr->ai_addr, servAddr->ai_addrlen);
84             if (numBytes < 0)
85                 DieWithSystemMessage("sendto() failed");
86             else if (numBytes != echoStringLen)
87                 DieWithUserMessage("sendto() error", "sent unexpected number of bytes");
88         } else
89             DieWithUserMessage("No Response", "unable to communicate with server");
90     } else
91         DieWithSystemMessage("recvfrom() failed");
92 }
93
94 //recvfrom() got something -- cancel the timeout
95 alarm(0);
96
97 buffer[echoStringLen] = '\0';      //Null-terminate the received data
98 printf("Received: %s\n", buffer); //Print the received data
99
100 close(sock);
101 exit(0);
102 }
103
104 //Handler for SIGALRM
105 void CatchAlarm(int ignored) {
106     tries += 1;
107 }
```

(1) 程序建立和参数解析：第 1~32 行。

(2) 超时设置：第 12~17 行。

`tries` 是一个全局变量，因此可以在信号处理程序中访问它。

(3) 为 SIGALRM 建立信号处理程序：第 53~61 行。

这类似于我们在 UDPEchoServer-SIGIO.c 中为 SIGIO 所做的事情。

(4) 启动报警计时器：第 76 行。

当报警计时器到期时，将会调用处理程序 `CatchAlarm()`。

(5) 重传循环：第 78~92 行。

我们在这里必须执行循环，因为 SIGALRM 将导致 `recvfrom()` 返回 -1。当发生这种情况时，我们将决定它是否超时，如果是，就重传应答请求。

- 尝试接收：第 78~79 行。
- 查明 `recvfrom()` 失败的原因：第 80~91 行。

如果 `errno` 等于 EINTR，`recvfrom()` 就会返回，因为它在等待数据报到达时被 SIGALRM 中断，而不是因为我们得到了分组。在这种情况下，我们假定应答请求或回复丢失。如果我们没有超过重传尝试的最大次数，就把请求重传给服务器；否则，就报告失败。在重传后，将重置报警计时器，以便在超时到期时再次唤醒我们。

(6) 处理应答响应的接收：第 95~98 行。

- 取消报警计时器：第 95 行。
- 确保消息是 null 终止的：第 97 行。

`printf()` 将输出一些字节，直至它遇到 null 字节，因此我们需要确保存在 null 字节（否则，我们的程序可能会崩溃）。

- 打印接收到的消息：第 98 行。

## 6.4 多任务处理

我们的 TCP 应答服务器一次只能处理一个客户。当一个客户已经在接受服务时，如果其他的客户要求连接，它们的连接将会建立并且它们能够发送它们的请求，但是服务器只有在处理完第一个客户的请求之后，才会发送回它们的数据作为应答。这种类型的套接字应用称为迭代服务器（iterative server）。迭代服务器最适合于每个客户只需要服务器做少量、有限工作的应用。不过，如果处理客户所需的时间可能很长，任何等待的客户所经历的总连接时间可能长得令人无法接受。为了演示这个问题，在 TCPEchoClient.c（参见 3.2.1 节）中的 `connect()` 语句后面添加一个 `sleep()`，并试验多个客户同时访问 TCP 应答服务器的情况（这里，`sleep()` 模拟一个要花费大量时间的操作，比如缓慢的文件或网络 I/O）。

现代操作系统对于这种进退两难的局面提供了一种解决方案。通过使用像进程或线程

那样的构造，我们可以把针对每个客户的责任承包给服务器的一个独立执行的副本。在本节中，我们将探讨这样的并发服务器（concurrent server）的多种模型，包括每个客户一个进程、每个客户一个线程和受限的多任务处理。

### 6.4.1 每个客户一个进程

进程是在同一台主机上独立执行的程序。在每个客户一个进程的服务器中，对于每个客户连接请求，我们简单地创建一个新的进程来处理通信。进程共享服务器主机的资源，每个服务器主机并发地为其客户提供服务。

在 UNIX 中，`fork()` 尝试创建新的进程，如果失败就返回 -1。如果成功，就会创建一个新进程，除了从 `fork()` 接收到的进程 ID 和返回值之外，它与调用进程的其他方面完全相同。然后，两个进程将独立执行。调用 `fork()` 的进程称为父（parent）进程，新创建的进程则称为子（child）进程。由于进程完全相同，那么进程怎么知道它们是父进程还是子进程呢？如果从 `fork()` 返回的值是 0，进程就知道它是子进程。对于父进程，`fork()` 将返回新的子进程的进程 ID。

当子进程终止时，它不会自动消失。按 UNIX 的说法，子进程变成了僵尸（zombie）进程。僵尸进程会消耗系统资源，直到它们的父进程通过调用 `waitpid()` 对它们进行“收尸”为止，在我们的下一个示例程序 `TCPEchoServer-Fork.c` 中演示了这一点。

我们通过修改其用于 TCP 应答服务器的部分，演示了这种每个客户一个进程的多任务处理方法。程序的绝大部分与原始的 `TCPEchoServer.c`（参见 3.2.2 节）完全相同。主要区别是：多任务处理服务器每次在接受一条新连接时都会创建它自身的一个新副本；每个副本处理一个客户，然后终止。无需更改 `TCPEchoClient.c`（参见 3.2.1 节），即可将其用于这种新服务器。

我们分解了这种新的应答服务器以改进可读性，并且允许在以后的示例中进行重用。我们的程序注释仅限于新服务器与 `TCPEchoServer.c`（参见 3.2.2 节）之间的区别。

图 6.1 描绘了在服务器与客户之间建立连接时所涉及的各个阶段。服务器会持续不断地运行，在指定端口上侦听连接，并反复执行以下操作：(1) 接受来自客户的进入的连接；(2) 创建新的进程处理该连接。注意：只有原始的服务器进程可以调用 `fork()`。

### TCPEchoServer-Fork.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <sys/wait.h>
5 #include "Practical.h"
6
```

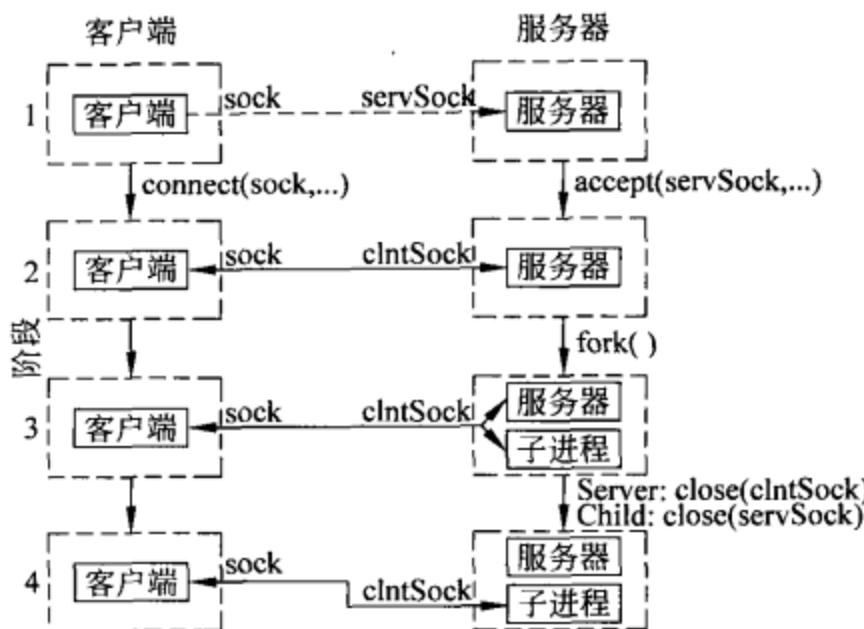


图 6.1 分叉的 TCP 应答服务器

```

7 int main(int argc, char *argv[]) {
8
9     if(argc != 2) //Test for correct number of arguments
10    DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
11
12    char *service = argv[1]; //First arg: local port/service
13    int servSock = SetupTCPServerSocket(service);
14    if (servSock < 0)
15        DieWithUserMessage("SetupTCPServerSocket() failed", "unable to establish");
16
17    unsigned int childProcCount = 0; //Number of child processes
18    for (;;) { //Run forever
19        //New connection creates a client socket
20        int clntSock = AcceptTCPConnection(servSock);
21        //Fork child process and report any errors
22        pid_t processID = fork();
23        if (processID < 0)
24            DieWithSystemMessage("fork() failed");
25        else if (processID == 0) { //If this is the child process
26            close(servSock); //Child closes parent socket
27            HandleTCPClient(clntSock);
28            exit(0); //Child process terminates
29        }
30
31        printf("with child process: %d\n", processID);
32        close(clntSock); //Parent closes child socket descriptor
33        childProcCount++; //Increment number of child processes
34

```

```

35     while (childProcCount) {      //Clean up all zombies
36         processID = waitpid((pid_t) - 1, NULL, WNOHANG); //Non-blocking wait
37         if (processID < 0)          //waitpid() error?
38             DieWithSystemMessage("waitpid() failed");
39         else if (processID == 0)    //No zombie to wait on
40             break;
41         else
42             childProcCount--;      //Cleaned up after a child
43     }
44 }
45 //NOT REACHED
46 }

```

(1) 用于 `waitpid()` 的额外的包括文件: 第 4 行。

(2) 创建服务器套接字: 第 13~15 行。

`SetupTCPServerSocket()` 分配、绑定服务器套接字，并将其标记为准备好接受进入的连接。

(3) 设置成处理多个进程: 第 17 行。

`childProcCount` 是用于统计进程数的变量。

(4) 进程分派循环: 第 18~44 行。

父进程会持续不断地循环，为每个连接请求分配一个进程。

- 获取下一个连接: 第 20 行。

`AcceptTCPConnection()` 会阻塞到建立了一条有效的连接为止，并返回该连接的套接字描述符。在图 6.1 中从第 1 阶段到第 2 阶段的过渡中描绘了连接建立。

- 创建子进程，用于处理新的连接: 第 22 行。

`fork()` 尝试复制调用进程。如果尝试失败，`fork()` 就会返回 -1。如果它成功，子进程就会接收到返回值 0，并且父进程接收到的返回值是子进程的进程 ID。当 `fork()` 创建一个新的子进程时，它会把套接字描述符从父进程复制给子进程；因此，在 `fork()` 之后，父进程和子进程都具有用于侦听套接字 (`servSock`) 以及新创建的和连接的客户套接字 (`clntSock`) 的描述符，如图 6.1 中的第 3 阶段所示。

- 子进程执行: 第 26~28 行。

子进程只负责处理新客户，因此它可以关闭侦听套接字描述符。不过，由于父进程仍然具有用于侦听套接字的描述符，这个关闭不会释放套接字。在图 6.1 中的从第 3 阶段到第 4 阶段的过渡中描绘了它。如果没有其他的进程具有指向套接字的引用，那么调用 `close()` 只会终止指定的套接字，注意到这一点很重要。子进程然后执行 `HandleTCPClient()` 处理连接。在处理客户之后，子进程将调用 `close()` 释放客户套接字。通过调用 `exit()` 终止子进程。

- 父进程继续执行：第 31~33 行。

由于子进程正在处理新客户，父进程可以关闭新的连接套接字的套接字描述符；同样，这不会释放套接字，因为子进程也包含一个引用<sup>1</sup>（参见图 6.1 中从第 3 阶段到第 4 阶段的过渡）。父进程将在 childProcCount 中保存未决的子进程的统计数。

- 处理僵尸进程：第 35~43 行。

在每个连接请求之后，父服务器进程将收回由于子进程终止而创建的僵尸进程。服务器将通过反复调用 waitpid() 来收回僵尸进程，直到它们都不存在为止。waitpid() 的第一个参数（-1）是一个通配符，指示它接受任何僵尸进程，而不管它的进程 ID 是什么。第二个参数是一个占位符，waitpid() 在其中返回僵尸进程的状态。由于我们不关心状态，因此指定 NULL，并且不返回状态。接下来是一个标志参数，用于自定义 waitpid() 的行为。如果未找到僵尸进程，WNOHANG 将导致它立即返回。waitpid() 将返回三种值类型之一：失败（返回 -1）、找到僵尸进程（返回僵尸进程的 pid），以及未找到僵尸进程（返回 0）。如果 waitpid() 找到僵尸进程，就需要递减 childProcCount，并且如果有更多未收回的子进程存在（childProcCount != 0），就寻找另一个僵尸进程。如果 waitpid() 在没有找到僵尸进程的情况下返回，父进程就会跳出收回僵尸进程的循环。

也可以用另外几种方式处理僵尸进程。在一些 UNIX 变体上，可以改变默认的子进程终止行为，使得不会创建僵尸进程（例如，把 SA\_NOCLDWAIT 标志传递给 sigaction()）。我们不会使用这种方法，因为它不是可移植的。另一种方法是为 SIGCHLD 信号建立处理函数。当子进程终止时，就把 SIGCHLD 信号递送给调用指定处理函数的原始进程。处理函数使用 waitpid() 收回任何僵尸进程。不幸的是，信号可能在任何时间中断，包括在阻塞的系统调用期间（参见 6.2 节）。用于重新启动中断的系统调用的正确方法在 UNIX 变体之间有所不同。在一些系统中，重新启动是默认的行为，在另外一些系统中，可以把 sigaction 结构的 sa\_flags 字段设置为 SA\_RESTART，以确保中断的系统调用会重新启动。在其他系统上，中断的系统调用会返回 -1，并把 errno 设置为 EINTR。在这种情况下，程序必须重新启动中断的函数。我们没有使用其中的任何一种方法，因为它们都不是可移植的，并且它们由于我们不会解决的问题而使程序变得复杂。我们把它留作一个练习，让读者修改我们的 TCP 应答服务器，以在他们的系统上使用 SIGCHLD。

对于这个示例以及本章中其余的服务器示例，我们分离出了用于建立服务器套接字的代码。可以在 TCPServerUtility.c 中找到该代码。

---

<sup>1</sup> 我们关于子进程和父进程执行的描述假定父进程在子进程之前对客户套接字执行 close()。不过，客户有可能抢先并在服务器之前对 clntSock 执行 close() 调用。在这种情况下，服务器的 close() 将执行实际的套接字释放，但是从客户的角度讲，这不会改变服务器的行为。

## SetupTCPServerSocket()

```
1 static const int MAXPENDING = 5; //Maximum outstanding connection requests
2
3 int SetupTCPServerSocket(const char *service) {
4     //Construct the server address structure
5     struct addrinfo addrCriteria;           //Criteria for address match
6     memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
7     addrCriteria.ai_family = AF_UNSPEC;      //Any address family
8     addrCriteria.ai_flags = AI_PASSIVE;       //Accept on any address/port
9     addrCriteria.ai_socktype = SOCK_STREAM;   //Only stream sockets
10    addrCriteria.ai_protocol = IPPROTO_TCP;   //Only TCP protocol
11
12    struct addrinfo *servAddr;                //List of server addresses
13    int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
14    if (rtnVal != 0)
15        DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
16
17    int servSock = -1;
18    for (struct addrinfo *addr=servAddr; addr != NULL; addr = addr->ai_next) {
19        //Create a TCP socket
20        servSock = socket(servAddr->ai_family, servAddr->ai_socktype,
21                          servAddr->ai_protocol);
22        if (servSock < 0)
23            continue;                      //Socket creation failed; try next address
24
25        //Bind to the local address and set socket to list
26        if ((bind(servSock, servAddr->ai_addr, servAddr->ai_addrlen) == 0) &&
27            (listen(servSock, MAXPENDING) == 0)) {
28            //Print local address of socket
29            struct sockaddr_storage localAddr;
30            socklen_t addrSize = sizeof(localAddr);
31            if (getsockname(servSock, (struct sockaddr *) &localAddr, &addrSize)<0)
32                DieWithSystemMessage("getsockname() failed");
33            fputs("Binding to", stdout);
34            PrintSocketAddress((struct sockaddr *) &localAddr, stdout);
35            fputc('\n', stdout);
36            break;                         //Bind and list successful
37        }
38
39        close(servSock);                //Close and try again
40        servSock = -1;
41    }
42}
```

```

43 //Free address list allocated by getaddrinfo()
44 freeaddrinfo(servAddr);
45
46 return servSock;
47 }

```

用于获取新的客户连接的代码调用 accept(), 并打印出关于客户地址的信息。

### **AcceptTCPConnection()**

```

1 int AcceptTCPConnection(int servSock) {
2     struct sockaddr_storage clntAddr; //Client address
3     //Set length of client address structure (in-out parameter)
4     socklen_t clntAddrLen = sizeof(clntAddr);
5
6     //Wait for a client to connect
7     int clntSock = accept(servSock, (struct sockaddr *) &clntAddr, &clntAddrLen);
8     if (clntSock < 0)
9         DieWithSystemMessage("accept() failed");
10
11    //clntSock is connected to a client!
12
13    fputs("Handling client ", stdout);
14    PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
15    fputc('\n', stdout);
16
17    return clntSock;
18 }

```

在连接建立后，专门通过套接字描述符（在这个示例中是 clntSock）完成指向子进程套接字的引用。在我们的分叉的 TCP 应答服务器中，客户的 IP 地址和端口只在 AcceptTCPConnection() 中是临时已知的。如果我们希望在 AcceptTCPConnection() 外面知道 IP 地址和端口，则该怎么办？如果我们不希望从 AcceptTCPConnection() 返回该信息，就可以使用 2.4.6 节中描述的 getpeername() 和 getsockname() 函数。

#### **6.4.2 每个客户一个线程**

把新进程进行分叉以处理每个客户的工作，但是这样做的代价很高。每次创建一个进程时，操作系统都必须复制父进程的全部状态，包括：内存、栈、文件/套接字描述符等。线程（thread）通过允许在同一个进程中进行多任务处理而减小了这种代价：新创建的线程只是简单地共享与父线程相同的地址空间（代码和数据），而无需复制父线程的状态。

下一个示例程序 TCPEchoServer-Thread.c 使用 POSIX 线程<sup>1</sup>（“PThreads”）演示了用于 TCP 应答服务器的每个客户一个线程的多任务处理方法。该程序的绝大部分都与 TCPEchoServer-Fork.c（参见 6.4.1 节）完全相同。同样，无需更改 TCPEchoClient.c（参见 3.2.1 节）即可适用于这个新的服务器。在程序中，只对与分叉的应答服务器不同的代码加了注释。

### TCPEchoServer-Thread.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <pthread.h>
5 #include "Practical.h"
6
7 void *ThreadMain(void *arg); //Main program of a thread
8
9 //Structure of arguments to pass to client thread
10 struct ThreadArgs {
11     int clntSock;           //Socket descriptor for client
12 };
13
14 int main(int argc, char *argv[]) {
15
16     if (argc != 2)          //Test for correct number of arguments
17         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
18
19     char *service = argv[1]; //First arg: local port/service
20     int servSock = SetupTCPServerSocket(service);
21     if (servSock < 0)
22         DieWithUserMessage("SetupTCPServerSocket() failed", "unable to establish");
23     for (;;) {              //Run forever
24         int clntSock = AcceptTCPConnection(servSock);
25
26         //Create separate memory for client argument
27         struct ThreadArgs *threadArgs = (struct ThreadArgs *) malloc(
28             sizeof(struct ThreadArgs));
29         if (threadArgs == NULL)
30             DieWithSystemMessage("malloc() failed");
31         threadArgs->clntSock = clntSock;

```

<sup>1</sup> 其他线程程序包的工作方式一般都是相同的。我们之所以选择 POSIX 线程，是因为对于大多数操作系统都存在一个 POSIX 线程的端口。

```

32
33     //Create client thread
34     pthread_t threadID;
35     int returnValue=pthread_create(&threadID, NULL, ThreadMain, threadArgs);
36     if (returnValue != 0)
37         DieWithUserMessage("pthread_create() failed", strerror(returnValue));
38     printf("with thread %lu\n", (unsigned long int) threadID);
39 }
40 //NOT REACHED
41 }
42
43 void *ThreadMain(void *threadArgs) {
44     //Guarantees that thread resources are deallocated upon return
45     pthread_detach(pthread_self());
46
47     //Extract socket file descriptor from argument
48     int clntSock = ((struct ThreadArgs *) threadArgs)->clntSock;
49     free(threadArgs);           //Deallocate memory for argument
50
51     HandleTCPClient(clntSock);
52
53     return (NULL);
54 }
```

(1) 用于 POSIX 线程的额外的包括文件：第 4 行。

(2) 设置线程：第 7~12。

ThreadMain()是使 POSIX 线程执行的函数。pthread\_create()允许调用者传递一个指针，作为要在新线程中执行的函数的参数。ThreadArgs 结构包含参数的“真实”列表。创建新线程的进程在调用 pthread\_create()之前分配和填充该结构。在这个程序中，线程函数只需一个参数 (clntSock)，因此，我们可以简单地传递一个指向整数的指针；不过，ThreadArgs 结构为线程参数传递提供了一个更一般的框架。

(3) 填充线程参数结构：第 26~31 行。

我们只把客户套接字描述符传递给新线程。

(4) 调用新线程：第 33~38 行。

(5) 线程执行：第 43~54 行。

当 pthread\_create() 创建新线程时，将会调用 ThreadMain 函数。该函数需由线程执行的原型是 void \*fcn(void \*)，这个函数接受一个 void \*类型的参数，并返回 void \*。

- 线程分离：第 45 行。

默认情况下，当线程的主函数返回时，将会维护关于函数返回代码的状态，直到父

线程收回结果为止。这非常类似于进程的行为。`pthread_detach()`允许在完成时无需父线程的干预即可立即释放线程状态。`pthread_self()`提供了当前线程的线程 ID 作为 `pthread_detach()` 的参数，这与 `getpid()` 给进程提供其进程 ID 的方式非常相似。

- 从 `ThreadArgs` 结构中提取参数：第 48~49 行。

用于这个程序的 `ThreadArgs` 结构只包含由 `accept()` 连接到客户套接字的套接字的描述符。由于 `ThreadArgs` 结构是基于每个连接分配的，一旦提取了参数，新线程就可以释放 `threadArgs`。

- `HandleTCPClient()`：第 51 行。

线程函数调用我们一直使用的相同的 `HandleTCPClient()` 函数。

- 线程返回：第 53 行。

在创建后，父线程无需与线程通信，因此线程可以返回一个 `NULL` 指针。

由于父线程与线程共享相同的地址空间（以及文件/套接字描述符），在继续处理下面的任务之前，父线程和每个连接的线程不会分别关闭连接和侦听套接字，就像分叉示例中的父进程和子进程所做的那样。图 6.2 演示了线程化的 TCP 应答服务器所采取的动作。使用线程代替进程有几个缺点：

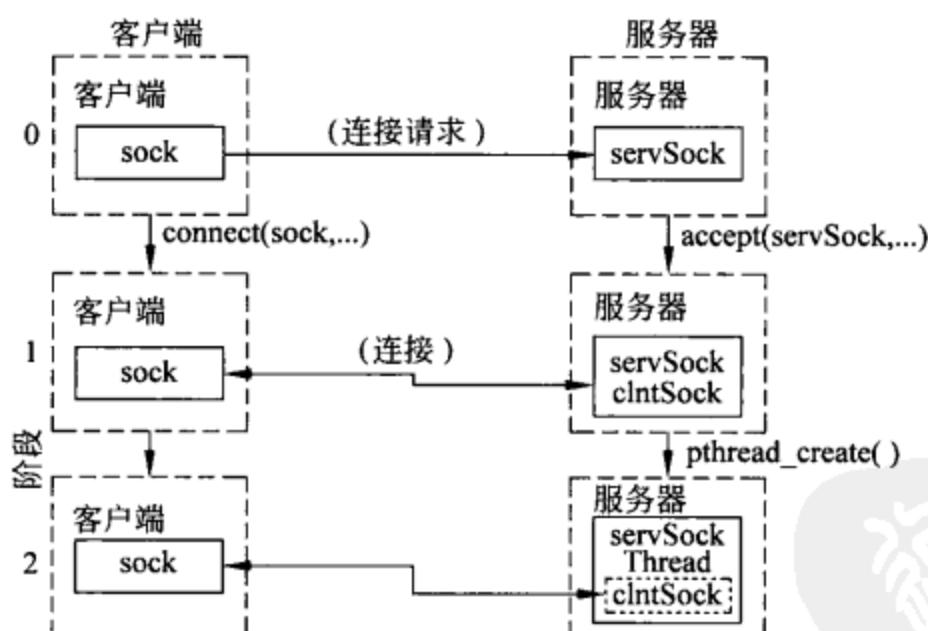


图 6.2 线程化的 TCP 应答服务器

- 如果子进程出现错误，很容易使用它的进程标识符从命令行监视并杀死它。在一些平台上，线程可能不会提供这种能力，因此必须提供额外的服务器功能，以便监视和杀死各个线程。
- 如果操作系统遗忘了线程的概念，它可能给每个进程分配相同长度的时间片。在这种情况下，处理数百个客户的线程化的 Web 服务器可能获得与单人纸牌游戏相同的 CPU 时间。

### 6.4.3 受限的多任务处理

创建进程和线程都会引发开销。此外，在许多进程和线程当中进行调度和环境切换也会导致系统做额外的工作。随着进程或线程数量的增加，操作系统需要花费越来越多的时间处理这种开销。最终，将会达到一个临界点，即增加另一个进程或线程实际上会降低总体性能。也就是说，如果把客户的连接请求进行排队，以等待前面的某个客户完成执行，而不是创建新的进程或线程为它提供服务，那么客户实际上可能会经历更短的服务时间。

我们可以通过限制服务器创建的进程数量来避免这个问题，把这种方法称为受限的多任务服务器（constrained-multitasking server）。我们展示了一个用于进程的解决方案，但是它也可以直接用于线程。在这个解决方案中，服务器首先是作为其他类型的服务器，用于创建、绑定和侦听套接字。然后，服务器创建一组固定数量（比如  $N$ ）的进程，其中每个进程都会持续不断地循环，接受来自（相同的）侦听套接字的连接。这可以工作，因为当多个进程同时在同一个侦听套接字上调用 `accept()` 时，在连接建立之前，它们都会阻塞。然后，系统选择一个进程，并且只在该进程中返回用于此新连接的套接字描述符；其他进程将保持阻塞，直到建立了下一个连接为止，然后就选择另一个幸运的进程，依此类推。

我们的下一个示例程序 `TCPEchoServer-ForkN.c` 通过修改原始的 `TCPEchoServer.c`（参见 3.2.2 节）实现了这种服务器模型，因此我们将只对它们之间的区别加注释。

#### TCPEchoServer-ForkN.c

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include "Practical.h"
5
6 void ProcessMain(int servSock); //Process main
7
8 int main(int argc, char *argv[]) {
9
10    if (argc != 3) //Test for correct number of arguments
11        DieWithUserMessage("Parameter(s)", "<Server Port/Service><Process Count>");
12
13    char *service = argv[1]; //First arg: local port
14    unsigned int processLimit = atoi(argv[2]); //Second arg: number of children
15
16    //Server socket
17    int servSock = SetupTCPSServerSocket(service);
18
19    //Fork limit-1 child processes
```

```

20   for (int processCt = 0; processCt < processLimit - 1; processCt++) {
21       //Fork child process and report any errors
22       pid_t processID = fork();
23       if (processID < 0)
24           DieWithSystemMessage("fork() failed");
25       else if (processID == 0) //If this is the child process
26           ProcessMain(servSock);
27   }
28
29   //Execute last process in parent
30   ProcessMain(servSock);
31   //NOT REACHED
32 }
33
34 void ProcessMain(int servSock) {
35     for (;;) { //Run forever
36         int clntSock = AcceptTCPConnection(servSock);
37         printf("with child process: %d\n", getpid());
38         HandleTCPClient(clntSock);
39     }
40 }
```

---

(1) “主”分叉进程的原型：第 6 行。

N 个进程中的每个进程都会执行 ProcessMain() 函数。

(2) 创建 processLimit - 1 个进程：第 19~27 行。

把循环执行 processLimit - 1 次，每次都分出一个进程，它通过把 servSock 作为参数来调用 ProcessMain()。

(3) 父进程变为最后一个进程：第 30 行。

(4) ProcessMain(): 第 34~40 行。

ProcessMain()持续不断地运行以处理客户请求。实际上，它与 TCPEchoServer.c（参见 3.2.2 节）中的 for(;;) 循环相同。

由于只会创建 N 个进程，就限制了调度的开销，并且由于每个进程都会永远存在以处理客户请求，就限制了进程创建的开销。当然，如果我们创建太少的进程，仍然可能会使客户不必要地等待服务。

## 6.5 多路复用

迄今为止，我们的程序是通过一条信道处理 I/O；我们的应答服务器的每个版本一次都只处理一条客户连接。不过，通常的情况是：应用程序需要能够同时在多条信道上执行 I/O。

例如，我们可能希望同时在多个端口上提供应答服务。一旦你考虑在服务器创建套接字并把它绑定到每个端口之后所发生的事情，这个问题就变得很明显。它准备好调用 `accept()` 接受连接，但是要选择哪个套接字呢？在一个套接字上调用 `accept()`（或 `recv()`）可能会阻塞，从而导致对另一个套接字建立的连接不必要地等待。可以使用非阻塞套接字解决这个问题，但是在这种情况下服务器最终会连续地轮询套接字，这是一种浪费。我们希望让服务器阻塞，直到某个套接字为 I/O 做好准备为止。

幸运的是，UNIX 提供了一种执行此任务的方式。利用 `select()` 函数，程序可以指定一份要为挂起的 I/O 检查的描述符列表；`select()` 会挂起程序，直到列表中的某个描述符准备好执行 I/O 并且返回哪些描述符已做好准备的指示为止。然后，程序可以在那个描述符上继续执行 I/O，并且确保操作将不会阻塞。

---

```
int select(int maxDescPlus1, fd_set *readDescs, fd_set *writeDescs,
           fd_set *exceptionDescs, struct timeval *timeout)
```

---

`select()` 监视三个单独的描述符列表（注意：这些描述符可能引用常规的文件——比如终端输入——以及套接字；在后面的示例代码中将看到一个这样的例子）。

- `readDescs`: 这个列表中的描述符用于检查即时的输入数据可用性；也就是说，调用 `recv()`（或者用于数据报套接字的 `recvfrom()`）将不会阻塞。
- `writeDescs`: 这个列表中的描述符用于检查是否能够立即写数据；也就是说，调用 `send()`（或者用于数据报套接字的 `sendto()`）将不会阻塞。
- `exceptionDescs`: 这个列表中的描述符用于检查挂起的异常或错误。对于 TCP 套接字，挂起的异常的一个例子是：TCP 套接字的远端已经关闭，而数据仍然在信道中；在这种情况下，下一个读或写操作将会失败，并且会返回 `ECONNRESET`。

为任何描述符向量传递 `NULL` 将使得 `select()` 忽略 I/O 的类型。例如，为 `exceptionDescs` 传递 `NULL` 将导致 `select()` 完全忽略任何套接字上的异常。为了节省空间，通常把所有这些描述符列表表示为一个位向量（bit vector）。为了在列表中包括一个描述符，我们把位向量中与其描述符的编号对应的位设置为 1（例如，`stdin` 是描述符 0，如果我们想监视它，那么就设置向量中的第一个位）。不过，程序不应该（并且不需要）知道这种实现策略，因为系统提供了一些宏，用于操纵 `fd_set` 类型的实例：

---

```
void FD_ZERO(fd_set *descriptorVector)
void FD_CLR(int descriptor, fd_set *descriptorVector)
void FD_SET(int descriptor, fd_set *descriptorVector)
int FD_ISSET(int descriptor, fd_set *descriptorVector)
```

---

`FD_ZERO` 用于清空描述符列表。`FD_CLR()` 和 `FD_SET()` 分别在列表中删除和添加描述

符。通过 FD\_ISSET()测试描述符在列表中的从属关系，如果给定的描述符在列表中，它就返回一个非 0 值；否则，就返回 0。

列表中可以获得的描述符的最大数量由系统定义的常量 FD\_SETSIZE 给出。虽然这个数字可能相当大，但是大多数应用程序都使用非常少的描述符。为了使实现更高效，select() 函数允许我们传递一个提示，指示在任何列表中需要考虑的最大描述符数。换句话说，maxDescPlus1 是不需要考虑的最小描述符数，它只是最大描述符值加 1。例如，如果在描述符列表中设置了描述符 0、3 和 5，就把 maxDescPlus1 设置为最大描述符值（5）加 1。注意：maxDescPlus1 适用于全部三种描述符列表。如果异常描述符列表的最大描述符是 7，而读和写描述符列表的最大描述符分别是 5 和 2，那么就把 maxDescPlus1 设置为 8。

如果希望能够为最多三种类型的 I/O 同时侦听如此之多的描述符，那么将要付出多大的代价？无需回答这个问题，因为 select()甚至可以做更多的工作！最后一个参数 (timeout) 允许控制 select() 将为某件事情发生而等待多长时间。timeout 是利用 timeval 数据结构指定的：

```
struct timeval {  
    time_t tv_sec; //Seconds  
    time_t tv_usec; //Microseconds  
};
```

如果在经过了 timeval 结构中指定的时间之后，任何指定的描述符都还没有为 I/O 做好准备，select()就会返回值 0。如果 timeout 是 NULL，select()就没有超时限制，并且会一直等待到某个描述符做好准备为止。把 tv\_sec 和 tv\_usec 都设置为 0 将导致 select() 立即返回，从而允许轮询 I/O 描述符。

如果没有错误发生，select()就会返回为 I/O 做好准备的描述符的总个数。为了指示描述符为 I/O 做好准备，select()会更改描述符列表，使得只会设置与做好准备的描述符对应的位置。例如，如果在初始读描述符列表中设置了描述符 0、3 和 5，写和异常描述符列表是 NULL，并且描述符 0 和 5 具有可供读取的数据，select()就会返回 2，并且在返回的读描述符列表中只会设置位置 0 和 5。通过返回值 -1 来指示 select() 中的错误。

让我们再次考虑在多个端口上运行应答服务的问题。如果为每个端口创建一个套接字，就可以在 readDescriptor 列表中列出这些套接字。给定这样一个列表，调用 select() 将挂起程序，直到针对其中至少一个描述符的应答请求到达为止。然后，我们可以处理连接建立，并应答那个特定的套接字。下一个示例程序 TCPEchoServer-Select.c 实现了这种模型。用户可以指定要监视的任意数量的端口。注意：连接请求被视作 I/O，并且准备一个套接字描述符，以便被 select() 读取。为了演示 select() 也可以在非套接字描述符上工作，这个服务器还会监视来自标准输入流的输入，服务器将把它解释为终止自身的信号。

## TCPEchoServer-Select.c

```
1 #include <sys/time.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6 #include <stdbool.h>
7 #include "Practical.h"
8
9 int main(int argc, char *argv[]) {
10
11     if(argc<3) //Test for correct number of arguments
12         DieWithUserMessage("Parameter(s)", "<Timeout(secs.)> <Port/Service1> ...");
13
14     long timeout = atol(argv[1]); //First arg: Timeout
15     int noPorts = argc - 2;           //Number of ports is argument count minus 2
16
17     //Allocate list of sockets for incoming connections
18     int servSock[noPorts];
19     //Initialize maxDescriptor for use by select()
20     int maxDescriptor = -1;
21
22     //Create list of ports and sockets to handle ports
23     for (int port = 0; port < noPorts; port++) {
24         //Create port socket
25         servSock[port] = SetupTCPSServerSocket(argv[port + 2]);
26
27         //Determine if new descriptor is the largest
28         if (servSock[port] > maxDescriptor)
29             maxDescriptor = servSock[port];
30     }
31
32     puts("Starting server: Hit return to shutdown");
33     bool running = true; //true if server should continue running
34     fd_set sockSet; //Set of socket descriptors for select()
35     while (running) {
36         /* Zero socket descriptor vector and set for server sockets
37         This must be reset every time select() is called */
38         FD_ZERO(&sockSet);
39         //Add keyboard to descriptor vector
40         FD_SET(STDIN_FILENO, &sockSet);
41         for (int port=0; port<noPorts; port++)
42             FD_SET(servSock[port], &sockSet);
```

```

43
44     //Timeout specification; must be reset every time select() is called
45     struct timeval selTimeout;      //Timeout for select()
46     selTimeout.tv_sec=timeout;    //Set timeout (secs.)
47     selTimeout.tv_usec=0;         //0 microseconds
48
49     //Suspend program until descriptor is ready or timeout
50     if (select(maxDescriptor+1, &sockSet, NULL, NULL, &selTimeout)==0)
51         printf("No echo requests for %ld secs...Server still alive\n", timeout);
52     else {
53         if(FD_ISSET(0, &sockSet)) { //Check keyboard
54             puts("Shutting down server");
55             getchar();
56             running = false;
57         }
58
59         //Process connection requests
60         for (int port = 0; port < noPorts; port++)
61             if (FD_ISSET(servSock[port], &sockSet)) {
62                 printf("Request on port %d: ", port);
63                 HandleTCPClient(AcceptTCPConnection(servSock[port]));
64             }
65     }
66 }
67
68 //Close sockets
69 for (int port = 0; port < noPorts; port++)
70     close(servSock[port]);
71
72 exit(0);
73 }

```

(1) 为每个端口建立套接字：第 23~30 行。

在一个数组中存储套接字描述符，每个套接字描述符对应于程序的一个参数。

(2) 为 select() 创建文件描述符的列表：第 25 行。

(3) 为 select() 设置计时器：第 44~47 行。

(4) 执行 select()：第 49~65 行。

- 处理超时：第 51 行。

- 检查键盘描述符：第 53~57 行。

如果用户按下回车键，描述符 STDIN\_FILENO 将准备好读取；在这种情况下，服务器会终止它自身。

- 检查套接字描述符：第 60~64 行。  
测试每个描述符，接受并处理有效的连接。

(5) 结尾：第 68~72 行。

关闭所有端口并退出。

`select()`是一个功能强大的函数。它也可用于实现任何阻塞的 I/O 函数（例如，`recv()`、`accept()`）的超时版本，而无需使用报警。

## 6.6 多个接收者

迄今为止，我们的所有程序处理的都是只涉及两个实体（通常是服务器和客户）的通信。这种一对一的通信有时称为单播（unicast），因为只会发送数据的一份（“uni”）副本。有时，我们希望把相同的信息发送给多个接收者。你可能知道办公室或家中的计算机通常连接到局域网，并且有时我们希望发送的信息可以被网络上的每一台主机接收到。例如，接有打印机的计算机可能通过像这样发送一条消息来通知网络上的其他主机使用它；其他机器的操作系统接收到这些通知，并且使共享资源可供其用户使用。对于这种情况，将不会把消息单播给网络上的每一台主机——这不仅需要我们知道网络上的每一台主机的地址，而且对于每一台主机都需要对消息调用一次 `sendto()`——我们希望能够只调用一次 `sendto()`，并使网络为我们处理重复性工作。

或者考虑一种典型的情况，其中发送者只有一条通往 Internet 的连接，比如电缆或 DSL 调制解调器。利用单播方法把相同的数据发送给遍布在 Internet 上的多个接收者需要通过该链路把相同的信息发送许多次（参见图 6.3a）。如果发送者的第一个跳段的连接限制了外出容量（比如说，1 Mbps 左右），在不超过第一个跳段的链路容量的情况下，它也许甚至不能把一些类型的信息——比如说，传输速率为 1 Mbps 的实时视频流——发送给多个接收者，从而导致许多丢失的分组和低劣的质量。显然，如果信息在通过第一条链路后可以进行复制，那么将会更高效，如图 6.3b 所示。这样可以节省带宽，并且简化了发送程序的工作。

你可能惊奇地获悉：经由 TCP/IP 的套接字接口允许像这样访问服务——虽然具有一些限制。有两种类型的网络复制服务：广播（broadcast）和多播（multicast）。利用广播，程序将调用 `sendto()`一次，并且会自动把消息递送到本地网络上的所有主机。利用多播，只发送一次消息，并把它递送到整个 Internet 上的一组特定的主机（这个组中可能一台主机也没有）——即那些指示网络它们应该接收发送到该组的消息的主机。

我们提过对于这些服务有一些限制。第一种限制是：只有 UDP 套接字可以使用广播和多播服务。第二种限制是：广播只能覆盖一个局部范围，通常是一个局域网。第三种限制是：大多数 Internet 服务提供商目前都不支持跨越整个 Internet 的多播。尽管有这些限制，这些服务通常可能是有用的。例如，在像校园网这样的站点内使用多播或者向本地主机广

播通常是有用的。

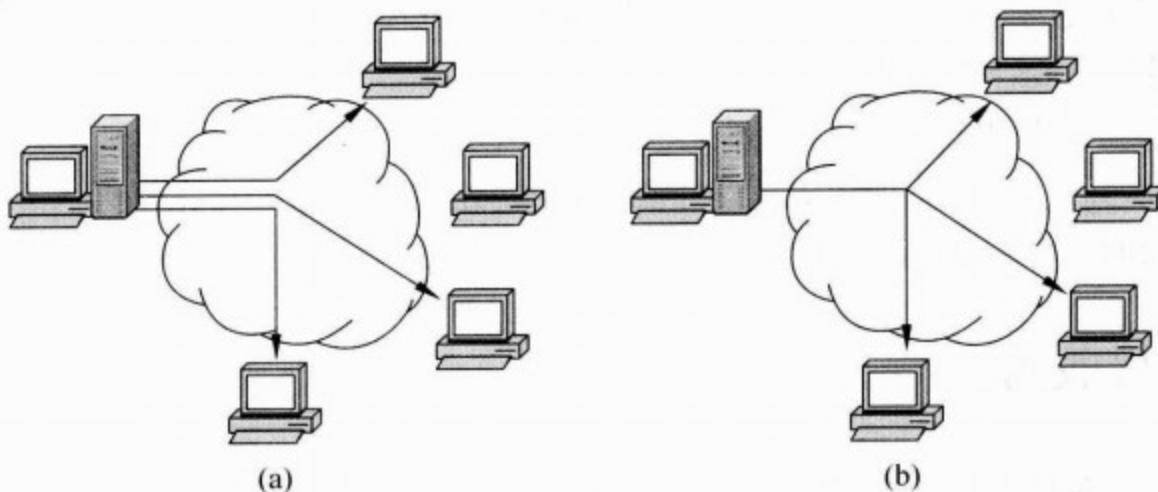


图 6.3

(a) 把相同的数据单播给多个接收者；(b) 利用多播做相同的事情

### 6.6.1 广播

可以通过把 UDP 数据报发送到特殊的地址，从而把它们发送给附带的本地网络上的所有节点。在 IPv4 中，这个特殊的地址被称为“受限的广播地址”，并且它是一种全 1 地址（在点分四组表示法中，它是 255.255.255.255）。在 IPv6 中，它被称为“全节点地址（链路范围）”，并且具有值 FF02::1。路由器不会转发寻址到上述任何一个地址的分组，因此任何一个地址都不会把数据报带到连接发送者的本地网络之外。不过，每个地址确实会把发送的分组递送到网络上的每个节点；通常，这是使用本地网络的硬件广播能力实现的（并非所有的链路都支持广播；特别是，点对点链路就不支持。如果主机的所有接口都不支持广播，那么任何使用它的尝试都将导致一个错误）。另请注意：仅当主机上的某个程序正在广播 UDP 数据报寻址到的端口上侦听数据报时，才可以在该主机上实际地“听到”数据报。

那么有没有用于向所有主机发送消息的网络级广播地址呢？答案是没有这样的地址。至于为什么没有，请考虑向 Internet 上的每台主机发送广播消息对网络产生的影响。利用这种地址发送单个数据报将导致路由器产生数量巨大的分组副本，并且会耗尽所有网络的带宽。误用（恶意的或意外的）该地址的后果会非常严重，因此 IP 协议的设计者故意没有定义这样一种 Internet 级的广播机制。即便有这些限制，在本地链路上进行广播也非常有用。在游戏玩家全都位于同一个局域网（例如，以太网）上的网络游戏中，它通常用于交换玩家之间的状态信息。

广播发送者与常规发送者之间还有另一个区别：在发送到广播地址之前，必须设置特殊的套接字选项 `SO_BROADCAST`。实际上，这要求系统“允许”广播。我们在 `BroadcastSender.c` 中演示了 UDP 广播的使用。我们的发送者每隔 3 秒钟向第一个参数指示

的地址族中受限的广播地址广播一个给定的字符串。

## BroadcastSender.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/socket.h>
6 #include <arpa/inet.h>
7 #include "Practical.h"
8
9 static const char *IN6ADDR_ALLNODES = "FF02::1"; //v6 addr not built in
10
11 int main(int argc, char *argv[]) {
12
13 if (argc != 4) //Test for correct number of arguments
14 DieWithUserMessage("Parameter(s)", "[4|6] <Port> <String to send>");
15
16 in_port_t port = htons((in_port_t) atoi(argv[2]));
17
18 struct sockaddr_storage destStorage;
19 memset(&destStorage, 0, sizeof(destStorage));
20
21 size_t addrSize = 0;
22 if (argv[1][0] == '4') {
23 struct sockaddr_in *destAddr4 = (struct sockaddr_in *) &destStorage;
24 destAddr4->sin_family = AF_INET;
25 destAddr4->sin_port = port;
26 destAddr4->sin_addr.s_addr = INADDR_BROADCAST;
27 addrSize = sizeof(struct sockaddr_in);
28 } else if (argv[1][0] == '6') {
29 struct sockaddr_in6 *destAddr6 = (struct sockaddr_in6 *) &destStorage;
30 destAddr6->sin6_family = AF_INET6;
31 destAddr6->sin6_port = port;
32 inet_pton(AF_INET6, IN6ADDR_ALLNODES, &destAddr6->sin6_addr);
33 addrSize = sizeof(struct sockaddr_in6);
34 } else {
35 DieWithUserMessage("Unknown address family", argv[1]);
36 }
37
38 struct sockaddr *destAddress = (struct sockaddr *) &destStorage;
39
40 size_t msgLen = strlen(argv[3]);
```

```

41 if (msgLen > MAXSTRINGLENGTH) //Input string fits?
42     DieWithUserMessage("String too long", argv[3]);
43
44 //Create socket for sending/receiving datagrams
45 int sock = socket(destAddress->sa_family, SOCK_DGRAM, IPPROTO_UDP);
46 if (sock < 0)
47     DieWithSystemMessage("socket() failed");
48
49 //Set socket to allow broadcast
50 int broadcastPerm = 1;
51 if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &broadcastPerm,
52     sizeof(broadcastPerm)) < 0)
53     DieWithSystemMessage("setsockopt() failed");
54
55 for (;;) {                                //Run forever
56     //Broadcast msgString in datagram to clients every 3 seconds
57     ssize_t numBytes = sendto(sock, argv[3], msgLen, 0, destAddress, addrSize);
58     if (numBytes < 0)
59         DieWithSystemMessage("sendto() failed");
60     else if (numBytes != msgLen)
61         DieWithUserMessage("sendto()", "sent unexpected number of bytes");
62
63     sleep(3);                            //Avoid flooding the network
64 }
65 //NOT REACHED
66 }

```

(1) 声明常量地址：第 9 行。

有些令人惊奇的是，没有把全 1 组地址定义为命名的系统常量，因此我们在这里给它提供一个名称。

(2) 参数处理：第 13~16 行。

(3) 目的地址存储：第 18~19 行。

我们使用 `sockaddr_storage` 结构保存目的广播地址，因为它可能是 IPv4 或 IPv6。

(4) 设置目的地址：第 21~38 行。

我们依据给定的类型设置目的地址，并记住大小以便以后使用。最后，在指向泛型 `sockaddr` 的指针中保存该地址。

(5) 套接字创建：第 44~47 行。

(6) 设置广播的许可：第 49~53 行。

默认情况下，套接字不能广播。为套接字设置 `SO_BROADCAST` 选项，以允许套接字广播。

(7) 反复广播：第 55~64 行。

每隔 3 秒钟向网络上的所有主机发送一次参数字符串。

注意：接收者程序不需要做任何特殊的事情以接收广播数据报（除了绑定到合适的端口之外）。我们把编写一个程序以接收由 BroadcastSender.c 发出的广播消息的任务留作一个练习。

## 6.6.2 多播

对于发送者来说，使用多播与使用单播非常相似。其区别在于地址的形式。多播地址标识一组接收者，它们“要求”网络把发送的消息递送到那个地址（这是接收者的职责；参见下文）。在 IPv4 和 IPv6 中为多播留出了一定范围的地址空间。IPv4 中的多播地址的范围是 224.0.0.0~239.255.255.255，IPv6 中的多播地址是指那些第一个字节中包含 0xFF（即全 1）的地址。IPv6 的多播地址空间具有相当复杂的结构，这在很大程度上超出了本书的范围（读者可以参考[4]，以了解详细信息）。对于我们的示例，我们将使用以 FF1E 开头的地址；对于在全局应用程序中的临时使用，它们是有效的（第三个十六进制数字“1”指示多播地址不是为任何特定目的而永久分配的，而第四个数字“E”则指示全局作用域）。一个例子是 FF1E::1234。

我们的示例多播发送者（如文件 MulticastSender.c 中所示）接受一个多播地址和端口作为参数，并且每隔 3 秒钟就把给定的字符串发送到那个地址和端口。

### MulticastSender.c

```

1 #include <stdlib.h>
2 #include <string.h>
3 #include <unistd.h>
4 #include <netdb.h>
5 #include "Practical.h"
6
7 int main(int argc, char *argv[]) {
8
9     if (argc < 4 || argc > 5)           //Test for number of parameters
10    DieWithUserMessage("Parameter(s)",
11                      "<Multicast Address> <Port> <Send String> [<TTL>]");
12
13    char *multicastIPString = argv[1]; //First arg: multicast IP address
14    char *service = argv[2];           //Second arg: multicast port/service
15    char *sendString = argv[3];        //Third arg: string to multicast
16
17    size_t sendStringLen = strlen(sendString);
18    if (sendStringLen > MAXSTRINGLENGTH) //Check input length

```

```

19     DieWithUserMessage("String too long", sendString);
20
21 //Fourth arg (optional): TTL for transmitting multicast packets
22 int multicastTTL = (argc == 5) ? atoi(argv[4]) : 1;
23
24 //Tell the system what kind(s) of address info we want
25 struct addrinfo addrCriteria;           //Criteria for address match
26 memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
27 addrCriteria.ai_family = AF_UNSPEC;      //v4 or v6 is OK
28 addrCriteria.ai_socktype = SOCK_DGRAM;    //Only datagram sockets
29 addrCriteria.ai_protocol = IPPROTO_UDP;   //Only UDP please
30 addrCriteria.ai_flags |= AI_NUMERICHOST;  //Don't try to resolve address
31
32 struct addrinfo *multicastAddr;          //Holder for returned address
33 int rtnVal= getaddrinfo(multicastIPString, service,
34                         &addrCriteria, &multicastAddr);
35 if (rtnVal != 0)
36     DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
37
38 //Create socket for sending datagrams
39 int sock = socket(multicastAddr->ai_family,multicastAddr->ai_socktype,
40                   multicastAddr->ai_protocol);
41 if (sock < 0)
42     DieWithSystemMessage("socket() failed");
43
44 //Set TTL of multicast packet. Unfortunately this requires
45 //address-family-specific code
46 if (multicastAddr->ai_family == AF_INET6) { //v6-specific
47     //The v6 multicast TTL socket option requires that the value be
48     //passed in as an integer
49     if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
50                   &multicastTTL, sizeof(multicastTTL)) < 0)
51         DieWithSystemMessage("setsockopt(IPV6_MULTICAST_HOPS) failed");
52 } else if (multicastAddr->ai_family == AF_INET) { //v4 specific
53     //The v4 multicast TTL socket option requires that the value be
54     //passed in an unsigned char
55     u_char mcTTL = (u_char) multicastTTL;
56     if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &mcTTL,
57                   sizeof(mcTTL)) < 0)
58         DieWithSystemMessage("setsockopt(IP_MULTICAST_TTL) failed");
59 } else {
60     DieWithUserMessage("Unable to set TTL", "invalid address family");
61 }
62

```

```

63   for (;;) { //Run forever
64     //Multicast the string to all who have joined the group
65     ssize_t numBytes = sendto(sock, sendString, sendStringLen, 0,
66                               multicastAddr->ai_addr, multicastAddr->ai_addrlen);
67     if (numBytes < 0)
68       DieWithSystemMessage("sendto() failed");
69     else if (numBytes != sendStringLen)
70       DieWithUserMessage("sendto()", "sent unexpected number of bytes");
71     sleep(3);
72   }
73 //NOT REACHED
74 }

```

注意：与广播发送者不同的是，多播发送者不需要设置多播的许可。另一方面，多播发送者可能为传输的数据报设置 TTL（“time-to-live（生存期）”）值。每个分组都包含一个计数器，在第一次发送分组时将把它初始化为某个默认值，并且每个处理分组的路由器都会递减它。当这个计数器到达 0 时，就会丢弃分组。TTL 机制（可以通过设置一个套接字选项来更改它）允许我们控制这个计数器的初始值，从而限制多播分组可能经过的跳段数。例如，通过设置 TTL=1，多播分组将不会到达本地网络之外。

如前所述，多播网络服务只会把消息复制并递送给一组特定的接收者。这组接收者称为多播组（multicast group），通过特定的多播（或组）地址确定。这些接收者需要某种机制来通知网络它们有兴趣接收发送到某一特定多播地址的消息。一旦得到通知，网络就可能开始把多播消息转发给接收者。网络的这种通知机制称为“加入一个组”（joining a group），通过底层协议实现（透明地）发送的多播请求（信令）消息来完成。为了使之发生，接收程序需要调用一个特定于地址族的多播套接字选项。对于 IPv4，它是 IP\_ADD\_MEMBERSHIP；对于 IPv6，它（令人足够惊奇地）是 IPV6\_ADD\_MEMBERSHIP。这个套接字选项接受一个结构，其中包含要加入的多播“组”的地址。对于两个版本，这个结构也是不同的：

```

struct ip_mreq {
    struct in_addr imr_multiaddr;      //Group address
    struct in_addr imr_interface;      //local interface to join on
};

```

IPv6 版本只在它所包含的地址的类型上有所区别：

```

struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;  //IPv6 multicast address of group
    unsigned int ipv6mr_interface;     //local interface to join no
};

```

我们的多播接收者包含相当多特定于版本的代码，用于处理这个加入过程：

### MulticastReceiver.c

```

1 #include <stdlib.h>
2 #include <string.h>
3 #include <unistd.h>
4 #include <netdb.h>
5 #include "Practical.h"
6
7 int main(int argc, char *argv[]) {
8
9     if (argc != 3)
10         DieWithUserMessage("Parameter(s)", "<Multicast Address> <Port>");
11
12     char *multicastAddrString = argv[1]; //First arg: multicast addr (v4 or v6!)
13     char *service = argv[2];           //Second arg: port/service
14
15     struct addrinfo addrCriteria;    //Criteria for address match
16     memset(&addrCriteria, 0, sizeof(addrCriteria)); //Zero out structure
17     addrCriteria.ai_family = AF_UNSPEC;        //v4 or v6 is OK
18     addrCriteria.ai_socktype = SOCK_DGRAM;      //Only datagram sockets
19     addrCriteria.ai_protocol = IPPROTO_UDP;    //Only UDP protocol
20     addrCriteria.ai_flags |= AI_NUMERICHOST;    //Don't try to resolve address
21
22     //Get address information
23     struct addrinfo *multicastAddr;           //List of server addresses
24     int rtnVal = getaddrinfo(multicastAddrString, service,
25                             &addrCriteria, &multicastAddr);
26     if (rtnVal != 0)
27         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
28
29     //Create socket to receive on
30     int sock = socket(multicastAddr->ai_family, multicastAddr->ai_socktype,
31                       multicastAddr->ai_protocol);
32     if (sock < 0)
33         DieWithSystemMessage("socket() failed");
34
35     if (bind(sock, multicastAddr->ai_addr, multicastAddr->ai_addrlen) < 0)
36         DieWithSystemMessage("bind() failed");
37
38     //Unfortunately we need some address-family-specific pieces
39     if (multicastAddr->ai_family == AF_INET6) {
40         //Now join the multicast "group" (address)

```

```

41     struct ipv6_mreq joinRequest;
42     memcpy(&joinRequest.ipv6mr_multiaddr, &((struct sockaddr_in6 *)
43             multicastAddr->ai_addr)->sin6_addr, sizeof(struct in6_addr));
44     joinRequest.ipv6mr_interface = 0; //Let system choose the i/f
45     puts("Joining IPv6 multicast group..."); 
46     if (setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
47         &joinRequest, sizeof(joinRequest)) < 0)
48         DieWithSystemMessage("setsockopt(IPV6_JOIN_GROUP) failed");
49 } else if (multicastAddr->ai_family == AF_INET) {
50     //Now join the multicast "group"
51     struct ip_mreq joinRequest;
52     joinRequest.imr_multiaddr =
53         ((struct sockaddr_in *) multicastAddr->ai_addr)->sin_addr;
54     joinRequest.imr_interface.s_addr = 0; //Let the system choose the i/f
55     printf("Joining IPv4 multicast group...\n");
56     if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
57         &joinRequest, sizeof(joinRequest)) < 0)
58         DieWithSystemMessage("setsockopt(IPV4_ADD_MEMBERSHIP) failed");
59 } else {
60     DieWithSystemMessage("Unknown address family");
61 }
62 //Free address structure(s) allocated by getaddrinfo()
63 freeaddrinfo(multicastAddr);
64
65 char recvString[MAXSTRINGLENGTH + 1]; //Buffer to receive into
66 //Receive a single datagram from the server
67 int recvStringLen = recvfrom(sock, recvString, MAXSTRINGLENGTH, 0, NULL,
0);
68 if (recvStringLen < 0)
69     DieWithSystemMessage("recvfrom() failed");
70
71 recvString[recvStringLen] = '\0'; //Terminate the received string
72 //Note: sender did not send the terminal 0
73 printf("Received: %s\n", recvString);
74
75 close(sock);
76 exit(0);
77 }

```

多播接收者加入组，等待接收一条消息，打印它，然后退出。

### 6.6.3 广播与多播

在应用程序中决定使用广播还是多播依赖于多种因素，包括：网络主机中对接收数据

感兴趣的主机所占的比率，以及通信各方所知道的事实。如果很大比率的网络主机希望接收到消息，广播就非常合适；不过，如果很少的主机需要接收到分组，广播就会为了少数几台主机而“强加于”网络中的所有主机。多播是首选的通信方式，因为它限制只把数据的副本发送给那些感兴趣的主机。多播的缺点是：（1）它目前不是受到全局支持的；（2）发送者和接收者必须提前协商一个 IP 多播地址。对于广播来说，则无需知道一个地址。在某些环境中（本地），广播是一个比多播更好、更易于发现的机制。所有的主机默认情况下都可以接收广播，因此，在一个网络中询问所有主机“打印机在哪儿？”是一件容易的事。另一方面，对于广域应用，多播是唯一的选择。

## 练习题

1. 请准确描述在什么条件下，更适合使用迭代服务器，而不是多任务处理服务器。
2. 你是否曾经需要在使用 TCP 的客户或服务器中实现超时机制？
3. 在 UDPEchoServer-SIGIO.c 中为什么使服务器套接字是非阻塞的？特别是，如果不这样做，可能会发生什么糟糕的事情？
4. 如何确定套接字的发送和接收缓冲区所允许的最小和最大大小？确定你的系统的最小缓冲区大小。
5. 这个练习更深入一点地考虑 SIGPIPE 机制背后的推理。回忆可知：当程序尝试在已经断开连接的 TCP 套接字上发送数据时将会递送 SIGPIPE。一种替代方法是利用 ECONNRESET 简单地使 send()失败。为什么基于信号的方法可能比通过返回值传送这种信息更好一些？
6. 当程序 BroadcastSender.c 在连接到相同 LAN 的主机上运行时，如果使用 MulticastReceiver.c 中的程序，你认为将会发生什么事情？

# 第 7 章

## 揭密

如果不理解套接字的具体实现所关联的数据结构和底层协议的工作细节，就很难领会网络编程的一些精妙之处。对于流（TCP）套接字来说更是如此。本章描述了在创建和使用套接字时“底层”所发生的一些事情。本章的初始讨论以及 7.5 节同时适用于数据报（UDP）和流（TCP）套接字；其余的讨论则只适用于 TCP 套接字。请注意，这里的描述仅仅涵盖了一些普通的事件序列，而略去了许多细节。尽管如此，我们相信即使是这种基础性的理解也是有用的。如果希望了解更详尽内容，可以参考 TCP 规范[13]，或者关于该主题的其他更全面的著作之一[2、17]。

图 7.1 是与套接字关联的一些信息（即通过调用 `socket()` 创建的对象）的简化视图。由 `socket()` 返回的整数最好被视作一个“句柄”，用于标识我们在本章中称为“套接字结构”的一个通信端点的数据结构的集合。如该图所示，可以用多个描述符指代相同的套接字结构。事实上，不同进程（*different process*）中的描述符可以指代相同的底层套接字结构。

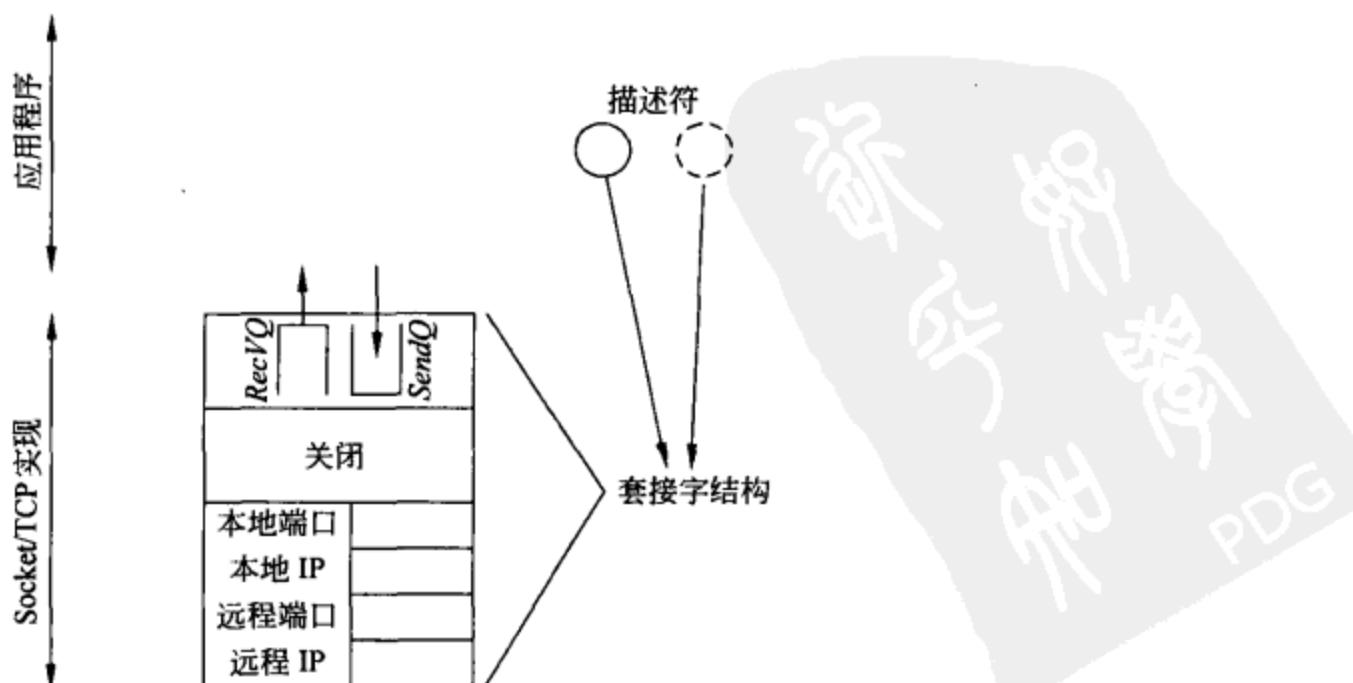


图 7.1 与套接字关联的数据结构

这里的“数据结构”意指套接字层以及包含与这个套接字抽象层相关状态信息的 TCP 实现中的所有数据结构。因此，套接字结构包含发送和接收队列以及其他信息，包括以下方面：

- 与套接字关联的本地和远程 Internet 地址以及端口号。本地 Internet 地址（图中标记为“本地 IP”）是赋予本地主机的那些地址之一；本地端口是在调用 `bind()` 时创建的。远程地址和端口用于标识本地套接字要连接到的远程套接字（如果有的话）。稍后将更详细地介绍如何以及何时确定这些值（7.5 节中包含一个简明的总结）。
- 一个等待递送的所接收数据的 FIFO 队列（“RecvQ”）和一个等待传输的数据的 FIFO 队列（“SendQ”）。
- 对于 TCP 套接字，还包含与打开和关闭 TCP 握手相关的额外的协议状态信息。在图 7.1 中，状态是“Closed”；所有的套接字状态都开始于 Closed 状态。

一些通用的操作系统为用户提供了获取底层数据结构“快照”的工具，其中之一是 `netstat`，通常可以在 UNIX（Linux）和 Windows 平台上使用它。只要给定合适的选项，`netstat` 就能准确显示图 7.1 中所示的那些信息：SendQ 和 RecvQ 中的字节数、本地和远程 IP 地址和端口号，以及连接状态等。`netstat` 的命令行选项可能有所不同，但它的输出看起来应该如下所示：

Active Internet connections (servers and established)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:36045	0.0.0.0:*	LISTEN
tcp	0	0	0.0.0.0:111	0.0.0.0:*	LISTEN
tcp	0	0	0.0.0.0:53363	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:25	0.0.0.0:*	LISTEN
tcp	0	0	128.133.190.219:34077	4.71.104.187:80	TIME_WAIT
tcp	0	0	128.133.190.219:43346	79.62.132.8:22	ESTABLISHED
tcp	0	0	128.133.190.219:875	128.133.190.43:2049	ESTABLISHED
tcp6	0	0	:::22	:::*	LISTEN

前四行和最后一行描述了正在侦听连接的服务器套接字（最后一行是一个绑定到 IPv6 地址的侦听套接字）。第五行对应于一条通往 Web 服务器（端口 80）的连接，该服务器已经单方面关闭（参见 7.4.2 节）。最后两行是现有的 TCP 连接。如果系统支持的话，你可能想使用 `netstat` 检查图 7.8~图 7.11 中描绘的场景中的连接状态。不过，要知道的是，由于这些图中描绘的状态之间的过渡非常迅速，可能很难在通过 `netstat` 提供的“快照”中捕获它们。

知道这些数据结构存在以及底层协议如何影响它们是非常有用的，因为它们控制着套接字行为的多个不同的方面。例如，由于 TCP 提供了可靠的（reliable）字节流服务，通过 TCP 套接字发送的任何数据的副本都必须由 TCP 实现（TCP implementation）保存起来，

直到其在连接的另一端被成功接收为止。一般来讲，在TCP套接字上完成`send()`调用并不意味着实际地传输了数据——而只是把它们复制到了本地缓冲区中。在正常情况下，它将被立即传输，但是精确的传输时间由TCP（而不是应用程序）控制。而且，字节流服务的性质意味着在输入流中不必保留消息边界。如在5.2.1节中所看到的，这意味着大多数应用程序协议都需要一种成帧（framing）机制，使得接收者可以断定它何时接收到了完整的消息。

另一方面，对于数据报（UDP）套接字，不会为了重传而缓冲分组，在`send()`的调用返回时，就把数据提供给网络子系统以进行传输。如果网络子系统由于某种原因无法处理消息，将不给出任何提示地丢弃分组（不过这种情况很少发生）。

下面3节讨论了利用TCP的字节流服务发送和接收数据的一些微妙之处。然后，7.4节考虑了TCP协议的连接建立和终止。最后，7.5节讨论了把传入的分组与套接字相匹配的过程以及关于绑定到端口号的一些规则。

## 7.1 缓冲和TCP

作为一名程序员，在使用TCP套接字时要记住的最重要的事情是：

**不能假设写到连接一端的数据大小与从连接另一端读取的数据大小之间存在任何一致性。**

特别是，在发送端通过调用一次`send()`传入的数据可以通过在另一端调用`recv()`多次来获取；而调用`recv()`一次可能返回调用`send()`多次所传入的数据。

为了查看这种情况，考虑如下程序：

```
rv = connect(s,...);
...
rv = send(s, buffer0, 1000, 0);
...
rv = send(s, buffer1, 2000, 0);
...
rv = send(s, buffer2, 5000, 0);
...
close(s);
```

其中，省略号表示在缓冲区中建立数据的代码，但不包含对`send()`的其他调用。这个TCP连接向接收者传输8000字节。在连接的接收端，这8000字节的分组方式取决于在连接两端调用`send()`和`recv()`之间的时间选择——以及提供给`recv()`调用的缓冲区大小。

我们可以把在TCP连接上直到特定的一刻所发送的所有字节序列（在一个方向上）视作被分成了三个FIFO队列：

(1) SendQ: 在发送者的底层实现中缓冲的字节, 它们已经写到输出流中, 但还没有成功地传输到接收主机。

(2) RecvQ: 在接收者的底层实现中缓冲的字节, 等待递送到接收程序——即从输入流中读取。

(3) Delivered: 接收者已经从输入流中读取的字节。

在接收端调用 `send()` 将向 `SendQ` 中追加字节。TCP 协议负责移动字节——从 `SendQ` 中到 `RecvQ` 中。这种转移不受用户程序控制或者不能被它直接观察到, 并且在块中发生, 这些块的大小或多或少地独立于 `send()` 传入的缓冲区的大小, 认识到这一点很重要。通过调用 `recv()` 把字节从 `RecvQ` 中移到 `Delivered` 中; 传输的块的大小依赖于 `RecvQ` 中的数据量以及提供给 `recv()` 的缓冲区的大小。

图 7.2 展示了在上面的示例中三次调用 `send()` 之后但在另一端执行任何 `recv()` 调用之前以上 3 个队列的一种可能的状态。不同的阴影图案分别指示在上面所示的三个不同的 `send()` 调用中传递的字节数。

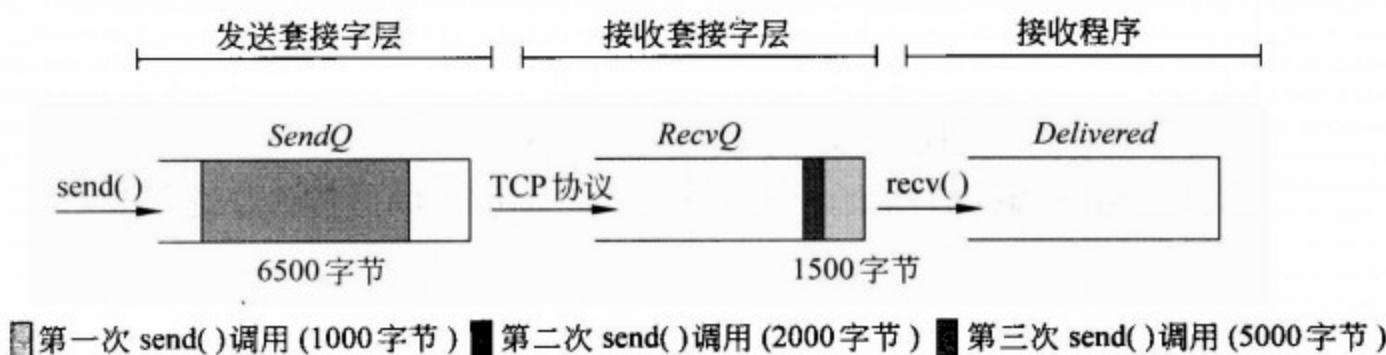


图 7.2 三次调用 `send()` 之后三个队列的状态

图 7.2 描绘了在发送主机上 `netstat` 的瞬时输出中将包含如下一行内容:

Active Internet connections					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	6500	10.21.44.33:43346	192.0.2.8:22	ESTABLISHED

在接收主机上, `netstat` 将显示如下内容:

Active Internet connections					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	1500	0	192.0.2.8:22	10.21.44.33:43346	ESTABLISHED

现在假设接收者利用大小为 2000 的字节数组调用 `recv()`。`recv()` 调用将把目前在 `RecvQ` (等待递送) 队列中的全部 1500 字节都移到该字节数组中, 并返回值 1500。注意: 这些数据包括在第一次和第二次调用 `send()` 时传递的字节。一段时间后, 当 TCP 传输完更多的数据之后, 这三部分的状态可能如图 7.3 所示。

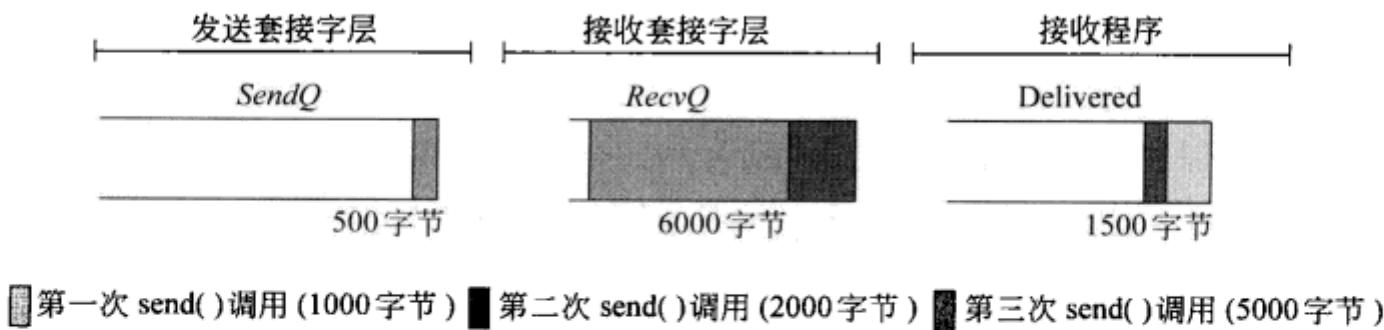


图 7.3 第一次调用 recv()之后

如果接收者现在利用大小为 4000 的缓冲区调用 `recv()`, 将把许多字节从 `RecvQ` (等待递送) 队列中转移到 `Delivered` (已递送) 队列中; 这包括第二次调用 `send()` 时剩余的 1500 字节以及第三次调用 `send()` 时的前 2500 字节。此时队列的状态如图 7.4 所示。

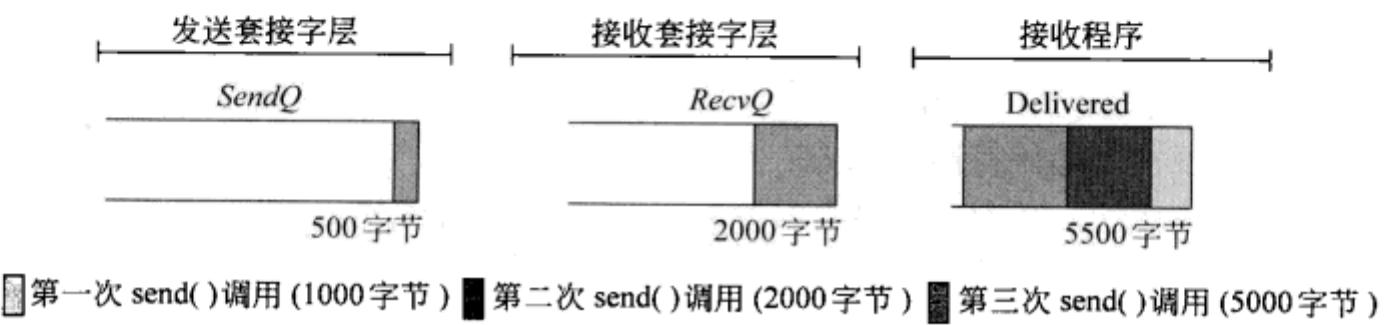


图 7.4 另外一次调用 recv()之后

下一次调用 `recv()` 时返回的字节数依赖于缓冲区的大小, 以及通过网络从发送端套接字/TCP 实现向接收端实现传输数据的时间选择。数据从 `SendQ` 到 `RecvQ` 缓冲区中的移动对应用程序协议的设计有重要的指导意义。我们已经遇到过当把带内 (in-band) 定界符用于成帧并且通过套接字接收消息时如何对它们进行解析的挑战 (参见 5.2 节)。在下面几节中, 我们将考虑另外两个更微妙的细节。

## 7.2 死锁风险

在设计应用程序协议时, 必须小心地避免死锁 (deadlock) —— 在这种状态中, 对应的每一方都被阻塞, 并等待另一方执行某种动作。例如, 如果在连接建立后客户和服务器都立即尝试接收数据, 显然将导致死锁。死锁还可能以不太即时的方式发生。

在实现中, `SendQ` 和 `RecvQ` 缓冲区都会限制其容量。尽管它们使用的实际内存大小可能会动态地增长和收缩, 还是需要有一个硬性的限制, 以防止行为异常的程序所控制的单独一条 TCP 连接将系统的内存全部耗尽。由于这些缓冲区的容量有限, 它们可能被填满, 事实也的确如此。如果与 TCP 的流量控制 (flow control) 机制结合使用, 则可能导致另一种形式的死锁。

一旦 RecvQ 被填满, TCP 的流量控制机制就会起作用, 并且阻止转移发送主机的 SendQ 中的任何字节, 直到接收者调用 `recv()` 而使 RecvQ 中具有可用的空间为止 (使用流量控制机制的目的是确保发送者传输数据的速度不会快于接收系统可以处理数据的速度)。发送程序可以持续不断地调用 `send()`, 直到 SendQ 被填满为止。不过, 一旦 SendQ 被填满, `send()` 将会阻塞, 直到有可以使用的空间为止, 也就是说直到把一些字节转移到接收套接字的 RecvQ 中为止。如果此时 RecvQ 也被填满, 所有操作都将停止, 直到接收程序调用 `recv()` 并将一些字节转移到 Delivered 中为止。

假设 SendQ 和 RecvQ 的大小分别为 SQS 和 RQS。如果调用 `send()` 传入一个大小为  $n$  (其中  $n > SQS$ ) 的缓冲区, 那么直到至少有  $n - SQS$  个字节转移到接收主机上的 RecvQ 中之后, 该调用才会返回。如果  $n$  的大小超过了  $(SQS + RQS)$ , 那么直到接收程序从输入流读取了至少  $n - (SQS + RQS)$  个字节后 `send()` 才会返回。如果接收程序没有调用 `recv()`, 涉及大量数据的 `send()` 调用可能不会成功地完成。特别是, 如果连接的两端同时调用 `send()` 并且每个调用都传递大于  $SQS + RQS$  字节的缓冲区时, 将发生死锁: 两个写操作永远都不会完成, 并且两个程序将永远保持阻塞状态。

举一个具体的例子, 考虑主机 A 上的程序和主机 B 上的程序之间的一条连接。假设 A 和 B 上的 SQS 和 RQS 都是 500 字节, 图 7.5 显示了当两个程序尝试同时发送 1500 字节时所发生的事情。主机 A 上的程序中的前 500 字节的数据已经转移到了另一端; 另外 500 字节已经复制到了主机 A 上的 SendQ 中。直到主机 B 上的 RecvQ 中有空闲空间时, 才能发送余下的 500 字节——因此直到此时 `send()` 才会返回。不幸的是, 主机 B 上的程序也存在相同的情况。因此, 两个程序的 `send()` 调用永远都不会返回!

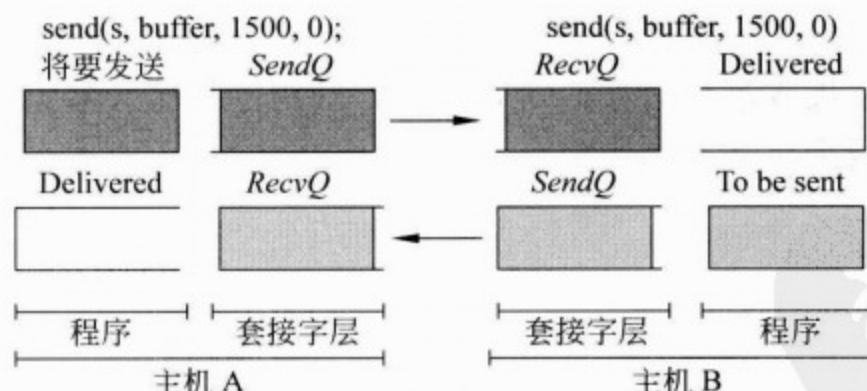


图 7.5 由于在连接的两端同时调用 `sends()` 发送到输出流而导致了死锁

这个故事的寓意是: 要仔细设计协议, 以避免在两个方向上同时发送大量的数据。

### 7.3 关于性能

TCP 实现要求在发送用户数据之前把它复制进 SendQ 中, 这还涉及到性能。特别是, SendQ 和 RecvQ 缓冲区的大小会影响可以通过 TCP 连接实现的吞吐量 (throughput)。“吞

“吞吐量”是指从发送者发送可供接收程序使用的用户数据字节时所采用的速率（rate）；在要传输大量数据的程序中，我们希望最大化每秒钟递送的字节数。在没有网络容量或其他限制的情况下，更大的缓冲区一般能够实现更高的吞吐量。

发生这种情况的原因与底层实现中将数据传入/传出缓冲区中的代价有关。如果要传输  $n$ （其中  $n$  比较大）字节的数据，利用大小为  $n$  的缓冲区调用一次 `send()` 通常比利用 1 个字节的缓冲区调用  $n$  次 `send()` 要高效得多。<sup>1</sup> 不过，如果利用比  $SQS$ （ $SendQ$  的大小）大得多的缓冲区调用 `send()`，系统必须将数据从用户地址空间中转移到  $SQS$  大小的块中。也就是说，套接字实现将填满  $SendQ$  缓冲区，等待 TCP 协议将数据转移出去，重新填充  $SendQ$ ，再次等待数据转移，如此等等。套接字实现每次都必须等待从  $SendQ$  中移出数据，这就以系统开销的形式（系统需要进行上下文切换）浪费了一些时间。这种开销与重新调用一次 `send()` 所引发的开销相当。因此，调用 `send()` 的有效缓冲区大小受到实际  $SQS$  的限制。这同样适用于接收端：无论传递给 `recv()` 的缓冲区有多大，数据都将被复制进不大于  $RQS$  的块中，并在块之间引发开销。

对于你正在编写的程序，如果吞吐量是一个重要的性能指标，你可能希望使用 `SO_RCVBUF` 和 `SO_SNDBUF` 套接字选项更改发送和接收缓冲区的大小。尽管每个缓冲区都有系统指定的最大容量，但是在现代系统上缓冲区的容量通常比默认的大小要大很多。记住：仅当程序需要发送的数据量远远大于缓冲区大小时，才需要考虑这些情况。另请注意：在 FILE 流中包装 TCP 套接字将增加另一个缓冲阶段和额外的开销，从而可能会对吞吐量产生负面影响。

## 7.4 TCP 套接字的生存期

在创建新的 TCP 套接字时，不能把它立即用于发送和接收数据。首先，需要把它连接到远程端点。因此，让我们更详细地考虑底层结构如何实现连接（或“Established”）状态。后面将看到，这些细节会影响可靠性的定义，以及把套接字绑定到前面使用过的特定端口的能力。

### 7.4.1 连接

`connect()` 调用与客户端建立连接时所关联的协议事件之间的关系如图 7.6 所示。在本节所有的示意图中，大箭头都表示导致底层套接字结构发生状态改变的外部事件。在应用程序中发生的事件（即方法调用和返回）显示在图的上部；像消息到达这样的事件则显示在

<sup>1</sup> 这一般也适用于接收，尽管利用更大的缓冲区调用 `recv()` 不保证将会返回更多的数据，一般来讲，只会返回在调用时存在的数据。

图的下部。在这些图中，时间顺序都是从左到右的。客户的 Internet 地址表示为 A.B.C.D，而服务器的 Internet 地址则表示为 W.X.Y.Z；服务器的口号是 Q（我们描述的是 IPv4 地址，不过这里介绍的所有内容同时适用于 IPv4 和 IPv6）。

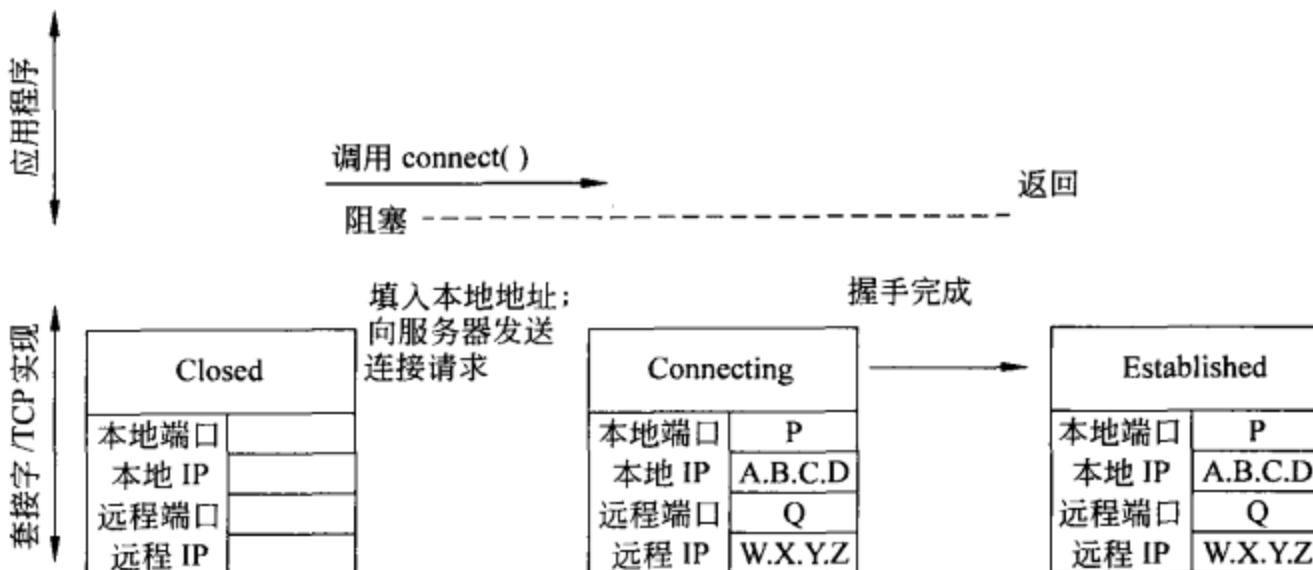


图 7.6 客户端连接建立（在这些图中事件按从左到右的顺序处理）

当客户利用服务器的 Internet 地址 (W.X.Y.Z) 和端口 (Q) 调用 `connect()` 时，底层实现将会创建一个套接字实例；它最初处于 `Closed` 状态。如果客户没有利用 `bind()` 指定本地地址/端口，实现就会选择尚未被另一个 TCP 套接字使用的本地端口号 (P)。还会分配本地 Internet 地址；如果没有显式指定它，就会使用将通过其向服务器发送分组的网络接口的地址。实现会把本地和远程地址和端口复制进底层套接字结构中，并发起 TCP 连接建立握手。

TCP 的开放式握手称为 3 次握手 (three-way handshake)，因为它通常涉及三条消息：从客户到服务器的连接请求、从服务器到客户的确认，以及从客户发回给服务器的另一个确认。一旦客户 TCP 接收到来自服务器的确认，它就会考虑要建立的连接。在正常情况下，这个过程发生得很快。不过，Internet 是一种“尽力而为”的网络，客户的初始消息或服务器的响应都可能会丢失。出于这个原因，TCP 实现将以逐渐增大的时间间隔把握手消息重传多次。如果客户 TCP 在一段时间后还没有接收到来自服务器的响应，它就会超时并放弃。在这种情况下，`connect()` 将返回 -1，并把 `errno` 设置为 `ETIMEDOUT`。实现在放弃前会非常努力地尝试完成连接，因此它可能会花费几分钟的时间来等待 `connect()` 调用失败。在发送初始握手消息之后并且在接收到来自服务器的回复消息之前（即图 7.6 的中间部分），客户主机上的 `netstat` 的输出将如下所示：

```
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address        State
tcp        0      0  A.B.C.D:P        W.X.Y.Z:Q      SYN_SENT
```

其中 SYN\_SENT 是第一条与第二条握手消息之间的客户状态的技术名称。

如果服务器没有接受连接——比如说，如果没有程序与目的地的给定端口相关联——服务器端的 TCP 将（立即）响应一条拒绝消息而不是一个确认，并且 `connect()` 返回 -1，同时将 `errno` 设置为 `ECONNREFUSED`。否则，在客户从服务器接收到肯定的回复后，`netstat` 的输出将如下所示：

```
Active Internet connections
Proto Recv-Q Send-Q      Local Address      Foreign Address      State
tcp        0      0      A.B.C.D:P      W.X.Y.Z:Q      ESTABLISHED
```

服务器端的事件序列差别很大；图 7.7、图 7.8 和图 7.9 中描述了它。服务器需要绑定到客户已知的特定 TCP 端口。通常，服务器只会在 `bind()` 调用中指定端口号（这里是 Q），并为本地 IP 地址提供特殊的通配符地址 `INADDR_ANY`。对于服务器主机具有多个 IP 地址的情况，这种技术允许套接字接收寻址到其任何 IP 地址的连接。当服务器调用 `listen()` 时，将把套接字的状态更改为“Listening”，指示它已经准备好接受新连接。图 7.7 中描绘了这些事件。在这个序列之后，服务器上的 `netstat` 的输出将包括如下一行内容：

```
Active Internet connections
Proto Recv-Q Send-Q      Local Address      Foreign Address      State
tcp        0      0      0.0.0.0:Q      0.0.0.0:0      LISTENING
```

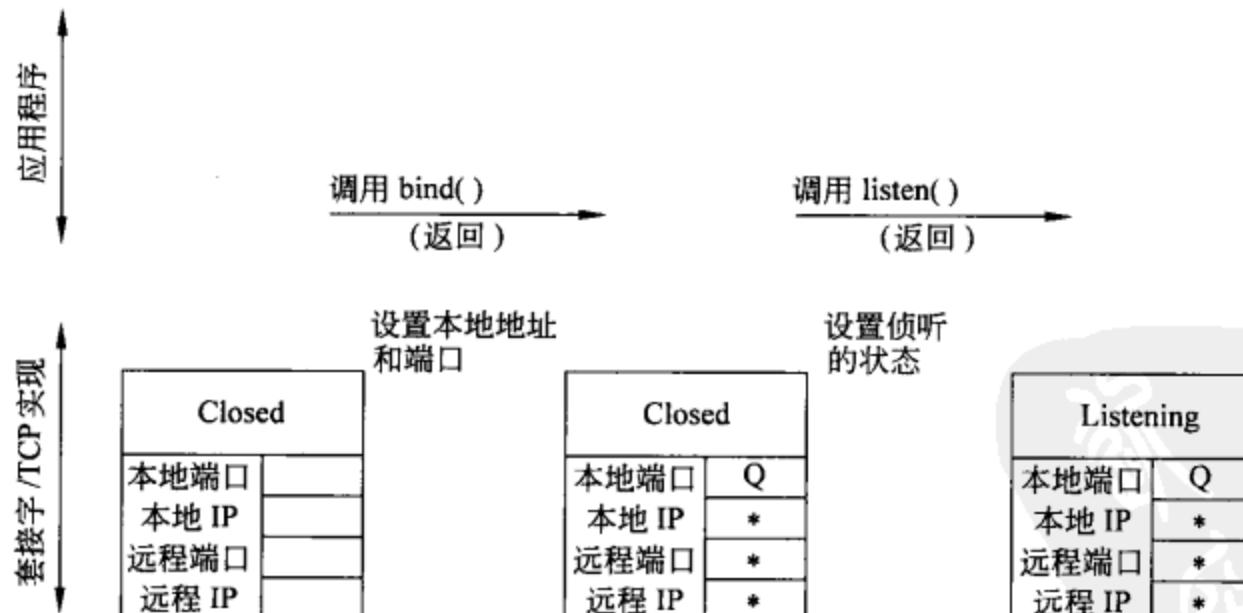


图 7.7 服务器端的套接字设置

**注意：**在调用 `listen()` 之前，到达服务器的任何客户连接请求都会被拒绝，即使它是在调用 `bind()` 之后到达的也是如此。

服务器所做的下一件事是调用 `accept()`，它会阻塞到与客户建立了一条连接为止。因此，在图 7.8 中，我们将重点关注当客户连接请求到达时在 TCP 实现中所发生的事情。注意：

该图中描绘的一切事情全都“隐蔽地”发生在 TCP 实现中。

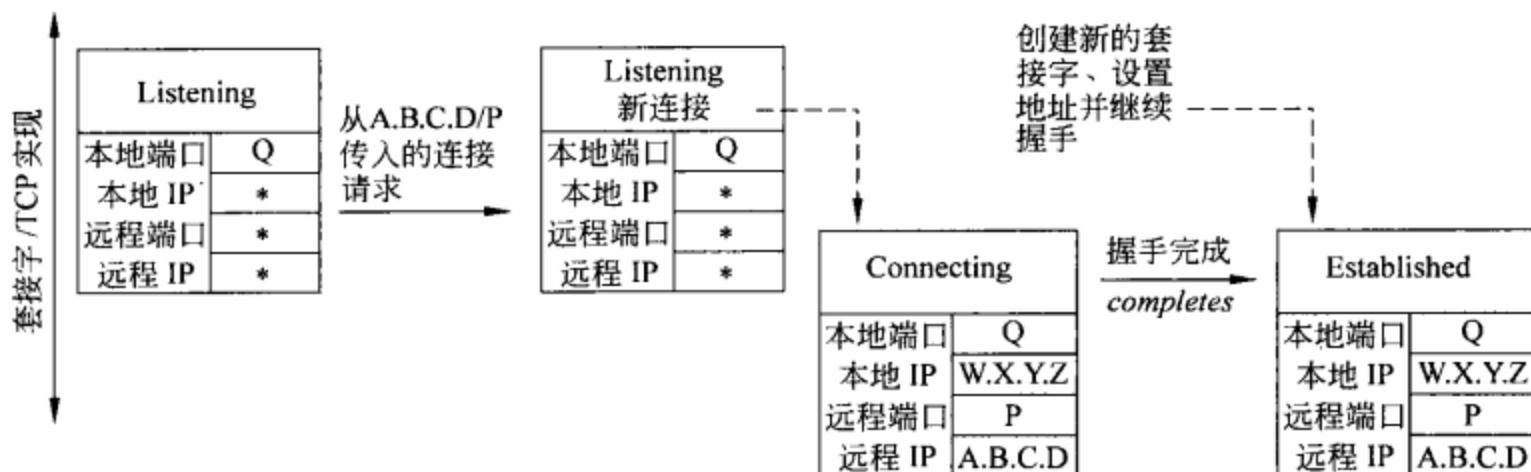


图 7.8 处理传入的连接请求

当客户的连接请求到达时，将为该连接创建一个新的套接字结构。基于到达的分组填入新套接字的地址。分组的目的 Internet 地址和端口（分别是 W.X.Y.Z 和 Q）变成了套接字的本地地址和端口；分组的源地址和端口（分别是 A.B.C.D 和 P）则变成了套接字的远程 Internet 地址和端口。注意：新套接字的本地端口号总是与侦听套接字的本地端口号相同。新套接字的状态被设置为 Connecting（在实现中实际上称为 SYN\_RCVD），并把它添加到与原始服务器套接字关联的未完全连接的套接字列表中。注意：原始服务器套接字不会改变状态。此时，netstat 的输出应该显示原始的侦听套接字和新创建的套接字：

Active Internet connections					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:Q	0.0.0.0:0	LISTENTING
tcp	0	0	W.X.Y.Z:Q	A.B.C.D:P	SYN_RCVD

除了要创建新的底层套接字结构之外，服务器端的 TCP 实现还要向客户发回一个 TCP 握手确认消息。不过，在接收到客户发来的 3 次握手中的第 3 条消息之前，服务器 TCP 并不会认为握手已经完成。当第 3 条握手消息最终到来后，就把新结构的状态设置为“ESTABLISHED”，然后将其移到与原始套接字关联的套接字结构的列表中，该列表代表准备好被接受的已建立的连接（如果第 3 条握手消息没有到达，最终会删除“Connecting”结构）。此时，netstat 的输出将包括以下内容：

Active Internet connections					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:Q	0.0.0.0:0	LISTENTING
tcp	0	0	W.X.Y.Z:Q	A.B.C.D:P	ESTABLISHED

现在，我们可以考虑（在图 7.9 中）当服务器程序调用 accept() 时所发生的事情。一旦

新连接的侦听套接字的列表中具有内容时，该调用将解除阻塞（注意：在调用 accept() 时，这个列表可能已经处于非空状态）。此时，就从列表中删除新的套接字结构，然后分配一个套接字描述符，并将其作为 accept() 的结果返回。

服务器套接字的关联列表中的每个结构都代表与另一端的客户之间的一条完全建立好的 TCP 连接，注意到这一点很重要。的确，一旦客户接收到开放式握手的第 2 条消息，它就可以发送数据——与服务器接受客户连接相比，这可能是在很久以前发生的！

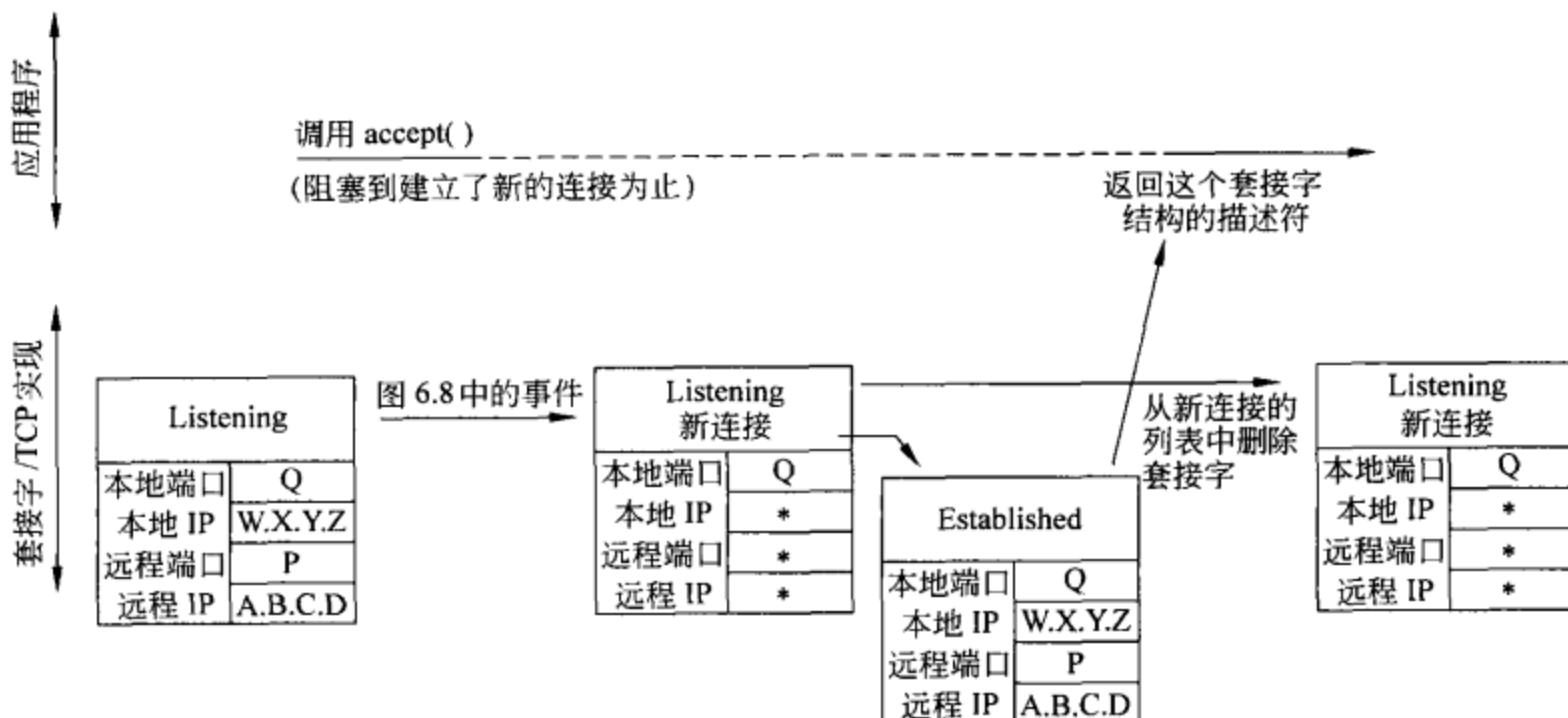


图 7.9 accept() 调用

#### 7.4.2 关闭 TCP 连接

TCP 有一个优雅的关闭（graceful close）机制，它允许应用程序在终止连接时不必担心可能仍在传输的数据会丢失。这种机制还设计成允许两个方向上的数据传输独立地终止。关闭机制的工作方式是：应用程序通过调用 close() 或 shutdown () 指示它在连接的套接字上完成了数据发送。此时，底层的 TCP 实现首先将传输保留在 SendQ 中的数据（这受到另一端的 RecvQ 中的可用空间的支配），然后向另一端发送一条关闭 TCP 的握手消息。该关闭握手消息可以看作是流终止的标志：它告诉接收 TCP 不会再有更多的字节传入 RecvQ 中了（注意：关闭握手消息本身并不会传递给接收应用程序，而是通过 recv() 返回 0 来指示其在字节流中的位置）。正在关闭的 TCP 将等待其关闭握手消息的确认，它指示在连接上发送的所有数据已经安全地传输到了 RecvQ 中。一旦收到了确认，该连接就变成“Half closed（半关闭）”状态。直到在连接的另一个方向上收到了对称的握手消息后（也就是说，连接的两端都指示它们再没有数据要发送了），连接才会完全关闭。

TCP 中的关闭事件序列可能以两种方式发生：一种方式是先由一个应用程序调用 `close()`（或 `shutdown()`），并在另一端调用 `close()` 之前完成其关闭握手消息；另一种方式是两端同时调用 `close()`，使得它们的关闭握手消息在网络中交叉传输。图 7.10 显示了当应用程序在另一端关闭之前调用 `close()` 时实现中的事件序列。发送关闭握手 (HS) 消息，将套接字结构的状态设置为“Closing”，然后 `close()` 调用返回。此后，在套接字上执行任何操作的其他尝试都将导致返回错误。当接收到关闭握手的确认之后，套接字结构的状态将变为“Half closed”，这种状态将一直持续，直到接收到另一端的关闭握手消息为止。此时，客户上的 `netstat` 的输出将把连接的状态显示为：

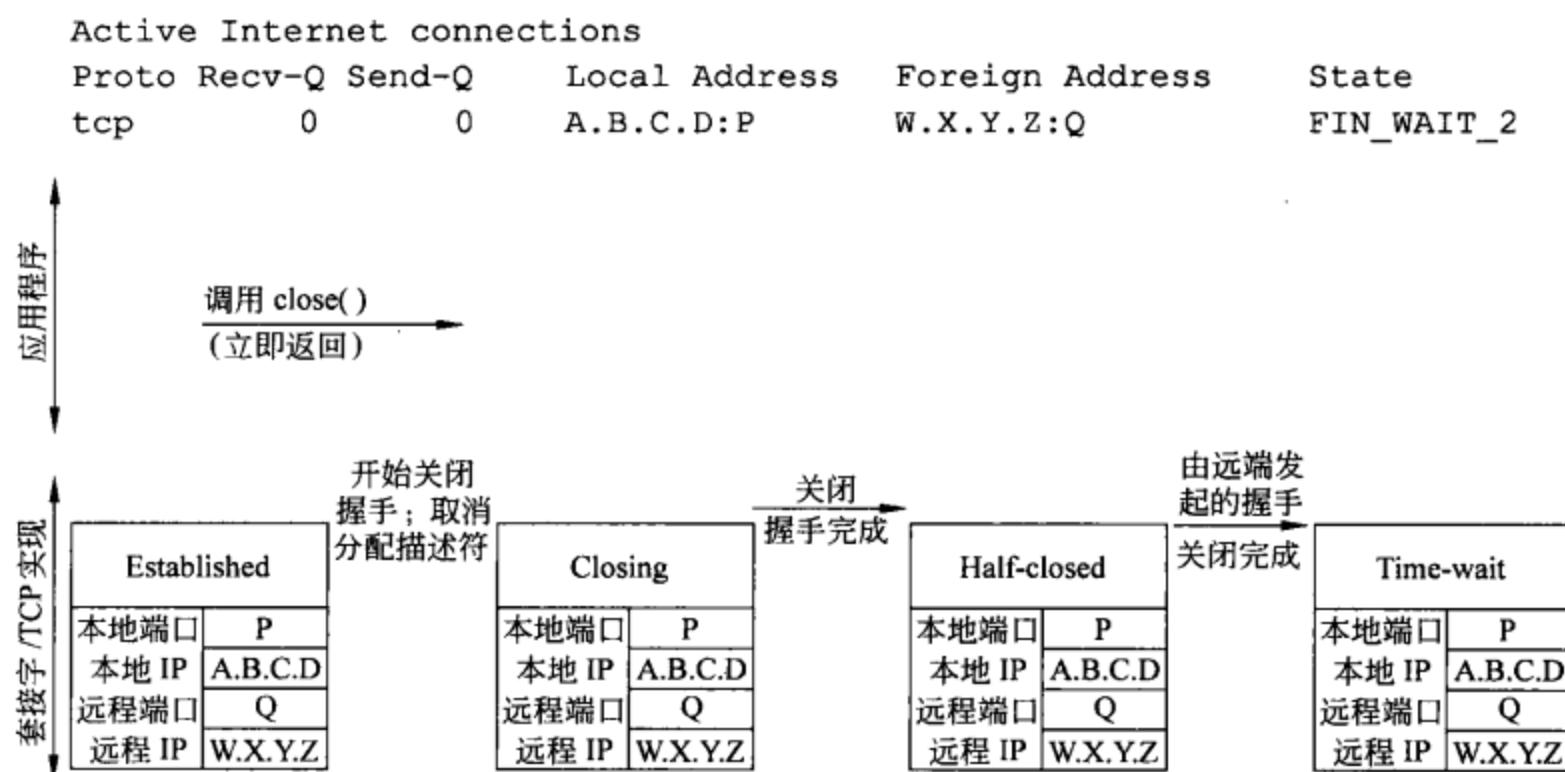


图 7.10 首先关闭 TCP 连接

(`FIN_WAIT_2` 是首先发起关闭的主机上的“Half-closed”状态的技术名称。在该图中由“Closing”指示的状态在技术上称为 `FIN_WAIT_1`，但它是一种瞬时状态，很难利用 `netstat` 捕获)。注意：如果当连接处于这种状态中时远程端点断开连接，那么本地底层结构将无限期地保持该状态。否则，当另一端的关闭握手消息到达时，就会发送一个确认，并把状态改为“Time-Wait”。尽管应用程序中的描述符可能早已被收回（甚至被重用），关联的底层套接字结构仍将在实现中继续存在一些时间；本节末尾讨论了出现这种情况的原因。

在图 7.10 的右边，`netstat` 的输出包括以下内容：

```
Active Internet connections
Proto Recv-Q Send-Q      Local Address    Foreign Address      State
tcp        0      0      A.B.C.D:P      W.X.Y.Z:Q      TIME_WAIT
```

图 7.11 显示了在没有首先发起关闭的端点上的更简单的事件序列。当关闭握手消息到达时，将会立即发送一个确认，并且连接状态将变为“Close-Wait”。该主机上的 netstat 的输出将显示以下内容：

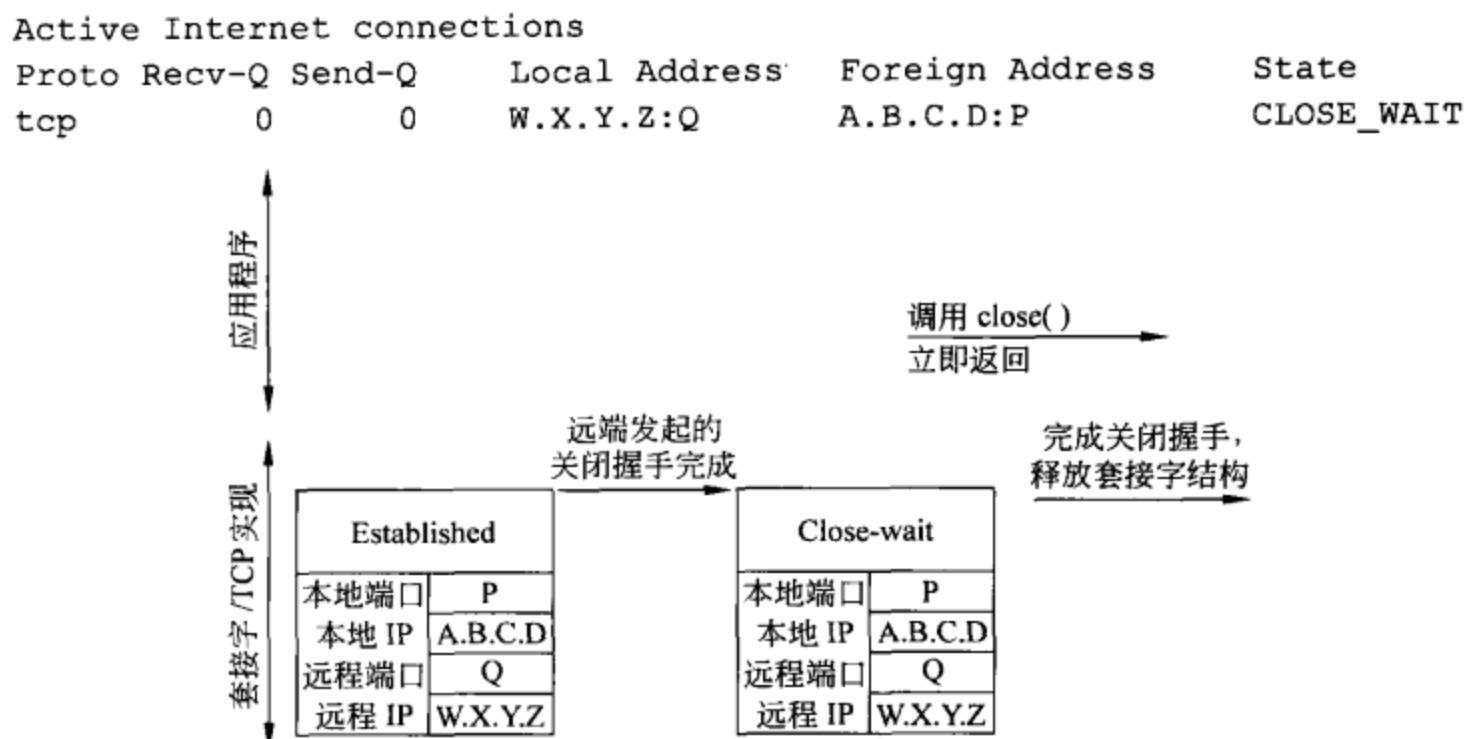


图 7.11 在另一端关闭后执行关闭

此时，一切都完成了：实现只需等待应用程序调用 `close()`。调用它之后，将取消分配套接字描述符，并发起最终的关闭握手。当它完成时，就会释放底层套接字结构。

尽管大多数应用程序使用 `close()`，但是 `shutdown()`实际上提供了更大的灵活性。调用 `close()`会终止两个方向上的传输，并导致与套接字关联的文件描述符被取消分配。保留在 `RecvQ` 中的任何未递送的数据都会被丢弃，并且流量控制机制可以阻止从另一端的 `SendQ` 中传输任何其他的数据。套接字的所有痕迹都会从调用程序中消失。不过，在底层，关联的套接字结构将继续存在，直到另一端发起它的关闭握手为止。与之相反，`shutdown()`允许独立地终止发送流和接收流。它接受一个额外的参数（`SHUT_RD`、`SHUT_WR` 或 `SHUT_RDWR` 中的一个），指示要关闭哪个流。利用第二个参数 `SHUT_WR` 调用 `shutdown()` 的程序可以继续在套接字上接收数据；而只会禁止发送。连接的另一端已关闭的事实是通过 `recv()` 返回 0（当然，一旦 `RecvQ` 为空）来指示的，以指示连接上将没有更多的数据可用。

注意这样一个事实：`close()` 和 `shutdown()` 无需等待关闭握手完成即可返回，你可能想知道发送者怎样能保证已发送的数据确实能够到达接收程序呢（即 `Delivered`）？事实上，当应用程序调用 `close()` 或 `shutdown()` 并成功关闭连接时，的确可能还有数据留存在 `SendQ` 中。如果连接的任何一端在数据传输到 `RecvQ` 中之前崩溃，数据将丢失，而发送端应用程序却

不知道这一点。

最佳的解决方案是设计一种应用程序协议，使得无论哪一端首先执行关闭，仅当它接收到关于数据已接收的应用程序级的保证后，它才能这样做。例如，当我们的 TCPEchoClient.c 程序接收到与它所发送的相同数量的字节时，那么在任何一个方向上应该都没有数据在传输，因此可以安全地关闭连接。注意：不能保证接收到的字节就是那些发送的字节；客户假定服务器实现了应答协议。在真实的应用程序中，客户当然不应该信任服务器会“做正确的事情”。

另一种解决方案是：在调用 `close()` 之前，通过设置 `SO_LINGER` 套接字选项来修改它的语义。`SO_LINGER` 指定 TCP 实现等待关闭握手完成的时间长度。使用 `linger` 结构把 `SO_LINGER` 的设置和指定的等待时间提供给 `setsockopt()`：

```
struct linger {
    int l_onoff;      //Nonzero to linger
    int l_linger;     //Time (secs.) to linger
};
```

要使用拖延行为，可以把 `l_onoff` 设置为一个非 0 值，并在 `l_linger` 中指定要拖延的时间。当设置了 `SO_LINGER` 之后，`close()` 将会阻塞，直到关闭握手完成或者经过了指定的时间为止。如果握手没有及时完成，就会返回一个错误指示（`ETIMEDOUT`）。因此，如果设置了 `SO_LINGER` 并且 `close()` 没有返回任何错误，应用程序就可以确保它发送的所有数据都到达了 `RecvQ` 中。

关闭 TCP 连接的最后一个微妙之处关注的是对 Time-Wait 状态的需要。TCP 规范要求当连接终止时，在两端的关闭握手都完成之后，至少要有一个套接字在 Time-Wait 状态下保持一段时间。之所以会提出这个要求，是由于消息在网络中传输时可能会延迟。如果连接两端一完成了关闭握手就移除它们的底层结构，并且在一对相同的套接字地址之间立即建立一条新连接，那么通过前一条连接传输的消息（它碰巧在网络中被延迟）就可能在刚刚建立好新连接之后到达。由于它将包含相同的源地址和目的地址，旧消息可能会被误认为是属于新连接的消息，并且可能（错误地）把它的数据递送给应用程序。

虽然这种情形可能很少见，TCP 还是使用了包括 Time-Wait 状态在内的多种机制来阻止它发生。Time-Wait 状态用于保证每条 TCP 连接都结束于一段平静的时间，在此期间不会有数据发送。平静时间的长度应该等于分组可以在网络中存留的最长时间的两倍。因此，当一条连接完全消失（即套接字结构离开 Time-Wait 状态并被释放），并且为在一对相同的地址之间建立新连接做好准备时，就不会再有旧实例发送的消息还存留在网络中。实际上，平静时间的长度要依赖于具体的实现，因为没有真正的机制可以限制分组在网络上能够延迟多长的时间。通常使用的时间范围从 4 分钟到 30 秒，或者甚至更短的时间。

Time-Wait 状态最重要的作用是：只要底层套接字结构还存在，就不允许其他的套接字绑定到相同的本地端口（下面将更详细地解释这一点）。

## 7.5 解多路复用揭密

在前面的讨论中已经隐含表明一个事实：即同一台机器上的不同套接字可以具有相同的本地地址和端口号。例如，在只有一个 IP 地址的机器上，通过侦听套接字利用 `accept()` 接受的每个新套接字都具有与侦听套接字相同的本地地址和端口号。显然，决定应该把传入的分组递送到那个套接字的过程（即解多路复用（demultiplexing）的过程）不仅仅涉及查看分组的目的地址和端口。否则，应该把传入的分组递送到哪个套接字就可能会存在歧义。对于 TCP 和 UDP 来说，将传入的分组匹配到某个套接字的过程是相同的，可以归纳为以下几点：

- 套接字结构中的本地端口号必须与传入的分组中的目的端口号相匹配。
- 在套接字结构中，任何包含有通配符值 (\*) 的地址字段都被认为与分组中相应字段中的任何值相匹配。
- 如果多个套接字结构与传入的分组之间对于全部 4 个地址字段都匹配，那么使用最少通配符进行匹配的地址字段将获得该分组。

例如，考虑一台具有两个 IP 地址（10.1.2.3 和 192.168.3.2）的主机及其活动 TCP 套接字结构的一个子集，如图 7.12 所示。标记为 0 的数据结构与一个侦听套接字相关联，并且具有一个通配符本地地址，端口号为 99。套接字结构 1 也用于同一个端口上的侦听套接字，但是指定了本地 IP 地址 10.1.2.3（因此它只接受对这个地址的连接请求）。套接字结构 2 用于通过套接字结构 0 的侦听套接字接受的连接，因此具有相同的本地端口号（99），但也填入了它的本地和远程 Internet 地址。其他套接字则属于其他活动的连接。现在考虑一个分组，其源 IP 地址是 172.16.1.10，源端口号是 56789，目的 IP 地址是 10.1.2.3，目的端口号是 99。它将被递送给与套接字结构 1 关联的套接字，因为该套接字结构利用最少的通配符进行匹配。

Listening		Established	
本地端口	99	本地端口	99
本地 IP	*	本地 IP	10.1.2.3
远程端口	*	远程端口	*
远程 IP	*	远程 IP	*

Listening		Established	
本地端口	99	本地端口	99
本地 IP	10.1.2.3	本地 IP	192.168.3.2
远程端口	*	远程端口	30001
远程 IP	*	远程 IP	172.16.1.9

图 7.12 利用多个匹配的套接字进行解多路复用

当程序试图调用 `bind()` 以绑定到特定的本地端口号时，将检查现有的套接字以确保没

有其他套接字已经在使用那个本地端口。如果任何套接字与 bind()的参数中指定的本地端口和本地 IP 地址（如果有的话）相匹配，那么 bind()调用将会失败并且会设置 EADDRINUSE。这在如下情形中将引发一些问题：

- (1) 服务器的侦听套接字绑定到某个特定的端口 P。
- (2) 服务器接受客户的连接，该连接进入了 Established 状态。
- (3) 服务器由于某个原因而终止——比如说，由于程序员创建了新的版本并且希望测试它。当服务器程序退出时，底层系统会自动地（并且实际地）对其所有的现有套接字调用 close()。处于 Established 状态下的套接字将立即过渡为 Time-Wait 状态。
- (4) 程序员启动了服务器的新实例，它尝试通过调用 bind()绑定到端口 P。

不幸的是，由于旧套接字处于 Time-Wait 状态下，新服务器在调用 bind()时将会失败，并且会设置 EADDRINUSE。

在编写本书时，可以用两种方式解决这个问题。一种方式是等待底层结构离开 Time-Wait 状态。另一种方式是使服务器在调用 bind()之前设置 SO\_REUSEADDR 套接字选项。不管是否有任何代表对服务器端口的以前连接的套接字存在，这都会使 bind()成功执行。这没有产生歧义的风险，因为现有的连接（仍然处于 Established 或 Time-Wait 状态）填入了远程地址，而要绑定的套接字则没有。一般来讲，SO\_REUSEADDR 选项还允许一个套接字绑定到另一个套接字已经绑定到的本地端口，只要它要绑定到的 IP 地址（通常是通配符地址 INADDR\_ANY）不同于现有套接字的那个 IP 地址即可。默认的 bind()行为是禁止这种请求。

## 练习题

1. TCP 协议被设计成允许同时进行连接尝试。也就是说，如果一个使用端口 P 和 Internet 地址 W.X.Y.Z 的应用程序试图连接到地址 A.B.C.D 和端口 Q，与此同时一个使用相同地址和端口的应用程序也尝试连接到地址 W.X.Y.Z 和端口 P，那么它们最终将成功地相互连接起来。当程序使用套接字 API 时，这种情况会发生吗？
2. 本章中的“缓冲区死锁”的第一个示例涉及连接两端的程序都试图发送很大的消息。不过，这不是形成死锁的必要条件。前面各章中的 TCPEchoClient 在连接到同一章中的 TCPEchoServer 时，怎样才会形成死锁？

# 第 8 章

## 用 C++ 进行套接字编程<sup>1</sup>

本书适用于希望理解套接字的人们，这些人不仅希望知道如何使两个程序通过网络进行通信，而且希望知道 Sockets API 如何以及为什么那样工作。当然，许多开发人员一直在没有真正理解这些细节的情况下使用套接字。人们通常通过一个库来使用套接字，这个库提供了一个简化的接口用于套接字创建、名称解析和消息传输。在像 C++ 和 Java 这样的面向对象语言中这尤其常见，在这些语言中，很容易把套接字功能包装在相关类的集合中。

开发 PracticalSocket 库的目的是帮助学生深入了解套接字编程的基础知识，而无需完全理解本书中其他地方介绍的一些材料。这个库是一个典型的包装了套接字功能的面向对象包装器；它尝试对最常用的功能提供一个简单的接口。PracticalSocket 库提供了 Windows 与 UNIX 平台之间的可移植性，并且它可以用于教学，因为其源代码很容易使用。

更熟悉 C 语言或者更喜欢首先理解底层细节的读者应该把本章留在最后阅读。它可以充当本书前面介绍的许多概念的总结和应用。对于经验丰富的 C++ 程序员或者喜欢更有选择性地学习套接字的读者，可以提前阅读本章，并且它可以充当前面各章中更详细介绍的一些概念的概述。这里展示的示例包括许多指针，它们指向本书前面适当的章节，因此本章可以充当本书中其他许多章节的入口点。

在本章中，我们介绍了 PracticalSocket 库，并在一个简单的应用程序中演示了它的使用。通过一系列更复杂的应用程序，我们展示了这个库的额外特性，并且演示了在实际中可能如何使用 PracticalSocket 或类似的库。PracticalSocket 库以及本章中用于演示它的示例程序可以从用于本书的 Web 站点上获取。

---

<sup>1</sup> 由 David Sturgill 撰稿。

## 8.1 PracticalSocket 库概述

图 8.1 演示了 PracticalSocket 中的类以及它们的继承关系。以“Socket”结尾的所有类都充当 TCP 或 UDP 套接字的包装器，并且提供了一个简单的接口，用于创建套接字并把它用于通信。SocketException 类提供了对错误处理的支持，SocketAddress 则充当地址和端口号的包装器。

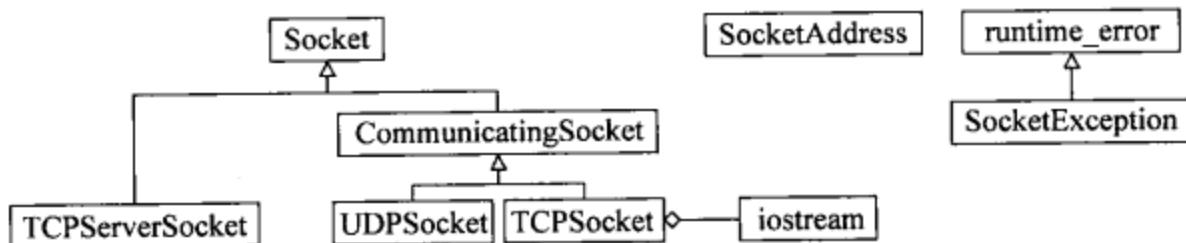


图 8.1 PracticalSocket 的类图

在开始使用这个库时，无需立即理解关于它的类和方法的一切知识。我们首先将只介绍足以编写简单应用程序的知识。此后，我们可以逐步介绍一些新特性。

TCPSocket 类是用于 TCP 通信的基础机制。它被实现为 TCP 套接字的包装器，并且充当双向通信信道中的端点。如果两个应用程序希望通信，它们都可以获取 TCPSocket 的一个实例。一旦连接了这两个套接字，就可以在一方接收从另一方发送的字节序列。

用于 TCPSocket 的功能分布在类自身以及它的两个基类 (CommunicatingSocket 和 Socket) 上。Socket 类位于继承层次结构的顶部，并且只包含所有套接字包装器的公共功能。其中，它具有记录底层套接字描述符的职责，并且当它被销毁时会自动关闭其描述符。

CommunicatingSocket 类是用于套接字的抽象，一旦连接，它就可以与对应的套接字交换数据。它提供了 send() 和 recv() 方法，它们是用于底层套接字描述符的 send() 和 recv() 调用的包装器。成功地调用 CommunicatingSocket 的 send() 方法将会把 buffer 指向的前 bufferLen 个字节发送给对应的 CommunicatingSocket。调用 CommunicatingSocket 的 recv() 方法将尝试从对应的 CommunicatingSocket 读取到 bufferLen 个字节，并把结果存储在 buffer 指向的内存中。recv() 方法将会阻塞到套接字上有数据可用为止，它将返回在套接字上接收到的字节数，并把它们写到 buffer 中。在套接字关闭后，recv() 将返回 0，指示将不能接收更多的数据。

---

```

void CommunicatingSocket::send(const void *buffer, int bufferLen)
    throw(SocketException)
int CommunicatingSocket::recv(void *buffer, int bufferLen)
    throw(SocketException)
int CommunicatingSocket::recvFully(void *buffer, int bufferLen)
    throw(SocketException)

```

---

在套接字之间传输的字节流在从发送者到接收者的传输过程中可以分成分组并通过缓冲进行重构。通过单个 `send()` 调用发送的字节组也许不能通过单个对应的 `recv()` 调用全部接收到。`recvFully()` 方法旨在对此提供帮助。它的工作方式类似于 `recv()`，只不过它会阻塞到恰好接收了 `bufferLen` 个字节或者套接字关闭。`recvFully()` 的返回值报告了接收到的字节数。一般来讲，这将与 `bufferLen` 相同。不过，如果在传输了所有请求的字节之前套接字关闭，就可能会返回一个小于 `bufferLen` 的值。

当出现错误时，`PracticalSocket` 库使用 C++ 异常来报告。这在上面的原型中显而易见。无论何时库中发生一个错误，都会抛出 `SocketException` 的一个实例。这个异常对象继承自 `runtime_error`，因此可以通过特定于应用程序的通信部分的错误处理代码将其作为 `SocketException` 的一个实例进行捕获。`SocketException` 可以由更一般的错误处理代码作为更一般的异常类型进行捕获。`SocketException` 的 `what()` 方法返回一个字符串，它简要描述了发生的特定错误。

获得 `TCPsocket` 的实例的方式依赖于应用程序的角色。要建立一对连接的 `TCPsocket` 对等实体，一个应用程序必须充当服务器，另一个则充当客户。服务器通过创建 `TCPserverSocket` 的一个实例来侦听新连接；另一方则直接创建 `TCPsocket`。`TCPserverSocket` 类从 `Socket`（而不是 `CommunicatingSocket`）派生而来。它用于同客户应用程序建立新的 TCP 套接字连接，但它自身并不用于发送和接收字节。服务器必须利用应用程序定义的端口号构造一个 `TCPserverSocket`。然后，`accept()` 调用将会阻塞，直到客户应用程序通过利用相同的端口号创建 `TCPsocket` 来尝试连接为止。当这发生时，`accept()` 将返回一个指向 `TCPsocket` 的一个新实例的指针，该实例被连接到客户上对应的 `TCPsocket`。

---

```
TCPServerSocket(in_port_t localPort) throw(SocketException)
TCPsocket *TCPServerSocket::accept() throw(SocketException)
```

---

客户应用程序通过简单地构造 `TCPsocket` 的一个示例并提供服务器的主机名称或地址以及相同的端口号，创建套接字连接中它自己的那一端。一旦创建了一对连接的 `TCPsocket` 对象，客户和服务器就可以使用 `send()` 和 `recv()` 方法通信，直到其中一个端点通过析构函数或者 `close()` 方法关闭了它的连接为止。

---

```
TCPsocket(const char *foreignAddress, in_port_t foreignPort)
    throw(SocketException)
void Socket::close()
```

---

## 8.2 加 1 服务

迄今为止介绍的少数几个类和方法足以让我们实现与前面几章中相似的简单客户和服务器。这里，通过 PracticalSocket 中的类访问套接字会在一定程度上产生更短的源代码，它隐藏了底层 API 的许多细节。“加 1”服务是一个执行递增操作的客户-服务器应用程序。客户把无符号的整数发送给服务器，服务器则发回一个比该整数大 1 的值。

### 8.2.1 加 1 服务器

`PlusOneServer.cpp` 是应用程序的服务器部分。它接受客户连接，从每个客户读取一个 32 位的无符号整数，递增它，然后发回它。

#### **PlusOneServer.cpp**

---

```

1 #include <iostream>
2 #include "PracticalSocket.h"
3
4 using namespace std;
5
6 int main(int argc, char *argv[]) {
7     try {
8         //Make a socket to listen for SurveyClient connections.
9         TCPServerSocket servSock(9431);
10
11     for (;;) {                                //Repeatedly accept connections
12         TCPSocket *sock = servSock.accept();    //Get next client connection
13
14         uint32_t val;                          //Read 32-bit int from client
15         if (sock->recvFully(&val, sizeof(val)) == sizeof(val)) {
16             val = ntohl(val);                  //Convert to local byte order
17             val++;                           //Increment the value
18             val = htonl(val);                //Convert to network byte order
19             sock->send(&val, sizeof(val));   //Send value back to client
20         }
21
22         delete sock;                      //Close and delete TCPSocket
23     }
24 } catch (SocketException &e) {
25     cerr << e.what() << endl;               //Report errors to the console
26 }
27

```

```
28     return 0;
29 }
```

(1) 应用程序建立：第 1~6 行。

对 Sockets API 的访问隐藏在 PracticalSocket 中的类后面。应用程序只需包括用于该库的头文件以及我们直接使用的任何调用即可。

(2) 错误处理：第 7 行、第 24~26 行。

错误处理是通过异常进行的。如果发生一个错误，就会在主函数末尾捕获它，并且程序会在终止前打印一条错误消息。

(3) 创建服务器套接字：第 9 行。

服务器创建一个 TCPServerSocket，它在端口 9431 上侦听连接。当像这样构造它时，TCPServerSocket 会自动调用 2.5 节、2.6 节和 2.7 节中所描述的 socket()、bind() 和 listen()。如果这些步骤中发生任何错误，就会抛出一个异常。

为了保持示例简单，在构造函数调用中硬编码了服务器的端口号。当然，在头文件中使用命名的常量作为端口号更容易维护。

(4) 反复接受客户连接：第 11~12 行。

服务器反复调用 accept() 方法，等待新的客户连接。当客户连接时，该方法返回一个指针，它指向用于同客户进行通信的新 TCPSocket。这个方法是 2.7 节中描述的 accept() 调用的包装器。如果 accept() 中出现错误，catch 块将报告错误，并且程序终止。

(5) 从客户读取整数，并转换字节顺序：第 14~16 行。

客户和服务器被编写成以 32 位无符号整数的形式交换固定大小的消息。服务器使用栈分配的整数 val 来保存接收到的消息。由于服务器期望一条 4 字节的消息，它将使用 TCPSocket 的 recvFully() 方法把消息直接读到用于 val 的存储器中。如果在完整的消息到达之前客户连接关闭，recvFully() 就会返回少于所期望的字节数，并且服务器会忽略消息。

尽管客户和服务器就消息的大小达成了一致意见，如果它们运行在不同的主机上，它们也可能使用不同的字节顺序表示整数。为了考虑到这种可能性，我们同意只传输具有大端字节顺序的值。如果必要，服务器将使用 ntohl() 把接收到的整数转换为本地字节顺序。

读取完整的消息和调整字节顺序实际上只是成帧和编码的问题。第 5 章特别关注了这些主题，并且提供了多种技术用于处理成帧和编码。

(6) 递增客户发送的整数并发回它：第 17~19 行。

服务器把客户提供的值加 1，然后把它转换回网络字节顺序。服务器通过发送用于表示 val 的 4 字节内存的副本来自发回这个值。

(7) 关闭客户连接：第 22 行。

当服务器销毁它的 TCPSocket 的实例时，就会关闭套接字连接。如果服务器忽略了销

毁这个对象，它不仅会泄漏内存，而且会泄漏带有每条客户连接的底层套接字描述符。具有这种错误的服务器可能很快就会达到操作系统对每个进程的资源施加的限制。

### 8.2.2 加 1 客户

PlusOneClient.cpp 是加 1 服务器的对应客户。它连接到服务器，给它发送在命令行上提供的整数值的副本，并打印出服务器发送回的值。

#### PlusOneClient.cpp

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "PracticalSocket.h"
4
5 using namespace std;
6
7 int main(int argc, char *argv[]) {
8     if (argc != 3) { //Check number of parameters
9         cerr << "Usage: PlusOneClient <server host> <starting value>" << endl;
10    return 1;
11 }
12
13 try {
14     TCPSocket sock(argv[1], 9431); //Connect to the server.
15
16     uint32_t val = atoi(argv[2]); //Parse user-supplied value
17     val = htonl(val); //Convert to network byte order
18     sock.send(&val, sizeof(val)); //Send to server.
19
20     //Read the server's response, convert to local byte order and print it
21     if (sock.recvFully(&val, sizeof(val)) == sizeof(val)) {
22         val = ntohl(val);
23         cout << "Server Response: " << val << endl;
24     }
25     //Socket is closed when it goes out of scope
26 } catch(SocketException &e) {
27     cerr << e.what() << endl;
28 }
29
30 return 0;
31 }
```

(1) 应用程序建立和参数检查：第 1~11 行。

客户使用 `PracticalSocket` 的类以及来自其他几个头文件的支持。在命令行上，客户期望输入服务器的地址或主机名，以及要发送给服务器的整数值。如果提供了错误数量的参数，则会打印一条有用的消息。

(2) 错误处理：第 13 行、第 26~28 行。

与服务器中一样，由程序末尾的代码捕获与套接字相关的异常，并且会打印一条错误消息。

(3) 连接到服务器：第 14 行。

客户创建 `TCPsocket` 的一个实例，为服务器传入用户提供的主机名和硬编码的端口号。这个构造函数隐藏了底层 Sockets API 中的许多细节。首先，把给定的主机名解析为一个或多个地址。如第 3 章中所述，`getaddrinfo()` 为给定的主机和端口返回所有匹配的地址。其中一些可能是 IPv4 地址，另外一些可能是 IPv6 地址。`TCPsocket` 构造函数为第一个地址创建一个套接字，并尝试连接到它。如果失败，就会试验每个连续的地址，直到可以建立连接为止。如果不能连接到任何地址，就会抛出一个异常。

(4) 解析、转换值，并把它发送给服务器：第 16~18 行。

客户解析用户从命令行提供的值，把它转换为网络字节顺序，并通过在值的起始地址中发送 4 个字节把它发送给服务器。

(5) 接收递增的值，转换并打印它：第 21~24 行。

使用 `recvFully()`，客户将会阻塞，直至它从服务器接收到一个 4 字节的整数或者套接字连接关闭为止。如果全部 4 个字节都到达了，就把接收到的值转换为主机字节顺序并打印它。

(6) 关闭套接字：第 25 行。

由于客户的 `TCPsocket` 是在栈上分配的，当套接字超出作用域时，它会自动关闭并销毁。

### 8.2.3 运行服务器和客户

可执行的 `PlusOneServer` 无需命令行参数。一旦编译，应该可以从命令提示符启动它，只要你想试验它，就允许长时间运行它。在服务器运行时，应该从不同的命令提示符（可能在不同的主机上）运行 `PlusOneClient` 的一个副本。在命令行上提供服务器主机的名称和一个整数值，利用来自服务器的一点帮助，它应该会产生一个比命令行参数大 1 的值。例如，如果服务器正在名为“venus”的主机上运行，就应该能够像下面这样运行客户：

`PlusOneClient` 连接到主机 `venus` 上的服务器：

```
% PlusOneClient venus 345318
Server Response: 345319
```

## 练习题

- 不要递增来自单个客户的值，修改服务器，使得它发回不同客户提供的两个值的和。在第一个客户连接后，服务器将等待第二个客户。服务器将读取来自两个客户的整数值，并把这两个值的和发回两个客户。
- 客户和服务器使用二进制编码的、固定大小的消息通信。修改这些程序，如 5.2.2 节中所描述的那样，使得它们使用长度可变的、文本编码的消息。发送者将把它的整数值作为 ASCII 编码的数字序列写到一个字符数组中。接收者将读取这个数组的一个副本，并解析出整数。

## 8.3 调查服务

在加 1 服务中演示的概念的基础上，使用第 5 章和第 6 章中介绍的一些技术，我们可以创建一个分布式应用程序，它可以做更有用一点的工作。调查客户和服务器实现了一个简单的分布式调查应用程序。在开始时，服务器从一个文本文件中读取调查问题和响应的列表。当客户连接后，服务器给它发送问题和响应选项的副本。客户打印每个问题，并把用户的响应发送回服务器，并在服务器中记录它们。

文件 `survey1.txt` 是服务器可能使用的示例调查文件。第一行给出问题的数量，其后接着每个问题的描述。在描述每个问题时，首先给出一行提示，其后接着一行给出响应的数量，然后依次列出每个响应的文本行。例如，关于下面的调查的第一个问题是：“What is your favorite flavor of ice cream?”。三个可能的响应是“Vanilla”、“Chocolate”或“Strawberry”。当用户接受调查时，服务器将记录选择了其中多少个响应。

### **survey1.txt**

---

```
1 2
2 What is your favorite flavor of ice cream?
3 3
4 Vanilla
5 Chocolate
6 Strawberry
7 Socket programming is:
8 4
9 Surprisingly easy
10 Empowering
11 Not for the faint of heart
```

12 Fun for the whole family

### 8.3.1 调查的支持函数

客户和服务器依赖于 SurveyCommon.h 和 SurveyCommon.cpp 中实现的公共函数。头文件提供了一个命名的常量用于服务器的端口号、一个 Question 类型用于表示调查问题，以及用于所提供函数的原型。

#### **SurveyCommon.h**

```
1 #ifndef __SURVEYCOMMON_H__
2 #define __SURVEYCOMMON_H__
3
4 #include "PracticalSocket.h"
5 #include <string>
6 #include <vector>
7
8 /** Port number used by the Survey Server */
9 const in_port_t SURVEY_PORT = 12543;
10
11 /** Write an encoding of val to the socket, sock. */
12 void sendInt(CommunicatingSocket *sock, uint32_t val) throw(SocketException);
13
14 /** Write an encoding of str to the socket, sock. */
15 void sendString(CommunicatingSocket *sock, const std::string &str)
16 throw(SocketException);
17
18 /** Read from sock an integer encoded by sendInt() and return it */
19 uint32_t recvInt(CommunicatingSocket *sock) throw(std::runtime_error);
20
21 /** Read from sock a string encoded by sendString() and return it */
22 std::string recvString(CommunicatingSocket *sock) throw(std::runtime_error);
23
24 /** Representation for a survey question */
25 struct Question {
26     std::string qText;           //Text of the question.
27     std::vector<std::string> rList; //List of response choices.
28 };
29
30 /** Read survey questions from the given stream and store them in qList. */
31 bool readSurvey(std::istream &stream, std::vector<Question> &qList);
32
33 #endif
```

**SurveyCommon.cpp**

```
1 #include "SurveyCommon.h"
2
3 using namespace std;
4
5 void sendInt(CommunicatingSocket *sock, uint32_t val) throw(SocketException)
{
6     val = htonl(val);           //Convert val to network byte order
7     sock->send(&val, sizeof(val)); //Send the value through the socket
8 }
9
10 void sendString(CommunicatingSocket *sock, const string &str)
11     throw(SocketException) {
12     sendInt(sock, str.length()); //Send the length of string
13     sock->send(str.c_str(), str.length()); //Send string contents
14 }
15
16 uint32_t recvInt(CommunicatingSocket *sock) throw(runtime_error) {
17     uint32_t val;             //Try to read a 32-bit int into val
18     if (sock->recvFully(&val, sizeof(val)) != sizeof(val))
19         throw runtime_error("Socket closed while reading int");
20
21     return ntohl(val);        //Convert to host byte order, return
22 }
23
24 string recvString(CommunicatingSocket *sock) throw(runtime_error) {
25     uint32_t len = recvInt(sock); //Read string length
26     char *buffer = new char [len + 1]; //Temp buffer to hold string
27     if (sock->recvFully(buffer, len) != len){ //Try to read whole string
28         delete [] buffer;
29         throw runtime_error("Socket closed while reading string");
30     }
31
32     buffer[len] = '\0';        //Null terminate the received string
33     string result(buffer);    //Convert to an std::string
34     delete [] buffer;         //Free temporary buffer
35     return result;
36 }
37
38 bool readSurvey(istream &stream, std::vector<Question> &qList) {
39     int count = 0;              //See how many questions there are
40     stream >> count;          //Skip past newline
```

```

42   qList = vector< Question >(count);
43
44 //Parse each question.
45 for (unsigned int q = 0; q < qList.size(); q++) {
46     getline(stream, qList[q].qText); //Get the text of the question
47
48     count = 0;
49     stream >> count;           //Read number of responses
50     stream.ignore();          //Skip past newline
51
52 //Initialize the response list and populate it.
53 qList[q].rList = vector< string >(count);
54 for (unsigned int r = 0; r < qList[q].rList.size(); r++)
55     getline(stream, qList[q].rList[r]);
56 }
57
58 return stream; //Return true if stream is still good
59 }

```

---

(1) 整数编码和解码：第5~8行、第16~22行。

在这个应用程序中，客户和服务器通过交换整数和字符串类型的值进行通信。整数的处理方式与加1服务中非常相似。要编码一个32位的整数，首先要把它转换为网络字节顺序，然后通过套接字发出它。要接收一个整数，我们尝试从套接字中读取4个字节，如果成功，就把它们转换为主机字节顺序，并返回结果。如果不能读取4个字节，就会抛出一个异常。由于这类错误不是在PracticalSocket库自身中产生的，因此会把异常报告为runtime\_error，而不是SocketException。

(2) 字符串编码和解码：第10~14行、第24~36行。

字符串的编码和解码更复杂，因为它们可以是任意长度。在编码字符串时，首先发送字符串的长度，其后接着字符串的内容。解码器尝试读取这两个值，如果成功，就把接收到的字符串内容转换为一个string对象，并返回它。

(3) 解析调查内容：第38~59行。

调查内容存储在一个文本文件中。readSurvey()函数从给定的输入流中读取调查内容，并利用问题序列填充给定的qList参数。

### 8.3.2 调查服务器

调查服务器负责维护调查问题的列表，记录用户的响应总数，并与客户交互。

加1服务器一次只能处理一条客户连接。如果多个客户希望使用服务，它们将不得不轮流进行。这对于简单的应用程序是有意义的，其中我们期望与每个客户进行非常短暂的

交流。不过，它可能不适合于调查服务。在这里，只要用户愿意，就可以仔细考虑每个问题。如果两个用户希望同时接受调查，使一个用户等待另一个用户完成调查就是不合理的。

为了同时与多个客户交互，调查服务器将创建一个单独的线程来处理与每个客户的交互。如 6.4.2 节中所述，新线程管理与客户的会话。每次服务器接收到一条响应时，都会在其响应计数中记录它。互斥性可以帮助服务器确保两个线程不会同时修改响应总数。

## **SurveyServer.cpp**

---

```

1 #include <iostream>
2 #include <fstream>
3 #include <pthread.h>
4 #include "PracticalSocket.h"
5 #include "SurveyCommon.h"
6
7 using namespace std;
8
9 static vector<Question> qList;           //List of survey questions
10 static pthread_mutex_t lock;             //Mutex to Protect critical sections
11 static vector<vector<int>> rCount;    //Response tallies for each question
12
13 /** Thread main function to administer a survey over the given socket */
14 static void *conductSurvey(void *arg);
15
16 int main(int argc, char *argv[]) {
17     if (argc != 2) {
18         cerr << "Usage: SurveyServer <Survey File>" << endl;
19         return 1;
20     }
21
22     ifstream input(argv[1]);           //Read survey from given file
23     if (!input || !readSurvey(input, qList) ) {
24         cerr << "Can't read survey from file: " << argv[1] << endl;
25         return 1;
26     }
27
28     //Initialize response tally for each question/response.
29     for (unsigned int i = 0; i < qList.size(); i++)
30         rCount.push_back(vector<int>(qList[i].rList.size(), 0));
31     pthread_mutex_init(&lock, NULL); //Initialize mutex
32
33     try {
34         //Make a socket to listen for SurveyClient connections.
35         TCPServerSocket servSock(SURVEY_PORT);

```

```
36
37     for (;;) { //Repeatedly accept connections and administer the survey.
38         TCPSocket *sock = servSock.accept();
39
40         pthread_t newThread;           //Give survey in a separate thread
41         if (pthread_create(&newThread, NULL, conductSurvey, sock) != 0) {
42             cerr << "Can't create new thread" << endl;
43             delete sock;
44         }
45     }
46 } catch (SocketException &e) {
47     cerr << e.what() << endl;          //Report errors to the console.
48 }
49
50 return 0;
51 }
52
53 static void *conductSurvey(void *arg) {
54     TCPSocket *sock = (TCPSocket *)arg; //Argument is really a socket
55     try {
56         sendInt(sock, qList.size());      //Tell client no of questions
57
58         for (unsigned int q = 0; q < qList.size(); q++) {
59             //For each question, send the question text and list of responses
60             sendString(sock, qList[q].qText);
61             sendInt(sock, qList[q].rList.size());
62             for (unsigned int r = 0; r < qList[q].rList.size(); r++)
63                 sendString(sock, qList[q].rList[r]);
64
65             //Get the client's response and count it if it's in range
66             unsigned int response = recvInt(sock);
67             if (response >= 0 && response < rCount[q].size()) {
68                 pthread_mutex_lock(&lock); //Lock the mutex
69                 rCount[q][response]++;
70                 pthread_mutex_unlock(&lock); //Release the lock
71             }
72         }
73     } catch (runtime_error e) {
74         cerr << e.what() << endl;          //Report errors to the console.
75     }
76
77     delete sock; //Free the socket object (and close the connection)
78     return NULL;
79 }
```

(1) 访问库函数：第 1~5 行。

服务器使用标准的 C++ I/O 机制和 POSIX 线程用于同多个并发客户进行通信。

(2) 调查表示：第 9~11 行。

调查被表示为 `Question` 实例的向量。变量 `rCount` 记录用户的响应次数，并且 `rCount` 的每个元素对应一个调查问题。由于每个问题都具有多种可能的响应，`rCount` 的每个元素都是总数的向量，其中每个总数用于一个响应。每次用户选择一个响应时，都会递增用于该问题和响应的计数。

如果多个客户同时连接到服务器，那么可能发生其中两个或多个客户同时尝试递增计数器的情况。依赖于硬件执行这种递增的方式，这可能使计数处于一种未知的状态。例如，在一个线程中执行的递增可能重写刚刚在另一个线程中完成的递增的结果。发生这种情况的几率极小，但是不应该允许这种可能性有机会发生。`lock` 变量是一个互斥锁，它用于管理对 `rCount` 的并发访问。通过使用这个同步对象，很容易确保一次只能有一个线程尝试修改 `rCount`。

服务器使用文件范围的变量来记录调查和响应。这使得很容易从任何线程中访问这些数据结构。使用 `static` 修饰符可以阻止这些变量破坏全局命名空间，因为它们只对单个实现文件可见。不过，如果我们希望从同一个应用程序运行多个调查，这种组织方式将会妨碍代码重用。

(3) 初始化服务器状态：第 17~31 行。

服务器解析来自用户提供的文件中的调查文本。如果在这个过程中出现任何错误，就会打印一条错误消息，并且服务器退出。在成功地读取了 `Question` 列表之后，我们就会知道有多少个问题和响应，因此就可以初始化并行的响应计数表示。在可以使用互斥锁 `lock` 之前还必须先初始化它。

(4) 创建服务器套接字并反复接受客户：第 35~45 行。

服务器创建一个 `TCPServerSocket`，用于侦听协商好的端口号。它反复等待客户连接，并为每条连接创建一个新线程来处理与客户之间的交互。在创建线程时，服务器提供一个指向 `conductSurvey` 函数的指针作为新线程的起点，并提供一个指向新 `TCPSocket` 的指针作为线程的参数。从此刻起，新线程就负责套接字对象以及与客户之间的交互。当然，如果不能创建新线程，服务器就会打印一条错误消息，并通过立即删除套接字来执行清理。

(5) 调查客户处理程序：第 53~79 行。

- 线程启动：第 53~54 行。

`conductSurvey` 函数充当每个新线程的主函数。一旦线程启动，就会执行这个函数，并且一旦它从函数中返回，它就会终止。这个函数的参数和返回类型由 Pthreads API 确定。`void` 指针旨在让我们传递指向所需的任何数据类型的指针。在这里，我们知道我们传入了一个指向 `TCPSocket` 实例的指针，因此我们所做的第一件事就是把它

强制转换为一种更具体的类型，使得更容易使用它。

依赖于客户连接的数量，可能同时运行 `conductSurvey` 的多个副本，并且每个副本都运行在它自己的线程上。虽然所有这些线程都可能运行在同一个函数中，但是其中每个线程都具有它自己的局部变量，并且每个线程都通过在线程启动时提供的 `TCPsocket` 的实例进行通信。

- 把每个问题都发送给客户：第 56~63 行。

服务器使用 `SurveyCommon.cpp` 提供的函数来简化与客户之间的交互。它给客户发送调查中的问题数量，然后发送每个问题及其响应列表，并等待问题之间的响应。

- 获取客户响应并记录它：第 66~71 行。

客户通过把用户所选响应的索引发回给服务器来指示响应。在记录响应之前，服务器确保它位于合理响应的范围内。这种检查非常重要。如果服务器使用客户提供的任意索引递增 `rCount`，就会允许客户在未知的内存位置执行更改。恶意客户可能利用这一点尝试使服务器崩溃，或者更糟糕的是接管对服务器主机的控制。当然，我们编写了客户和服务器代码。你将看到，我们甚至在发送响应之前都会对客户执行类似的检查，那么对服务器也执行检查有意义吗？我们不关心正常运行的客户的行为。我们更关心的是：如果恶意用户创建一个不像我们编写的那样良好运行的客户，那么会发生什么事情。

如果接收到合理的响应，客户线程就会锁定互斥锁，确保没有其他线程尝试同时修改 `rCount`。然后，它会递增合适的计数，并立即取消锁定互斥锁，以允许其他线程可以根据需要更改 `rCount`。这是多线程应用程序的典型操作。一般来讲，我们希望把其他线程锁定尽可能短的时间。考虑一下，如果 `conductSurvey` 一启动线程就锁定互斥锁然后在完成时释放它，则会发生什么事情。这样将无需对每个响应锁定和取消锁定互斥锁，但是它将完全禁止并发性；使得一次只能有一个线程可以与其客户交互。

- 错误处理：第 55 行、第 73~75 行。

出现在客户线程中的异常将会终止线程，但是服务器将继续运行，并接受新的连接。这是有意义的，因为这种异常可能仅仅是由于客户意外地终止而产生的。如果在客户交互期间捕获到异常，就无法执行线程退出代码。注意：异常是作为更一般的类型 `runtime_exception` 捕获的，因为它可能由 `PracticalSocket` 抛出，或者由我们自己的 `recvInt()` 或 `recvString()` 函数抛出。

- 关闭套接字并退出：第 77~78 行。

由于 `conductSurvey` 中的线程负责其动态分配的 `TCPClient` 实例，因此在它退出前必须释放该实例。像 C++ 流一样，删除对象会自动关闭底层套接字。

### 8.3.3 调查客户

调查客户不像其服务器那样复杂。在这里，我们不必关心多线程、并发性或者维护响应计数。

#### **SurveyClient.cpp**

```

1 #include <iostream>
2 #include <iomanip>
3 #include "PracticalSocket.h"
4 #include "SurveyCommon.h"
5
6 using namespace std;
7
8 int main(int argc, char *argv[]) {
9     if (argc != 2) { //Make sure the user gives a host
10         cerr << "Usage: SurveyClient <Survey Server Host>" << endl;
11         return 1;
12     }
13
14     try {
15         //Connect to the server.
16         TCPSocket sock(argv[1], SURVEY_PORT);
17
18         //Find out how many questions there are.
19         int qCount = recvInt(&sock);
20         for (int q = 0; q < qCount; q++) {
21             //Show each the question to the user and print the list of responses.
22             cout << "Q" << q << ":" << recvString(&sock) << endl;
23             int rCount = recvInt(&sock);
24             for (int r = 0; r < rCount; r++)
25                 cout << setw(2) << r << " " << recvString(&sock) << endl;
26
27             //Keep prompting the user until we get a legal response.
28             int response = rCount;
29             while (response < 0 || response >= rCount) {
30                 cout << "> ";
31                 cin >> response;
32             }
33
34             //Send the server the user's response
35             sendInt(&sock, response);
36     }

```

```
37 } catch(runtime_error &e) {
38     cerr << e.what() << endl; //Report errors to the console.
39     return 1;
40 }
41
42 return 0;
43 }
```

(1) 访问库函数：第1~4行。

(2) 连接到服务器：第9~16行。

客户期望在命令行上输入服务器的主机名。如果没有提供它，就会打印一条错误消息，并且客户退出。客户尝试使用 SurveyCommon.h 中定义的主机名和服务器的端口号创建一个TCPSocket对象。如果不能建立连接，异常处理代码将报告一个错误并且退出。

(3) 接收并打印调查问题：第19~25行。

使用 SurveyCommon.cpp 提供的函数，客户读取问题的数量，然后读取每个问题及其响应列表。

(4) 读取用户响应，并把它们发送给服务器：第28~35行。

对于每个问题，提示用户选择一个响应。在把响应发送给服务器之前，客户将对其进行检查，以确保它位于正确的范围内。

### 8.3.4 运行服务器和客户

SurveyServer 需要一个命令行参数，即包含调查问题的文件的名称。对于如上所示的调查文件，可以像下面这样运行服务器：

SurveyServer 提供了来自 survey1.txt 中的问题

```
% SurveyServer survey1.txt
```

SurveyClient 需要一个命令行参数，即服务器的主机名或地址。如果服务器运行在名为“earth”的系统上，就可以像下面这样运行客户：

SurveyClient 连接到主机 earth 上的服务器

```
% SurveyClient earth
Q0: What is your favorite flavor of ice cream?
0 Vanilla
1 Chocolate
2 Strawberry
>
```

从这时起，用户就可以通过输入想要的响应的索引来响应问题。一旦回答了所有的问题，客户将向服务器发送一个结束消息，从而结束会话。

题，客户就会终止。

## 8.4 第二种样式的调查服务

尽管调查服务原来可以执行一个比较有用的函数，但是也可以增强它。在本节中，添加了一个管理界面用于报告响应计数，使服务器保存一份完成调查的客户的 IP 地址的列表，并且探讨了一种用于在客户与服务器之间编码和解码消息的替代技术。在对调查服务执行这些增强时，需要用到 PracticalSocket 库的另外几种特性。

### 8.4.1 套接字地址支持

PracticalSocket 库提供了一个 `SocketAddress` 类，它封装了 IP 地址和端口号。它实际上只是 2.4 节中介绍的 `sockaddr_storage` 结构的包装器。这种类型的对象适合用于记录套接字连接的端点，并且库在许多不同的位置使用了它们。

`SocketAddress` 通过其构造函数和 `lookupAddresses()` 静态方法，提供了用于名称解析的简单接口。可以利用地址或主机名以及端口号构造 `SocketAddress` 的实例。此外，也可以使用服务名称（而不是数字）描述端口。两个构造函数都是第 3 章中描述的 `getaddrinfo()` 函数的包装器。底层 `getaddrinfo()` 实际上返回匹配地址的列表，并且这些构造函数简单地使用返回的第一个地址。`lookupAddresses()` 静态方法接受与这些构造函数相同的参数，并在一个 STL 向量中返回所有匹配地址的列表。对于需要在 IPv4 和 IPv6 之间做出选择的应用程序或者当一台主机上有多个网络接口可用时，这可能是有用的。

---

```
SocketAddress(const char *host, in_port_t port) throw(SocketException)
SocketAddress(const char *host, const char *service) throw(SocketException)
static std::vector<SocketAddress>SocketAddress::lookupAddresses(
    const char *host, in_port_t port) throw(SocketException)
static std::vector<SocketAddress>lookupAddresses(const char *host,
    const char *service) throw(SocketException)
```

---

也可以通过查询套接字连接两端的地址获得 `SocketAddress` 的实例。基类 `Socket` 提供了一个 `getLocalAddress()` 方法，它返回套接字绑定到的本地地址的 `SocketAddress`。对于 `TCPServerSocket`，这将给出服务器正在侦听的地址；对于 `TCPSocket`，这将给出连接的本地端的地址。`CommunicatingSocket` 类提供了一个 `getForeignAddress()` 方法，它返回连接的远程端的 `SocketAddress`。一旦获得它，就可以通过 `getAddress()` 和 `getPort()` 方法分别检查 `SocketAddress` 的主机地址和端口号。

---

```
SocketAddress Socket::getLocalAddress() throw(SocketException)
SocketAddress CommunicatingSocket::getForeignAddress()
```

```
        throw(SocketException)
std::string SocketAddress::getAddress() const throw(SocketException)
in_port_t SocketAddress::getPort() const throw(SocketException)
```

对于那些希望对如何建立套接字进行更多控制的用户，提供了 bind() 和 connect() 方法，它们接受 SocketAddress 的一个实例作为参数。应用程序可以使用默认的构造函数创建一个 TCPSocket 或 TCPServerSocket。然后套接字可以利用 bind() 方法绑定到本地地址，对于 TCPSocket，则利用 connect() 方法连接到远程服务器。这允许程序员利用更类似于第 2 章中所描述的控制级别来管理套接字创建。

```
void TCPSocket::bind(const SocketAddress &localAddress)
    throw(SocketException)
void TCPSocket::connect(const SocketAddress &foreignAddress)
    throw(SocketException)
void TCPServerSocket::bind(const SocketAddress &localAddress)
    throw(SocketException)
```

#### 8.4.2 套接字的 iostream 接口

CommunicatingSocket 提供的 send() 和 recv() 方法是用于底层套接字的 send() 和 recv() 函数的非常简化的版本。这种接口适用于一些类型的消息，但它并不最适合用于处理文本编码的消息，如 5.2.2 节中所述。

CommunicatingSocket 的 getStream() 方法返回一个指向由套接字支持的 iostream 的引用。对于 C++ 程序员，这可以充当一个非常方便的接口，用于读、写文本编码的信息。它是与 5.1.5 节中描述的 fopen() 机制类似的 C++ 技术，将通过套接字发送写到 iostream 的字符序列，并且可以从 iostream 读取通过网络接收到的字符序列。利用这个接口，程序员可以使用自己熟悉的适合于 istream 和 ostream 的技术来编码和解码消息。

```
std::iostream &CommunicatingSocket::getStream() throw(SocketException)
```

指向 CommunicatingSocket 的 iostream 接口提供了固定大小的缓冲区，用于存储将要发送和接收的字符序列的一部分。写到 iostream 的文本将保存在内存中，直到缓冲区填满或者冲洗它为止。这可以提供一些性能优势，因为可以一次一个字段地把消息写到 iostream，然后在完成后把它推送给网络。不过，这种缓冲级别使得直接调用 send() 和 recv() 方法通过 iostream 混合 I/O 操作会出现问题。例如，考虑通过 iostream 把某个消息 A 写到套接字。如果随后不冲洗流，消息 A 的一部分可能仍然缓冲在 iostream 中。之后，如果通过套接字的 send() 方法直接发送消息 B，那么将在消息 A 的缓冲的部分之前把消息 B 推送给网络。

### 8.4.3 增强的调查服务器

增强的服务器使用 `SocketAddress` 对象记录完成调查的客户的地址列表。对于每个客户，服务器通过保存表示套接字远程端的 `SocketAddress` 的副本 来记录主机地址。

服务器还提供了一个管理套接字接口，用于报告调查的当前状态。服务器通过只允许传入运行在同一台主机上的客户的连接，限制对管理接口的访问。为此，服务器不能使用上一个示例中使用的用于 `TCPServerSocket` 的方便的构造函数。服务器必须创建一个处于未绑定状态下的 `TCPServerSocket`，然后显式地把它绑定到用于回送接口的 `SocketAddress`。

增强的调查服务器广泛使用了 `CommunicatingSocket` 的 `iostream` 接口。这使得消息的发送和接收看起来很像读、写文件。它还允许在客户与服务器之间重用一些额外的代码。服务器无需使用新格式把调查问题发送给客户，而是使用它在启动时读取的调查文件的相同格式编码它们。客户可以使用 `SurveyCommon.cpp` 提供的相同的 `readSurvey()` 函数一次性读取调查问题的列表。

#### **SurveyServer2.cpp**

```

1 #include <iostream>
2 #include <iomanip>
3 #include <fstream>
4 #include <pthread.h>
5 #include "PracticalSocket.h"
6 #include "SurveyCommon.h"
7
8 using namespace std;
9
10 static vector<Question> qList;           //List of survey questions
11 static pthread_mutex_t lock;               //Mutex to Protect critical sections
12 static vector<vector<int>> rCount;      //Response tallies for each question
13 static vector<SocketAddress> addrList; //Address list for client history
14
15 /** Thread main function to administer a survey over the given socket */
16 void *conductSurvey(void *arg);
17
18 /** Thread main function to monitor an administrative connection and
19     give reports over it. */
20 void *adminServer(void *arg);
21
22 int main(int argc, char *argv[]) {
23     if (argc != 2) {

```

```
24     cerr << "Usage: SurveyServer <Survey File>" << endl;
25     return 1;
26 }
27
28 ifstream input(argv[1]);           //Read survey from given file
29 if (!input || !readSurvey(input, qList) ) {
30     cerr << "Can't read survey from file: " << argv[1] << endl;
31     return 1;
32 }
33
34 //Initialize response tally for each question/response.
35 for (unsigned int i = 0; i < qList.size(); i++)
36     rCount.push_back(vector<int>(qList[i].rList.size(), 0));
37 pthread_mutex_init(&lock, NULL);    //Initialize mutex
38
39 try {
40     pthread_t newThread;
41
42     //Make a thread to provide the administrative interface.
43     if (pthread_create(&newThread, NULL, adminServer, NULL) != 0) {
44         cerr << "Can't create administrative thread" << endl;
45         return 1;
46     }
47
48     //Make a socket to listen for SurveyClient connections.
49     TCPSServerSocket servSock(SURVEY_PORT);
50
51     for (;;) { //Repeatedly accept connections and administer the survey.
52         TCPSocket *sock = servSock.accept();
53         if (pthread_create(&newThread, NULL, conductSurvey, sock) != 0) {
54             cerr << "Can't create new thread" << endl;
55             delete sock;
56         }
57     }
58 } catch (SocketException &e) {
59     cerr << e.what() << endl;          //Report errors to the console.
60 }
61
62     return 0;
63 }
64
65 void *conductSurvey(void *arg) {
66     TCPSocket *sock = (TCPSocket *)arg; //Argument is really a socket.
67     try {
```

```
68 //Write out the survey in the same format as the input file.
69 iostream &stream = sock->getStream();
70 stream << qList.size() << "\n";
71
72 for (unsigned int q = 0; q < qList.size(); q++) {
73     stream << qList[q].qText << "\n";
74     stream << qList[q].rList.size() << "\n";
75     for (unsigned int r = 0; r < qList[q].rList.size(); r++)
76         stream << qList[q].rList[r] << "\n";
77 }
78 stream.flush();
79
80 //Read client responses to questions and record them
81 for (unsigned int q = 0; q < qList.size(); q++) {
82     unsigned int response;
83     stream >> response;
84     if (response >= 0 && response < rCount[q].size()) {
85         pthread_mutex_lock(&lock); //Lock the mutex
86         rCount[q][response]++; //Increment count for chosen item
87         pthread_mutex_unlock(&lock); //Release the lock
88     }
89 }
90
91 //Log this client as completing the survey.
92 pthread_mutex_lock(&lock);
93 addrList.push_back(sock->getForeignAddress());
94 pthread_mutex_unlock(&lock);
95 } catch (runtime_error e) {
96     cerr << e.what() << endl; //Report errors to the console.
97 }
98
99 delete sock; //Free the socket object (and close the connection)
100 return NULL;
101 }
102
103 void *adminServer(void *arg) {
104     try {
105         //Make a ServerSocket to listen for admin connections
106         TCPServerSocket adminSock;
107         adminSock.bind(SocketAddress("127.0.0.1", SURVEY_PORT + 1));
108
109         for (;;) { //Repeatedly accept administrative connections
110             TCPSocket *sock = adminSock.accept();
111             iostream &stream = sock->getStream();
```

```

112
113     try {
114         //Copy response counts and address lists
115         pthread_mutex_lock(&lock);
116         vector<vector<int>> myCount = rCount;
117         vector<SocketAddress> myList = addrList;
118         pthread_mutex_unlock(&lock);
119
120         for (unsigned int q = 0; q < qList.size(); q++) {
121             //Give a report for each question.
122             stream << "Q" << q << ":" << qList[q].qText << "\n";
123             for (unsigned int r = 0; r < qList[q].rList.size(); r++)
124                 stream << setw(5) << myCount[q][r] << " " << qList[q].rList[r]
125                 << "\n";
126         }
127
128         //Report the list of client addresses.
129         stream << "Client Addresses:" << endl;
130         for (unsigned int c = 0; c < myList.size(); c++)
131             stream << " " << myList[c].getAddress() << "\n";
132
133         stream.flush();
134     } catch (runtime_error e) {
135         cerr << e.what() << endl;
136     }
137
138     delete sock; //Free the socket object (and close the connection)
139 }
140 } catch (SocketException e) {
141     cerr << e.what() << endl; //Report errors to the console.
142 }
143 return NULL; //Reached only on error
144 }

```

(1) 访问库函数：第1~6行。

(2) 调查表示：第10~13行。

服务器的这个版本添加了一个 `SocketAddress` 对象的向量，用于记录完成调查的客户的列表。由于多个客户线程可能同时尝试访问这个列表，因此通过互斥锁 `lock` 保护该列表的使用。

(3) 初始化服务器状态：第23~37行。

(4) 创建用于管理接口的线程：第43~46行。

服务器必须接受通过其主 `TCPServerSocket` 及其管理性 `TCPServerSocket` 传入的连接。

如果同一个线程尝试为这两个套接字都提供服务，那么在其中一个套接字上调用 `accept()` 将使之被阻塞，并且不能接受来自另一个套接字的连接。为了同时给它们提供服务，服务器创建了一个新线程，用于处理通过其管理接口传入的连接。`adminServer()` 函数实现了这个接口。

(5) 创建服务器套接字，并利用新线程处理每个客户：第 49~57 行。

(6) 调查客户处理程序：第 65~101 行。

- 编码调查并发送给客户：第 69~78 行。

客户线程首先使用与原始输入文件相同的格式把整个调查的一个副本写到客户。它使用 `getStream()` 为其 `TCPSocket` 获得 `iostream` 的一个副本，然后写出调查，就像把它写到一个文件中一样。我们将不会使用 `endl` 结束每一行，而是使用换行符。使用 `endl` 将在每一行后冲洗流。我们希望尽可能地缓冲完整的消息或者其中的一大部分，然后当消息完整时就发送它。服务器写出完整的消息，然后调用流的 `flush()` 方法，开始发送任何缓冲的数据。

- 获取客户响应并记录它们：第 81~89 行。

客户具有完整的调查，因此服务器只需一个接一个地读取它的响应即可。

- 记录客户：第 92~94 行。

一旦客户完成了调查，服务器就会记录其地址的一个副本。

(7) 管理客户处理程序：第 103~144 行。

- 创建管理套接字：第 106~107 行。

管理接口使用与调查端口相差 1 的端口号。由于服务器只打算接受来自本地主机的管理连接，因此不得不通过更显式的步骤构造 `TCPServerSocket`，并把它绑定到本地地址。

- 反复接受管理连接：第 109~110 行。

利用单个线程为管理客户提供服务。如果两个管理客户尝试连接，第二个管理客户将不得不等待先给第一个管理客户提供服务。

- 快照报告的结构：第 115~118 行。

在发送报告之前，服务器将创建客户地址历史记录和响应计数列表的副本。这使服务器在写出报告时无需反复锁定这些结构，但是更重要的是，它提供了状态的连续快照。在服务器打印出客户地址的报告列表时，它将不会增大。如果服务器只是简单地锁定了用于整个报告生成的互斥锁，而不是复制这些列表，那么在服务器尝试把报告发送给管理客户时所有的客户线程都将会被阻塞。恶意的管理客户可能无法从其套接字读取数据，并且会无限期地挂起调查的实施。

- 编写管理报告：第 120~133 行。

我们使服务器负责格式化管理报告。它把每个问题和响应的计数写到套接字的

iostream，其后接着每个完成调查的客户的主机地址。

#### 8.4.4 增强的调查客户

为了使用文本编码的消息与服务器通信，客户需要进行几处改变。首先，它将不会单独读取调查的问题，而是获取套接字的 iostream，并使用 `readSurvey()` 一次性读取完整的调查。客户一次一个问题地把调查提供给用户，并通过把相应的响应写到流来发送它们。

#### **SurveyClient2.cpp**

```

1 #include <iostream>
2 #include <iomanip>
3 #include "PracticalSocket.h"
4 #include "SurveyCommon.h"
5
6 using namespace std;
7
8 int main(int argc, char *argv[]) {
9     if (argc != 2) { //Make sure the user gives a host
10        cerr << "Usage: SurveyClient <Survey Server Host>" << endl;
11        return 1;
12    }
13
14    try {
15        //Connect to the server.
16        TCPSocket sock(argv[1], SURVEY_PORT);
17        iostream &stream = sock.getStream();
18        vector<Question> qList; //Read the whole survey
19        readSurvey(stream, qList);
20
21        for (unsigned int q = 0; q < qList.size(); q++) {
22            //Show each the question to the user and print the list of responses.
23            cout << "Q" << q << ":" << qList[q].qText << endl;
24            for (unsigned int r = 0; r < qList[q].rList.size(); r++)
25                cout << setw(2) << r << " " << qList[q].rList[r] << endl;
26
27            //Keep prompting the user until we get a legal response.
28            unsigned int response = qList[q].rList.size();
29            while (response < 0 || response >= qList[q].rList.size()) {
30                cout << "> ";
31                cin >> response;
32            }

```

```

33
34     stream << response << endl; //Send user response to server
35     stream.flush();
36 }
37 } catch(SocketException &e) {
38     cerr << e.what() << endl; //Report errors to the console.
39     return 1;
40 }
41
42 return 0;
43 }

```

---

#### 8.4.5 管理客户

非常简单的客户就足以访问管理接口。由于服务器以一种人类可读的格式写出响应计数和地址列表，客户正好可以把它从服务器接收到的每个字节写到控制台。管理客户首先尝试连接到在本地主机上运行的服务器。由于客户尝试使用管理端口号进行连接，服务器将自动给它发送一份报告，然后关闭套接字连接。

客户创建一个固定大小的缓冲区，然后一个接一个地把填满的缓冲区复制到控制台。即使服务器使用其 `iostream` 接口写到套接字，套接字也只是简单地传送所写的字节序列，并且客户无需使用 `recv()` 方法读取该序列。在这种情况下，`recv()` 可能是比 `iostream` 更方便的接口，用于把服务器的输出发送到控制台。客户不必关心服务器的报告是如何格式化的，它包含多少行内容，或者每一行的长度是多少。

服务器的管理报告只是可打印的字符序列，在报告中的任意位置都没有 `null` 终止符。要像字符串一样打印出填满报告的缓冲区，客户必须利用 `null` 终止符标记缓冲区的末尾。由于 `recv()` 方法返回在缓冲区中成功保存的字节数，因此很容易在末尾填充 `null` 终止符。当然，在每次读取缓冲区时，客户都必须确信给这个额外的字符留出空间。这就是为什么客户在调用 `recv()` 时告知缓冲区比其实际容量少 1 字节的原因。

#### **AdminClient2.cpp**

```

1 #include <iostream>
2 #include "PracticalSocket.h"
3 #include "SurveyCommon.h"
4
5 using namespace std;
6
7 int main(int argc, char *argv[]) {
8     try {

```

```

9   //Connect to the server's administrative interface.
10  TCPSocket sock("localhost", SURVEY_PORT + 1);
11
12  //Read the server's report a block at a time.
13  char buffer[ 1025 ];
14  int len;
15  while ((len = sock.recv(buffer, sizeof(buffer) - 1)) != 0) {
16      buffer[len] = '\0';           //Null terminate the sequence
17      cout << buffer;           //And print it like as a string
18  }
19 } catch(SocketException &e) {
20     cerr << e.what() << endl;    //Report errors to the console.
21     exit(1);
22 }
23
24 return 0;
25 }

```

---

#### 8.4.6 运行服务器和客户

增强的调查服务器及其调查客户都利用与原始调查服务器和客户相同的参数运行。管理客户无需任何参数，因为它会自动尝试连接到在本地主机上的已知端口号上运行的调查服务器的实例。

### 练习题

1. 在用于管理接口的服务器代码中，在我们开始生成报告之前，我们努力获取响应计数和客户地址列表的一致的快照。不过，由于服务器的 `conductSurvey()` 函数一接收到客户响应就会记录它们，管理报告可能仍会包括部分完成的调查的结果。修改服务器，使得仅当客户完成了调查之后才会记录响应。这样，响应计数和客户地址的列表只会反映实际地完成了调查的客户。
2. 对于本章中编写的服务器，客户有可能导致服务器崩溃。如果在服务器给客户发送消息时客户选择了它的套接字连接，服务器将会接收到一个 `SIGPIPE` 信号。参考 6.2 节并修改服务器，使得它在这些情况下不会终止。
3. 调查服务器会维护代表每个接受调查的客户的开放套接字和线程。如果客户从未完成调查，那么服务器永远都不会收回这些资源。扩展服务器，使得在客户开始接受调查 5 分钟之后关闭套接字，并且允许客户线程自动终止。当客户线程启动时，还将启动另一个

称为睡眠者的线程。睡眠者将简单地等待 5 分钟，然后检查客户线程是否仍然在运行。如果是，睡眠者将关闭客户的套接字，它将解除阻塞服务器线程，并给它提供一个退出的机会。要正确地执行该任务，将需要服务器维护某种额外的每个客户的状态，并且执行某种额外的同步。例如，如果其客户线程已经完成并且删除了套接字，睡眠者线程就不应该调用 `close()`。

# 参考文献

- [1] Comer, Douglas E. *Internetworking with TCP/IP, volume 1, Principles, Protocols, and Architecture* (fifth edition). Prentice Hall, 2005.
- [2] Comer, Douglas E., and Stevens, David L. *Internetworking with TCP/IP, volume 2, Design, Implementation, and Internals* (third edition). Prentice Hall, 1999.
- [3] Comer, Douglas E., and Stevens, David L. *Internetworking with TCP/IP, volume 3, Client-Server Programming and Applications* (BSD version, second edition). Prentice Hall, 1996.
- [4] Hinden, R., and Deering, S. "IP Version 6 Addressing Architecture." Internet Request for Comments 4291, February 2006.
- [5] Deering, S., and Hinden, R. "Internet Protocol, Version 6 (IPv6) Specification." Internet Request for Comments 2460, December 1998.
- [6] Gilligan, R., Thomson, S., Bound, J., McCann, J., and Stevens, W. "Basic Socket Interface Extensions for IPv6." Internet Request for Comments 3493, February 2003.
- [7] Srisuresh, P., and Egevang, S. "Traditional IP Network Address Translator (Traditional NAT)." Internet Request for Comments 3022, January 2001.
- [8] Mockapetris, Paul. "Domain Names: Concepts and Facilities." Internet Request for Comments 1034, November 1987.
- [9] Mockapetris, Paul. "Domain Names: Implementation and Specification." Internet Request for Comments 1035, November 1987.
- [10] Peterson, Larry L., and Davie, Bruce S. *Computer Networks: A Systems Approach* (fourth edition). Morgan Kaufmann, 2007.
- [11] M-K. Shin, et al. "Application Aspects of IPv6 Transition." Internet Request for Comments 4038, March 2005.
- [12] Postel, John. "Internet Protocol." Internet Request for Comments 791, September 1981.
- [13] Postel, John. "Transmission Control Protocol." Internet Request for Comments 793, September 1981.

- [14] Postel, John. "User Datagram Protocol." Internet Request for Comments 768, August 1980.
- [15] Stevens, W. Richard. *TCP/IP Illustrated*, volume 1, *The Protocols*. Addison-Wesley, 1994.
- [16] Stevens, W. Richard. *UNIX Network Programming: Networking APIs: Sockets and XTI* (second edition). Prentice Hall, 1997.
- [17] Wright, Gary R., and Stevens, W. Richard. *TCP/IP Illustrated*, volume 2, *The Implementation*. Addison-Wesley, 1995.