计 算 机 科 学 丛 书

Addison-Wesley

TCP/IP详解

卷1：协议

TCP/IP Illustrated Volume 1: The Protocols

(美) W. Richard Stevens 著

范建华 胥光辉 张 涛 等译

谢希仁 校

机械工业出版社
China Machine Press

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

第1章　概　　述

1.1　引言

很多不同的厂家生产各种型号的计算机，它们运行完全不同的操作系统，但 TCP/IP协议族允许它们互相进行通信。这一点很让人感到吃惊，因为它的作用已远远超出了起初的设想。TCP/IP起源于60年代末美国政府资助的一个分组交换网络研究项目，到 90年代已发展成为计算机之间最常应用的组网形式。它是一个真正的开放系统，因为协议族的定义及其多种实现可以不用花钱或花很少的钱就可以公开地得到。它成为被称作“全球互联网”或“因特网(Internet)”的基础，该广域网（WAN）已包含超过100万台遍布世界各地的计算机。

本章主要对TCP/IP协议族进行概述，其目的是为本书其余章节提供充分的背景知识。如果读者要从历史的角度了解有关 TCP/IP的早期发展情况，请参考文献 [Lynch 1993]。

1.2　分层

网络协议通常分不同层次进行开发，每一层分别负责不同的通信功能。一个协议族，比如 TCP/IP，是一组不同层次上的多个协议的组合。 TCP/IP通常被认为是一个四层协议系统，如图 1-1所示。

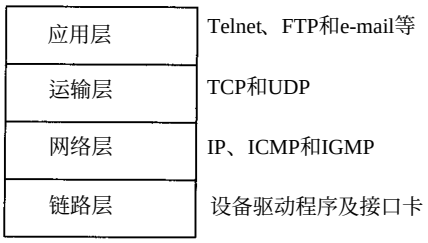


图1-1　TCP/IP协议族的四个层次

每一层负责不同的功能：

1) 链路层，有时也称作数据链路层或网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆（或其他任何传输媒介）的物理接口细节。
2) 网络层，有时也称作互联网层，处理分组在网络中的活动，例如分组的选路。在 TCP/IP协议族中，网络层协议包括 IP协议（网际协议），ICMP协议（Internet互联网控制报文协议），以及IGMP协议（Internet组管理协议）。
3) 运输层主要为两台主机上的应用程序提供端到端的通信。在 TCP/IP协议族中，有两个互不相同的传输协议：TCP（传输控制协议）和UDP（用户数据报协议）。
   TCP为两台主机提供高可靠性的数据通信。它所做的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最后确认分组的超时时钟等。由于运输层提供了高可靠性的端到端的通信，因此应用层可以忽略所有这些细节。
   而另一方面， UDP则为应用层提供一种非常简单的服务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。任何必需的可靠性必须由应用层来提供。
   这两种运输层协议分别在不同的应用程序中有不同的用途，这一点将在后面看到。
4) 应用层负责处理特定的应用程序细节。几乎各种不同的 TCP/IP实现都会提供下面这些通用的应用程序：

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

- Telnet 远程登录。
- FTP 文件传输协议。
- SMTP 简单邮件传送协议。
- SNMP 简单网络管理协议。

另外还有许多其他应用，在后面章节中将介绍其中的一部分。

假设在一个局域网（LAN）如以太网中有两台主机，二者都运行 FTP协议，图1-2列出了该过程所涉及到的所有协议。

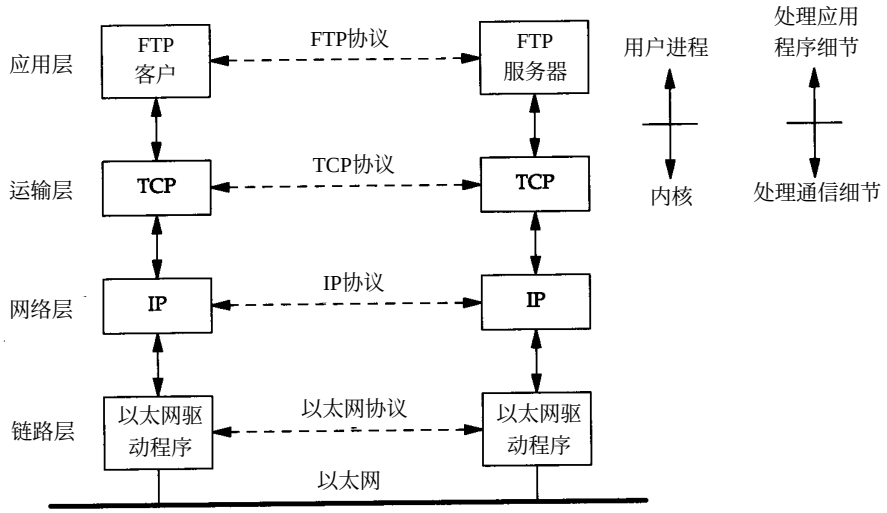


图1-2 局域网上运行FTP的两台主机

这里，我们列举了一个FTP客户程序和另一个FTP服务器程序。大多数的网络应用程序都被设计成客户—服务器模式。服务器为客户提供某种服务，在本例中就是访问服务器所在主机上的文件。在远程登录应用程序Telnet中，为客户提供的服务是登录到服务器主机上。

在同一层上，双方都有对应的一个或多个协议进行通信。例如，某个协议允许 TCP层进行通信，而另一个协议则允许两个IP层进行通信。

在图1-2的右边，我们注意到应用程序通常是一个用户进程，而下三层则一般在（操作系统）内核中执行。尽管这不是必需的，但通常都是这样处理的，例如 UNIX操作系统。

在图1-2中，顶层与下三层之间还有另一个关键的不同之处。应用层关心的是应用程序的细节，而不是数据在网络中的传输活动。下三层对应用程序一无所知，但它们要处理所有的通信细节。

在图1-2中列举了四种不同层次上的协议。 FTP是一种应用层协议， TCP是一种运输层协议，IP是一种网络层协议，而以太网协议则应用于链路层上。 TCP/IP协议族是一组不同的协议组合在一起构成的协议族。尽管通常称该协议族为 TCP/IP，但TCP和IP只是其中的两种协议而已（该协议族的另一个名字是Internet协议族(Internet Protocol Suite)）。

网络接口层和应用层的目的是很显然的——前者处理有关通信媒介的细节（以太网、令牌环网等），而后者处理某个特定的用户应用程序（ FTP、Telnet等）。但是，从表面上看，网络层和运输层之间的区别不那么明显。为什么要把它们划分成两个不同的层次呢？为了理解这一点，我们必须把视野从单个网络扩展到一组网络。

Linux公社 www.linuxidc.com

在80年代，网络不断增长的原因之一是大家都意识到只有一台孤立的计算机构成的“孤岛”没有太大意义，于是就把这些孤立的系统组在一起形成网络。随着这样的发展，到了 90年代，我们又逐渐认识到这种由单个网络构成的新的更大的“岛屿”同样没有太大的意义。于是，人们又把多个网络连在一起形成一个网络的网络，或称作互连网 (internet)。一个互连网就是一组通过相同协议族互连在一起的网络。

构造互连网最简单的方法是把两个或多个网络通过路由器进行连接。它是一种特殊的用于网络互连的硬件盒。路由器的好处是为不同类型的物理网络提供连接：以太网、令牌环网、点对点的链接和FDDI（光纤分布式数据接口）等等。

这些盒子也称作IP路由器（IP Router），但我们这里使用路由器(Router)这个术语。从历史上说，这些盒子称作网关（gateway），在很多TCP/IP文献中都使用这个术语。现在网关这个术语只用来表示应用层网关：一个连接两种不同协议族的进程（例如，TCP/IP和IBM的SNA），它为某个特定的应用程序服务（常常是电子邮件或文件传输）。

图1-3是一个包含两个网络的互连网：一个以太网和一个令牌环网，通过一个路由器互相连接。尽管这里是两台主机通过路由器进行通信，实际上以太网中的任何主机都可以与令牌环网中的任何主机进行通信。

在图1-3中，我们可以划分出端系统（End system）（两边的两台主机）和中间系统（Intermediate system）（中间的路由器）。应用层和运输层使用端到端（End-to-end）协议。在图中，只有端系统需要这两层协议。但是，网络层提供的却是逐跳（Hop-by-hop）协议，两个端系统和每个中间系统都要使用它。

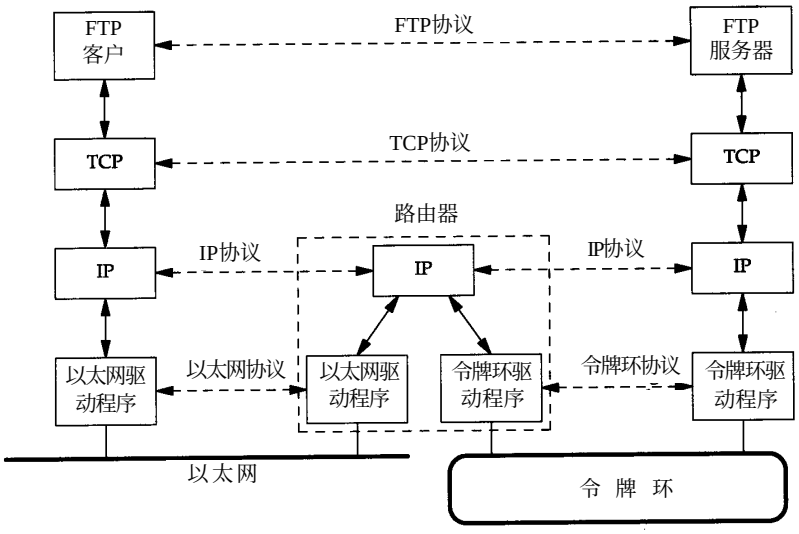


图1-3 通过路由器连接的两个网络

在TCP/IP协议族中，网络层IP提供的是一种不可靠的服务。也就是说，它只是尽可能快地把分组从源结点送到目的结点，但是并不提供任何可靠性保证。而另一方面， TCP在不可靠的IP层上提供了一个可靠的运输层。为了提供这种可靠的服务， TCP采用了超时重传、发送和接收端到端的确认分组等机制。由此可见，运输层和网络层分别负责不同的功能。

从定义上看，一个路由器具有两个或多个网络接口层（因为它连接了两个或多个网络）。

Linux公社 www.linuxidc.com

任何具有多个接口的系统，英文都称作是多接口的(multihomed)。一个主机也可以有多个接口，但一般不称作路由器，除非它的功能只是单纯地把分组从一个接口传送到另一个接口。同样，路由器并不一定指那种在互联网中用来转发分组的特殊硬件盒。大多数的 TCP/IP实现也允许一个多接口主机来担当路由器的功能，但是主机为此必须进行特殊的配置。在这种情况下，我们既可以称该系统为主机（当它运行某一应用程序时，如 FTP或Telnet），也可以称之为路由器（当它把分组从一个网络转发到另一个网络时）。在不同的场合下使用不同的术语。

互联网的目的之一是在应用程序中隐藏所有的物理细节。虽然这一点在图 1-3由两个网络组成的互联网中并不很明显，但是应用层不能关心（也不关心）一台主机是在以太网上，而另一台主机是在令牌环网上，它们通过路由器进行互连。随着增加不同类型的物理网络，可能会有20个路由器，但应用层仍然是一样的。物理细节的隐藏使得互联网功能非常强大，也非常有用。

连接网络的另一个途径是使用网桥。网桥是在链路层上对网络进行互连，而路由器则是在网络层上对网络进行互连。网桥使得多个局域网（ LAN）组合在一起，这样对上层来说就好像是一个局域网。

TCP /IP倾向于使用路由器而不是网桥来连接网络，因此我们将着重介绍路由器。文献[Perlman 1992]的第12章对路由器和网桥进行了比较。

1.3 TCP/IP的分层

在TCP/IP协议族中，有很多种协议。图 1-4给出了本书将要讨论的其他协议。

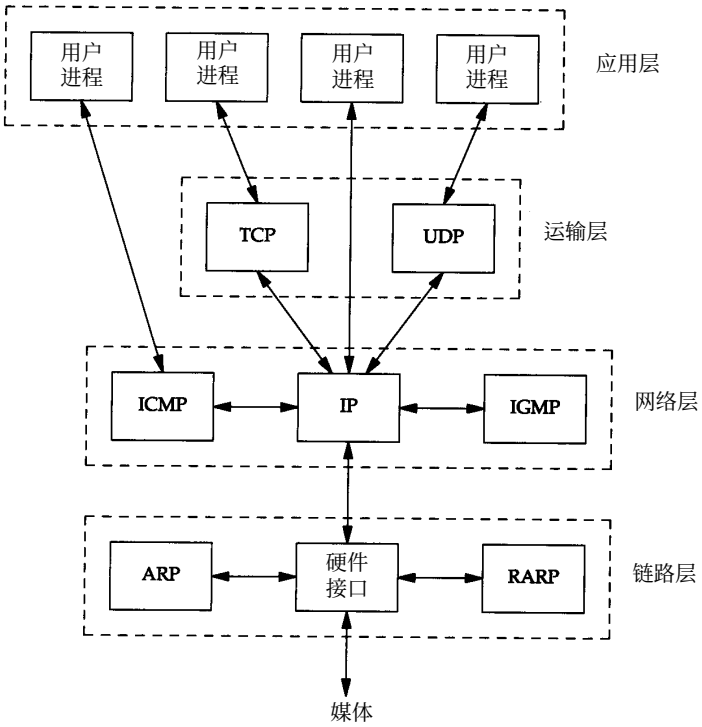


图1-4 TCP/IP协议族中不同层次的协议

Linux公社 www.linuxidc.com

TCP和UDP是两种最为著名的运输层协议，二者都使用IP作为网络层协议。

虽然TCP使用不可靠的IP服务，但它却提供一种可靠的运输层服务。本书第17～22章将详细讨论TCP的内部操作细节。然后，我们将介绍一些TCP的应用，如第26章中的Telnet和Rlogin、第27章中的FTP以及第28章中的SMTP等。这些应用通常都是用户进程。

UDP为应用程序发送和接收数据报。一个数据报是指从发送方传输到接收方的一个信息单元（例如，发送方指定的一定字节数的信息）。但是与TCP不同的是，UDP是不可靠的，它不能保证数据报能安全无误地到达最终目的。本书第11章将讨论UDP，然后在第14章（DNS:域名系统），第15章（TFTP：简单文件传送协议），以及第16章（BOOTP：引导程序协议）介绍使用UDP的应用程序。SNMP也使用了UDP协议，但是由于它还要处理许多其他的协议，因此本书把它留到第25章再进行讨论。

IP是网络层上的主要协议，同时被TCP和UDP使用。TCP和UDP的每组数据都通过端系统和每个中间路由器中的IP层在互联网中进行传输。在图1-4中，我们给出了一个直接访问IP的应用程序。这是很少见的，但也是可能的（一些较老的选路协议就是以这种方式来实现的。当然新的运输层协议也有可能使用这种方式）。第3章主要讨论IP协议，但是为了使内容更加有针对性，一些细节将留在后面的章节中进行讨论。第9章和第10章讨论IP如何进行选路。

ICMP是IP协议的附属协议。IP层用它来与其他主机或路由器交换错误报文和其他重要信息。第6章对ICMP的有关细节进行讨论。尽管ICMP主要被IP使用，但应用程序也有可能访问它。我们将分析两个流行的诊断工具，Ping和Traceroute（第7章和第8章），它们都使用了ICMP。

IGMP是Internet组管理协议。它用来把一个UDP数据报多播到多个主机。我们在第12章中描述广播（把一个UDP数据报发送到某个指定网络上的所有主机）和多播的一般特性，然后在第13章中对IGMP协议本身进行描述。

ARP（地址解析协议）和RARP（逆地址解析协议）是某些网络接口（如以太网和令牌环网）使用的特殊协议，用来转换IP层和网络接口层使用的地址。我们分别在第4章和第5章对这两种协议进行分析和介绍。

1.4 互联网的地址

互联网上的每个接口必须有一个唯一的Internet地址（也称作IP地址）。IP地址长32 bit。Internet地址并不采用平面形式的地址空间，如1、2、3等。IP地址具有一定的结构，五类不同的互联网地址格式如图1-5所示。

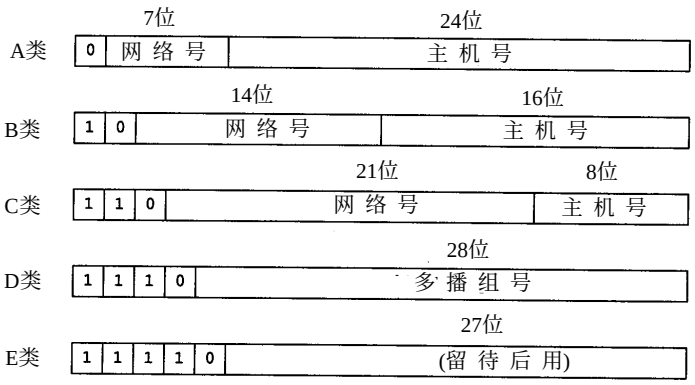


图1-5 五类互联网地址

Linux公社 www.linuxidc.com

这些32位的地址通常写成四个十进制的数，其中每个整数对应一个字节。这种表示方法称作“点分十进制表示法（Dotted decimal notation）”。例如，作者的系统就是一个B类地址，它表示为：140.252.13.33。

区分各类地址的最简单方法是看它的第一个十进制整数。图1-6列出了各类地址的起止范围，其中第一个十进制整数用加黑字体表示。

类型	范 围
A	**0**.0.0.0 到 **127**.255.255.255
B	**128**.0.0.0 到 **191**.255.255.255
C	**192**.0.0.0 到 **223**.255.255.255
D	**224**.0.0.0 到 **239**.255.255.255
E	**240**.0.0.0 到 **247**.255.255.255

图1-6 各类IP地址的范围

需要再次指出的是，多接口主机具有多个IP地址，其中每个接口都对应一个IP地址。

由于互联网上的每个接口必须有一个唯一的IP地址，因此必须要有一个管理机构为接入互联网的网络分配IP地址。这个管理机构就是互联网络信息中心（Internet Network Information Centre），称作InterNIC。InterNIC只分配网络号。主机号的分配由系统管理员来负责。

Internet注册服务(IP地址和DNS域名)过去由NIC来负责，其网络地址是`nic.ddn.mil`。1993年4月1日，InterNIC成立。现在，NIC只负责处理国防数据网的注册请求，所有其他的Internet用户注册请求均由InterNIC负责处理，其网址是：`rs.internic.net`。

事实上InterNIC由三部分组成：注册服务（`rs.internic.net`），目录和数据库服务（`ds.internic.net`），以及信息服务（`is.internic.net`）。有关InterNIC的其他信息参见习题1.8。

有三类IP地址：单播地址（目的为单个主机）、广播地址（目的端为给定网络上的所有主机）以及多播地址（目的端为同一组内的所有主机）。第12章和第13章将分别讨论广播和多播的更多细节。

在3.4节中，我们在介绍IP选路以后将进一步介绍子网的概念。图3-9给出了几个特殊的IP地址：主机号和网络号为全0或全1。

1.5 域名系统

尽管通过IP地址可以识别主机上的网络接口，进而访问主机，但是人们最喜欢使用的还是主机名。在TCP/IP领域中，域名系统（DNS）是一个分布的数据库，由它来提供IP地址和主机名之间的映射信息。我们在第14章将详细讨论DNS。

现在，我们必须理解，任何应用程序都可以调用一个标准的库函数来查看给定名字的主机的IP地址。类似地，系统还提供一个逆函数——给定主机的IP地址，查看它所对应的主机名。

大多数使用主机名作为参数的应用程序也可以把IP地址作为参数。例如，在第4章中当我们用Telnet进行远程登录时，既可以指定一个主机名，也可以指定一个IP地址。

1.6 封装

当应用程序用TCP传送数据时，数据被送入协议栈中，然后逐个通过每一层直到被当作一串比特流送入网络。其中每一层对收到的数据都要增加一些首部信息（有时还要增加尾部信息），该过程如图1-7所示。TCP传给IP的数据单元称作TCP报文段或简称为TCP段（TCP segment）。IP传给网络接口层的数据单元称作IP数据报(IP datagram)。通过以太网传输的比特流称作帧(Frame)。

Linux公社 www.linuxidc.com

图1-7中帧头和帧尾下面所标注的数字是典型以太网帧首部的字节长度。在后面的章节中我们将详细讨论这些帧头的具体含义。

以太网数据帧的物理特性是其长度必须在 46～1500字节之间。我们将在4.5节遇到最小长度的数据帧，在2.8节中遇到最大长度的数据帧。

所有的Internet标准和大多数有关TCP/IP的书都使用octet这个术语来表示字节。使用这个过分雕琢的术语是有历史原因的，因为TCP/IP的很多工作都是在DEC-10系统上进行的，但是它并不使用8 bit的字节。由于现在几乎所有的计算机系统都采用8 bit的字节，因此我们在本书中使用字节（byte）这个术语。

更准确地说，图1-7中IP和网络接口层之间传送的数据单元应该是分组（packet）。分组既可以是一个IP数据报，也可以是IP数据报的一个片（fragment）。我们将在11.5节讨论IP数据报分片的详细情况。

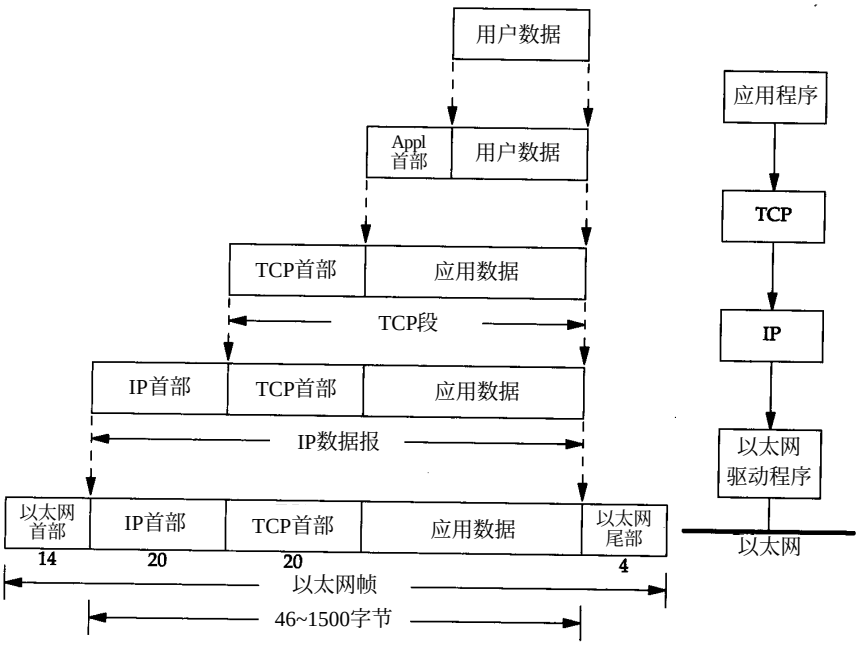


图1-7 数据进入协议栈时的封装过程

UDP数据与TCP数据基本一致。唯一的不同是 UDP传给IP的信息单元称作 UDP数据报（UDP datagram），而且UDP的首部长为8字节。

回想1.3节中的图1-4，由于TCP、UDP、ICMP和IGMP都要向IP传送数据，因此IP必须在生成的IP首部中加入某种标识，以表明数据属于哪一层。为此， IP在首部中存入一个长度为8bit的数值，称作协议域。1表示为ICMP协议，2表示为IGMP协议，6表示为TCP协议，17表示为UDP协议。

类似地，许多应用程序都可以使用 TCP或UDP来传送数据。运输层协议在生成报文首部时要存入一个应用程序的标识符。TCP和UDP都用一个16bit的端口号来表示不同的应用程序。TCP和UDP把源端口号和目的端口号分别存入报文首部中。

网络接口分别要发送和接收IP、ARP和RARP数据，因此也必须在以太网的帧首部中加入

Linux公社 www.linuxidc.com

某种形式的标识，以指明生成数据的网络层协议。为此，以太网的帧首部也有一个 16 bit的帧类型域。

1.7 分用

当目的主机收到一个以太网数据帧时，数据就开始从协议栈中由底向上升，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。这个过程称作分用（Demultiplexing），图1-8显示了该过程是如何发生的。

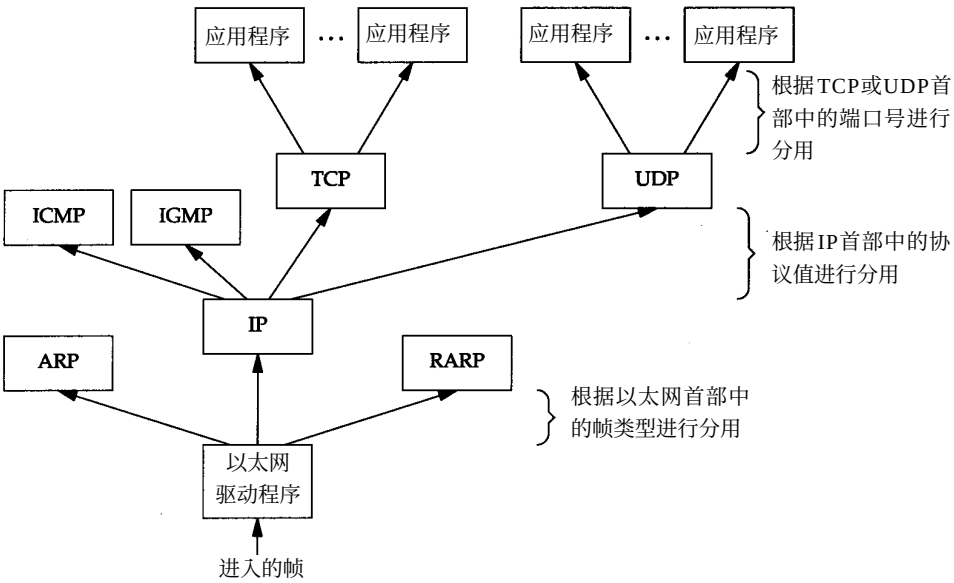


图1-8 以太网数据帧的分用过程

为协议ICMP和IGMP定位一直是一件很棘手的事情。在图1-4中，把它们与IP放在同一层上，那是因为事实上它们是IP的附属协议。但是在这里，我们又把它们放在IP层的上面，这是因为ICMP和IGMP报文都被封装在IP数据报中。

对于ARP和RARP，我们也遇到类似的难题。在这里把它们放在以太网设备驱动程序的上方，这是因为它们和IP数据报一样，都有各自的以太网数据帧类型。但在图2-4中，我们又把ARP作为以太网设备驱动程序的一部分，放在IP层的下面，其原因在逻辑上是合理的。

这些分层协议盒并不都是完美的。

当进一步描述TCP的细节时，我们将看到协议确实是通过目的端口号、源 IP地址和源端口号进行解包的。

1.8 客户-服务器模型

大部分网络应用程序在编写时都假设一端是客户，另一端是服务器，其目的是为了让服务器为客户提供一些特定的服务。

可以将这种服务分为两种类型：重复型或并发型。重复型服务器通过以下步骤进行交互：

Linux公社 www.linuxidc.com

I1. 等待一个客户请求的到来。

I2. 处理客户请求。

I3. 发送响应给发送请求的客户。

I4. 返回I1步。

重复型服务器主要的问题发生在I2状态。在这个时候，它不能为其他客户机提供服务。

相应地，并发型服务器采用以下步骤：

C1. 等待一个客户请求的到来。

C2. 启动一个新的服务器来处理这个客户的请求。在这期间可能生成一个新的进程、任务或线程，并依赖底层操作系统的支持。这个步骤如何进行取决于操作系统。生成的新服务器对客户的全部请求进行处理。处理结束后，终止这个新服务器。

C3. 返回C1步。

并发服务器的优点在于它是利用生成其他服务器的方法来处理客户的请求。也就是说，每个客户都有它自己对应的服务器。如果操作系统允许多任务，那么就可以同时为多个客户服务。

对服务器，而不是对客户进行分类的原因是因为对于一个客户来说，它通常并不能够辨别自己是与一个重复型服务器或并发型服务器进行对话。

一般来说，TCP服务器是并发的，而UDP服务器是重复的，但也存在一些例外。我们将在11.12节对UDP对其服务器产生的影响进行详细讨论，并在18.11节对TCP对其服务器的影响进行讨论。

1.9 端口号

前面已经指出过，TCP和UDP采用16 bit的端口号来识别应用程序。那么这些端口号是如何选择的呢？

服务器一般都是通过知名端口号来识别的。例如，对于每个TCP/IP实现来说，FTP服务器的TCP端口号都是21，每个Telnet服务器的TCP端口号都是23，每个TFTP(简单文件传送协议)服务器的UDP端口号都是69。任何TCP/IP实现所提供的服务都用知名的1～1023之间的端口号。这些知名端口号由Internet号分配机构（Internet Assigned Numbers Authority, IANA）来管理。

> 到1992年为止，知名端口号介于1～255之间。256～1023之间的端口号通常都是由Unix系统占用，以提供一些特定的Unix服务——也就是说，提供一些只有Unix系统才有的、而其他操作系统可能不提供的服务。现在IANA管理1～1023之间所有的端口号。
>
> Internet扩展服务与Unix特定服务之间的一个差别就是Telnet和Rlogin。它们二者都允许通过计算机网络登录到其他主机上。Telnet是采用端口号为23的TCP/IP标准且几乎可以在所有操作系统上进行实现。相反，Rlogin最开始时只是为Unix系统设计的（尽管许多非Unix系统现在也提供该服务），因此在80年代初，它的有名端口号为513。

客户端通常对它所使用的端口号并不关心，只需保证该端口号在本机上是唯一的就可以了。客户端口号又称作临时端口号（即存在时间很短暂）。这是因为它通常只是在用户运行该客户程序时才存在，而服务器则只要主机开着的，其服务就运行。

大多数TCP/IP实现给临时端口分配1024～5000之间的端口号。大于5000的端口号是为其

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

他服务器预留的（Internet上并不常用的服务）。我们可以在后面看见许多这样的给临时端口分配端口号的例子。

Solaris 2.2是一个很有名的例外。通常TCP和UDP的缺省临时端口号从32768开始。在E.4节中，我们将详细描述系统管理员如何对配置选项进行修改以改变这些缺省项。

大多数Unix系统的文件/etc/services都包含了人们熟知的端口号。为了找到 Telnet服务器和域名系统的端口号，可以运行以下语句：

```
sun % grep telnet /etc/services
telnet     23/tcp     称它使用TCP端口号23
sun % grep domain /etc/services
domain     53/udp     称它使用UDP端口号53和TCP端口号53
domain     53/tcp
```

保留端口号

Unix系统有保留端口号的概念。只有具有超级用户特权的进程才允许给它自己分配一个保留端口号。

这些端口号介于1～1023之间，一些应用程序（如有名的 Rlogin，26.2节）将它作为客户与服务器之间身份认证的一部分。

1.10 标准化过程

究竟是谁控制着 TCP/IP协议族，又是谁在定义新的标准以及其他类似的事情？事实上，有四个小组在负责Internet技术。

1) Internet协会（ISOC，Internet Society）是一个推动、支持和促进Internet不断增长和发展的专业组织，它把Internet作为全球研究通信的基础设施。

2) Internet体系结构委员会（IAB，Internet Architecture Board）是一个技术监督和协调的机构。它由国际上来自不同专业的 15个志愿者组成，其职能是负责 Internet标准的最后编辑和技术审核。IAB隶属于ISOC。

3) Internet工程专门小组（IETF，Internet Engineering Task Force）是一个面向近期标准的组织，它分为9个领域（应用、寻径和寻址、安全等等）。IETF开发成为Internet标准的规范。为帮助IETF主席，又成立了Internet工程指导小组（IESG，Internet Engineering Steering Group）。

4) Internet研究专门小组（IRIF，Internet Research Task Force）主要对长远的项目进行研究。

IRTF和IETF都隶属于IAB。文献[Crocker 1993]提供了关于Internet内部标准化进程更为详细的信息，同时还介绍了它的早期历史。

1.11 RFC

所有关于Internet的正式标准都以RFC（Request for Comment）文档出版。另外，大量的RFC并不是正式的标准，出版的目的只是为了提供信息。 RFC的篇幅从1页到200页不等。每一项都用一个数字来标识，如RFC 1122，数字越大说明RFC的内容越新。

所有的RFC都可以通过电子邮件或用 FTP从Internet上免费获取。如果发送下面这份电子邮件，就会收到一份获取RFC的方法清单：

Linux公社 www.linuxidc.com

To: rfc-info@ISI.EDU
Subject: getting rfcs
help: ways_to_get_rfcs

最新的RFC索引总是搜索信息的起点。这个索引列出了 RFC被替换或局部更新的时间。下面是一些重要的RFC文档：

1) 赋值RFC（Assigned Numbers RFC）列出了所有Internet协议中使用的数字和常数。至本书出版时为止，最新 RFC的编号是1340 [Reynolds和Postel 1992]。所有著名的Internet端口号都列在这里。
   当这个RFC被更新时(通常每年至少更新一次)，索引清单会列出RFC 1340被替换的时间。
2) Internet正式协议标准，目前是RFC 1600[Postel 1994]。这个RFC描述了各种Internet协议的标准化现状。每种协议都处于下面几种标准化状态之一：标准、草案标准、提议标准、实验标准、信息标准和历史标准。另外，对每种协议都有一个要求的层次、必需的、建议的、可选择的、限制使用的或者不推荐的。
   与赋值RFC一样，这个RFC也定期更新。请随时查看最新版本。
3) 主机需求RFC，1122和1123[Braden 1989a, 1989b]。RFC 1122针对链路层、网络层和运输层；RFC 1123针对应用层。这两个RFC对早期重要的RFC文档作了大量的纠正和解释。如果要查看有关协议更详细的细节内容，它们通常是一个入口点。它们列出了协议中关于“必须”、“应该”、“可以”、“不应该”或者“不能”等特性及其实现细节。
   文献[Borman 1993b]提供了有关这两个RFC的实用内容。RFC 1127[Braden 1989c]对工作小组开发主机需求RFC过程中的讨论内容和结论进行了非正式的总结。
4) 路由器需求RFC，目前正式版是RFC 1009[Braden and Postel 1987]，但一个新版已接近完成[Almquist 1993]。它与主机需求RFC类似，但是只单独描述了路由器的需求。

1.12 标准的简单服务

有一些标准的简单服务几乎每种实现都要提供。在本书中我们将使用其中的一些服务程序，而客户程序通常选择 Telnet。图1-9描述了这些服务。从该图可以看出，当使用 TCP和UDP提供相同的服务时，一般选择相同的端口号。

名 字	TCP端口号	UDP端口号	RFC	描 述
echo	7	7	862	服务器返回客户发送的所有内容
discard	9	9	863	服务器丢弃客户发送的所有内容
daytime	13	13	867	服务器以可读形式返回时间和日期
chargen	19	19	864	当客户发送一个数据报时，TCP服务器发送一串连续的字符流，直到客户中断连接。UDP服务器发送一个随机长度的数据报
time	37	37	868	服务器返回一个二进制形式的32 bit数，表示从UTC时间1900年1月1日午夜至今的秒数

图1-9 大多数实现都提供的标准的简单服务

Linux公社 www.linuxidc.com

如果仔细检查这些标准的简单服务以及其他标准的 TCP/IP服务（如Telnet、FTP、SMTP等）的端口号时，我们发现它们都是奇数。这是有历史原因的，因为这些端口号都是从NCP端口号派生出来的（NCP，即网络控制协议，是ARPANET的运输层协议，是TCP的前身）。NCP是单工的，不是全双工的，因此每个应用程序需要两个连接，需预留一对奇数和偶数端口号。当 TCP和UDP成为标准的运输层协议时，每个应用程序只需要一个端口号，因此就使用了NCP中的奇数。

1.13 互联网

在图1-3中，我们列举了一个由两个网络组成的互联网 —— 一个以太网和一个令牌环网。在1.4节和1.9节中，我们讨论了世界范围内的互联网 —— Internet，以及集中分配IP地址的需要（InterNIC），还讨论了知名端口号（IANA）。internet这个词第一个字母是否大写决定了它具有不同的含义。

internet意思是用一个共同的协议族把多个网络连接在一起。而 Internet指的是世界范围内通过TCP/IP互相通信的所有主机集合（超过 100万台）。Internet是一个internet，但internet不等于Internet。

1.14 实现

既成事实标准的 TCP/IP软件实现来自于位于伯克利的加利福尼亚大学的计算机系统研究小组。从历史上看，软件是随同4.x BSD系统（Berkeley Software Distribution）的网络版一起发布的。它的源代码是许多其他实现的基础。

图1-10列举了各种 BSD版本发布的时间，并标注了重要的 TCP/IP特性。列在左边的BSD网络版，其所有的网络源代码可以公开得到：包括协议本身以及许多应用程序和工具（如Telnet和FTP）。

在本书中，我们将使用“伯克利派生系统”来指 SunOS 4.x、SVR4以及AIX 3.2等那些基于伯克利源代码开发的系统。这些系统有很多共同之处，经常包含相同的错误。

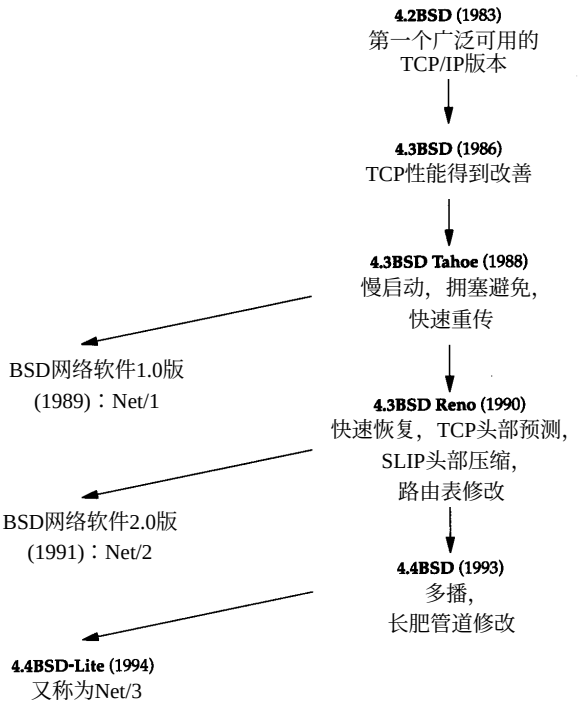


图1-10 不同的BSD版及其重要的TCP/IP特性

起初关于Internet的很多研究现在仍然在伯克利系统中应用 —— 新的拥塞控制算法（21.7节）、多播（12.4节）、“长肥管道”修改（24.3节）以及其他类似的研究。

1.15 应用编程接口

使用TCP/IP协议的应用程序通常采用两种应用编程接口（API）：socket和TLI（运输层接

Linux公社 www.linuxidc.com

口：Transport Layer Interface）。前者有时称作“ Berkeley socket ”，表明它是从伯克利版发展而来的。后者起初是由AT&T开发的，有时称作XTI（X/Open运输层接口），以承认X/Open这个自己定义标准的国际计算机生产商所做的工作。 XTI实际上是TLI的一个超集。

本书不是一本编程方面的书，但是偶尔会引用一些内容来说明 TCP/IP的特性，不管大多数的API（socket）是否提供它们。所有关于 socket和TLI的编程细节请参阅文献 [Stevens 1990]。

1.16 测试网络

图1-11是本书中所有的例子运行的测试网络。为阅读时参考方便，该图还复制在本书扉页前的插页中。

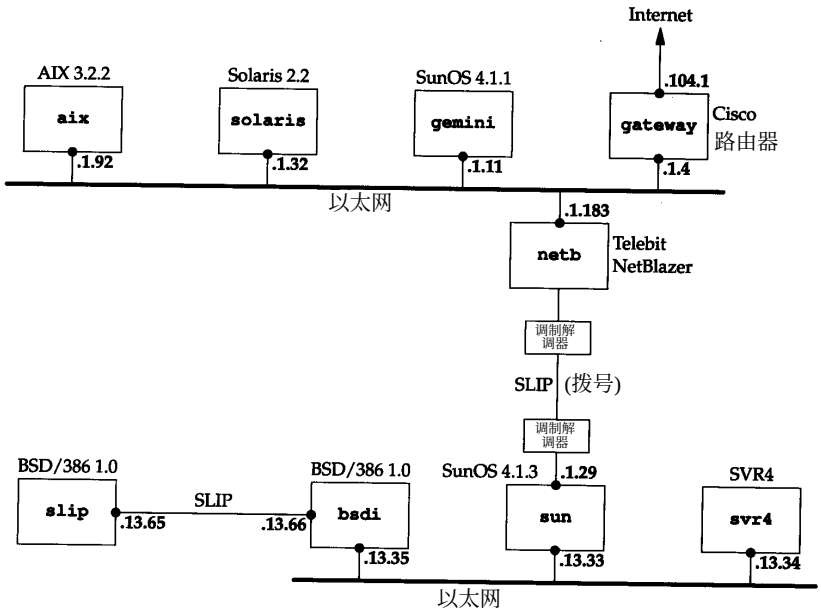


图1-11 本书中所有例子运行的测试网络，所有的IP地址均从140.252开始编址

在这个图中（作者的子网），大多数的例子都运行在下面四个系统中。图中所有的 IP地址属于B类地址，网络号为 140.252。所有的主机名属于 `.tuc.noao.edu`这个域（`noao`代表National Optical Astronomy Observatories，`tuc`代表Tucson）。例如，右下方的系统有一个完整的名字：`svr4.tuc.noao.edu`，其IP地址是：140.252.13.34。每个方框上方的名称是该主机运行的操作系统。这一组系统和网络上的主机及路由器运行于不同的 TCP/IP实现。

需要指出的是，`noao.edu`这个域中的网络和主机要比图 1-11中的多得多。这里列出来的只是本书中将要用到的系统。

在3.4节中，我们将描述这个网络所用到的子网形式。在 4.6节中将介绍`sun`与`netb`之间的拨号SLIP的有关细节。2.4节将详细讨论SLIP。

1.17 小结

本章快速地浏览了TCP/IP协议族，介绍了在后面的章节中将要详细讨论的许多术语和协议。

Linux公社 www.linuxidc.com

TCP/IP协议族分为四层：链路层、网络层、运输层和应用层，每一层各有不同的责任。在TCP/IP中，网络层和运输层之间的区别是最为关键的：网络层（ IP）提供点到点的服务，而运输层（TCP和UDP）提供端到端的服务。

一个互联网是网络的网络。构造互联网的共同基石是路由器，它们在 IP层把网络连在一起。第一个字母大写的Internet是指分布在世界各地的大型互联网，其中包括 1万多个网络和超过100万台主机。

在一个互联网上，每个接口都用 IP地址来标识，尽管用户习惯使用主机名而不是 IP地址。域名系统为主机名和 IP地址之间提供动态的映射。端口号用来标识互相通信的应用程序。服务器使用知名端口号，而客户使用临时设定的端口号。

习题

1.1 请计算最多有多少个A类、B类和C类网络号。

1.2 用匿名FTP（见27.3节）从主机`nic.merit.edu`上获取文件`nsfnet/statistics/history.netcount`。该文件包含在NSFNET网络上登记的国内和国外的网络数。画一坐标系，横坐标代表年，纵坐标代表网络总数的对数值。纵坐标的最大值是习题 1.1的结果。如果数据显示一个明显的趋势，请估计按照当前的编址体制推算，何时会用完所有的网络地址（3.10节讨论解决该难题的建议）。

1.3 获取一份主机需求RFC拷贝[Braden 1989a]，阅读有关应用于TCP/IP协议族每一层的稳健性原则。这个原则的参考对象是什么？

1.4 获取一份最新的赋值RFC拷贝。“ quote of the day ”协议的有名端口号是什么？哪个 RFC对该协议进行了定义？

1.5 如果你有一个接入 TCP/IP互联网的主机帐号，它的主 IP地址是多少？这台主机是否接入了Internet？它是多接口主机吗？

1.6 获取一份RFC 1000的拷贝，了解RFC这个术语从何而来。

1.7 与Internet协会联系，`isoc@isoc.org`或者+1 703 648 9888，了解有关加入的情况。

1.8 用匿名FTP从主机`is.internic.net`处获取文件`about-internic/information-about-the-internic`。

Linux公社 www.linuxidc.com

第2章　链　路　层

2.1　引言

从图1-4中可以看出，在TCP/IP协议族中，链路层主要有三个目的：（1）为IP模块发送和接收IP数据报；（2）为ARP模块发送ARP请求和接收ARP应答；（3）为RARP发送RARP请求和接收RARP应答。TCP/IP支持多种不同的链路层协议，这取决于网络所使用的硬件，如以太网、令牌环网、FDDI（光纤分布式数据接口）及RS-232串行线路等。

在本章中，我们将详细讨论以太网链路层协议，两个串行接口链路层协议（SLIP和PPP），以及大多数实现都包含的环回（loopback）驱动程序。以太网和SLIP是本书中大多数例子使用的链路层。对MTU（最大传输单元）进行了介绍，这个概念在本书的后面章节中将多次遇到。我们还讨论了如何为串行线路选择MTU。

2.2　以太网和IEEE 802封装

以太网这个术语一般是指数字设备公司（Digital Equipment Corp.）、英特尔公司（Intel Corp.）和Xerox公司在1982年联合公布的一个标准。它是当今TCP/IP采用的主要的局域网技术。它采用一种称作CSMA/CD的媒体接入方法，其意思是带冲突检测的载波侦听多路接入（Carrier Sense, Multiple Access with Collision Detection）。它的速率为10 Mb/s，地址为48 bit。

几年后，IEEE（电子电气工程师协会）802委员会公布了一个稍有不同的标准集，其中802.3针对整个CSMA/CD网络，802.4针对令牌总线网络，802.5针对令牌环网络。这三者的共同特性由802.2标准来定义，那就是802网络共有的逻辑链路控制（LLC）。不幸的是，802.2和802.3定义了一个与以太网不同的帧格式。文献[Stallings 1987]对所有的IEEE 802标准进行了详细的介绍。

在TCP/IP世界中，以太网IP数据报的封装是在RFC 894[Hornig 1984]中定义的，IEEE 802网络的IP数据报封装是在RFC 1042[Postel and Reynolds 1988]中定义的。主机需求RFC要求每台Internet主机都与一个10 Mb/s的以太网电缆相连接：

1) 必须能发送和接收采用RFC 894（以太网）封装格式的分组。

2) 应该能接收与RFC 894混合的RFC 1042（IEEE 802）封装格式的分组。

3) 也许能够发送采用RFC 1042格式封装的分组。如果主机能同时发送两种类型的分组数据，那么发送的分组必须是可以设置的，而且默认条件下必须是RFC 894分组。

最常使用的封装格式是RFC 894定义的格式。图2-1显示了两种不同形式的封装格式。图中每个方框下面的数字是它们的字节长度。

两种帧格式都采用48 bit（6字节）的目的地址和源地址（802.3允许使用16 bit的地址，但一般是48 bit地址）。这就是我们在本书中所称的硬件地址。ARP和RARP协议（第4章和第5章）对32 bit的IP地址和48 bit的硬件地址进行映射。

接下来的2个字节在两种帧格式中互不相同。在802标准定义的帧格式中，长度字段是指

Linux公社 www.linuxidc.com

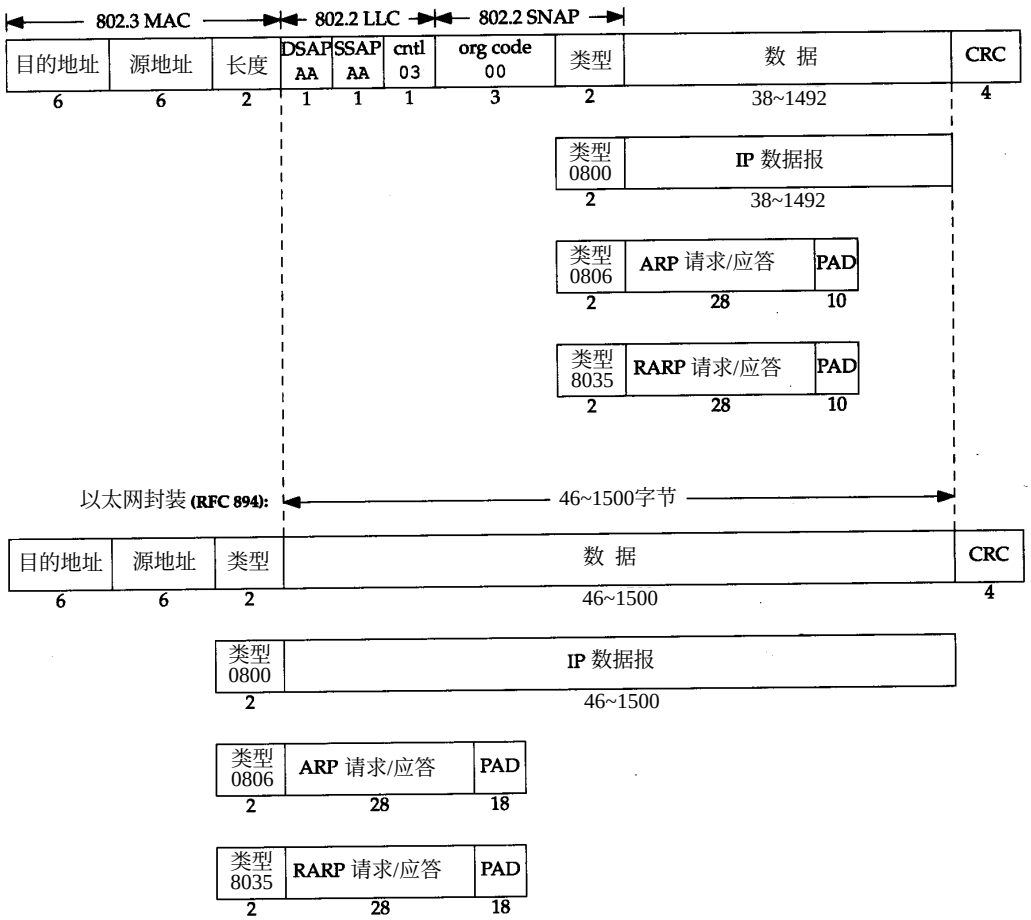


图2-1 IEEE 802.2/802.3（RFC 1042）和以太网的封装格式（RFC 894）

它后续数据的字节长度，但不包括 CRC检验码。以太网的类型字段定义了后续数据的类型。在802标准定义的帧格式中，类型字段则由后续的子网接入协议（ Sub-network Access Protocol，SNAP）的首部给出。幸运的是，802定义的有效长度值与以太网的有效类型值无一相同，这样，就可以对两种帧格式进行区分。

在以太网帧格式中，类型字段之后就是数据；而在 802帧格式中，跟随在后面的是 3字节的802.2 LLC和5字节的802.2 SNAP。目的服务访问点（Destination Service Access Point, DSAP）和源服务访问点（Source Service Access Point, SSAP）的值都设为0xaa。Ctrl字段的值设为3。随后的3个字节org code都置为0。再接下来的2个字节类型字段和以太网帧格式一样（其他类型字段值可以参见RFC 1340 [Reynolds and Postel 1992]）。

CRC字段用于帧内后续字节差错的循环冗余码检验（检验和）（它也被称为FCS或帧检验序列）。

802.3标准定义的帧和以太网的帧都有最小长度要求。802.3规定数据部分必须至少为38字节，而对于以太网，则要求最少要有 46字节。为了保证这一点，必须在不足的空间插入填充（pad）字节。在开始观察线路上的分组时将遇到这种最小长度的情况。

在本书中，我们在需要的时候将给出以太网的封装格式，因为这是最为常见的封装格式。

Linux公社 www.linuxidc.com

2.3 尾部封装

RFC 893[Leffler and Karels 1984]描述了另一种用于以太网的封装格式，称作尾部封装（trailer encapsulation）。这是一个早期BSD系统在DEC VAX机上运行时的试验格式，它通过调整IP数据报中字段的次序来提高性能。在以太网数据帧中，开始的那部分是变长的字段（IP首部和TCP首部）。把它们移到尾部（在CRC之前），这样当把数据复制到内核时，就可以把数据帧中的数据部分映射到一个硬件页面，节省内存到内存的复制过程。 TCP数据报的长度是512字节的整数倍，正好可以用内核中的页表来处理。两台主机通过协商使用 ARP扩展协议对数据帧进行尾部封装。这些数据帧需定义不同的以太网帧类型值。

现在，尾部封装已遭到反对，因此我们不对它举任何例子。有兴趣的读者请参阅 RFC 893以及文献[Leffler et al. 1989]的11.8节。

2.4 SLIP：串行线路IP

SLIP的全称是Serial Line IP。它是一种在串行线路上对IP数据报进行封装的简单形式，在RFC 1055[Romkey 1988]中有详细描述。SLIP适用于家庭中每台计算机几乎都有的 RS-232串行端口和高速调制解调器接入Internet。

下面的规则描述了SLIP协议定义的帧格式：

1) IP数据报以一个称作END（0xc0）的特殊字符结束。同时，为了防止数据报到来之前的线路噪声被当成数据报内容，大多数实现在数据报的开始处也传一个 END字符（如果有线路噪声，那么END字符将结束这份错误的报文。这样当前的报文得以正确地传输，而前一个错误报文交给上层后，会发现其内容毫无意义而被丢弃）。

2) 如果IP报文中某个字符为 END，那么就要连续传输两个字节 0xdb和0xdc来取代它。0xdb这个特殊字符被称作SLIP的ESC字符，但是它的值与ASCII码的ESC字符（0x1b）不同。

3) 如果IP报文中某个字符为SLIP的ESC字符，那么就要连续传输两个字节 0xdb和0xdd来取代它。

图2-2中的例子就是含有一个END字符和一个ESC字符的IP报文。在这个例子中，在串行线路上传输的总字节数是原IP报文长度再加4个字节。

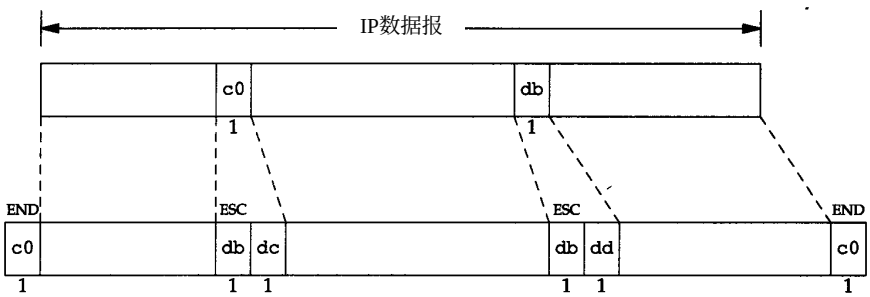


图2-2 SLIP报文的封装

SLIP是一种简单的帧封装方法，还有一些值得一提的缺陷：

1) 每一端必须知道对方的IP地址。没有办法把本端的IP地址通知给另一端。

2) 数据帧中没有类型字段（类似于以太网中的类型字段）。如果一条串行线路用于 SLIP，那么它不能同时使用其他协议。

Linux公社 www.linuxidc.com

3) SLIP没有在数据帧中加上检验和（类似于以太网中的 CRC字段）。如果SLIP传输的报文被线路噪声影响而发生错误，只能通过上层协议来发现（另一种方法是，新型的调制解调器可以检测并纠正错误报文）。这样，上层协议提供某种形式的 CRC就显得很重要。在第 3章和第17章中，我们将看到IP首部和TCP首部及其数据始终都有检验和。在第 11章中，将看到UDP首部及其数据的检验和却是可选的。

尽管存在这些缺点，SLIP仍然是一种广泛使用的协议。

SLIP的历史要追溯到1984年，Rick Adams第一次在4.2BSD系统中实现。尽管它本身的描述是一种非标准的协议，但是随着调制解调器的速率和可靠性的提高，SLIP越来越流行。现在，它的许多产品可以公开获得，而且很多厂家都支持这种协议。

2.5 压缩的SLIP

由于串行线路的速率通常较低（19200 b/s或更低），而且通信经常是交互式的（如 Telnet和Rlogin，二者都使用TCP），因此在SLIP线路上有许多小的TCP分组进行交换。为了传送1个字节的数据需要20个字节的IP首部和20个字节的TCP首部，总数超过40个字节（19.2节描述了Rlogin会话过程中，当敲入一个简单命令时这些小报文传输的详细情况）。

既然承认这些性能上的缺陷，于是人们提出一个被称作 CSLIP（即压缩SLIP）的新协议，它在RFC 1144[Jacobson 1990a]中被详细描述。CSLIP一般能把上面的40个字节压缩到3或5个字节。它能在CSLIP的每一端维持多达16个TCP连接，并且知道其中每个连接的首部中的某些字段一般不会发生变化。对于那些发生变化的字段，大多数只是一些小的数字和的改变。这些被压缩的首部大大地缩短了交互响应时间。

现在大多数的SLIP产品都支持CSLIP。作者所在的子网（参见封面内页）中有两条SLIP链路，它们均是CSLIP链路。

2.6 PPP：点对点协议

PPP，点对点协议修改了SLIP协议中的所有缺陷。PPP包括以下三个部分：

1) 在串行链路上封装 IP数据报的方法。PPP既支持数据为 8位和无奇偶检验的异步模式（如大多数计算机上都普遍存在的串行接口），还支持面向比特的同步链接。

2) 建立、配置及测试数据链路的链路控制协议（LCP：Link Control Protocol）。它允许通信双方进行协商，以确定不同的选项。

3) 针对不同网络层协议的网络控制协议（NCP：Network Control Protocol）体系。当前RFC定义的网络层有IP、OSI网络层、DECnet以及AppleTalk。例如，IP NCP允许双方商定是否对报文首部进行压缩，类似于CSLIP（缩写词NCP也可用在TCP的前面）。

RFC 1548[Simpson 1993]描述了报文封装的方法和链路控制协议。RFC 1332[McGregor 1992]描述了针对IP的网络控制协议。

PPP数据帧的格式看上去很像 ISO的HDLC（高层数据链路控制）标准。图 2-3是PPP数据帧的格式。

每一帧都以标志字符0x7e开始和结束。紧接着是一个地址字节，值始终是 0xff，然后是一个值为0x03的控制字节。

Linux公社 www.linuxidc.com

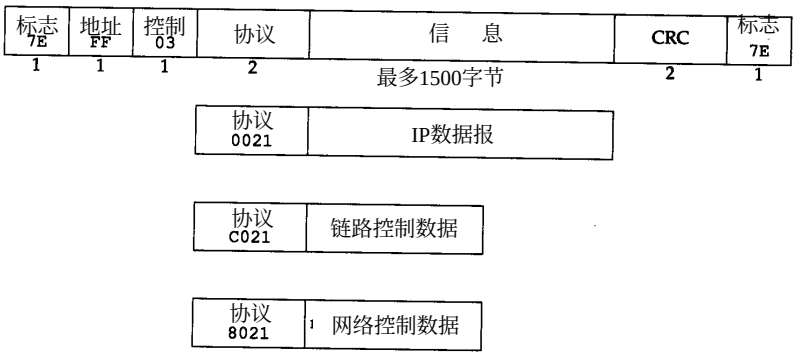


图2-3 PPP数据帧的格式

接下来是协议字段，类似于以太网中类型字段的功能。当它的值为 0x0021时，表示信息字段是一个IP数据报；值为0xc021时，表示信息字段是链路控制数据；值为 0x8021时，表示信息字段是网络控制数据。

CRC字段（或FCS，帧检验序列）是一个循环冗余检验码，以检测数据帧中的错误。

由于标志字符的值是0x7e，因此当该字符出现在信息字段中时， PPP需要对它进行转义。在同步链路中，该过程是通过一种称作比特填充 (bit stuffing)的硬件技术来完成的[Tanenbaum 1989]。在异步链路中，特殊字符 0x7d用作转义字符。当它出现在 PPP数据帧中时，那么紧接着的字符的第6个比特要取其补码，具体实现过程如下：

1) 当遇到字符0x7e时，需连续传送两个字符：0x7d和0x5e，以实现标志字符的转义。

2) 当遇到转义字符0x7d时，需连续传送两个字符：0x7d和0x5d，以实现转义字符的转义。

3) 默认情况下，如果字符的值小于 0x20（比如，一个ASCII控制字符），一般都要进行转义。例如，遇到字符0x01时需连续传送0x7d和0x21两个字符（这时，第6个比特取补码后变为1，而前面两种情况均把它变为0）。

这样做的原因是防止它们出现在双方主机的串行接口驱动程序或调制解调器中，因为有时它们会把这些控制字符解释成特殊的含义。另一种可能是用链路控制协议来指定是否需要对这32个字符中的某一些值进行转义。默认情况下是对所有的 32个字符都进行转义。

与SLIP类似，由于PPP经常用于低速的串行链路，因此减少每一帧的字节数可以降低应用程序的交互时延。利用链路控制协议，大多数的产品通过协商可以省略标志符和地址字段，并且把协议字段由 2个字节减少到 1个字节。如果我们把 PPP的帧格式与前面的 SLIP的帧格式（图2-2）进行比较会发现， PPP只增加了 3个额外的字节：1个字节留给协议字段，另 2个给CRC字段使用。另外，使用IP网络控制协议，大多数的产品可以通过协商采用 Van Jacobson报文首部压缩方法（对应于CSLIP压缩），减小IP和TCP首部长度。

总的来说， PPP比SLIP具有下面这些优点：(1) PPP支持在单根串行线路上运行多种协议，不只是IP协议；(2) 每一帧都有循环冗余检验；(3) 通信双方可以进行IP地址的动态协商(使用IP网络控制协议)；(4) 与CSLIP类似，对TCP和IP报文首部进行压缩；(5) 链路控制协议可以对多个数据链路选项进行设置。为这些优点付出的代价是在每一帧的首部增加 3个字节，当建立链路时要发送几帧协商数据，以及更为复杂的实现。

尽管PPP比SLIP有更多的优点，但是现在的SLIP用户仍然比PPP用户多。随着产品越来越多，产家也开始逐渐支持PPP，因此最终PPP应该取代SLIP。

Linux公社 www.linuxidc.com

2.7 环回接口

大多数的产品都支持环回接口（Loopback Interface），以允许运行在同一台主机上的客户程序和服务器程序通过 TCP/IP进行通信。A类网络号127就是为环回接口预留的。根据惯例，大多数系统把IP地址127.0.0.1分配给这个接口，并命名为localhost。一个传给环回接口的IP数据报不能在任何网络上出现。

我们想象，一旦传输层检测到目的端地址是环回地址时，应该可以省略部分传输层和所有网络层的逻辑操作。但是大多数的产品还是照样完成传输层和网络层的所有过程，只是当IP数据报离开网络层时把它返回给自己。

图2-4是环回接口处理IP数据报的简单过程。

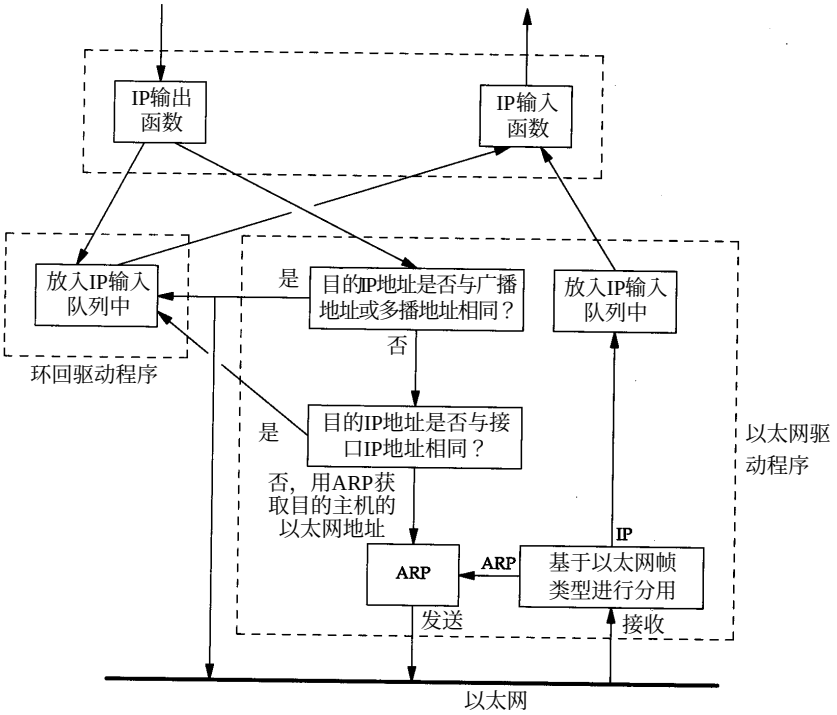


图2-4 环回接口处理IP数据报的过程

图中需要指出的关键点是：

1) 传给环回地址（一般是127.0.0.1）的任何数据均作为IP输入。

2) 传给广播地址或多播地址的数据报复制一份传给环回接口，然后送到以太网上。这是因为广播传送和多播传送的定义（第12章）包含主机本身。

3) 任何传给该主机IP地址的数据均送到环回接口。

看上去用传输层和IP层的方法来处理环回数据似乎效率不高，但它简化了设计，因为环回接口可以被看作是网络层下面的另一个链路层。网络层把一份数据报传送给环回接口，就像传给其他链路层一样，只不过环回接口把它返回到IP的输入队列中。

在图2-4中，另一个隐含的意思是送给主机本身IP地址的IP数据报一般不出现在相应的网络上。例如，在一个以太网上，分组一般不被传出去然后读回来。某些 BSD以太网的设备驱动程序的注释说明，许多以太网接口卡不能读回它们自己发送出去的数据。由于一台主机必

Linux公社 www.linuxidc.com

须处理发送给自己的IP数据报，因此图2-4所示的过程是最为简单的处理办法。

4.4BSD系统定义了变量useloopback，并初始化为1。但是，如果这个变量置为0，以太网驱动程序就会把本地分组送到网络，而不是送到环回接口上。它也许不能工作，这取决于所使用的以太网接口卡和设备驱动程序。

2.8 最大传输单元MTU

正如在图2-1看到的那样，以太网和802.3对数据帧的长度都有一个限制，其最大值分别是1500和1492字节。链路层的这个特性称作MTU，最大传输单元。不同类型的网络大多数都有一个上限。

如果IP层有一个数据报要传，而且数据的长度比链路层的 MTU还大，那么IP层就需要进行分片（fragmentation），把数据报分成若干片，这样每一片都小于 MTU。我们将在11.5节讨论IP分片的过程。

网　　络	MTU字节
超通道	65535
16 Mb/s令牌环(IBM)	17914
4 Mb/s令牌环(IEEE 802.5)	4464
FDDI	4352
以太网	1500
IEEE 802.3/802.2	1492
X.25	576
点对点(低时延)	296

图2-5 几种常见的最大传输单元（MTU）

图2-5列出了一些典型的MTU值，它们摘自RFC 1191[Mogul and Deering 1990]。点到点的链路层（如SLIP和PPP）的MTU并非指的是网络媒体的物理特性。相反，它是一个逻辑限制，目的是为交互使用提供足够快的响应时间。在2.10节中，我们将看到这个限制值是如何计算出来的。

在3.9节中，我们将用netstat命令打印出网络接口的MTU。

2.9 路径MTU

当在同一个网络上的两台主机互相进行通信时，该网络的 MTU是非常重要的。但是如果两台主机之间的通信要通过多个网络，那么每个网络的链路层就可能有不同的 MTU。重要的不是两台主机所在网络的MTU的值，重要的是两台通信主机路径中的最小 MTU。它被称作路径MTU。

两台主机之间的路径MTU不一定是个常数。它取决于当时所选择的路由。而选路不一定是对称的（从A到B的路由可能与从B到A的路由不同），因此路径MTU在两个方向上不一定是一致的。

RFC 1191[Mogul and Deering 1990]描述了路径MTU的发现机制，即在任何时候确定路径MTU的方法。我们在介绍了ICMP和IP分片方法以后再来看它是如何操作的。在 11.6节中，我们将看到ICMP的不可到达错误就采用这种发现方法。在 11.7节中，还会看到，traceroute程序也是用这个方法来确定到达目的节点的路径 MTU。在11.8节和24.2节，将介绍当产品支持路径MTU的发现方法时，UDP和TCP是如何进行操作的。

2.10 串行线路吞吐量计算

如果线路速率是9600 b/s，而一个字节有8 bit，加上一个起始比特和一个停止比特，那么线路的速率就是960 B/s（字节/秒）。以这个速率传输一个1024字节的分组需要1066 ms。如果

Linux公社 www.linuxidc.com

用SLIP链接运行一个交互式应用程序，同时还运行另一个应用程序如 FTP发送或接收1024字节的数据，那么一般来说就必须等待一半的时间（ 533 ms）才能把交互式应用程序的分组数据发送出去。

假定交互分组数据可以在其他“大块”分组数据发送之前被发送出去。大多数的 SLIP实现确实提供这类服务排队方法，把交互数据放在大块的数据前面。交互通信一般有 Telnet、Rlogin以及FTP的控制部分（用户的命令，而不是数据）。

这种服务排队方法是不完善的。它不能影响已经进入下游（如串行驱动程序）队列的非交互数据。同时，新型的调制解调器具有很大的缓冲区，因此非交互数据可能已经进入该缓冲区了。

对于交互应用来说，等待 533 ms是不能接受的。关于人的有关研究表明，交互响应时间超过100～200 ms就被认为是不好的[Jacobson 1990a]。这是发送一份交互报文出去后，直到接收到响应信息（通常是出现一个回显字符）为止的往返时间。

把SLIP的MTU缩短到256就意味着链路传输一帧最长需要 266 ms，它的一半是133 ms（这是一般需要等待的时间）。这样情况会好一些，但仍然不完美。我们选择它的原因（与 64或128相比）是因为大块数据提供良好的线路利用率（如大文件传输）。假设CSLIP的报文首部是5个字节，数据帧总长为261个字节，256个字节的数据使线路的利用率为98.1%，帧头占了1.9%，这样的利用率是很不错的。如果把 MTU降到256以下，那么将降低传输大块数据的最大吞吐量。

在图2-5列出的MTU值中，点对点链路的MTU是296个字节。假设数据为256字节，TCP和IP首部占40个字节。由于MTU是IP向链路层查询的结果，因此该值必须包括通常的 TCP和IP首部。这样就会导致IP如何进行分片的决策。IP对于CSLIP的压缩情况一无所知。

我们对平均等待时间的计算（传输最大数据帧所需时间的一半）只适用于 SLIP链路（或PPP链路）在交互通信和大块数据传输这两种情况下。当只有交互通信时，如果线路速率是9600 b/s，那么任何方向上的1字节数据（假设有5个字节的压缩帧头）往返一次都大约需要12.5 ms。它比前面提到的100~200 ms要小得多。需要注意的是，由于帧头从40个字节压缩到5个字节，使得1字节数据往返时间从85 ms减到12.5 ms。

不幸的是，当使用新型的纠错和压缩调制解调器时，这样的计算就更难了。这些调制解调器所采用的压缩方法使得在线路上传输的字节数大大减少，但纠错机制又会增加传输的时间。不过，这些计算是我们进行合理决策的入口点。

在后面的章节中，我们将用这些串行线路吞吐量的计算来验证数据从串行线路上通过的时间。

2.11 小结

本章讨论了Internet协议族中的最底层协议，链路层协议。我们比较了以太网和 IEEE 802.2/802.3的封装格式，以及SLIP和PPP的封装格式。由于SLIP和PPP经常用于低速的链路，二者都提供了压缩不常变化的公共字段的方法。这使交互性能得到提高。

大多数的实现都提供环回接口。访问这个接口可以通过特殊的环回地址，一般为127.0.0.1。也可以通过发送IP数据报给主机所拥有的任一IP地址。当环回数据回到上层的协议栈中时，它已经过传输层和IP层完整的处理过程。

Linux公社 www.linuxidc.com

我们描述了很多链路都具有的一个重要特性， MTU，相关的一个概念是路径 MTU。根据典型的串行线路MTU，对SLIP和CSLIP链路的传输时延进行了计算。

本章的内容只覆盖了当今 TCP/IP所采用的部分数据链路公共技术。 TCP/IP成功的原因之一是它几乎能在任何数据链路技术上运行。

习题

2.1 如果你的系统支持 `netstat`(1)命令（参见3.9节），那么请用它确定系统上的接口及其MTU。

Linux公社 www.linuxidc.com

第3章　IP：网际协议

3.1　引言

IP是TCP/IP协议族中最为核心的协议。所有的TCP、UDP、ICMP及IGMP数据都以IP数据报格式传输（见图1-4）。许多刚开始接触TCP/IP的人对IP提供不可靠、无连接的数据报传送服务感到很奇怪，特别是那些具有X.25或SNA背景知识的人。

不可靠（unreliable）的意思是它不能保证IP数据报能成功地到达目的地。IP仅提供最好的传输服务。如果发生某种错误时，如某个路由器暂时用完了缓冲区，IP有一个简单的错误处理算法：丢弃该数据报，然后发送ICMP消息报给信源端。任何要求的可靠性必须由上层来提供（如TCP）。

无连接（connectionless）这个术语的意思是IP并不维护任何关于后续数据报的状态信息。每个数据报的处理是相互独立的。这也说明，IP数据报可以不按发送顺序接收。如果一信源向相同的信宿发送两个连续的数据报（先是A，然后是B），每个数据报都是独立地进行路由选择，可能选择不同的路线，因此B可能在A到达之前先到达。

在本章，我们将简要介绍IP首部中的各个字段，讨论IP路由选择和子网的有关内容。还要介绍两个有用的命令：`ifconfig`和`netstat`。关于IP首部中一些字段的细节，将留在以后使用这些字段的时候再进行讨论。RFC 791[Postel 1981a]是IP的正式规范文件。

3.2　IP首部

IP数据报的格式如图3-1所示。普通的IP首部长为20个字节，除非含有选项字段。

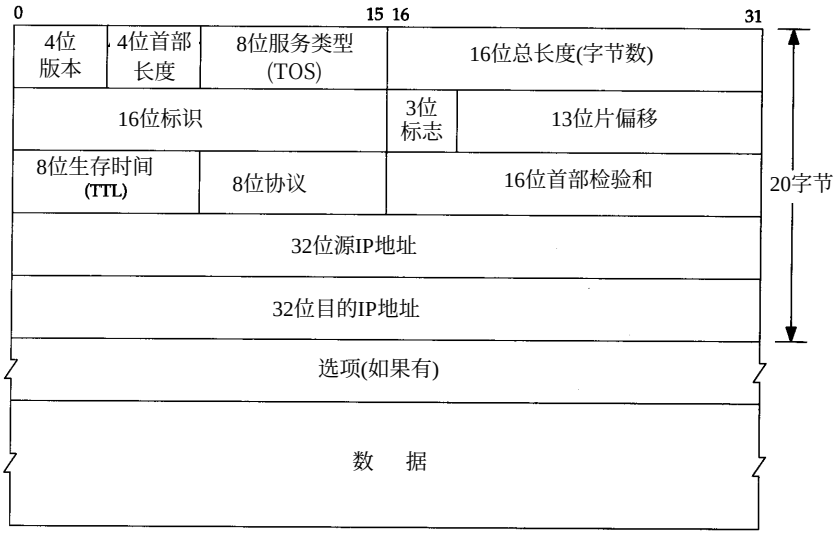


图3-1　IP数据报格式及首部中的各字段

Linux公社 www.linuxidc.com

分析图3-1中的首部。最高位在左边，记为0 bit；最低位在右边，记为31 bit。

4个字节的32 bit值以下面的次序传输：首先是0～7 bit，其次8～15 bit，然后16～23 bit，最后是24~31 bit。这种传输次序称作big endian字节序。由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。以其他形式存储二进制整数的机器，如little endian格式，则必须在传输数据之前把首部转换成网络字节序。

目前的协议版本号是4，因此IP有时也称作IPv4。3.10节将对一种新版的IP协议进行讨论。

首部长度指的是首部占32 bit字的数目，包括任何选项。由于它是一个 4比特字段，因此首部最长为60个字节。在第8章中，我们将看到这种限制使某些选项如路由记录选项在当今已没有什么用处。普通IP数据报（没有任何选择项）字段的值是5。

服务类型（TOS）字段包括一个3 bit的优先权子字段（现在已被忽略），4 bit的TOS子字段和1 bit未用位但必须置0。4 bit的TOS分别代表：最小时延、最大吞吐量、最高可靠性和最小费用。4 bit中只能置其中1 bit。如果所有4 bit均为0，那么就意味着是一般服务。RFC 1340 [Reynolds and Postel 1992]描述了所有的标准应用如何设置这些服务类型。 RFC 1349 [Almquist 1992]对该RFC进行了修正，更为详细地描述了TOS的特性。

图3-2列出了对不同应用建议的TOS值。在最后一列中给出的是十六进制值，因为这就是在后面将要看到的tcpdump命令输出。

应 用 程 序	最小时延	最大吞吐量	最高可靠性	最小费用	16进制值
Telnet/Rlogin	1	0	0	0	0x10
FTP					
控制	1	0	0	0	0x10
数据	0	1	0	0	0x08
任意块数据	0	1	0	0	0x08
TFTP	1	0	0	0	0x10
SMTP					
命令阶段	1	0	0	0	0x10
数据阶段	0	1	0	0	0x08
DNS					
UDP查询	1	0	0	0	0x10
TCP查询	0	0	0	0	0x00
区域传输	0	1	0	0	0x08
ICMP					
差错	0	0	0	0	0x00
查询	0	0	0	0	0x00
任何IGP	0	0	1	0	0x04
SNMP	0	0	1	0	0x04
BOOTP	0	0	0	0	0x00
NNTP	0	0	0	1	0x02

图3-2 服务类型字段推荐值

Telnet和Rlogin这两个交互应用要求最小的传输时延，因为人们主要用它们来传输少量的交互数据。另一方面， FTP文件传输则要求有最大的吞吐量。最高可靠性被指明给网络管理（SNMP）和路由选择协议。用户网络新闻（ Usenet news, NNTP）是唯一要求最小费用的应用。

现在大多数的TCP/IP实现都不支持TOS特性，但是自4.3BSD Reno以后的新版系统都对它进行了设置。另外，新的路由协议如OSPF和IS-IS都能根据这些字段的值进行路由决策。

在2.10节中，我们提到SLIP一般提供基于服务类型的排队方法，允许对交互通信

Linux公社 www.linuxidc.com

数据在处理大块数据之前进行处理。由于大多数的实现都不使用 TOS字段，因此这种排队机制由SLIP自己来判断和处理，驱动程序先查看协议字段（确定是否是一个 TCP段），然后检查TCP信源和信宿的端口号，以判断是否是一个交互服务。一个驱动程序的注释这样认为，这种“令人厌恶的处理方法”是必需的，因为大多数实现都不允许应用程序设置TOS字段。

总长度字段是指整个IP数据报的长度，以字节为单位。利用首部长度字段和总长度字段，就可以知道IP数据报中数据内容的起始位置和长度。由于该字段长 16比特，所以IP数据报最长可达65535字节（回忆图 2-5，超级通道的 MTU为65535。它的意思其实不是一个真正的MTU —— 它使用了最长的IP数据报）。当数据报被分片时，该字段的值也随着变化，这一点将在11.5节中进一步描述。

尽管可以传送一个长达 65535字节的IP数据报，但是大多数的链路层都会对它进行分片。而且，主机也要求不能接收超过 576字节的数据报。由于 TCP把用户数据分成若干片，因此一般来说这个限制不会影响 TCP。在后面的章节中将遇到大量使用 UDP的应用（RIP，TFTP，BOOTP，DNS，以及SNMP），它们都限制用户数据报长度为 512字节，小于576字节。但是，事实上现在大多数的实现（特别是那些支持网络文件系统 NFS的实现）允许超过 8192字节的IP数据报。

总长度字段是IP首部中必要的内容，因为一些数据链路（如以太网）需要填充一些数据以达到最小长度。尽管以太网的最小帧长为 46字节（见图2-1），但是IP数据可能会更短。如果没有总长度字段，那么IP层就不知道46字节中有多少是IP数据报的内容。

标识字段唯一地标识主机发送的每一份数据报。通常每发送一份报文它的值就会加 1。在11.5节介绍分片和重组时再详细讨论它。同样，在讨论分片时再来分析标志字段和片偏移字段。

RFC 791 [Postel 1981a]认为标识字段应该由让IP发送数据报的上层来选择。假设有两个连续的IP数据报，其中一个是由TCP生成的，而另一个是由UDP生成的，那么它们可能具有相同的标识字段。尽管这也可以照常工作（由重组算法来处理），但是在大多数从伯克利派生出来的系统中，每发送一个IP数据报，IP层都要把一个内核变量的值加1，不管交给IP的数据来自哪一层。内核变量的初始值根据系统引导时的时间来设置。

TTL（time-to-live）生存时间字段设置了数据报可以经过的最多路由器数。它指定了数据报的生存时间。TTL的初始值由源主机设置（通常为32或64），一旦经过一个处理它的路由器，它的值就减去1。当该字段的值为0时，数据报就被丢弃，并发送 ICMP报文通知源主机。第8章我们讨论Traceroute程序时将再回来讨论该字段。

我们已经在第1章讨论了协议字段，并在图 1-8中示出了它如何被 IP用来对数据报进行分用。根据它可以识别是哪个协议向 IP传送数据。

首部检验和字段是根据IP首部计算的检验和码。它不对首部后面的数据进行计算。 ICMP、IGMP、UDP和TCP在它们各自的首部中均含有同时覆盖首部和数据检验和码。

为了计算一份数据报的 IP检验和，首先把检验和字段置为 0。然后，对首部中每个 16 bit进行二进制反码求和（整个首部看成是由一串 16 bit的字组成），结果存在检验和字段中。当收到一份IP数据报后，同样对首部中每个 16 bit进行二进制反码的求和。由于接收方在计算过

Linux公社 www.linuxidc.com

程中包含了发送方存在首部中的检验和，因此，如果首部在传输过程中没有发生任何差错，那么接收方计算的结果应该为全1。如果结果不是全1（即检验和错误），那么IP就丢弃收到的数据报。但是不生成差错报文，由上层去发现丢失的数据报并进行重传。

ICMP、IGMP、UDP和TCP都采用相同的检验和算法，尽管TCP和UDP除了本身的首部和数据外，在IP首部中还包含不同的字段。在RFC 1071[Braden, Borman and Patridge 1988]中有关于如何计算Internet检验和的实现技术。由于路由器经常只修改TTL字段（减1），因此当路由器转发一份报文时可以增加它的检验和，而不需要对IP整个首部进行重新计算。RFC 1141[Mallory and Kullberg 1990]为此给出了一个很有效的方法。

但是，标准的BSD实现在转发数据报时并不是采用这种增加的办法。

每一份IP数据报都包含源IP地址和目的IP地址。我们在1.4节中说过，它们都是32 bit的值。

最后一个字段是任选项，是数据报中的一个可变长的可选信息。目前，这些任选项定义如下：

- 安全和处理限制（用于军事领域，详细内容参见RFC 1108[Kent 1991]）
- 记录路径（让每个路由器都记下它的IP地址，见7.3节）
- 时间戳（让每个路由器都记下它的IP地址和时间，见7.4节）
- 宽松的源站选路（为数据报指定一系列必须经过的IP地址，见8.5节）
- 严格的源站选路（与宽松的源站选路类似，但是要求只能经过指定的这些地址，不能经过其他的地址）。

这些选项很少被使用，并非所有的主机和路由器都支持这些选项。

选项字段一直都是以32 bit作为界限，在必要的时候插入值为0的填充字节。这样就保证IP首部始终是32 bit的整数倍（这是首部长度字段所要求的）。

3.3 IP路由选择

从概念上说，IP路由选择是简单的，特别对于主机来说。如果目的主机与源主机直接相连（如点对点链路）或都在一个共享网络上（以太网或令牌环网），那么IP数据报就直接送到目的主机上。否则，主机把数据报发往一默认的路由器上，由路由器来转发该数据报。大多数的主机都是采用这种简单机制。

在本节和第9章中，我们将讨论更一般的情况，即IP层既可以配置成路由器的功能，也可以配置成主机的功能。当今的大多数多用户系统，包括几乎所有的Unix系统，都可以配置成一个路由器。我们可以为它指定主机和路由器都可以使用的简单路由算法。本质上的区别在于主机从不把数据报从一个接口转发到另一个接口，而路由器则要转发数据报。内含路由器功能的主机应该从不转发数据报，除非它被设置成那样。在9.4小节中，我们将进一步讨论配置的有关问题。

在一般的体制中，IP可以从TCP、UDP、ICMP和IGMP接收数据报（即在本地生成的数据报）并进行发送，或者从一个网络接口接收数据报（待转发的数据报）并进行发送。IP层在内存中有一个路由表。当收到一份数据报并进行发送时，它都要对该表搜索一次。当数据报来自某个网络接口时，IP首先检查目的IP地址是否为本机的IP地址之一或者IP广播地址。如果确实是这样，数据报就被送到由IP首部协议字段所指定的协议模块进行处理。如果数据报的

Linux公社 www.linuxidc.com

目的不是这些地址，那么（1）如果IP层被设置为路由器的功能，那么就对数据报进行转发（也就是说，像下面对待发出的数据报一样处理）；否则（2）数据报被丢弃。

路由表中的每一项都包含下面这些信息：

- 目的IP地址。它既可以是一个完整的主机地址，也可以是一个网络地址，由该表目中的标志字段来指定（如下所述）。主机地址有一个非0的主机号（见图1-5），以指定某一特定的主机，而网络地址中的主机号为0，以指定网络中的所有主机（如以太网，令牌环网）。
- 下一站（或下一跳）路由器（next-hop router）的IP地址，或者有直接连接的网络IP地址。下一站路由器是指一个在直接相连网络上的路由器，通过它可以转发数据报。下一站路由器不是最终的目的，但是它可以把传送给它的数据报转发到最终目的。
- 标志。其中一个标志指明目的IP地址是网络地址还是主机地址，另一个标志指明下一站路由器是否为真正的下一站路由器，还是一个直接相连的接口（我们将在9.2节中详细介绍这些标志）。
- 为数据报的传输指定一个网络接口。

IP路由选择是逐跳地（hop-by-hop）进行的。从这个路由表信息可以看出，IP并不知道到达任何目的的完整路径（当然，除了那些与主机直接相连的目的）。所有的IP路由选择只为数据报传输提供下一站路由器的IP地址。它假定下一站路由器比发送数据报的主机更接近目的，而且下一站路由器与该主机是直接相连的。

IP路由选择主要完成以下这些功能：

1) 搜索路由表，寻找能与目的IP地址完全匹配的表目（网络号和主机号都要匹配）。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。
2) 搜索路由表，寻找能与目的网络号相匹配的表目。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。目的网络上的所有主机都可以通过这个表目来处置。例如，一个以太网上的所有主机都是通过这种表目进行寻径的。

   这种搜索网络的匹配方法必须考虑可能的子网掩码。关于这一点我们在下一节中进行讨论。
3) 搜索路由表，寻找标为“默认（default）”的表目。如果找到，则把报文发送给该表目指定的下一站路由器。

如果上面这些步骤都没有成功，那么该数据报就不能被传送。如果不能传送的数据报来自本机，那么一般会向生成数据报的应用程序返回一个“主机不可达”或“网络不可达”的错误。

完整主机地址匹配在网络号匹配之前执行。只有当它们都失败后才选择默认路由。默认路由，以及下一站路由器发送的ICMP间接报文（如果我们为数据报选择了错误的默认路由），是IP路由选择机制中功能强大的特性。我们在第9章对它们进行讨论。

为一个网络指定一个路由器，而不必为每个主机指定一个路由器，这是IP路由选择机制的另一个基本特性。这样做可以极大地缩小路由表的规模，比如Internet上的路由器有只有几千个表目，而不会是超过100万个表目。

举例

首先考虑一个简单的例子：我们的主机bsdi有一个IP数据报要发送给主机sun。双方都在

Linux公社 www.linuxidc.com

同一个以太网上（参见扉页前图）。数据报的传输过程如图3-3所示。

当IP从某个上层收到这份数据报后，它搜索路由表，发现目的 IP地址（140.252.13.33）在一个直接相连的网络上（以太网 140.252.13.0）。于是，在表中找到匹配网络地址（在下一节中，我们将看到，由于以太网的子网掩码的存在，实际的网络地址是 140.252.13.32，但是这并不影响这里所讨论的路由选择）。

数据报被送到以太网驱动程序，然后作为一个以太网数据帧被送到 sun主机上（见图 2-1）。IP数据报中的目的地址是sun的IP地址（140.252.13.33），而在链路层首部中的目的地址是 48 bit的sun主机的以太网接口地址。这个 48 bit的以太网地址是用 ARP协议获得的，我们将在下一章对此进行描述。

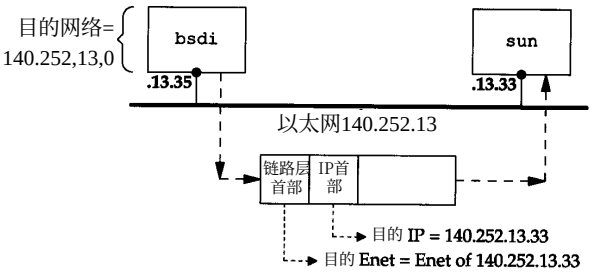


图3-3 数据报从主机bsdi到sun的传送过程

现在来看另一个例子：主机 bsdi有一份IP数据报要传到ftp.uu.net主机上，它的IP地址是192.48.96.9。经过的前三个路由器如图3-4所示。首先，主机bsdi搜索路由表，但是没有找到与主机地址或网络地址相匹配的表目，因此只能用默认的表目，把数据报传给下一站路由器，即主机 sun。当数据报从bsdi被传到sun主机上以后，目的IP地址是最终的信宿机地址（192.48.96.9），但是链路层地址却是sun主机的以太网接口地址。这与图 3-3不同，在那里数据报中的目的IP地址和目的链路层地址都指的是相同的主机（ sun）。

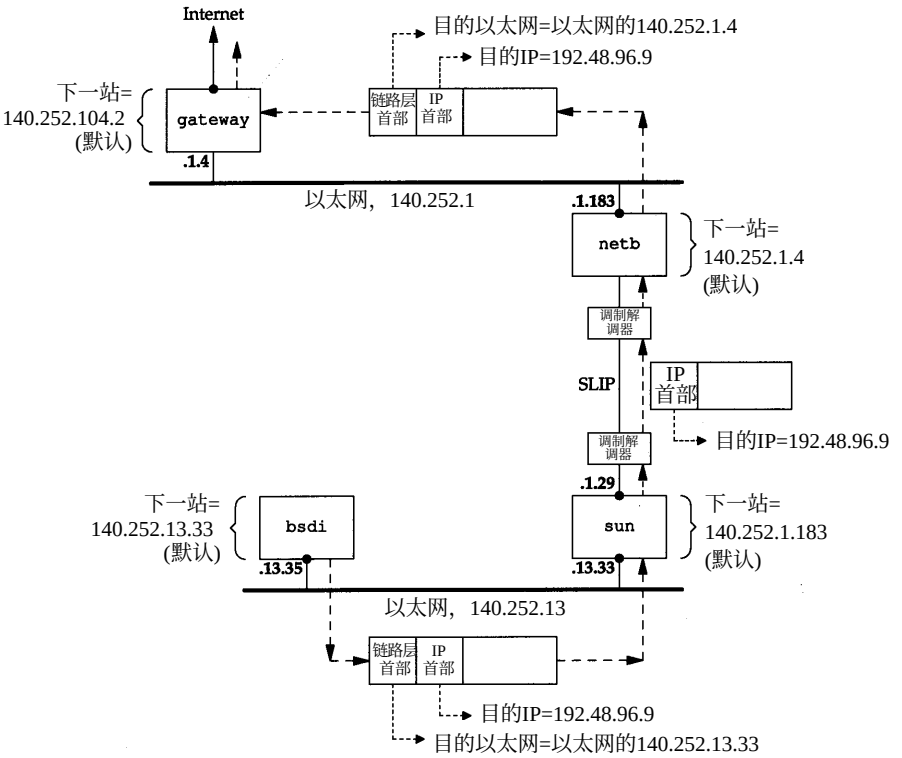


图3-4 从bsdi 到ftp.uu.net (192.48.96.9)的初始路径

Linux公社 www.linuxidc.com

当sun收到数据报后，它发现数据报的目的IP地址并不是本机的任一地址，而sun已被设置成具有路由器的功能，因此它把数据报进行转发。经过搜索路由表，选用了默认表目。根据sun的默认表目，它把数据报转发到下一站路由器netb，该路由器的地址是140.252.1.183。数据报是经过点对点SLIP链路被传送的，采用了图2-2所示的最小封装格式。这里，我们没有给出像以太网链路层数据帧那样的首部，因为在SLIP链路中没有那样的首部。

当netb收到数据报后，它执行与sun主机相同的步骤：数据报的目的地址不是本机地址，而netb也被设置成具有路由器的功能，于是它也对该数据报进行转发。采用的也是默认路由表目，把数据报送到下一站路由器gateway（140.252.1.4）。位于以太网140.252.1上的主机netb用ARP获得对应于140.252.1.4的48 bit以太网地址。这个以太网地址就是链路层数据帧头上的目的地址。

路由器gateway也执行与前面两个路由器相同的步骤。它的默认路由表目所指定的下一站路由器IP地址是140.252.104.2（我们将在图8-4中证实，使用Traceroute程序时，它就是gateway使用的下一站路由器）。

对于这个例子需要指出一些关键点：

1) 该例子中的所有主机和路由器都使用了默认路由。事实上，大多数主机和一些路由器可以用默认路由来处理任何目的，除非它在本地局域网上。

2) 数据报中的目的IP地址始终不发生任何变化（在8.5节中，我们将看到，只有使用源路由选项时，目的IP地址才有可能被修改，但这种情况很少出现）。所有的路由选择决策都是基于这个目的IP地址。

3) 每个链路层可能具有不同的数据帧首部，而且链路层的目的地址（如果有的话）始终指的是下一站的链路层地址。在例子中，两个以太网封装了含有下一站以太网地址的链路层首部，但是SLIP链路没有这样做。以太网地址一般通过ARP获得。

在第9章，我们在描述了ICMP之后将再次讨论IP路由选择问题。我们将看到一些路由表的例子，以及如何用它们来进行路由决策的。

3.4 子网寻址

现在所有的主机都要求支持子网编址（RFC 950 [Mogul and Postel 1985]）。不是把IP地址看成由单纯的一个网络号和一个主机号组成，而是把主机号再分成一个子网号和一个主机号。

这样做的原因是因为A类和B类地址为主机号分配了太多的空间，可分别容纳的主机数为$2^{24}-2$和$2^{16}-2$。事实上，在一个网络中人们并不安排这么多的主机（各类IP地址的格式如图1-5所示）。由于全0或全1的主机号都是无效的，因此我们把总数减去2。

在InterNIC获得某类IP网络号后，就由当地的系统管理员来进行分配，由他（或她）来决定是否建立子网，以及分配多少比特给子网号和主机号。例如，这里有一个B类网络地址（140.252），在剩下的16 bit中，8 bit用于子网号，8 bit用于主机号，格式如图3-5所示。这样就允许有254个子网，每个子网可以有254台主机。

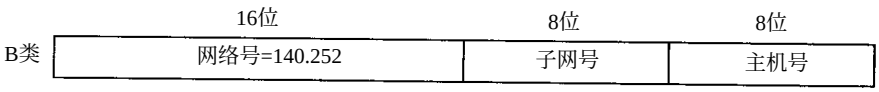


图3-5 B类地址的一种子网编址

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

许多管理员采用自然的划分方法，即把 B类地址中留给主机的16 bit中的前8 bit作为子网地址，后8bit作为主机号。这样用点分十进制方法表示的 IP地址就可以比较容易确定子网号。但是，并不要求A类或B类地址的子网划分都要以字节为划分界限。

大多数的子网例子都是B类地址。其实，子网还可用于C类地址，只是它可用的比特数较少而已。很少出现A类地址的子网例子是因为A类地址本身就很少（但是，大多数 A类地址都是进行子网划分的）。

子网对外部路由器来说隐藏了内部网络组织（一个校园或公司内部）的细节。在我们的网络例子中，所有的IP地址都有一个B类网络号140.252。但是其中有超过30个子网，多于400台主机分布在这些子网中。由一台路由器提供了 Internet的接入，如图3-6所示。

在这个图中，我们把大多数的路由器编号为 R*n*，*n*是子网号。我们给出了连接这些子网的路由器，同时还包括了扉页前图中的九个系统。在图中，以太网用粗线表示，点对点链路用虚线表示。我们没有画出不同子网中的所有主机。例如，在子网 140.252.3上，就超过50台主机，而在子网140.252.1上则超过100台主机。

与30个C类地址相比，用一个包含30个子网的B类地址的好处是，它可以缩小Internet路由表的规模。B类地址140.252被划分为若干子网的事实对于所有子网以外的 Internet路由器都是透明的。为了到达 IP地址开始部分为 140.252的主机，外部路由器只需要知道通往 IP地址140.252.104.1的路径。这就是说，对于网络 140.252只需一个路由表目，而如果采用 30个C类地址，则需要30个路由表目。因此，子网划分缩减了路由表的规模（在 10.8小节中，我们将介绍一种新技术，即使用C类地址也可以缩减路由表的规模）。

子网对于子网内部的路由器是不透明的。如图 3-6所示，一份来自 Internet的数据报到达 `gateway`，它的目的地址是140.252.57.1。路由器`gateway`需要知道子网号是57，然后把它送到`kpno`。同样，`kpno`必须把数据报送到R55，最后由R55把它送到R57。

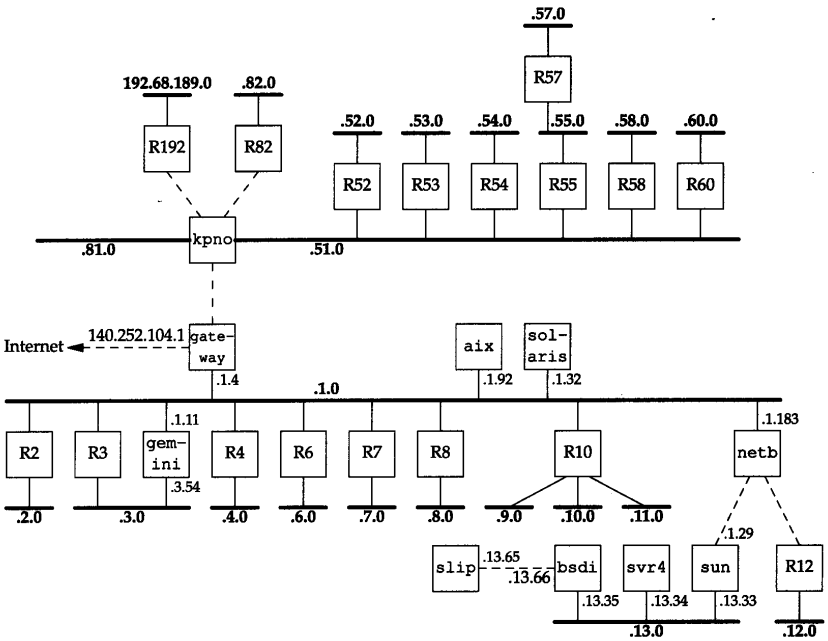


图3-6 网络noao.edu（140.252）中的大多数子网安排

Linux公社 www.linuxidc.com

3.5 子网掩码

任何主机在引导时进行的部分配置是指定主机 IP地址。大多数系统把 IP地址存在一个磁盘文件里供引导时读用。在第5章我们将讨论一个无盘系统如何在引导时获得 IP地址。

除了IP地址以外，主机还需要知道有多少比特用于子网号及多少比特用于主机号。这是在引导过程中通过子网掩码来确定的。这个掩码是一个 32 bit的值，其中值为1的比特留给网络号和子网号，为0的比特留给主机号。图3-7是一个B类地址的两种不同的子网掩码格式。第一个例子是noao.edu网络采用的子网划分方法，如图3-5所示，子网号和主机号都是8 bit宽。第二个例子是一个B类地址划分成10 bit的子网号和6 bit的主机号。

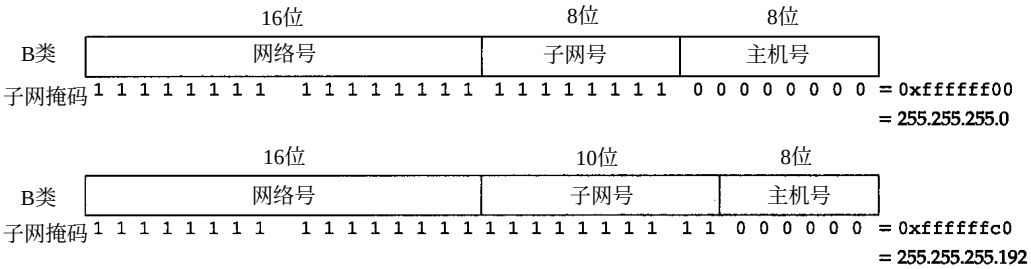


图3-7 两种不同的B类地址子网掩码的例子

尽管IP地址一般以点分十进制方法表示，但是子网掩码却经常用十六进制来表示，特别是当界限不是一个字节时，因为子网掩码是一个比特掩码。

给定IP地址和子网掩码以后，主机就可以确定 IP数据报的目的是：(1)本子网上的主机；(2)本网络中其他子网中的主机；(3)其他网络上的主机。如果知道本机的 IP地址，那么就知道它是否为A类、B类或C类地址(从IP地址的高位可以得知)，也就知道网络号和子网号之间的分界线。而根据子网掩码就可知道子网号与主机号之间的分界线。

举例

假设我们的主机地址是140.252.1.1（一个B类地址），而子网掩码为255.255.255.0（其中8 bit为子网号，8 bit为主机号）。

- 如果目的IP地址是140.252.4.5，那么我们就知道B类网络号是相同的（140.252），但是子网号是不同的（1和4）。用子网掩码在两个IP地址之间的比较如图3-8所示。
- 如果目的IP地址是140.252.1.22，那么B类网络号还是一样的（140.252），而且子网号也是一样的（1），但是主机号是不同的。
- 如果目的IP地址是192.43.235.6（一个C类地址），那么网络号是不同的，因而进一步的比较就不用再进行了。

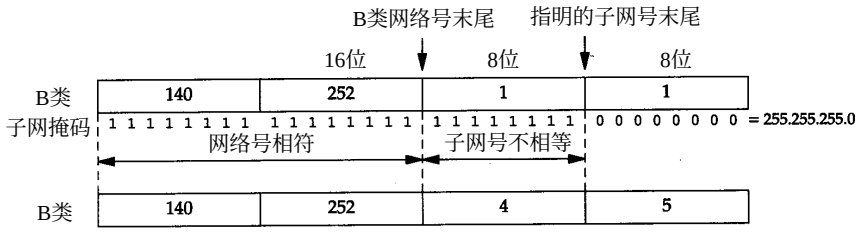


图3-8 使用子网掩码的两个B类地址之间的比较

Linux公社 www.linuxidc.com

给定两个IP地址和子网掩码后，IP路由选择功能一直进行这样的比较。

3.6 特殊情况的IP地址

经过子网划分的描述，现在介绍7个特殊的IP地址，如图3-9所示。在这个图中，0表示所有的比特位全为0；-1表示所有的比特位全为1；netid、subnetid和hostid分别表示不为全0或全1的对应字段。子网号栏为空表示该地址没有进行子网划分。

IP 地 址			可 以 为		描 述
网络号	子网号	主机号	源 端	目的端	
0		0	OK	不可能	网络上的主机（参见下面的限制）
0		主机号	OK	不可能	网络上的特定主机（参见下面的限制）
127		任何值	OK	OK	环回地址（2.7节）
-1		-1	不可能	OK	受限的广播（永远不被转发）
netid		-1	不可能	OK	以网络为目的向*netid*广播
netid	*subnetid*	-1	不可能	OK	以子网为目的向*netid*、*subnetid*广播
netid	-1	-1	不可能	OK	以所有子网为目的向*netid*广播

图3-9 特殊情况的IP地址

我们把这个表分成三个部分。表的头两项是特殊的源地址，中间项是特殊的环回地址，最后四项是广播地址。

表中的头两项，网络号为0，如主机使用BOOTP协议确定本机IP地址时只能作为初始化过程中的源地址出现。

在12.2节中，我们将进一步分析四类广播地址。

3.7 一个子网的例子

这个例子是本文中采用的子网，以及如何使用两个不同的子网掩码。具体安排如图 3-10所示。

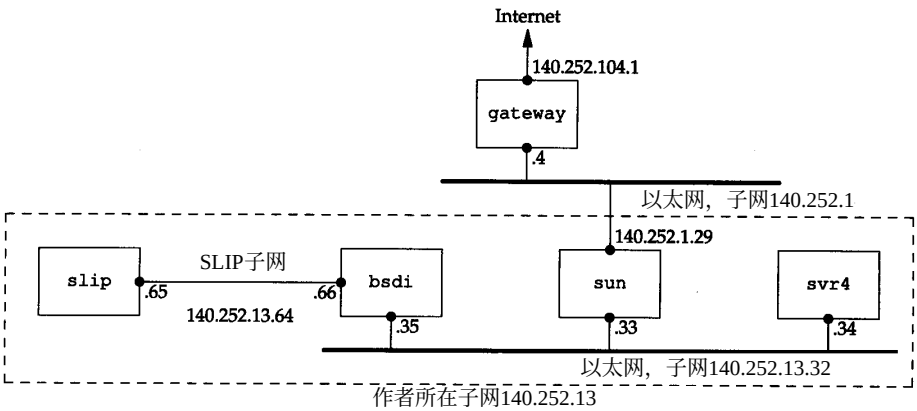


图3-10 作者所在子网中的主机和网络安排

如果把该图与扉页前图相比，就会发现在图 3-10中省略了从路由器sun到上面的以太网之间的连接细节，实际上它们之间的连接是拨号 SLIP。这个细节不影响本节中讨论的子网划分

Linux公社 www.linuxidc.com

问题。我们在4.6节讨论ARP代理时将再回头讨论这个细节。

问题是我们在子网13中有两个分离的网络：一个以太网和一个点对点链路（硬件连接的SLIP链路）（点对点链接始终会带来问题，因为它一般在两端都需要 IP地址）。将来或许会有更多的主机和网络，但是为了不让主机跨越不同的网络就得使用不同的子网号。我们的解决方法是把子网号从 8 bit扩充到11bit，把主机号从 8 bit减为5 bit。这就叫作变长子网，因为140.252网络中的大多数子网都采用8 bit子网掩码，而我们的子网却采用11 bit的子网掩码。

> RFC 1009[Braden and Postel 1987]允许一个含有子网的网络使用多个子网掩码。新的路由器需求RFC[Almquist 1993]则要求支持这一功能。
>
> 但是，问题在于并不是所有的路由选择协议在交换目的网络时也交换子网掩码。在第10章中，我们将看到RIP不支持变长子网，RIP第2版和OSPF则支持变长子网。在我们的例子中不存在这种问题，因为在我的子网中不要求使用RIP协议。

作者子网中的IP地址结构如图3-11所示，11位子网号中的前8 bit始终是13。在剩下的3 bit中，我们用二进制001表示以太网，010表示点对点SLIP链路。这个变长子网掩码在140.252网络中不会给其他主机和路由器带来问题——只要目的是子网140.252.13的所有数据报都传给路由器sun（IP地址是140.252.1.29），如图3-11所示。如果sun知道子网13中的主机有11 bit子网号，那么一切都好办了。

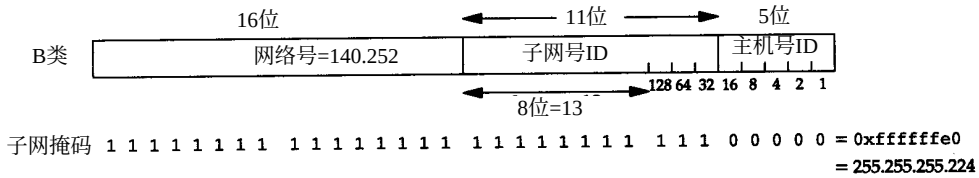


图3-11 变长子网

140.252.13子网中的所有接口的子网掩码是255.255.255.224，或0xffffffe0。这表明最右边的5 bit留给主机号，左边的27 bit留给网络号和子网号。

图3-10中所有接口的IP地址和子网掩码的分配情况如图3-12所示。

主机	IP地址	子网掩码	网络号/子网号	主机号	注　　释
sun	140.252.1.29 140.252.13.33	255.255.255.0 255.255.255.224	140.252.1 140.252.13.32	29 1	在子网1上 在作者所在子网上
svr4	140.252.13.34	255.255.255.224	140.252.13.32	2	
bsdi	140.252.13.35 140.252.13.66	255.255.255.224 255.255.255.224	140.252.13.32 140.252.13.64	3 2	在以太网上 点对点
slip	140.252.13.65	255.255.255.224	140.252.13.64	1	点对点
	140.252.13.63	255.255.255.224	140.252.13.32	31	以太网上的广播地址

图3-12 作者子网的IP地址

第1栏标为是“主机”，但是sun和bsdi也具有路由器的功能，因为它们是多接口的，可以把分组数据从一个接口转发到另一个接口。

这个表中的最后一行是图 3-10中的广播地址 140.252.13.63：它是根据以太网子网号（140.252.13.32）和图3-11中的低5位置1（16＋8＋4＋2＋1＝31）得来的（我们在第12章中将看到，这个地址被称作以子网为目的的广播地址（subnet-directed broadcast address））。

Linux公社 www.linuxidc.com

3.8 ifconfig命令

到目前为止，我们已经讨论了链路层和 IP层，现在可以介绍 TCP/IP对网络接口进行配置和查询的命令了。ifconfig(8)命令一般在引导时运行，以配置主机上的每个接口。

由于拨号接口可能会经常接通和挂断（如 SLIP链路），每次线路接通和挂断时，ifconfig都必须（以某种方法）运行。这个过程如何完成取决于使用的 SLIP软件。

下面是作者子网接口的有关参数。请把它们与图 3-12的值进行比较。

```
sun % /usr/etc/ifconfig -a              在所有接口报告的选项
le0: flags=63<UP,BROADCAST,NOTRAILERS,RUNNING>
        inet 140.252.13.33 netmask ffffffe0 broadcast 140.252.13.63
sl0: flags=1051<UP,POINTOPOINT,RUNNING,LINK0>
        inet 140.252.1.29 --> 140.252.1.183 netmask ffffff00
lo0: flags=49<UP,LOOPBACK,RUNNING>
        inet 127.0.0.1 netmask ff000000
```

环回接口（2.7节）被认为是一个网络接口。它是一个 A类地址，没有进行子网划分。

需要注意的是以太网没有采用尾部封装（ 2.3节），而且可以进行广播，而 SLIP链路是一个点对点的链接。

SLIP接口的标志LINK0是一个允许压缩slip的数据（CSLIP，参见2.5节）的配置选项。其他的选项有LINK1（如果从另一端收到一份压缩报文，就允许采用 CSLIP）和LINK2（所有外出的ICMP报文都被丢弃）。我们在4.6节中将讨论SLIP链接的目的地址。

> 安装指南中的注释对最后这个选项进行了解释：“一般它不应设置，但是由于一些不当的ping操作，可能会导致吞吐量降到0。”

bsdi是另一台路由器。由于- a参数是SunOS操作系统具有的功能，因此我们必须多次执行ifconfig，并指定接口名字参数：

```
bsdi % /sbin/ifconfig we0
we0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,SIMPLEX>
        inet 140.252.13.35 netmask ffffffe0 broadcast 140.252.13.63
bsdi % /sbin/ifconfig sl0
sl0: flags=1011<UP,POINTOPOINT,LINK0>
        inet 140.252.13.66 --> 140.252.13.65 netmask ffffffe0
```

这里，我们看到以太网接口（we0）的一个新选项：SIMPLEX。这个4.4BSD标志表明接口不能收到本机传送的数据。在 BSD/386中所有的以太网都这样设置。一旦这样设置后，如果接口发送一帧数据到广播地址，那么就会为本机拷贝一份数据送到环回地址（在 6.3小节我们将举例说明这一点）。

在主机slip中，SLIP接口的设置基本上与上面的bsdi一致，只是两端的IP地址进行了互换：

```
slip % /sbin/ifconfig sl0
sl0: flags=1011<UP,POINTOPOINT,LINK0>
        inet 140.252.13.65 --> 140.252.13.66 netmask ffffffe0
```

最后一个接口是主机 svr4上的以太网接口。它与前面的以太网接口类似，只是 SVR4版的ifconfig没有打印RUNNING标志：

```
svr4 % /usr/sbin/ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask ffffffe0 broadcast 140.252.13.63
```

Linux公社 www.linuxidc.com

ifconfig命令一般支持TCP/IP以外的其他协议族，而且有很多参数。关于这些细节可以查看系统说明书。

3.9 netstat命令

netstat(1)命令也提供系统上的接口信息。-i参数将打印出接口信息，-n参数则打印出IP地址，而不是主机名字。

```
sun % netstat -in
Name  Mtu  Net/Dest       Address         Ipkts  Ierrs Opkts  Oerrs Collis Queue
le0   1500 140.252.13.32 140.252.13.33  67719   0     92133   0     1      0
sl0   552  140.252.1.183 140.252.1.29   48035   0     54963   0     0      0
lo0   1536 127.0.0.0     127.0.0.1      15548   0     15548   0     0      0
```

这个命令打印出每个接口的MTU、输入分组数、输入错误、输出分组数、输出错误、冲突以及当前的输出队列长度。

在第9章将用netstat命令检查路由表，那时再回头讨论该命令。另外，在第13章将用它的一个改进版本来查看活动的广播组。

3.10 IP的未来

IP主要存在三个方面的问题。这是Internet在过去几年快速增长所造成的结果（参见习题1.2）。

1) 超过半数的B类地址已被分配。根据估计，它们大约在1995年耗尽。

2) 32 bit的IP地址从长期的Internet增长角度来看，一般是不够用的。

3) 当前的路由结构没有层次结构，属于平面型(flat)结构，每个网络都需要一个路由表目。随着网络数目的增长，一个具有多个网络的网站就必须分配多个C类地址，而不是一个B类地址，因此路由表的规模会不断增长。

无类别的域间路由选择CIDR（Classless Interdomain Routing）提出了一个可以解决第三个问题的建议，对当前版本的IP（IP版本4）进行扩充，以适应21世纪Internet的发展。对此我们将在10.8节进一步详细介绍。

对新版的IP，即下一代IP，经常称作IPng，主要有四个方面的建议。1993年5月发行的IEEE *Network* (vol.7, no.3)对前三个建议进行了综述，同时有一篇关于CIDR的论文。RFC 1454 [Dixon 1993]对前三个建议进行了比较。

1) SIP，简单Internet协议。它针对当前的IP提出了一个最小幅度的修改建议，采用64位地址和一个不同的首部格式（首部的前4比特仍然包含协议的版本号，其值不再是4）。

2) PIP。这个建议也采用了更大的、可变长度的和有层次结构的地址，而且首部格式也不相同。

3) TUBA，代表“TCP and UDP with Bigger Address”，它基于OSI的CLNP（Connectionless Network Protocol，无连接网络协议），一个与IP类似的OSI协议。它提供大得多的地址空间：可变长度，可达20个字节。由于CLNP是一个现有的协议，而SIP和PIP只是建议，因此关于CLNP的文档已经出现。RFC 1347[Callon 1992]提供了TUBA的有关细节。文献[Perlman 1992]的第7章对IPv4和CLNP进行了比较。许多路由器已经支持CLNP，但是很少有主机也提供支持。

Linux公社 www.linuxidc.com

4) TP/IX，由RFC 1475 [Ullmann 1993]对它进行了描述。虽然SIP采用了64 bit的址址，但是它还改变了TCP和UDP的格式：两个协议均为32 bit的端口号，64 bit的序列号，64 bit的确认号，以及TCP的32 bit窗口。

前三个建议基本上采用了相同版本的TCP和UDP作为传输层协议。

由于四个建议只能有一个被选为IPv4的替换者，而且在你读到此书时可能已经做出选择，因此我们对它们不进行过多评论。虽然CIDR即将实现以解决目前的短期问题，但是IPv4后继者的实现则需要经过许多年。

3.11 小结

本章开始描述了IP首部的格式，并简要讨论了首部中的各个字段。我们还介绍了IP路由选择，并指出主机的路由选择可以非常简单：如果目的主机在直接相连的网络上，那么就把数据报直接传给目的主机，否则传给默认路由器。

在进行路由选择决策时，主机和路由器都使用路由表。在表中有三种类型的路由：特定主机型、特定网络型和默认路由型。路由表中的表目具有一定的优先级。在选择路由时，主机路由优先于网络路由，最后在没有其他可选路由存在时才选择默认路由。

IP路由选择是通过逐跳来实现的。数据报在各站的传输过程中目的IP地址始终不变，但是封装和目的链路层地址在每一站都可以改变。大多数的主机和许多路由器对于非本地网络的数据报都使用默认的下一站路由器。

A类和B类地址一般都要进行子网划分。用于子网号的比特数通过子网掩码来指定。我们为此举了一个实例来详细说明，即作者所在的子网，并介绍了变长子网的概念。子网的划分缩小了Internet路由表的规模，因为许多网络经常可以通过单个表目就可以访问了。接口和网络的有关信息通过`ifconfig`和`netstat`命令可以获得，包括接口的IP地址、子网掩码、广播地址以及MTU等。

在本章的最后，我们对Internet协议族潜在的改进建议——下一代IP进行了讨论。

习题

3.1 环回地址必须是127.0.0.1吗？

3.2 在图3-6中指出有两个网络接口的路由器。

3.3 子网号为16 bit的A类地址与子网号为8 bit的B类地址的子网掩码有什么不同？

3.4 阅读RFC 1219 [Tsuchiya 1991]，学习分配子网号和主机号的有关推荐技术。

3.5 子网掩码255.255.0.255是否对A类地址有效？

3.6 你认为为什么3.9小节中打印出来的环回接口的MTU要设置为1536？

3.7 TCP/IP协议族是基于一种数据报的网络技术，即IP层，其他的协议族则基于面向连接的网络技术。阅读文献[Clark 1988]，找出数据报网络层提供的三个优点。

Linux公社 www.linuxidc.com

第4章　ARP：地址解析协议

4.1　引言

本章我们要讨论的问题是只对 TCP/IP协议簇有意义的IP地址。数据链路如以太网或令牌环网都有自己的寻址机制（常常为 48 bit地址），这是使用数据链路的任何网络层都必须遵从的。一个网络如以太网可以同时被不同的网络层使用。例如，一组使用 TCP/IP协议的主机和另一组使用某种PC网络软件的主机可以共享相同的电缆。

当一台主机把以太网数据帧发送到位于同一局域网上的另一台主机时，是根据 48 bit的以太网地址来确定目的接口的。设备驱动程序从不检查 IP数据报中的目的IP地址。

地址解析为这两种不同的地址形式提供映射： 32 bit的IP地址和数据链路层使用的任何类型的地址。 RFC 826 [Plummer 1982]是ARP规范描述文档。

本章及下一章我们要讨论的两种协议如图 4-1所示：ARP（地址解析协议）和RARP（逆地址解析协议）。

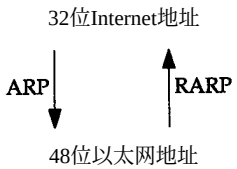


图4-1　地址解析协议：ARP和RARP

ARP为IP地址到对应的硬件地址之间提供动态映射。我们之所以用动态这个词是因为这个过程是自动完成的，一般应用程序用户或系统管理员不必关心。

RARP是被那些没有磁盘驱动器的系统使用（一般是无盘工作站或 X终端），它需要系统管理员进行手工设置。我们在第 5章对它进行讨论。

4.2　一个例子

任何时候我们敲入下面这个形式的命令：

```
% ftp bsdi
```

都会进行以下这些步骤。这些步骤的序号如图 4-2所示。

1) 应用程序FTP客户端调用函数gethostbyname(3)把主机名（bsdi）转换成32 bit的IP地址。这个函数在DNS（域名系统）中称作解析器，我们将在第 14章对它进行介绍。这个转换过程或者使用DNS，或者在较小网络中使用一个静态的主机文件（/etc/hosts）。
2) FTP客户端请求TCP用得到的IP地址建立连接。
3) TCP发送一个连接请求分段到远端的主机，即用上述 IP地址发送一份 IP数据报（在第 18章我们将讨论完成这个过程的细节）。
4) 如果目的主机在本地网络上（如以太网、令牌环网或点对点链接的另一端），那么IP数据报可以直接送到目的主机上。如果目的主机在一个远程网络上，那么就通过 IP选路函数来确定位于本地网络上的下一站路由器地址，并让它转发 IP数据报。在这两种情况下，IP数据报都是被送到位于本地网络上的一台主机或路由器。
5) 假定是一个以太网，那么发送端主机必须把 32 bit的IP地址变换成48 bit的以太网地址。

Linux公社 www.linuxidc.com

从逻辑Internet地址到对应的物理硬件地址需要进行翻译。这就是 ARP的功能。

ARP本来是用于广播网络的，有许多主机或路由器连在同一个网络上。

6) ARP发送一份称作ARP请求的以太网数据帧给以太网上的每个主机。这个过程称作广播，如图 4-2中的虚线所示。 ARP请求数据帧中包含目的主机的 IP地址（主机名为 bsdi），其意思是“如果你是这个IP地址的拥有者，请回答你的硬件地址。”

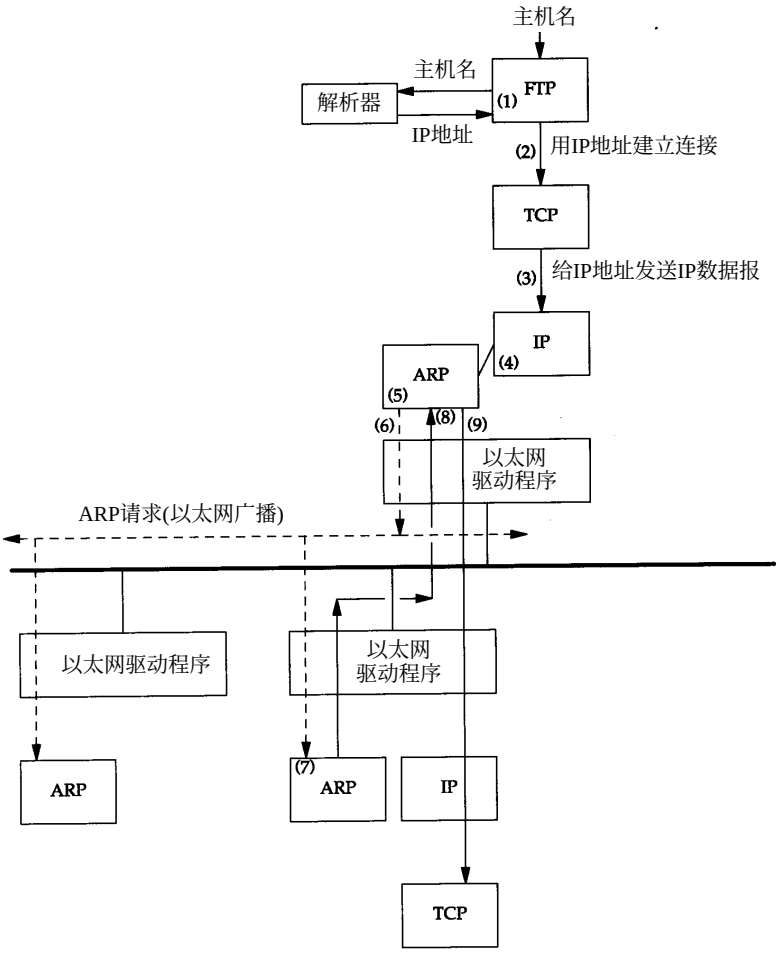


图4-2 当用户输入命令“ftp 主机名”时ARP的操作

7) 目的主机的ARP层收到这份广播报文后，识别出这是发送端在寻问它的 IP地址，于是发送一个ARP应答。这个ARP应答包含IP地址及对应的硬件地址。

8) 收到ARP应答后，使ARP进行请求—应答交换的IP数据报现在就可以传送了。

9) 发送IP数据报到目的主机。

在ARP背后有一个基本概念，那就是网络接口有一个硬件地址（一个 48 bit的值，标识不同的以太网或令牌环网络接口）。在硬件层次上进行的数据帧交换必须有正确的接口地址。但是，TCP/IP有自己的地址：32 bit的IP地址。知道主机的IP地址并不能让内核发送一帧数据给主机。内核（如以太网驱动程序）必须知道目的端的硬件地址才能发送数据。 ARP的功能是在32 bit的IP地址和采用不同网络技术的硬件地址之间提供动态映射。

点对点链路不使用ARP。当设置这些链路时（一般在引导过程进行），必须告知内核链路

Linux公社 www.linuxidc.com

每一端的IP地址。像以太网地址这样的硬件地址并不涉及。

4.3 ARP高速缓存

ARP高效运行的关键是由于每个主机上都有一个 ARP高速缓存。这个高速缓存存放了最近Internet地址到硬件地址之间的映射记录。高速缓存中每一项的生存时间一般为 20分钟，起始时间从被创建时开始算起。

我们可以用arp(8)命令来检查ARP高速缓存。参数-a的意思是显示高速缓存中所有的内容。

```
bsdi %arp -a
sun (140.252.13.33) at 8:0:20:3:f6:42
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
```

48 bit的以太网地址用6个十六进制的数来表示，中间以冒号隔开。在 4.8小节我们将讨论arp命令的其他功能。

4.4 ARP的分组格式

在以太网上解析IP地址时，ARP请求和应答分组的格式如图 4-3所示（ARP可以用于其他类型的网络，可以解析 IP地址以外的地址。紧跟着帧类型字段的前四个字段指定了最后四个字段的类型和长度）。

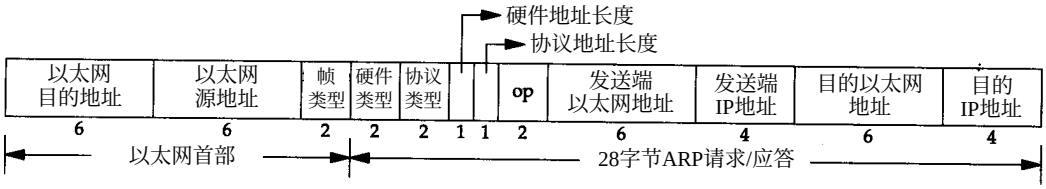


图4-3 用于以太网的ARP请求或应答分组格式

以太网报头中的前两个字段是以太网的源地址和目的地址。目的地址为全 1的特殊地址是广播地址。电缆上的所有以太网接口都要接收广播的数据帧。

两个字节长的以太网帧类型表示后面数据的类型。对于 ARP请求或应答来说，该字段的值为0x0806。

形容词hardware(硬件)和protocol(协议)用来描述ARP分组中的各个字段。例如，一个 ARP请求分组询问协议地址（这里是 IP地址）对应的硬件地址（这里是以太网地址）。

硬件类型字段表示硬件地址的类型。它的值为 1即表示以太网地址。协议类型字段表示要映射的协议地址类型。它的值为 0x0800即表示IP地址。它的值与包含 IP数据报的以太网数据帧中的类型字段的值相同，这是有意设计的（参见图 2-1）。

接下来的两个1字节的字段，硬件地址长度和协议地址长度分别指出硬件地址和协议地址的长度，以字节为单位。对于以太网上 IP地址的ARP请求或应答来说，它们的值分别为 6和4。

操作字段指出四种操作类型，它们是 ARP请求（值为1）、ARP应答（值为2）、RARP请求（值为3）和RARP应答（值为4）（我们在第5章讨论RARP）。这个字段必需的，因为 ARP请求和ARP应答的帧类型字段值是相同的。

接下来的四个字段是发送端的硬件地址（在本例中是以太网地址）、发送端的协议地址（IP地址）、目的端的硬件地址和目的端的协议地址。注意，这里有一些重复信息：在以太网

Linux公社 www.linuxidc.com

的数据帧报头中和ARP请求数据帧中都有发送端的硬件地址。

对于一个ARP请求来说，除目的端硬件地址外的所有其他的字段都有填充值。当系统收到一份目的端为本机的ARP请求报文后，它就把硬件地址填进去，然后用两个目的端地址分别替换两个发送端地址，并把操作字段置为2，最后把它发送回去。

4.5 ARP举例

在本小节中，我们用tcpdump命令来看一看运行像Telnet这样的普通TCP工具软件时ARP会做些什么。附录A包含tcpdump命令的其他细节。

4.5.1 一般的例子

为了看清楚ARP的运作过程，我们执行telnet命令与无效的服务器连接。

```
bsdi % arp -a                          检验ARP高速缓存是空的
bsdi % telnet svr4 discard             连接无效的服务器
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
^]                                     键入Ctrl和右括号，使Telnet回到提示符并关闭
telnet> quit
Connection closed.
```

当我们在另一个系统（sun）上运行带有-e选项的tcpdump命令时，显示的是硬件地址（在我们的例子中是48 bit的以太网地址）。

图4-4中的tcpdump的原始输出如附录A中的图A-3所示。由于这是本书第一个tcpdump输出例子，你应该去查看附录中的原始输出，看看我们作了哪些修改。

```
1  0.0                 0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff arp 60:
                       arp who-has svr4 tell bsdi
2  0.002174 (0.0022)   0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60:
                       arp reply svr4 is-at 0:0:c0:c2:9b:26
3  0.002831 (0.0007)   0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60:
                       bsdi.1030 > svr4.discard: S 596459521:596459521(0)
                       win 4096 <mss 1024> [tos 0x10]
4  0.007834 (0.0050)   0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60:
                       svr4.discard > bsdi.1030: S 3562228225:3562228225(0)
                       ack 596459522 win 4096 <mss 1024>
5  0.009615 (0.0018)   0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60:
                       bsdi.1030 > svr4.discard: . ack 1 win 4096 [tos 0x10]
```

图4-4 TCP连接请求产生的ARP请求和应答

我们删除了tcpdump命令输出的最后四行，因为它们是结束连接的信息（我们将在第18章进行讨论），与这里讨论的内容不相关。

在第1行中，源端主机（bsdi）的硬件地址是0:0:c0:6f:2d:40。目的端主机的硬件地址是ff:ff:ff:ff:ff:ff，这是一个以太网广播地址。电缆上的每个以太网接口都要接收这个数据帧并对它进行处理，如图4-2所示。

第1行中紧接着的一个输出字段是arp，表明帧类型字段的值是0x0806，说明此数据帧是一个ARP请求或回答。

在每行中，单词arp或ip后面的值60指的是以太网数据帧的长度。由于ARP请求或回答

Linux公社 www.linuxidc.com

的数据帧长都是42字节（28字节的ARP数据，14字节的以太网帧头），因此，每一帧都必须加入填充字符以达到以太网的最小长度要求：60字节。

请参见图1-7，这个最小长度60字节包含14字节的以太网帧头，但是不包括4个字节的以太网帧尾。有一些书把最小长度定为64字节，它包括以太网的帧尾。我们在图1-7中把最小长度定为46字节，是有意不包括14字节的帧首部，因为对应的最大长度（1500字节）指的是MTU——最大传输单元（见图2-5）。我们使用MTU经常是因为它对IP数据报的长度进行限制，但一般与最小长度无关。大多数的设备驱动程序或接口卡自动地用填充字符把以太网数据帧充满到最小长度。第3，4和5行中的IP数据报（包含TCP段）的长度都比最小长度短，因此都必须填充到60字节。

第1行中的下一个输出字段`arp who-has`表示作为ARP请求的这个数据帧中，目的IP地址是svr4的地址，发送端的IP地址是`bsdi`的地址。`tcpdump`打印出主机名对应的默认IP地址（在4.7节中，我们将用`-n`选项来查看ARP请求中真正的IP地址。）

从第2行中可以看到，尽管ARP请求是广播的，但是ARP应答的目的地址却是`bsdi`（0:0:c0:6f:2d:40）。ARP应答是直接送到请求端主机的，而是广播的。

`tcpdump`打印出`arp reply`的字样，同时打印出响应者的主机名和硬件地址。

第3行是第一个请求建立连接的TCP段。它的目的硬件地址是目的主机（`svr4`）。我们将在第18章讨论这个段的细节内容。

在每一行中，行号后面的数字表示`tcpdump`收到分组的时间（以秒为单位）。除第1行外，其他每行在括号中还包含了与上一行的时间差异（以秒为单位）。从这个图可以看出，发送ARP请求与收到ARP回答之间的延时是2.2 ms。而在0.7 ms之后发出第一段TCP报文。在本例中，用ARP进行动态地址解析的时间小于3 ms。

最后需要指出的一点，在`tcpdump`命令输出中，我们没有看到svr4在发出第一段TCP报文（第4行）之前发出的ARP请求。这是因为可能在`svr4`的ARP高速缓存中已经有`bsdi`的表项。一般情况下，当系统收到ARP请求或发送ARP应答时，都要把请求端的硬件地址和IP地址存入ARP高速缓存。在逻辑上可以假设，如果请求端要发送IP数据报，那么数据报的接收端将很可能会发送一个应答。

4.5.2 对不存在主机的ARP请求

如果查询的主机已关机或不存在会发生什么情况呢？为此我们指定一个并不存在的Internet地址——根据网络号和子网号所对应的网络确实存在，但是并不存在所指定的主机号。从图3-10可以看出，主机号从36到62的主机并不存在（主机号为63是广播地址）。这里，我们用主机号36来举例。

```
                                    这次是Telnet的一个地址，而不是主机名
bsdi % date ; telnet 140.252.13.36 ; date
Sat Jan 30 06:46:33 MST 1993
Trying 140.252.13.36...
telnet: Unable to connect to remote host: Connection timed out
Sat Jan 30 06:47:49 MST 1993        在前一个日期输出后76秒

bsdi % arp -a                       检查ARP高速缓存
? (140.252.13.36) at (incomplete)
```

tcpdump命令的输出如图4-5所示。

Linux公社 www.linuxidc.com

```
1    0.0                    arp who-has 140.252.13.36 tell bsdi
2    5.509069 ( 5.5091)     arp who-has 140.252.13.36 tell bsdi
3   29.509745 (24.0007)     arp who-has 140.252.13.36 tell bsdi
```

图4-5 对不存在主机的ARP请求

这一次，我们没有用-e选项，因为已经知道ARP请求是在网上广播的。

令人感兴趣的是看到多次进行ARP请求：第1次请求发生后5.5秒进行第2次请求，在24秒之后又进行第3次请求（在第21章我们将看到TCP的超时和重发算法的细节）。tcpdump命令输出的超时限制为29.5秒。但是，在telnet命令使用前后分别用date命令检查时间，可以发现Telnet客户端的连接请求似乎在大约75秒后才放弃。事实上，我们在后面将看到，大多数的BSD实现把完成TCP连接请求的时间限制设置为75秒。

在第18章中，当我们看到建立连接的TCP报文段序列时，会发现ARP请求对应于TCP试图发送的初始TCPSYN（同步）段。

注意，在线路上始终看不到TCP的报文段。我们能看到的是ARP请求。直到ARP回答返回时，TCP报文段才可以被发送，因为硬件地址到这时才可能知道。如果我们用过滤模式运行tcpdump命令，只查看TCP数据，那么将没有任何输出。

4.5.3 ARP高速缓存超时设置

在ARP高速缓存中的表项一般都要设置超时值（在4.8小节中，我们将看到管理员可以用arp命令把地址放入高速缓存中而不设置超时值）。从伯克利系统演变而来的系统一般对完整的表项设置超时值为20分钟，而对不完整的表项设置超时值为3分钟（在前面的例子中我们已见过一个不完整的表项，即在以太网上对一个不存在的主机发出ARP请求。）当这些表项再次使用时，这些实现一般都把超时值重新设为20分钟。

Host Requirements RFC表明即使表项正在使用时，超时值也应该启动，但是大多数从伯克利系统演变而来的系统没有这样做——它们每次都是在访问表项时重设超时值。

4.6 ARP代理

如果ARP请求是从一个网络的主机发往另一个网络上的主机，那么连接这两个网络的路由器就可以回答该请求，这个过程称作委托ARP或ARP代理(Proxy ARP)。这样可以欺骗发起ARP请求的发送端，使它误以为路由器就是目的主机，而事实上目的主机是在路由器的“另一边”。路由器的功能相当于目的主机的代理，把分组从其他主机转发给它。

举例是说明ARP代理的最好方法。如图3-10所示，系统sun与两个以太网相连。但是，我们也指出过，事实上并不是这样，请把它与封内图1进行比较。在sun和子网140.252.1之间实际存在一个路由器，就是这个具有ARP代理功能的路由器使得sun就好像在子网140.252.1上一样。具体安置如图4-6所示，路由器Telebit NetBlazer，取名为netb，在子网和主机sun之间。

当子网140.252.1（称作gemini）上的其他主机有一份IP数据报要传给地址为140.252.1.29的sun时，gemini比较网络号（140.252）和子网号（1），因为它们都是相同的，因而在图4-6上面的以太网中发送IP地址140.252.1.29的ARP请求。路由器netb识别出该IP地址属于它的一个拨号主机，于是把它的以太网接口地址140.252.1作为硬件地址来回答。主机gemini通过以太网发送IP数据报到netb，netb通过拨号SLIP链路把数据报转发到sun。这个过程对于所有

Linux公社 www.linuxidc.com

140.252.1子网上的主机来说都是透明的，主机sun实际上是在路由器netb后面进行配置的。

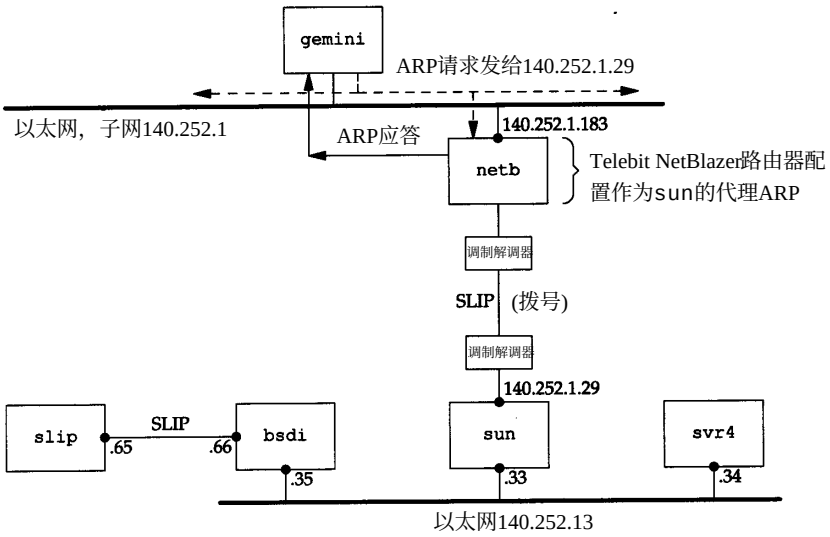


图4-6 ARP代理的例子

如果在主机gemini上执行arp命令，经过与主机sun通信以后，我们发现在同一个子网140.252.1上的netb和sun的IP地址映射的硬件地址是相同的。这通常是使用委托ARP的线索。

```
gemini %arp -a
                                    这里是子网140.252.1上其他主机的输出行
netb (140.252.1.183) at 0:80:ad:3:6a:80
sun (140.252.1.29) at 0:80:ad:3:6a:80
```

图4-6中的另一个需要解释的细节是在路由器 netb的下方（SLIP链路）显然缺少一个 IP地址。为什么在拨号 SLIP链路的两端只拥有一个 IP地址，而在bsdi和slip之间的两端却分别有一个IP地址？在3.8小节我们已经指出，用ifconfig命令可以显示拨号 SLIP链路的目的地址，它是140.252.1.183。NetBlazer不需要知道拨号 SLIP链路每一端的IP地址（这样做会用更多的IP地址）。相反，它通过分组到达的串行线路接口来确定发送分组的拨号主机，因此对于连接到路由器的每个拨号主机不需要用唯一的 IP地址。所有的拨号主机使用同一个 IP地址140.252.1.183作为SLIP链路的目的地址。

ARP代理可以把数据报传送到路由器 sun上，但是子网140.252.13上的其他主机是如何处理的呢？选路必须使数据报能到达其他主机。这里需要特殊处理，选路表中的表项必须在网络140.252的某个地方制定，使所有数据报的目的端要么是子网 140.252.13，要么是子网上的某个主机，这样都指向路由器 netb。而路由器netb知道如何把数据报传到最终的目的端，即通过路由器sun。

ARP代理也称作混合ARP（promiscuousARP）或ARP 出租(ARP hack)。这些名字来自于ARP代理的其他用途：通过两个物理网络之间的路由器可以互相隐藏物理网络。在这种情况下，两个物理网络可以使用相同的网络号，只要把中间的路由器设置成一个 ARP代理，以响应一个网络到另一个网络主机的 ARP请求。这种技术在过去用来隐藏一组在不同物理电缆上运行旧版TCP/IP的主机。分开这些旧主机有两个共同的理由，其一是它们不能处理子网划分，其二是它们使用旧的广播地址（所有比特值为 0的主机号，而不是目前使用的所有比特值为 1

Linux公社 www.linuxidc.com

的主机号）。

4.7 免费ARP

我们可以看到的另一个ARP特性称作免费ARP (gratuitous ARP)。它是指主机发送ARP查找自己的IP地址。通常，它发生在系统引导期间进行接口配置的时候。

在互联网中，如果我们引导主机bsdi并在主机sun上运行tcpdump命令，可以看到如图4-7所示的分组。

```
1  0.0                    0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff arp 60:
                          arp who-has 140.252.13.35 tell 140.252.13.35
```

图4-7 免费ARP的例子

（我们用-n选项运行tcpdump命令，打印出点分十进制的地址，而不是主机名）。对于ARP请求中的各字段来说，发送端的协议地址和目的端的协议地址是一致的：即主机 bsdi的地址140.252.13.35。另外，以太网报头中的源地址0:0:c0:6f:2d:40，正如tcpdump命令显示的那样，等于发送端的硬件地址（见图4-4）。

免费ARP可以有两个方面的作用：

1) 一个主机可以通过它来确定另一个主机是否设置了相同的 IP地址。主机bsdi并不希望对此请求有一个回答。但是，如果收到一个回答，那么就会在终端日志上产生一个错误消息“以太网地址：a:b:c:d:e:f发送来重复的IP地址”。这样就可以警告系统管理员，某个系统有不正确的设置。

2) 如果发送免费ARP的主机正好改变了硬件地址（很可能是主机关机了，并换了一块接口卡，然后重新启动），那么这个分组就可以使其他主机高速缓存中旧的硬件地址进行相应的更新。一个比较著名的ARP协议事实[Plummer 1982]是，如果主机收到某个IP地址的ARP请求，而且它已经在接收者的高速缓存中，那么就要用 ARP请求中的发送端硬件地址（如以太网地址）对高速缓存中相应的内容进行更新。主机接收到任何 ARP请求都要完成这个操作（ARP请求是在网上广播的，因此每次发送ARP请求时网络上的所有主机都要这样做）。

文献[Bhide、Elnozahy和Morgan 1991]中有一个应用例子，通过发送含有备份硬件地址和故障服务器的IP地址的免费ARP请求，使得备份文件服务器可以顺利地接替故障服务器进行工作。这使得所有目的地为故障服务器的报文都被送到备份服务器那里，客户程序不用关心原来的服务器是否出了故障。

> 不幸的是，作者却反对这个做法，因为这取决于所有不同类型的客户端都要有正确的ARP协议实现。他们显然碰到过客户端的ARP协议实现与规范不一致的情况。
>
> 通过检查作者所在子网上的所有系统可以发现，SunOS 4.1.3和4.4BSD在引导时都发送免费ARP，但是SVR4却没有这样做。

4.8 arp命令

我们已经用过这个命令及参数-a来显示ARP高速缓存中的所有内容。这里介绍其他参数的功能。

超级用户可以用选项-d来删除ARP高速缓存中的某一项内容（这个命令格式可以在运行

Linux公社 www.linuxidc.com

一些例子之前使用，以让我们看清楚ARP的交换过程）。

另外，可以通过选项-s来增加高速缓存中的内容。这个参数需要主机名和以太网地址：对应于主机名的IP地址和以太网地址被增加到高速缓存中。新增加的内容是永久性的（比如，它没有超时值），除非在命令行的末尾附上关键字temp。

位于命令行末尾的关键字pub和-s选项一起，可以使系统起着主机ARP代理的作用。系统将回答与主机名对应的IP地址的ARP请求，并以指定的以太网地址作为应答。如果广播的地址是系统本身，那么系统就为指定的主机名起着委托ARP代理的作用。

4.9 小结

在大多数的TCP/IP实现中，ARP是一个基础协议，但是它的运行对于应用程序或系统管理员来说一般是透明的。ARP高速缓存在它的运行过程中非常关键，我们可以用arp命令对高速缓存进行检查和操作。高速缓存中的每一项内容都有一个定时器，根据它来删除不完整和完整的表项。arp命令可以显示和修改ARP高速缓存中的内容。

我们介绍了ARP的一般操作，同时也介绍了一些特殊的功能：委托ARP（当路由器对来自于另一个路由器接口的ARP请求进行应答时）和免费ARP（发送自己IP地址的ARP请求，一般发生在引导过程中）。

习题

4.1 当输入命令以生成类似图4-4那样的输出时，发现本地ARP快速缓存为空以后，输入命令

```
bsdi % rsh svr4 arp -a
```

如果发现目的主机上的ARP快速缓存也是空的，那将发生什么情况？（该命令将在svr4主机上运行arp -a命令）。

4.2 请描述如何判断一个给定主机是否能正确处理接收到的非必要的ARP请求的方法。

4.3 由于发送一个数据包后ARP将等待响应，因此4.2节所描述的步骤7可能会持续一段时间。你认为ARP将如何处理在这期间收到相同目的IP地址发来的多个数据包？

4.4 在4.5节的最后，我们指出Host Requirements RFC和伯克利派生系统在处理活动ARP表目的超时时存在差异。那么如果我们在一个由伯克利派生系统的客户端上，试图与一个正在更换以太网卡而处于关机状态的服务器主机联系，这时会发生什么情况？如果服务器在引导过程中广播一份免费ARP，这种情况是否会发生变化？

Linux公社 www.linuxidc.com

第5章　RARP：逆地址解析协议

5.1　引言

具有本地磁盘的系统引导时，一般是从磁盘上的配置文件中读取 IP地址。但是无盘机，如X终端或无盘工作站，则需要采用其他方法来获得 IP地址。

网络上的每个系统都具有唯一的硬件地址，它是由网络接口生产厂家配置的。无盘系统的RARP实现过程是从接口卡上读取唯一的硬件地址，然后发送一份 RARP请求（一帧在网络上广播的数据），请求某个主机响应该无盘系统的 IP地址（在RARP应答中）。

在概念上这个过程是很简单的，但是实现起来常常比 ARP要困难，其原因在本章后面介绍。RARP的正式规范是RFC 903 [Finlayson et al. 1984]。

5.2　RARP的分组格式

RARP分组的格式与 ARP分组基本一致（见图 4-3）。它们之间主要的差别是 RARP请求或应答的帧类型代码为0x8035，而且RARP请求的操作代码为3，应答操作代码为4。

对应于ARP，RARP请求以广播方式传送，而RARP应答一般是单播(unicast)传送的。

5.3　RARP举例

在互联网中，我们可以强制 sun主机从网络上引导，而不是从本地磁盘引导。如果在主机bsdi上运行RARP服务程序和tcpdump命令，就可以得到如图5-1那样的输出。用-e参数使得tcpdump命令打印出硬件地址：

```
1  0.0                 8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42

2  0.13 (0.13)         0:0:c0:6f:2d:40 8:0:20:3:f6:42 rarp 42:
                       rarp reply 8:0:20:3:f6:42 at sun

3  0.14 (0.01)         8:0:20:3:f6:42 0:0:c0:6f:2d:40 ip 65:
                       sun.26999 > bsdi.tftp: 23 RRQ "8CFC0D21.SUN4C"
```

图5-1　RARP请求和应答

RARP请求是广播方式（第 1行），而第2行的RARP应答是单播方式。第 2行的输出中 at sun表示RARP应答包含主机sun的IP地址（140.252.13.33）。

在第3行中，我们可以看到，一旦 sun收到IP地址，它就发送一个TFTP读请求（RRQ）给文件8CFC0D21.SUN4C（TFTP表示简单文件传送协议。我们将在第 15章详细介绍）。文件名中的8个十六进制数字表求主机 sun的IP地址140.252.13.33。这个IP地址在RARP应答中返回。文件名的后缀SUN4C表示被引导系统的类型。

tcpdump在第3行中指出IP数据报的长度是65个字节，而不是一个UDP数据报（实际上是一个UDP数据报），因为我们运行tcpdump命令时带有 - e参数，以查看硬件层的地址。在图5-1中

Linux公社 www.linuxidc.com

需要指出的另一点是，第2行中的以太网数据帧长度比最小长度还要小（在4.5节中我们说过应该是60字节）。其原因是我们在发送该以太网数据帧的系统（bsdi）上运行tcpdump命令。应用程序rarpd写42字节到BSD分组过滤设备上（其中14字节为以太网数据帧的报头，剩下的28字节是RARP应答），这就是tcpdump收到的副本。但是以太网设备驱动程序要把这一短帧填充空白字符以达到最小传输长度（60）。如果我们在另一个系统上运行tcpdump命令，其长度将会是60。

从这个例子可以看出，当无盘系统从 RARP应答中收到它的IP地址后，它将发送TFTP请求来读取引导映象。在这一点上我们将不再进一步详细讨论无盘系统是如何引导的（第 16章将描述无盘X终端利用RARP、BOOTP以及TFTP进行引导的过程）。

当网络上没有RARP服务器时，其结果如图5-2所示。每个分组的目的地址都是以太网广播地址。在who- 后面的以太网地址是目的硬件地址，跟在tell后面的以太网地址是发送端的硬件地址。

请注意重发的频度。第一次重发是在 6.55秒以后，然后增加到42.80秒，然后又减到5.34秒和6.55秒，然后又回到42.79秒。这种不确定的情况一直继续下去。如果计算一下两次重发之间的时间间隔，我们发现存在一种双倍的关系：从 5.34到6.55是1.21秒，从 6.55到8.97是2.42秒，从8.97到13.80是4.83秒，一直这样继续下去。当时间间隔达到某个阈值时（大于42.80秒），它又重新置为5.34秒。

```
1     0.0                8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
2     6.55 ( 6.55)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
3    15.52 ( 8.97)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
4    29.32 (13.80)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
5    52.78 (23.46)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
6    95.58 (42.80)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
7   100.92 ( 5.34)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
8   107.47 ( 6.55)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
9   116.44 ( 8.97)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
10  130.24 (13.80)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
11  153.70 (23.46)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
12  196.49 (42.79)       8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                         rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
```

图5-2 网络中没有RARP服务器的RARP请求

超时间隔采用这样的递增方法比每次都采用相同值的方法要好。在图 6-8中，我们将看到一种错误的超时重发方法，以及在第21章中将看到TCP的超时重发机制。

5.4 RARP服务器的设计

虽然RARP在概念上很简单，但是一个RARP服务器的设计与系统相关而且比较复杂。相反，提供一个ARP服务器很简单，通常是TCP/IP在内核中实现的一部分。由于内核知道 IP地

Linux公社 www.linuxidc.com

址和硬件地址，因此当它收到一个询问 IP地址的ARP请求时，只需用相应的硬件地址来提供应答就可以了。

5.4.1 作为用户进程的RARP服务器

RARP服务器的复杂性在于，服务器一般要为多个主机（网络上所有的无盘系统）提供硬件地址到IP地址的映射。该映射包含在一个磁盘文件中（在 Unix系统中一般位于/etc/ethers目录中）。由于内核一般不读取和分析磁盘文件，因此 RARP服务器的功能就由用户进程来提供，而不是作为内核的TCP/IP实现的一部分。

更为复杂的是，RARP请求是作为一个特殊类型的以太网数据帧来传送的（帧类型字段值为0x8035，如图2-1所示）。这说明RARP服务器必须能够发送和接收这种类型的以太网数据帧。在附录A中，我们描述了BSD分组过滤器、Sun的网络接口栓以及SVR4数据链路提供者接口都可用来接收这些数据帧。由于发送和接收这些数据帧与系统有关，因此 RARP服务器的实现是与系统捆绑在一起的。

5.4.2 每个网络有多个RARP服务器

RARP服务器实现的一个复杂因素是 RARP请求是在硬件层上进行广播的，如图 5-2所示。这意味着它们不经过路由器进行转发。为了让无盘系统在RARP服务器关机的状态下也能引导，通常在一个网络上（例如一根电缆）要提供多个 RARP服务器。

当服务器的数目增加时（以提供冗余备份），网络流量也随之增加，因为每个服务器对每个RARP请求都要发送RARP应答。发送RARP请求的无盘系统一般采用最先收到的 RARP应答（对于ARP，我们从来没有遇到这种情况，因为只有一台主机发送 ARP应答）。另外，还有一种可能发生的情况是每个RARP服务器同时应答，这样会增加以太网发生冲突的概率。

5.5 小结

RARP协议是许多无盘系统在引导时用来获取 IP地址的。RARP分组格式基本上与ARP分组一致。一个RARP请求在网络上进行广播，它在分组中标明发送端的硬件地址，以请求相应IP地址的响应。应答通常是单播传送的。

RARP带来的问题包括使用链路层广播，这样就阻止大多数路由器转发 RARP请求，只返回很少信息：只是系统的 IP地址。在第16章中，我们将看到BOOTP在无盘系统引导时会返回更多的信息：IP地址和引导主机的名字等。

虽然RARP在概念上很简单，但是 RARP服务器的实现却与系统相关。因此，并不是所有的TCP/IP实现都提供RARP服务器。

习题

5.1 RARP需要不同的帧类型字段吗？ARP和RARP都使用相同的值0x0806吗？

5.2 在一个有多个RARP服务器的网络上，如何防止它们的响应发生冲突？

Linux公社 www.linuxidc.com

第6章　ICMP：Internet控制报文协议

6.1　引言

ICMP经常被认为是IP层的一个组成部分。它传递差错报文以及其他需要注意的信息。ICMP报文通常被IP层或更高层协议（TCP或UDP）使用。一些ICMP报文把差错报文返回给用户进程。

ICMP报文是在IP数据报内部被传输的，如图6-1所示。

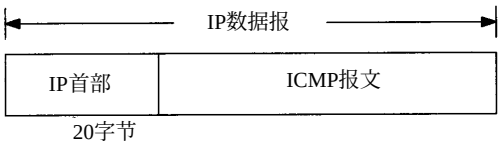


图6-1　ICMP封装在IP数据报内部

ICMP 的正式规范参见 RFC 792 [Posterl 1981b]。

ICMP报文的格式如图6-2所示。所有报文的前4个字节都是一样的，但是剩下的其他字节则互不相同。下面我们将逐个介绍各种报文格式。

类型字段可以有15个不同的值，以描述特定类型的ICMP报文。某些ICMP报文还使用代码字段的值来进一步描述不同的条件。

检验和字段覆盖整个ICMP报文。使用的算法与我们在3.2节中介绍的IP首部检验和算法相同。ICMP的检验和是必需的。

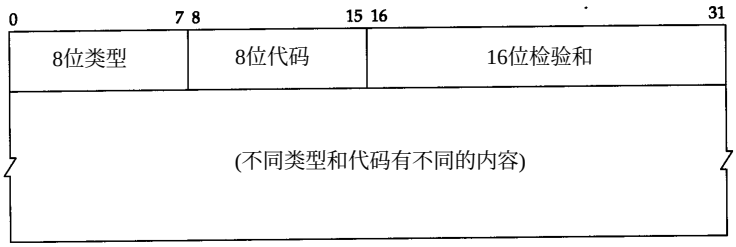


图6-2　ICMP报文

在本章中，我们将一般地讨论ICMP报文，并对其中一部分作详细介绍：地址掩码请求和应答、时间戳请求和应答以及不可达端口。我们将详细介绍第 27章Ping程序所使用的回应请求和应答报文和第9章处理IP路由的ICMP报文。

6.2　ICMP报文的类型

各种类型的ICMP报文如图6-3所示，不同类型由报文中的类型字段和代码字段来共同决定。

图中的最后两列表明ICMP报文是一份查询报文还是一份差错报文。因为对 ICMP差错报文有时需要作特殊处理，因此我们需要对它们进行区分。例如，在对 ICMP差错报文进行响应时，永远不会生成另一份ICMP差错报文（如果没有这个限制规则，可能会遇到一个差错产生另一个差错的情况，而差错再产生差错，这样会无休止地循环下去）。

当发送一份ICMP差错报文时，报文始终包含IP的首部和产生ICMP差错报文的IP数据报的前8个字节。这样，接收ICMP差错报文的模块就会把它与某个特定的协议（根据 IP数据报首

Linux公社 www.linuxidc.com

类型	代码	描述	查询	差错
0	0	回显应答(Ping应答，第7章)	•	
3		目的不可达：		
	0	网络不可达（9.3节）		•
	1	主机不可达（9.3节）		•
	2	协议不可达		•
	3	端口不可达（6.5节）		•
	4	需要进行分片但设置了不分片比特（11.6节）		•
	5	源站选路失败（8.5节）		•
	6	目的网络不认识		•
	7	目的主机不认识		•
	8	源主机被隔离（作废不用）		•
	9	目的网络被强制禁止		•
	10	目的主机被强制禁止		•
	11	由于服务类型TOS，网络不可达（9.3节）		•
	12	由于服务类型TOS，主机不可达（9.3节）		•
	13	由于过滤，通信被强制禁止		•
	14	主机越权		•
	15	优先权中止生效		•
4	0	源端被关闭（基本流控制，11.11节）		•
5		重定向（9.5节）：		•
	0	对网络重定向		•
	1	对主机重定向		•
	2	对服务类型和网络重定向		•
	3	对服务类型和主机重定向		•
8	0	请求回显（Ping请求，第7章）	•	
9	0	路由器通告（9.6节）	•	
10	0	路由器请求（9.6节）	•	
11		超时：		
	0	传输期间生存时间为0（Traceroute, 第8章）		•
	1	在数据报组装期间生存时间为0（11.5节）		•
12		参数问题：		
	0	坏的IP首部（包括各种差错）		•
	1	缺少必需的选项		•
13	0	时间戳请求（6.4节）	•	
14	0	时间戳应答（6.4节）	•	
15	0	信息请求（作废不用）	•	
16	0	信息应答（作废不用）	•	
17	0	地址掩码请求（6.3节）	•	
18	0	地址掩码应答（6.3节）	•	

图6-3 ICMP报文类型

部中的协议字段来判断）和用户进程（根据包含在 IP数据报前8个字节中的TCP或UDP报文首部中的TCP或UDP端口号来判断）联系起来。6.5节将举例来说明一点。

下面各种情况都不会导致产生ICMP差错报文：

1) ICMP差错报文（但是，ICMP查询报文可能会产生ICMP差错报文）。

2) 目的地址是广播地址（见图3-9）或多播地址（D类地址，见图1-5）的IP数据报。

3) 作为链路层广播的数据报。

4) 不是IP分片的第一片（将在11.5节介绍分片）。

5) 源地址不是单个主机的数据报。这就是说，源地址不能为零地址、环回地址、广播地址或多播地址。

这些规则是为了防止过去允许ICMP差错报文对广播分组响应所带来的广播风暴。

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

6.3 ICMP地址掩码请求与应答

ICMP地址掩码请求用于无盘系统在引导过程中获取自己的子网掩码（ 3.5节）。系统广播它的ICMP请求报文（这一过程与无盘系统在引导过程中用 RARP获取IP地址是类似的）。无盘系统获取子网掩码的另一个方法是 BOOTP协议，我们将在第 16章中介绍。ICMP地址掩码请求和应答报文的格式如图6-4所示。

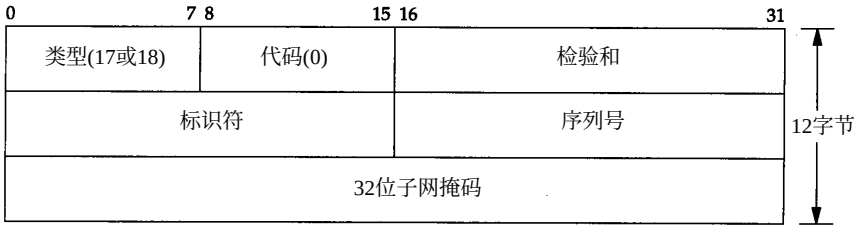


图6-4 ICMP地址掩码请求和应答报文

ICMP报文中的标识符和序列号字段由发送端任意选择设定，这些值在应答中将被返回。这样，发送端就可以把应答与请求进行匹配。

我们可以写一个简单的程序（取名为 icmpaddrmask），它发送一份ICMP地址掩码请求报文，然后打印出所有的应答。由于一般是把请求报文发往广播地址，因此这里我们也这样做。目的地址（140.252.13.63）是子网140.252.13.32的广播地址（见图3-12）。

```
sun %icmpaddrmask 140.252.13.63
received mask = ffffffe0, from 140.252.13.34 来自本机
received mask = ffffffe0, from 140.252.13.35 来自bsdi
received mask = ffff0000, from 140.252.13.34 来自svr4
```

在输出中我们首先注意到的是，从 svr4返回的子网掩码是错的。显然，尽管 svr4接口已经设置了正确的子网掩码，但是 SVR4还是返回了一个普通的 B类地址掩码，就好像子网并不存在一样。

```
svr4 %ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask ffffffe0 broadcast 140.252.13.63
```

SVR4处理ICMP地址掩码请求过程存在差错。

我们用tcpdump命令来查看主机bsdi上的情况，输出如图6-5所示。我们用 - e选项来查看硬件地址。

```
1  0.0              8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff ip 60:
                    sun > 140.252.13.63: icmp: address mask request
2  0.00 (0.00)      0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff ip 46:
                    bsdi > sun: icmp: address mask is 0xffffffe0
3  0.01 (0.01)      0:0:c0:c2:9b:26 8:0:20:3:f6:42 ip 60:
                    svr4 > sun: icmp: address mask is 0xffff0000
```

图6-5 发到广播地址的ICMP地址掩码请求

注意，尽管在线路上什么也看不见，但是发送主机 sun也能接收到ICMP应答（带有上面“来自本机”的输出行）。这是广播的一般特性：发送主机也能通过某种内部环回机制收到一份广播报文拷贝。由于术语“广播”的定义是指局域网上的所有主机，因此它必须包括发送

Linux公社 www.linuxidc.com

主机在内（参见图2-4，当以太网驱动程序识别出目的地址是广播地址后，它就把分组送到网络上，同时传一份拷贝到环回接口）。

接下来，bsdi广播应答，而svr4却只把应答传给请求主机。通常，应答地址必须是单播地址，除非请求端的源IP地址是0.0.0.0。本例不属于这种情况，因此，把应答发送到广播地址是BSD/386的一个内部差错。

> RFC规定，除非系统是地址掩码的授权代理，否则它不能发送地址掩码应答（为了成为授权代理，它必须进行特殊配置，以发送这些应答。参见附录E）。但是，正如我们从本例中看到的那样，大多数主机在收到请求时都发送一个应答，甚至有一些主机还发送差错的应答。

最后一点可以通过下面的例子来说明。我们向本机 IP地址和环回地址分别发送地址掩码请求：

```
sun % icmpaddrmask sun
received mask= ff000000, from 140.252.13.33
sun % icmpaddrmask localhost
received mask= ff000000, from 127.0.0.1
```

上述两种情况下返回的地址掩码对应的都是环回地址，即 A类地址127.0.0.1。还有，我们从图2-4可以看到，发送给本机 IP地址的数据报（140.252.12.33）实际上是送到环回接口。ICMP地址掩码应答必须是收到请求接口的子网掩码（这是因为多接口主机每个接口有不同的子网掩码），因此两种情况下地址掩码请求都来自于环回接口。

6.4 ICMP时间戳请求与应答

ICMP时间戳请求允许系统向另一个系统查询当前的时间。返回的建议值是自午夜开始计算的毫秒数，协调的统一时间（Coordinated Universal Time, UTC）（早期的参考手册认为UTC是格林尼治时间）。这种ICMP报文的好处是它提供了毫秒级的分辨率，而利用其他方法从别的主机获取的时间（如某些 Unix系统提供的rdate命令）只能提供秒级的分辨率。由于返回的时间是从午夜开始计算的，因此调用者必须通过其他方法获知当时的日期，这是它的一个缺陷。

ICMP时间戳请求和应答报文格式如图6-6所示。

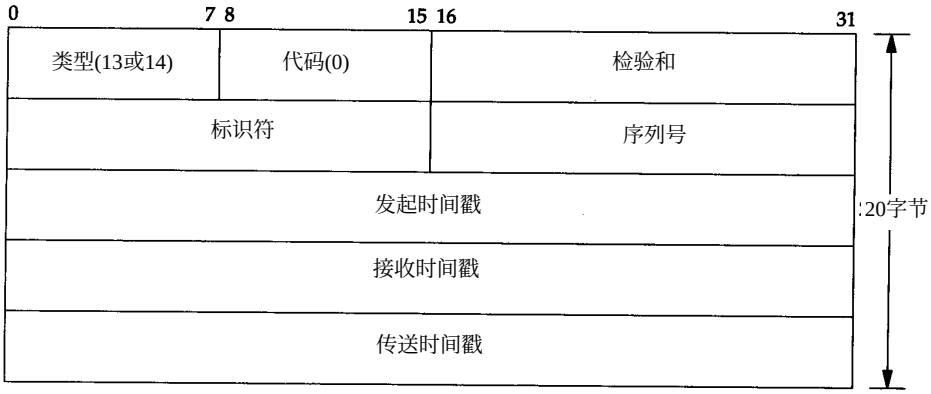


图6-6 ICMP时间戳请求和应答报文

Linux公社 www.linuxidc.com

请求端填写发起时间戳，然后发送报文。应答系统收到请求报文时填写接收时间戳，在发送应答时填写发送时间戳。但是，实际上，大多数的实现把后面两个字段都设成相同的值（提供三个字段的原因是可以让发送方分别计算发送请求的时间和发送应答的时间）。

6.4.1 举例

我们可以写一个简单程序（取名为 icmptime），给某个主机发送ICMP时间戳请求，并打印出返回的应答。它在我们的小互联网上运行结果如下：

```
sun % icmptime bsdi
orig = 83573336, recv = 83573330, xmit = 83573330, rtt = 2 ms
difference = -6 ms

sun % icmptime bsdi
orig = 83577987, recv = 83577980, xmit = 83577980, rtt = 2 ms
difference = -7 ms
```

程序打印出ICMP报文中的三个时间戳：发起时间戳（orig）、接收时间戳（recv）以及发送时间戳（xmit）。正如我们在这个例子以及下面的例子中所看到的那样，所有的主机把接收时间戳和发送时间戳都设成相同的值。

我们还能计算出往返时间（rtt），它的值是收到应答时的时间值减去发送请求时的时间值。difference的值是接收时间戳值减去发起时间戳值。这些值之间的关系如图6-7所示。

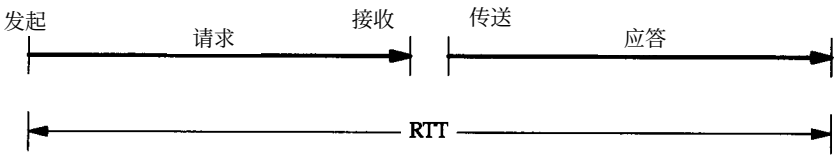


图6-7 icmptime 程序输出的值之间的关系

如果我们相信RTT的值，并且相信RTT的一半用于请求报文的传输，另一半用于应答报文的传输，那么为了使本机时钟与查询主机的时钟一致，本机时钟需要进行调整，调整值是difference减去RTT的一半。在前面的例子中，bsdi的时钟比sun的时钟要慢7 ms和8 ms。

由于时间戳的值是自午夜开始计算的毫秒数，即UTC，因此它们的值始终小于86 400 000 (24×60×60×1000)。这些例子都是在下午4:00以前运行的，并且在一个比UTC慢7个小时的时区，因此它们的值比82 800 000（2300小时）要大是有道理的。

如果对主机bsdi重复运行该程序数次，我们发现接收时间戳和发送时间戳的最后一位数总是0。这是因为该版本的软件（0.9.4版）只能提供10ms的时间分辨率（说明参见附录B）。

如果对主机svr4运行该程序两次，我们发现SVR4时间戳的最后三位数始终为0：

```
sun % icmptime svr4
orig = 83588210, recv = 83588000, xmit = 83588000, rtt = 4 ms
difference = -210 ms

sun % icmptime svr4
orig = 83591547, recv = 83591000, xmit = 83591000, rtt = 4 ms
difference = -547 ms
```

由于某种原因，SVR4在ICMP时间戳中不提供毫秒级的分辨率。这样，对秒以下的时间差调整将不起任何作用。

如果我们对子网140.252.1上的其他主机运行该程序，结果表明其中一台主机的时钟与

Linux公社 www.linuxidc.com

sun相差3.7秒，而另一个主机时钟相差近75秒：

```
sun % icmptime gemini
orig = 83601883, recv = 83598140, xmit = 83598140, rtt = 247 ms
difference = -3743 ms

sun % icmptime aix
orig = 83606768, recv = 83532183, xmit = 83532183, rtt = 253 ms
difference = -74585 ms
```

另一个令人感兴趣的例子是路由器gateway（一个Cisco路由器）。它表明，当系统返回一个非标准时间戳值时（不是自午夜开始计算的毫秒数， UTC），它就用32 bit时间戳中的高位来表示。我们的程序证明了一点，在尖括号中打印出了接收和发送的时间戳值（在关闭高位之后）。另外，不能计算发起时间戳和接收时间戳之间的时间差，因为它们的单位不一致。

```
sun % icmptime gateway
orig = 83620811, recv = <4871036>, xmit = <4871036>, rtt = 220 ms

sun % icmptime gateway
orig = 83641007, recv = <4891232>, xmit = <4891232>, rtt = 213 ms
```

如果我们在这台主机上运行该程序数次，会发现时间戳值显然具有毫秒级的分辨率，而且是从某个起始点开始计算的毫秒数，但是起始点并不是午夜 UTC（例如，可能是从路由器引导时开始计数的毫秒数）。

作为最后一个例子，我们来比较sun主机和另一个已知是准确的系统时钟 —— 一个NTP stratum 1服务器（下面我们会更多地讨论NTP，网络时间协议）。

```
sun % icmptime clock.llnl.gov
orig = 83662791, recv = 83662919, xmit = 83662919, rtt = 359 ms
difference = 128 ms

sun % icmptime clock.llnl.gov
orig = 83670425, recv = 83670559, xmit = 83670559, rtt = 345 ms
difference = 134 ms
```

如果我们把difference的值减去RTT的一半，结果表明sun主机上的时钟要快38.5～51.5 ms。

6.4.2 另一种方法

还可以用另一种方法来获得时间和日期。

1) 在1.12节中描述了日期服务程序和时间服务程序。前者是以人们可读的格式返回当前的时间和日期，是一行ASCII字符。可以用telnet命令来验证这个服务：

```
sun % telnet bsdi daytime
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.          前三行是Telnet客户的输出
Wed Feb  3 16:38:33 1993           这是日期时间服务器的输出
Connection closed by foreign host. 这也是Telnet客户的输出
```

另一方面，时间服务程序返回的是一个32bit的二制进数值，表示自UTC，1900年1月1日午夜起算的秒数。这个程序是以秒为单位提供的日期和时间（前面我们提过的 rdate命令使用的是TCP时间服务程序）。

2) 严格的计时器使用网络时间协议（ NTP），该协议在RFC 1305中给出了描述 [Mills 1992]。这个协议采用先进的技术来保证 LAN或WAN上的一组系统的时钟误差在毫秒级以内。对计算机精确时间感兴趣的读者应该阅读这份 RFC文档。

3) 开放软件基金会（OSF）的分布式计算环境（DCE）定义了分布式时间服务（DTS），

Linux公社 www.linuxidc.com

它也提供计算机之间的时钟同步。文献 [Rosenberg, Kenney and Fisher 1992]提供了该服务的其他细节描述。

4) 伯克利大学的Unix系统提供守护程序 timed(8)，来同步局域网上的系统时钟。不像NTP和DTS，timed不在广域网范围内工作。

6.5 ICMP端口不可达差错

最后两小节我们来讨论ICMP查询报文 —— 地址掩码和时间戳查询及应答。现在来分析一种ICMP差错报文，即端口不可达报文，它是 ICMP目的不可到达报文中的一种，以此来看一看ICMP差错报文中所附加的信息。使用UDP（见第11章）来查看它。

UDP的规则之一是，如果收到一份UDP数据报而目的端口与某个正在使用的进程不相符，那么UDP返回一个ICMP不可达报文。可以用TFTP来强制生成一个端口不可达报文（TFTP将在第15章描述）。

对于TFTP服务器来说， UDP的公共端口号是 69。但是大多数的 TFTP客户程序允许用connect命令来指定一个不同的端口号。这里，我们就用它来指定 8888端口：

```
bsdi % tftp
tftp> connect svr4 8888          指定主机名和端口号
tftp> get temp.foo               试图得到一个文件
Transfer timed out.              大约25秒后
tftp> quit
```

connect命令首先指定要连接的主机名及其端口号，接着用 get命令来取文件。敲入 get命令后，一份UDP数据报就发送到主机svr4上的8888端口。tcpdump命令引起的报文交换结果如图6-8所示。

```
 1   0.0                  arp who-has svr4 tell bsdi
 2   0.002050 (0.0020)    arp reply svr4 is-at 0:0:c0:c2:9b:26

 3   0.002723 (0.0007)    bsdi.2924 > svr4.8888: udp 20
 4   0.006399 (0.0037)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 5   5.000776 (4.9944)    bsdi.2924 > svr4.8888: udp 20
 6   5.004304 (0.0035)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 7  10.000887 (4.9966)    bsdi.2924 > svr4.8888: udp 20
 8  10.004416 (0.0035)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 9  15.001014 (4.9966)    bsdi.2924 > svr4.8888: udp 20
10  15.004574 (0.0036)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

11  20.001177 (4.9966)    bsdi.2924 > svr4.8888: udp 20
12  20.004759 (0.0036)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable
```

图6-8 由TFTP产生的ICMP端口不可达差错

在UDP数据报送到svr4之前，要先发送一份ARP请求来确定它的硬件地址（第1行）。接着返回ARP应答（第2行），然后才发送UDP数据报（第3行）（在tcpdump的输出中保留ARP请求和应答是为了提醒我们，这些报文交换可能在第一个 IP数据报从一个主机发送到另一个主机之前是必需的。在本书以后的章节中，如果这些报文与讨论的题目不相关，那么我们将省略它们）。

一个ICMP端口不可达差错是立刻返回的（第 4行）。但是， TFTP客户程序看上去似乎忽略了这个ICMP报文，而在5秒钟之后又发送了另一份 UDP数据报（第5行）。在客户程序放弃

Linux公社 www.linuxidc.com

之前重发了三次。

注意，ICMP报文是在主机之间交换的，而不用目的端口号，而每个 20字节的UDP数据报则是从一个特定端口（2924）发送到另一个特定端口（8888）。

跟在每个UDP后面的数字20指的是UDP数据报中的数据长度。在这个例子中， 20字节包括TFTP的2个字节的操作代码， 9个字节以空字符结束的文件名 `temp.foo`，以及9个字节以空字符结束的字符串`netascii`（TFTP报文的详细格式参见图15-1）。

如果用-e选项运行同样的例子，我们可以看到每个返回的 ICMP端口不可达报文的完整长度。这里的长度为70字节，各字段分配如图6-9所示。

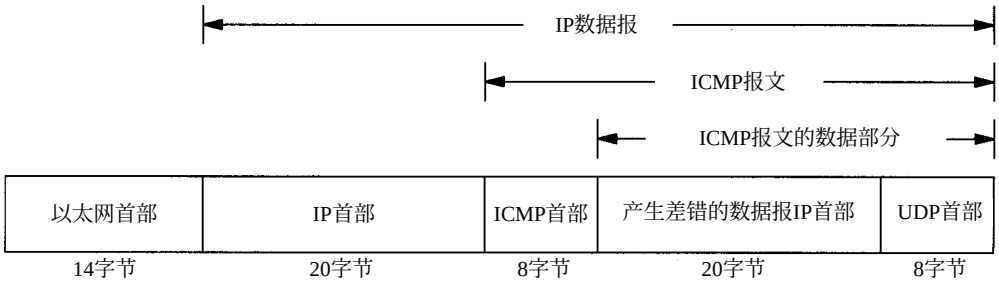


图6-9 “UDP端口不可达”例子中返回的ICMP报文

ICMP的一个规则是， ICMP差错报文（参见图6-3的最后一列）必须包括生成该差错报文的数据报IP首部（包含任何选项），还必须至少包括跟在该 IP首部后面的前8个字节。在我们的例子中，跟在IP首部后面的前8个字节包含UDP的首部（见图11-2）。

一个重要的事实是包含在 UDP首部中的内容是源端口号和目的端口号。就是由于目的端口号（8888）才导致产生了ICMP端口不可达的差错报文。接收 ICMP的系统可以根据源端口号（2924）来把差错报文与某个特定的用户进程相关联（在本例中是 TFTP客户程序）。

导致差错的数据报中的 IP首部要被送回的原因是因为 IP首部中包含了协议字段，使得ICMP可以知道如何解释后面的 8个字节（在本例中是 UDP首部）。如果我们来查看 TCP首部（图17-2）,可以发现源端口和目的端口被包含在 TCP首部的前8个字节中。

ICMP不可达报文的一般格式如图 6-10所示。

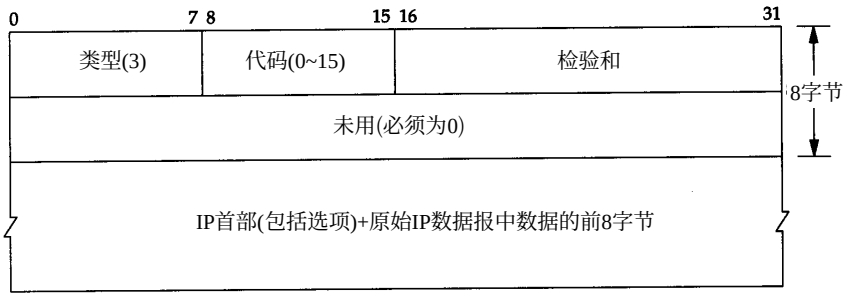


图6-10 ICMP不可达报文

在图6-3中，我们注意到有16种不同类型的ICMP不可达报文，代码分别从0到15。ICMP端口不可达差错代码是3。另外，尽管图6-10指出了在ICMP报文中的第二个32 bit字必须为0，但是当代码为4时（“需要分片但设置了不分片比特”），路径MTU发现机制（2.9节）却允许路由器把外

Linux公社 www.linuxidc.com

出接口的MTU填在这个32 bit字的低16 bit中。我们在11.6节中给出了一个这种差错的例子。

尽管ICMP规则允许系统返回多于8个字节的产生差错的IP数据报中的数据，但是大多数从伯克利派生出来的系统只返回 8个字节。Solaris 2.2的ip_icmp_return_data_bytes选项默认条件下返回前64个字节（E.4节）。

tcpdump时间系列

在本书的后面章节中，我们还要以时间系列的格式给出tcpdump命令的输出，如图6-11所示。

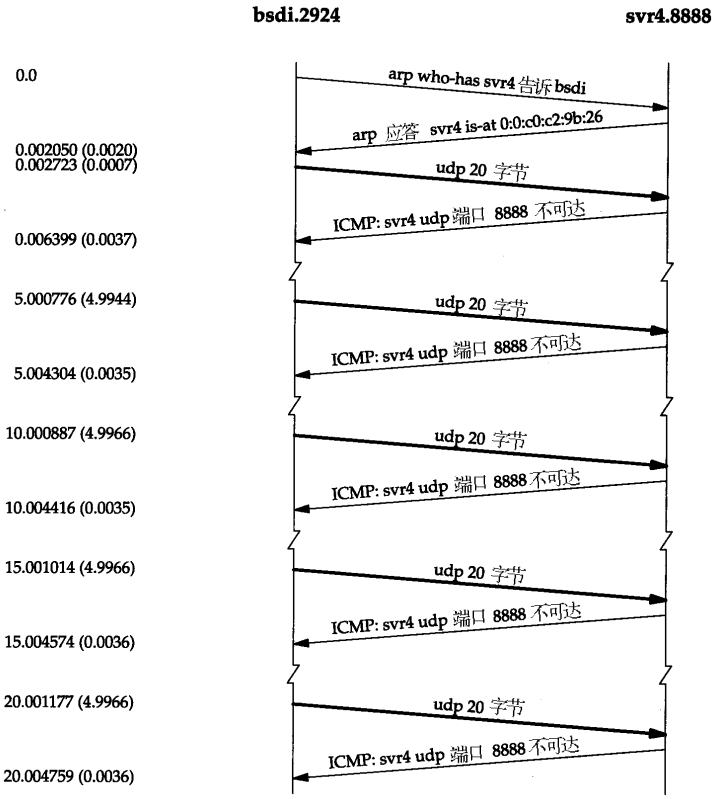


图6-11 发送到无效端口的TFTP请求的时间系列

时间随着向下而递增，在图左边的时间标记与tcpdump命令的输出是相同的（见图6-8）。位于图顶部的标记是通信双方的主机名和端口号。需要指出的是，随着页面向下的y坐标轴与真正的时间值不是成比例的。当出现一个有意义的时间段时，在本例中是每5秒之间的重发，我们就在时间系列的两侧作上标记。当UDP或TCP数据正在被传送时，我们用粗线的行来表示。

当ICMP报文返回时，为什么TFTP客户程序还要继续重发请求呢？这是由于网络编程中的一个因素，即BSD系统不把从插口(socket)接收到的ICMP报文中的UDP数据通知用户进程，除非该进程已经发送了一个connect命令给该插口。标准的BSD TFTP客户程序并不发送connect命令，因此它永远也不会收到ICMP差错报文的通知。

这里需要注意的另一点是TFTP客户程序所采用的不太好的超时重传算法。它只是假定5秒是足够的，因此每隔5秒就重传一次，总共需要25秒钟的时间。在后面我们将看到TCP有一个较好的超时重发算法。

Linux公社 www.linuxidc.com

TFTP客户程序所采用的超时重传算法已被RFC所禁用。不过，在作者所在子网上的三个系统以及Solaris 2.2仍然在使用它。AIX 3.2.2采用一种指数退避方法来设置超时值，分别在0、5、15和35秒时重发报文，这正是所推荐的方法。我们将在第21章更详细地讨论超时问题。

最后需要指出的是，ICMP报文是在发送UDP数据报3.5 ms后返回的，这与第7章我们所看到的Ping应答的往返时间差不多。

6.6 ICMP报文的4.4BSD处理

由于ICMP覆盖的范围很广，从致命差错到信息差错，因此即使在一个给定的系统实现中，对每个ICMP报文的处理都是不相同的。图6-12的内容与图6-3相同，它显示的是4.4BSD系统对每个可能的ICMP报文的处理方法。

类型	代码	描述	处理方法
0	0	回显应答	用户进程
3		目的不可达：	
	0	网络不可达	“无路由到达主机”
	1	主机不可达	“无路由到达主机”
	2	协议不可达	“连接被拒绝”
	3	端口不可达	“连接被拒绝”
	4	需要进行分片但设置了不分片比特DF	“报文太长”
	5	源站选路失败	“无路由到达主机”
	6	目的网络不认识	“无路由到达主机”
	7	目的主机不认识	“无路由到达主机”
	8	源主机被隔离（作废不用）	“无路由到达主机”
	9	目的网络被强制禁止	“无路由到达主机”
	10	目的主机被强制禁止	“无路由到达主机”
	11	由于服务类型TOS，网络不可达	“无路由到达主机”
	12	由于服务类型TOS，主机不可达	“无路由到达主机”
	13	由于过滤，通信被强制禁止	（忽略）
	14	主机越权	（忽略）
	15	优先权中止生效	（忽略）
4	0	源站被抑制(quench)	TCP由内核处理，UDP则忽略
5		重定向	
	0	对网络重定向	内核更新路由表
	1	对主机重定向	内核更新路由表
	2	对服务类型和网络重定向	内核更新路由表
	3	对服务类型和主机重定向	内核更新路由表
8	0	回显请求	
9	0	路由器通告	用户进程
10	0	路由器请求	用户进程
11		超时：	
	0	传输期间生存时间为0	用户进程
	1	在数据报组装期间生存时间为0	用户进程
12		参数问题：	
	0	坏的IP首部（包括各种差错）	“协议不可用”
	1	缺少必需的选项	“协议不可用”
13	0	时间戳请求	内核产生应答
14	0	时间戳应答	用户进程
15	0	信息请求（作废不用）	（忽略）
16	0	信息应答（作废不用）	用户进程
17	0	地址掩码请求	内核产生应答
18	0	地址掩码应答	用户进程

图6-12 4.4BSD系统对ICMP报文的处理

Linux公社 www.linuxidc.com

如果最后一列标明是“内核”，那么ICMP就由内核来处理。如果最后一列指明是“用户进程”，那么报文就被传送到所有在内核中登记的用户进程，以读取收到的 ICMP报文。如果不存在任何这样的用户进程，那么报文就悄悄地被丢弃（这些用户进程还会收到所有其他类型的ICMP报文的拷贝，虽然它们应该由内核来处理，当然用户进程只有在内核处理以后才能收到这些报文）。有一些报文完全被忽略。最后，如果最后一列标明的是引号内的一串字符，那么它就是对应的Unix差错。其中一些差错，如TCP对发送端关闭的处理等，我们将在以后的章节中对它们进行讨论。

6.7 小结

本章对每个系统都必须包括的 Internet控制报文协议进行了讨论。图 6-3列出了所有的ICMP报文类型，其中大多数都将在以后的章节中加以讨论。

我们详细讨论了ICMP地址掩码请求和应答以及时间戳请求和应答。这些是典型的请求—应答报文。二者在ICMP报文中都有标识符和序列号。发送端应用程序在标识字段内存入一个唯一的数值，以区别于其他进程的应答。序列号字段使得客户程序可以在应答和请求之间进行匹配。

我们还讨论了ICMP端口不可达差错，一种常见的ICMP差错。对返回的ICMP差错信息进行了分析：导致差错的IP数据报的首部及后续8个字节。这个信息对于ICMP差错的接收方来说是必要的，可以更多地了解导致差错的原因。这是因为 TCP和UDP都在它们的首部前8个字节中存入源端口号和目的端口号。

最后，我们第一次给出了按时间先后的 `tcpdump`输出，这种表示方式在本书后面的章节中会经常用到。

习题

6.1 在6.2节的末尾，我们列出了5种不发送ICMP差错报文的特殊条件。如果这些条件不满足而我们又在局域网上向一个似乎不存在的端口号发送一份广播 UDP数据报，这时会发生什么样的情况？

6.2 阅读RFC [Braden 1989a]，注意生成一个ICMP端口不可达差错是否为“必须”，“应该”或者“可能”。这些信息所在的页码和章节是多少？

6.3 阅读RFC 1349 [Almquist 1992]，看看IP的服务类型字段（见图3-2）是如何被ICMP设置的？

6.4 如果你的系统提供`netstat`命令，请用它来查看接收和发送的ICMP报文类型。

Linux公社 www.linuxidc.com

第7章　Ping程序

7.1　引言

“ping”这个名字源于声纳定位操作。Ping程序由Mike Muuss编写，目的是为了测试另一台主机是否可达。该程序发送一份ICMP回显请求报文给主机，并等待返回ICMP回显应答（图6-3列出了所有的ICMP报文类型）。

一般来说，如果不能Ping到某台主机，那么就不能Telnet或者FTP到那台主机。反过来，如果不能Telnet到某台主机，那么通常可以用Ping程序来确定问题出在哪里。Ping程序还能测出到这台主机的往返时间，以表明该主机离我们有“多远”。

在本章中，我们将使用Ping程序作为诊断工具来深入剖析ICMP。Ping还给我们提供了检测IP记录路由和时间戳选项的机会。文献[Stevens 1990]的第11章提供了Ping程序的源代码。

> 几年前我们还可以作出这样没有限定的断言，如果不能Ping到某台主机，那么就不能Telnet或FTP到那台主机。随着Internet安全意识的增强，出现了提供访问控制清单的路由器和防火墙，那么像这样没有限定的断言就不再成立了。一台主机的可达性可能不只取决于IP层是否可达，还取决于使用何种协议以及端口号。Ping程序的运行结果可能显示某台主机不可达，但我们可以用Telnet远程登录到该台主机的25号端口（邮件服务器）。

7.2　Ping程序

我们称发送回显请求的ping程序为客户，而称被ping的主机为服务器。大多数的TCP/IP实现都在内核中直接支持Ping服务器——这种服务器不是一个用户进程（在第6章中描述的两种ICMP查询服务，地址掩码和时间戳请求，也都是直接在内核中进行处理的）。

ICMP回显请求和回显应答报文如图7-1所示。

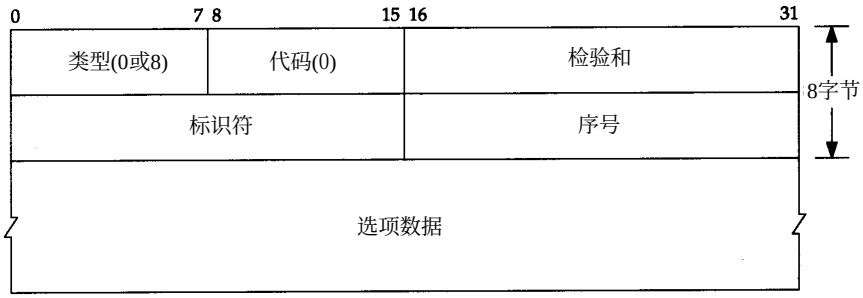


图7-1　ICMP回显请求和回显应答报文格式

对于其他类型的ICMP查询报文，服务器必须响应标识符和序列号字段。另外，客户发送的选项数据必须回显，假设客户对这些信息都会感兴趣。

Linux公社 www.linuxidc.com

Unix系统在实现ping程序时是把ICMP报文中的标识符字段置成发送进程的 ID号。这样即使在同一台主机上同时运行了多个ping程序实例，ping程序也可以识别出返回的信息。

序列号从0开始，每发送一次新的回显请求就加 1。ping程序打印出返回的每个分组的序列号，允许我们查看是否有分组丢失、失序或重复。 IP是一种最好的数据报传递服务，因此这三个条件都有可能发生。

旧版本的ping程序曾经以这种模式运行，即每秒发送一个回显请求，并打印出返回的每个回显应答。但是，新版本的实现需要加上- s选项才能以这种模式运行。默认情况下，新版本的ping程序只发送一个回显请求。如果收到回显应答，则输出“ host is alive ”；否则，在20秒内没有收到应答就输出“ no answer（没有回答）”。

7.2.1 LAN输出

在局域网上运行ping程序的结果输出一般有如下格式：

```
bsdi % ping svr4
PING svr4 (140.252.13.34): 56 data bytes
64 bytes from 140.252.13.34: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=2 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=3 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=4 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=5 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=6 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=7 ttl=255 time=0 ms
^?                                          键入中断键来停止显示
--- svr4 ping statistics ---
8 packets transmitted, 8 packets received, 0% packet loss
round-trip min/avg/max = 0/0/0 ms
```

当返回ICMP回显应答时，要打印出序列号和 TTL，并计算往返时间（TTL位于IP首部中的生存时间字段。当前的 BSD系统中的ping程序每次收到回显应答时都打印出收到的 TTL——有些系统并不这样做。我们将在第 8章中通过traceroute程序来介绍TTL的用法）。

从上面的输出中可以看出，回显应答是以发送的次序返回的（ 0，1，2等）。

ping程序通过在ICMP报文数据中存放发送请求的时间值来计算往返时间。当应答返回时，用当前时间减去存放在 ICMP报文中的时间值，即是往返时间。注意，在发送端 bsdi上，往返时间的计算结果都为 0 ms。这是因为程序使用的计时器分辨率低的原因。 BSD/386版本0.9.4系统只能提供10 ms级的计时器（在附录B中有更详细的介绍）。在后面的章节中，当我们在具有较高分辨率计时器的系统上（ Sun）查看tcpdump输出时会发现， ICMP回显请求和回显应答的时间差在4 ms以下。

输出的第一行包括目的主机的 IP地址，尽管指定的是它的名字（ svr4）。这说明名字已经经过解析器被转换成 IP地址了。我们将在第 14章介绍解析器和DNS。现在，我们发现，如果敲入ping命令，几秒钟过后会在第 1行打印出IP地址， DNS就是利用这段时间来确定主机名所对应的IP地址。

本例中的tcpdump输出如图7-2所示。

从发送回显请求到收到回显应答，时间间隔始终为 3.7 ms。还可以看到，回显请求大约每隔1秒钟发送一次。

通常，第1个往返时间值要比其他的大。这是由于目的端的硬件地址不在 ARP高速缓存中

Linux公社 www.linuxidc.com

```
 1  0.0                  bsdi > svr4: icmp: echo request
 2  0.003733 (0.0037)    svr4 > bsdi: icmp: echo reply

 3  0.998045 (0.9943)    bsdi > svr4: icmp: echo request
 4  1.001747 (0.0037)    svr4 > bsdi: icmp: echo reply

 5  1.997818 (0.9961)    bsdi > svr4: icmp: echo request
 6  2.001542 (0.0037)    svr4 > bsdi: icmp: echo reply

 7  2.997610 (0.9961)    bsdi > svr4: icmp: echo request
 8  3.001311 (0.0037)    svr4 > bsdi: icmp: echo reply

 9  3.997390 (0.9961)    bsdi > svr4: icmp: echo request
10  4.001115 (0.0037)    svr4 > bsdi: icmp: echo reply

11  4.997201 (0.9961)    bsdi > svr4: icmp: echo request
12  5.000904 (0.0037)    svr4 > bsdi: icmp: echo reply

13  5.996977 (0.9961)    bsdi > svr4: icmp: echo request
14  6.000708 (0.0037)    svr4 > bsdi: icmp: echo reply

15  6.996764 (0.9961)    bsdi > svr4: icmp: echo request
16  7.000479 (0.0037)    svr4 > bsdi: icmp: echo reply
```

图7-2 在LAN上运行ping程序的结果

的缘故。正如我们在第4章中看到的那样，在发送第一个回显请求之前要发送一个 ARP请求并接收ARP应答，这需要花费几毫秒的时间。下面的例子说明了这一点：

```
sun % arp -a                                  保证ARP高速缓存是空的

sun % ping svr4
PING svr4: 56 data bytes
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=7. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=4. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=2. time=4. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=3. time=4. ms
^?                                            键入中断键来停止显示
----svr4 PING Statistics----
4 packets transmitted, 4 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 4/4/7
```

第1个RTT中多出的3 ms很可能就是因为发送ARP请求和接收ARP应答所花费的时间。

这个例子运行在sun主机上，它提供的是具有微秒级分辨率的计时器，但是 ping程序只能打印出毫秒级的往返时间。在前面运行于 BSD/386 0.9.4版上的例子中，打印出来的往返时间值为0 ms，这是因为计时器只能提供10 ms的误差。下面的例子是BSD/386 1.0版的输出，它提供的计时器也具有微秒级的分辨率，因此， ping程序的输出结果也具有较高的分辨率。

```
bsdi % ping svr4
PING svr4 (140.252.13.34): 56 data bytes
64 bytes from 140.252.13.34: icmp_seq=0 ttl=255 time=9.304 ms
64 bytes from 140.252.13.34: icmp_seq=1 ttl=255 time=6.089 ms
64 bytes from 140.252.13.34: icmp_seq=2 ttl=255 time=6.079 ms
64 bytes from 140.252.13.34: icmp_seq=3 ttl=255 time=6.096 ms
^?                                            键入中断键来停止显示
--- svr4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 6.079/6.880/9.304 ms
```

7.2.2 WAN输出

在一个广域网上，结果会有很大的不同。下面的例子是在某个工作日的下午即 Internet具

Linux公社 www.linuxidc.com

有正常通信量时的运行结果：

```
gemini % ping vangogh.cs.berkeley.edu
PING vangogh.cs.berkeley.edu: 56 data bytes
64 bytes from (128.32.130.2): icmp_seq=0. time=660. ms
64 bytes from (128.32.130.2): icmp_seq=5. time=1780. ms
64 bytes from (128.32.130.2): icmp_seq=7. time=380. ms
64 bytes from (128.32.130.2): icmp_seq=8. time=420. ms
64 bytes from (128.32.130.2): icmp_seq=9. time=390. ms
64 bytes from (128.32.130.2): icmp_seq=14. time=110. ms
64 bytes from (128.32.130.2): icmp_seq=15. time=170. ms
64 bytes from (128.32.130.2): icmp_seq=16. time=100. ms
^?                                    键入中断来停止显示
----vangogh.CS.Berkeley.EDU PING Statistics----
17 packets transmitted, 8 packets received, 52% packet loss
round-trip (ms)  min/avg/max = 100/501/1780
```

这里，序列号为1、2、3、4、6、10、11、12和13的回显请求或回显应答在某个地方丢失了。另外，我们注意到往返时间发生了很大的变化（像 52%这样高的分组丢失率是不正常的。即使是在工作日的下午，对于Internet来说也是不正常的）。

通过广域网还有可能看到重复的分组（即相同序列号的分组被打印两次或更多次），失序的分组（序列号为$N+1$的分组在序列号为N的分组之前被打印）。

7.2.3 线路SLIP链接

让我们再来看看SLIP链路上的往返时间，因为它们经常运行于低速的异步方式，如 9600 b/s或更低。回想我们在 2.10节计算的串行线路吞吐量。针对这个例子，我们把主机 bsdi和slip之间的SLIP链路传输速率设置为1200 b/s。

下面我们可以来估计往返时间。首先，从前面的 Ping程序输出例子中可以注意到，默认情况下发送的ICMP报文有56个字节。再加上20个字节的IP首部和8个字节的ICMP首部，IP数据报的总长度为84字节（我们可以运行 tcpdump-e命令查看以太网数据帧来验证这一点）。另外，从2.4节可以知道，至少要增加两个额外的字节：在数据报的开始和结尾加上 END字符。此外，SLIP帧还有可能再增加一些字节，但这取决于数据报中每个字节的值。对于 1200 b/s这个速率来说，由于每个字节含有 8 bit数据、1 bit起始位和1 bit结束位，因此传输速率是每秒120个字节，或者说每个字节 8.33 ms。所以我们可以估计需要 1433（86×8.33×2）ms（乘2是因为我们计算的是往返时间）。

下面的输出证实了我们的计算：

```
svr4 % ping -s slip
PING slip: 56 data bytes
64 bytes from slip (140.252.13.65): icmp_seq=0. time=1480. ms
64 bytes from slip (140.252.13.65): icmp_seq=1. time=1480. ms
64 bytes from slip (140.252.13.65): icmp_seq=2. time=1480. ms
64 bytes from slip (140.252.13.65): icmp_seq=3. time=1480. ms
^?
----slip PING Statistics----
5 packets transmitted, 4 packets received, 20% packet loss
round-trip (ms)  min/avg/max = 1480/1480/1480
```

（对于SVR4来说，如果每秒钟发送一次请求则必须带- s选项）。往返时间大约是 1.5秒，但是程序仍然每间隔 1秒钟发送一次 ICMP回显请求。这说明在第 1个回显应答返回之前（1.480秒时刻）就已经发送了两次回显请求（分别在 0秒和1秒时刻）。这就是为什么总结行指

Linux公社 www.linuxidc.com

出丢失了一个分组。实际上分组并未丢失，很可能仍然在返回的途中。

我们在第8章讨论traceroute程序时将回头再讨论这种低速的SLIP链路。

7.2.4 拨号SLIP链路

对于拨号SLIP链路来说，情况有些变化，因为在链路的两端增加了调制解调器。用在sun和netb系统之间的调制解调器提供的是V.32调制方式（9600 b/s）、V.42错误控制方式（也称作LAP-M）以及V.42bis数据压缩方式。这表明我们针对线路链路参数进行的简单计算不再准确了。

很多因素都有可能影响。调制解调器带来了时延。随着数据的压缩，分组长度可能会减小，但是由于使用了错误控制协议，分组长度又可能会增加。另外，接收端的调制解调器只能在验证了循环检验字符（检验和）后才能释放收到的数据。最后，我们还要处理每一端的计算机异步串行接口，许多操作系统只能在固定的时间间隔内，或者收到若干字符后才去读这些接口。

作为一个例子，我们在sun主机上ping主机gemini，输出结果如下：

```
sun % ping gemini
PING gemini: 56 data bytes
64 bytes from gemini (140.252.1.11): icmp_seq=0. time=373. ms
64 bytes from gemini (140.252.1.11): icmp_seq=1. time=360. ms
64 bytes from gemini (140.252.1.11): icmp_seq=2. time=340. ms
64 bytes from gemini (140.252.1.11): icmp_seq=3. time=320. ms
64 bytes from gemini (140.252.1.11): icmp_seq=4. time=330. ms
64 bytes from gemini (140.252.1.11): icmp_seq=5. time=310. ms
64 bytes from gemini (140.252.1.11): icmp_seq=6. time=290. ms
64 bytes from gemini (140.252.1.11): icmp_seq=7. time=300. ms
64 bytes from gemini (140.252.1.11): icmp_seq=8. time=280. ms
64 bytes from gemini (140.252.1.11): icmp_seq=9. time=290. ms
64 bytes from gemini (140.252.1.11): icmp_seq=10. time=300. ms
64 bytes from gemini (140.252.1.11): icmp_seq=11. time=280. ms
----gemini PING Statistics----
12 packets transmitted, 12 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 280/314/373
```

注意，第1个RTT不是10 ms的整数倍，但是其他行都是10 ms的整数倍。如果我们运行该程序若干次，发现每次结果都是这样（这并不是由sun主机上的时钟分辨率造成的结果，因为根据附录B中的测试结果可以知道它的时钟能提供毫秒级的分辨率）。

另外还要注意，第1个RTT要比其他的大，而且依次递减，然后徘徊在280～300 ms之间。我们让它运行1~2分钟，RTT一直处于这个范围，不会低于260 ms。如果我们以9600 b/s的速率计算RTT（习题7.2），那么观察到的值应该大约是估计值的1.5倍。

如果运行ping程序60秒钟并计算观察到的RTT的平均值，我们发现在V.42和V.42bis模式下平均值为277 ms（这比上个例子打印出来的平均值要好，因为运行时间较长，这样就把开始较长的时间平摊了）。如果我们关闭V.42bis数据压缩方式，平均值为330 ms。如果我们关闭V.42错误控制方式（它同时也关闭了V.42bis数据压缩方式），平均值为300 ms。这些调制解调器的参数对RTT的影响很大，使用错误控制和数据压缩方式似乎效果最好。

7.3 IP记录路由选项

ping程序为我们提供了查看IP记录路由（RR）选项的机会。大多数不同版本的ping程

Linux公社 www.linuxidc.com

序都提供-R选项，以提供记录路由的功能。它使得ping程序在发送出去的IP数据报中设置IP RR选项（该IP数据报包含ICMP回显请求报文）。这样，每个处理该数据报的路由器都把它的IP地址放入选项字段中。当数据报到达目的端时，IP地址清单应该复制到ICMP回显应答中，这样返回途中所经过的路由器地址也被加入清单中。当ping程序收到回显应答时，它就打印出这份IP地址清单。

这个过程听起来简单，但存在一些缺陷。源端主机生成RR选项，中间路由器对RR选项的处理，以及把ICMP回显请求中的RR清单复制到ICMP回显应答中，所有这些都是选项功能。幸运的是，现在的大多数系统都支持这些选项功能，只是有一些系统不把ICMP请求中的IP清单复制到ICMP应答中。

但是，最大的问题是IP首部中只有有限的空间来存放IP地址。我们从图3-1可以看到，IP首部中的首部长度字段只有4 bit，因此整个IP首部最长只能包括15个32 bit长的字（即60个字节）。由于IP首部固定长度为20字节，RR选项用去3个字节（下面我们再讨论），这样只剩下37个字节（60-20-3）来存放IP地址清单，也就是说只能存放9个IP地址。对于早期的ARPANET来说，9个IP地址似乎是很多了，但是现在看来是非常有限的（在第8章中，我们将用Traceroute工具来确定数据报的路由）。除了这些缺点，记录路由选项工作得很好，为详细查看如何处理IP选项提供了一个机会。

IP数据报中的RR选项的一般格式如图7-3所示。

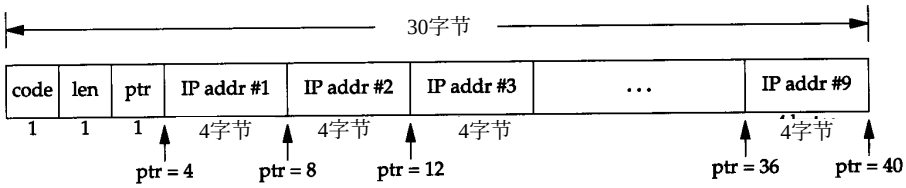


图7-3 IP首部中的记录路由选项的一般格式

code是一个字节，指明IP选项的类型。对于RR选项来说，它的值为7。len是RR选项总字节长度，在这种情况下为39（尽管可以为RR选项设置比最大长度小的长度，但是ping程序总是提供39字节的选项字段，最多可以记录9个IP地址。由于IP首部中留给选项的空间有限，它一般情况都设置成最大长度）。

ptr称作指针字段。它是一个基于1的指针，指向存放下一个IP地址的位置。它的最小值为4，指向存放第一个IP地址的位置。随着每个IP地址存入清单，ptr的值分别为8，12，16，最大到36。当记录下9个IP地址后，ptr的值为40，表示清单已满。

当路由器（根据定义应该是多穴的）在清单中记录IP地址时，它应该记录哪个地址呢？是入口地址还是出口地址？为此，RFC 791 [Postel 1981a]指定路由器记录出口IP地址。我们在后面将看到，当原始主机（运行ping程序的主机）收到带有RR选项的ICMP回显应答时，它也要把它的入口IP地址放入清单中。

7.3.1 通常的例子

我们举一个用RR选项运行ping程序的例子，在主机svr4上运行ping程序到主机slip。一个中间路由器(bsdi)将处理这个数据报。下面是svr4的输出结果：

Linux公社 www.linuxidc.com

```
svr4 % ping -R slip
PING slip (140.252.13.65): 56 data bytes
64 bytes from 140.252.13.65: icmp_seq=0 ttl=254 time=280 ms
RR:     bsdi (140.252.13.66)
        slip (140.252.13.65)
        bsdi (140.252.13.35)
        svr4 (140.252.13.34)
64 bytes from 140.252.13.65: icmp_seq=1 ttl=254 time=280 ms (same route)
64 bytes from 140.252.13.65: icmp_seq=2 ttl=254 time=270 ms (same route)
^?
--- slip ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 270/276/280 ms
```

分组所经过的四站如图7-4所示（每个方向各有两站），每一站都把自己的IP地址加入RR清单。

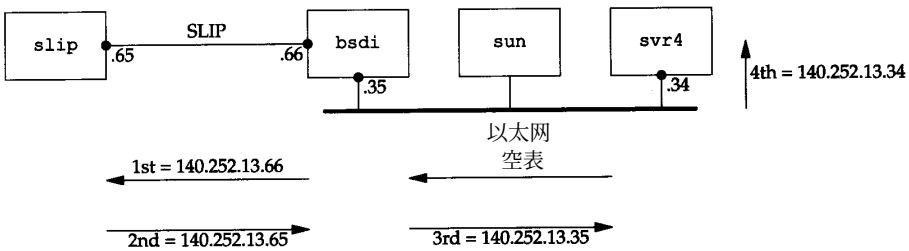


图7-4 带有记录路由选项的ping程序

路由器bsdi在不同方向上分别加入了不同的IP地址。它始终是把出口的IP地址加入清单。我们还可以看到，当ICMP回显应答到达原始系统（svr4）时，它把自己的入口IP地址也加入清单中。

还可以通过运行带有-v选项的tcpdump命令来查看主机sun上进行的分组交换（参见IP选项）。输出如图7-5所示。

```
1  0.0                  svr4 > slip: icmp: echo request (ttl 32, id 35835,
                        optlen=40 RR{39}= RR{#0.0.0.0/0.0.0.0/0.0.0.0/
                        0.0.0.0/ 0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0} EOL)

2  0.267746 (0.2677)    slip > svr4: icmp: echo reply (ttl 254, id 1976,
                        optlen=40 RR{39}= RR{140.252.13.66/140.252.13.65/
                        140.252.13.35/#0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0/
                        0.0.0.0/0.0.0.0} EOL)
```

图7-5 记录路由选项的tcpdump 输出

输出中optlen=40表示在IP首部中有40个字节的选项空间（IP首部长度必须为4字节的整数倍）。RR{39}的意思是记录路由选项已被设置，它的长度字段是39。然后是9个IP地址，符号“#”用来标记RR选项中的ptr字段所指向的IP地址。由于我们是在主机sun上观察这些分组（参见图7-4），因此所能看到ICMP回显请求中的IP地址清单是空的，而ICMP回显应答中有3个IP地址。我们省略了tcpdump输出中的其他行，因为它们与图7-5基本一致。

位于路由信息末尾的标记EOL表示IP选项“end of list（清单结束）”的值。EOL选项的值可以为0。这时表示39个字节的RR数据位于IP首部中的40字节空间中。由于在数据报发送之前空间选项被设置为0，因此跟在39个字节的RR数据之后的0字符就被解释为EOL。这正是我

Linux公社 www.linuxidc.com

们所希望的结果。如果在IP首部中的选项字段中有多个选项，在开始下一个选项之前必须填入空白字符，另外还可以用另一个值为1的特殊字符NOP（“no operation”）。

在图7-5中，SVR4把回显请求中的TTL字段设为32，BSD/386设为255（它打印出的值为254是因为路由器bsdi已经将其减去1）。新的系统都把ICMP报文中的TTL设为最大值（255）。

在作者使用的三个TCP/IP系统中，BSD/386和SVR4都支持记录路由选项。这就是说，当转发数据报时，它们都能正确地更新RR清单，而且能正确地把接收到的ICMP回显请求中的RR清单复制到出口ICMP回显应答中。虽然SunOS 4.1.3在转发一个数据报时能正确更新RR清单，但是不能复制RR清单。Solaris 2.x对这个问题已作了修改。

7.3.2 异常的输出

下面的例子是作者观察到的，把它作为第9章讨论ICMP间接报文的起点。在子网140.252.1上ping主机aix（在主机sun上通过拨号SLIP连接可以访问），并带有记录路由选项。在slip主机上运行有如下输出结果：

```
slip % ping -R aix
PING aix (140.252.1.92): 56 data bytes
64 bytes from 140.252.1.92: icmp_seq=0 ttl=251 time=650 ms
RR:     bsdi (140.252.13.35)
        sun (140.252.1.29)
        netb (140.252.1.183)
        aix (140.252.1.92)
        gateway (140.252.1.4)        为什么用这个路由器？
        netb (140.252.1.183)
        sun (140.252.13.33)
        bsdi (140.252.13.66)
        slip (140.252.13.65)
64 bytes from aix: icmp_seq=1 ttl=251 time=610 ms (same route)
64 bytes from aix: icmp_seq=2 ttl=251 time=600 ms (same route)
^?
--- aix ping statistics ---
4 packets transmitted, 3 packets received, 25% packet loss
round-trip min/avg/max = 600/620/650 ms
```

我们已经在主机bsdi上运行过这个例子。现在选择slip来运行它，观察RR清单中所有的9个IP地址。

在输出中令人感到疑惑的是，为什么传出的数据报（ICMP回显请求）直接从netb传到aix，而返回的数据报（ICMP回显应答）却从aix开始经路由器gateway再到netb？这里看到的正是下面将要描述的IP选路的一个特点。数据报经过的路由如图7-6所示。

问题是aix不知道要把目的地为子网140.252.13的IP数据报发到主机netb上。相反，aix在它的路由表中有一个默认项，它指明当没有明确某个目的主机的路由时，就把所有的数据报发往默认项指定的路由器gateway。路由器gateway比子网140.252.1上的任何主机都具备更强的选路能力（在这个以太网上有超过150台主机，每台主机的路由表中都有一个默认项指向路由器gateway，这样就不用在每台主机上都运行一个选路守护程序）。

这里没有应答的一个问题是为什么gateway不直接发送ICMP报文重定向到aix（9.5节），以更新它的路由表？由于某种原因（很可能是由于数据报产生的重定向是一份ICMP回显请求报文），重定向并没有产生。但是如果我们用Telnet登录到aix上的daytime服务器，ICMP就会

Linux公社 www.linuxidc.com

产生重定向，因而它在aix上的路由表也随之更新。如果接着执行ping程序并带有记录路由选项，其路由显示表明数据报从netb到aix，然后返回netb，而不再经过路由器gateway。在9.5节中将更详细地讨论ICMP重定向的问题。

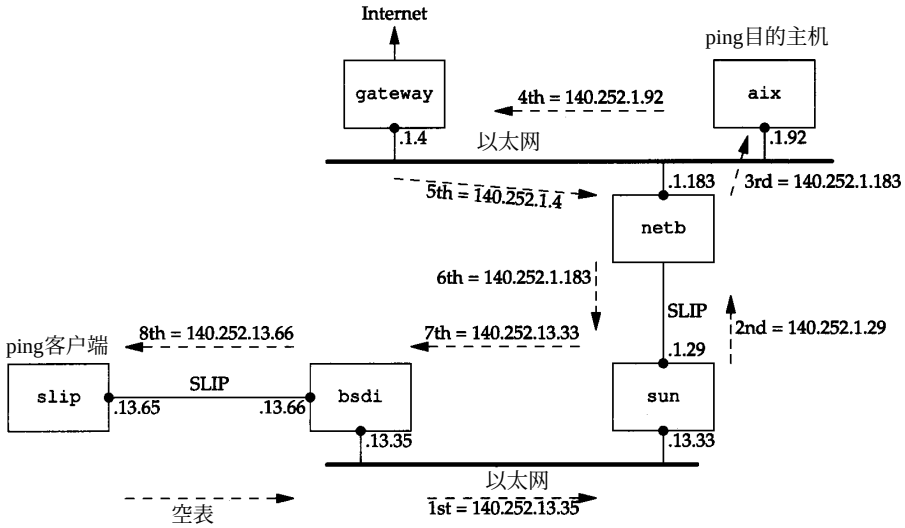


图7-6 运行带有记录路由选项的ping 程序，显示IP选路的特点

7.4 IP时间戳选项

IP时间戳选项与记录路由选项类似。IP时间戳选项的格式如图7-7所示（请与图7-3进行比较）。

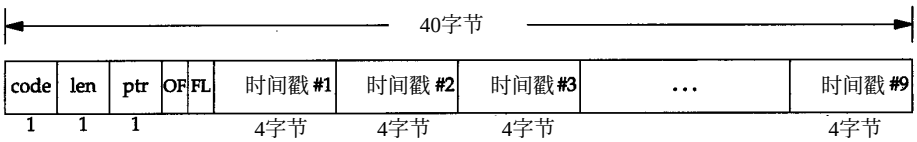


图7-7 IP首部中时间戳选项的一般格式

时间戳选项的代码为0x44。其他两个字段len和ptr与记录路由选项相同：选项的总长度（一般为36或40）和指向下一个可用空间的指针（5，9，13等）。

接下来的两个字段是4 bit的值：OF表示溢出字段，FL表示标志字段。时间戳选项的操作根据标志字段来进行，如图7-8所示。

标　志	描　述
0	只记录时间戳，正如我们在图7-7看到的那样
1	每台路由器都记录它的IP地址和时间戳。在选项列表中只有存放4对地址和时间戳的空间
3	发送端对选项列表进行初始化，存放了4个IP地址和4个取值为0的时间戳值。只有当列表中的下一个IP地址与当前路由器地址相匹配时，才记录它的时间戳

图7-8 时间戳选项不同标志字段值的意义

如果路由器由于没有空间而不能增加时间戳选项，那么它将增加溢出字段的值。

Linux公社 www.linuxidc.com

时间戳的取值一般为自UTC午夜开始计的毫秒数，与ICMP时间戳请求和应答相类似。如果路由器不使用这种格式，它就可以插入任何它使用的时间表示格式，但是必须打开时间戳中的高位以表明为非标准值。

与我们遇到的记录路由选项所受到的限制相比，时间戳选项遇到情况要更坏一些。如果我们要同时记录IP地址和时间戳（标志位为1），那么就可以同时存入其中的四对值。只记录时间戳是没有用处的，因为我们没有标明时间戳与路由器之间的对应关系（除非有一个永远不变的拓扑结构）。标志值取3会更好一些，因为我们可以插入时间戳的路由器。一个更为基本的问题是，很可能无法控制任何给定路由器上时间戳的正确性。这使得试图用 IP选项来计算路由器之间的跳站数是徒劳的。我们将看到（第8章）traceroute程序可以提供一种更好的方法来计算路由器之间的跳站数。

7.5 小结

ping程序是对两个TCP/IP系统连通性进行测试的基本工具。它只利用 ICMP回显请求和回显应答报文，而不用经过传输层（TCP/UDP）。Ping服务器一般在内核中实现ICMP的功能。

我们分析了在LAN、WAN以及SLIP链路（拨号和线路）上运行 ping程序的输出结果，并对串行线路上的SLIP链路吞吐量进行了计算。我们还讨论并使用了 ping程序的IP记录路由选项。利用该IP选项，可以看到它是如何频繁使用默认路由的。在第 9章我们将再次回到这个讨论主题。另外，还讨论了IP 时间戳选项，但它在实际使用时有所限制。

习题

7.1 请画出7.2节中ping输出的时间线。

7.2 若把bsdi和slip主机之间的SLIP链路设置为9600 b/s，请计算这时的RTT。假定默认的数据是56字节。

7.3 当前BSD版中的ping程序允许我们为ICMP报文的数据部分指定一种模式（数据部分的前8个字节不用来存放模式，因为它要存放发送报文的时间）。如果我们指定的模式为0xc0，请重新计算上一题中的答案（提示：阅读2.4节）。

7.4 使用压缩SLIP（CSLIP，见2.5节）是否会影响我们在7.2节中看到的ping输出中的时间值？

7.5 在图2-4中，ping环回地址与ping主机以太网地址会出现什么不同？

Linux公社 www.linuxidc.com

第8章　Traceroute程序

8.1　引言

由Van Jacobson编写的Traceroute程序是一个能更深入探索TCP/IP协议的方便可用的工具。尽管不能保证从源端发往目的端的两份连续的 IP数据报具有相同的路由，但是大多数情况下是这样的。Traceroute程序可以让我们看到IP数据报从一台主机传到另一台主机所经过的路由。Traceroute程序还可以让我们使用IP源路由选项。

> 使用手册上说："程序由Steve Deering提议，由Van Jacobson实现，并由许多其他人根据C. Philip Wood, Tim Seaver 及Ken Adelman等人提出的令人信服的建议或补充意见进行调试。"

8.2　Traceroute程序的操作

在7.3节中，我们描述了IP记录路由选项（RR）。为什么不使用这个选项而另外开发一个新的应用程序？有三个方面的原因。首先，原先并不是所有的路由器都支持记录路由选项，因此该选项在某些路径上不能使用（Traceroute程序不需要中间路由器具备任何特殊的或可选的功能）。

其次，记录路由一般是单向的选项。发送端设置了该选项，那么接收端不得不从收到的 IP首部中提取出所有的信息，然后全部返回给发送端。在7.3节中，我们看到大多数Ping服务器的实现（内核中的ICMP回显应答功能）把接收到的RR清单返回，但是这样使得记录下来的IP地址翻了一番（一来一回）。这样做会受到一些限制，这一点我们在下一段讨论（Traceroute程序只需要目的端运行一个UDP模块——其他不需要任何特殊的服务器应用程序）。

最后一个原因也是最主要的原因是， IP首部中留给选项的空间有限，不能存放当前大多数的路径。在IP首部选项字段中最多只能存放9个IP地址。在原先的ARPANET中这是足够的，但是对现在来说是远远不够的。

Traceroute程序使用ICMP报文和IP首部中的TTL字段（生存周期）。TTL字段是由发送端初始设置一个8 bit字段。推荐的初始值由分配数字 RFC指定，当前值为64。较老版本的系统经常初始化为15或32。我们从第7章中的一些ping程序例子中可以看出，发送 ICMP回显应答时经常把TTL设为最大值255。

每个处理数据报的路由器都需要把 TTL的值减1或减去数据报在路由器中停留的秒数。由于大多数的路由器转发数据报的时延都小于 1秒钟，因此TTL最终成为一个跳站的计数器，所经过的每个路由器都将其值减1。

> RFC 1009 [Braden and Postel 1987]指出，如果路由器转发数据报的时延超过1秒，那么它将把TTL值减去所消耗的时间（秒数）。但很少有路由器这么实现。新的路由器需求文档RFC [Almquist 1993]为此指定它为可选择功能，允许把TTL看成一个跳站计数器。

Linux公社 www.linuxidc.com

TTL字段的目的是防止数据报在选路时无休止地在网络中流动。例如，当路由器瘫痪或者两个路由器之间的连接丢失时，选路协议有时会去检测丢失的路由并一直进行下去。在这段时间内，数据报可能在循环回路被终止。 TTL字段就是在这些循环传递的数据报上加上一个生存上限。

当路由器收到一份IP数据报，如果其TTL字段是0或1，则路由器不转发该数据报（接收到这种数据报的目的主机可以将它交给应用程序，这是因为不需要转发该数据报。但是在通常情况下，系统不应该接收 TTL字段为0的数据报）。相反，路由器将该数据报丢弃，并给信源机发一份ICMP“超时”信息。Traceroute程序的关键在于包含这份ICMP信息的IP报文的信源地址是该路由器的IP地址。

我们现在可以猜想一下Traceroute程序的操作过程。它发送一份TTL字段为1的IP数据报给目的主机。处理这份数据报的第一个路由器将 TTL值减1，丢弃该数据报，并发回一份超时ICMP报文。这样就得到了该路径中的第一个路由器的地址。然后 Traceroute程序发送一份TTL值为2的数据报，这样我们就可以得到第二个路由器的地址。继续这个过程直至该数据报到达目的主机。但是目的主机哪怕接收到 TTL值为1的IP数据报，也不会丢弃该数据报并产生一份超时ICMP报文，这是因为数据报已经到达其最终目的地。那么我们该如何判断是否已经到达目的主机了呢？

Traceroute程序发送一份UDP数据报给目的主机，但它选择一个不可能的值作为 UDP端口号（大于30 000），使目的主机的任何一个应用程序都不可能使用该端口。因为，当该数据报到达时，将使目的主机的 UDP模块产生一份“端口不可达”错误（见 6.5节）的ICMP报文。这样， Traceroute程序所要做的就是区分接收到的 ICMP报文是超时还是端口不可达，以判断什么时候结束。

> Traceroute程序必须可以为发送的数据报设置TTL字段。并非所有与TCP/IP接口的程序都支持这项功能，同时并非所有的实现都支持这项能力，但目前大部分系统都支持这项功能，并可以运行Traceroute程序。这个程序界面通常要求用户具有超级用户权限，这意味着它可能需要特殊的权限以在你的主机上运行该程序。

8.3 局域网输出

现在已经做好运行Traceroute程序并观察其输出的准备了。我们将使用从 `svr4`到`slip`，经路由器bsdi的简单互联网（见内封面）。`bsdi`和`slip`之间是9600 b/s的SLIP链路。

```
svr4 % traceroute slip
traceroute to slip (140.252.13.65), 30 hops max, 40 byte packets
 1  bsdi (140.252.13.35)  20 ms  10 ms  10 ms
 2  slip (140.252.13.65)  120 ms  120 ms  120 ms
```

输出的第1个无标号行给出了目的主机名和其 IP地址，指出traceroute程序最大的TTL字段值为30。40字节的数据报包含20字节IP首部、8字节的UDP首部和12字节的用户数据（12字节的用户数据包含每发一个数据报就加 1的序列号，送出TTL的副本以及发送数据报的时间）。

输出的后面两行以TTL开始，接下来是主机或路由器名以及其IP地址。对于每个TTL值，发送3份数据报。每接收到一份ICMP报文，就计算并打印出往返时间。如果在5秒种内仍未收到3份数据报的任意一份的响应，则打印一个星号，并发送下一份数据报。在上述输出结果中，TTL字段为1的前3份数据报的ICMP报文分别在20 ms、10 ms和10 ms收到。TTL字段为2的3份数

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

据报的ICMP报文则在120 ms后收到。由于TTL字段为2到达最终目的主机，因此程序就此停止。

往返时间是由发送主机的traceroute程序计算的。它是指从traceroute程序到该路由器的总往返时间。如果我们对每段路径的时间感兴趣，可以用 TTL字段为N+1所打印出来的时间减去TTL字段为N的时间。

图8-1给出了tcpdump的运行输出结果。正如我们所预想的那样，第 1个发往bsdi的探测数据报的往返时间是 20 ms、而后面两个数据报往返时间是 10 ms的原因是发生了一次 ARP交换。tcpdump结果证实了确实是这种情况。

```
 1  0.0                    arp who-has bsdi tell svr4
 2  0.000586 (0.0006)      arp reply bsdi is-at 0:0:c0:6f:2d:40

 3  0.003067 (0.0025)      svr4.42804 > slip.33435: udp 12 [ttl 1]
 4  0.004325 (0.0013)      bsdi > svr4: icmp: time exceeded in-transit

 5  0.069810 (0.0655)      svr4.42804 > slip.33436: udp 12 [ttl 1]
 6  0.071149 (0.0013)      bsdi > svr4: icmp: time exceeded in-transit

 7  0.085162 (0.0140)      svr4.42804 > slip.33437: udp 12 [ttl 1]
 8  0.086375 (0.0012)      bsdi > svr4: icmp: time exceeded in-transit

 9  0.118608 (0.0322)      svr4.42804 > slip.33438: udp 12
10  0.226464 (0.1079)      slip > svr4: icmp: slip udp port 33438 unreachable

11  0.287296 (0.0608)      svr4.42804 > slip.33439: udp 12
12  0.395230 (0.1079)      slip > svr4: icmp: slip udp port 33439 unreachable

13  0.409504 (0.0143)      svr4.42804 > slip.33440: udp 12
14  0.517430 (0.1079)      slip > svr4: icmp: slip udp port 33440 unreachable
```

图8-1 从svr4到slip的traceroute程序示例的tcpdump输出结果

目的主机UDP端口号最开始设置为 33435，且每发送一个数据报加 1。可以通过命令行选项来改变开始的端口号。 UDP数据报包含12个字节的用户数据，我们在前面 traceroute程序输出的40字节数据报中已经对其进行了描述。

后面tcpdump打印出了TTL字段为1的IP数据报的注释 [t t l 1]。当TTL值为0或1时，tcpdump打印出这条信息，以提示我们数据报中有些不太寻常之处。在这里可以预见到 TTL值为1；而在其他一些应用程序中，它可以警告我们数据报可能无法到达其最终目的主机。我们不可能看到路由器传送一个TTL值为0的数据报，除非发出该数据报的该路由器已经崩溃。

因为bsdi路由器将TTL值减到0，因此我们预计它将发回“传送超时”的 ICMP报文。即使这份被丢弃的IP报文发送往slip，路由器也会发回ICMP报文。

有两种不同的ICMP“超时”报文（见6.2节的图6-3），它们的ICMP报文中code字段不同。图8-2给出了这种ICMP差错报文的格式。

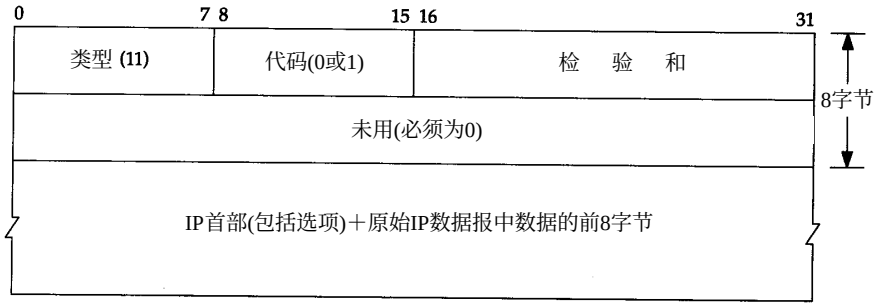


图8-2 ICMP超时报文

Linux公社 www.linuxidc.com

我们所讨论的ICMP报文是在TTL值等于0时产生的，其code字段为0。

主机在组装分片时可能发生超时，这时，它将发送一份“组装报文超时”的 ICMP报文（我们将在11.5节讨论分片和组装）。这种差错报文将code字段置1。

图8-1的第9~14行对应于TTL为2的3份数据报。这3份报文到达最终目的主机，并产生一份ICMP端口不可达报文。

计算出SLIP链路的往返时间是很有意义的，就象我们在 7.2节中所举的Ping例子，将链路值设置为1200b/s一样。发送出的UDP数据报共42个字节，包括12字节的数据、8字节UDP首部、20字节的IP首部以及（至少）2字节的SLIP帧（2.4节）。但是与Ping不一样的是，返回的数据报大小是变化的。从图 6-9可以看出，返回的ICMP报文包含发生差错的数据报的 IP首部以及紧随该IP首部的8字节数据（在traceroute程序中，即UDP首部）。这样，总共就是20 + 8 + 20 + 8 + 2，即58字节。在数据速率为960 b/s的情况下，预计的RTT就是（42 + 58/960），即104 ms。这个值与svr4上所估算出来的110 ms是吻合的。

图8-1中的源端口号（42804）看起来有些大。traceroute程序将其发送的UDP数据报的源端口号设置为 Unix进程号与 32768之间的逻辑或值。对于在同一台主机上多次运行traceroute程序的情况，每个进程都查看ICMP返回的UDP首部的源端口号，并且只处理那些对自己发送应答的报文。

关于traceroute程序，还有一些必须指出的事项。首先，并不能保证现在的路由也是将来所要采用的路由，甚至两份连续的 IP数据报都可能采用不同的路由。如果在运行程序时，路由发生改变，就会观察到这种变化，这是因为对于一个给定的 TTL，如果其路由发生变化，traceroute程序将打印出新的IP地址。

第二，不能保证ICMP报文的路由与traceroute程序发送的UDP数据报采用同一路由。这表明所打印出来的往返时间可能并不能真正体现数据报发出和返回的时间差（如果 UDP数据报从信源到路由器的时间是 1秒，而ICMP报文用另一条路由返回信源用了 3秒时间，则打印出来的往返时间是4秒）。

第三，返回的ICMP报文中的信源IP地址是UDP数据报到达的路由器接口的 IP地址。这与IP记录路由选项（7.3节）不同，记录的IP地址指的是发送接口地址。由于每个定义的路由器都有2个或更多的接口，因此，从 A主机到B主机上运行traceroute程序和从B主机到A主机上运行traceroute程序所得到的结果可能是不同的。事实上，如果我们从 slip主机到svr4上运行traceroute程序，其输出结果变成了：

```
slip % traceroute svr4
traceroute to svr4 (140.252.13.34), 30 hops max, 40 byte packets
 1  bsdi (140.252.13.66)  110 ms  110 ms  110 ms
 2  svr4 (140.252.13.34)  110 ms  120 ms  110 ms
```

这次打印出来的bsdi主机的IP地址是140.252.13.66，对应于SLIP接口；而上次的地址是140.252.13.35，是以太网接口地址。由于 traceroute程序同时也打印出与 IP地址相关的主机名，因而主机名也可能变化（在我们的例子中， bsdi上的两个接口都采用相同的名字）。

考虑图8-3的情况。它给出了两个局域网通过一个路由器相连的情况。两个路由器通过一个点对点的链路相连。如果我们在左边 LAN的一个主机上运行traceroute程序，那么它将发现路由器的IP地址为if1和if3。但在另一种情况下，就会发现打印出来的 IP地址为if4和if2。if2和if3有着同样的网络号，而另两个接口则有着不同的网络号。

Linux公社 www.linuxidc.com

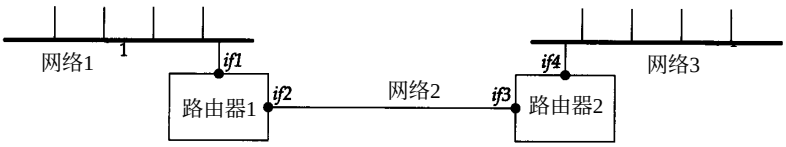


图8-3 traceroute 程序打印出的接口标识

最后，在广域网情况下，如果traceroute程序的输出是可读的域名形式，而不是IP地址形式，那么会更好理解一些。但是由于traceroute程序接收到ICMP报文时，它所获得的唯一信息就是IP地址，因此，在给定IP地址的情况下，它做一个“反向域名查看”工作来获得域名。这就需要路由器或主机的管理员正确配置其反向域名查看功能（并非所有的情况下都是如此）。我们将在14.5节描述如何使用DNS将一个IP地址转换成域名。

8.4 广域网输出

前面所给出的小互联网的输出例子对于查看协议运行过程来说是足够了，但对于像全球互联网这样的大互联网来说，应用traceroute程序就需要一些更为实际的东西。

图8-4是从sun主机到NIC (Network Information Center)的情况。

```
sun % traceroute nic.ddn.mil
traceroute to nic.ddn.mil (192.112.36.5), 30 hops max, 40 byte packets

 1  netb.tuc.noao.edu (140.252.1.183)  218 ms  227 ms  233 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  204 ms

 3  butch.telcom.arizona.edu (140.252.104.2)  204 ms  228 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  234 ms  228 ms  204 ms
 5  NSIgate.Telcom.Arizona.EDU (192.80.43.3)  233 ms  228 ms  234 ms

 6  JPL1.NSN.NASA.GOV (128.161.88.2)  234 ms  590 ms  262 ms
 7  JPL3.NSN.NASA.GOV (192.100.15.3)  238 ms  223 ms  234 ms
 8  GSFC3.NSN.NASA.GOV (128.161.3.33)  293 ms  318 ms  324 ms
 9  GSFC8.NSN.NASA.GOV (192.100.13.8)  294 ms  318 ms  294 ms
10  SURA2.NSN.NASA.GOV (128.161.166.2)  323 ms  319 ms  294 ms
11  nsn-FIX-pe.sura.net (192.80.214.253)  294 ms  318 ms  294 ms
12  GSI.NSN.NASA.GOV (128.161.252.2)  293 ms  318 ms  324 ms

13  NIC.DDN.MIL (192.112.36.5)  324 ms  321 ms  324 ms
```

图8-4 从sun主机到nic.ddn.mil 的traceroute 程序

由于运行的这个例子包含文本，非DDN站点（如，非军方站点）的NIC已经从nic.ddn.mil转移到rs.internic.net，即新的“InterNIC”。

一旦数据报离开tuc.noao.edu网，它们就进入了telcom.arizona.edu网络。然后这些数据报进入NASA Science Internet，nsn.nasa.gov。TTL字段为6和7的路由器位于JPL (Jet Propulsion Laboratory)上。TTL字段为11所输出的sura.net网络位于Southeastern Universities Research Association Network上。TTL字段为12的域名GSI是Government Systems, Inc., NIC的运营者。

TTL字段为6的第2个RTT（590）几乎是其他两个RTT值（234和262）的两倍 。它表明IP路由的动态变化。在发送主机和这个路由器之间发生了使该数据报速度变慢的事件。同样，我们不能区分是发出的数据报还是返回的ICMP差错报文被拦截。

TTL字段为3的第1个RTT探测值（204）比TTL字段为2的第1个探测值（233）值还小。由

Linux公社 www.linuxidc.com

于每个打印出来的RTT值是从发送主机到路由器的总时间，因此这种情况是可能发生的。

图8-5的例子是从sun主机到作者出版商之间的运行例子。

```
sun % traceroute aw.com
traceroute to aw.com (192.207.117.2), 30 hops max, 40 byte packets

 1  netb.tuc.noao.edu (140.252.1.183)  227 ms  227 ms  234 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  234 ms

 3  butch.telcom.arizona.edu (140.252.104.2)  233 ms  229 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  264 ms  228 ms  234 ms
 5  Westgate.Telcom.Arizona.EDU (192.80.43.2)  234 ms  228 ms  234 ms

 6  uu-ua.AZ.westnet.net (192.31.39.233)  263 ms  258 ms  264 ms
 7  enss142.UT.westnet.net (192.31.39.21)  263 ms  258 ms  264 ms

 8  t3-2.Denver-cnss97.t3.ans.net (140.222.97.3)  293 ms  288 ms  275 ms
 9  t3-3.Denver-cnss96.t3.ans.net (140.222.96.4)  283 ms  263 ms  261 ms
10  t3-1.St-Louis-cnss80.t3.ans.net (140.222.80.2)  282 ms  288 ms  294 ms
11  t3-1.Chicago-cnss24.t3.ans.net (140.222.24.2)  293 ms  288 ms  294 ms
12  t3-2.Cleveland-cnss40.t3.ans.net (140.222.40.3)  294 ms  288 ms  294 ms
13  t3-1.New-York-cnss32.t3.ans.net (140.222.32.2)  323 ms  318 ms  324 ms
14  t3-1.Washington-DC-cnss56.t3.ans.net (140.222.56.2)  323 ms  318 ms  324 ms
15  t3-0.Washington-DC-cnss58.t3.ans.net (140.222.58.1)  324 ms  318 ms  324 ms
16  t3-0.enss136.t3.ans.net (140.222.136.1)  323 ms  318 ms  324 ms

17  Washington.DC.ALTER.NET (192.41.177.248)  323 ms  377 ms  324 ms
18  Boston.MA.ALTER.NET (137.39.12.2)  324 ms  347 ms  324 ms
19  AW-gw.ALTER.NET (137.39.62.2)  353 ms  378 ms  354 ms

20  aw.com (192.207.117.2)  354 ms  349 ms  354 ms
```

图8-5 从sun.tuc.noao.edu 主机到aw.com 的traceroute 程序

在这个例子中，数据报离开 telcom.arizona.edu网络后就进行了地区性的网络westnet.net (TTL字段值为6和7)。然后进行了由 Advanced Network & Services运营的NSFNET主干网，t3.ans.net，（T3是对于主干网采用的45 Mb/s电话线的一般缩写。）最后的网络是alter.net，即aw.com与互联网的连接点。

8.5 IP源站选路选项

通常IP路由是动态的，即每个路由器都要判断数据报下面该转发到哪个路由器。应用程序对此不进行控制，而且通常也并不关心路由。它采用类似 Traceroute程序的工具来发现实际的路由。

源站选路(source routing)的思想是由发送者指定路由。它可以采用以下两种形式：

- 严格的源路由选择。发送端指明IP数据报所必须采用的确切路由。如果一个路由器发现源路由所指定的下一个路由器不在其直接连接的网络上，那么它就返回一个“源站路由失败”的ICMP差错报文。
- 宽松的源站选路。发送端指明了一个数据报经过的 IP地址清单，但是数据报在清单上指明的任意两个地址之间可以通过其他路由器。

Traceroute程序提供了一个查看源站选路的方法，我们可以在选项中指明源站路由，然后检查其运行情况。

一些公开的Traceroute程序源代码包中包含指明宽松的源站选路的补丁。但是在标准版中通常并不包含此项。这些补丁的解释是“ Van Jacobson的原始Traceroute程序

Linux公社 www.linuxidc.com

（1988年春）支持该特性，但后来因为有人提出会使网关崩溃而将此功能去除。”对于本章中所给出的例子，作者将这些补丁安装上去，并将它们设置成允许宽松的源站选路和严格的源站选路。

图8-6给出了源站路由选项的格式。

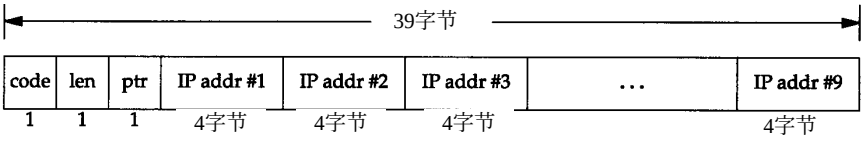


图8-6 IP首部源站路由选项的通用格式

这个格式与我们在图7-3中所示的记录路由选项格式基本一致。不同之处是，对于源站选路，我们必须在发送IP数据报前填充IP地址清单；而对于记录路由选项，我们需要为IP地址清单分配并清空一些空间，并让路由器填充该清单中的各项。同时，对于源站选路，只要为所需要的IP地址数分配空间并进行初始化，通常其数量小于9。而对于记录路由选项来说，必须尽可能地分配空间，以达到9个地址。

对于宽松的源站选路来说，code字段的值是0x83；而对于严格的源站选路，其值为0x89。len和ptr字段与7.3节中所描述的一样。

源站路由选项的实际称呼为“源站及记录路由”（对于宽松的源站选路和严格的源站选路，分别用LSRR和SSRR表示），这是因为在数据报沿路由发送过程中，对IP地址清单进行了更新。下面是其运行过程：

- 发送主机从应用程序接收源站路由清单，将第1个表项去掉（它是数据报的最终目的地址），将剩余的项移到1个项中（如图8-6所示），并将原来的目的地址作为清单的最后一项。指针仍然指向清单的第1项（即，指针的值为4）。
- 每个处理数据报的路由器检查其是否为数据报的最终地址。如果不是，则正常转发数据报（在这种情况下，必须指明宽松源站选路，否则就不能接收到该数据报）。
- 如果该路由器是最终目的，且指针不大于路径的长度，那么(1)由ptr所指定的清单中的下一个地址就是数据报的最终目的地址；(2)由外出接口(outgoing interface)相对应的IP地址取代刚才使用的源地址；(3)指针加4。

可以用下面这个例子很好地解释上述过程。在图8-7中，我们假设主机S上的发送应用程序发送一份数据报给*D*，指定源路由为*R*1，*R*2和*R*3。

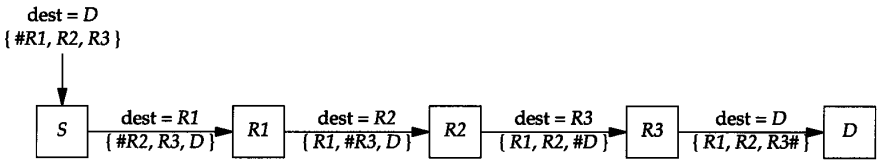


图8-7 IP源路由示例

在上图中，#表示指针字段，其值分别是4、8、12和16。长度字段恒为15（三个IP地址加上三个字节首部）。可以看出，每一跳IP数据报中的目的地址都发生改变。

当一个应用程序接收到由信源指定路由的数据时，在发送应答时，应该读出接收到的路由值，并提供反向路由。

Linux公社 www.linuxidc.com

Host Requirements RFC指明，TCP客户必须能指明源站选路，同时，TCP服务器必须能够接收源站选路，并且对于该TCP连接的所有报文段都能采用反向路由。如果TCP服务器下面接收到一个不同的源站选路，那么新的源站路由将取代旧的源站路由。

8.5.1 宽松的源站选路的traceroute程序示例

使用traceroute程序的-g选项，可以为宽松的源站选路指明一些中间路由器。采用该选项可以最多指定8个中间路由器（其个数是8而不是9的原因是，所使用的编程接口要求最后的表目是目的主机）。

在图8-4中，去往NIC，即nic.ddn.mil的路由经过NASA Science Internet。在图8-8中，我们通过指定路由器enss142.UT.westnet.net (192.31.39.21) 作为中间路由器来强制数据报通过NSFNET：

```
sun % traceroute -g 192.31.39.21  nic.ddn.mil
traceroute to nic.ddn.mil (192.112.36.5), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  259 ms  256 ms  235 ms

 2  butch.telcom.arizona.edu (140.252.104.2)  234 ms  228 ms  234 ms
 3  Gabby.Telcom.Arizona.EDU (128.196.128.1)  234 ms  257 ms  233 ms

 4  enss142.UT.westnet.net (192.31.39.21)  294 ms  288 ms  295 ms

 5  t3-2.Denver-cnss97.t3.ans.net (140.222.97.3)  294 ms  286 ms  293 ms
 6  t3-3.Denver-cnss96.t3.ans.net (140.222.96.4)  293 ms  288 ms  294 ms
 7  t3-1.St-Louis-cnss80.t3.ans.net (140.222.80.2)  294 ms  318 ms  294 ms
 8  * t3-1.Chicago-cnss24.t3.ans.net (140.222.24.2)  318 ms  295 ms
 9  t3-2.Cleveland-cnss40.t3.ans.net (140.222.40.3)  319 ms  318 ms  324 ms
10  t3-1.New-York-cnss32.t3.ans.net (140.222.32.2)  324 ms  318 ms  324 ms
11  t3-1.Washington-DC-cnss56.t3.ans.net (140.222.56.2)  353 ms  348 ms  325 ms
12  t3-0.Washington-DC-cnss58.t3.ans.net (140.222.58.1)  348 ms  347 ms  325 ms
13  t3-0.enss145.t3.ans.net (140.222.145.1)  353 ms  348 ms  325 ms

14  nsn-FIX-pe.sura.net (192.80.214.253)  353 ms  348 ms  325 ms
15  GSI.NSN.NASA.GOV (128.161.252.2)  353 ms  348 ms  354 ms
16  NIC.DDN.MIL (192.112.36.5)  354 ms  347 ms  354 ms
```

图8-8 采用宽松源站选路通过NSFNET到达nic.ddn.mil 的traceroute 程序

在这种情况下，看起来路径中共有16跳，其平均RTT大约是350 ms。而图8-4的通常选路则只有13跳，其平均RTT约为322 ms。默认路径看起来更好一些（在建立路径时，还需要考虑其他的一些因素。其中一些必须考虑的因素是所包含网络的组织及政治因素）。

前面我们说看起来有16跳，这是因为将其输出结果与前面的通过NSFNET（图8-5）的示例比较，发现在本例采用宽松源路由，选择了3个路由器（这可能是因为路由器对源站选路数据报产生ICMP超时差错报文上存在一些差错）。在netb和butch路由器之间的gateway.tuc.noao.edu路由器丢失了，同时，位于Gabby和enss142.UT.west.net之间的Westgate.Telcom.Arizona.edu和uu-ua.AZ.westnet.net两个路由器也丢失了。在这些丢失的路由器上可能发生了与接收到宽松的源站选路选项数据报有关的程序问题。实际上，当采用NSFNET时，信源和NIC之间的路径有19跳。本章习题8.5继续对这些丢失路由器进行讨论。

同时本例也指出了另一个问题。在命令行，我们必须指定路由器enss142.UT.westnet.net的点分十进制IP地址，而不能以其域名代替。这是因为，反向域名解析（14.5节中描述的通过IP

Linux公社 www.linuxidc.com

地址返回域名）将域名与IP地址相关联，但是前向解析（即给出域名返回 IP地址）则无法做到。在DNS中，前向映射和反向映射是两个独立的文件，而并非所有的管理者都同时拥有这两个文件。因此，在一个方向是工作正常而另一个方向却失败的情况并不少见。

还有一种以前没有碰到过的情况是在 TTL字段为8的情况下，对于第一个 RTT，打印一个星号。这表明，发生超时，在5秒内未收到本次探查的应答信号。

将本图与图8-4相比较，还可以得出一个结论，即路由器 nsn-FIX-pe.sura.net同时与NSFNET和NASA Science Internet相连。

8.5.2 严格的源站选路的traceroute程序示例

在作者的traceroute程序版本中，-G选项与前面所描述的-g选项是完全一样的，不过此时是严格的源站选路而不是宽松的源站选路。我们可以采用这个选项来观察在指明无效的严格的源站选路时其结果会是什么样的。从图 8-5可以看出来，从作者的子网发往 NSFNET的数据报的正常路由器顺序是 netb，gateway，butch和gabby（为了便于查看，后面所有的输出结果中，均省略了域名后缀 .tuc.noao.edu和.telcom.arizona.edu）。我们指定了一个严格源路由，使其试图将数据报从gateway直接发送到gabby，而省略了butch。我们可以猜测到其结果会是失败的，正如图 8-9所给出的结果。

```
sun % traceroute -G netb -G gateway -G gabby westgate
traceroute to westgate (192.80.43.2), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  272 ms  257 ms  261 ms
 2  gateway (140.252.1.4)  263 ms  259 ms  234 ms
 3  gateway (140.252.1.4)  263 ms !S *  235 ms !S
```

图8-9 采用严格源站路由失败的traceroute程序

这里的关键是在于TTL字段为3的输出行中，RTT后面的!S。这表明traceroute程序接收到ICMP“源站路由失败”的差错报文：即图 6-3中type字段为3，而code字段为5。TTL字段为3的第二个 RTT位置的星号表示未收到这次探查的应答信号。这与我们所猜想的一样，gateway不可能直接发送数据报给gabby，这是因为它们之间没有直接的连接。

TTL字段为2和3的结果都来自于gateway，对于TTL字段为2的应答来自gateway，是因为gateway接收到TTL字段为1的数据报。在它查看到（无效的）严格的源站选路之前，就发现TTL已过期，因此发送回ICMP超时报文。TTL字段等于3的行，在进入gateway时其TTL字段为2，因此，它查看严格的源站选路，发现它是无效的，因此发送回 ICMP源站选路失败的差错报文。

图8-10给出了与本例相对应的 tcpdump输出结果。该输出结果是在 sun和netb之间的SLIP链路上遇到的。我们必须在 tcpdump中指定-v选项以显示出源站路由信息。这样，会输出一些像数据报 ID这样我们并不需要的结果，我们在给出结果中将这些不需要的结果删除掉。同样，用SSRR表示“严格的源站及记录路由”。

首先注意到，sun所发送的每个 UDP数据报的目的地址都是 netb，而不是目的主机(westgate)。这一点可以用图8-7的例子来解释。类似地，-G选项所指定的另外两个路由器（gateway和gabby）以及最终目(westgate)成为第一跳的SSRR选项。

从这个输出结果中，还可以看出，traceroute程序所采用的定时时间（第 15行和16行

Linux公社 www.linuxidc.com

之间的时间差）是5秒。

```
1  0.0                  sun.33593 > netb.33435: udp 12 [ttl 1]
                        (optlen=16 SSRR{#gateway gabby westgate} EOL)
2  0.270278 (0.2703)    netb > sun: icmp: time exceeded in-transit
3  0.284784 (0.0145)    sun.33593 > netb.33436: udp 12 [ttl 1]
                        (optlen=16 SSRR{#gateway gabby westgate} EOL)
4  0.540338 (0.2556)    netb > sun: icmp: time exceeded in-transit
5  0.550062 (0.0097)    sun.33593 > netb.33437: udp 12 [ttl 1]
                        (optlen=16 SSRR{#gateway gabby westgate} EOL)
6  0.810310 (0.2602)    netb > sun: icmp: time exceeded in-transit
7  0.818030 (0.0077)    sun.33593 > netb.33438: udp 12 (ttl 2,
                        optlen=16 SSRR{#gateway gabby westgate} EOL)
8  1.080337 (0.2623)    gateway > sun: icmp: time exceeded in-transit
9  1.092564 (0.0122)    sun.33593 > netb.33439: udp 12 (ttl 2,
                        optlen=16 SSRR{#gateway gabby westgate} EOL)
10 1.350322 (0.2578)    gateway > sun: icmp: time exceeded in-transit
11 1.357382 (0.0071)    sun.33593 > netb.33440: udp 12 (ttl 2,
                        optlen=16 SSRR{#gateway gabby westgate} EOL)
12 1.590586 (0.2332)    gateway > sun: icmp: time exceeded in-transit
13 1.598926 (0.0083)    sun.33593 > netb.33441: udp 12 (ttl 3,
                        optlen=16 SSRR{#gateway gabby westgate} EOL)
14 1.860341 (0.2614)    gateway > sun:
                        icmp: gateway unreachable - source route failed
15 1.875230 (0.0149)    sun.33593 > netb.33442: udp 12 (ttl 3,
                        optlen=16 SSRR{#gateway gabby westgate} EOL)
16 6.876579 (5.0013)    sun.33593 > netb.33443: udp 12 (ttl 3,
                        optlen=16 SSRR{#gateway gabby westgate} EOL)
17 7.110518 (0.2339)    gateway > sun:
                        icmp: gateway unreachable - source route failed
```

图8-10 失败的严格源站选路traceroute 程序的tcpdump 输出结果

8.5.3 宽松的源站选路traceroute程序的往返路由

我们在前面已经说过，从A到B的路径并不一定与从B到A的路径完全一样。除非同时在两个系统中登录并在每个终端上运行 traceroute程序，否则很难发现两条路径是否不同。但是，采用宽松的源站选路，就可以决定两个方向上的路径。

这里的窍门就在于指定一个宽松的源站路由，该路由的目的端和宽松路径一样，但发送端为目的主机。例如，在sun主机上，我们可以查看到发往以及来自bruno.cs.colorado.edu的结果如图8-11所示。

发出路径（TTL字段为1~11）的结果与返回路径（TTL字段为11~21）不同，这很好地说明了在Internet 上，选路可能是不对称的。

该输出同时还说明了我们在图8-3中所讨论的问题。比较TTL字段为2和19的输出结果：它们都是路由器gateway.tuc.noao.edu，但两个IP地址却是不同的。由于traceroute程序以进入接口作为其标识，而我们从两条不同的方向经过该路由器，一条是发出路径（ TTL字段为2），另一条是返回路径（TTL字段为19），因此可以猜想到这个结果。通过比较 TTL字段为3和18、4和17的结果，可以看到同样的结果。

Linux公社 www.linuxidc.com

```
sun % traceroute -g bruno.cs.colorado.edu sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  230 ms  227 ms  233 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  234 ms
 3  butch.telcom.arizona.edu (140.252.104.2)  234 ms  229 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  233 ms  231 ms  234 ms
 5  NSIgate.Telcom.Arizona.EDU (192.80.43.3)  294 ms  258 ms  234 ms
 6  JPL1.NSN.NASA.GOV (128.161.88.2)  264 ms  258 ms  264 ms
 7  JPL2.NSN.NASA.GOV (192.100.15.2)  264 ms  258 ms  264 ms
 8  NCAR.NSN.NASA.GOV (128.161.97.2)  324 ms *  295 ms
 9  cu-gw.ucar.edu (192.43.244.4)  294 ms  318 ms  294 ms
10  engr-gw.Colorado.EDU (128.138.1.3)  294 ms  288 ms  294 ms
11  bruno.cs.colorado.edu (128.138.243.151)  293 ms  317 ms  294 ms
12  engr-gw-ot.cs.colorado.edu (128.138.204.1)  323 ms  317 ms  384 ms
13  cu-gw.Colorado.EDU (128.138.1.1)  294 ms  318 ms  294 ms
14  enss.ucar.edu (192.43.244.10)  323 ms  318 ms  294 ms
15  t3-1.Denver-cnss97.t3.ans.net (140.222.97.2)  294 ms  288 ms  384 ms
16  t3-0.enss142.t3.ans.net (140.222.142.1)  293 ms  288 ms  294 ms
17  Gabby.Telcom.Arizona.EDU (192.80.43.1)  294 ms  288 ms  294 ms
18  Butch.Telcom.Arizona.EDU (128.196.128.88)  293 ms  317 ms  294 ms
19  gateway.tuc.noao.edu (140.252.104.1)  294 ms  289 ms  294 ms
20  netb.tuc.noao.edu (140.252.1.183)  324 ms  321 ms  294 ms
21  sun.tuc.noao.edu (140.252.13.33)  534 ms  529 ms  564 ms
```

图8-11 显示非对称路径的traceroute 程序

8.6 小结

在一个TCP/IP网络中，traceroute程序是不可缺少的工具。其操作很简单：开始时发送一个TTL字段为1的UDP数据报，然后将TTL字段每次加1，以确定路径中的每个路由器。每个路由器在丢弃UDP数据报时都返回一个ICMP超时报文2，而最终目的主机则产生一个ICMP端口不可达的报文。

我们给出了在LAN和WAN上运行traceroute程序的例子，并用它来考察IP源站选路。我们用宽松的源站选路来检测发往目的主机的路由是否与从目的主机返回的路由一样。

习题

8.1 当IP将接收到的TTL字段减1，发现它为0时，将会发生什么结果？

8.2 traceroute程序是如何计算RTT的？将这种计算RTT的方法与ping相比较。

8.3 （本习题与下一道习题是基于开发traceroute程序过程中遇到的实际问题，它们来自于traceroute程序源代码注释）。假设源主机和目的主机之间有三个路由器（R1、R2和R3），而中间的路由器（R2）在进入TTL字段为1时，将TTL字段减1，但却错误地将该IP数据报发往下一个路由器。请描述会发生什么结果。在运行traceroute程序时会看到什么样的现象？

8.4 同样，假设源主机和目的主机之间有三个路由器。由于目的主机上存在错误，因此，它总是将进入TTL值作为外出ICMP报文的TTL值。请描述这将发生什么结果，你会看到什么现象。

Linux公社 www.linuxidc.com

8.5 在图8-8运行例子中，我们可以在sun和netb之间的SLIP链路上运行tcpdump程序。如果指定-v选项，就可以看到返回ICMP报文的TTL值。这样，我们可以看到进入netb、butch、Gabby和enss142.UT.westnet.net的TTL值分别为255、253、252和249。这是否为我们判断是否存在丢失路由器提供了额外的信息？

8.6 SunOS和SVR4都提供了带-l选项的ping版本，以提供松源选路。手册上说明，该选项可以与-R选项（指定记录路由选项）一起使用。如果已经进入到这些系统中，请尝试同时用这两个选项。其结果是什么？如果采用tcpdump来观测数据报，请描述其过程。

8.7 比较ping和traceroute程序在处理同一台主机上客户的多个实例的不同点。

8.8 比较ping和traceroute程序在计算往返时间上的不同点。

8.9 我们已经说过，traceroute程序选取开始UDP目的主机端口号为33453，每发送一个数据报将此数加1。在1.9节中，我们说过暂时端口号通常是1024~5000之间的值，因此traceroute程序的目的主机端口号不可能是目的主机上所使用的端口号。在Solaris2.2系统中的情况也是如此吗？（提示：查看E.4节）

8.10 RFC 1393 [Malkin 1993b]提出了另一种判断到目的主机路径的方法。请问其优缺点是什么？

Linux公社 www.linuxidc.com

第9章　IP选路

9.1　引言

选路是IP最重要的功能之一。图 9-1是IP层处理过程的简单流程。需要进行选路的数据报可以由本地主机产生，也可以由其他主机产生。在后一种情况下，主机必须配置成一个路由器，否则通过网络接口接收到的数据报，如果目的地址不是本机就要被丢弃（例如，悄无声息地被丢弃）。

在图9-1中，我们还描述了一个路由守护程序（ daemon），通常这是一个用户进程。在Unix系统中，大多数普通的守护程序都是路由程序和网关程序（术语 daemon指的是运行在后台的进程，它代表整个系统执行某些操作。 daemon一般在系统引导时启动，在系统运行期间一直存在）。在某个给定主机上运行何种路由协议，如何在相邻路由器上交换选路信息，以及选路协议是如何工作的，所有这些问题都是非常复杂的，其本身就可以用整本书来加以讨论（有兴趣的读者可以参考文献 [Perlman 1992]以获得更详细的信息）。在第10章中，我们将简单讨论动态选路和选路信息协议 RIP（Routing Information Protocol）。在本章中，我们主要的目的是了解单个IP层如何作出路由决策。

图9-1所示的路由表经常被 IP访问（在一个繁忙的主机上，一秒钟内可能要访问几百次），但是它被路由守护程序更新的频度却要低得多（可能大约 30秒种一次）。当接收到ICMP重定向，报文时，路由表也要被更新，这一点我们将在 9.5节讨论route命令时加以介绍。在本章中，我们还将用`netstat`命令来显示路由表。

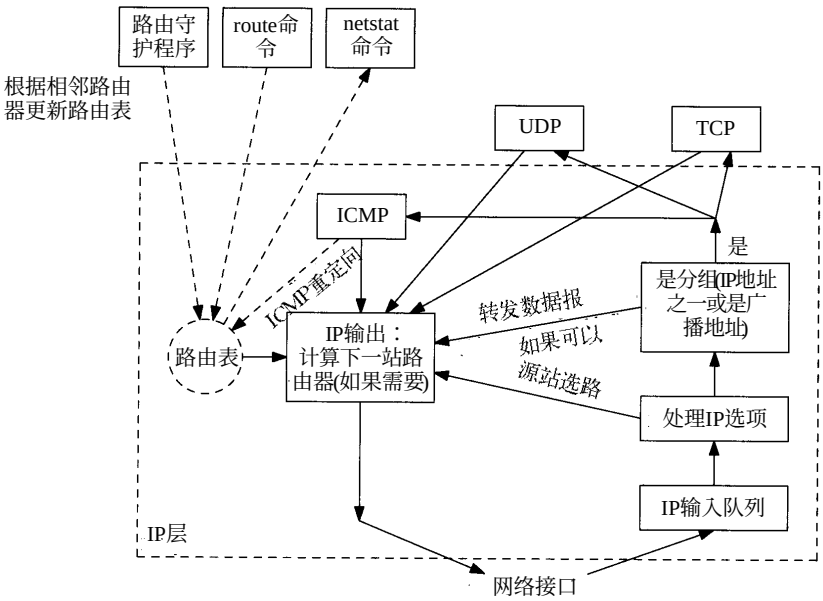


图9-1　IP层工作流程

Linux公社 www.linuxidc.com

9.2 选路的原理

开始讨论IP选路之前，首先要理解内核是如何维护路由表的。路由表中包含的信息决定了IP层所做的所有决策。

在3.3节中，我们列出了IP搜索路由表的几个步骤：

1) 搜索匹配的主机地址；

2) 搜索匹配的网络地址；

3) 搜索默认表项（默认表项一般在路由表中被指定为一个网络表项，其网络号为 0）。

匹配主机地址步骤始终发生在匹配网络地址步骤之前。

IP层进行的选路实际上是一种选路机制，它搜索路由表并决定向哪个网络接口发送分组。这区别于选路策略，它只是一组决定把哪些路由放入路由表的规则。 IP执行选路机制，而路由守护程序则一般提供选路策略。

9.2.1 简单路由表

首先来看一看一些典型的主机路由表。在主机svr4上，我们先执行带-r选项的netstat命令列出路由表，然后以-n选项再次执行该命令，以数字格式打印出IP地址（我们这样做是因为路由表中的一些表项是网络地址，而不是主机地址。如果没有- n选项，netstat命令将搜索文件/etc/networks并列出其中的网络名。这样会与另一种形式的名字——网络名加主机名相混淆）。

```
svr4 % netstat -rn
Routing tables
Destination        Gateway            Flags    Refcnt Use       Interface
140.252.13.65      140.252.13.35      UGH      0      0         emd0
127.0.0.1          127.0.0.1          UH       1      0         lo0
default            140.252.13.33      UG       0      0         emd0
140.252.13.32      140.252.13.34      U        4      25043     emd0
```

第1行说明，如果目的地是 140.252.13.65（slip主机），那么网关（路由器）将把分组转发给140.252.13.35（bsdi）。这正是我们所期望的，因为主机 slip通过SLIP链路与bsdi相连接，而bsdi与该主机在同一个以太网上。

对于一个给定的路由器，可以打印出五种不同的标志（ flag）：

U 该路由可以使用。

G 该路由是到一个网关（路由器）。如果没有设置该标志，说明目的地是直接相连的。

H 该路由是到一个主机，也就是说，目的地址是一个完整的主机地址。如果没有设置该标志，说明该路由是到一个网络，而目的地址是一个网络地址：一个网络号，或者网络号与子网号的组合。

D 该路由是由重定向报文创建的（9.5节）。

M 该路由已被重定向报文修改（9.5节）。

标志G是非常重要的，因为由它区分了间接路由和直接路由（对于直接路由来说是不设置标志G的）。其区别在于，发往直接路由的分组中不但具有指明目的端的 IP地址，还具有其链路层地址（见图 3-3）。当分组被发往一个间接路由时， IP地址指明的是最终的目的地，但是链路层地址指明的是网关（即下一站路由器）。我们在图3-4已看到这样的例子。在这个路由表例子中，有一个间接路由（设置了标志 G），因此采用这一项路由的分组其 IP地址是最终的目的地（140.252.13.65），但是其链路层地址必须对应于路由器 140.252.13.35。

Linux公社 www.linuxidc.com

理解G和H标志之间的区别是很重要的。G标志区分了直接路由和间接路由，如上所述。但是H标志表明，目的地址（netstat命令输出第一行）是一个完整的主机地址。没有设置H标志说明目的地址是一个网络地址（主机号部分为 0）。当为某个目的IP地址搜索路由表时，主机地址项必须与目的地址完全匹配，而网络地址项只需要匹配目的地址的网络号和子网号就可以了。另外，大多数版本的 netstat命令首先打印出所有的主机路由表项，然后才是网络路由表项。

参考记数Refcnt（Reference count）列给出的是正在使用路由的活动进程个数。面向连接的协议如TCP在建立连接时要固定路由。如果在主机 svr4和slip之间建立Telnet连接，可以看到参考记数值变为1。建立另一个Telnet连接时，它的值将增加为2，依此类推。

下一列（“ use ”）显示的是通过该路由发送的分组数。如果我们是这个路由的唯一用户，那么运行ping程序发送5个分组后，它的值将变为 5。最后一列（interface）是本地接口的名字。

输出的第2行是环回接口（2.7节），它的名字始终为lo0。没有设置G标志，因为该路由不是一个网关。H标志说明目的地址（127.0.0.1）是一个主机地址，而不是一个网络地址。由于没有设置G标志，说明这是一个直接路由，网关列给出的是外出 IP地址。

输出的第3行是默认路由。每个主机都有一个或多个默认路由。这一项表明，如果在表中没有找到特定的路由，就把分组发送到路由器 140.252.13.33（sun主机）。这说明当前主机（svr4）利用这一个路由表项就可以通过 Internet经路由器sun（及其SLIP链路）访问其他的系统。建立默认路由是一个功能很强的概念。该路由标志（ UG）表明它是一个网关，这是我们所期望的。

> 这里，我们有意称sun为路由器而不是主机，因为它被当作默认路由器来使用，它发挥的是IP转发功能，而不是主机功能。
>
> Host Requirements RFC文档特别说明，IP层必须支持多个默认路由。但是，许多实现系统并不支持这一点。当存在多个默认路由时，一般的技术就成为它们周围的知更鸟了，例如，Solaris 2.2就是这样做的。

输出中的最后一行是所在的以太网。H标志没有设置，说明目的地址（140.252.13.32）是一个网络地址，其主机地址部分设为 0。事实上，是它的低 5位设为0（见图3-11）。由于这是一个直接路由（G标志没有被设置），网关列指出的IP地址是外出地址。

netstat命令输出的最后一项还隐含了另一个信息，那就是目的地址（ 140.252.13.32）的子网掩码。如果要把该目的地址与 140.252.13.33进行比较，那么在比较之前首先要把它与目的地址掩码（0xffffffe0，3.7节）进行逻辑与。由于内核知道每个路由表项对应的接口，而且每个接口都有一个对应的子网掩码，因此每个路由表项都有一个隐含的子网掩码。

主机路由表的复杂性取决于主机所在网络的拓扑结构。

1) 最简单的（也是最不令人感兴趣的）情况是主机根本没有与任何网络相连。 TCP/IP协议仍然能用于这样的主机，但是只能与自己本身通信！这种情况下的路由表只包含环回接口一项。

2) 接下来的情况是主机连在一个局域网上，只能访问局域网上的主机。这时路由表包含两项：一项是环回接口，另一项是局域网（如以太网）。

3) 如果主机能够通过单个路由器访问其他网络（如 Internet）时，那么就要进行下一步。

Linux公社 www.linuxidc.com

一般情况下增加一个默认表项指向该路由器。

4) 如果要新增其他的特定主机或网络路由，那么就要进行最后一步。在我们的例子中，到主机slip的路由要通过路由器bsdi就是这样的例子。

我们根据上述IP操作的步骤使用这个路由表为主机svr4上的一些分组例子选择路由。

1) 假定目的地址是主机sun，140.252.13.33。首先进行主机地址的匹配。路由表中的两个主机地址表项（slip和localhost）均不匹配，接着进行网络地址匹配。这一次匹配成功，找到表项140.252.13.32（网络号和子网号都相同），因此使用emd0接口。这是一个直接路由，因此链路层地址将是目的端的地址。

2) 假定目的地址是主机slip，140.252.13.65。首先在路由表搜索主机地址，并找到一个匹配地址。这是一个间接路由，因此目的端的 IP地址仍然是140.252.13.65，但是链路层地址必须是网关140.252.13.65的链路层地址，其接口名为emd0。

3) 这一次我们通过Internet给主机aw.com（192.207.117.2）发送一份数据报。首先在路由表中搜索主机地址，失败后进行网络地址匹配。最后成功地找到默认表项。该路由是一个间接路由，通过网关140.252.13.33，并使用接口名为emd0。

4) 在我们最后一个例子中，我们给本机发送一份数据报。有四种方法可以完成这件事，如用主机名、主机IP地址、环回名或者环回IP地址：

```
ftp svr4
ftp 140.252.13.34
ftp localhost
ftp 127.0.0.1
```

在前两种情况下，对路由表的第 2次搜索得到一个匹配的网络地址 140.252.13.32，并把IP报文传送给以太网驱动程序。正如图 2-4所示的那样， IP报文中的目的地址为本机 IP地址，因此报文被送给环回驱动程序，然后由驱动程序把报文放入 IP输出队列中。

在后两种情况下，由于指定了环回接口的名字或 IP地址，第一次搜索就找到匹配的主机地址，因此报文直接被送给环回驱动程序，然后由驱动程序把报文放入 IP输出队列中。

上述四种情况报文都要被送给环回驱动程序，但是采用的两种路由决策是不相同的。

9.2.2 初始化路由表

我们从来没有说过这些路由表是如何被创建的。每当初始化一个接口时（通常是用ifconfig命令设置接口地址），就为接口自动创建一个直接路由。对于点对点链路和环回接口来说，路由是到达主机（例如，设置 H标志）。对于广播接口来说，如以太网，路由是到达网络。

到达主机或网络的路由如果不是直接相连的，那么就必须加入路由表。一个常用的方法是在系统引导时显式地在初始化文件中运行 route命令。在主机svr4上，我们运行下面两个命令来添加路由表中的表项：

```
route add default sun 1
route add slip bsdi 1
```

第3个参数（default和slip）代表目的端，第4个参数代表网关（路由器），最后一个参数代表路由的度量(metric)。route命令在度量值大于0时要为该路由设置G标志，否则，当耗费值为0时就不设置G标志。

Linux公社 www.linuxidc.com

不幸的是，几乎没有系统愿意在启动文件中包含route命令。在4.4BSD和BSD/386系统中，启动文件是 /etc/netstart；在SVR4系统中，启动文件是/etc/inet/rc.inet；在Solaris 2.x中，启动文件是/etc/rc2.d/S69inet；在SunOS 4.1.x中，启动文件是/etc/rc.local；而AIX 3.2.2则使用文件/etc/rc.net。

一些系统允许在某个文件中指定默认的路由器，如 /etc/defaultrouter。于是在每次重新启动系统时都要在路由表中加入该默认项。

初始化路由表的其他方法是运行路由守护程序（第 10章）或者用较新的路由器发现协议（9.6节）。

9.2.3 较复杂的路由表

在我们的子网上，主机 sun是所有主机的默认路由器，因为它有拨号 SLIP链路连接到Internet上（参见扉页前图）。

```
sun % netstat -rn
Routing tables
Destination        Gateway            Flags    Refcnt Use      Interface
140.252.13.65      140.252.13.35      UGH      0      171      le0
127.0.0.1          127.0.0.1          UH       1      766      lo0
140.252.1.183      140.252.1.29       UH       0      0        sl0
default            140.252.1.183      UG       1      2955     sl0
140.252.13.32      140.252.13.33      U        8      99551    le0
```

前两项与主机svr4的前两项一致：通过路由器bsdi到达slip的特定主机路由，以及环回路由。

第3行是新加的。这是一个直接到达主机的路由 (没有设置G标志，但设置了H标志)，对应于点对点的链路，即SLIP接口。如果我们把它与ifconfig命令的输出进行比较：

```
sun % ifconfig sl0
sl0: flags=1051<UP,POINTOPOINT,RUNNING>
      inet 140.252.1.29 --> 140.252.1.183 netmask ffffff00
```

可以发现路由表中的目的地址就是点对点链路的另一端 (即路由器netb)，网关地址为外出接口的本地IP地址(140.252.1.29)(前面已经说过，netstat为直接路由打印出来的网关地址就是本地接口所用的IP地址)。

默认的路由表项是一个到达网络的间接路由 (设置了G标志，但没有设置H标志)，这正是我们所希望的。网关地址是路由器的地址 (140.252.1.183，SLIP链路的另一端)，而不是SLIP链路的本地IP地址(140.252.1.29)。其原因还是因为是间接路由，不是直接路由。

还应该指出的是，netstat输出的第3和第4行(接口名为sl0)由SLIP软件在启动时创建，并在关闭时删除.

9.2.4 没有到达目的地的路由

我们所有的例子都假定对路由表的搜索能找到匹配的表项，即使匹配的是默认项。如果路由表中没有默认项，而又没有找到匹配项，这时会发生什么情况呢？

结果取决于该IP数据报是由主机产生的还是被转发的（例如，我们就充当一个路由器）。如果数据报是由本地主机产生的，那么就给发送该数据报的应用程序返回一个差错，或者是“主机不可达差错”或者是“网络不可达差错”。如果是被转发的数据报，那么就给原始发送

Linux公社 www.linuxidc.com

端发送一份ICMP主机不可达的差错报文。下一节将讨论这种差错。

9.3 ICMP主机与网络不可达差错

当路由器收到一份IP数据报但又不能转发时，就要发送一份ICMP“主机不可达”差错报文（ICMP主机不可达报文的格式如图6-10所示）。可以很容易发现，在我们的网络上把接在路由器sun上的拨号SLIP链路断开，然后试图通过该SLIP链路发送分组给任何指定sun为默认路由器的主机。

> 较老版本的BSD产生一个主机不可达或者网络不可达差错，这取决于目的端是否处于一个局域子网上。4.4 BSD只产生主机不可达差错。

我们在上一节通过在路由器sun上运行netstat命令可以看到，当接通SLIP链路启动时就要在路由表中增加一项使用SLIP链路的表项，而当断开SLIP链路时则删除该表项。这说明当SLIP链路断开时，sun的路由表中就没有默认项了。但是我们不想改变网络上其他主机的路由表，即同时删除它们的默认路由。相反，对于sun不能转发的分组，我们对它产生的ICMP主机不可达差错报文进行计数。

在主机svr4上运行ping程序就可以看到这一点，它在拨号SLIP链路的另一端（拨号链路已被断开）：

```
svr4 % ping gemini
ICMP Host Unreachable from gateway sun (140.252.13.33)
ICMP Host Unreachable from gateway sun (140.252.13.33)
^?                                          键入中断键停止显示
```

在主机bsdi上运行tcpdump命令的输出如图9-2所示。

```
1  0.0                  svr4 > gemini: icmp: echo request
2  0.00 (0.00)          sun > svr4: icmp: host gemini unreachable

3  0.99 (0.99)          svr4 > gemini: icmp: echo request
4  0.99 (0.00)          sun > svr4: icmp: host gemini unreachable
```

图9-2 响应ping命令的ICMP主机不可达报文

当路由器sun发现找不到能到达主机gemini的路由时，它就响应一个主机不可达的回显请求报文。

如果把SLIP链路接到Internet上，然后试图ping一个与Internet没有连接的IP地址，那么应该会产生差错。但令人感兴趣的是，我们可以看到在返回差错报文之前，分组要在Internet上传送多远：

```
sun % ping 192.82.148.1
PING 192.82.148.1: 56 data bytes        该IP地址没有连接到Internet上
ICMP Host Unreachable from gateway enss142.UT.westnet.net (192.31.39.21)
 for icmp from sun (140.252.1.29) to 192.82.148.1
```

从图8-5可以看出，在发现该IP地址是无效的之前，该分组已通过了6个路由器。只有当它到达NSFNET骨干网的边界时才检测到差错。这说明，6个路由器之所以能转发分组是因为路由表中有默认项。只有当分组到达NSFNET骨干网时，路由器才能知道每个连接到Internet上的每个网络的信息。这说明许多路由器只能在局部范围内工作。

参考文献[Ford, Rekhter, and Braun 1993]定义了顶层选路域（top-level routing domain），由它来维护大多数Internet网站的路由信息，而不使用默认路由。他们指出，在Internet上存在

Linux公社 www.linuxidc.com

5个这样的顶层选路域：NSFNET主干网、商业互联网交换（Commercial Internet Exchange: CIX）、NASA科学互联网（NASA Science Internet: NSI）、SprintLink以及欧洲IP主干网（EBONE）。

9.4 转发或不转发

前面我们已经提过几次，一般都假定主机不转发 IP数据报，除非对它们进行特殊配置而作为路由器使用。如何进行这样的配置呢？

大多数伯克利派生出来的系统都有一个内核变量 `ipforwarding`，或其他类似的名字（参见附录E）。一些系统（如BSD/386和SVR4）只有在该变量值不为0的情况下才转发数据报。SunOS 4.1.x允许该变量可以有三个不同的值：-1表示始终不转发并且始终不改变它的值；0表示默认条件下不转发，但是当打开两个或更多个接口时就把该值设为 1；1表示始终转发。Solaris 2.x把这三个值改为0（始终不转发）、1（始终转发）和2（在打开两个或更多个接口时才转发）。

较早版本的4.2BSD主机在默认条件下可以转发数据报，这给没有进行正确配置的系统带来了许多问题。这就是内核选项为什么要设成默认的“始终不转发”的原因，除非系统管理员进行特殊设置。

9.5 ICMP重定向差错

当IP数据报应该被发送到另一个路由器时，收到数据报的路由器就要发送 ICMP重定向差错报文给IP数据报的发送端。这在概念上是很简单的，正如图 9-3所示的那样。只有当主机可以选择路由器发送分组的情况下，我们才可能看到 ICMP重定向报文（回忆我们在图 7-6中看过的例子）。

1) 我们假定主机发送一份IP数据报给R1。这种选路决策经常发生，因为R1是该主机的默认路由。

2) R1收到数据报并且检查它的路由表，发现R2是发送该数据报的下一站。当它把数据报发送给R2时，R1检测到它正在发送的接口与数据报到达接口是相同的（即主机和两个路由器所在的LAN）。这样就给路由器发送重定向报文给原始发送端提供了线索。

3) R1发送一份ICMP重定向报文给主机，告诉它以后把数据报发送给 R2而不是R1。

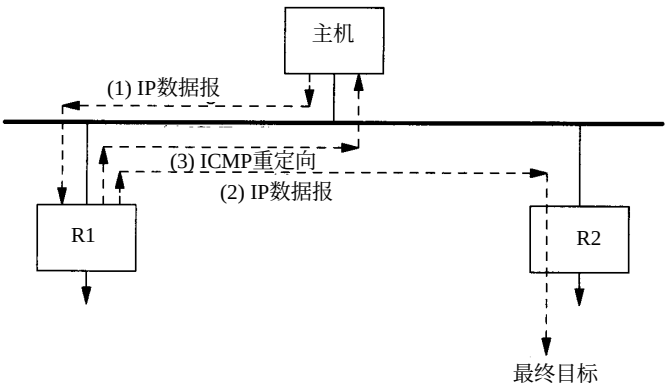


图9-3 ICMP重定向的例子

Linux公社 www.linuxidc.com

重定向一般用来让具有很少选路信息的主机逐渐建立更完善的路由表。主机启动时路由表中可以只有一个默认表项（在图 9-3所示的例子中，为 R1或R2）。一旦默认路由发生差错，默认路由器将通知它进行重定向，并允许主机对路由表作相应的改动。 ICMP重定向允许TCP/IP主机在进行选路时不需要具备智能特性，而把所有的智能特性放在路由器端。显然，在我们的例子中，R1和R2 必须知道有关相连网络的更多拓扑结构的信息，但是连在 LAN上的所有主机在启动时只需一个默认路由，通过接收重定向报文来逐步学习。

9.5.1 一个例子

可以在我们的网络上观察到 ICMP重定向的操作过程（见封二的图）。尽管在拓扑图中只画出了三台主机（aix,solaris和gemini）和两台路由器（gateway和netb），但是整个网络有超过150台主机和10台另外的路由器。大多数的主机都把 gateway指定为默认路由器，因为它提供了Internet的入口。

子网140.252.1上的主机是如何访问作者所在子网（图中底下的四台主机）的呢？首先，如果在SLIP链路的一端只有一台主机，那么就要使用代理 ARP（4.6节）。这意味着位于拓扑图顶部的子网（140.252.1）中的主机不需要其他特殊条件就可以访问主机 sun（140.252.1.29）。位于netb上的代理ARP软件处理这些事情。

但是，当网络位于 SLIP链路的另一端时，就要涉及到选路了。一个办法是让所有的主机和路由器都知道路由器 netb是网络140.252.13的网关。这可以在每个主机的路由表中设置静态路由，或者在每个主机上运行守护程序来实现。另一个更简单的办法（也是实际采用的方法）是利用ICMP重定向报文来实现。

在位于网络顶部的主机 solaris上运行ping程序到主机bsdi(140.252.13.35)。由于子网号不相同，代理 ARP不能使用。假定没有安装静态路由，发送的第一个分组将采用到路由器gateway的默认路由。下面是我们运行 ping程序之前的路由表：

```
solaris % netstat -rn
Routing Table:
  Destination        Gateway        Flags  Ref   Use   Interface
---------------- ---------------- ----- ----- ------ ---------
127.0.0.1        127.0.0.1         UH       0    848  lo0
140.252.1.0      140.252.1.32      U        3  15042  le0
224.0.0.0        140.252.1.32      U        3      0  le0
default          140.252.1.4       UG       0   5747
```

（224.0.0.0所在的表项是IP广播地址。我们将在第 12章讨论）。如果为ping程序指定 - v选项，可以看到主机接收到的任何ICMP报文。我们需要指定该选项以观察发送的重定向报文。

```
solaris % ping -sv bsdi
PING bsdi: 56 data bytes
ICMP Host redirect from gateway gateway (140.252.1.4)
 to netb (140.252.1.183) for bsdi (140.252.13.35)
64 bytes from bsdi (140.252.13.35): icmp_seq=0. time=383. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=1. time=364. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=2. time=353. ms
^?                                    键入中断键停止显示
----bsdi PING Statistics----
4 packets transmitted, 3 packets received, 25% packet loss
round-trip (ms)  min/avg/max = 353/366/383
```

在收到ping程序的第一个响应之前，主机先收到一份来自默认路由器 gateway发来的

Linux公社 www.linuxidc.com

ICMP重定向报文。如果这时查看路由表，就会发现已经插入了一个到主机 bsdi的新路由（该表项如以下黑体字所示）。

```
solaris % netstat -rn
Routing Table:
  Destination        Gateway        Flags  Ref   Use   Interface
---------------- ---------------- ----- ----- ------ ---------
127.0.0.1        127.0.0.1        UH        0    848  lo0
140.252.13.35    140.252.1.183    UGHD      0      2
140.252.1.0      140.252.1.32     U         3  15045  le0
224.0.0.0        140.252.1.32     U         3      0  le0
default          140.252.1.4      UG        0   5749
```

这是我们第一次看到D标志，表示该路由是被ICMP重定向报文创建的。G标志说明这是一份到达gateway(netb）的间接路由，H标志则说明这是一个主机路由（正如我们期望的那样），而不是一个网络路由。

由于这是一个被主机重定向报文增加的主机路由，因此它只处理到达主机 bsdi的报文。如果我们接着访问主机 svr4，那么就要产生另一个 ICMP重定向报文，创建另一个主机路由。类似地，访问主机 slip也创建另一个主机路由。位于子网上的三台主机（ bsdi,svr4和slip）还可以由一个指向路由器 sun的网络路由来进行处理。但是 ICMP重定向报文创建的是主机路由，而不是网络路由，这是因为在本例中，产生 ICMP重定向报文的路由器并不知道位于140.252.13网络上的子网信息。

9.5.2 更多的细节

ICMP重定向报文的格式如图9-4所示。

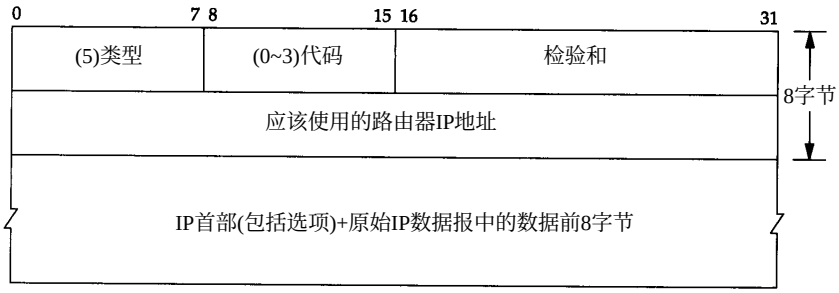


图9-4 ICMP重定向报文

有四种不同类型的重定向报文，有不同的代码值，如图 9-5所示。

ICMP重定向报文的接收者必须查看三个 IP地址：(1)导致重定向的IP地址（即ICMP重定向报文的数据位于IP数据报的首部）；(2)发送重定向报文的路由器的IP地址（包含重定向信息的 IP数据报中的源地址；(3)应该采用的路由器IP地址（在ICMP报文中的4~7字节）。

代码	描　述
0	网络重定向
1	主机重定向
2	服务类型和网络重定向
3	服务类型和主机重定向

图9-5 ICMP重定向报文的不同代码值

关于ICMP重定向报文有很多规则。首先，重定向报文只能由路由器生成，而不能由主机生成。另外，重定向报文是为主机而不是为路由器使用的。假定路由器和其他一些路由器共同参与某一种选路协议，则该协议就能消除重定向的需要（这意味着在图 9-1中的路由表应该

Linux公社 www.linuxidc.com

消除或者能被选路守护程序修改，或者能被重定向报文修改，但不能同时被二者修改）。

在4.4BSD系统中，当主机作为路由器使用时，要进行下列检查。在生成 ICMP重定向报文之前这些条件都要满足。

1) 出接口必须等于入接口。

2) 用于向外传送数据报的路由不能被 ICMP重定向报文创建或修改过，而且不能是路由器的默认路由。

3) 数据报不能用源站选路来转发。

4) 内核必须配置成可以发送重定向报文。

内核变量取名为ip_sendredirects或其他类似的名字（参见附录E）。大多数当前的系统（例如BSD、 SunOS 4.1.x、Solaris 2.x 及AIX 3.2.2）在默认条件下都设置该变量，使系统可以发送重定向报文。其他系统如SVR4则关闭了该项功能。

另外，一台4.4BSD主机收到ICMP重定向报文后，在修改路由表之前要作一些检查。这是为了防止路由器或主机的误操作，以及恶意用户的破坏，导致错误地修改系统路由表。

1) 新的路由器必须直接与网络相连接。

2) 重定向报文必须来自当前到目的地所选择的路由器。

3) 重定向报文不能让主机本身作为路由器。

4) 被修改的路由必须是一个间接路由。

关于重定向最后要指出的是，路由器应该发送的只是对主机的重定向（代码 1或3，如图9-5所示），而不是对网络的重定向。子网的存在使得难于准确指明何时应发送对网络的重定向而不是对主机的重定向。只当路由器发送了错误的类型时，一些主机才把收到的对网络的重定向当作对主机的重定向来处理。

9.6 ICMP路由器发现报文

在本章前面已提到过一种初始化路由表的方法，即在配置文件中指定静态路由。这种方法经常用来设置默认路由。另一种新的方法是利用 ICMP路由器通告和请求报文。

一般认为，主机在引导以后要广播或多播传送一份路由器请求报文。一台或更多台路由器响应一份路由器通告报文。另外，路由器定期地广播或多播传送它们的路由器通告报文，允许每个正在监听的主机相应地更新它们的路由表。

RFC 1256 [Deering 1991]确定了这两种ICMP报文的格式。ICMP路由器请求报文的格式如图9-6所示。ICMP路由器通告报文的格式如图 9-7所示。

路由器在一份报文中可以通告多个地址。地址数指的是报文中所含的地址数。地址项大小指的是每个路由器地址 32 bit字的数目，始终为 2。生存期指的是通告地址有效的时间（秒数）。

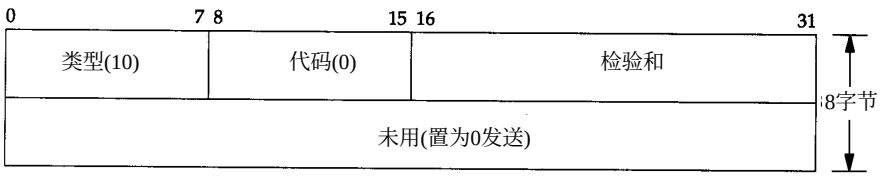


图9-6 ICMP路由器请求报文格式

Linux公社 www.linuxidc.com

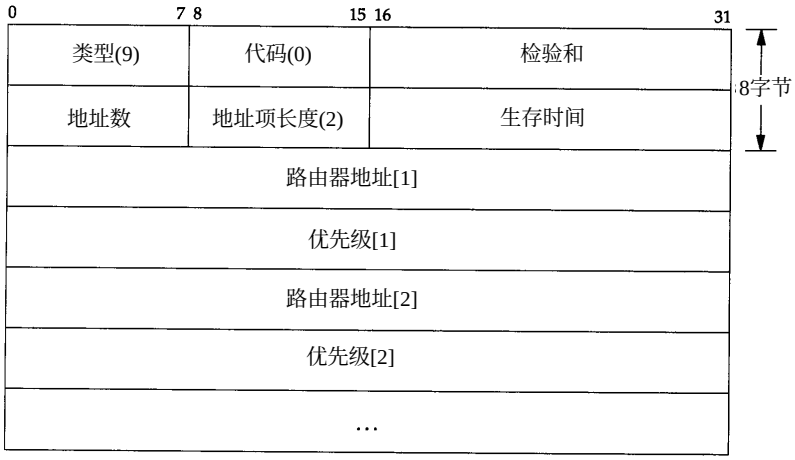


图9-7 ICMP路由器通告报文格式

接下来是一对或多对IP地址和优先级。IP地址必须是发送路由器的某个地址。优先级是一个有符号的32 bit整数，指出该IP地址作为默认路由器地址的优先等级，这是与子网上的其他路由器相比较而言的。值越大说明优先级越高。优先级为 0x80000000说明对应的地址不能作为默认路由器地址使用，尽管它也包含中通告报文中。优先级的默认值一般为 0。

9.6.1 路由器操作

当路由器启动时，它定期在所有广播或多播传送接口上发送通告报文。准确地说，这些通告报文不是定期发送的，而是随机传送的，以减小与子网上其他路由器发生冲突的概率。一般每两次通告间隔450秒和600秒。一份给定的通告报文默认生命周期是 30分钟。

使用生命周期域的另一个时机是当路由器上的某个接口被关闭时。在这种情况下，路由器可以在该接口上发送最后一份通告报文，并把生命周期值设为 0。

除了定期发送主动提供的通告报文以外，路由器还要监听来自主机的请求报文，并发送路由器通告报文以响应这些请求报文。

如果子网上有多台路由器，由系统管理员为每个路由器设置优先等级。例如，主默认路由器就要比备份路由器具有更高的优先级。

9.6.2 主机操作

主机在引导期间一般发送三份路由器请求报文，每三秒钟发送一次。一旦接收到一个有效的通告报文，就停止发送请求报文。

主机也监听来自相邻路由器的请求报文。这些通告报文可以改变主机的默认路由器。另外，如果没有接收到来自当前默认路由器的通告报文，那么默认路由器会超时。

只要有一般的默认路由器，该路由器就会每隔 10分钟发送通告报文，报文的生命周期是30分钟。这说明主机的默认表项是不会超时的，即使错过一份或两份通告报文。

9.6.3 实现

路由器发现报文一般由用户进程（守护程序）创建和处理。这样，在图 9-1中就有另一个

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

修改路由表的程序，尽管它只增加或删除默认表项。守护程序必须把它配置成一台路由器或主机来使用。

这两种ICMP报文是新加的，不是所有的系统都支持它们。在我们的网络中，只有Solaris 2.x支持这两种报文（`in.rdisc`守护程序）。尽管RFC建议尽可能用IP多播传送，但是路由器发现还可以利用广播报文来实现。

9.7 小结

IP路由操作对于运行TCP／IP的系统来说是最基本的，不管是主机还是路由器。路由表项的内容很简单，包括：5 bit标志、目的IP地址（主机、网络或默认）、下一站路由器的IP地址（间接路由）或者本地接口的IP地址（直接路由）及指向本地接口的指针。主机表项比网络表项具有更高的优先级，而网络表项比默认项具有更高的优先级。

系统产生的或转发的每份IP数据报都要搜索路由表，它可以被路由守护程序或ICMP重定向报文修改。系统在默认情况下不转发数据报，除非进行特殊的配置。用`route`命令可以进入静态路由，可以利用新ICMP路由器发现报文来初始化默认表项，并进行动态修改。主机在启动时只有一个简单的路由表，它可以被来自默认路由器的ICMP重定向报文动态修改。

在本章中，我们集中讨论了单个系统是如何利用路由表的。在下一章，我们将讨论路由器之间是如何交换路由信息的。

习题

9.1 为什么你认为存在两类ICMP重定向报文——网络和主机？

9.2 在9.4节开头列出的`svr4`主机上的路由表中，到主机`slip`（140.252.13.65）的特定路由是必需的吗？如果把这一项从路由表中删除会有什么变化？

9.3 考虑有一电缆连接4.2BSD主机和4.3BSD主机。假定网络号是140.1。4.2BSD主机把主机号为全0的地址识别为广播地址(140.1.0.0)，而4.3BSD通常使用全1的主机号（140.1.255.255）发送广播。另外，4.2BSD主机在默认条件下要尽力转发接收到的数据报，尽管它们只有一个接口。

请描述当4.2BSD主机收到一份目的地址为140.1.255.255的IP数据报时会发生什么事。

9.4 继续前一个习题，假定有人在子网140.1上的某个系统ARP高速缓存中增加了一项（用arp命令）内容，指定IP地址140.1.255.255对应的以太网地址为全1（以太网广播地址）。请描述此时发生的情况。

9.5 检查你所使用的系统上的路由表，并解释每一项内容。

Linux公社 www.linuxidc.com

第10章　动态选路协议

10.1　引言

在前面各章中，我们讨论了静态选路。在配置接口时，以默认方式生成路由表项（对于直接连接的接口），并通过route命令增加表项（通常从系统自引导程序文件），或是通过ICMP重定向生成表项（通常是在默认方式出错的情况下）。

在网络很小，且与其他网络只有单个连接点且没有多余路由时（若主路由失败，可以使用备用路由），采用这种方法是可行的。如果上述三种情况不能全部满足，通常使用动态选路。

本章讨论动态选路协议，它用于路由器间的通信。我们主要讨论 RIP，即选路信息协议(Routing Infromation Protocol)，大多数TCP/IP实现都提供这个应用广泛的协议。然后讨论两种新的选路协议， OSPF和BGP。本章的最后研究一种名叫无分类域间选路的新的选路技术，现在Internet上正在开始采用该协议以保持B类网络的数量。

10.2　动态选路

当相邻路由器之间进行通信，以告知对方每个路由器当前所连接的网络，这时就出现了动态选路。路由器之间必须采用选路协议进行通信，这样的选路协议有很多种。路由器上有一个进程称为路由守护程序（ routing daemon），它运行选路协议，并与其相邻的一些路由器进行通信。正如图9-1所示，路由守护程序根据它从相邻路由器接收到的信息，更新内核中的路由表。

动态选路并不改变我们在 9.2节中所描述的内核在IP层的选路方式。这种选路方式称为选路机制（routing mechanism）。内核搜索路由表，查找主机路由、网络路由以及默认路由的方式并没有改变。仅仅是放置到路由表中的信息改变了 —— 当路由随时间变化时，路由是由路由守护程序动态地增加或删除，而不是来自于自引导程序文件中的 route命令。

正如前面所描述的那样，路由守护程序将选路策略（ routing policy）加入到系统中，选择路由并加入到内核的路由表中。如果守护程序发现前往同一信宿存在多条路由，那么它（以某种方法）将选择最佳路由并加入内核路由表中。如果路由守护程序发现一条链路已经断开（可能是路由器崩溃或电话线路不好），它可以删除受影响的路由或增加另一条路由以绕过该问题。

在像Internet这样的系统中，目前采用了许多不同的选路协议。 Internet是以一组自治系统(AS，Autonomous System)的方式组织的，每个自治系统通常由单个实体管理。常常将一个公司或大学校园定义为一个自治系统。 NSFNET的Internet骨干网形成一个自治系统，这是因为骨干网中的所有路由器都在单个的管理控制之下。

每个自治系统可以选择该自治系统中各个路由器之间的选路协议。这种协议我们称之为内部网关协议IGP（Interior Gateway Protocol）或域内选路协议（intradomain routing protocol）。

Linux公社 www.linuxidc.com

最常用的IGP是选路信息协议 RIP。一种新的IGP是开放最短路径优先 OSPF（Open Shortest Path First）协议。它意在取代RIP。另一种1986年在原来NSFNET骨干网上使用的较早的IGP协议 —— HELLO，现在已经不用了。

新的RFC [Almquist 1993]规定，实现任何动态选路协议的路由器必须同时支持OSPF和RIP，还可以支持其他IGP协议。

外部网关协议EGP（Exterier Gateway Protocol）或域内选路协议的分隔选路协议用于不同自治系统之间的路由器。在历史上，（令人容易混淆）改进的EGP有着一个与它名称相同的协议：EGP。新EGP是当前在NSFNET骨干网和一些连接到骨干网的区域性网络上使用的是边界网关协议BGP（Border Gateway Protocol）。BGP意在取代EGP。

10.3 Unix选路守护程序

Unix系统上常常运行名为`routed`路由守护程序。几乎在所有的TCP/IP实现中都提供该程序。该程序只使用RIP进行通信，我们将在下一节中讨论该协议。这是一种用于小型到中型网络中的协议。

另一个程序是`gated`。IGP和EGP都支持它。[Fedor 1998]描述了早期开发的`gated`。图10-1对`routed`和两种不同版本的`gated`所支持的不同选路协议进行了比较。大多数运行路由守护程序的系统都可以运行`routed`，除非它们需要支持`gated`所支持的其他协议。

守护程序	内部网点协议			外部网点协议	
	HELLO	RIP	OSPF	EGP	BGP
routed		V1			
gated, 版本2	•	V1		•	V1
gated, 版本3	•	V1, V2	V2	•	V2, V3

图10-1 routed 和gated 所支持的选路协议

我们在下一节中描述RIP 版本1，10.5节描述它与RIP版本2的不同点，10.6节描述OSPF，10.7节描述BGP。

10.4 RIP：选路信息协议

本节对RIP进行了描述，这是因为它是最广为使用（也是最受攻击）的选路协议。对于 RIP的正式描述文件是RFC 1058 [Hedrick 1988a]，但是该RFC是在该协议实现数年后才出现的。

10.4.1 报文格式

RIP报文包含中在UDP数据报中，如图10-2所示（在第11章中对UDP进行更为详细的描述）。

图10-3给出了使用IP地址时的RIP报文格式。

命令字段为1表示请求，2表示应答。还有两个舍弃不用的命令（3和4），两个非正式的命令：轮询（5）和轮询表项（6）。请求表示要求其他系统发送其全部或部分路由

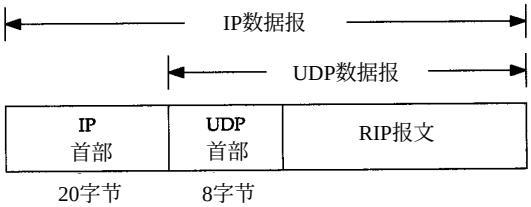


图10-2 封装在UDP数据报中的RIP报文

Linux公社 www.linuxidc.com

表。应答则包含发送者全部或部分路由表。

版本字段通常为1，而第2版RIP（10.5节）将此字段设置为2。

紧跟在后面的20字节指定地址系列（address family）（对于IP地址来说，其值是2）、IP地址以及相应的度量。在本节的后面可以看出， RIP的度量是以跳计数的。

采用这种20字节格式的RIP报文可以通告多达25条路由。上限25是用来保证RIP报文的总长度为20×25 + 4 = 504，小于512字节。由于每个报文最多携带 25个路由，因此为了发送整个路由表，经常需要多个报文。

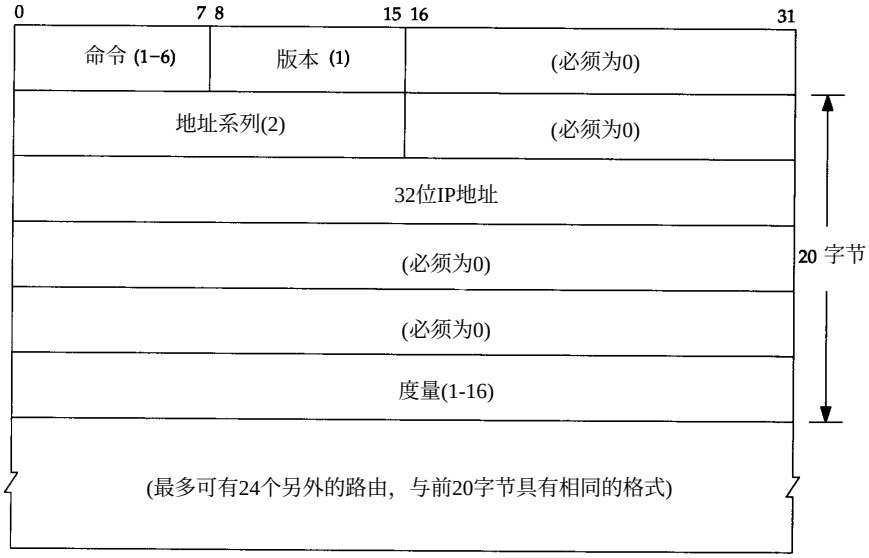


图 10-3

10.4.2 正常运行

让我们来看一下采用RIP协议的routed程序正常运行的结果。RIP常用的UDP端口号是520。

- 初始化：在启动一个路由守护程序时，它先判断启动了哪些接口，并在每个接口上发送一个请求报文，要求其他路由器发送完整路由表。在点对点链路中，该请求是发送给其他终点的。如果网络支持广播的话，这种请求是以广播形式发送的。目的 UDP端口号是520（这是其他路由器的路由守护程序端口号）。

  这种请求报文的命令字段为 1，但地址系列字段设置为 0，而度量字段设置为 16。这是一种要求另一端完整路由表的特殊请求报文。
- 接收到请求。如果这个请求是刚才提到的特殊请求，那么路由器就将完整的路由表发送给请求者。否则，就处理请求中的每一个表项：如果有连接到指明地址的路由，则将度量设置成我们的值，否则将度量置为 16（度量为16是一种称为“无穷大”的特殊值，它意味着没有到达目的的路由）。然后发回响应。
- 接收到响应。使响应生效，可能会更新路由表。可能会增加新表项，对已有的表项进行修改，或是将已有表项删除。
- 定期选路更新。每过30秒，所有或部分路由器会将其完整路由表发送给相邻路由器。发送路由表可以是广播形式的（如在以太网上），或是发送给点对点链路的其他终点的。

Linux公社 www.linuxidc.com

• 触发更新。每当一条路由的度量发生变化时，就对它进行更新。不需要发送完整路由表，而只需要发送那些发生变化的表项。

每条路由都有与之相关的定时器。如果运行 RIP的系统发现一条路由在 3分钟内未更新，就将该路由的度量设置成无穷大（ 16），并标注为删除。这意味着已经在 6个30秒更新时间里没收到通告该路由的路由器的更新了。再过 60秒，将从本地路由表中删除该路由，以保证该路由的失效已被传播开。

10.4.3 度量

RIP所使用的度量是以跳 (hop)计算的。所有直接连接接口的跳数为 1。考虑图10-4所示的路由器和网络。画出的 4条虚线是广播 RIP报文。

路由器R1通过发送广播到 N1通告它与N2之间的跳数是 1（发送给 N1的广播中通告它与N1之间的路由是无用的）。同时也通过发送广播给N2通告它与N1之间的跳数为1。同样， R2通告它与N2的度量为1，与N3的度量为1。

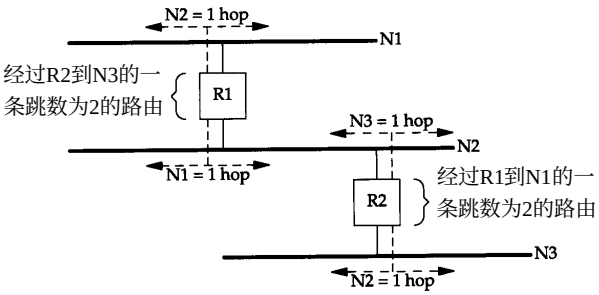


图10-4 路由器和网络示例

如果相邻路由器通告它与其他网络路由的跳数为1，那么我们与那个网络的度量就是 2，这是因为为了发送报文到该网络，我们必须经过那个路由器。在我们的例子中， R2到N1的度量是2，与R1到N3的度量一样。

由于每个路由器都发送其路由表给邻站，因此，可以判断在同一个自治系统 AS内到每个网络的路由。如果在该 AS内从一个路由器到一个网络有多条路由，那么路由器将选择跳数最小的路由，而忽略其他路由。

跳数的最大值是 15，这意味着 RIP只能用在主机间最大跳数值为 15的AS内。度量为16表示到无路由到达该IP地址。

10.4.4 问题

这种方法看起来很简单，但它有一些缺陷。首先， RIP没有子网地址的概念。例如，如果标准的B类地址中16 bit的主机号不为0，那么RIP无法区分非零部分是一个子网号，或者是一个主机地址。有一些实现中通过接收到的RIP信息，来使用接口的网络掩码，而这有可能出错。

其次，在路由器或链路发生故障后，需要很长的一段时间才能稳定下来。这段时间通常需要几分钟。在这段建立时间里，可能会发生路由环路。在实现 RIP时，必须采用很多微妙的措施来防止路由环路的出现，并使其尽快建立。 RFC 1058 [Hedrick 1988a]中指出了很多实现RIP的细节。

采用跳数作为路由度量忽略了其他一些应该考虑的因素。同时，度量最大值为 15则限制了可以使用RIP的网络的大小。

10.4.5 举例

我们将使用`ripquery`程序来查询一些路由器中的路由表，该程序可以从 `gated`中得到。

Linux公社 www.linuxidc.com

ripquery程序通过发送一个非正式请求（图10-3中命令字段为5的“poll”）给路由器，要求得到其完整的路由表。如果在5秒内未收到响应，则发送标准的RIP请求（command字段为1）（前面提到过的，将地址系列字段置为0，度量字段置为16的请求，要求其他路由器发送其完整路由表）。

图10-5给出了将从sun主机上查询其路由表的两个路由器。如果在主机sun上执行ripquery程序，以得到其下一站路由器netb的选路信息，那么可以得到下面的结果：

```
sun % ripquery -n netb
504 bytes from netb (140.252.1.183):  第一份报文包含504字节
                                      这里删除了许多行
      140.252.1.0, metric 1           图10-5中上面的以太网
      140.252.13.0, metric 1          图10-5中下面的以太网
244 bytes from netb (140.252.1.183):  第二份报文包含剩下的244字节下面删除了许多行
```

正如我们所猜想的那样，netb告诉我们子网的度量为1。另外，与netb相连的位于机端的以太网（140.252.1.0）的metric也是1（-n参数表示直接打印IP地址而不需要去查看其域名）。在本例中，将netb配置成认为所有位于140.252.13子网的主机都与其直接相连——即，netb并不知道哪些主机真正与140.252.13子网相连。由于与140.252.13子网只有一个连接点，因此，通告每个主机的度量实际上没有太大意义。

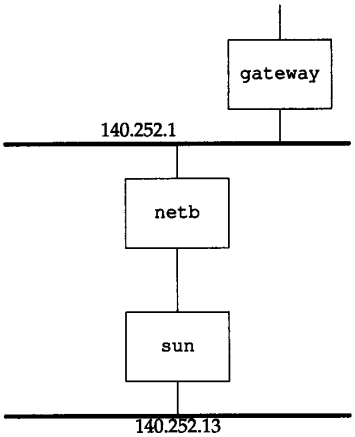


图10-5 查询其路由表内容的两个路由器netb和gateway

图10-6给出了使用tcpdump交换的报文。采用-i s10选项指定SLIP接口。

第1个请求发出一个RIP轮询命令（第1行）。这个请求在5秒后超时，发出一个常规的RIP请求（第2行）。第1行和第2行最后的24表示请求报文的长度：4个字节的RIP首部（包括命令和版本），然后是单个20字节的地址和度量。

第3行是第一个应答报文。该行最后的25表示包含了25个地址和度量对，我们在前面已经计算过，其字节数为504。这是上面的ripquery程序所打印出来的结果。我们为tcpdump程序指定-s600选项，以让它从网络中读取600个字节。这样，它可以接收整个UDP数据报（而不是报文的前半部），然后打印出RIP响应的内容。该输出结果省略了。

```
   sun % tcpdump -s600 -i sl0
1  0.0                  sun.2879 > netb.route: rip-poll 24
2  5.014702 (5.0147)    sun.2879 > netb.route: rip-req 24
3  5.560427 (0.5457)    netb.route > sun.2879: rip-resp 25:
4  5.710251 (0.1498)    netb.route > sun.2879: rip-resp 12:
```

图10-6 运行ripquery程序的tcpdump输出结果

第4行是来自路由器的第二个响应报文，它包含后面的12个地址和度量对。可以计算出该报文的长度为12×20+4=244，这正是ripquery程序所打印出来的结果。

如果越过netb路由器，到gateway，那么可以预测到我们子网（140.252.13.0）的度量为2。可以运行下面的命令来进行验证：

Linux公社 www.linuxidc.com

```
sun %ripquery -n gateway
504 bytes from gateway (140.252.1.4):
                                                    这里删除了许多行
      140.252.1.0, metric 1          图10-5上面的以太网
      140.252.13.0, metric 2                  图10-5下面的以太网
```

这里，位于图10-5上面的以太网（140.252.1.0）的度量依然是1，这是因为该以太网直接与gateway和netb相连。而我们的子网140.252.13.0正如预想的一样，其度量为2。

10.4.6 另一个例子

现在察看以太网上所有非主动请求的RIP更新，以看一看RIP定期给其邻站发送的信息。图10-7是noao.edu网络的多种排列情况。为了简化，我们不用本文其他地方所采用的路由器表示方式，而以Rn来代表路由器，其中n是子网号。以虚线表示点对点链路，并给出了这些链路对端的IP地址。

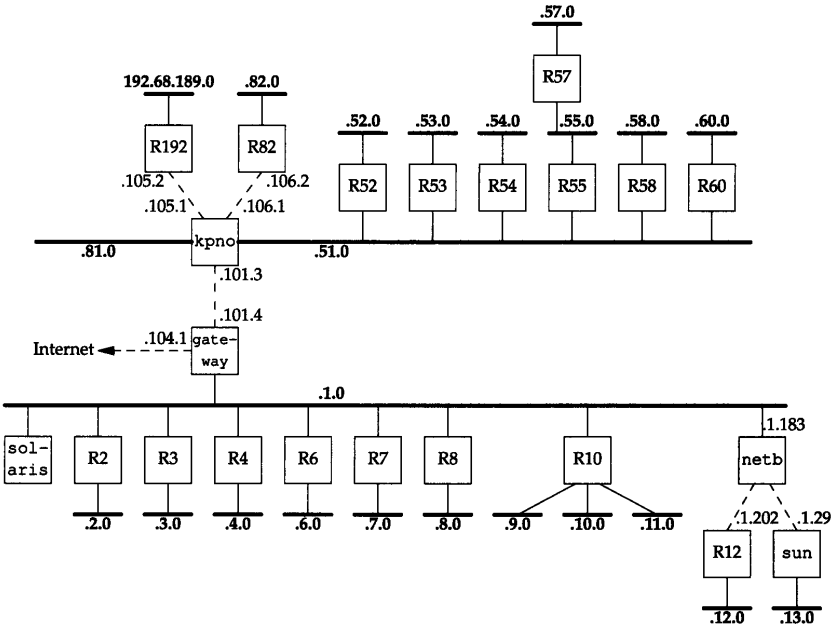


图10-7 noao.edu 140.252的多个网络

在主机solaris上运行Solaris 2.x的snoop程序，它与tcpdump相类似。我们可以在不需要超用户权限的条件下运行该程序，但它只捕获广播报文、多播报文以及发送给主机的报文。图10-8给出了在60秒内所捕获的报文。在这里，我们将大部分正式的主机名以Rn来表示。

-P标志以非混杂模式捕获报文，-tr打印出相应的时戳，而udp port 520只捕获信源或信宿端口号为520的UDP数据报。

来自R6、R4、R2、R7、R8和R3的前6个报文，每个报文只通告一个网络。查看这些报文，可以发现R2通告前往140.252.6.0的跳数为1的一条路由，R4通告前往140.252.4.0的跳数为1的一条路由，等等。

但是，gateway路由器却通告了15条路由。我们可以通过运行snoop程序时加上-v参数来查看RIP报文的全部内容（这个标志输出全部报文的全部内容：以太网首部、IP首部、UDP首部以及RIP报文。我们只保留了RIP信息而删除了其他信息）。图10-9给出了输出结果。

Linux公社 www.linuxidc.com

```
solaris % snoop -P -tr udp port 520
  0.00000 R6.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
  4.49708 R4.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
  6.30506 R2.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 11.68317 R7.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 16.19790 R8.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 16.87131 R3.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 17.02187 gateway.tuc.noao.edu -> 140.252.1.255 RIP R (15 destinations)
 20.68009 R10.tuc.noao.edu -> BROADCAST    RIP R (4 destinations)

 29.87848 R6.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 34.50209 R4.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 36.32385 R2.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 41.34565 R7.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 46.19257 R8.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 46.52199 R3.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 47.01870 gateway.tuc.noao.edu -> 140.252.1.255 RIP R (15 destinations)
 50.66453 R10.tuc.noao.edu -> BROADCAST    RIP R (4 destinations)
```

图10-8 solaris 在60秒内所捕获到的RIP广播报文

把这些子网140.252.1上通告报文经过的路由与图10-7中的拓扑结构进行比较。

使人迷惑不解的一个问题是为什么图 10-8输出结果中，R10通告其有4个网络而在图10-7中显示的只有3个。如果查看带snoop的RIP报文，就会得到以下通告路由：

```
RIP: Address        Metric
RIP:  140.251.0.0    16 (not reachable)
RIP:  140.252.9.0    1
RIP:  140.252.10.0   1
RIP:  140.252.11.0   1
```

前往B类网络140.251的路由是假的，不应该通告它（它属于其他机构而不是noao.edu）。

```
solaris % snoop -P -v -tr udp port 520 host gateway
                                        删去许多行
RIP:  Opcode = 2 (route response)
RIP:  Version = 1

RIP:  Address         Metric

RIP:  140.252.101.0   1
RIP:  140.252.104.0   1

RIP:  140.252.51.0    2
RIP:  140.252.81.0    2
RIP:  140.252.105.0   2
RIP:  140.252.106.0   2

RIP:  140.252.52.0    3
RIP:  140.252.53.0    3
RIP:  140.252.54.0    3
RIP:  140.252.55.0    3
RIP:  140.252.58.0    3
RIP:  140.252.60.0    3
RIP:  140.252.82.0    3
RIP:  192.68.189.0    3

RIP:  140.252.57.0    4
```

图10-9 来自gateway 的RIP响应

图10-8中，对于 R10发送的RIP报文，snoop输出“BROADCAST”符号，它表示目的IP地址是有限的广播地址255.255.255.255（12.2节），而不是其他路由器用来指向子网的广播地

Linux公社 www.linuxidc.com

址（140.252.1.255）。

10.5 RIP版本2

RFC 1388 [Malkin 1993a]中对RIP定义进行了扩充，通常称其结果为 RIP-2。这些扩充并不改变协议本身，而是利用图10-3中的一些标注为“必须为0”的字段来传递一些额外的信息。如果RIP忽略这些必须为0的字段，那么，RIP和RIP-2可以互操作。

图10-10重新给出了由RIP-2定义的图。对于RIP-2来说，其版本字段为2。

图10-10 RIP-2报文格式

选路域(routing domain)是一个选路守护程序的标识符，它指出了这个数据报的所有者。在一个Unix实现中，它可以是选路守护程序的进程号。该域允许管理者在单个路由器上运行多个RIP实例，每个实例在一个选路域内运行。

选路标记(routing tag)是为了支持外部网关协议而存在的。它携带着一个 EGP和BGP的自治系统号。

每个表项的子网掩码应用于相应的 IP地址上。下一站IP地址指明发往目的IP地址的报文该发往哪里。该字段为0意味着发往目的地址的报文应该发给发送 RIP报文的系统。

RIP-2提供了一种简单的鉴别机制。可以指定 RIP报文的前20字节表项地址系列为 0xffff，路由标记为2。表项中的其余16字节包含一个明文口令。

最后，RIP-2除了广播（第12章）外，还支持多播。这可以减少不收听 RIP-2报文的主机的负载。

10.6 OSPF：开放最短路径优先

OSPF是除RIP外的另一个内部网关协议。它克服了 RIP的所有限制。RFC 1247 [Moy 1991]中对第2版OSPF进行了描述。

与采用距离向量的 RIP协议不同的是，OSPF是一个链路状态协议。距离向量的意思是，RIP发送的报文包含一个距离向量（跳数）。每个路由器都根据它所接收到邻站的这些距离向

Linux公社 www.linuxidc.com

量来更新自己的路由表。

在一个链路状态协议中，路由器并不与其邻站交换距离信息。它采用的是每个路由器主动地测试与其邻站相连链路的状态，将这些信息发送给它的其他邻站，而邻站将这些信息在自治系统中传播出去。每个路由器接收这些链路状态信息，并建立起完整的路由表。

从实际角度来看，二者的不同点是链路状态协议总是比距离向量协议收敛更快。收敛的意思是在路由发生变化后，例如在路由器关闭或链路出故障后，可以稳定下来。 [Perlman 1992]的9.3节对这两种类型的选路协议的其他方面进行了比较。

OSPF与RIP（以及其他选路协议）的不同点在于， OSPF直接使用IP。也就是说，它并不使用UDP或TCP。对于IP首部的protocol字段，OSPF有其自己的值（图3-1）。

另外，作为一种链路状态协议而不是距离向量协议， OSPF还有着一些优于RIP的特点。

1) OSPF可以对每个IP服务类型（图3-2）计算各自的路由集。这意味着对于任何目的，可以有多个路由表表项，每个表项对应着一个IP服务类型。

2) 给每个接口指派一个无维数的费用。可以通过吞吐率、往返时间、可靠性或其他性能来进行指派。可以给每个IP服务类型指派一个单独的费用。

3) 当对同一个目的地址存在着多个相同费用的路由时，OSPF在这些路由上平均分配流量。我们称之为流量平衡。

4) OSPF支持子网：子网掩码与每个通告路由相连。这样就允许将一个任何类型的 IP地址分割成多个不同大小的子网（我们在 3.7节中给出了这样的一个例子，称之为变长度子网）。到一个主机的路由是通过全1子网掩码进行通告的。默认路由是以IP地址为0.0.0.0、网络掩码为全0进行通告的。

5) 路由器之间的点对点链路不需要每端都有一个 IP地址，我们称之为无编号网络。这样可以节省IP地址 —— 现在非常紧缺的一种资源。

6) 采用了一种简单鉴别机制。可以采用类似于 RIP-2机制（10.5节）的方法指定一个明文口令。

7) OSPF采用多播（第12章），而不是广播形式，以减少不参与OSPF的系统负载。

随着大部分厂商支持OSPF，在很多网络中OSPF将逐步取代RIP。

10.7 BGP：边界网关协议

BGP是一种不同自治系统的路由器之间进行通信的外部网关协议。 BGP是ARPANET所使用的老EGP的取代品。 RFC1267 [Lougheed and Rekhter 1991] 对第3版的BGP进行了描述。

RFC 1268 [Rekhter and Gross 1991] 描述了如何在Internet中使用BGP。下面对于BGP的大部分描述都来自于这两个RFC文档。同时， 1993年开发第4版的BGP（见RFC 1467 [Topolcic 1993]），以支持我们将在10.8节描述的CIDR。

BGP系统与其他BGP系统之间交换网络可到达信息。这些信息包括数据到达这些网络所必须经过的自治系统AS中的所有路径。这些信息足以构造一幅自治系统连接图。然后，可以根据连接图删除选路环，制订选路策略。

首先，我们将一个自治系统中的 IP数据报分成本地流量和通过流量。在自治系统中，本地流量是起始或终止于该自治系统的流量。也就是说，其信源 IP地址或信宿IP地址所指定的主机位于该自治系统中。其他的流量则称为通过流量。在 Internet中使用BGP的一个目的就是

Linux公社 www.linuxidc.com

减少通过流量。

可以将自治系统分为以下几种类型：

1) 残桩自治系统(stub AS)，它与其他自治系统只有单个连接。stub AS只有本地流量。

2) 多接口自治系统(multihomed AS)，它与其他自治系统有多个连接，但拒绝传送通过流量。

3) 转送自治系统(transit AS)，它与其他自治系统有多个连接，在一些策略准则之下，它可以传送本地流量和通过流量。

这样，可以将Internet的总拓扑结构看成是由一些残桩自治系统、多接口自治系统以及转送自治系统的任意互连。残桩自治系统和多接口自治系统不需要使用 BGP——它们通过运行EGP在自治系统之间交换可到达信息。

BGP允许使用基于策略的选路。由自治系统管理员制订策略，并通过配置文件将策略指定给BGP。制订策略并不是协议的一部分，但指定策略允许 BGP实现在存在多个可选路径时选择路径，并控制信息的重发送。选路策略与政治、安全或经济因素有关。

BGP与RIP和OSPF的不同之处在于BGP使用TCP作为其传输层协议。两个运行BGP的系统之间建立一条TCP连接，然后交换整个BGP路由表。从这个时候开始，在路由表发生变化时，再发送更新信号。

BGP是一个距离向量协议，但是与（通告到目的地址跳数的） RIP不同的是，BGP列举了到每个目的地址的路由（自治系统到达目的地址的序列号）。这样就排除了一些距离向量协议的问题。采用16 bit数字表示自治系统标识。

BGP通过定期发送keepalive报文给其邻站来检测TCP连接对端的链路或主机失败。两个报文之间的时间间隔建议值为 30秒。应用层的keepalive报文与TCP的keepalive选项（第23章）是独立的。

10.8 CIDR：无类型域间选路

在第3章中，我们指出了B类地址的缺乏，因此现在的多个网络站点只能采用多个 C类网络号，而不采用单个B类网络号。尽管分配这些C类地址解决了一个问题（B类地址的缺乏），但它却带来了另一个问题：每个 C类网络都需要一个路由表表项。无类型域间选路（ CIDR）是一个防止Internet路由表膨胀的方法，它也称为超网（ supernetting）。在RFC 1518 [Rekher and Li 1993] 和RFC 1519 [Fuller et al. 1993]中对它进行了描述，而[Ford, Rekhter, and Braun 1993]是它的综述。CIDR有一个Internet Architecture Board ’s blessing [Huitema 1993]。RFC 1467 [Topolcic 1993] 对Internet中CIDR的开发状况进行了小结。

CIDR的基本观点是采用一种分配多个 IP地址的方式，使其能够将路由表中的许多表项总和(summarization)成更少的数目。例如，如果给单个站点分配 16个C类地址，以一种可以用总和的方式来分配这 16个地址，这样，所有这 16个地址可以参照Internet上的单个路由表表项。同时，如果有8个不同的站点是通过同一个 Internet服务提供商的同一个连接点接入 Internet的，且这8个站点分配的8个不同IP地址可以进行总和，那么，对于这 8个站点，在Internet上，只需要单个路由表表项。

要使用这种总和，必须满足以下三种特性：

1) 为进行选路要对多个IP地址进行总和时，这些IP地址必须具有相同的高位地址比特。

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

2) 路由表和选路算法必须扩展成根据 32 bit IP地址和32 bit掩码做出选路决策。

3) 必须扩展选路协议使其除了 32 bit地址外，还要有32 bit掩码。OSPF（10.6节）和RIP-2（10.5节）都能够携带第4版BGP所提出的32 bit掩码。

例如，RFC 1466 [Gerich 1993] 建议欧洲新的C类地址的范围是194.0.0.0～195.255.255.255。以16进制表示，这些地址的范围是0xc2000000～0xc3ffffff。它代表了65536个不同的C类网络号，但它们地址的高 7 bit是相同的。在欧洲以外的国家里，可以采用 IP地址为0xc2000000和32 bit 0xfe000000 (254.0.0.0) 为掩码的单个路由表表项来对所有这些 65536个C类网络号选路到单个点上。C类地址的后面各比特位（即在 194或195后面各比特）也可以进行层次分配，例如以国家或服务提供商分配，以允许对在欧洲路由器之间使用除了这 32 bit掩码的高7 bit外的其他比特进行概括。

CIDR同时还使用一种技术，使最佳匹配总是最长的匹配：即在 32 bit掩码中，它具有最大值。我们继续采用上一段中所用的例子，欧洲的一个服务提供商可能会采用一个与其他欧洲服务提供商不同的接入点。如果给该提供商分配的地址组是从 194.0.16.0到194.0.31.255 (16个C类网络号)，那么可能只有这些网络的路由表项的 IP地址是 194.0.16.0，掩码为255.255.240.0 (0xfffff000)。发往194.0.22.1地址的数据报将同时与这个路由表表项和其他欧洲C类地址的表项进行匹配。但是由于掩码 255.255.240比254.0.0.0更“长”，因此将采用具有更长掩码的路由表表项。

“无类型”的意思是现在的选路决策是基于整个 32 bit IP地址的掩码操作，而不管其 IP地址是A类、B类或是C类，都没有什么区别。

CIDR最初是针对新的C类地址提出的。这种变化将使Internet路由表增长的速度缓慢下来，但对于现存的选路则没有任何帮助。这是一个短期解决方案。作为一个长期解决方案，如果将CIDR应用于所有IP地址，并根据各洲边界和服务提供商对已经存在的IP地址进行重新分配（且所有现有主机重新进行编址！），那么[Ford, Rekhter, and Braun 1993] 宣称，目前包含10 000网络表项的路由表将会减少成只有200个表项。

10.9 小结

有两种基本的选路协议，即用于同一自治系统各路由器之间的内部网关协议（ IGP）和用于不同自治系统内路由器通信的外部网关协议（ EGP）。

最常用的IGP是路由信息协议（RIP），而OSPF是一个正在得到广泛使用的新 IGP。一种新近流行的EGP是边界网关协议（BGP）。在本章中，我们讨论了 RIP及其交换的报文类型。第 2版RIP是其最近的一个改进版，它支持子网，还有一些其他改进技术。同时也对 OSPF、BGP和无类型域间选路（ CIDR）进行了描述。CIDR是一种新技术，可以减小 Internet路由表的大小。

你可能还会遇到一些其他的 OSI选路协议。域间选路协议（ IDRP）最开始时，是一个为了使用OSI地址而不是IP地址，而进行修改的 BGP版本。Intermediate System to Intermediate System 协议（IS-IS）是OSI的标准IGP。可以用它来选路 CLNP（无连接网络协议），这是一种与IP类似的OSI协议。IS-IS和OSPF相似。

动态选路仍然是一个网间互连的研究热点。对使用的选路协议和运行的路由守护程序进行选择，是一项复杂的工作。[Perlman 1992]提供了许多细节。

Linux公社 www.linuxidc.com

习题

10.1 在图10-9中哪些路由是从路由器kpno进入gateway的？

10.2 假设一个路由器要使用RIP通告30个路由，这需要一个包含25条路由和另一个包含5条路由的数据报。如果每过一个小时，第一个包含25条路由的数据报丢失一次，那么其结果如何？

10.3 OSPF报文格式中有一个检验和字段，而RIP报文则没有此项，这是为什么？

10.4 像OSPF这样的负载平衡，对于传输层的影响是什么？

10.5 查阅RFC1058关于实现RIP的其他资料。在图10-8中，140.252.1网络的每个路由器只通告它所提供的路由，而它并不能通过其他路由器的广播中知道任何其他路由。这种技术的名称是什么？

10.6 在3.4节中，我们说过除了图10-7中所示的8个路由器外，140.252.1子网上还有超过100个主机。那么这100个主机是如何处理每30秒到达它们的8个广播信息呢（图10-8）？

Linux公社 www.linuxidc.com

第11章　UDP：用户数据报协议

11.1　引言

UDP是一个简单的面向数据报的运输层协议：进程的每个输出操作都正好产生一个 UDP数据报，并组装成一份待发送的 IP数据报。这与面向流字符的协议不同，如 TCP，应用程序产生的全体数据与真正发送的单个 IP数据报可能没有什么联系。

UDP数据报封装成一份 IP数据报的格式如图11-1所示。

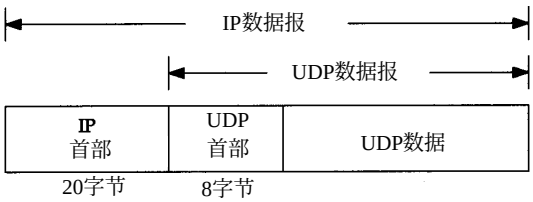


图11-1　UDP封装

RFC 768 [Postel 1980] 是UDP的正式规范。

UDP不提供可靠性：它把应用程序传给 IP层的数据发送出去，但是并不保证它们能到达目的地。由于缺乏可靠性，我们似乎觉得要避免使用 UDP而使用一种可靠协议如 TCP。我们在第17章讨论完TCP后将再回到这个话题，看看什么样的应用程序可以使用 UDP。

应用程序必须关心IP数据报的长度。如果它超过网络的 MTU（2.8节），那么就要对IP数据报进行分片。如果需要，源端到目的端之间的每个网络都要进行分片，并不只是发送端主机连接第一个网络才这样做（我们在 2.9节中已定义了路径MTU的概念）。在11.5节中，我们将讨论IP分片机制。

11.2　UDP首部

UDP首部的各字段如图11-2所示。

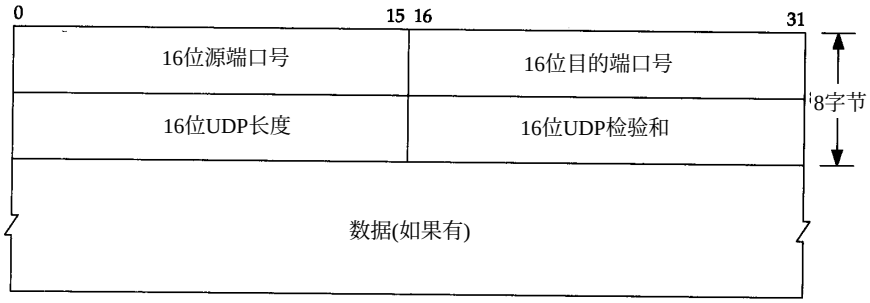


图11-2　UDP首部

端口号表示发送进程和接收进程。在图 1-8中，我们画出了TCP和UDP用目的端口号来分用来自IP层的数据的过程。由于IP层已经把IP数据报分配给TCP或UDP（根据IP首部中协议字段值），因此TCP端口号由TCP来查看，而UDP端口号由UDP来查看。TCP端口号与UDP端口号是相互独立的。

Linux公社 www.linuxidc.com

尽管相互独立，如果TCP和UDP同时提供某种知名服务，两个协议通常选择相同的端口号。这纯粹是为了使用方便，而不是协议本身的要求。

UDP长度字段指的是UDP首部和UDP数据的字节长度。该字段的最小值为8字节（发送一份0字节的UDP数据报是OK）。这个UDP长度是有冗余的。IP数据报长度指的是数据报全长（图3-1），因此UDP数据报长度是全长减去IP首部的长度（该值在首部长度字段中指定，如图3-1所示）。

11.3 UDP检验和

UDP检验和覆盖UDP首部和UDP数据。回想IP首部的检验和，它只覆盖IP的首部——并不覆盖IP数据报中的任何数据。

UDP和TCP在首部中都有覆盖它们首部和数据的检验和。UDP的检验和是可选的，而TCP的检验和是必需的。

尽管UDP检验和的基本计算方法与我们在3.2节中描述的IP首部检验和计算方法相类似（16 bit字的二进制反码和），但是它们之间存在不同的地方。首先，UDP数据报的长度可以为奇数字节，但是检验和算法是把若干个16 bit字相加。解决方法是必要时在最后增加填充字节0，这只是为了检验和的计算（也就是说，可能增加的填充字节不被传送）。

其次，UDP数据报和TCP段都包含一个12字节长的伪首部，它是为了计算检验和而设置的。伪首部包含IP首部一些字段。其目的是让UDP两次检查数据是否已经正确到达目的地（例如，IP没有接受地址不是本主机的数据报，以及IP没有把应传给另一高层的数据报传给UDP）。UDP数据报中的伪首部格式如图11-3所示。

图11-3 UDP检验和计算过程中使用的各个字段

在该图中，我们特地举了一个奇数长度的数据报例子，因而在计算检验和时需要加上填充字节。注意，UDP数据报的长度在检验和计算过程中出现两次。

如果检验和的计算结果为0，则存入的值为全1（65535），这在二进制反码计算中是等效的。如果传送的检验和为0，说明发送端没有计算检验和。

Linux公社 www.linuxidc.com

如果发送端没有计算检验和而接收端检测到检验和有差错，那么 UDP数据报就要被悄悄地丢弃。不产生任何差错报文（当IP层检测到IP首部检验和有差错时也这样做）。

UDP检验和是一个端到端的检验和。它由发送端计算，然后由接收端验证。其目的是为了发现UDP首部和数据在发送端到接收端之间发生的任何改动。

尽管UDP检验和是可选的，但是它们应该总是在用。在 80年代，一些计算机产商在默认条件下关闭UDP检验和的功能，以提高使用 UDP协议的NFS（Network File System）的速度。在单个局域网中这可能是可以接受的，但是在数据报通过路由器时，通过对链路层数据帧进行循环冗余检验（如以太网或令牌环数据帧）可以检测到大多数的差错，导致传输失败。不管相信与否，路由器中也存在软件和硬件差错，以致于修改数据报中的数据。如果关闭端到端的UDP检验和功能，那么这些差错在 UDP数据报中就不能被检测出来。另外，一些数据链路层协议（如SLIP）没有任何形式的数据链路检验和。

> Host Requirements RFC声明，UDP检验和选项在默认条件下是打开的。它还声明，如果发送端已经计算了检验和，那么接收端必须检验接收到的检验和（如接收到检验和不为0）。但是，许多系统没有遵守这一点，只是在出口检验和选项被打开时才验证接收到的检验和。

11.3.1 tcpdump输出

很难知道某个特定系统是否打开了 UDP检验和选项。应用程序通常不可能得到接收到的UDP首部中的检验和。为了得到这一点，作者在 `tcpdump`程序中增加了一个选项，以打印出接收到的UDP检验和。如果打印出的值为0，说明发送端没有计算检验和。

测试网络上三个不同系统的输出如图 11-4所示（参见封面二）。运行我们自编的sock程序（附录C），发送一份包含9个字节数据的UDP数据报给标准回显服务器。

```
1   0.0                  sun.1900 > gemini.echo: udp 9 (UDP cksum=6e90)
2   0.303755 ( 0.3038)   gemini.echo > sun.1900: udp 9 (UDP cksum=0)

3  17.392480 (17.0887)   sun.1904 > aix.echo: udp 9 (UDP cksum=6e3b)
4  17.614371 ( 0.2219)   aix.echo > sun.1904: udp 9 (UDP cksum=6e3b)

5  32.092454 (14.4781)   sun.1907 > solaris.echo: udp 9 (UDP cksum=6e74)
6  32.314378 ( 0.2219)   solaris.echo > sun.1907: udp 9 (UDP cksum=6e74)
```

图11-4 tcpdump 输出，观察其他主机是否打开UDP检验和选项

从这里可以看出，三个系统中有两个打开了 UDP检验和选项。

还要注意的是，在这个简单例子中，送出的数据报与收到的数据报具有相同的检验和值（第3和第4行，第5和第6行）。从图11-3可以看出，两个IP地址进行了交换，正如两个端口号一样。伪首部和UDP首部中的其他字段都是相同的，就像数据回显一样。这再次表明 UDP检验和（事实上，TCP/IP协议簇中所有的检验和）是简单的16 bit和。它们检测不出交换两个16 bit的差错。

> 作者在14.2节中在8个域名服务器中各进行了一次DNS查询。DNS主要使用UDP，结果只有两台服务器打开了UDP检验和选项。

11.3.2 一些统计结果

文献[Mogul 1992]提供了在一个繁忙的NFS服务器上所发生的不同检验和差错的统计结果，

Linux公社 www.linuxidc.com

时间持续了40天。统计数字结果如图11-5所示。

层　次	检验和差错数	近似总分组数
以太网	446	170 000 000
IP	14	170 000 000
UDP	5	140 000 000
TCP	350	30 000 000

图11-5　检测到不同检验和差错的分组统计结果

最后一列是每一行的大概总数，因为以太网和IP层还使用其他的协议。例如，不是所有的以太网数据帧都是IP数据报，至少以太网还要使用ARP协议。不是所有的IP数据报都是UDP或TCP数据，因为ICMP也用IP传送数据。

注意，TCP发生检验和差错的比例与UDP相比要高得多。这很可能是因为在该系统中的TCP连接经常是“远程”连接（经过许多路由器和网桥等中间设备），而UDP一般为本地通信。

从最后一行可以看出，不要完全相信数据链路（如以太网，令牌环等）的CRC检验。应该始终打开端到端的检验和功能。而且，如果你的数据很有价值，也不要完全相信UDP或TCP的检验和，因为这些都只是简单的检验和，不能检测出所有可能发生的差错。

11.4　一个简单的例子

用我们自己编写的sock程序生成一些可以通过tcpdump观察的UDP数据报：

```
bsdi %sock -v -u -i -n4 svr4 discard
connected on 140.252.13.35.1108 to 140.252.13.34.9
bsdi %sock -v -u -i -n4 -w0 svr4 discard
connected on 140.252.13.35.1110 to 140.252.13.34.9
```

第1次执行这个程序时，我们指定verbose模式（-v）来观察ephemeral端口号，指定UDP（-u）而不是默认的TCP，并且指定源模式（-i）来发送数据，而不是读写标准的输入和输出。-n4选项指明输出4份数据报（默认条件下为1024），目的主机为svr4。在1.12节描述了丢弃服务。每次写操作的输出长度取默认值1024。

第2次运行该程序时我们指定-w0，意思是写长度为0的数据报。两个命令的tcpdump输出结果如图11-6所示。

```
1   0.0                  bsdi.1108 > svr4.discard: udp 1024
2   0.002424 ( 0.0024)   bsdi.1108 > svr4.discard: udp 1024
3   0.006210 ( 0.0038)   bsdi.1108 > svr4.discard: udp 1024
4   0.010276 ( 0.0041)   bsdi.1108 > svr4.discard: udp 1024

5  41.720114 (41.7098)   bsdi.1110 > svr4.discard: udp 0
6  41.721072 ( 0.0010)   bsdi.1110 > svr4.discard: udp 0
7  41.722094 ( 0.0010)   bsdi.1110 > svr4.discard: udp 0
8  41.723070 ( 0.0010)   bsdi.1110 > svr4.discard: udp 0
```

图11-6　向一个方向发送UDP数据报时的tcpdump　输出

输出显示有四份1024字节的数据报，接着有四份长度为0的数据报。每份数据报间隔几毫秒（输入第2个命令花了41秒的时间）。

在发送第1份数据报之前，发送端和接收端之间没有任何通信（在第17章，我们将看到TCP在发送数据的第1个字节之前必须与另一端建立连接）。另外，当收到数据时，接收端没有任何确认。在这个例子中，发送端并不知道另一端是否已经收到这些数据报。

最后要指出的是，每次运行程序时，源端的UDP端口号都发生变化。第一次是1108，然后是110。在1.9节我们已经提过，客户程序使用ephemeral端口号一般在1024～5000之间，正

Linux公社 www.linuxidc.com

如我们现在看到的这样。

11.5 IP分片

正如我们在2.8节描述的那样，物理网络层一般要限制每次发送数据帧的最大长度。任何时候IP层接收到一份要发送的IP数据报时，它要判断向本地哪个接口发送数据（选路），并查询该接口获得其MTU。IP把MTU与数据报长度进行比较，如果需要则进行分片。分片可以发生在原始发送端主机上，也可以发生在中间路由器上。

把一份IP数据报分片以后，只有到达目的地才进行重新组装（这里的重新组装与其他网络协议不同，它们要求在下一站就进行进行重新组装，而不是在最终的目的地）。重新组装由目的端的IP层来完成，其目的是使分片和重新组装过程对运输层（ TCP和UDP）是透明的，除了某些可能的越级操作外。已经分片过的数据报有可能会再次进行分片（可能不止一次）。IP首部中包含的数据为分片和重新组装提供了足够的信息。

回忆IP首部（图3-1），下面这些字段用于分片过程。对于发送端发送的每份IP数据报来说，其标识字段都包含一个唯一值。该值在数据报分片时被复制到每个片中（我们现在已经看到这个字段的用途）。标志字段用其中一个比特来表示“更多的片”。除了最后一片外，其他每个组成数据报的片都要把该比特置 1。片偏移字段指的是该片偏移原始数据报开始处的位置。另外，当数据报被分片后，每个片的总长度值要改为该片的长度值。

最后，标志字段中有一个比特称作“不分片”位。如果将这一比特置 1，IP将不对数据报进行分片。相反把数据报丢弃并发送一个 ICMP差错报文（“需要进行分片但设置了不分片比特”，见图6-3）给起始端。在下一节我们将看到出现这个差错的例子。

当IP数据报被分片后，每一片都成为一个分组，具有自己的 IP首部，并在选择路由时与其他分组独立。这样，当数据报的这些片到达目的端时有可能会失序，但是在 IP首部中有足够的信息让接收端能正确组装这些数据报片。

尽管IP分片过程看起来是透明的，但有一点让人不想使用它：即使只丢失一片数据也要重传整个数据报。为什么会发生这种情况呢？因为IP层本身没有超时重传的机制——由更高层来负责超时和重传（TCP有超时和重传机制，但UDP没有。一些UDP应用程序本身也执行超时和重传）。当来自TCP报文段的某一片丢失后，TCP在超时后会重发整个TCP报文段，该报文段对应于一份IP数据报。没有办法只重传数据报中的一个数据报片。事实上，如果对数据报分片的是中间路由器，而不是起始端系统，那么起始端系统就无法知道数据报是如何被分片的。就这个原因，经常要避免分片。文献[Kent and Mogul 1987]对避免分片进行了论述。

使用UDP很容易导致IP分片（在后面我们将看到，TCP试图避免分片，但对于应用程序来说几乎不可能强迫TCP发送一个需要进行分片的长报文段）。我们可以用`sock`程序来增加数据报的长度，直到分片发生。在一个以太网上，数据帧的最大长度是 1500字节（见图2-1），其中1472字节留给数据，假定IP首部为20字节，UDP首部为8字节。我们分别以数据长度为1471, 1472, 1473和1474字节运行sock程序。最后两次应该发生分片：

```
bsdi %sock -u -i -nl -w1471 svr4 discard
bsdi %sock -u -i -nl -w1472 svr4 discard
bsdi %sock -u -i -nl -w1473 svr4 discard
bsdi %sock -u -i -nl -w1474 svr4 discard
```

相应的`tcpdump`输出如图11-7所示。

Linux公社 www.linuxidc.com

```
1   0.0                 bsdi.1112 > svr4.discard: udp 1471
2  21.008303 (21.0083)  bsdi.1114 > svr4.discard: udp 1472
3  50.449704 (29.4414)  bsdi.1116 > svr4.discard: udp 1473 (frag 26304:1480@0+)
4  50.450040 ( 0.0003)  bsdi > svr4: (frag 26304:1@1480)
5  75.328650 (24.8786)  bsdi.1118 > svr4.discard: udp 1474 (frag 26313:1480@0+)
6  75.328982 ( 0.0003)  bsdi > svr4: (frag 26313:2@1480)
```

图11-7 观察UDP数据报分片

前两份UDP数据报（第1行和第2行）能装入以太网数据帧，没有被分片。但是对应于写1473字节的IP数据报长度为1501，就必须进行分片（第3行和第4行）。同理，写1474字节产生的数据报长度为1502，它也需要进行分片（第5行和第6行）。

当IP数据报被分片后，`tcpdump`打印出其他的信息。首先，`frag 26304`（第3行和第4行）和`frag 26313`(第5行和第6行）指的是IP首部中标识字段的值。

分片信息中的下一个数字，即第3行中位于冒号和@号之间的1480，是除IP首部外的片长。两份数据报第一片的长度均为1480：UDP首部占8字节，用户数据占1472字节（加上IP首部的20字节分组长度正好为1500字节）。第1份数据报的第2片（第4行）只包含1字节数据——剩下的用户数据。第2份数据报的第2片（第6行）包含剩下的2字节用户数据。

在分片时，除最后一片外，其他每一片中的数据部分（除 IP首部外的其余部分）必须是8字节的整数倍。在本例中，1480是8的整数倍。

位于@符号后的数字是从数据报开始处计算的片偏移值。两份数据报第1片的偏移值均为0(第3行和第5行)，第2片的偏移值为1480（第4行和第6行）。跟在偏移值后面的加号对应于IP首部中3 bit标志字段中的“更多片”比特。设置这一比特的目的是让接收端知道在什么时候完成所有的分片组装。

最后，注意第4行和第6行（不是第1片）省略了协议名（UDP）、源端口号和目的端口号。协议名是可以打印出来的，因为它在IP首部并被复制到各个片中。但是，端口号在UDP首部，只能在第1片中被发现。

发送的第3份数据报（用户数据为1473字节）分片情况如图11-8所示。需要重申的是，任何运输层首部只出现在第1片数据中。

另外需要解释几个术语：IP数据报是指IP层端到端的传输单元（在分片之前和重新组装之后），分组是指在IP层和链路层之间传送的数据单元。一个分组可以是一个完整的IP数据报，也可以是IP数据报的一个分片。

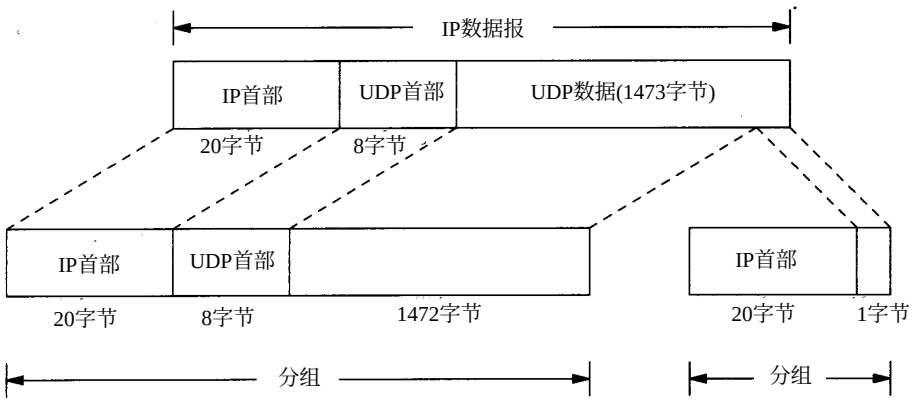


图11-8 UDP分片举例

Linux公社 www.linuxidc.com

11.6 ICMP不可达差错（需要分片）

发生ICMP不可达差错的另一种情况是，当路由器收到一份需要分片的数据报，而在 IP首部又设置了不分片（DF）的标志比特。如果某个程序需要判断到达目的端的路途中最小 MTU是多少 —— 称作路径MTU发现机制（2.9节），那么这个差错就可以被该程序使用。

这种情况下的ICMP不可达差错报文格式如图 11-9所示。这里的格式与图 6-10不同，因为在第2个32 bit字中，16~31 bit可以提供下一站的MTU，而不再是0。

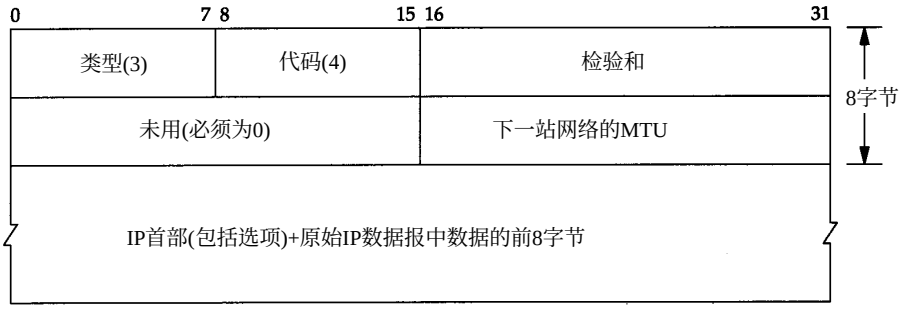


图11-9 需要分片但又设置不分片标志比特时的ICMP不可达差错报文格式

如果路由器没有提供这种新的ICMP差错报文格式，那么下一站的MTU就设为0。

新版的路由器需求RFC [Almquist 1993]声明，在发生这种ICMP不可达差错时，路由器必须生成这种新格式的报文。

例子

关于分片作者曾经遇到过一个问题，ICMP差错试图判断从路由器netb到主机sun之间的拨号SLIP链路的MTU。我们知道从sun到netb的链路的MTU：当SLIP被安装到主机sun时，这是SLIP配置过程中的一部分，加上在3.9节中已经通过netstat命令观察过。现在，我们想从另一个方向来判断它的MTU（在第25章，将讨论如何用SNMP来判断）。在点到点的链路中，不要求两个方向的MTU为相同值。

所采用的技术是在主机solaris上运行ping程序到主机bsdi，增加数据分组长度，直到看见进入的分组被分片为止。如图 11-10所示。

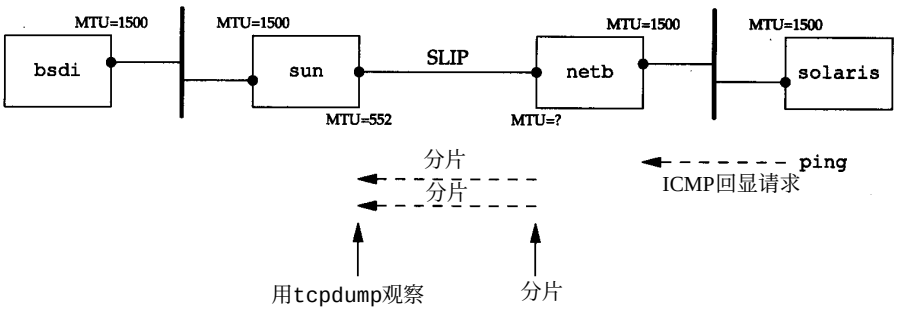


图11-10 用来判断从netb 到sun 的SLIP链路MTU的系统

在主机sun上运行tcpdump，观察SLIP链路，看什么时候发生分片。开始没有观察到分片，一切都很正常直到ping分组的数据长度从500增加到600字节。可以看到接收到的回显请

Linux公社 www.linuxidc.com

求（仍然没有分片），但不见回显应答。

为了跟踪下去，也在主机 bsdi上运行tcpdump，观察它接收和发送的报文。输出如图11-11所示。

```
1  0.0                    solaris > bsdi: icmp: echo request (DF)
2  0.000000 (0.0000)      bsdi > solaris: icmp: echo reply (DF)
3  0.000000 (0.0000)      sun > bsdi: icmp: solaris unreachable -
                          need to frag, mtu = 0 (DF)
4  0.738400 (0.7384)      solaris > bsdi: icmp: echo request (DF)
5  0.748800 (0.0104)      bsdi > solaris: icmp: echo reply (DF)
6  0.748800 (0.0000)      sun > bsdi: icmp: solaris unreachable -
                          need to frag, mtu = 0 (DF)
```

图11-11 600字节的IP数据报从solaris 主机ping 到bsdi 主机时的tcpdump 输出

首先，每行中的标记（DF）说明在IP首部中设置了不分片比特。这意味着 Solaris 2.2 一般把不分片比特置1，作为实现路径MTU发现机制的一部分。

第1行显示的是回显请求通过路由器 netb到达sun主机，没有进行分片，并设置了 DF比特，因此我们知道还没有达到netb的SLIP MTU。

接下来，在第2行注意到DF标志被复制到回显应答报文中。这就带来了问题。回显应答与回显请求报文长度相同（超过 600字节），但是sun外出的SLIP接口MTU为552。因此回显应答需要进行分片，但是DF标志比特又被设置了。这样， sun就产生一个ICMP不可达差错报文返回给bsdi（报文在bsdi处被丢弃）。

这就是我们在主机 solaris上没有看到任何回显应答的原因。这些应答永远不能通过sun。分组的路径如图11-12所示。

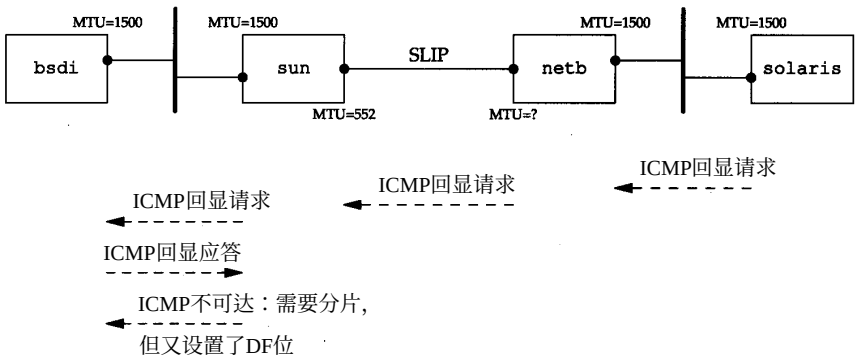


图11-12 例子中的分组交换

最后，在图11-11中的第3行和第6行中， mtu=0表示主机sun没有在ICMP不可达报文中返回出口MTU值，如图11-9所示（在25.9节中，将重新回到这个问题，用 SNMP判断netb上的SLIP接口MTU值为1500）。

11.7 用Traceroute确定路径MTU

尽管大多数的系统不支持路径 MTU发现功能，但可以很容易地修改 traceroute程序（第8章），用它来确定路径 MTU。要做的是发送分组，并设置“不分片”标志比特。发送的第一个分组的长度正好与出口MTU相等，每次收到ICMP“不能分片”差错时（在上一节讨论

Linux公社 www.linuxidc.com

的）就减小分组的长度。如果路由器发送的ICMP差错报文是新格式，包含出口的MTU，那么就用该MTU值来发送，否则就用下一个最小的 MTU值来发送。正如RFC 1191 [Mogul and Deering 1990]声明的那样，MTU值的个数是有限的，因此在我们的程序中有一些由近似值构成的表，取下一个最小MTU值来发送。

首先，我们尝试判断从主机sun到主机slip的路径MTU，知道SLIP链路的MTU为296。

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  15 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, trying new MTU = 1492
fragmentation required and DF set, trying new MTU = 1006
fragmentation required and DF set, trying new MTU = 576
fragmentation required and DF set, trying new MTU = 552
fragmentation required and DF set, trying new MTU = 544
fragmentation required and DF set, trying new MTU = 512
fragmentation required and DF set, trying new MTU = 508
fragmentation required and DF set, trying new MTU = 296
 2  slip (140.252.13.65)  377 ms  377 ms  377 ms
```

在这个例子中，路由器bsdi没有在ICMP差错报文中返回出口MTU，因此我们选择另一个MTU近似值。TTL为2的第1行输出打印的主机名为bsdi，但这是因为它是返回ICMP差错报文的路由器。TTL为2的最后一行正是我们所要找的。

在bsdi上修改ICMP代码使它返回出口MTU值并不困难，如果那样做并再次运行该程序，得到如下输出结果：

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  53 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, next hop MTU = 296
 2  slip (140.252.13.65)  377 ms  378 ms  377 ms
```

这时，在找到正确的MTU值之前，我们不用逐个尝试8个不同的MTU值——路由器返回了正确的MTU值。

全球互联网

作为一个实验，我们多次运行修改以后的traceroute程序，目的端为世界各地的主机。可以到达15个国家（包括南极洲），使用了多个跨大西洋和跨太平洋的链路。但是，在这样做之前，作者所在子网与路由器netb之间的拨号SLIP链路MTU（见图11-12）增加到1500，与以太网相同。

在18次运行当中，只有其中2次发现的路径MTU小于1500。其中一个跨大西洋的链路MTU值为572（其近似值甚至在RFC 1191中也没有被列出），而路由器返回的是新格式的ICMP差错报文。另外一条链路，在日本的两个路由器之间，不能处理1500字节的数据帧，并且路由器没有返回新格式的ICMP差错报文。把MTU值设成1006则可以正常工作。

从这个实验可以得出结论，现在许多但不是所有的广域网都可以处理大于512字节的分组。利用路径MTU发现机制，应用程序就可以充分利用更大的MTU来发送报文。

Linux公社 www.linuxidc.com

11.8 采用UDP的路径MTU发现

下面对使用UDP的应用程序与路径MTU发现机制之间的交互作用进行研究。看一看如果应用程序写了一个对于一些中间链路来说太长的数据报时会发生什么情况。

例子

由于我们所使用的支持路径MTU发现机制的唯一系统就是Solaris 2.x，因此，将采用它作为源站发送一份650字节数据报经slip。由于slip主机位于MTU为296的SLIP链路后，因此，任何长于268字节（296－20－8）且“不分片”比特置为1的UDP数据都会使bsdi路由器产生ICMP“不能分片”差错报文。图11-13给出了拓扑结构和MTU。

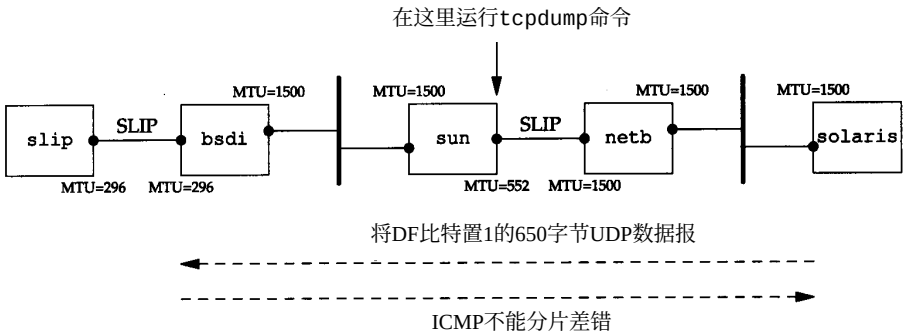


图11-13 使用UDP进行路径MTU发现的系统

可以用下面的命令行来产生650字节UDP数据报，每两个UDP数据报之间的间隔是5秒：

```
solaris %sock -u -i -n10 -w650 -p5 slip discard
```

图11-14是tcpdump的输出结果。在运行这个例子时，将bsdi设置成在ICMP“不能分片”差错中，不返回下一跳MTU信息。

在发送的第一个数据报中将DF比特置1（第1行），其结果是从bsdi路由器发回我们可以猜测的结果（第2行）。令人不解的是，发送一个DF比特置1的数据报（第3行），其结果是同样的ICMP差错（第4行）。我们预计这个数据报在发送时应该将DF比特置0。

第5行结果显示，IP已经知道了发往该目的地址的数据报不能将DF比特置1，因此，IP进而将数据报在源站主机上进行分片。这与前面的例子中，IP发送经过UDP的数据报，允许具有较小MTU的路由器（在本例中是bsdi）对它进行分片的情况不一样。由于ICMP“不能分片”报文并没有指出下一跳的MTU，因此，看来IP猜测MTU为576就行了。第一次分片（第5行）包含544字节的UDP数据、8字节UDP首部以及20字节IP首部，因此，总IP数据报长度是572字节。第2次分片（第6行）包含剩余的106字节UDP数据和20字节IP首部。

不幸的是，第7行的下一个数据报将其DF比特置1，因此bsdi将它丢弃并返回ICMP差错。这时发生了IP定时器超时，通知IP查看是不是因为路径MTU增大了而将DF比特再一次置1。我们可以从第19行和20行看出这个结果。将第7行与19行进行比较，可以看出IP每过30秒就将DF比特置1，以查看路径MTU是否增大了。

这个30秒的定时器值看来太短。RFC1191建议其值取10分钟。可以通过修改ip_ire_pathmtu_interval（E.4节）参数来改变该值。同时，Solaris 2.2无法对单个

Linux公社 www.linuxidc.com

UDP应用或所有UDP应用关闭该路径MTU发现。只能通过修改ip_path_mtu_discovery参数，在系统一级开放或关闭它。正如在这个例子里所能看到的那样，如果允许路径MTU发现，那么当UDP应用程序写入可能被分片数据报时，该数据报将被丢弃。

```
1   0.0                 solaris.38196 > slip.discard: udp 650 (DF)
2   0.004218 (0.0042)   bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)
3   4.980528 (4.9763)   solaris.38196 > slip.discard: udp 650 (DF)
4   4.984503 (0.0040)   bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)
5   9.870407 (4.8859)   solaris.38196 > slip.discard: udp 650 (frag 47942:552@0+)
6   9.960056 (0.0896)   solaris > slip: (frag 47942:106@552)
7  14.940338 (4.9803)   solaris.38196 > slip.discard: udp 650 (DF)
8  14.944466 (0.0041)   bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)
9  19.890015 (4.9455)   solaris.38196 > slip.discard: udp 650 (frag 47944:552@0+)
10 19.950463 (0.0604)   solaris > slip: (frag 47944:106@552)
11 24.870401 (4.9199)   solaris.38196 > slip.discard: udp 650 (frag 47945:552@0+)
12 24.960038 (0.0896)   solaris > slip: (frag 47945:106@552)
13 29.880182 (4.9201)   solaris.38196 > slip.discard: udp 650 (frag 47946:552@0+)
14 29.940498 (0.0603)   solaris > slip: (frag 47946:106@552)
15 34.860607 (4.9201)   solaris.38196 > slip.discard: udp 650 (frag 47947:552@0+)
16 34.950051 (0.0894)   solaris > slip: (frag 47947:106@552)
17 39.870216 (4.9202)   solaris.38196 > slip.discard: udp 650 (frag 47948:552@0+)
18 39.930443 (0.0602)   solaris > slip: (frag 47948:106@552)
19 44.940485 (5.0100)   solaris.38196 > slip.discard: udp 650 (DF)
20 44.944432 (0.0039)   bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)
```

图11-14 使用UDP路径MTU发现

solaris的IP层所假设的最大数据报长度（576字节）是不正确的。在图11-13中，我们看到，实际的MTU值是296字节。这意味着经solaris分片的数据报还将被bsdi分片。图11-15给出了在目的主机（slip）上所收集到的tcpdump对于第一个到达数据报的输出结果（图11-14的第5行和第6行）。

```
1  0.0                 solaris.38196 > slip.discard: udp 650 (frag 47942:272@0+)
2  0.304513 (0.3045)   solaris > slip: (frag 47942:272@272+)
3  0.334651 (0.0301)   solaris > slip: (frag 47942:8@544+)
4  0.466642 (0.1320)   solaris > slip: (frag 47942:106@552)
```

图11-15 从solaris到达slip的第一个数据报

在本例中，solaris不应该对外出数据报分片，它应该将DF比特置0，让具有最小MTU的路由器来完成分片工作。

现在我们运行同一个例子，只是对路由器bsdi进行修改使其在ICMP“不能分片”差错中返回下一跳MTU。图11-16给出了tcpdump输出结果的前6行。

与图11-14一样，前两个数据报同样是将DF比特置1后发送出去的。但是在知道了下一跳MTU后，只产生了3个数据报片，而图11-15中的bsdi路由器则产生了4个数据报片。

Linux公社 www.linuxidc.com

```
1  0.0                  solaris.37974 > slip.discard: udp 650 (DF)
2  0.004199 (0.0042)    bsdi > solaris: icmp:
                               slip unreachable - need to frag, mtu = 296 (DF)

3  4.950193 (4.9460)    solaris.37974 > slip.discard: udp 650 (DF)
4  4.954325 (0.0041)    bsdi > solaris: icmp:
                               slip unreachable - need to frag, mtu = 296 (DF)

5  9.779855 (4.8255)    solaris.37974 > slip.discard: udp 650 (frag 35278:272@0+)
6  9.930018 (0.1502)    solaris > slip: (frag 35278:272@272+)
7  9.990170 (0.0602)    solaris > slip: (frag 35278:114@544)
```

图11-16 使用UDP的路径MTU发现

11.9 UDP和ARP之间的交互作用

使用UDP，可以看到UDP与ARP典型实现之间的有趣的（而常常未被人提及）交互作用。

我们用sock程序来产生一个包含8192字节数据的UDP数据报。预测这将会在以太网上产生6个数据报片（见习题11.3）。同时也确保在运行该程序前， ARP缓存是清空的，这样，在发送第一个数据报片前必须交换ARP请求和应答。

```
bsdi %arp -a                                    验证ARP高速缓存是空的
bsdi %sock -u -i -nl -w8192 svr4 discard
```

预计在发送第一个数据报片前会先发送一个 ARP请求。IP还会产生5个数据报片，这样就提出了我们必须用tcpdump来回答的两个问题：在接收到ARP回答前，其余数据报片是否已经做好了发送准备？如果是这样，那么在 ARP等待应答时，它会如何处理发往给定目的的多个报文？图11-17给出了tcpdump的输出结果。

```
1  0.0                  arp who-has svr4 tell bsdi
2  0.001234 (0.0012)    arp who-has svr4 tell bsdi
3  0.001941 (0.0007)    arp who-has svr4 tell bsdi
4  0.002775 (0.0008)    arp who-has svr4 tell bsdi
5  0.003495 (0.0007)    arp who-has svr4 tell bsdi
6  0.004319 (0.0008)    arp who-has svr4 tell bsdi
7  0.008772 (0.0045)    arp reply svr4 is-at 0:0:c0:c2:9b:26
8  0.009911 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
9  0.011127 (0.0012)    bsdi > svr4: (frag 10863:800@7400)
10 0.011255 (0.0001)    arp reply svr4 is-at 0:0:c0:c2:9b:26
11 0.012562 (0.0013)    arp reply svr4 is-at 0:0:c0:c2:9b:26
12 0.013458 (0.0009)    arp reply svr4 is-at 0:0:c0:c2:9b:26
13 0.014526 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
14 0.015583 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
```

图11-17 在以太网上发送8192字节UDP数据报时的报文交换

在这个输出结果中有一些令人吃惊的结果。首先，在第一个 ARP应答返回以前，总共产生了6个ARP请求。我们认为其原因是 IP很快地产生了6个数据报片，而每个数据报片都引发了一个ARP请求。

第二，在接收到第一个ARP应答时（第7行），只发送最后一个数据报片（第9行）！看来似乎将前5个数据报片全都丢弃了。实际上，这是 ARP的正常操作。在大多数的实现中，在等待一个ARP应答时，只将最后一个报文发送给特定目的主机。

Host Requirements RFC要求实现中必须防止这种类型的ARP洪泛（ARP flooding，

Linux公社 www.linuxidc.com

即以高速率重复发送到同一个IP地址的ARP请求）。建议最高速率是每秒一次。而这里却在4.3 ms内发出了6个ARP请求。

Host Requirements RFC规定，ARP应该保留至少一个报文，而这个报文必须是最后一个报文。这正是我们在这里所看到的结果。

另一个无法解释的不正常的现象是，svr4发回7个，而不是6个ARP应答。

最后要指出的是，在最后一个ARP应答返回后，继续运行tcpdump程序5分钟，以看看svr4是否会返回ICMP“组装超时”差错。并没有发送ICMP差错（我们在图8-2中给出了该消息的格式。code字段为1表示在重新组装数据报时发生了超时）。

在第一个数据报片出现时，IP层必须启动一个定时器。这里“第一个”表示给定数据报的第一个到达数据报片，而不是第一个数据报片（数据报片偏移为0）。正常的定时器值为30或60秒。如果定时器超时而该数据报的所有数据报片未能全部到达，那么将这些数据报片丢弃。如果不这么做，那些永远不会到达的数据报片（正如我们在本例中所看到的那样）迟早会引起接收端缓存满。

这里我们没看到ICMP消息的原因有两个。首先，大多数从Berkeley派生的实现从不产生该差错！这些实现会设置定时器，也会在定时器溢出时将数据报片丢弃，但是不生成ICMP差错。第二，并未接收到包含UDP首部的偏移量为0的第一个数据报片（这是被ARP所丢弃的5个报文的第1个）。除非接收到第一个数据报片，否则并不要求任何实现产生ICMP差错。其原因是因为没有运输层首部，ICMP差错的接收者无法区分出是哪个进程所发送的数据报被丢弃。这里假设上层（TCP或使用UDP的应用程序）最终会超时并重传。

在本节中，我们使用IP数据报片来查看UDP与ARP之间的交互作用。如果发送端迅速发送多个UDP数据报，也可以看到这个交互过程。我们选择采用分片的方法，是因为IP可以生成报文的速度，比一个用户进程生成多个数据报的速度更快。

尽管本例看来不太可能，但它确实经常发生。NFS发送的UDP数据报长度超过8192字节。在以太网上，这些数据报以我们所指出的方式进行分片，如果适当的ARP缓存入口发生超时，那么就可以看到这里所显示的现象。NFS将超时并重传，但是由于ARP的有限队列，第一个IP数据报仍可能被丢弃。

11.10 最大UDP数据报长度

理论上，IP数据报的最大长度是65535字节，这是由IP首部（图3-1）16比特总长度字段所限制的。去除20字节的IP首部和8个字节的UDP首部，UDP数据报中用户数据的最长长度为65507字节。但是，大多数实现所提供的长度比这个最大值小。

我们将遇到两个限制因素。第一，应用程序可能会受到其程序接口的限制。socket API提供了一个可供应用程序调用的函数，以设置接收和发送缓存的长度。对于UDP socket，这个长度与应用程序可以读写的最大UDP数据报的长度直接相关。现在的大部分系统都默认提供了可读写大于8192字节的UDP数据报（使用这个默认值是因为8192是NFS读写用户数据数的默认值）。

第二个限制来自于TCP/IP的内核实现。可能存在一些实现特性（或差错），使IP数据报长度小于65535字节。

作者使用sock程序对不同UDP数据报长度进行了试验。在SunOS 4.1.3下使用环回

Linux公社 www.linuxidc.com

接口的最大IP数据报长度是32767字节。比它大的值都会发生差错。但是从BSD/386到SunOS 4.1.3的情况下，Sun所能接收到最大IP数据报长度为32786字节（即32758字节用户数据）。在Solaris 2.2下使用环回接口，最大可收发IP数据报长度为65535字节。从Solaris 2.2到AIX 3.2.2，发送的最大IP数据报长度可以是65535字节。很显然，这个限制与源端和目的端的实现有关。

我们在3.2节中提过，要求主机必须能够接收最短为576字节的IP数据报。在许多UDP应用程序的设计中，其应用程序数据被限制成512字节或更小，因此比这个限制值小。例如，我们在10.4节中看到，路径信息协议总是发送每份数据报小于 512字节的数据。我们还会在其他UDP应用程序如DNS（第14章）、TFTP（第15章）、BOOTP（第16章）以及SNMP（第25章）中遇到这个限制。

数据报截断

由于IP能够发送或接收特定长度的数据报并不意味着接收应用程序可以读取该长度的数据。因此，UDP编程接口允许应用程序指定每次返回的最大字节数。如果接收到的数据报长度大于应用程序所能处理的长度，那么会发生什么情况呢？

不幸的是，该问题的答案取决于编程接口和实现。

典型的Berkeley版socket API对数据报进行截断，并丢弃任何多余的数据。应用程序何时能够知道，则与版本有关（4.3BSD Reno及其后的版本可以通知应用程序数据报被截断）。

SVR4下的socket API(包括Solaris 2.x) 并不截断数据报。超出部分数据在后面的读取中返回。它也不通知应用程序从单个UDP数据报中多次进行读取操作。

TLI API不丢弃数据。相反，它返回一个标志表明可以获得更多的数据，而应用程序后面的读操作将返回数据报的其余部分。

在讨论TCP时，我们发现它为应用程序提供连续的字节流，而没有任何信息边界。 TCP以应用程序读操作时所要求的长度来传送数据，因此，在这个接口下，不会发生数据丢失。

11.11 ICMP源站抑制差错

我们同样也可以使用UDP产生ICMP“源站抑制(source quench)”差错。当一个系统（路由器或主机）接收数据报的速度比其处理速度快时，可能产生这个差错。注意限定词“可能”。即使一个系统已经没有缓存并丢弃数据报，也不要求它一定要发送源站抑制报文。

图11-18给出了ICMP源站抑制差错报文的格式。有一个很好的方案可以在我们的测试网络里产生该差错报文。可以从bsdi通过必须经过拨号SLIP链路的以太网，将数据报发送给路由器sun。由于SLIP链路的速度大约只有以太网的千分之一，因此，我们很容易就可以使其缓存用完。下面的命令行从主机bsdi通过路由器sun发送100个1024字节长数据报给solaris。我们将数据报发送给标准的丢弃服务，这样，这些数据报将被忽略：

```
bsdi %sock -u -i -w1024 -n100 solaris discard
```

图11-19给出了与此命令行相对应的tcpdump输出结果

在这个输出结果中，删除了很多行，这只是一个模型。接收前 26个数据报时未发生差

Linux公社 www.linuxidc.com

错；我们只给出了第一个数据报的结果。然而，从第 27个数据报开始，每发送一份数据报，就会接收到一份源站抑制差错报文。总共有 26 +（74×2）=174行输出结果。

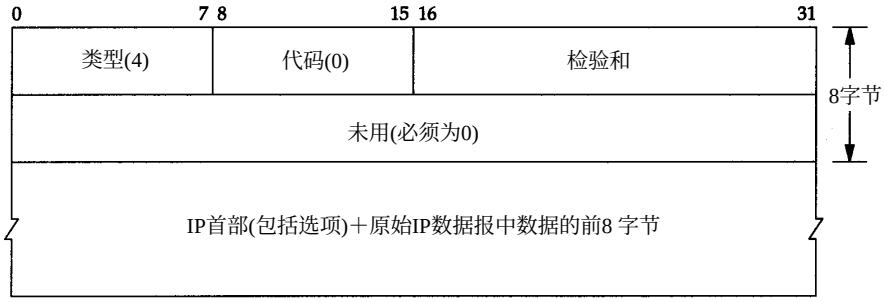


图11-18 ICMP源站抑制差错报文格式

```
 1 0.0                    bsdi.1403 > solaris.discard: udp 1024
                                   26 lines that we don't show
27 0.10 (0.00)            bsdi.1403 > solaris.discard: udp 1024
28 0.11 (0.01)            sun > bsdi: icmp: source quench

29 0.11 (0.00)            bsdi.1403 > solaris.discard: udp 1024
30 0.11 (0.00)            sun > bsdi: icmp: source quench
                                   142 lines that we don't show
173 0.71 (0.06)           bsdi.1403 > solaris.discard: udp 1024
174 0.71 (0.00)           sun > bsdi: icmp: source quench
```

图11-19 来自路由器sun 的ICMP源站抑制

从2.10节的并行线吞吐率计算结果可以知道，以 9600 b/s速率传送1024字节数据报只需要1秒时间（由于从sun到netb的SLIP链路的MTU为552字节，因此在我们的例子中，20 + 8 + 1024字节数据报将进行分片，因此，其时间会稍长一些）。但是我们可以从图 11-19的时间中看出，sun路由器在不到1秒时间内就处理完所有的 100个数据报，而这时，第一份数据报还未通过SLIP链路。因此我们用完其缓存就不足不奇了。

尽管RFC 1009 [Braden and Postel 1987] 要求路由器在没有缓存时产生源站抑制差错报文，但是新的Router Requirements RFC [Almquist 1993] 对此作了修改，提出路由器不应该产生源站抑制差错报文。由于源站抑制要消耗网络带宽，且对于拥塞来说是一种无效而不公平的调整，因此现在人们对于源站抑制差错的态度是不支持的。

在本例中，还需要指出的是，sock程序要么没有接收到源站抑制差错报文，要么接收到却将它们忽略了。结果是如果采用 UDP协议，那么BSD实现通常忽略其接收到的源站抑制报文（正如我们在21.10节所讨论的那样，TCP接受源站抑制差错报文，并将放慢在该连接上的数据传输速度）。其部分原因在于，在接收到源站抑制差错报文时，导致源站抑制的进程可能已经中止了。实际上，如果使用 Unix 的time程序来测定sock程序所运行的时间，其结果是它只运行了大约0.5秒时间。但是从图 11-19中可以看到，在发送第一份数据报过后 0.71秒才接收到一些源站抑制，而此时该进程已经中止。其原因是我们的程序写入了 100个数据报然后中止了。但是所有的100个数据报都已发送出去 —— 有一些数据报在输出队列中。

这个例子重申了 UDP是一个非可靠的协议，它说明了端到端的流量控制。尽管 sock程序成功地将100个数据报写入其网络，但只有 26个数据报真正发送到了目的端。其他 74个数据报

Linux公社 www.linuxidc.com

可能被中间路由器丢弃。除非在应用程序中建立一些应答机制，否则发送端并不知道接收端是否收到了这些数据。

11.12 UDP服务器的设计

使用UDP的一些蕴含对于设计和实现服务器会产生影响。通常，客户端的设计和实现比服务器端的要容易一些，这就是我们为什么要讨论服务器的设计，而不是讨论客户端的设计的原因。典型的服务器与操作系统进行交互作用，而且大多数需要同时处理多个客户。

通常一个客户启动后直接与单个服务器通信，然后就结束了。而对于服务器来说，它启动后处于休眠状态，等待客户请求的到来。对于 UDP来说，当客户数据报到达时，服务器苏醒过来，数据报中可能包含来自客户的某种形式的请求消息。

在这里我们所感兴趣的并不是客户和服务器的编程方面（ [Stevens 1990]对这些方面的细节进行了讨论），而是UDP那些影响使用该协议的服务器的设计和实现方面的协议特性（我们在 18.11节中对TCP服务器的设计进行了描述）。尽管我们所描述的一些特性取决于所使用UDP的实现，但对于大多数实现来说，这些特性是公共的。

11.12.1 客户IP地址及端口号

来自客户的是UDP数据报。IP首部包含源端和目的端 IP地址，UDP首部包含了源端和目的端的UDP端口号。当一个应用程序接收到 UDP数据报时，操作系统必须告诉它是谁发送了这份消息，即源IP地址和端口号。

这个特性允许一个交互 UDP服务器对多个客户进行处理。给每个发送请求的客户发回应答。

11.12.2 目的IP地址

一些应用程序需要知道数据报是发送给谁的，即目的 IP地址。例如，Host Requirements RFC规定，TFTP服务器必须忽略接收到的发往广播地址的数据报（我们分别在第 12章和第15章对广播和TFTP进行描述）。

这要求操作系统从接收到的 UDP数据报中将目的IP地址交给应用程序。不幸的是，并非所有的实现都提供这个功能。

> socket API以IP_RECVDSTADDR socket选项提供了这个功能。对于本文中使用的系统，只有BSD/386、4.4BSD和AIX 3.2.2支持该选项。SVR4、SunOS 4.x和Solaris 2.x都不支持该选项。

11.12.3 UDP输入队列

我们在1.8节中说过，大多数UDP服务器是交互服务器。这意味着，单个服务器进程对单个UDP端口上（服务器上的名知端口）的所有客户请求进行处理。

通常程序所使用的每个 UDP端口都与一个有限大小的输入队列相联系。这意味着，来自不同客户的差不多同时到达的请求将由 UDP自动排队。接收到的UDP数据报以其接收顺序交给应用程序（在应用程序要求交送下一个数据报时）。

Linux公社 www.linuxidc.com

然而，排队溢出造成内核中的 UDP模块丢弃数据报的可能性是存在的。可以进行以下试验。我们在作为UDP服务器的bsdi主机上运行sock程序：

```
bsdi %sock -s -u -v -E -R256 -P30 6666
from 140.252.13.33, to 140.252.13.63: 1111111111 从sun发送到广播地址
from 140.252.13.34, to 140.252.13.35: 4444444444 从svr4发送到单播地址
```

我们指明以下标志：-s表示作为服务器运行，-u表示UDP，-v表示打印客户的 IP地址，-E表示打印目的IP地址（该系统支持这个功能）。另外，我们将这个端口的UDP接收缓存设置为256字节（-R），其每次应用程序读取的大小也是这个数（-r）。标志-P30表示创建UDP端口后，先暂停30秒后再读取第一个数据报。这样，我们就有时间在另两台主机上启动客户程序，发送一些数据报，以查看接收队列是如何工作的。

服务器一开始工作，处于其 30秒的暂停时间内，我们就在 sun主机上启动一个客户，并发送三个数据报：

```
sun % sock -u -v 140.252.13.63 6666            到以太网广播地址
connected on 140.252.13.33.1252 to 140.252.13.63.6666
1111111111                                     11字节的数据（新行）
222222222                                      10字节的数据（新行）
33333333333                                    12字节的数据（新行）
```

目的地址是广播地址（140.252.13.63）。我们同时也在主机svr4上启动第2个客户，并发送另外三个数据报：

```
svr4 %sock -u -v bsdi 6666
connected on 0.0.0.0.1042 to 140.252.13.35.6666
4444444444444                                  14字节的数据（新行）
555555555555555                                16字节的数据（新行）
66666666                                       9字节的数据（新行）
```

首先，我们早些时候在bsdi上所看到的结果表明，应用程序只接收到 2个数据报：来自sun的第一个全1报文，和来自svr4的第一个全4报文。其他4个数据报看来全被丢弃。

图11-20给出的tcpdump输出结果表明，所有6个数据报都发送给了目的主机。两个客户的数据报以交替顺序键入：第一个来自 sun，然后是来自svr4的，以此类推。同时也可以看出，全部6个数据报大约在12秒内发送完毕，也就是在服务器休眠的 30秒内完成的。

```
1   0.0                  sun.1252 > 140.252.13.63.6666: udp 11
2   2.499184 (2.4992)    svr4.1042 > bsdi.6666: udp 14
3   4.959166 (2.4600)    sun.1252 > 140.252.13.63.6666: udp 10
4   7.607149 (2.6480)    svr4.1042 > bsdi.6666: udp 16
5  10.079059 (2.4719)    sun.1252 > 140.252.13.63.6666: udp 12
6  12.415943 (2.3369)    svr4.1042 > bsdi.6666: udp 9
```

图11-20 两个客户发送UDP数据报的tcpdump 输出结果

我们还可以看到，服务器的-E选项使其可以知道每个数据报的目的 IP地址。如果需要，它可以选择如何处理其接收到的第一个数据报，这个数据报的地址是广播地址。

我们可以从本例中看到以下几个要点。首先，应用程序并不知道其输入队列何时溢出。只是由UDP对超出数据报进行丢弃处理。同时，从 tcpdump输出结果，我们看到，没有发回任何信息告诉客户其数据报被丢弃。这里不存在像 ICMP源站抑制这样发回发送端的消息。最后，看来UDP输出队列是FIFO（先进先出）的，而我们在 11.9节中所看到的 ARP输入却是

Linux公社 www.linuxidc.com

LIFO（后进先出）的。

11.12.4 限制本地IP地址

大多数UDP服务器在创建UDP端点时都使其本地IP地址具有通配符(wildcard)的特点。这就表明进入的UDP数据报如果其目的地为服务器端口，那么在任何本地接口均可接收到它。例如，我们以端口号777启动一个UDP服务器：

```
sun % sock -u -s 7777
```

然后，用netstat命令观察端点的状态：

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address      Foreign Address      (state)
udp       0      0     *.7777              *.*
```

这里，我们删除了许多行，只保留了其中感兴趣的东西。 -a选项表示报告所有网络端点的状态。 -n选项表示以点数格式打印IP地址而不用DNS把地址转换成名字，打印数字端口号而不是服务名称。 -f inet选项表示只报告TCP和UDP端点。

本地地址以*.7777格式打印，星号表示任何本地IP地址。

当服务器创建端点时，它可以把其中一个主机本地 IP地址包括广播地址指定为端点的本地IP地址。只有当目的IP地址与指定的地址相匹配时，进入的 UDP数据报才能被送到这个端点。用我们的sock程序，如果在端口号之前指定一个IP地址，那么该IP地址就成为该端点的本地IP地址。例如：

```
sun % sock -u -s 140.252.1.29 7777
```

就限制服务器在SLIP接口(140.252.1.29)处接收数据报。netstat输出结果显示如下：

```
Proto Recv-Q Send-Q     Local Address          Foreign Address      (state)
udp       0      0     140.252.1.29.7777              *.*
```

如果我们试图在以太网上的主机bsdi以地址140.252.13.35向该服务器发送一份数据报，那么将返回一个ICMP端口不可达差错。服务器永远看不到这份数据报。这种情形如图 11-21所示。

```
1  0.0                  bsdi.1723 > sun.7777: udp 13
2  0.000822 (0.0008)    sun > bsdi: icmp: sun udp port 7777 unreachable
```

图11-21 服务器本地地址绑定导致拒绝接收UDP数据报

有可能在相同的端口上启动不同的服务器，每个服务器具有不同的本地 IP地址。但是，一般必须告诉系统应用程序重用相同的端口号没有问题。

使用sockets API时，必须指定SO_REUSEADDR socket选项。在sock程序中是通过-A选项来完成的。

在主机sun上，可以在同一个端口号（8888）上启动5个不同的服务器：

```
sun % sock -u -s 140.252.1.29 8888          对于SLIP链路
sun % sock -u -s -A 140.252.13.33 8888      对于以太网
sun % sock -u -s -A 127.0.0.1 8888          对于环回接口
sun % sock -u -s -A 140.252.13.63 8888      对于以太网广播
sun % sock -u -s -A 8888                    其他(IP地址通配)
```

Linux公社 www.linuxidc.com

除了第一个以外，其他的服务器都必须以-A选项启动，告诉系统可以重用同一个端口号。5个服务器的netstat输出结果如下所示：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
udp        0      0  *.8888                 *.*
udp        0      0  140.252.13.63.8888     *.*
udp        0      0  127.0.0.1.8888         *.*
udp        0      0  140.252.13.33.8888     *.*
udp        0      0  140.252.1.29.8888      *.*
```

在这种情况下，到达服务器的数据报中，只有带星号的本地 IP地址，其目的地址为140.252.1.255，因为其他4个服务器占用了其他所有可能的IP地址。

如果存在一个含星号的IP地址，那么就隐含了一种优先级关系。如果为端点指定了特定IP地址，那么在匹配目的地址时始终优先匹配该 IP地址。只有在匹配不成功时才使用含星号的端点。

11.12.5 限制远端IP地址

在前面所有的netstat输出结果中，远端IP地址和远端端口号都显示为 *.*，其意思是该端点将接受来自任何IP地址和任何端口号的UDP数据报。大多数系统允许UDP端点对远端地址进行限制。

这说明端点将只能接收特定IP地址和端口号的UDP数据报。sock程序用-f选项来指定远端IP地址和端口号：

```
sun % sock -u -s -f 140.252.13.35.4444 5555
```

这样就设置了远端IP地址140.252.13.35（即主机bsdi）和远端端口号4444 。服务器的有名端口号为5555。如果运行netstat命令，我们发现本地IP地址也被设置了，尽管我们没有指定。

```
Proto Recv-Q Send-Q    Local Address          Foreign Address      (state)
udp       0      0   140.252.13.33.5555     140.252.13.35.4444
```

这是在伯克利派生系统中指定远端 IP地址和端口号带来的副作用：如果在指定远端地址时没有选择本地地址，那么将自动选择本地地址。它的值就成为选择到达远端 IP地址路由时将选择的接口 IP地址。事实上，在这个例子中， sun在以太网上的 IP地址与远端地址140.252.13.33相连。

图11-22总结了UDP服务器本身可以创建的三类地址绑定。

本地地址	远端地址	描 述
localIP.lport	foreignIP.fport	只限于一个客户
localIP.lport	*.*	限于到达一个本地接口的数据报：localIP
*.lport	*.*	接收发送到lport的所有数据报

图11-22 为UDP服务器指定本地和远端IP地址及端口号

在所有情况下，lport指的是服务器有名端口号，localIP必须是本地接口的IP地址。表中这三行的排序是UDP模块在判断用哪个端点接收数据报时所采用的顺序。最为确定的地址（第一行）首先被匹配，最不确定的地址（最后一行IP地址带有两个星号）最后进行匹配。

11.12.6 每个端口有多个接收者

尽管在RFC中没有指明，但大多数的系统在某一时刻只允许一个程序端点与某个本地 IP

Linux公社 www.linuxidc.com

地址及UDP端口号相关联。当目的地为该IP地址及端口号的UDP数据报到达主机时，就复制一份传给该端点。端点的IP地址可以含星号，正如我们前面讨论的那样。

例如，在SunOS 4.1.3中，我们启动一个端口号为9999的服务器，本地IP地址含有星号：

```
sun % sock -u -s 9999
```

接着，如果启动另一个具有相同本地地址和端口号的服务器，那么它将不运行，尽管我们指定了-A选项：

```
sun % sock -u -s 9999        我们预计它会失败
can't bind local address: Address already in use
sun % sock -u -s -A 9999     因此，这次尝试-A参数
can't bind local address: Address already in use
```

在一个支持多播的系统上（第12章），这种情况将发生变化。多个端点可以使用同一个IP地址和UDP端口号，尽管应用程序通常必须告诉 API是可行的（如，用 -A标志来指明SO_REUSEADDR socket选项）。

> 4.4BSD支持多播传送，需要应用程序设置一个不同的socket选项（SO_REUSEPORT）以允许多个端点共享同一个端口。另外，每个端点必须指定这个选项，包括使用该端口的第一个端点。

当UDP数据报到达的目的IP地址为广播地址或多播地址，而且在目的 IP地址和端口号处有多个端点时，就向每个端点传送一份数据报的复制（端点的本地 IP地址可以含有星号，它可匹配任何目的IP地址）。但是，如果UDP数据报到达的是一个单播地址，那么只向其中一个端点传送一份数据报的复制。选择哪个端点传送数据取决于各个不同的系统实现。

11.13 小结

UDP是一个简单协议。它的正式规范是RFC 768 [Postel 1980]，只包含三页内容。它向用户进程提供的服务位于IP层之上，包括端口号和可选的检验和。我们用 UDP来检查检验和，并观察分片是如何进行的。

接着，我们讨论了ICMP不可达差错，它是新的路径MTU发现功能中的一部分（2.9节）。用Traceroute和UDP来观察路径MTU发现过程。还查看了UDP和ARP之间的接口，大多数的ARP实现在等待ARP应答时只保留最近传送给目的端的数据报。

当系统接收IP数据报的速率超过这些数据报被处理的速率时，系统可能发送 ICMP源站抑制差错报文。使用UDP时很容易产生这样的ICMP差错。

习题

11.1 在11.5节中，向UDP数据报中写入1473字节用户数据时导致以太网数据报片的发生。在采用以太网IEEE 802封装格式时，导致分片的最小用户数据长度为多少？

11.2 阅读RFC 791[Postel 1981a]，理解为什么除最后一片外，其他片中的数据长度均要求为8字节的整数倍？

11.3 假定有一个以太网和一份8192字节的UDP数据报，那么需要分成多少个数据报片，每个数据报片的偏移和长度为多少？

11.4 继续前一习题，假定这些数据报片要经过一条 MTU为552的SLIP链路。必须记住每一个

Linux公社 www.linuxidc.com

数据报片中的数据（除IP首部外）为8字节的整数倍。那么又将分成多少个数据报片？每个数据报片的偏移和长度为多少？

11.5 一个用UDP发送数据报的应用程序，它把数据报分成 4个数据报片。假定第 1片和第2片到达目的端，而第3片和第4片丢失了。应用程序在10秒钟后超时重发该UDP数据报，并且被分成相同的4片（相同的偏移和长度）。假定这一次接收主机重新组装的时间为60秒，那么当重发的第3片和第4片到达目的端时，原先收到的第 1片和第2片还没有被丢弃。接收端能否把这4片数据重新组装成一份IP数据报？

11.6 你是如何知道图11-15中的片实际上与图11-14中第5行和第6行相对应？

11.7 主机`gemini`开机33天后，`netstat`程序显示48 000 000份IP数据报中由于首部检验和差错被丢弃129份，在30 000 000个TCP段中由于TCP检验和差错而被丢弃 20个。但是，在大约18 000 000份UDP数据报中，因为UDP检验和差错而被丢弃的数据报一份也没有。请说明两个方面的原因（提示：参见图 11-4）。

11.8 在讨论分片时没有提及任何关于 IP首部中的选项——它们是否也要被复制到每个数据报片中，或者只留在第一个数据报片中？我们已经讨论过下面这些 IP选项：记录路由（7.3节）、时间戳（7.4节）、严格和宽松的源站选路（ 8.5节）。你希望分片如何处理这些选项？对照RFC 791检查你的答案。

11.9 在图1-8中，我们说UDP数据报是根据目的UDP端口号进行分配的。这正确吗？

Linux公社 www.linuxidc.com

第12章　广播和多播

12.1　引言

在第1章中我们提到有三种IP地址：单播地址、广播地址和多播地址。本章将更详细地介绍广播和多播。

广播和多播仅应用于 UDP，它们对需将报文同时传往多个接收者的应用来说十分重要。TCP是一个面向连接的协议，它意味着分别运行于两主机（由 IP地址确定）内的两进程（由端口号确定）间存在一条连接。

考虑包含多个主机的共享信道网络如以太网。每个以太网帧包含源主机和目的主机的以太网地址（48bit）。通常每个以太网帧仅发往单个目的主机，目的地址指明单个接收接口，因而称为单播(unicast)。在这种方式下，任意两个主机的通信不会干扰网内其他主机（可能引起争夺共享信道的情况除外）。

然而，有时一个主机要向网上的所有其他主机发送帧，这就是广播。通过 ARP和RARP可以看到这一过程。多播(multicast) 处于单播和广播之间：帧仅传送给属于多播组的多个主机。

为了弄清广播和多播，需要了解主机对由信道传送过来帧的过滤过程。图12-1说明了这一过程。

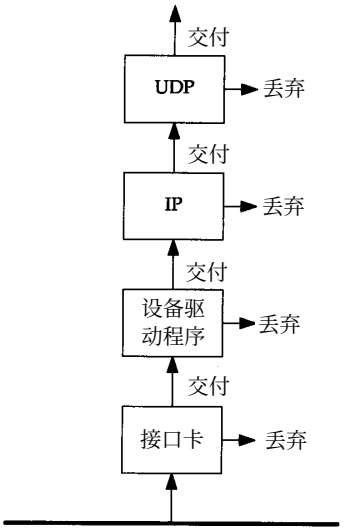


图12-1　协议栈各层对收到帧的过滤过程

首先，网卡查看由信道传送过来的帧，确定是否接收该帧，若接收后就将它传往设备驱动程序。通常网卡仅接收那些目的地址为网卡物理地址或广播地址的帧。另外，多数接口均被设置为混合模式，这种模式能接收每个帧的一个复制。作为一个例子，`tcpdump`使用这种模式。

目前，大多数的网卡经过配置都能接收目的地址为多播地址或某些子网多播地址的帧。对于以太网，当地址中最高字节的最低位设置为 1时表示该地址是一个多播地址，用十六进制可表示为 01:00:00:00:00:00（以太网广播地址ff:ff:ff:ff:ff:ff可看作是以太网多播地址的特例）。

如果网卡收到一个帧，这个帧将被传送给设备驱动程序（如果帧检验和错，网卡将丢弃该帧）。设备驱动程序将进行另外的帧过滤。首先，帧类型中必须指定要使用的协议（ IP、ARP等等）。其次，进行多播过滤来检测该主机是否属于多播地址说明的多播组。

设备驱动程序随后将数据帧传送给下一层，比如，当帧类型指定为 IP数据报时，就传往IP层。IP根据IP地址中的源地址和目的地址进行更多的过滤检测。如果正常，就将数据报传送给下一层（如TCP或UDP）。

每次UDP收到由IP传送来的数据报，就根据目的端口号，有时还有源端口号进行数据报

Linux公社 www.linuxidc.com

过滤。如果当前没有进程使用该目的端口号，就丢弃该数据报并产生一个 ICMP不可达报文（TCP根据它的端口号作相似的过滤）。如果UDP数据报存在检验和错，将被丢弃。

使用广播的问题在于它增加了对广播数据不感兴趣主机的处理负荷。拿一个使用 UDP广播应用作为例子。如果网内有 50个主机，但仅有20个参与该应用，每次这 20个主机中的一个发送UDP广播数据时，其余 30个主机不得不处理这些广播数据报。一直到 UDP层，收到的UDP广播数据报才会被丢弃。这 30个主机丢弃UDP广播数据报是因为这些主机没有使用这个目的端口。

多播的出现减少了对应用不感兴趣主机的处理负荷。使用多播，主机可加入一个或多个多播组。这样，网卡将获悉该主机属于哪个多播组，然后仅接收主机所在多播组的那些多播帧。

12.2 广播

在图3-9中，我们知道了四种IP广播地址，下面对它们进行更详细的介绍。

12.2.1 受限的广播

受限的广播地址是 255.255.255.255。该地址用于主机配置过程中 IP数据报的目的地址，此时，主机可能还不知道它所在网络的网络掩码，甚至连它的 IP地址也不知道。

在任何情况下，路由器都不转发目的地址为受限的广播地址的数据报，这样的数据报仅出现在本地网络中。

一个未解的问题是：如果一个主机是多接口的，当一个进程向本网广播地址发送数据报时，为实现广播，是否应该将数据报发送到每个相连的接口上？如果不是这样，想对主机所有接口广播的应用必须确定主机中支持广播的所有接口，然后向每个接口发送一个数据报复制。

大多数BSD系统将255.255.255.255看作是配置后第一个接口的广播地址，并且不提供向所属具备广播能力的接口传送数据报的功能。不过，`routed`（见10.3节）和`rwhod`（BSD `rwho`客户的服务器）是向每个接口发送 UDP数据报的两个应用程序。这两个应用程序均用相似的启动过程来确定主机中的所有接口，并了解哪些接口具备广播能力。同时，将对应于那种接口的指向网络的广播地址作为发往该接口的数据报的目的地址。

> Host Requirements RFC没有进一步涉及多接口主机是否应当向其所有的接口发送受限的广播。

12.2.2 指向网络的广播

指向网络的广播地址是主机号为全 1的地址。A类网络广播地址为netid.255.255.255，其中netid为A类网络的网络号。

一个路由器必须转发指向网络的广播，但它也必须有一个不进行转发的选择。

12.2.3 指向子网的广播

指向子网的广播地址为主机号为全 1且有特定子网号的地址。作为子网直接广播地址的 IP地址需要了解子网的掩码。例如，如果路由器收到发往 128.1.2.255的数据报，当 B类网络

Linux公社 www.linuxidc.com

128.1的子网掩码为255.255.255.0时，该地址就是指向子网的广播地址；但如果该子网的掩码为255.255.254.0，该地址就不是指向子网的广播地址。

12.2.4 指向所有子网的广播

指向所有子网的广播也需要了解目的网络的子网掩码，以便与指向网络的广播地址区分开。指向所有子网的广播地址的子网号及主机号为全 1。例如，如果目的子网掩码为255.255.255.0，那么IP地址128.1.255.255是一个指向所有子网的广播地址。然而，如果网络没有划分子网，这就是一个指向网络的广播。

当前的看法[Almquist 1993]是这种广播是陈旧过时的，更好的方式是使用多播而不是对所有子网的广播。

> [Almquist 1993] 指出RFC 922要求将一个指向所有子网的广播传送给所有子网，但当前的路由器没有这么做。这很幸运，因为一个因错误配置而没有子网掩码的主机会把它的本地广播传送到所有子网。例如，如果IP地址为128.1.2.3的主机没有设置子网掩码，它的广播地址在正常情况下的默认值是128.1.255.255。但如果子网掩码被设置为255.255.255.0，那么由错误配置的主机发出的广播将指向所有的子网。
>
> 1983年问世的4.2BSD是第一个影响广泛的TCP/IP的实现，它使用主机号全0作为广播地址。一个最早提到广播IP地址的是IEN 212 [Gurwitz and Hinden 1982]，它提出用主机号中的1比特来表示IP广播地址（IENs 是互联网试验注释，基本上是RFC的前身）。RFC 894 [Hornig 1984]认为4.2BSD使用不标准的广播地址，但RFC 906 [Finlayson 1984]注意到对广播地址还没有Internet标准。RFC编辑在RFC 906中加了一个脚注承认缺少标准的广播地址，并强烈推荐将主机号全1作为广播地址。尽管1986年的4.3BSD采用主机号全1表示广播地址，但直到90年代早期，操作系统（著名的是SunOS 4.x）还继续使用非标准的广播地址。

12.3 广播的例子

广播是怎样传送的？路由器及主机又如何处理广播？很遗憾，这是难以回答的问题，因为它依赖于广播的类型、应用的类型、TCP/IP实现方法以及有关路由器的配置。

首先，应用程序必须支持广播。如果执行

```
sun %ping 255.255.255.255
/usr/etc/ping: unknown host 255.255.255.255
```

打算在本地电缆上进行广播。但它无法进行，原因在于该应用程序（ping）中存在一个程序设计上的问题。大多数应用程序收到点分十进制的 IP地址或主机名后，会调用函数inet_addr(3)来把它们转化为32 bit的二进制IP地址。假定要转化的是一个主机名，如果转化失败，该库函数将返回-1来表明存在某种差错（例如是字符而不是数字或串中有小数点）。但本网广播地址（255.255.255.255）也被当作存在差错而返回-1。大多数程序均假定接收到的字符串是主机名，然后查找DNS（第14章），失败后输出差错信息如“未知主机”。

如果我们修复ping程序中这个欠缺，结果也并不总是令人满意的。在6个不同系统的测试中，仅有一个像预期的那样产生了一个本网广播数据报。大多数则在路由表中查找IP地址255.255.255.255，而该地址被用作默认路由器地址，因此向默认路由器单播一个数据报。最

Linux公社 www.linuxidc.com

终该数据报被丢弃。

指向子网的广播是我们应该使用的。在6.3节中，我们向测试网络（见扉页前图）中IP地址为140.252.13.63 的以太网发送数据报，并接收以太网中所有主机的应答。与子网广播地址关联的每个接口是用于命令ifconfig（见3.8节）的值。如果我们ping那个地址，预期的结果是：

```
sun % arp -a                                    ARP高速缓存空

sun % ping 140.252.13.63
PING 140.252.13.63: 56 data bytes
64 bytes from sun (140.252.13.33): icmp_seq=0. time=4. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=0. time=172. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=192. ms

64 bytes from sun (140.252.13.33): icmp_seq=1. time=1. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=1. time=52. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=90. ms
^?                                              键入中断以停止显示
----140.252.13.63 PING Statistics----
2 packets transmitted, 6 packets received, -200% packet loss
round-trip (ms)  min/avg/max = 1/85/192
                                                再检验ARP缓存
sun % arp -a
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

IP通过目的地址（140.252.13.63）来确定，这是指向子网的广播地址，然后向链路层的广播地址发送该数据报。

在6.3节提到的这种广播类型的接收对象为局域网中包括发送主机在内的所有主机，因此可以看到除了收到网内其他主机的答复外，还收到来自发送主机（ sun）的答复。

在这个例子中，我们也显示了执行 ping广播地址前后ARP缓存的内容。这可以显示广播与ARP之间的相互作用。执行 ping命令前ARP缓存是空的，而执行后是满的（也就是说，对网内其他每个响应回显请求的主机在 ARP缓存中均有一个条目）。我们提到的该以太网数据帧被传送到链路层的广播地址（ 0xffffffff）是如何发生的呢？由 sun主机发送的数据帧不需要ARP。

如果使用tcpdump来观察ping的执行过程，可以看到广播数据帧的接收者在发送它的响应之前，首先产生一个对 sun主机的ARP请求，因为它的应答是单播的。在 4.5节我们介绍了一个ARP请求的接收者（该例中是sun）通常在发送ARP应答外，还将请求主机的IP地址和物理地址加入到ARP缓存中去。这基于这样一个假定：如果请求者向我们发送一个数据报，我们也很可能想向它发回什么。

我们使用的ping程序有些特殊，原因在于它使用的编程接口（在大多数 Unix实现中是低级插口(raw socket)）通常允许向一个广播地址发送数据报。如果使用不支持广播的应用如TFTP，情况又如何呢？（TFTP将在第15章详细介绍。）

```
bsdi %tftp                          启动客户程序
tftp> connect 140.252.13.63         说明服务器的IP地址
tftp> get temp.foo                  试图从服务器或获取一个文件
tftp: sendto: Permission denied
tftp> quit                          终止客户程序
```

在这个例子中，程序立即产生了一个差错，但不向网络发送任何信息。产生这一切的原因在于，插口提供的应用程序接口API只有在进程明确打算进行广播时才允许它向广播地址发送UDP

Linux公社 www.linuxidc.com

数据报。这主要是为了防止用户错误地采用了广播地址（正如此例）而应用程序却不打算广播。

在广播UDP数据报之前，使用插口中API的应用程序必须设置SO_BROADCAST插口选项。

并非所有系统均强制使用这个限制。某些系统中无需进程进行这个说明就能广播UDP数据报。而某些系统则有更多的限制，需要有超级用户权限的进程才能广播。

下一个问题是是否转发广播数据。有些系统内核和路由器有一选项来控制允许或禁止这一特性（见附录E）。

如果让路由器bsdi能够转发广播数据，然后在主机slip上运行ping程序，就能够观察到由路由器bsdi转发的子网广播数据报。转发广播数据报意味着路由器接收广播数据，确定该目的地址是对哪个接口的广播，然后用链路层广播向对应的网络转发数据报。

```
slip % ping 140.252.13.63
PING 140.252.13.63 (140.252.13.63): 56 data bytes
64 bytes from 140.252.13.35: icmp_seq=0 ttl=255 time=190 ms
64 bytes from 140.252.13.33: icmp_seq=0 ttl=254 time=280 ms (DUP!)
64 bytes from 140.252.13.34: icmp_seq=0 ttl=254 time=360 ms (DUP!)

64 bytes from 140.252.13.35: icmp_seq=1 ttl=255 time=180 ms
64 bytes from 140.252.13.33: icmp_seq=1 ttl=254 time=270 ms (DUP!)
64 bytes from 140.252.13.34: icmp_seq=1 ttl=254 time=360 ms (DUP!)
^?                                  键入中断以停止显示
--- 140.252.13.63 ping statistics ---
3 packets transmitted, 2 packets received, +4 duplicates, 33% packet loss
round-trip min/avg/max = 180/273/360 ms
```

我们观察到它的确正常工作了，同时也看到BSD系统中的ping程序检查重复的数据报序列号。如果出现重复序列号的数据报就显示DUP!，这意味着一个数据报已经在某处重复了，然而它正是我们所期望看到的，因为我们正向一个广播地址发送数据。

我们还可以从远离广播所指向的网络上的主机上来进行这个试验。在主机angogh.cx.berkeley.edu（和我们的网络距离14跳）上运行ping程序，如果路由器sun被设置为能够转发所指向的广播，它还能正常工作。在这种情况下，这个IP数据报（传送ICMP回显请求）被路径上的每个路由器像正常的数据报一样转发，它们均不知道传送的实际上是广播数据。接着最后一个路由器netb看到主机号为63，就将其转发给路由器sun。路由器sun觉察到该目的IP地址事实上是一个相连子网接口上的广播地址，就将该数据报以链路层广播传往相应网络。

广播是一种应该谨慎使用的功能。在许多情况下，IP多播被证明是一个更好的解决办法。

12.4 多播

IP多播提供两类服务：

1) 向多个目的地址传送数据。有许多向多个接收者传送信息的应用：例如交互式会议系统和向多个接收者分发邮件或新闻。如果不采用多播，目前这些应用大多采用TCP来完成（向每个目的地址传送一个单独的数据复制）。然而，即使使用多播，某些应用可能继续采用TCP来保证它的可靠性。

2) 客户对服务器的请求。例如，无盘工作站需要确定启动引导服务器。目前，这项服务是通过广播来提供的（正如第16章的BOOTP），但是使用多播可降低不提供这项服务主机的负担。

Linux公社 www.linuxidc.com

12.4.1 多播组地址

图12-2显示了D类IP地址的格式。

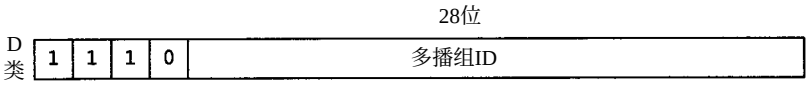


图12-2 D类IP地址格式

不像图1-5所示的其他三类IP地址（A、B和C），分配的28 bit均用作多播组号而不再表示其他。

多播组地址包括为1110的最高4 bit和多播组号。它们通常可表示为点分十进制数，范围从224.0.0.0到239.255.255.255。

能够接收发往一个特定多播组地址数据的主机集合称为主机组 (host group)。一个主机组可跨越多个网络。主机组中成员可随时加入或离开主机组。主机组中对主机的数量没有限制，同时不属于某一主机组的主机可以向该组发送信息。

一些多播组地址被IANA确定为知名地址。它们也被当作永久主机组，这和 TCP及UDP中的熟知端口相似。同样，这些知名多播地址在 RFC最新分配数字中列出。注意这些多播地址所代表的组是永久组，而它们的组成员却不是永久的。

例如，224.0.0.1代表“该子网内的所有系统组”，224.0.0.2代表“该子网内的所有路由器组”。多播地址224.0.1.1用作网络时间协议NTP，224.0.0.9用作RIP-2(见10.5节)，224.0.1.2用作SGI公司的dogfight应用。

12.4.2 多播组地址到以太网地址的转换

IANA拥有一个以太网地址块，即高位 24 bit为00:00:5e（十六进制表示），这意味着该地址块所拥有的地址范围从00:00:5e:00:00:00到00:00:5e:ff:ff:ff。IANA将其中的一半分配为多播地址。为了指明一个多播地址，任何一个以太网地址的首字节必须是 01，这意味着与IP多播相对应的以太网地址范围从01:00:5e:00:00:00到01:00:5e:7f:ff:ff。

> 这里对CSMA/CD或令牌网使用的是Internet标准比特顺序，和在内存中出现的比特顺序一样。这也是大多数程序设计员和系统管理员采用的顺序。IEEE文档采用了这种比特传输顺序。Assigned Numbers RFC给出了这些表示的差别。

这种地址分配将使以太网多播地址中的 23bit与IP多播组号对应起来，通过将多播组号中的低位23bit映射到以太网地址中的低位23bit实现，这个过程如图12-3所示。

由于多播组号中的最高 5 bit在映射过程中被忽略，因此每个以太网多播地址对应的多播组是不唯一的。 32个不同的多播组号被映射为一个以太网地址。例如，多播地址224.128.64.32（十六进制e0.80.40.20）和224.0.64.32（十六进制e0.00.40.20）都映射为同一以太网地址01:00:5e:00:40:20。

既然地址映射是不唯一的，那么设备驱动程序或 IP层（见图12-1）就必须对数据报进行过滤。因为网卡可能接收到主机不想接收的多播数据帧。另外，如果网卡不提供足够的多播数据帧过滤功能，设备驱动程序就必须接收所有多播数据帧，然后对它们进行过滤。

Linux公社 www.linuxidc.com

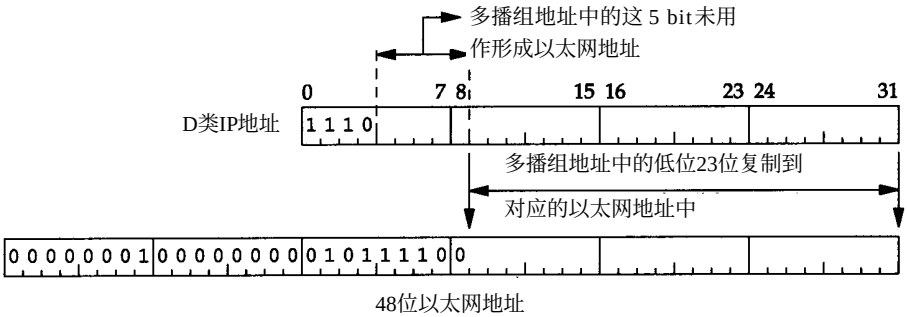


图12-3 D类IP地址到以太网多播地址的映射

局域网网卡趋向两种处理类型：一种是网卡根据对多播地址的散列值实行多播过滤，这意味仍会接收到不想接收的多播数据；另一种是网卡只接收一些固定数目的多播地址，这意味着当主机想接收超过网卡预先支持多播地址以外的多播地址时，必须将网卡设置为“多播混杂(multicast promiscuous)”模式。因此，这两种类型的网卡仍需要设备驱动程序检查收到的帧是否真是主机所需要的。

即使网卡实现了完美的多播过滤（基于48 bit的硬件地址），由于从D类IP地址到48 bit的硬件地址的映射不是一对一的，过滤过程仍是必要的。

尽管存在地址映射不完美和需要硬件过滤的不足，多播仍然比广播好。

单个物理网络的多播是简单的。多播进程将目的 IP地址指明为多播地址，设备驱动程序将它转换为相应的以太网地址，然后把数据发送出去。这些接收进程必须通知它们的 IP层，它们想接收的发往给定多播地址的数据报，并且设备驱动程序必须能够接收这些多播帧。这个过程就是“加入一个多播组”（使用“接收进程”复数形式的原因在于对一确定的多播信息，在同一主机或多个主机上存在多个接收者，这也是为什么要首先使用多播的原因）。当一个主机收到多播数据报时，它必须向属于那个多播组的每个进程均传送一个复制。这和单个进程收到单播UDP数据报的UDP不同。使用多播，一个主机上可能存在多个属于同一多播组的进程。

当把多播扩展到单个物理网络以外需要通过路由器转发多播数据时，复杂性就增加了。需要有一个协议让多播路由器了解确定网络中属于确定多播组的任何一个主机。这个协议就是Internet组管理协议（IGMP），也是下一章介绍的内容。

12.4.3 FDDI和令牌环网络中的多播

FDDI网络使用相同的D类IP地址到48 bit FDDI地址的映射过程[Katz 1990]。令牌环网络通常使用不同的地址映射方法，这是因为大多数令牌控制中的限制。

12.5 小结

广播是将数据报发送到网络中的所有主机（通常是本地相连的网络），而多播是将数据报发送到网络的一个主机组。这两个概念的基本点在于当收到送往上一个协议栈的数据帧时采用不同类型的过滤。每个协议层均可以因为不同的理由丢弃数据报。

目前有四种类型的广播地址：受限的广播、指向网络的广播、指向子网的广播和指向所有子网的广播。最常用的是指向子网的广播。受限的广播通常只在系统初始启动时才会用到。

Linux公社 www.linuxidc.com

试图通过路由器进行广播而发生的问题，常常是因为路由器不了解目的网络的子网掩码。结果与多种因素有关：广播地址类型、配置参数等等。

D类IP地址被称为多播组地址。通过将其低位 23 bit映射到相应以太网地址中便可实现多播组地址到以太网地址的转换。由于地址映射是不唯一的，因此需要其他的协议实现额外的数据报过滤。

习题

12.1 广播是否增加了网络通信量？

12.2 考虑一个拥有50台主机的以太网：20台运行TCP/IP，其他30台运行其他的协议族。主机如何处理来自运行另一个协议族主机的广播？

12.3 登录到一个过去从来没有用过的Unix系统，并且打算找出所有支持广播的接口的指向子网的广播地址。如何做到这点？

12.4 如果我们用ping程序向一个广播地址发送一个长的分组，如

```
sun % ping 140.252.13.63 1472
PING 140.252.13.63: 1472 data bytes
1480 bytes from sun (140.252.13.33): icmp_seq=0. time=6. ms
1480 bytes from svr4 (140.252.13.34): icmp_seq=0. time=84. ms
1480 bytes from bsdi (140.252.13.35): icmp_seq=0. time=128. ms
```

它正常工作，但将分组的长度再增加一个字节后出现如下差错：

```
sun % ping 140.252.13.63 1473
PING 140.252.13.63: 1473 data bytes
sendto: Message too long
```

究竟出了什么问题？

12.5 重做习题10.6，假定8个RIP报文是通过多播而不是广播（使用 RIP 版本2）。有什么变化？

Linux公社 www.linuxidc.com

第13章　IGMP：Internet组管理协议

13.1　引言

12.4节概述了IP多播给出，并介绍了D类IP地址到以太网地址的映射方式。也简要说明了在单个物理网络中的多播过程，但当涉及多个网络并且多播数据必须通过路由器转发时，情况会复杂得多。

本章将介绍用于支持主机和路由器进行多播的Internet组管理协议（IGMP）。它让一个物理网络上的所有系统知道主机当前所在的多播组。多播路由器需要这些信息以便知道多播数据报应该向哪些接口转发。 IGMP在RFC 1112中定义 [Deering 1989]。

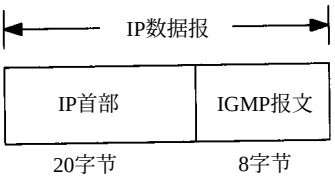


图13-1　IGMP报文封装在IP数据报中

正如ICMP一样， IGMP也被当作IP层的一部分。IGMP报文通过IP数据报进行传输。不像我们已经见到的其他协议， IGMP有固定的报文长度，没有可选数据。图13-1显示了IGMP报文如何封装在IP数据报中。

IGMP报文通过IP首部中协议字段值为2来指明。

13.2　IGMP报文

图13-2显示了长度为8字节的IGMP报文格式。

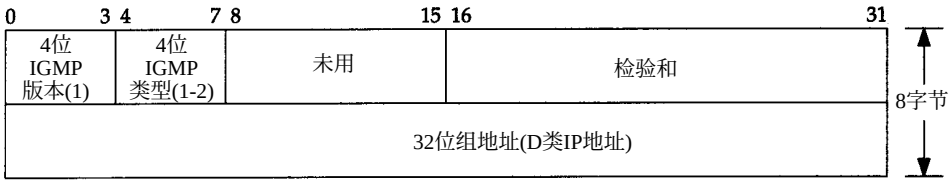


图13-2　IGMP报文的字段格式

这是版本为1的IGMP。IGMP类型为1说明是由多播路由器发出的查询报文，为 2说明是主机发出的报告报文。检验和的计算和ICMP协议相同。

组地址为D类IP地址。在查询报文中组地址设置为 0，在报告报文中组地址为要参加的组地址。在下一节中，当介绍IGMP如何操作时，我们将会更详细地了解它们。

13.3　IGMP 协议

13.3.1　加入一个多播组

多播的基础就是一个进程的概念（使用的术语进程是指操作系统执行的一个程序），该进程在一个主机的给定接口上加入了一个多播组。在一个给定接口上的多播组中的成员是动态

Linux公社 www.linuxidc.com

的——它随时因进程加入和离开多播组而变化。

这里所指的进程必须以某种方式在给定的接口上加入某个多播组。进程也能离开先前加入的多播组。这些是一个支持多播主机中任何 API所必需的部分。使用限定词“接口”是因为多播组中的成员是与接口相关联的。一个进程可以在多个接口上加入同一多播组。

> Stanford大学伯克利版Unix中的IP 多播详细说明了有关socket API的变化，这些变化在Solaris 2.x和ip(7)的文档中也提供了。

这里暗示一个主机通过组地址和接口来识别一个多播组。主机必须保留一个表，此表中包含所有至少含有一个进程的多播组以及多播组中的进程数量。

13.3.2 IGMP 报告和查询

多播路由器使用IGMP报文来记录与该路由器相连网络中组成员的变化情况。使用规则如下：

1) 当第一个进程加入一个组时，主机就发送一个 IGMP报告。如果一个主机的多个进程加入同一组，只发送一个IGMP报告。这个报告被发送到进程加入组所在的同一接口上。

2) 进程离开一个组时，主机不发送 IGMP报告，即便是组中的最后一个进程离开。主机知道在确定的组中已不再有组成员后，在随后收到的 IGMP查询中就不再发送报告报文。

3) 多播路由器定时发送IGMP查询来了解是否还有任何主机包含有属于多播组的进程。多播路由器必须向每个接口发送一个 IGMP查询。因为路由器希望主机对它加入的每个多播组均发回一个报告，因此IGMP查询报文中的组地址被设置为0。

4) 主机通过发送IGMP报告来响应一个IGMP查询，对每个至少还包含一个进程的组均要发回IGMP报告。

使用这些查询和报告报文，多播路由器对每个接口保持一个表，表中记录接口上至少还包含一个主机的多播组。当路由器收到要转发的多播数据报时，它只将该数据报转发到（使用相应的多播链路层地址）还拥有属于那个组主机的接口上。

图13-3显示了两个IGMP报文，一个是主机发送的报告，另一个是路由器发送的查询。该路由器正在要求那个接口上的每个主机说明它加入的每个多播组。

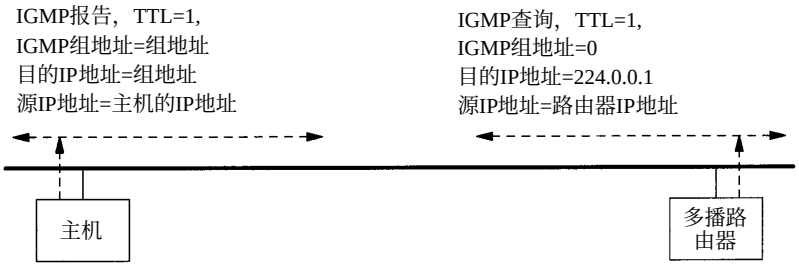


图13-3 IGMP的报告和查询

对TTL字段我们将在本节的后面介绍。

13.3.3 实现细节

为改善该协议的效率，有许多实现的细节要考虑。首先，当一个主机首次发送 IGMP报告

Linux公社 www.linuxidc.com

（当第一个进程加入一个多播组）时，并不保证该报告被可靠接收（因为使用的是IP交付服务）。下一个报告将在间隔一段时间后发送。这个时间间隔由主机在 0~10秒的范围内随机选择。

其次，当一个主机收到一个从路由器发出的查询后，并不立即响应，而是经过一定的时间间隔后才发出一些响应（采用“响应”的复数形式是因为该主机必须对它参加的每个组均发送一个响应）。既然参加同一多播组的多个主机均能发送一个报告，可将它们的发送间隔设置为随机时延。在一个物理网络中的所有主机将收到同组其他主机发送的所有报告，因为如图13-3所示的报告中的目的地址是那个组地址。这意味着如果一个主机在等待发送报告的过程中，却收到了发自其他主机的相同报告，则该主机的响应就可以不必发送了。因为多播路由器并不关心有多少主机属于该组，而只关心该组是否还至少拥有一个主机。的确，一个多播路由器甚至不关心哪个主机属于一个多播组。它仅仅想知道在给定的接口上的多播组中是否还至少有一个主机。

在没有任何多播路由器的单个物理网络中，仅有的 IGMP通信量就是在主机加入一个新的多播组时，支持IP多播的主机所发出的报告。

13.3.4 生存时间字段

在图13-3中，我们注意到IGMP报告和查询的生存时间 (TTL)均设置为1，这涉及到IP首部中的TTL字段。一个初始TTL为0的多播数据报将被限制在同一主机。在默认情况下，待传多播数据报的TTL被设置为1，这将使多播数据报仅局限在同一子网内传送。更大的 TTL值能被多播路由器转发。

回顾6.2节，对发往一个多播地址的数据报从不会产生 ICMP差错。当TTL值为0时，多播路由器也不产生ICMP“超时”差错。

> 在正常情况下，用户进程不关心传出数据报的TTL。然而，一个例外是Traceroute程序（第8章），它主要依据设置TTL值来完成。既然多播应用必须能够设置要传送数据报的TTL值，这意味着程序设计接口必须为用户进程提供这种能力。

通过增加TTL值的方法，一个应用程序可实现对一个特定服务器的扩展环搜索 (expanding ring search)。第一个多播数据报以TTL等于1发送。如果没有响应，就尝试将 TTL设置为2，然后3，等等。在这种方式下，该应用能找到以跳数来度量的最近的服务器。

从224.0.0.0到224.0.0.255的特殊地址空间是打算用于多播范围不超过 1跳的应用。不管TTL值是多少，多播路由器均不转发目的地址为这些地址中的任何一个地址的数据报。

13.3.5 所有主机组

在图13-3中，我们看到了路由器的 IGMP查询被送到目的IP地址224.0.0.1。该地址被称为所有主机组地址。它涉及在一个物理网络中的所有具备多播能力的主机和路由器。当接口初始化后，所有具备多播能力接口上的主机均自动加入这个多播组。这个组的成员无需发送IGMP报告。

13.4 一个例子

现在我们已经了解了一些 IP多播的细节，再来看看所包含的信息。我们使 sun主机能够支

Linux公社 www.linuxidc.com

持多播，并将采用一些多播软件所提供的测试程序来观察具体的过程。

首先，采用一个经过修改的netstat命令来报告每个接口上的多播组成员情况（在 3.9节显示了netstat-ni命令的输出结果）。在下面的输出中，用黑体表示有关的多播组。

```
sun % netstat -nia
Name  Mtu   Network     Address            Ipkts Ierrs    Opkts Oerrs  Coll
le0   1500  140.252.13. 140.252.13.33       4370     0     4924     0     0
                        224.0.0.1
                        08:00:20:03:f6:42
                        01:00:5e:00:00:01
sl0   552   140.252.1   140.252.1.29       13587     0    15615     0     0
                        224.0.0.1
lo0   1536  127         127.0.0.1           1351     0     1351     0     0
                        224.0.0.1
```

其中，-n参数将以数字形式显示 IP地址（而不是按名字来显示它们），-i参数将显示接口的统计结果，-a参数将显示所有配置的接口。

输出结果中的第2行le0（以太网）显示了这个接口属于主机组 224.0.0.1（“所有主机”），和两行地址，后一行显示相应的以太网地址为：01:00:5e:00:00:01。这正是我们期望看到的以太网地址，和 12.4节介绍的地址映射一致。我们还看到其他两个支持多播的接口：SLIP接口sl0和回送接口lo0，它们也属于所有主机组。

我们也必须显示IP路由表，用于多播的路由表同正常的路由表一样。黑体表项显示了所有传往224.0.0.0的数据报均被送往以太网：

```
sun % netstat -rn
Routing tables
Destination          Gateway              Flags    Refcnt Use     Interface
140.252.13.65        140.252.13.35        UGH      0      32      le0
127.0.0.1            127.0.0.1            UH       1      381     lo0
140.252.1.183        140.252.1.29         UH       0      6       sl0
default              140.252.1.183        UG       0      328     sl0
224.0.0.0            140.252.13.33        U        0      66      le0
140.252.13.32        140.252.13.33        U        8      5581    le0
```

如果将这个路由表与 9.2节中sun路由器的路由表作比较，会发现只是多了有关多播的条目。

现在使用一个测试程序来让我们能在一个接口上加入一个多播组（不再显示使用这个测试程序的过程）。在以太网接口（140.252.13.33）上加入多播组224.1.2.3。执行netstat程序看到内核已加入这个组，并得到期望的以太网地址。用黑体字来突出显示和前面 netstat输出的不同。

```
sun % netstat -nia
Name  Mtu   Network     Address            Ipkts Ierrs    Opkts Oerrs  Coll
le0   1500  140.252.13. 140.252.13.33       4374     0     4929     0     0
                        224.1.2.3
                        224.0.0.1
                        08:00:20:03:f6:42
                        01:00:5e:01:02:03
                        01:00:5e:00:00:01
sl0   552   140.252.1   140.252.1.29       13862     0    15943     0     0
                        224.0.0.1
lo0   1536  127         127.0.0.1           1360     0     1360     0     0
                        224.0.0.1
```

我们在输出中再次显示了其他两个接口：sl0和lo0，目的是为了重申加入多播组只发生在一个接口上。

Linux公社 www.linuxidc.com

图13-4显示了tcpdump对进程加入这个多播组的跟踪过程。

```
1  0.0                  8:0:20:3:f6:42 1:0:5e:1:2:3 ip 60:
                        sun > 224.1.2.3: igmp report 224.1.2.3 [ttl 1]

2  6.94 (6.94)          8:0:20:3:f6:42 1:0:5e:1:2:3 ip 60:
                        sun > 224.1.2.3: igmp report 224.1.2.3 [ttl 1]
```

图13-4 当一个主机加入1个多播组时tcpdump 的输出结果

当主机加入多播组时产生第1行的输出显示。第2行是经过时延后的IGMP报告，我们介绍过报告重发的时延是10秒内的随机时延。

在两行中显示硬件地址证实了以太网目的地址就是正确的多播地址。我们也看到了源 IP地址为相应的sun主机地址，而目的IP地址是多播组地址。同时，报告的地址和期望的多播组地址是一致的。

最后，我们注意到，正像指明的那样， TTL是1。当TTL的值为0或1时，tcpdump在打印时用方括号将它们括起来，这是因为 TTL在正常情况下均高于这些值。然而，使用多播我们期望看到许多TTL为1的IP数据报。

在这个输出中暗示了一个多播路由器必须接收在它所有接口上的所有多播数据报。路由器无法确定主机可能加入哪个多播组。

多播路由器的例子

继续前面的例子，但我们将在 sun主机中启动一个多播选路的守护程序。这里我们感兴趣的并不是多播选路协议，而是要研究所交换的 IGMP查询和报告。即使多播选路守护程序只运行在支持多播的主机（ sun）上，所有的查询和报告都将在那个以太网上进行多播，所以我们在该以太网中的其他系统中也能观察到它们。

在启动选路守护程序之前，加入另外一个多播组 224.9.9.9，图13-5显示了输出的结果。

```
1    0.0                 sun > 224.0.0.4: igmp report 224.0.0.4

2    0.00 (  0.00)       sun > 224.0.0.1: igmp query
3    5.10 (  5.10)       sun > 224.9.9.9: igmp report 224.9.9.9

4    5.22 (  0.12)       sun > 224.0.0.1: igmp query
5    7.90 (  2.68)       sun > 224.1.2.3: igmp report 224.1.2.3
6    8.50 (  0.60)       sun > 224.0.0.4: igmp report 224.0.0.4
7   11.70 (  3.20)       sun > 224.9.9.9: igmp report 224.9.9.9

8  125.51 (113.81)       sun > 224.0.0.1: igmp query
9  125.70 (  0.19)       sun > 224.9.9.9: igmp report 224.9.9.9
10 128.50 (  2.80)       sun > 224.1.2.3: igmp report 224.1.2.3
11 129.10 (  0.60)       sun > 224.0.0.4: igmp report 224.0.0.4

12 247.82 (118.72)       sun > 224.0.0.1: igmp query
13 248.09 (  0.27)       sun > 224.1.2.3: igmp report 224.1.2.3
14 248.69 (  0.60)       sun > 224.0.0.4: igmp report 224.0.0.4
15 255.29 (  6.60)       sun > 224.9.9.9: igmp report 224.9.9.9
```

图13-5 当多播选路守护程序运行时tcpdump 的输出结果

在这个输出中没有包括以太网地址，因为已经证实了它们是正确的。也删去了 TTL等于1的说明，同样因为它们也是我们期望的那样。

当选路守护程序启动时，输出第1行。它发出一个已经加入了组224.0.0.4的报告。多播地址224.0.0.4是一个知名的地址，它被当前用于多播选路的距离向量多播选路协议 DVMRP

Linux公社 www.linuxidc.com

(Distance Vector Multicast Routing Protocol)所使用（DVMRP在RFC 1075中定义[Waitzman, Partridge, and Deering]）。

在该守护程序启动时，它也发送一个 IGMP查询（第 2行）。该查询的目的 IP地址为224.0.0.1（所有主机组），如图13-3所示。

第一个报告（第3行）大约在5秒后收到，报告给组224.9.9.9。这是在下一个查询发出之前（第4行）收到的唯一报告。当守护程序启动后，两次查询（第 2行和第4行）发出的间隔很短，这是因为守护程序要将其多播路由表尽快建立起来。

第5、6和7行正是我们期望看到的：sun主机针对它所属的每个组发出一个报告。注意组224.0.0.4是被报告的，而其他两个组则是明确加入的，因为只要选路守护程序还在运行，它始终要属于组224.0.0.4。

下一个查询位于第 8行，大约在前一个查询的 2分钟后发出。它再次引发三个我们所期望的报告（第9、10和11行）。这些报告的时间顺序与前面不同，因为接收查询和发送报告的时间是随机的。

最后的查询在前一个查询的大约2分钟后发出，我们再次得到了期望的响应。

13.5 小结

多播是一种将报文发往多个接收者的通信方式。在许多应用中，它比广播更好，因为多播降低了不参与通信的主机的负担。简单的主机成员报告协议 (IGMP)是多播的基本模块。

在一个局域网中或跨越邻近局域网的多播需要使用本章介绍的技术。广播通常局限在单个局域网中，对目前许多使用广播的应用来说，可采用多播来替代广播。

然而，多播还未解决的一个问题是在广域网内的多播。 [Deering and Cheriton 1990] 提出扩展目前的路由协议来支持多播。 9.13节中的[Perlman 1992]讨论了广域网多播的一些问题。

[Casner and Deering 1992] 介绍了使用多播和一个称为MBONE（多播主干）的虚拟网络在整个Internet上传送IETF会议的情况。

习题

13.1 我们知道主机通过设置随机时延来调度 IGMP的发送。一个局域网中的主机采取什么措施才能避免两台主机产生相同的随机时延？

13.2 在[Casner and Deering 1992]中，他们提到UDP缺少两个通过MBONE传送音频采样数据的条件：分组失序检测和分组重复检测。你怎样在 UDP上增加这些功能？

Linux公社 www.linuxidc.com

第14章　DNS：域名系统

14.1　引言

域名系统（DNS）是一种用于TCP/IP应用程序的分布式数据库，它提供主机名字和 IP地址之间的转换及有关电子邮件的选路信息。这里提到的分布式是指在 Internet上的单个站点不能拥有所有的信息。每个站点（如大学中的系、校园、公司或公司中的部门）保留它自己的信息数据库，并运行一个服务器程序供 Internet上的其他系统（客户程序）查询。 DNS提供了允许服务器和客户程序相互通信的协议。

从应用的角度上看，对 DNS的访问是通过一个地址解析器（ resolver）来完成的。在Unix主机中，该解析器主要是通过两个库函数 `gethostbyname`(3) 和`gethostbyaddr`(3)来访问的，它们在编译应用程序时与应用程序连接在一起。前者接收主机名字返回 IP地址，而后者接收IP地址来寻找主机名字。解析器通过一个或多个名字服务器来完成这种相互转换。

图4-2中指出了解析器通常是应用程序的一部分。解析器并不像 TCP/IP协议那样是操作系统的内核。该图指出的另一个基本概念就是：在一个应用程序请求 TCP打开一个连接或使用UDP发送一个数据报之前。心须将一个主机名转换为一个 IP地址。操作系统内核中的 TCP/IP协议族对于DNS一点都不知道。

本章我们将了解地址解析器如何使用 TCP/IP协议（主要是UDP）与名字服务器通信。我们不介绍运行名字服务器或有关可选参数的细节，这些技术细节的内容可以覆盖整整一本书。（见[Albitz and Liu 1992]标准Unix解析器和名字服务器介绍）。

RFC 1034 [Mockapetris 1987a] 说明了DNS的概念和功能， RFC 1035 [Mockapetris 1987b] 详细说明了DNS 的规范和实现。DNS最常用的版本（包括解析器和名字服务器）是 BIND——伯克利Internet域名服务器。该服务器称作 named。[Danzig、Obraczka和Kumar 1992]分析了DNS 在广域网中产生的通信量。

14.2　DNS 基础

DNS的名字空间和Unix的文件系统相似，也具有层次结构。图 14-1 显示了这种层次的组织形式。

每个结点（图 14-1中的圆圈）有一个至多 63个字符长的标识。这颗树的树根是没有任何标识的特殊结点。命名标识中一律不区分大写和小写。命名树上任何一个结点的域名就是将从该结点到最高层的域名串连起来，中间使用一个点“ . ”分隔这些域名（注意这和 Unix文件系统路径的形成不同，文件路径是由树根依次向下的形成的）。域名树中的每个结点必须有一个唯一的域名，但域名树中的不同结点可使用相同的标识。

以点“ . ”结尾的域名称为绝对域名或完全合格的域名 FQDN（Full Qualified Domain Name），例如`sun.tuc.noao.edu.`。如果一个域名不以点结尾，则认为该域名是不完全的。如何使域名完整依赖于使用的 DNS软件。如果不完整的域名由两个或两个以上的标号组成，

Linux公社 www.linuxidc.com

则认为它是完整的；或者在该域名的右边加入一个局部后缀。例如域名 sun通过加上局部后缀.tuc.noao.edu.成为完整的。

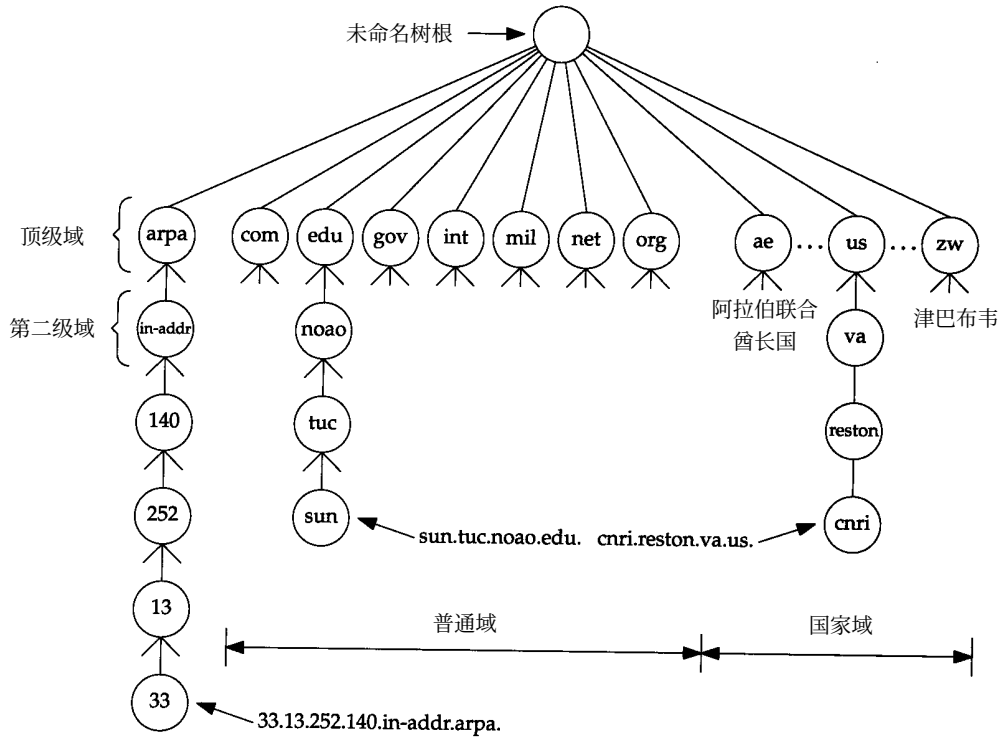


图14-1 DNS的层次组织

顶级域名被分为三个部分：

1) arpa是一个用作地址到名字转换的特殊域（我们将在 14.5节介绍）。

2) 7个3字符长的普通域。有些书也将这些域称为组织域。

3) 所有2字符长的域均是基于ISO3166中定义的国家代码，这些域被称为国家域，或地理域。

图14-2列出了7个普通域的正式划分。

域	描 述
com	商业组织
edu	教育机构
gov	其他美国政府部门
int	国际组织
mil	美国军事网点
net	网络
org	其他组织

图14-2 3字符长的普通域

在DNS中，通常认为3字符长的普通域仅用于美国的组织机构，2字符长的国家域则用于每个国家，但情况并不总是这样。许多非美国的组织机构仍然使用普通域，而一些美国的组织机构也使用 .us的国家域（RFC 1480 [Cooper and Postel 1993] 详细描述了.us域）。普通域中只有.gov和.mil域局限于美国。

许多国家将它们的二级域组织成类似于普通域的结构：例如，.ac.uk是英国研究机构的二级域名，.co.uk则是英国商业机构的二级域名。

DNS的一个没在如图 14-1中表示出来的重要特征是 DNS中域名的授权。没有哪个机构来管理域名树中的每个标识，相反，只有一个机构，即网络信息中心 NIC负责分配顶级域和委派其他指定地区域的授权机构。

Linux公社 www.linuxidc.com

一个独立管理的 DNS子树称为一个区域 (zone)。一个常见的区域是一个二级域，如 noao.edu。许多二级域将它们的区域划分成更小的区域。例如，大学可能根据不同的系来划分区域，公司可能根据不同的部门来划分区域。

如果你熟悉Unix的文件系统，会注意到DNS树中区域的划分同一个逻辑Unix文件系统到物理磁盘分区的划分很相似。正如无法确定图 14-1中区域的具体位置，我们也不知道一个Unix文件系统中的目录位于哪个磁盘分区。

一旦一个区域的授权机构被委派后，由它负责向该区域提供多个名字服务器。当一个新系统加入到一个区域中时，该区域的 DNS管理者为该新系统申请一个域名和一个 IP地址，并将它们加到名字服务器的数据库中。这就是授权机构存在的必要性。例如，在一个小规模的大学，一个人就能完成每次新系统的加入。但对一个规模较大的大学来说，这一工作必须被专门委派的机构（可能是系）来完成，因为一个人已无法维持这一工作。

一个名字服务器负责一个或多个区域。一个区域的管理者必须为该区域提供一个主名字服务器和至少一个辅助名字服务器。主、辅名字服务器必须是独立和冗余的，以便当某个名字服务器发生故障时不会影响该区域的名字服务。

主、辅名字服务器的主要区别在于主名字服务器从磁盘文件中调入该区域的所有信息，而辅名字服务器则从主服务器调入所有信息。我们将辅名字服务器从主服务器调入信息称为区域传送。

当一个新主机加入一个区域时，区域管理者将适当的信息（最少包括名字和 IP地址）加入到运行在主名字服务器上的一个磁盘文件中，然后通知主名字服务器重新调入它的配置文件。辅名字服务器定时（通常是每隔 3小时）向主名字服务器询问是否有新数据。如果有新数据，则通过区域传送方式获得新数据。

当一个名字服务器没有请求的信息时，它将如何处理？它必须与其他的名字服务器联系。(这正是DNS的分布特性)。然而，并不是每个名字服务器都知道如何同其他名字服务器联系。相反，每个名字服务器必须知道如何同根的名字服务器联系。 1993年4月时有8个根名字服务器，所有的主名字服务器都必须知道根服务器的 IP地址（这些IP地址在主名字服务器的配置文件中，主服务器必须知道根服务器的 IP地址，而不是它们的域名)。根服务器则知道所有二级域中的每个授权名字服务器的名字和位置（即 IP地址)。这意味着这样一个反复的过程：正在处理请求的名字服务器与根服务器联系，根服务器告诉它与另一个名字服务器联系。在本章的后面我们将通过一些例子来详细了解这一过程。

你可以通过匿名的FTP获取当前的根服务器清单。具体是从ftp.rs.internic.net或nic.ddn.mil 获取文件netinfo/root-servers.txt。

DNS的一个基本特性是使用超高速缓存。即当一个名字服务器收到有关映射的信息（主机名字到IP地址）时，它会将该信息存放在高速缓存中。这样若以后遇到相同的映射请求，就能直接使用缓存中的结果而无需通过其他服务器查询。 14.7节显示了一个使用高速缓存的例子。

14.3 DNS的报文格式

DNS定义了一个用于查询和响应的报文格式。图 14-3显示这个报文的总体格式。

Linux公社 www.linuxidc.com

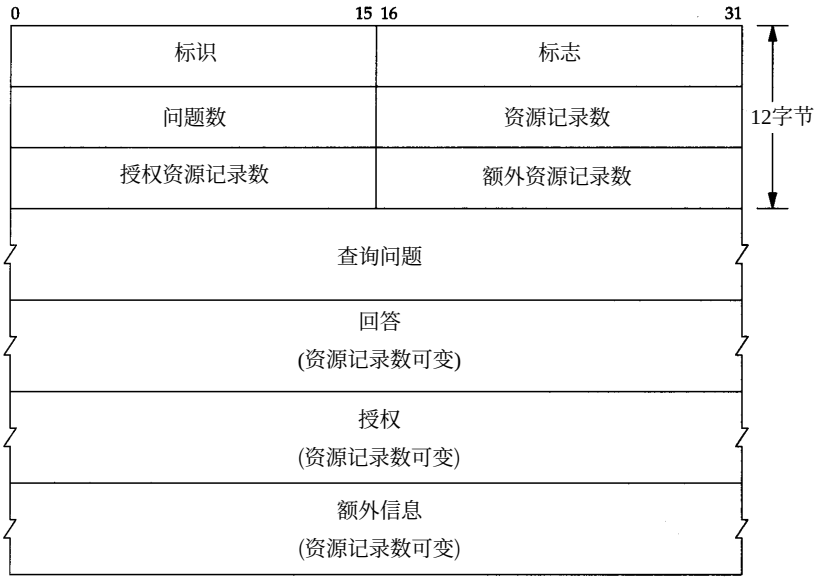


图14-3 DNS查询和响应的一般格式

这个报文由12字节长的首部和4个长度可变的字段组成。

标识字段由客户程序设置并由服务器返回结果。客户程序通过它来确定响应与查询是否匹配。

16 bit的标志字段被划分为若干子字段，如图 14-4所示。

图14-4 DNS报文首部中的标志字段

我们从最左位开始依次介绍各子字段：

- QR 是1 bit字段：0表示查询报文，1表示响应报文。
- opcode是一个4 bit字段：通常值为0（标准查询），其他值为1（反向查询）和2（服务器状态请求）。
- AA是1 bit标志，表示“授权回答（authoritative answer）”。该名字服务器是授权于该域的。
- TC是1 bit字段，表示“可截断的（truncated）”。使用UDP时，它表示当应答的总长度超过512字节时，只返回前512个字节。
- RD是1 bit字段表示“期望递归（recursion desired）”。该比特能在一个查询中设置，并在响应中返回。这个标志告诉名字服务器必须处理这个查询，也称为一个递归查询。如果该位为0，且被请求的名字服务器没有一个授权回答，它就返回一个能解答该查询的其他名字服务器列表，这称为迭代查询。在后面的例子中，我们将看到这两种类型查询的例子。
- RA是1 bit字段，表示“可用递归”。如果名字服务器支持递归查询，则在响应中将该比特设置为1。在后面的例子中可看到大多数名字服务器都提供递归查询，除了某些根服务器。

Linux公社 www.linuxidc.com

- 随后的3 bit字段必须为0。
- rcode是一个4 bit的返回码字段。通常的值为0（没有差错）和3（名字差错）。名字差错只有从一个授权名字服务器上返回，它表示在查询中制定的域名不存在。

随后的4个16 bit字段说明最后4个变长字段中包含的条目数。对于查询报文，问题(question)数通常是1，而其他3项则均为0。类似地，对于应答报文，回答数至少是1，剩下的两项可以是0或非0。

14.3.1 DNS查询报文中的问题部分

问题部分中每个问题的格式如图14-5所示，通常只有一个问题。

图14-5 DNS查询报文中问题部分的格式

查询名是要查找的名字，它是一个或多个标识符的序列。每个标识符以首字节的计数值来说明随后标识符的字节长度，每个名字以最后字节为0结束，长度为0的标识符是根标识符。计数字节的值必须是0~63的数，因为标识符的最大长度仅为63（在本节的后面我们将看到计数字节的最高两比特为1，即值192~255，将用于压缩格式）。不像我们已经看到的许多其他报文格式，该字段无需以整32 bit边界结束，即无需填充字节。

图14-6显示了如何存储域名gemini.tuc.noao.edu。

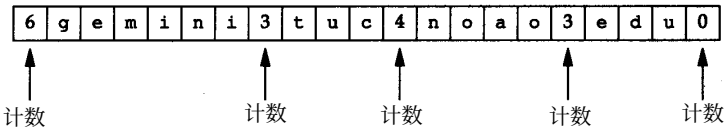


图14-6 域名gemini.tuc.noao.edu 的表示

每个问题有一个查询类型，而每个响应（也称一个资源记录，我们下面将谈到）也有一个类型。大约有20个不同的类型值，其中的一些目前已经过时。图14-7显示了其中的一些值。查询类型是类型的一个超集(superset)：图中显示的类型值中只有两个能用于查询类型。

名　字	数　值	描　述	类型?	查询类型
A	1	IP地址	•	•
NS	2	名字服务器	•	•
CNAME	5	规范名称	•	•
PTR	12	指针记录	•	•
HINFO	13	主机信息	•	•
MX	15	邮件交换记录	•	•
AXFR	252	对区域转换的请求		•
* 或ANY	255	对所有记录的请求		•

图14-7 DNS问题和响应的类型值和查询类型值

Linux公社 www.linuxidc.com

最常用的查询类型是A类型，表示期望获得查询名的IP地址。一个PTR查询则请求获得一个IP地址对应的域名。这是一个指针查询，我们将在14.5节介绍。其他的查询类型将在14.6节介绍。

查询类通常是1，指互联网地址（某些站点也支持其他非IP地址）。

14.3.2 DNS响应报文中的资源记录部分

DNS报文中最后的三个字段，回答字段、授权字段和附加信息字段，均采用一种称为资源记录RR（Resource Record）的相同格式。图14-8显示了资源记录的格式。

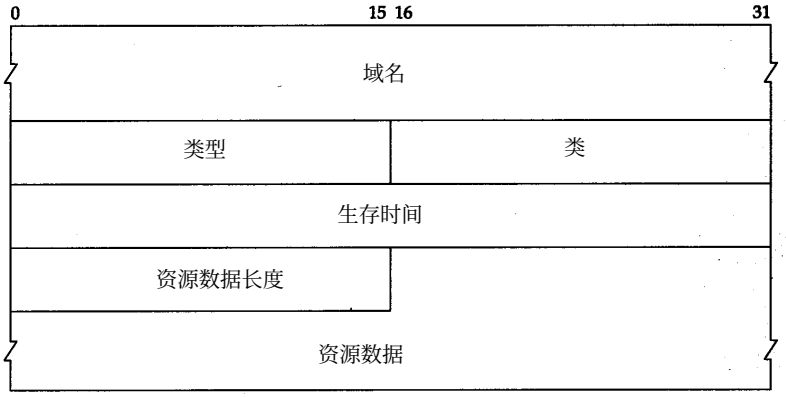


图14-8 DNS资源记录格式

域名是记录中资源数据对应的名字。它的格式和前面介绍的查询名字段格式（图 14-6）相同。

类型说明RR的类型码。它的值和前面介绍的查询类型值是一样的。类通常为 1，指Internet数据。

生存时间字段是客户程序保留该资源记录的秒数。资源记录通常的生存时间值为2天。

资源数据长度说明资源数据的数量。该数据的格式依赖于类型字段的值。对于类型 1（A记录）资源数据是4字节的IP地址。

现在已经介绍了DNS查询和响应的基本格式，我们将使用 `tcpdump`程序来观察具体的交换过程。

14.4 一个简单的例子

让我们从一个简单的例子来了解一个名字解析器与一个名字服务器之间的通信过程。在`sun`主机上运行Telnet客户程序远程登录到`gemini`主机上，并连接daytime服务器：

```
sun % telnet gemini daytime
Trying 140.252.1.11 ...                前3行的输出是从Telnet客户
Connected to gemini.tuc.noao.edu.
Escape character is '^]'.
Wed Mar 24 10:44:17 1993               这是从daytime服务器的输出
Connection closed by foreign host.     这是从Telnet客户的输出
```

在这个例子中，我们引导 `sun`主机（运行Telnet客户程序）上的名字解析器来使用位于`noao.edu`（140.252.1.54）的名字服务器。图14-9显示了这三个系统的排列情况。

Linux公社 www.linuxidc.com

和以前提到的一样，名字解析器是客户程序的一部分，并且在Telnet客户程序与daytime服务器建立TCP连接之前，名字解析器就能通过名字服务器获取IP地址。

在这个图中，省略了sun主机与140.252.1以太网的连接实际上是一个SLIP连接的细节（参见封2的插图），因为它不影响我们的讨论。通过在SLIP链路上运行tcpdump程序来了解名字解析器与名字服务器之间的分组交换。

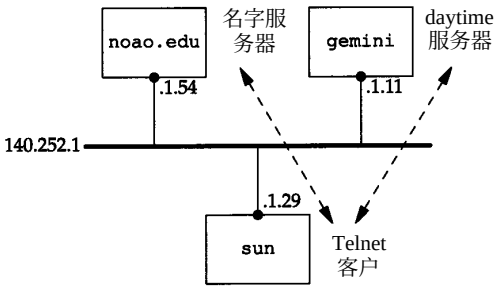


图14-9 用于简单DNS例子的系统

sun主机上的文件/etc/resolv.conf将告诉名字解析器作什么：

```
sun % cat /etc/resolv.conf
nameserver 140.252.1.54
domain tuc.noao.edu
```

第1行给出名字服务器——主机noao.edu的IP地址。最多可说明3个名字服务器行来提供足够的后备以防名字服务器故障或不可达。域名行说明默认域名。如果要查找的域名不是一个完全合格的域名（没有以句点结束），那末默认的域名.tuc.noao.edu将加到待查名后。

图14-10显示了名字解析器与名字服务器之间的分组交换。

```
1  0.0                  140.252.1.29.1447 > 140.252.1.54.53: 1+ A?
                        gemini.tuc.noao.edu. (37)

2  0.290820 (0.2908)    140.252.1.54.53 > 140.252.1.29.1447: 1* 2/0/0 A
                        140.252.1.11 (69)
```

图14-10 向名字服务器查询主机名gemini.tuc.noao.edu 的输出

让tcpdump程序不再显示每个IP数据报的源地址和目的地址。相反，它显示客户（resolver）的IP地址140.252.1.29和名字服务器的IP地址140.252.1.54。客户的临时端口号为1447，而名字服务器则使用熟知端口53。如果让tcpdump程序显示名字而不是IP地址，它可能会和同一个名字服务器联系（作指示查询），以致产生混乱的输出结果。

第1行中冒号后的字段（1+）表示标识字段为1，加号“+”表示RD标志（期望递归）为1。默认情况下，名字解析器要求递归查询方式。

下一个字段为A?，表示查询类型为A（我们需要一个IP地址），该问号指明它是一个查询（不是一个响应）。待查名字显示在后面：gemini.tuc.noao.edu.。名字解析器在待查名字后加上句点号指明它是一个绝对字段名。

在UDP数据报中的用户数据长度显示为37字节：12字节为固定长度的报文首部（图14-3）；21字节为查询名字（图14-6），以及用于查询类型和查询类的4个字节。在DNS报文中无需填充数据。

tcpdump程序的第2行显示的是从名字服务器发回的响应。1*是标识字段，星号表示设置AA标志（授权回答）（该服务器是noao.edu域的主域名服务器，其回答在该域内是可相信的。）

输出结果2/0/0表示在响应报文中最后3个变长字段的资源记录数：回答RR数为2，授权RR和附加信息RR数均为0。tcpdump仅显示第一个回答，回答类型为A（IP地址），值为140.252.1.11。

Linux公社 www.linuxidc.com

为什么我们的查询会得到两个回答？这是因为gemini是多接口主机，因此得到两个IP地址。事实上，另一个有用的DNS工具是一个称为host的公开程序，它能将查询传递给名字服务器，并显示返回的结果。如果使用这个程序，就能看到这个多地址主机的两个 IP地址：

```
sun % host gemini
gemini.tuc.noao.edu    A      140.252.1.11
gemini.tuc.noao.edu    A      140.252.3.54
```

图14-10中的第一个回答与host命令的第一行输出均是在同一子网（140.252.1）的IP地址。这不是偶然的。如果名字服务器和发出请求的主机位于相同的网络（或子网），那么BIND会排列显示的结果以便在相同网络的地址优先显示。

我们还可以使用其他的地址来访问gemini主机，但它可能不太有效。在这个例子中，使用traceroute显示出从子网140.252.1到140.252.3的正常路由不经过gemini主机，而是经过连接这两个网络的另一个路由器。因此在这种情况下，如果通过其他的IP地址（140.252.3.54）来访问gemini主机，所有分组均需经过额外的一跳。我们将在25.9节重新回到这个例子来探讨替换路由，那时可使用SNMP来查看一个路由器的路由表。

还有其他一些程序能很容易地对DNS进行交互访问。nslookup是大多数DNS实现中包含的程序。[Albitz and Liu 1992]的第10章详细介绍了该程序的使用方法。dig（“域名Internet搜索(Domain Internet Groper)”)程序是另一个查询DNS服务器的公开工具。doc（“域名模糊控制(Domain Obscenity Control)”)是一个使用dig的外壳脚本程序，它能向合适的名字服务器发送查询来诊断含义不清的域名，并对返回的查询结果进行简单的分析。附录F有如何获得这些程序的详细介绍。

在这个例子中要说明的最后一个问题是在查询结果中的 UDP数据长度：69字节。为说明这些字节需要知道以下两点：

1) 在返回的结果中包含查询问题。

2) 在返回的结果中会有许多重复的域名，因此使用压缩方式。在这个例子中，域名gemini.tuc.noao.edu出现了三次。

压缩方法很简单，当一个域名中的标识符是压缩的，它的单计数字节（范围由 0～63）中的最高两位将被设置为11。这表示它是一个16 bit指针而不再是8 bit的计数字节。指针中的剩下14 bit说明在该DNS报文中标识符所在的位置（起始位置由标识字段的第一字节起算）。我们明确说明只要一个标识符是压缩的，就可以使用这种指针，而不一定非要一个完整的域名压缩时才能使用。因为一个指针可能指向一个完整的域名，也可能只指向域名的结尾部分（这是因为给定域名的结尾标识符是相同的）。

图14-11显示了对应于图14-10的第2行的DNS应答的格式。我们也显示了 IP首部和UDP首部来重申DNS报文被封装在UDP数据报中。还明确显示了在问题部分的域名中各标识符的计数字节。返回的两个回答除了返回的 IP地址不同外，其余都是一样的。在这个例子中，每个回答中的指针值为12，表示从DNS首部开始的偏移量。

在这个例子中最后要注意的是使用telnet命令后输出的第2行，这里重复一下：

```
sun % telnet gemini daytime          我们只键入gemini
Trying 140.252.1.11 ...
Connected to gemini.tuc.noao.edu.  但Telnet客户输出FQDN
```

Linux公社 www.linuxidc.com

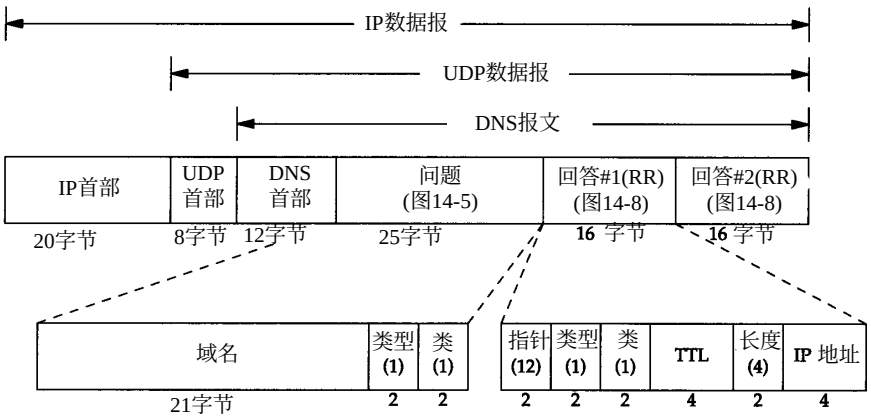


图14-11 对应于图14-10中第2行DNS应答的格式

我们仅仅输入了主机名(gemini)而不是FQDN，但Telnet客户程序部输出了FQDN。这是由于Telnet程序通过调用名字解析器（gethostbyname）对输入的名字进行查询，返回的结果包括IP地址和FQDN。Telnet程序就输出它试图与之建立TCP连接的IP地址，当连接建立后，它就输出FQDN。

如果在输入Telnet命令后间隔很长时间才显示IP地址，这个时延是由名字解析器和名字服务器在由域名到IP地址的解析所引起的。而显示Trying 到显示Connected to的时延则是由客户与服务器建立TCP连接所引起的，与DNS无关。

14.5 指针查询

DNS中一直难于理解的部分就是指针查询方式，即给定一个 IP地址，返回与该地址对应的域名。

首先回到图14-1，查看一下顶级域arpa，及它下面的in-addr域。当一个组织加入Internet，并获得DNS域名空间的授权，如noao.edu，则它们也获得了对应IP地址的in-addr.arpa域名空间的授权。在noao.edu这个例子中，它是网络号为140.252的B类网络。在DNS树中结点in-addr.arpa的下一级必须是该IP地址的第一字节（例中为140），再下一级为该IP地址的下一个字节（252），依此类推。但应牢记的是DNS名字是由DNS树的底部逐步向上书写的。这意味着对于IP地址为140.252.13.33的sun主机，它的DNS名字为33.13.252.140.in-addr.arpa。

必须写出4字节的IP地址，因为授权的代表是基于网络号：A类地址是第一字节，B类地址是第一、二字节，C类地址则是第一、二、三字节。IP地址的第一字节一定位于in-addr的下一级，但FQDN却是自树底往上书写的。如果FQDN由顶往下书写，则这个IP地址的DNS名字将是arpa.in-addr.140.252.13.33，而它所对应的域名将是edu.noao.tuc.sun。

如果DNS树中没有独立的分支来处理这种地址—名字的转换，将无法进行这种反向转换，除非从树根开始依次尝试每个顶级域。毫不夸张地说，这将需要数天或数周的时间。虽然反写IP地址和特殊的域名会造成某些混乱，但in-addr解决方案仍是一种最有效的方式。

只有在使用host程序或tcpdump程序直接同DNS打交道时，才会担心in-addr域和反写IP地址影响我们。从应用的角度上看，正常的名字解析器函数（gethostbyaddr）将接收一个IP地址并返回对应主机的有关信息。反转这些字节和添加in-addr.arpa域均由该函数自动完成。

Linux公社 www.linuxidc.com

14.5.1 举例

使用host程序完成一个指针查询，并使用tcpdump程序来观察这些分组。例子中的设置和图14-9相同，在sun主机上运行host程序，名字服务器在主机noao.edu上。我们指明svr4主机的IP地址：

```
sun % host 140.252.13.34
Name: svr4.tuc.noao.edu
Address: 140.252.13.34
```

既然IP地址是仅有的命令行参数，host程序将自动产生指针查询。图14-12显示了tcpdump的输出。

```
1  0.0                  140.252.1.29.1610 > 140.252.1.54.53: 1+ PTR?
                        34.13.252.140.in-addr.arpa. (44)

2  0.332288 (0.3323)    140.252.1.54.53 > 140.252.1.29.1610: 1* 1/0/0 PTR
                        svr4.tuc.noao.edu. (75)
```

图14-12 一个指针查询的tcpdump 输出

第1行显示标识符为1，期望递归标志设置为1（加号“+”），查询类型为PTR（应注意：问号“？”表示它是一个查询而不是响应）。44字节的数据包括12字节的DNS报文首部、28字节的域名标识符和4字节的查询类型和查询类。

查询结果包含一个回答RR，且为授权回答比特置1（带星号）。RR的类型是PTR，资源数据中包含该域名。

从名字解析器传递给名字服务器的指针查询不再是32 bit的IP地址，而是域名34.13.252.140.in-addr.arpa。

14.5.2 主机名检查

当一个IP数据报到达一个作为服务器的主机时，无论是UDP数据报还是TCP连接请求，服务器进程所能获得的是客户的IP地址和端口号（UDP或TCP）。某些服务器需要客户的IP地址来获得在DNS中的指针记录。在27.3节会看到这样的例子，从未知的IP地址使用匿名FTP访问服务器。

其他的一些服务器如Rlogin服务器（第26章）不但需要客户的IP地址来获得指针记录，还要向DNS询问该IP地址所对应的域名，并检查返回的地址中是否有地址与收到的数据报中的源IP地址匹配。该检查是因为.rhosts文件（见26.2节）中的条目仅包含主机名，而没有IP地址，因此主机需要证实该主机名是否对应源IP地址。

某些厂商将该项检查自动并入其名字解析器的例程中，特别是函数gethostbyaddr。这使得任何使用名字解析器的程序均可获得这种检查，而无需在应用中人为地进行这项检查。

来看一个使用SunOS 4.13名字解析器库的例子。我们编制了一个简单的程序通过调用函数gethostbyaddr来完成一个指针查询。我们已在文件/etc/resolv.conf中将名字服务器设置为noao.edu，sun主机通过SLIP链路与它相连。图14-13显示了当调用函数gethostbyaddr获取与IP地址140.252.1.29（sun主机）对应的名字时，tcpdump在SLIP链路上收到的内容。

第1行是预期的指针查询，第2行是预期的响应。但第3行显示了该名字解析器函数自动对第2行返回的名字发出一个IP地址查询。既然sun主机有两个IP地址，第4行的响应就包括两个

Linux公社 www.linuxidc.com

回答记录。如果这两个地址中没有与 gethostbyaddr输入参数匹配的地址，函数会向系统的日志发送一条报文，并向应用程序返回差错。

```
1  0.0                  sun.1812 > noao.edu.domain: 1+ PTR?
                        29.1.252.140.in-addr.arpa. (43)
2  0.339091 (0.3391)    noao.edu.domain > sun.1812: 1* 1/0/0 PTR
                        sun.tuc.noao.edu. (73)

3  0.344348 (0.0053)    sun.1813 > noao.edu.domain: 2+ A?
                        sun.tuc.noao.edu. (33)
4  0.669022 (0.3247)    noao.edu.domain > sun.1813: 2* 2/0/0 A
                        140.252.1.29 (69)
```

图14-13 调用名字解析器函数执行指针查询

14.6 资源记录

至今我们已经见到了一些不同类型的资源记录（RR）：IP地址查询为A类型，指针查询为类型PTR。也已看到了由名字服务器返回的资源记录：回答RR、授权RR和附加信息RR。现有大约20种不同类型的资源记录，下面将介绍其中的一些。另外，随着时间的推移，会加入更多类型的RR。

A　一个A记录定义了一个IP地址，它存储32 bit的二进制数。

PTR　指针记录用于指针查询。 IP地址被看作是 in-addr.arpa域下的一个域名（标识符串）。

CNAME　这表示“规范名字(canonical name)”。它用来表示一个域名（标识符串），而有规范名字的域名通常被称为别名 (alias)。某些FTP服务器使用它向其他的系统提供一个易于记忆的别名。

例如，gated服务器（10.3节提到）可通过匿名FTP从gated.cornell.edu获得，但这里并没有叫做gated的系统，这仅是为其他系统提供的别名。其他系统的规范名为gated.cornell.edu。

```
sun % host -t cname gated.cornell.edu
gated.cornell.edu  CNAM  COMET.CIT.CORNELL.EDU
```

这里使用的-t选项来指明它是特定的查询类型。

HINFO　表示主机信息：包括说明主机 CPU和操作系统的两个字符串。并非所有的站点均提供它们系统的HINFO记录，并且提供的信息也可能不是最新的。

```
sun %host -t hinfo sun
sun.tuc.noao.edu  HINFO Sun-4/25        Sun4.1.3
```

MX　邮件交换记录，用于以下一些场合：(1) 一个没有连到Internet的站点能将一个连到Internet的站点作为它的邮件交换器。这两个站点能够用一种交替的方式交换到达的邮件，而通常使用的协议是UUCP协议。(2) MX记录提供了一种将无法到达其目的主机的邮件传送到一个替代主机的方式。(3) MX记录允许机构提供供他人发送邮件的虚拟主机，如cs.university.edu，即使这样的主机名根本不存在。(4) 防火墙网关能使用MX记录来限制外界与内部系统的连接。许多不能与Internet连接的站点通过 UUCP链路与一个连接在 Internet上的站点如UUNET相连接。通过MX记录能使用user@host这种邮件地址向那个站点发送电子邮件。例如，一个假想的域 foo.com可能有下面的MX记录：

Linux公社 www.linuxidc.com

```
sun %host -t mx foo.com
foo.com                MX      relay1.UU.NET
foo.com                MX        relay2.UU.NET
```

MX记录能被连接在互联网主机中的邮件处理器使用。在这个例子中，其他的邮件处理器则被告知“如果有邮件要发往 user@foo.com，就将邮件送到relay1.uu.net或relay2.uu.net。”

每个MX记录被赋于一个16 bit的整数值，该值称为优先值。如果一个目的主机有多个MX记录，它们按优先值由小到大的顺序使用。

另一个MX记录的例子是处理主机脱机工作或不可达的情况。邮件处理器仅在无法使用TCP与目的主机连接时才使用MX记录。作者的主系统通过SLIP链路与互联网相连，它在大多数时间内是脱机工作的，我们有

```
sun % host -tv mx sun
Query about sun for record types MX
Trying sun within tuc.noao.edu ...
Query done, 2 answers, authoritative status: no error
sun.tuc.noao.edu     86400  IN    MX    0  sun.tuc.noao.edu
sun.tuc.noao.edu     86400  IN    MX    10 noao.edu
```

为了显示优先值，我们使用了-v选项（该选项也会导致其他字段的输出）。第二个字段，86400，是寿命值，单位为秒。因此该TTL值为24小时（24×60×60）。第3列，IN，是（Internet）类。我们看到直接传送给主机自身（第一个MX记录）有最低的优先值0。如果没有工作（即SLIP链路断开），会使用下一个更高优先值（10）的邮件记录，并试图向主机 noao.edu传送。如果它仍没有成功，发送将超时并在以后重新发送。

NS　名字服务器记录。它说明一个域的授权名字服务器。它由域名表示（符号串）。在下节将看到这些类型的例子。

这些是RR的常用类型。将在后面的例子中遇到它们。

14.7 高速缓存

为了减少Internet上DNS的通信量，所有的名字服务器均使用高速缓存。在标准的 Unix实现中，高速缓存是由名字服务器而不是由名字解析器维护的。既然名字解析器作为每个应用的一部分，而应用又不可能总处于工作状态，因此将高速缓存放在只要系统（名字服务器）处于工作状态就能起作用的程序中显得很重要。这样任何一个使用名字服务器的应用均可获得高速缓存。在该站点使用这个名字服务器的任何其他主机也能共享服务器的高速缓存。

在迄今为止（图14-9）所举例子的网络环境中，在 sun主机上运行客户程序，通过主机noao.edu的SLIP链路访问名字服务器。现在将改变这种设置，在 sun主机上运行名字服务器。在这种情况下，如果使用 tcpdump监视在SLIP链路上的DNS通信量，将只能看到服务器因超出其高速缓存而不能处理的查询。

在默认情况下，名字解析器将在本地主机上（ UDP端口号为53或TCP端口号为53）寻找名字服务器。从名字解析器文件中删除 nameserver行，而留下domain行：

```
sun % cat  /etc/resolv.conf
domain tuc.noao.edu
```

在这个文件中缺少namerserver指示将导致名字解析器使用本地主机上的名字服务器。

Linux公社 www.linuxidc.com

使用host命令执行下列查询：

```
sun %host ftp.uu.net
ftp.uu.net            A            192.48.96.9
```

图14-14显示了这个查询的输出结果。

```
1   0.0                 sun.tuc.noao.edu.domain > NS.NIC.DDN.MIL.domain:
                        2 A? ftp.uu.net. (28)
2   0.559285 ( 0.5593)  NS.NIC.DDN.MIL.domain > sun.tuc.noao.edu.domain:
                        2- 0/5/5 (229)

3   0.564449 ( 0.0052)  sun.tuc.noao.edu.domain > ns.UU.NET.domain:
                        3+ A? ftp.uu.net. (28)
4   1.009476 ( 0.4450)  ns.UU.NET.domain > sun.tuc.noao.edu.domain:
                        3* 1/0/0 A ftp.UU.NET (44)
```

图14-14 执行host ftp.uu.net后的tcpdump 输出

这次在tcpdump中使用了新的选项。使用-w选项来收集进出UDP或TCP 53号端口的所有数据。将这些原始数据记录在一个文件中供以后处理，同时防止tcpdump试图调用名字解析器来显示与那个IP地址相对应的域名。执行查询后，终止tcpdump并使用-r选项再次运行它。它会读取含有原始数据的文件并产生正式的输出显示（如图14-14）。这个过程要花费几秒钟，因为tcpdump调用了它自己的名字解析器。

在tcpdump输出中要注意的第一点是标识符(identifier)是小整数（2和3）。这是因为我们关闭这个名字服务器，后又重新启动它来强制清空它的高速缓存。当名字服务器启动时，它将标识符初始化为1。

当键入查询，查找主机ftp.uu.net的IP地址，该名字服务器就同8个根名字服务器中的一个ns.nic.ddn.mil（第1行）取得联系。这是以前见到的正常的 A类型查询，但要注意的是它的期望递归表示没有说明（如果该标志被设置，在标识符 2的后边会跟着一个加号）。在以前的例子中，经常看到名字解析器设置期望递归标志，但这里的名字服务器在与某个根服务器联系时没有设置这个标志。这是因为不应该向根名字服务器发出期望递归的查询，它们仅用来寻找其他授权名字服务器的地址。

第2行显示返回的响应中没有回答资源记录，而包含 5个授权资源记录和5个附加信息资源记录。标识符2后的减号表示期望递归标志（RA）没有被设置。即使我们要求进行递归查询，这个根名字服务器也不会回答期望递归查询。

尽管tcpdump没有显示返回的10个资源记录，我们也能执行host命令来查看高速缓存的内容：

```
sun % host -v ftp.uu.net
Query about ftp.uu.net for record types A
Trying ftp.uu.net ...
Query done, 1 answer, status: no error
The following answer is not authoritative:
ftp.uu.net              19109   IN      A       192.48.96.9
Authoritative nameservers:
UU.NET                  170308  IN      NS      NS.UU.NET
UU.NET                  170308  IN      NS      UUNET.UU.NET
UU.NET                  170308  IN      NS      UUCP-GW-1.PA.DEC.COM
UU.NET                  170308  IN      NS      UUCP-GW-2.PA.DEC.COM
UU.NET                  170308  IN      NS      NS.EU.NET
Additional information:
NS.UU.NET               170347  IN      A       137.39.1.3
UUNET.UU.NET            170347  IN      A       192.48.96.2
UUCP-GW-1.PA.DEC.COM    170347  IN      A       16.1.0.18
UUCP-GW-2.PA.DEC.COM    170347  IN      A       16.1.0.19
NS.EU.NET               170347  IN      A       192.16.202.11
```

Linux公社 www.linuxidc.com

这次采用-v选项查看的不仅仅只是A记录。它显示出对于域uu.net有5个授权名字服务器，而由根名字服务器返回的5个附加信息资源记录中含有这5个名字服务器的IP地址。这避免了在查找其中的某个名字服务器的地址时，无需再次与根名字服务器联系。这是DNS中的另一个实现优化。

host命令指出这个回答不是授权的，这是因为这个回答来自名字服务器的高速缓存，而不是来自授权名字服务器。

回到图14-14中的第3行，我们的名字服务器与第一个授权名字服务器（ns.uu.net）询问同一个问题：ftp.uu.net的IP地址？这次我们的服务器设置了期望递归标志。返回的应答（第4行）包含一个回答资源记录。

而后我们再次执行host命令，询问相同的名字：

```
sun % host ftp.uu.net
ftp.uu.net                  A           192.48.96.9
```

这时tcpdump没有输出，这正是我们所期望的，因为由host命令返回的回答来自于名字服务器的高速缓存。

再次执行host命令，查找ftp.ee.lbl.gov的地址：

```
sun % host ftp.ee.lbl.gov
ftp.ee.lbl.gov          CNAME       ee.lbl.gov
ee.lbl.gov              A           128.3.112.20
```

图14-15显示了这时的tcpdump输出。

```
1  18.664971 (17.6555)   sun.tuc.noao.edu.domain > c.nyser.net.domain:
                         4 A? ftp.ee.lbl.gov. (32)
2  19.429412 ( 0.7644)   c.nyser.net.domain > sun.tuc.noao.edu.domain:
                         4 0/4/4 (188)

3  19.432271 ( 0.0029)   sun.tuc.noao.edu.domain > ns1.lbl.gov.domain:
                         5+ A? ftp.ee.lbl.gov. (32)
4  19.909242 ( 0.4770)   ns1.lbl.gov.domain > sun.tuc.noao.edu.domain:
                         5* 2/0/0 CNAME ee.lbl.gov. (72)
```

图14-15 对ftp.ee.lbl.gov 主机的tcpdump 输出

这时第1行显示我们的服务器与另一个根名字服务器(c.nyser.net)联系。一个名字服务器通常轮询不同的根名字服务器来获得往返时间估计，然后选择往返时间最小的服务器。

既然我们的服务器向一个根服务器发出查询，那么期望递归标志不应被设置。正如我们在图14-14中所看到的该名字服务器并不清除期望递归标志（即便这样，一个名字服务器还是不应该向一个根名字服务器发出期望递归的查询）。

在第2行返回的响应中不包含回答资源记录，但含有4个授权记录和4个附加信息资源记录。正如我们所猜测的那样，4个授权资源记录是供主机ftp.ee.lbl.gov进行域名服务的名字服务器名，其他4个记录则是这4个服务器的IP地址。

第3行是向名字服务器ns1.lbl.gov（第2行中返回的4个名字服务器中的第一个）发出的查询请求。它的期望递归标志是被设置的。

第4行返回的响应和以往的响应不同。返回了两个回答资源记录，tcpdump指出其中的第一个是CNAME资源记录。ftp.ee.lbl.gov的规范名称是ee.lbl.gov。

Linux公社 www.linuxidc.com

这是CNAME记录常见的用法。LBL的FTP站点的名字通常是以ftp开始的，但它可能不时地从一个主机移到另一个主机。用户只需要知道ftp.ee.lbl.gov，必要时DNS会用它的规范名进行替换。

记得我们在运行host程序时，它显示了规范域名的 CNAME和IP地址。这是因为响应（图14-15中的第4行）中含有两个回答资源记录，第一个是 CNAME，而第二个是A记录。如果A记录没有随CNAME记录返回，我们的服务器将发出另一个查询请求，询问 ee.lbl.gov的IP地址。这是另一个DNS的实现优化 —— 在一个响应中同时返回一个规范域名的 CNAME记录和A记录。

14.8 用UDP还是用TCP

注意到DNS名字服务器使用的熟知端口号无论对 UDP还是TCP都是53。这意味着DNS均支持UDP和TCP访问，但我们使用tcpdump观察的所有例子都是采用 UDP。那么这两种协议都在什么情况下采用以及采用的理由都是什么呢？

当名字解析器发出一个查询请求，并且返回响应中的 TC（删减标志）比特被设置为 1时，它就意味着响应的长度超过了 512个字节，而仅返回前 512个字节。在遇到这种情况时，名字解析器通常使用TCP重发原来的查询请求，它将允许返回的响应超过 512个字节（回想在11.10节讨论的UDP数据报的最大长度）。既然TCP能将用户的数据流分为一些报文段，它就能用多个报文段来传送任意长度的用户数据。

此外，当一个域的辅助名字服务器在启动时，将从该域的主名字服务器执行区域传送。我们也说过辅助服务器将定时（通常是 3小时）向主服务器进行查询以便了解主服务器数据是否发生变动。如果有变动，将执行一次区域传送。区域传送将使用 TCP，因为这里传送的数据远比一个查询或响应多得多。

既然DNS主要使用UDP，无论是名字解析器还是名字服务器都必须自己处理超时和重传。此外，不像其他的使用 UDP的Internet应用（TFTP、BOOTP和SNMP），大部分操作集中在局域网上，DNS查询和响应通常经过广域网。分组丢失率和往返时间的不确定性在广域网上比局域网上更大。这样对于DNS客户程序，一个好的重传和超时程序就显得更重要了。

14.9 另一个例子

让我们通过另一个例子将已经介绍的许多 DNS特性作一个综合性回顾。先启动 Rlogin 客户程序，然后连接到一个位于其他域的 Rlogin服务器。图14-16显示了发生的分组交换过程。下面发生的11个步骤都假定客户和服务器的高速缓存中没有任何信息。

1) 客户程序启动后，调用它的名字解析器函数将我们键入的主机名转换为一个 IP地址。一个A类型的查询请求被送往一个根服务器。

2) 由根服务器返回的响应中包含为该服务器所在域服务的名字服务器名。

3) 客户端的名字解析器将向该服务器的名字服务器重发上述 A类型查询，这个查询通常是将期望递归标志设置为1。

4) 返回的应答中包含Rlogin服务器的IP地址。

5) Rlogin客户和Rlogin服务器建立一个TCP连接（第18章将提供该步骤的细节）。客户和服务器的TCP模块间将交换3个分组。

Linux公社 www.linuxidc.com

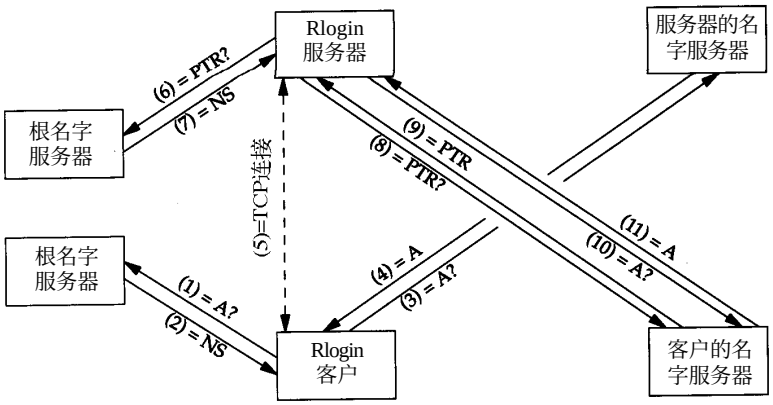


图14-16 启动Rlogin客户和服务器的分组交换过程

6) Rlogin服务器收到来自客户的连接请求后，调用它的名字解析器通过 TCP连接请求中的IP地址获得客户主机名。这是一个 PTR查询请求，由一个根名字服务器处理。这个根名字服务器可以不同于步骤1中客户使用的根名字服务器。

7) 这个根名字服务器的响应中含有为客户的 `in-addr.arpa`域的名字服务器。

8) 服务器上的名字解析器将向客户的名字服务器重传上述 PTR查询。

9) 返回的PTR应答中含有客户主机的FQDN。

10) 服务器的名字解析器向客户的名字服务器发送一个 A类型查询请求，查找前一步返回的名字对应的IP地址。这可能由服务器中的`gethostbyaddr`函数自动完成，正如我们在14.5节中介绍的那样，否则 Rlogin服务器将完成这一步。此外，客户的名字服务器常常就是客户的`in-addr.arpa`名字服务器，但这不是必需的。

11) 从客户的名字服务器返回的响应含有客户主机的 A记录。Rlogin服务器将客户的TCP连接请求中的IP地址与A记录作比较。

高速缓存将减少这个图中交换的分组数目。

14.10 小结

DNS是任何与Internet相连主机必不可少的一部分，同时它也广泛用于专用的互联网。层次树是组成DNS域名空间的基本组织形式。

应用程序通过名字解析器将一个主机名转换为一个 IP地址，也可将一个IP地址转换为与之对应的主机名。名字解析器将向一个本地名字服务器发出查询请求，这个名字服务器可能通过某个根名字服务器或其他名字服务器来完成这个查询。

所有的DNS查询和响应都有相同的报文格式。这个报文格式中包含查询请求和可能的回答资源记录、授权资源记录和附加资源记录。通过许多例子了解了名字解析器的配置文件以及DNS的优化措施：指向域名的指针（减少报文的长度）、查询结果的高速缓存、`in-addr.arpa`域（查找IP地址对应的域名）以及返回的附加资源记录（避免主机重发同一查询请求）。

习题

14.1 讨论一个DNS 名字解析器和一个DNS名字服务器作为客户程序、服务器或同时作为客

Linux公社 www.linuxidc.com

户和服务器的情况。

14.2 说明图14-12中构成响应的75个字节的含义。

14.3 在12.3节我们指出，一个既可接受点分十进制形式的 IP地址、也可接收主机名的应用程序，应先假定输入的是IP地址，如果失败，再假定是主机名。如果改变这个测试顺序会出现什么情况？

14.4 每个UDP数据报有一个相应的长度。一个接收 UDP数据报的进程将被告知这个长度。当名字解析器使用TCP而不是UDP来处理查询请求时，由于 TCP是没有任何记录标记的字节流，那么应用程序是如何知道有多少数据返回？注意在 DNS的报文首部（图14-3）中没有任何长度字段（提示：查阅 RFC 1035）

14.5 我们说一个名字服务器必须知道根名字服务器的 IP地址，这一信息可通过匿名 FTP获得。不幸的是当根名字服务器表发生变化时，并不是所有的系统管理员都会更新他们的 DNS配置文件（根名字服务表的确会发生变化，尽管不是经常的）你认为 DNS如何处理这个问题？

14.6 利用习题1.8指明的文件来确定谁应负责维护根名字服务器。名字服务器更新的频度是怎样的？

14.7 维护一个名字服务器和一个无状态的名字解析器高速缓存的问题分别是什么？

14.8 在图14-10的讨论中，我们指出名字服务器将对 A类型记录进行排序以便在公共网中的地址先出现。谁对A类型记录进行这种排序，是名字服务器还是名字解析器？

Linux公社 www.linuxidc.com

第15章　TFTP：简单文件传送协议

15.1　引言

TFTP(Trivial File Transfer Protocol)即简单文件传送协议，最初打算用于引导无盘系统（通常是工作站或X终端）。和将在第27章介绍的使用TCP的文件传送协议（FTP）不同，为了保持简单和短小，TFTP将使用UDP。TFTP的代码（和它所需要的UDP、IP和设备驱动程序）都能适合只读存储器。

本章对TFTP只作一般介绍，因为在下一章引导程序协议（Bootstrap Protocol）中还会遇到TFTP。在图5-1中，当从网络上引导sun主机时，也曾遇到过TFTP，sun主机通过RARP获得它的IP地址后，将发出一个TFTP请求。

RFC 1350 [Sollins 1992]是第2版TFTP的正式规范。第12章 [Stevens 1990] 提供了实现TFTP客户和服务器的全部源代码，并介绍了一些使用TFTP的编程技术。

15.2　协议

在开始工作时，TFTP的客户与服务器交换信息，客户发送一个读请求或写请求给服务器。在一个无盘系统进行系统引导的正常情况下，第一个请求是读请求（RRQ）。图15-1显示了5种TFTP报文格式（操作码为1和2的报文使用相同的格式）。

TFTP报文的头两个字节表示操作码。对于读请求和写请求（WRQ），文件名字段说明客户要读或写的位于服务器上的文件。这个文件字段以0字节作为结束（见图15-1）。模式字段是一个ASCII码串netascii或octet（可大小写任意组合），同样以0字节结束。netascii表示数据是以成行的ASCII码字符组成，以两个字节——回车字符后跟换行字符（称为CR/LF）作为行结束符。这两个行结束字符在这种格式和本地主机使用的行定界符之间进行转化。octet则将数据看作8 bit一组的字节流而不作任何解释。

每个数据分组包含一个块编号字段，它以后要在确认分组中使用。以读一个文件作为例子，TFTP客户需要发送一个读请求说明要读的文件名和文件模式 (mode)。如果这个文件能被这个客户读取，TFTP服务器就返回一个块编号为1的数据分组。TFTP客户又发送一个块编号为1的ACK。TFTP服务器随后发送块编号为2的数据。TFTP客户发回块编号为2的ACK。重复这个过程直到这个文件传送完。除了最后一个数据分组可含有不足512字节的数据，其他每个数据分组均含有512字节的数据。当TFTP客户收到一个不足512字节的数据分组，就知道它收到最后一个数据分组。

在写请求的情况下，TFTP 客户发送WRQ指明文件名和模式。如果该文件能被 该客户写，TFTP 服务器就返回块编号为0的ACK包。该客户就将文件的头512字节以块编号为1发出。服务器则返回块编号为1的ACK。

这种类型的数据传输称为停止等待协议。它只用在一些简单的协议如 TFTP中。在20.3节中将看到TCP提供了不同形式的确认，能提供更高的系统吞吐量。 TFTP的优点在于实现的简

Linux公社 www.linuxidc.com

单而不是高的系统吞吐量。

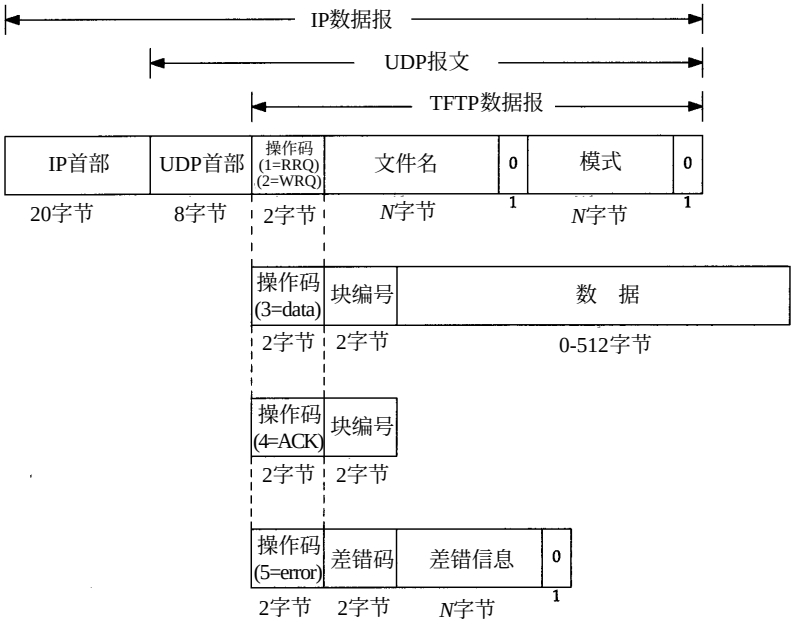


图15-1 5种TFTP报文格式

最后一种TFTP报文类型是差错报文，它的操作码为 5。它用于服务器不能处理读请求或写请求的情况。在文件传输过程中的读和写差错也会导致传送这种报文，接着停止传输。差错编号字段给出一个数字的差错码，跟着是一个 ASCII表示的差错报文字段，可能包含额外的操作系统说明的信息。

既然TFTP使用不可靠的UDP，TFTP就必须处理分组丢失和分组重复。分组丢失可通过发送方的超时与重传机制解决（注意存在一种称为“魔术新手综合症 (sorcerer's apprentice syndrome)”的潜在问题，如果双方都超时与重传，就可能出现这个问题。 12.2 节 [Stevens 1990] 介绍了这个问题是如何发生的)。和许多UDP应用程序一样， TFTP报文中没有检验和，它假定任何数据差错都将被UDP的检验和检测到（参见11.3节）。

15.3 一个例子

让我们通过观察协议的工作情况来了解 TFTP。在bsdi主机上运行TFTP 客户程序，并从主机svr4读取一个文本文件：

```
bsdi % tftp svr4                          启动TFTP客户进程
tftp> get test1.c                         从服务器读取文件
Received 962 bytes in 0.3 seconds
tftp> quit                                结束

bsdi % ls -l test1.c                      查看我们读取的文件大小
-rw-r--r--  1 rstevens  staff  914 Mar 20 11:41 test1.c

bsdi % wc -l test1.c                      文件行数？
      48 test1.c
```

最先引起我们注意的是在Unix系统下接收的文件长度是914字节，而TFTP则传送了962个字节。使用wc程序我们看到文件共有 48行，因此48个Unix的换行符被转化成 48个CR/CF对，

Linux公社 www.linuxidc.com

因为默认情况下TFTP使用netascii模式传送。

图15-2显示了发生的分组交换过程。

```
1  0.0                    bsdi.1106 > svr4.tftp: 19 RRQ "test1.c"

2  0.287080 (0.2871)      svr4.1077 > bsdi.1106: udp 516
3  0.291178 (0.0041)      bsdi.1106 > svr4.1077: udp 4

4  0.299446 (0.0083)      svr4.1077 > bsdi.1106: udp 454
5  0.312320 (0.0129)      bsdi.1106 > svr4.1077: udp 4
```

图15-2 使用TFTP传输一个文件的分组交换过程

第1行显示了客户向服务器发送的读请求。由于目的 UDP端口是TFTP熟知端口（69），tcpdump将解释TFTP分组，并显示RRQ和文件名。19字节的UDP数据包括2字节的操作码，7字节的的文件名，1字节的0，8字节的netascii模式以及另1字节的0结束。

下一个分组由服务器发回（第2行），共包含516字节：2字节的操作码，2字节的数据块号和512字节的数据。第3行是这个数据块的确认，它包括2字节的操作码和2字节的数据块号。

最后的数据分组（第4行）包含450字节的数据。这450字节的数据加上第2行的512字节的数据就是向该客户传送的962字节的数据。注意tcpdump仅在第1行解释TFTP报文，而在2～5行都不显示任何 TFTP协议信息。这是因为服务器进程的端口在第 1行和第2行发生了变化。TFTP协议需要客户进程向服务器进程的 UDP熟知端口（69）发送第一个分组（RRQ或WRQ）。之后服务器进程便向服务器主机申请一个尚未使用的端口（1077，见图15-2），服务器进程使用这个端口来进行请求客户进程与服务器进程间的其他数据交换。客户进程的端口号（在这个例子中为1106）没有变化。tcpdump无法知道主机srv4上的1077端口是一个TFTP服务器进程。

服务器进程端口变化的原因是服务器进程不能占用这个熟知端口来完成需一些时间的文件传输（可能是几十秒甚至数分钟）。相反，在传输当前文件的过程中，这个熟知端口要留出来供其他的TFTP客户进程发送它们的请求。

回顾图10-6，当RIP服务器向客户发送的数据超过 512字节，两个UDP数据报都使用服务器的熟知端口。在那个例子中，即使服务器进程必须写多个数据报以便将所有数据发回，服务器进程也是先写一个，再写一个，它们都使用它的熟知端口。然而， TFTP协议与它不同，因为客户与服务器间的连接需要持续一个较长的时间（可能是数秒或数分钟）。如果一个服务器进程使用熟知端口来进行文件传输，那么在文件传输期间，它要么拒绝任何来自其他客户的请求，要么一个服务器进程在同一端口（69）同时对多个客户进程进行多个文件传输。最简单的办法是让服务器进程在收到 RRQ或WRQ后，改用新的端口。当然，客户进程在收到第一个数据分组（图 15-2的第2行）后必须探测到这个新的端口，并将之后的所有确认（第 3行和第5行）发送到那个新的端口。

在16.3节我们将看到当X终端在进行系统引导时将使用 TFTP。

15.4 安全性

注意在TFTP分组（图15-1）中并不提供用户名和口令。这是 TFTP的一个特征（即“安全漏洞”）。由于TFTP是设计用于系统引导进程，它不可能提供用户名和口令。

TFTP的这一特性被许多解密高手用于获取 Unix口令文件的复制，然后来猜测用户口令。

Linux公社 www.linuxidc.com

为防止这种类型的访问，目前大多数 TFTP服务器提供了一个选项来限制只能访问特定目录下的文件（Unix系统中通常是`/tftpboot`）。这个目录中只包含无盘系统进行系统引导时所需的文件。

对其他的安全性，Unix系统下的TFTP服务器通常将它的用户ID和组ID设置为不会赋给任何真正用户的值。这只允许访问具有读或写属性的文件。

15.5 小结

TFTP是一个简单的协议，适合于只读存储器，仅用于无盘系统进行系统引导。它只使用几种报文格式，是一种停止等待协议。

为了允许多个客户端同时进行系统引导，TFTP服务器必须提供一定形式的并发。因为UDP在一个客户与一个服务器之间并不提供唯一连接（TCP也一样），TFTP服务器通过为每个客户提供一个新的UDP端口来提供并发。这允许不同的客户输入数据报，然后由服务器中的UDP模块根据目的端口号进行区分，而不是由服务器本身来进行区分。

TFTP协议没有提供安全特性。大多数执行指望TFTP服务器的系统管理员来限制客户的访问，只允许它们访问引导所必须的文件。

第27章介绍的文件传输协议（FTP）是设计用于一般目的的、高吞吐量的文件传输。

习题

15.1 阅读Host Requirements RFC，了解如果一个TFTP 服务器收到的请求的目的IP地址是一个广播地址，它将做什么。

15.2 当TFTP块号由65535跳回到0时，你认为会发生什么？RFC 1350提到了如何处理这一问题吗？

15.3 TFTP发送方采用超时重发来处理分组丢失。当 TFTP作为引导进程的一部分时，这种方法对TFTP的使用有何影响？

15.4 使用TFTP时，影响传输文件所需时间的限制性因素是什么？

Linux公社 www.linuxidc.com

第16章　BOOTP：引导程序协议

16.1　引言

在第5章我们介绍了一个无盘系统，它在不知道自身 IP地址的情况下，在进行系统引导时能够通过RARP来获取它的IP地址。然而使用RARP有两个问题：（1）IP地址是返回的唯一结果；（2）既然RARP使用链路层广播，RARP请求就不会被路由器转发（迫使每个实际网络设置一个RARP 服务器）。本章将介绍一种用于无盘系统进行系统引导的替代方法，又称为引导程序协议，或BOOTP。

BOOTP使用UDP，且通常需与 TFTP（参见第 15章）协同工作。RFC 951 [Croft and Gilmore 1985]是BOOTP的正式规范，RFC 1542 [Wimer 1993]则对它作了说明。

16.2　BOOTP 的分组格式

BOOTP 请求和应答均被封装在UDP数据报中，如图16-1所示。

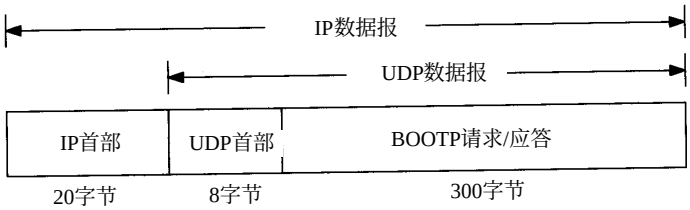


图16-1　BOOTP 请求和应答封装在一个UDP数据报内

图16-2显示了长度为300字节的BOOTP请求和应答的格式。

“操作码”字段为1表示请求，为2表示应答。硬件类型字段为 1表示10 Mb/s的以太网，这和ARP请求或应答（图 4-3）中同名字段表示的含义相同。类似地，对于以太网，硬件地址长度字段为6字节。

“跳数”字段由客户设置为0，但也能被一个代理服务器设置（参见 16.5节）。

“事务标识”字段是一个由客户设置并由服务器返回的 32 bit整数。客户用它对请求和应答进行匹配。对每个请求，客户应该将该字段设置为一个随机数。

客户开始进行引导时，将“秒数”字段设置为一个时间值。服务器能够看到这个时间值，备用服务器在等待时间超过这个时间值后才会响应客户的请求，这意味着主服务器没有启动。

如果该客户已经知道自身的 IP地址，它将写入“客户 IP地址”字段。否则，它将该字段设置为0。对于后面这种情况，服务器用该客户的 IP地址写入“你的IP地址”字段。“服务器IP地址”字段则由服务器填写。如果使用了某个代理服务器（见 16.5节），则该代理服务器就填写“网关IP地址”字段。

客户必须设置它的“客户硬件地址”字段。尽管这个值与以太网数据帧头中的值相同，UDP数据报中也设置这个字段，但任何接收这个数据报的用户进程能很容易地获得它（例如

Linux公社 www.linuxidc.com

一个BOOTP 服务器）。一个进程通过查看UDP数据报来确定以太网帧首部中的该字段通常是很困难的（或者说是不可能的）。

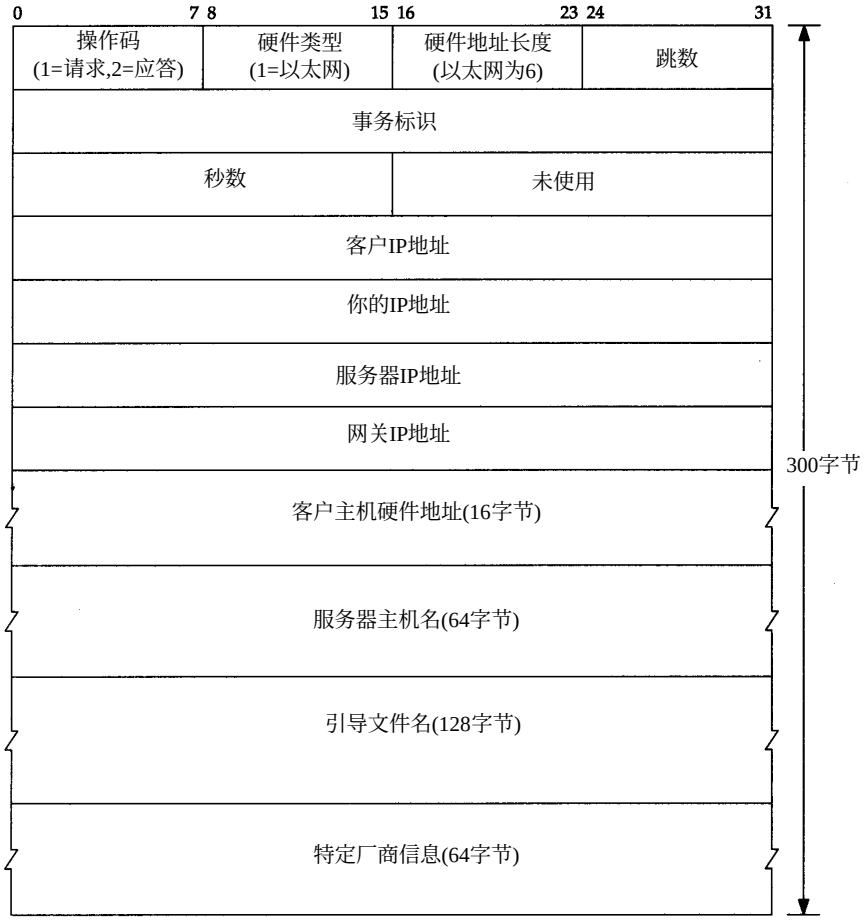


图16-2 BOOTP请求和应答的格式

“服务器主机名”字段是一个空值终止串，由服务器填写。服务器还将在“引导文件名字段”填入包括用于系统引导的文件名及其所在位置的路径全名。

“特定厂商区域”字段用于对BOOTP进行不同的扩展。16.6节将介绍这些扩展中的一些。

当一个客户使用BOOTP（操作码为1）进行系统引导时，引导请求通常是采用链路层广播，IP首部中的目的IP地址为255.255.255.255（受限的广播，12.2节）。源IP地址通常是0.0.0.0，因为此时客户还不知道它本身的IP地址。回顾图3-9，在系统进行自引导时，0.0.0.0是一个有效的IP地址。

端口号

BOOTP有两个熟知端口：BOOTP 服务器为67，BOOTP 客户为68。这意味着BOOTP 客户不会选择未用的临时端口，而只用端口68。选择两个端口而不是仅选择一个端口为BOOTP服务器用的原因是：服务器的应答可以进行广播（但通常是不用广播的）。

如果服务器的应答是通过广播传送的，同时客户又选择未用的临时端口，那么这些广播

Linux公社 www.linuxidc.com

也能被其他的主机中碰巧使用相同临时端口的应用进程接收到。因此，采用随机端口（即临时端口）对广播来说是一个不好的选择。

如果客户也使用服务器的知名端口（67）作为它的端口，那么网络内的所有服务器会被唤醒来查看每个广播应答（如果所有的服务器都被唤醒，它们将检查操作码，如果是一个应答而不是请求，就不作处理）。因此可以让所有的客户使用与服务器知名端口不同的同一知名端口。

如果多个客户同时进行系统引导，并且服务器广播所有应答，这样每个客户都会收到其他客户的应答。客户可以通过BOOTP首部中的事务标识字段来确认应答是否与请求匹配，或者可以通过检查返回的客户硬件地址加以区分。

16.3 一个例子

让我们看一个用BOOTP引导一个X终端的例子。图16-3显示了tcpdump的输出结果（例中客户名为proteus，服务器名为mercury。这个tcpdump的输出是在不同的网络上获得的，这个应用程序是其他例子中一直使用的）。

```
 1   0.0                  0.0.0.0.68 > 255.255.255.255.bootp:
                          secs:100 ether 0:0:a7:0:62:7c
 2   0.355446 (0.3554)    mercury.bootp > proteus.68: secs:100 Y:proteus
                          S:mercury G:mercury ether 0:0:a7:0:62:7c
                          file "/local/var/bootfiles/Xncd19r"
 3   0.355447 (0.0000)    arp who-has proteus tell 0.0.0.0
 4   0.851508 (0.4961)    arp who-has proteus tell 0.0.0.0
 5   1.371070 (0.5196)    arp who-has proteus tell proteus
 6   1.863226 (0.4922)    proteus.68 > 255.255.255.255.bootp:
                          secs:100 ether 0:0:a7:0:62:7c
 7   1.871038 (0.0078)    mercury.bootp > proteus.68: secs:100 Y:proteus
                          S:mercury G:mercury ether 0:0:a7:0:62:7c
                          file "/local/var/bootfiles/Xncd19r"
 8   3.871038 (2.0000)    proteus.68 > 255.255.255.255.bootp:
                          secs:100 ether 0:0:a7:0:62:7c
 9   3.878850 (0.0078)    mercury.bootp > proteus.68: secs:100 Y:proteus
                          S:mercury G:mercury ether 0:0:a7:0:62:7c
                          file "/local/var/bootfiles/Xncd19r"
10   5.925786 (2.0469)    arp who-has mercury tell proteus
11   5.929692 (0.0039)    arp reply mercury is-at 8:0:2b:28:eb:1d
12   5.929694 (0.0000)    proteus.tftp > mercury.tftp: 37 RRQ
                          "/local/var/bootfiles/Xncd19r"
13   5.996094 (0.0664)    mercury.2352 > proteus.tftp: 516 DATA block 1
14   6.000000 (0.0039)    proteus.tftp > mercury.2352: 4 ACK
                                   这里删除了许多行
15  14.980472 (8.9805)    mercury.2352 > proteus.tftp: 516 DATA block 2463
16  14.984376 (0.0039)    proteus.tftp > mercury.2352: 4 ACK
17  14.984377 (0.0000)    mercury.2352 > proteus.tftp: 228 DATA block 2464
18  14.984378 (0.0000)    proteus.tftp > mercury.2352: 4 ACK
```

图16-3 用BOOTP引导一个X终端的例子

在第1行中，我们看到客户请求来自0.0.0.0.68，发送目的站是255.255.255.255.67。该客户已经填写的字段是秒数和自身的以太网地址。我们看到客户通常将秒数设置为100。tcpdump没有显示跳数和事务标识，因为它们均为0(事务标识为0表示该客户忽略这个字段，

Linux公社 www.linuxidc.com

因为如果打算对返回响应进行验证，它将把这个字段设置为一个随机数值)。

第2行是服务器返回的应答。由服务器填写的字段是该客户的 IP地址（tcpdump显示为名字proteus）、服务器的 IP地址（显示为名字 mercury）、网关的 IP地址（显示为名字 mercury）和引导文件名。

在收到BOOTP应答后，该客户立即发送一个ARP请求来了解网络中其他主机是否有IP地址。跟在who-has后的名字proteus对应目的IP地址（图4-3），发送者的IP地址被设置为0.0.0.0。它在0.5秒后再发一个相同的ARP请求，之后再过0.5秒又发一个。在第3个ARP请求（第5行）中，它将发送者的IP地址改变为它自己的IP地址。这是一个没有意义的ARP请求（见4.7节）。

第6行显示该客户在等待另一个0.5秒后，广播另一个BOOTP请求。这个请求与第1行的唯一不同是此时客户将它的IP地址写入IP首部中。它收到来自同一个服务器的相同应答（第7行）。该客户在等待2秒后，又广播一个 BOOTP请求（第8行），同样收到来自同一服务器的相同应答。

该客户等待2秒后，向它的服务器mercury发送一个ARP请求（第10行）。收到这个ARP应答后，它立即发送一个 TFTP读请求，请求读取它的引导文件（第 12行）。文件传送过程包括2464个TFTP数据分组和确认，传送的数据量为 512×2463 + 224 = 1 261 280字节。这将操作系统调入X终端。我们已在图16-3中删除了大多数TFTP行。

当和图15-2比较TFTP的数据交换过程时，要注意的是这儿的客户在整个传输过程中使用TFTP的知名端口（69）。既然通信双方中的一方使用了端口 69，tcpdump就知道这些分组是TFTP报文，因此它能用TFTP协议来解释每个分组。这就是为什么图 16-3能指明哪些包含有数据，哪些包含有确认，以及每个分组的块编号。在图 15-2中我们并不能获得这些额外的信息，因为通信双方均没有使用 TFTP的知名端口进行数据传送。由于 TFTP 服务器作为一个多用户系统，且使用TFTP的知名端口，因此通常TFTP客户不能使用那个端口。但这里的系统处于正被引导的过程中，无法提供一个 TFTP服务器，因此允许该客户在传输期间使用 TFTP的知名端口。这也暗示在mercury上的TFTP服务器并不关心客户的端口号是什么 —— 它只将数据传送到客户的端口上，而不管发生了什么。

从图16-3可以看出在9秒内共传送了1 261 280字节。数据速率大约为140 000 bps。这比大多数以FTP文件传送形式访问一个以太网要慢，但对于一个简单的停止等待协议如 TFTP来说已经很好了。

X终端系统引导后，还需使用 TFTP传送终端的字体文件、某些 DNS名字服务器查询，然后进行X协议的初始化。图16-3中的所有步骤大概需要 15秒钟，其余的步骤需要 6秒钟，这样无盘X终端系统引导的总时间是21秒。

16.4 BOOTP服务器的设计

BOOTP客户通常固化在无盘系统只读存储器中，因此了解 BOOTP 服务器的实现将更有意义。

首先，BOOTP 服务器将从它的熟知端口（67）读取UDP数据报。这没有特别的地方。它不同于RARP 服务器 （5.4节），它必须读取类型字段为“ RARP请求 ”的以太网帧。BOOTP协议通过将客户的硬件地址放入 BOOTP分组中，使得服务器很容易获取客户的硬件地址 (图16-2)。

Linux公社 www.linuxidc.com

这里出现了一个有趣的问题：TFTP 服务器如何能将一个响应直接送回 BOOTP 客户？这个响应是一个UDP数据报，而服务器知道该客户的 IP地址（可能通过读取服务器上的配置文件）。但如果这个客户向那个 IP地址发送一个UDP数据报（正常情况下会处理 UDP的输出），BOOTP 服务器的主机就可能向那个IP地址发送一个ARP请求。但这个客户不能响应这个 ARP请求，因为它还不知道它自己的IP地址！（这就是在RFC951中被称作“鸡和蛋”的问题。）

有两种解决办法：第一种，通常被Unix 服务器采用，是服务器发一个ioctl(2)请求给内核，为该客户在ARP高速缓存中设置一个条目（这就是命令 `arp-s`所做的工作，见4.8节）。服务器能一直这么做直到它知道客户的硬件地址和 IP地址。这意味着当服务器发送 UDP数据报（即BOOTP应答）时，服务器的ARP将在ARP 高速缓存中找到该客户的IP地址。

另一种可选的解决办法是服务器广播这个 BOOTP应答而不直接将应答发回该客户。既然通常期望网络广播越少越好，因此这种解决方案应该只在服务器无法在它的 ARP 高速缓存设置一个条目的情况下使用。通常只有拥有超级用户权限才能在 ARP高速缓存设置一个条目，如果没有这种权限就只能广播BOOTP应答。

16.5 BOOTP穿越路由器

我们在5.4节中提到RARP的一个缺点就是它使用链路层广播，这种广播通常不会由路由器转发。这就需要在每个物理网络内设置一个 RARP 服务器。如果路由器支持 BOOTP协议，那么BOOTP能够由路由器转发（绝大多数路由器厂商的产品都支持这个功能）。

这个功能主要用于无盘路由器，因为如果在磁盘的多用户系统被用作路由器，它就能够自己运行BOOTP 服务器。此外，常用的 Unix BOOTP服务器（附录 F）支持这种中继模式（relay mode）。但如果在这个物理网络内运行一个BOOTP 服务器，通常没有必要将BOOTP请求转发到在另外网络中的另一个服务器。

研究一下当路由器（也称作“ BOOTP 中继代理”）在服务器的熟知端口（ 67）接收到BOOTP请求时将会发生什么。当收到一个 BOOTP请求时，中继代理将它的 IP地址填入收到BOOTP请求中的“网关IP地址字段”，然后将该请求发送到真正的 BOOTP服务器（由中继代理填入网关字段的地址是收到的 BOOTP请求接口的IP地址）。该代理中继还将跳数字段值加 1（这是为防止请求被无限地在网络内转发。 RFC 951认为如果跳数值到达3就可以丢弃该请求）。既然发出的请求是一个单播的数据报（与发起的客户的请求是广播的相反），它能按照一定的路由通过其他的路由器到达真正的 BOOTP服务器。真正的BOOTP服务器收到这个请求后，产生BOOTP应答，并将它发回中继代理，而不是请求的客户。既然请求网关字段不为零，真正的BOOTP服务器知道这个请求是经过转发的。中继代理收到应答后将它发给请求的客户。

16.6 特定厂商信息

在图16-2中我们看到了 64字节的“特定厂商区域”。RFC 1533 [Alexander and Droms 1993] 定义了这个区域的格式。这个区域含有服务器返回客户的可选信息。

如果有信息要提供，这个区域的前 4个字节被设置为IP地址99.130.83.99。这可称作魔术甜饼(magic cookie)，表示该区域内包含信息。

这个区域的其余部分是一个条目表。每个条目的开始是 1字节标志字段。其中的两个条目仅有标志字段：标志为 0的条目作为填充字节（为使后面的条目有更好的字节边界），标志为

Linux公社 www.linuxidc.com

255的条目表示结尾条目。第一个结尾条目后剩余的字节都应设置为这个数值（ 255）。

除了这两个1字节的条目，其他的条目还包含一个单字节的长度字段，后面是相应的信息。图16-4显示了厂商说明区域中一些条目的格式。

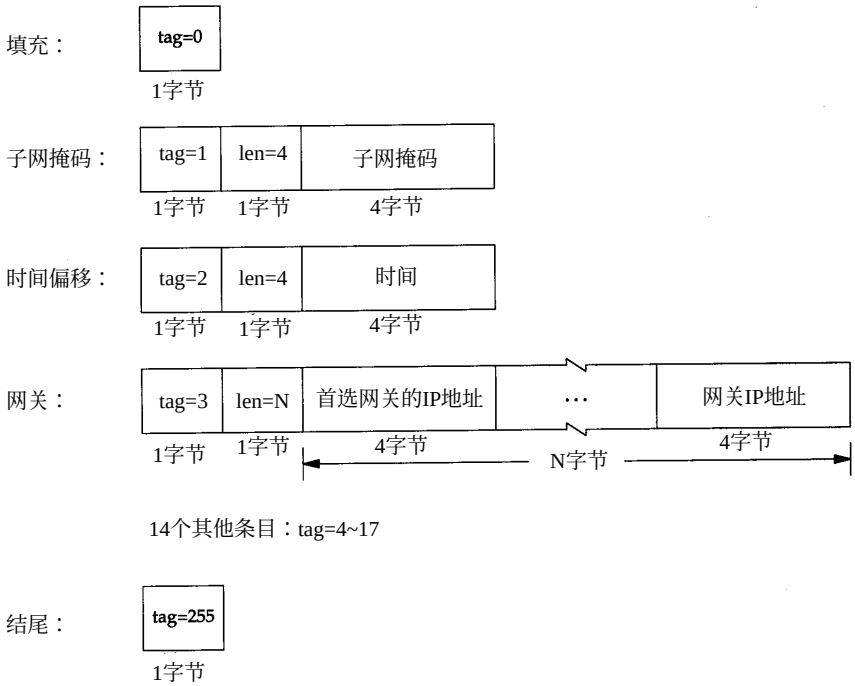


图16-4 厂商说明区域中一些条目的格式

子网掩码条目和时间值条目都是定长条目，因为它们的值总是占 4个字节。时间偏移值是从1900年1月1日0时以来的秒数（UTC）。

网关条目是变长条目。长度通常是4 的倍数，这个值是一个或多个供客户使用的网关（路由器）的IP地址。返回的第一个必须是首选的网关。

RFC 1533还定义了其他14个条目。其中最重要的可能是 DNS名字服务器的IP地址条目，条目的志为6。其他的条目包括打印服务器、时间服务器等的 IP地址。详细情况可参考 RFC文档。

回到在图16-3中的例子，我们从未看到客户广播一个 ICMP地址掩码请求（6.3节）来获取它的子网掩码。尽管tcpdump不能显示出来，但我们可认为客户所在网络的子网掩码在返回的BOOTP应答的厂商说明区域内。

Host Requirements RFC文档推荐一个系统使用BOOTP来获悉它的子网掩码，而不是采用ICMP。

厂商说明区域的大小被限制为 64字节。这对某些应用是个约束。一个新的称为动态主机配置协议DHCP（Dynamic Host Configuration Protocol）已经出现，但它不是替代BOOTP的。DHCP将这个区域的长度扩展到312字节，它在RFC 1541 [Droms 1993] 中定义。

16.7 小结

BOOTP使用UDP，它为引导无盘系统获得它的 IP地址提供了除RARP外的另外一种选择。

Linux公社 www.linuxidc.com

BOOTP还能返回其他的信息，如路由器的IP地址、客户的子网掩码和名字服务器的IP地址。

既然BOOTP用于系统引导过程，一个无盘系统需要下列协议才能在只读存储器中完成：BOOTP、TFTP、UDP、IP和一个局域网的驱动程序。

BOOTP服务器比RARP服务器更易于实现，因为BOOTP请求和应答是在UDP数据报中，而不是特殊的数据链路层帧。一个路由器还能作为真正BOOTP服务器的代理，向位于不同网络的真正BOOTP服务器转发客户的BOOTP请求。

习题

16.1 我们说BOOTP优于RARP的一个方面是BOOTP能穿越路由器，而RARP由于使用链路层广播则不能。在16.5节为使BOOTP穿越路由器，我们必须定义特殊的方式。如果在路由器中增加允许转发RARP请求的功能会发生什么？

16.2 我们说过，当有多个客户程序同时向一个服务器发出引导请求时，因为服务器要广播多个BOOTP应答，BOOTP客户就必须使用事务标识来使响应与请求相匹配。但在图16-3中，事务标识为0，表示这个客户不考虑事务标识。你认为这个客户将如何将这些响应与其请求匹配。

Linux公社 www.linuxidc.com

第17章　TCP：传输控制协议

17.1　引言

本章将介绍TCP为应用层提供的服务，以及TCP首部中的各个字段。随后的几章我们在了解TCP的工作过程中将对这些字段作详细介绍。

对TCP的介绍将由本章开始，并一直包括随后的 7章。第18章描述如何建立和终止一个TCP连接，第19和第20章将了解正常的数据传输过程，包括交互使用（远程登录）和批量数据传送（文件传输）。第21章提供TCP超时及重传的技术细节，第22和第23章将介绍两种其他的定时器。最后，第24章概述TCP新的特性以及TCP的性能。

17.2　TCP的服务

尽管TCP和UDP都使用相同的网络层（IP），TCP却向应用层提供与UDP完全不同的服务。TCP提供一种面向连接的、可靠的字节流服务。

面向连接意味着两个使用 TCP的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个 TCP连接。这一过程与打电话很相似，先拨号振铃，等待对方摘机说“喂”，然后才说明是谁。在第 18章我们将看到一个TCP连接是如何建立的，以及当一方通信结束后如何断开连接。

在一个TCP连接中，仅有两方进行彼此通信。在第 12章介绍的广播和多播不能用于TCP。

TCP通过下列方式来提供可靠性：

- 应用数据被分割成TCP认为最适合发送的数据块。这和 UDP完全不同，应用程序产生的数据报长度将保持不变。由 TCP传递给IP的信息单位称为报文段或段（ segment）（参见图1-7）。在18.4节我们将看到TCP如何确定报文段的长度。
- 当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。在第 21章我们将了解TCP协议中自适应的超时及重传策略。
- 当TCP收到发自TCP连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒，这将在 19.3节讨论。
- TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错， TCP将丢弃这个报文段和不确认收到此报文段（希望发端超时并重发）。
- 既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此 TCP报文段的到达也可能会失序。如果必要， TCP将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
- 既然IP数据报会发生重复， TCP的接收端必须丢弃重复的数据。
- TCP还能提供流量控制。 TCP连接的每一方都有固定大小的缓冲空间。 TCP的接收端只

Linux公社 www.linuxidc.com

允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

两个应用程序通过TCP连接交换8 bit字节构成的字节流。TCP不在字节流中插入记录标识符。我们将这称为字节流服务（byte stream service）。如果一方的应用程序先传10字节，又传20字节，再传50字节，连接的另一方将无法了解发方每次发送了多少字节。收方可以分4次接收这80个字节，每次接收20字节。一端将字节流放到TCP连接上，同样的字节流将出现在TCP连接的另一端。

另外，TCP对字节流的内容不作任何解释。TCP不知道传输的数据字节流是二进制数据，还是ASCII字符、EBCDIC字符或者其他类型数据。对字节流的解释由TCP连接双方的应用层解释。

这种对字节流的处理方式与Unix操作系统对文件的处理方式很相似。Unix的内核对一个应用读或写的内容不作任何解释，而是交给应用程序处理。对Unix的内核来说，它无法区分一个二进制文件与一个文本文件。

17.3 TCP的首部

TCP数据被封装在一个IP数据报中，如图17-1所示。

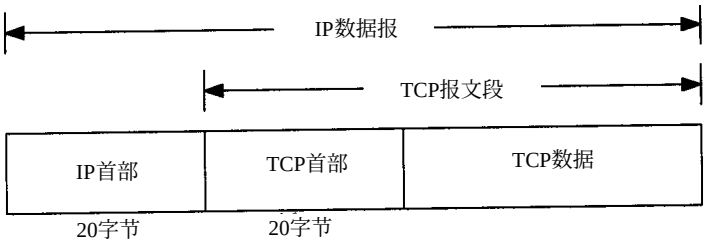


图17-1 TCP数据在IP数据报中的封装

图17-2显示TCP首部的数据格式。如果不计任选字段，它通常是20个字节。

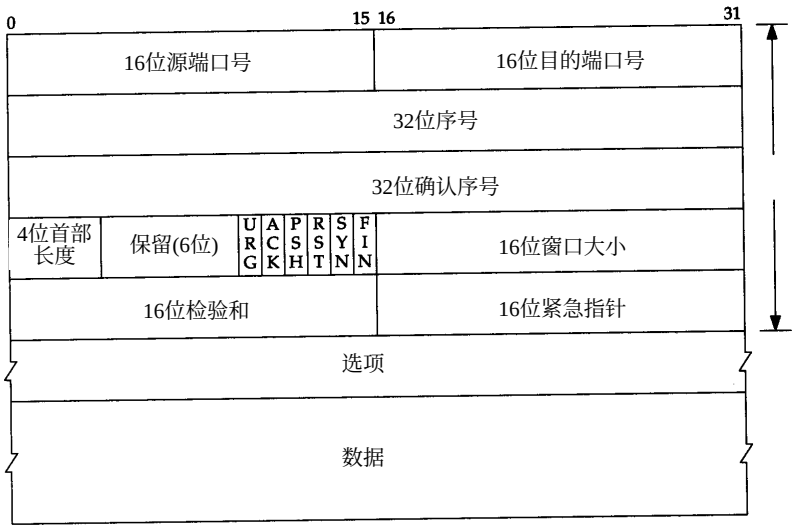


图17-2 TCP包首部

Linux公社 www.linuxidc.com

每个TCP段都包含源端和目的端的端口号，用于寻找发端和收端应用进程。这两个值加上IP首部中的源端IP地址和目的端IP地址唯一确定一个TCP连接。

有时，一个IP地址和一个端口号也称为一个插口（socket）。这个术语出现在最早的TCP规范（RFC793）中，后来它也作为表示伯克利版的编程接口（参见1.15节）。插口对（socket pair）(包含客户IP地址、客户端口号、服务器IP地址和服务器端口号的四元组)可唯一确定互联网络中每个TCP连接的双方。

序号用来标识从TCP发端向TCP收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则TCP用序号对每个字节进行计数。序号是32 bit的无符号数，序号到达$2^{32}-1$后又从0开始。

当建立一个新的连接时，SYN标志变1。序号字段包含由这个主机选择的该连接的初始序号ISN（Initial Sequence Number）。该主机要发送数据的第一个字节序号为这个ISN加1，因为SYN标志消耗了一个序号（将在下章详细介绍如何建立和终止连接，届时我们将看到FIN标志也要占用一个序号）。

既然每个传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加1。只有ACK标志（下面介绍）为1时确认序号字段才有效。

发送ACK无需任何代价，因为32 bit的确认序号字段和ACK标志一样，总是TCP首部的一部分。因此，我们看到一旦一个连接建立起来，这个字段总是被设置，ACK标志也总是被设置为1。

TCP为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

TCP可以表述为一个没有选择确认或否认的滑动窗口协议（滑动窗口协议用于数据传输将在20.3节介绍）。我们说TCP缺少选择确认是因为TCP首部中的确认序号表示发方已成功收到字节，但还不包含确认序号所指的字节。当前还无法对数据流中选定的部分进行确认。例如，如果1～1024字节已经成功收到，下一报文段中包含序号从2049～3072的字节，收端并不能确认这个新的报文段。它所能做的就是发回一个确认序号为1025的ACK。它也无法对一个报文段进行否认。例如，如果收到包含1025～2048字节的报文段，但它的检验和错，TCP接收端所能做的就是发回一个确认序号为1025的ACK。在21.7节我们将看到重复的确认如何帮助确定分组已经丢失。

首部长度给出首部中32 bit字的数目。需要这个值是因为任选字段的长度是可变的。这个字段占4 bit，因此TCP最多有60字节的首部。然而，没有任选字段，正常的长度是20字节。

在TCP首部中有6个标志比特。它们中的多个可同时被设置为1。我们在这儿简单介绍它们的用法，在随后的章节中有更详细的介绍。

URG　紧急指针（urgent pointer）有效（见20.8节）。

ACK　确认序号有效。

PSH　接收方应该尽快将这个报文段交给应用层。

RST　重建连接。

SYN　同步序号用来发起一个连接。这个标志和下一个标志将在第18章介绍。

FIN　发端完成发送任务。

Linux公社 www.linuxidc.com

TCP的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端正期望接收的字节。窗口大小是一个 16 bit字段，因而窗口大小最大为65535字节。在24.4节我们将看到新的窗口刻度选项，它允许这个值按比例变化以提供更大的窗口。

检验和覆盖了整个的TCP报文段：TCP首部和TCP数据。这是一个强制性的字段，一定是由发端计算和存储，并由收端进行验证。 TCP检验和的计算和UDP检验和的计算相似，使用如11.3节所述的一个伪首部。

只有当URG标志置1时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。 TCP的紧急方式是发送端向另一端发送紧急数据的一种方式。我们将在20.8节介绍它。

最常见的可选字段是最长报文大小，又称为 MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段（为建立连接而设置 SYN标志的那个段）中指明这个选项。它指明本端所能接收的最大长度的报文段。我们将在 18.4节更详细地介绍MSS选项，TCP的其他选项中的一些将在第24章中介绍。

从图17-2中我们注意到TCP报文段中的数据部分是可选的。我们将在 18章中看到在一个连接建立和一个连接终止时，双方交换的报文段仅有 TCP首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

17.4 小结

TCP提供了一种可靠的面向连接的字节流运输层服务。我们简单地介绍了 TCP首部中的各个字段，并在随后的几章里详细讨论它们。

TCP将用户数据打包构成报文段；它发送数据后启动一个定时器；另一端对收到的数据进行确认，对失序的数据重新排序，丢弃重复数据； TCP提供端到端的流量控制，并计算和验证一个强制性的端到端检验和。

许多流行的应用程序如Telnet、Rlogin、FTP和SMTP都使用TCP。

习题

17.1 我们已经介绍了以下几种分组格式： IP、ICMP、IGMP、UDP和TCP。每一种格式的首部中均包含一个检验和。对每种分组，说明检验和包括 IP数据报中的哪些部分，以及该检验和是强制的还是可选的。

17.2 为什么我们已经讨论的所有Internet协议（IP, ICMP, IGMP, UDP, TCP）收到有检验和错的分组都仅作丢弃处理？

17.3 TCP提供了一种字节流服务，而收发双方都不保持记录的边界。应用程序如何提供它们自己的记录标识？

17.4 为什么在TCP首部的开始便是源和目的的端口号？

17.5 为什么TCP首部有一个首部长度字段而UDP首部（图11-2）中却没有？

Linux公社 www.linuxidc.com

第18章　TCP连接的建立与终止

18.1　引言

TCP是一个面向连接的协议。无论哪一方向另一方发送数据之前，都必须先在双方之间建立一条连接。本章将详细讨论一个TCP连接是如何建立的以及通信结束后是如何终止的。

这种两端间连接的建立与无连接协议如 UDP不同。我们在第11章看到一端使用UDP向另一端发送数据报时，无需任何预先的握手。

18.2　连接的建立与终止

为了了解一个TCP连接在建立及终止时发生了什么，我们在系统 svr4上键入下列命令：

```
svr4 % telnet bsdi discard
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
```

键入Ctrl和右括号，使Telnet客户进程终止连接

`telnet`命令在与丢弃(discard)服务（参见1.12节）对应的端口上与主机 `bsdi`建立一条TCP连接。这服务类型正是我们需要观察的一条连接建立与终止的服务类型，而不需要服务器发起任何数据交换。

18.2.1　tcpdump的输出

图18-1显示了这条命令产生TCP报文段的`tcpdump`输出。

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>

2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522 win 4096
                                                    <mss 1024>

3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1823083522 win 4096

4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                    ack 1823083522 win 4096

5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 1415531523 win 4096

6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                    ack 1415531523 win 4096

7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

图18-1　TCP连接建立与终止的tcpdump 输出显示

这7个TCP报文段仅包含TCP首部。没有任何数据。

对于TCP段，每个输出行开始按如下格式显示：

Linux公社 www.linuxidc.com

源 > 目的: 标志

这里的标志代表TCP首部（图17-2）中6个标志比特中的4个。图18-2显示了表示标志的5个字符的含义。

标志	3字符缩写	描 述
S	SYN	同步序号
F	FIN	发送方完成数据发送
R	RST	复位连接
P	PSH	尽可能快地将数据送往接收进程
.		以上四个标志比特均置0

图18-2 tcpdump 对TCP首部中部分标志比特的字符表示

在这个例子中，我们看到了S、F和句点“.”标志符。我们将在以后看到其他的两个标志（R和P）。TCP首部中的其他两个标志比特——ACK 和 URG——tcpdump将作特殊显示。

图18-2所示的4个标志比特中的多个可能同时出现在一个报文段中，但通常一次只见到一个。

> RFC 1025 [Postel 1987]，“TCP and IP Bake Off”，将一种报文段称为Kamikaze分组[⊖]，在这样的报文段中有最大数量的标志比特同时被置为1（SYN, URG, PSH, FIN和1字节的数据）。这样的报文段也叫作nastygram, 圣诞树分组，灯测试报文段(lamp test segment)。

在第1行中，字段1415531521:1415531521(0)表示分组的序号是1415531521，而报文段中数据字节数为0。tcpdump显示这个字段的格式是开始的序号、一个冒号、隐含的结尾序号及圆括号内的数据字节数。显示序号和隐含结尾序号的优点是便于了解数据字节数大于 0时的隐含结尾序号。这个字段只有在满足条件（1）报文段中至少包含一个数据字节；或者（2）SYN、FIN或RST被设置为1时才显示。图18-1中的第1、2、4和6行是因为标志比特被置为1而显示这个字段的，在这个例子中通信双方没有交换任何数据。

在第2行中，字段ack 1415531522表示确认序号。它只有在首部中的ACK标志比特被设置1时才显示。

每行显示的字段win 4096表示发端通告的窗口大小。在这些例子中，我们没有交换任何数据，窗口大小就维持默认情况下的4096（我们将在20.4节中讨论TCP窗口大小）。

图18-1中的最后一个字段<mss 1024>表示由发端指明的最大报文段长度选项。发端将不接收超过这个长度的TCP报文段。这通常是为了避免分段（见 11.5节）。我们将在18.4节讨论最大报文段长度，而在18.10节介绍不同TCP 选项的格式。

18.2.2 时间系列

图18-3显示了这些分组序列的时间系列（在图6-11中已经首次介绍了这些时间系列的一些基本特性）。这个图显示出哪一端正在发送分组。我们也将对tcpdump输出作一些扩展（例如，印出SYN而不是S）。在这个时间系列中也省略窗口大小的值，因为它和我们的讨论无关。

18.2.3 建立连接协议

现在让我们回到图18-3所示的TCP协议中来。为了建立一条TCP连接：

⊖ Kamikaze是神风队队员或神风队所使用的飞机。在第二次世界大战末期，日本空军的神风队队员驾驶满载炸弹的飞机去撞击轰炸目标，企图与之同归于尽。

Linux公社 www.linuxidc.com

1) 请求端（通常称为客户）发送一个 SYN段指明客户打算连接的服务器的端口，以及初始序号（ISN，在这个例子中为1415531521）。这个SYN段为报文段1。

2) 服务器发回包含服务器的初始序号的 SYN报文段（报文段2）作为应答。同时，将确认序号设置为客户的ISN加1以对客户的SYN报文段进行确认。一个SYN将占用一个序号。

3) 客户必须将确认序号设置为服务器的 ISN加1以对服务器的SYN报文段进行确认（报文段3）。

这三个报文段完成连接的建立。这个过程也称为三次握手（ three-way handshake）。

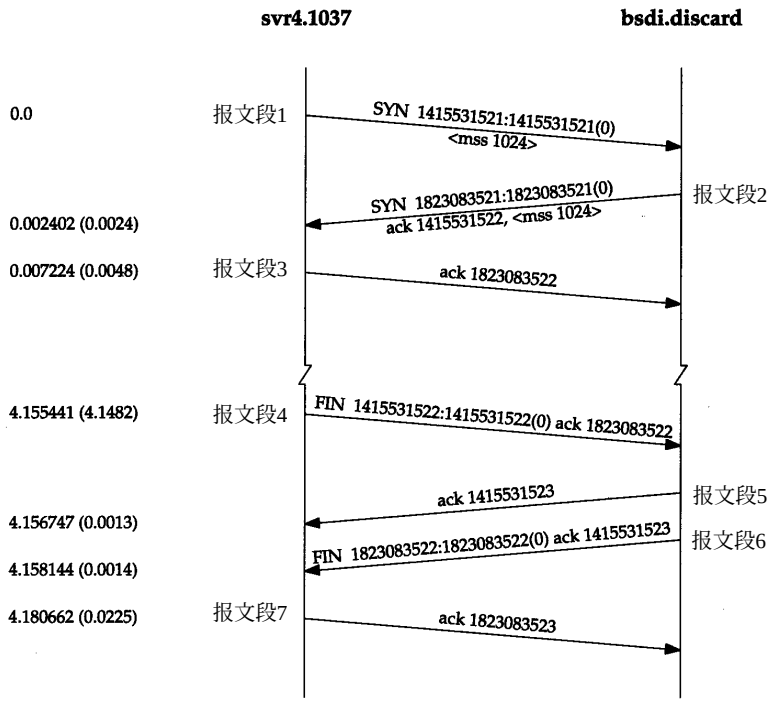


图18-3 连接建立与终止的时间系列

发送第一个SYN的一端将执行主动打开（active open）。接收这个SYN并发回下一个SYN的另一端执行被动打开（passive open）（在18.8节我们将介绍双方如何都执行主动打开）。

当一端为建立连接而发送它的 SYN时，它为连接选择一个初始序号。 ISN随时间而变化，因此每个连接都将具有不同的ISN。RFC 793 [Postel 1981c]指出ISN可看作是一个32比特的计数器，每4ms加1。这样选择序号的目的在于防止在网络中被延迟的分组在以后又被传送，而导致某个连接的一方对它作错误的解释。

> 如何进行序号选择？在4.4BSD（和多数的伯克利的实现版）中，系统初始化时初始的发送序号被初始化为1。这种方法违背了Host Requirements RFC（在这个代码中的一个注释确认这是一个错误）。这个变量每0.5秒增加64000，并每隔9.5小时又回到0（对应这个计数器每8 ms加1，而不是每4 ms加1）。另外，每次建立一个连接后，这个变量将增加64000。

报文段3与报文段4之间4.1秒的时间间隔是建立 TCP连接到向telnet键入quit命令来中止该连接的时间。

Linux公社 www.linuxidc.com

18.2.4 连接终止协议

建立一个连接需要三次握手，而终止一个连接要经过 4次握手。这由TCP的半关闭（half-close）造成的。既然一个TCP连接是全双工（即数据在两个方向上能同时传递），因此每个方向必须单独地进行关闭。这原则就是当一方完成它的数据发送任务后就能发送一个 FIN来终止这个方向连接。当一端收到一个 FIN，它必须通知应用层另一端几经终止了那个方向的数据传送。发送FIN通常是应用层进行关闭的结果。

收到一个FIN只意味着在这一方向上没有数据流动。一个 TCP连接在收到一个FIN后仍能发送数据。而这对利用半关闭的应用来说是可能的，尽管在实际应用中只有很少的 TCP应用程序这样做。正常关闭过程如图 18-3所示。我们将在18.5节中详细介绍半关闭。

首先进行关闭的一方（即发送第一个 FIN）将执行主动关闭，而另一方（收到这个 FIN）执行被动关闭。通常一方完成主动关闭而另一方完成被动关闭，但我们将在 18.9节看到双方如何都执行主动关闭。

图18-3中的报文段4发起终止连接，它由Telnet客户端关闭连接时发出。这在我们键入 quit命令后发生。它将导致TCP客户端发送一个FIN，用来关闭从客户到服务器的数据传送。

当服务器收到这个 FIN，它发回一个ACK，确认序号为收到的序号加 1（报文段5）。和SYN一样，一个FIN将占用一个序号。同时TCP服务器还向应用程序（即丢弃服务器）传送一个文件结束符。接着这个服务器程序就关闭它的连接，导致它的 TCP端发送一个FIN（报文段6），客户必须发回一个确认，并将确认序号设置为收到序号加1（报文段7）。

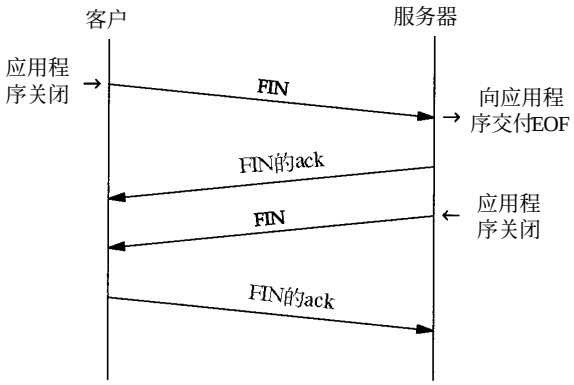


图18-4 连接终止期间报文段的正常交换

图18-4显示了终止一个连接的典型握手顺序。我们省略了序号。在这个图中，发送FIN将导致应用程序关闭它们的连接，这些FIN的ACK是由TCP软件自动产生的。

连接通常是由客户端发起的，这样第一个 SYN从客户传到服务器。每一端都能主动关闭这个连接（即首先发送 FIN）。然而，一般由客户端决定何时终止连接，因为客户进程通常由用户交互控制，用户会键入诸如“ quit ”一样的命令来终止进程。在图 18-4中，我们能改变上边的标识，将左方定为服务器，右方定为客户，一切仍将像显示的一样工作（例如在 14.4节中的第一个例子中就是由daytime服务器关闭连接的）。

18.2.5 正常的tcpdump输出

对所有的数值很大的序号进行排序是很麻烦的，因此默认情况下 `tcpdump`只在显示SYN报文段时显示完整的序号，而对其后的序号则显示它们与初始序号的相对偏移值（为了得到图18-1的输出显示必须加上 `-S`选项）。对应于图18-1的正常`tcpdump`显示如图18-5所示：

除非我们需要显示完整的序号，否则将在以下的例子中使用这种形式的输出显示。

Linux公社 www.linuxidc.com

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 2 win 4096
```

图18-5 连接建立与终止的正常tcpdump 输出

18.3 连接建立的超时

有很多情况导致无法建立连接。一种情况是服务器主机没有处于正常状态。为了模拟这种情况，我们断开服务器主机的电缆线，然后向它发出telnet命令。图18-6显示了tcpdump的输出。

```
1   0.0                  bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                     win 4096 <mss 1024>
                                                     [tos 0x10]
2   5.814797 ( 5.8148)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                     win 4096 <mss 1024>
                                                     [tos 0x10]
3  29.815436 (24.0006)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                     win 4096 <mss 1024>
                                                     [tos 0x10]
```

图18-6 建立连接超时的tcpdump 输出

在这个输出中有趣的一点是客户间隔多长时间发送一个 SYN，试图建立连接。第2个SYN与第1个的间隔是5.8秒，而第3个与第2个的间隔是24秒。

作为一个附注，这个例子运行38分钟后客户重新启动。这对应初始序号为291 008 001（约为38×60×64000×2）。我们曾经介绍过使用典型的伯克利实现版的系统将初始序号初始化为1，然后每隔0.5秒就增加64 000。

另外，因为这是系统启动后的第一个TCP连接，因此客户的端口号是1024。

图18-6中没有显示客户端在放弃建立连接尝试前进行 SYN重传的时间。为了了解它我们必须对telnet命令进行计时：

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

时间差值是 76秒。大多数伯克利系统将建立一个新连接的最长时间限制为 75秒。我们将在21.4节看到由客户发出的第 3个分组大约在16:25:29超时， 客户在它第3个分组发出后48秒而不是75秒后放弃连接。

18.3.1 第一次超时时间

在图18-6中一个令人困惑的问题是第一次超时时间为 5.8秒，接近6秒，但不准确，相比之

Linux公社 www.linuxidc.com

下第二个超时时间几乎准确地为 24秒。运行十多次测试，发现第一次超时时间在 5.59秒~5.93秒之间变化。然而，第二次超时时间则总是 24.00秒（精确到小数点后面两位）。

这是因为BSD版的TCP软件采用一种500 ms的定时器。这种500 ms的定时器用于确定本章中所有的各种各样的TCP超时。当我们键入telnet命令，将建立一个6秒的定时器（12个时钟滴答（tick）），但它可能在之后的 5.5秒~6秒内的任意时刻超时。图 18-7显示了这一发生过程。尽管定时器初始化为 12个时钟滴答，但定时计数器会在设置后的第一个 0~500 ms中的任意时刻减1。从那以后，定时计数器大约每隔 500 ms减1，但在第1个500 ms内是可变的（我们使用限定词“大约”是因为在 TCP每隔500 ms获得系统控制的瞬间，系统内核可能会优先处理其他中断）。

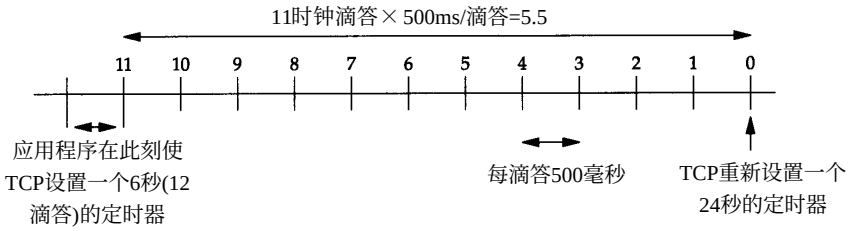


图18-7 TCP的500 ms定时器

当滴答计数器为 0时，6秒的定时器便会超时（见图 18-7），这个定时器会在以后的 24秒（48个滴答）重新复位。之后的下一个定时器将更接近 24秒，因为当TCP的500 ms定时器被内核调用时，它就会被修改一次。

18.3.2 服务类型字段

在图18-6中，出现了符号 [tos 0x10]。这是IP数据报内的服务类型（TOS）字段（参见图3-2）。BSD/386中的Telnet客户进程将这个字段设置为最小时延。

18.4 最大报文段长度

最大报文段长度（MSS）表示TCP传往另一端的最大块数据的长度。当一个连接建立时，连接的双方都要通告各自的 MSS。我们已经见过 MSS都是1024。这导致IP数据报通常是 40字节长：20字节的TCP首部和20字节的IP首部。

在有些书中，将它看作可“协商”选项。它并不是任何条件下都可协商。当建立一个连接时，每一方都有用于通告它期望接收的 MSS选项（MSS选项只能出现在SYN报文段中）。如果一方不接收来自另一方的 MSS值，则MSS就定为默认值536字节（这个默认值允许 20字节的IP首部和20字节的TCP首部以适合576字节IP数据报）。

一般说来，如果没有分段发生， MSS还是越大越好（这也并不总是正确，参见图 24-3和图24-4中的例子）。报文段越大允许每个报文段传送的数据就越多，相对 IP和TCP首部有更高的网络利用率。当 TCP发送一个SYN时，或者是因为一个本地应用进程想发起一个连接，或者是因为另一端的主机收到了一个连接请求，它能将 MSS值设置为外出接口上的 MTU长度减去固定的IP首部和TCP首部长度。对于一个以太网， MSS值可达1460字节。使用IEEE 802.3的封装（参见2.2节），它的MSS可达1452字节。

在本章见到的涉及 BSD/386和SVR4的MSS为1024，这是因为许多 BSD的实现版本需要

Linux公社 www.linuxidc.com

MSS为512的倍数。其他的系统，如SunOS 4.1.3、Solaris 2.2 和AIX 3.2.2，当双方都在一个本地以太网上时都规定MSS为1460。[Mogul 1993] 的比较显示了在以太网上1460的MSS在性能上比1024的MSS更好。

如果目的IP地址为“非本地的(nonlocal)”，MSS通常的默认值为536。而区分地址是本地还是非本地是简单的，如果目的IP地址的网络号与子网号都和我们的相同，则是本地的；如果目的IP地址的网络号与我们的完全不同，则是非本地的；如果目的 IP地址的网络号与我们的相同而子网号与我们的不同，则可能是本地的，也可能是非本地的。大多数 TCP实现版都提供了一个配置选项（附录E和图E-1），让系统管理员说明不同的子网是属于本地还是非本地。这个选项的设置将确定 MSS可以选择尽可能的大（达到外出接口的 MTU长度）或是默认值536。

MSS让主机限制另一端发送数据报的长度。加上主机也能控制它发送数据报的长度，这将使以较小MTU连接到一个网络上的主机避免分段。

考虑我们的主机slip，通过MTU为296的SLIP链路连接到路由器bsdi上。图18-8显示这些系统和主机sun。

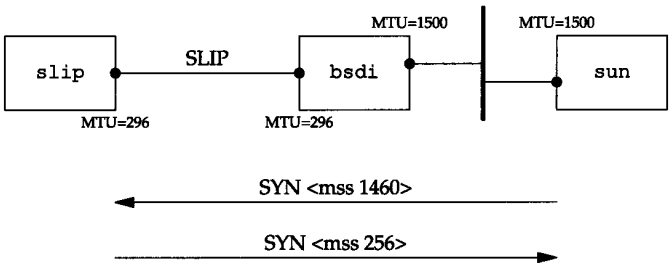


图18-8 显示sun与slip间TCP连接的MSS值

从sun向slip发起一个TCP连接，并使用tcpdump来观察报文段。图18-9显示这个连接的建立（省略了通告窗口大小）。

```
1  0.0                 sun.1093 > slip.discard: S 517312000:517312000(0)
                                                  <mss 1460>
2  0.10 (0.00)         slip.discard > sun.1093: S 509556225:509556225(0)
                                                  ack 517312001 <mss 256>
3  0.10 (0.00)         sun.1093 > slip.discard: . ack 1
```

图18-9 tcpdump 显示了从sun向slip建立连接的过程

在这个例子中，sun发送的报文段不能超过256字节的数据，因为它收到的MSS选项值为256（第2行）。此外，由于slip知道它外出接口的MTU长度为296，即使sun已经通告它的MSS为1460，但为避免将数据分段，它不会发送超过256字节数据的报文段。系统允许发送的数据长度小于另一端的MSS值。

只有当一端的主机以小于576字节的MTU直接连接到一个网络中，避免这种分段才会有效。如果两端的主机都连接到以太网上，都采用536的MSS，但中间网络采用296的MTU，也将会出现分段。使用路径上的MTU发现机制（参见24.2节）是关于这个问题的唯一方法。

18.5 TCP的半关闭

TCP提供了连接的一端在结束它的发送后还能接收来自另一端数据的能力。这就是所谓

Linux公社 www.linuxidc.com

的半关闭。正如我们早些时候提到的只有很少的应用程序使用它。

为了使用这个特性，编程接口必须为应用程序提供一种方式来说明“我已经完成了数据传送，因此发送一个文件结束（FIN）给另一端，但我还想接收另一端发来的数据，直到它给我发来文件结束（FIN）”。

如果应用程序不调用close而调用shutdown，且第2个参数值为1，则插口的API支持半关闭。然而，大多数的应用程序通过调用close终止两个方向的连接。

图18-10显示了一个半关闭的典型例子。让左方的客户端开始半关闭，当然也可以由另一端开始。开始的两个报文段和图18-4是相同的：初始端发出的FIN，接着是另一端对这个FIN的ACK报文段。但后面就和图18-4不同，因为接收半关闭的一方仍能发送数据。我们只显示一个数据报文段和一个ACK报文段，但可能发送了许多数据报文段（将在第19章讨论数据报文段和确认报文段的交换）。当收到半关闭的一端在完成它的数据传送后，将发送一个FIN关闭这个方向的连接，这将传送一个文件结束符给发起这个半关闭的应用进程。当对第二个FIN进行确认后，这个连接便彻底关闭了。

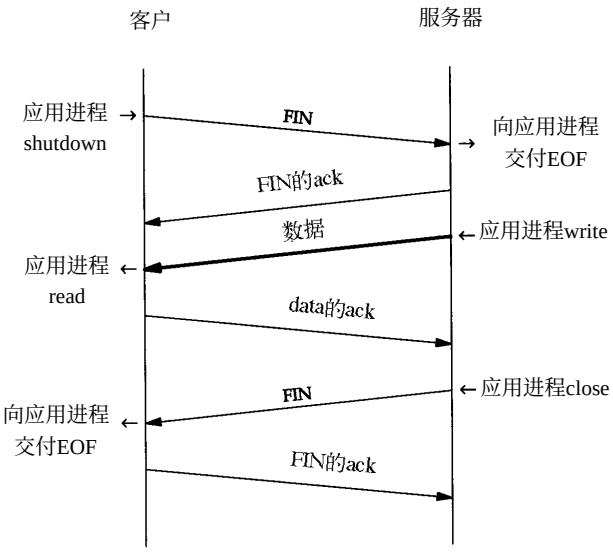


图18-10 TCP半关闭的例子

为什么要有半关闭？一个例子是Unix中的`rsh`(1)命令，它将完成在另一个系统上执行一个命令。命令

```
sun % rsh bsdi sort < datafile
```

将在主机`bsdi`上执行`sort`排序命令，`rsh`命令的标准输入来自文件`datafile`。`rsh`将在它与在另一主机上执行的程序间建立一个TCP连接。`rsh`的操作很简单：它将标准输入（`datafile`）复制给TCP连接，并将结果从TCP连接中复制给标准输出（我们的终端）。图18-11显示了这个建立过程（牢记TCP连接是全双工的）。

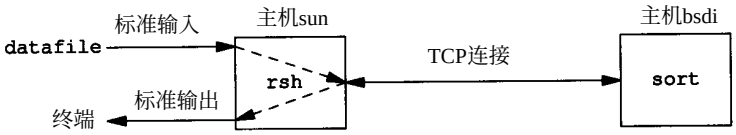


图18-11 命令：rsh bsdi sort < datafile

在远端主机`bsdi`上，`rshd`服务器将执行`sort`程序，它的标准输入和标准输出都是TCP连接。第14章的[Stevens 1990]详细介绍了有关Unix进程的结构，但这儿涉及的是使用TCP连接以及需要使用TCP的半关闭。

`sort`程序只有读取到所有输入数据后才能产生输出。所有的原始数据通过TCP连接从`rsh`客户端传送到`sort`服务器进行排序。当输入（`datafile`）到达文件尾时，`rsh`客户端

Linux公社 www.linuxidc.com

执行这个TCP连接的半关闭。接着sort服务器在它的标准输入（这个TCP连接）上收到一个文件结束符，对数据进行排序，并将结果写在它的标准输出上（TCP连接）。rsh客户端继续接收来自TCP连接另一端的数据，并将排序的文件复制到它的标准输出上。

没有半关闭，需要其他的一些技术让客户通知服务器，客户端已经完成了它的数据传送，但仍要接收来自服务器的数据。使用两个TCP连接也可作为一个选择，但使用半关闭的单连接更好。

18.6 TCP的状态变迁图

我们已经介绍了许多有关发起和终止TCP连接的规则。这些规则都能从图18-12所示的状态变迁图中得出。

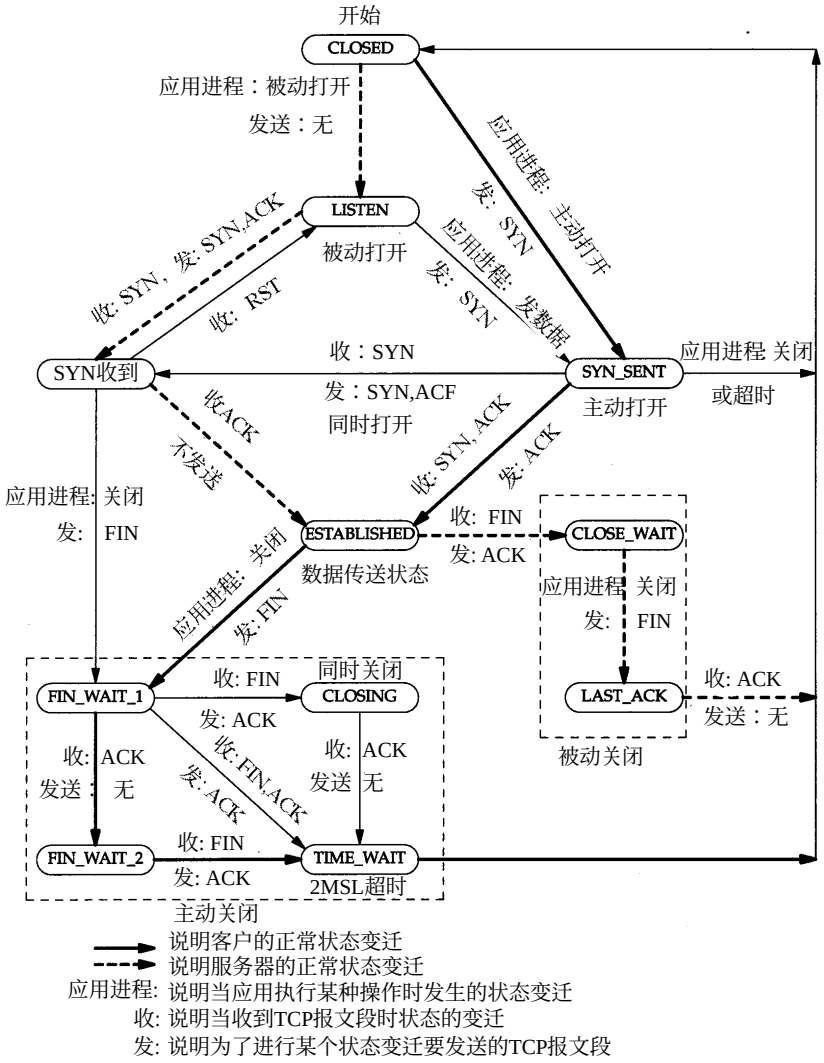


图18-12 TCP的状态变迁图

在这个图中要注意的第一点是一个状态变迁的子集是“典型的”。我们用粗的实线箭头表示正常的客户端状态变迁，用粗的虚线箭头表示正常的服务器状态变迁。

Linux公社 www.linuxidc.com

第二点是两个导致进入ESTABLISHED状态的变迁对应打开一个连接，而两个导致从ESTABLISHED状态离开的变迁对应关闭一个连接。ESTABLISHED状态是连接双方能够进行双向数据传递的状态。以后的章节将介绍这个状态。

将图中左下角4个状态放在一个虚线框内，并标为“主动关闭”。其他两个状态（CLOSE_WAIT和LAST_ACK）也用虚线框住，并标为“被动关闭”。

在这个图中11个状态的名称（CLOSED, LISTEN, SYN_SENT等）是有意与`netstat`命令显示的状态名称一致。`netstat`对状态的命名几乎与在RFC 793中的最初描述一致。CLOSED状态不是一个真正的状态，而是这个状态图的假想起点和终点。

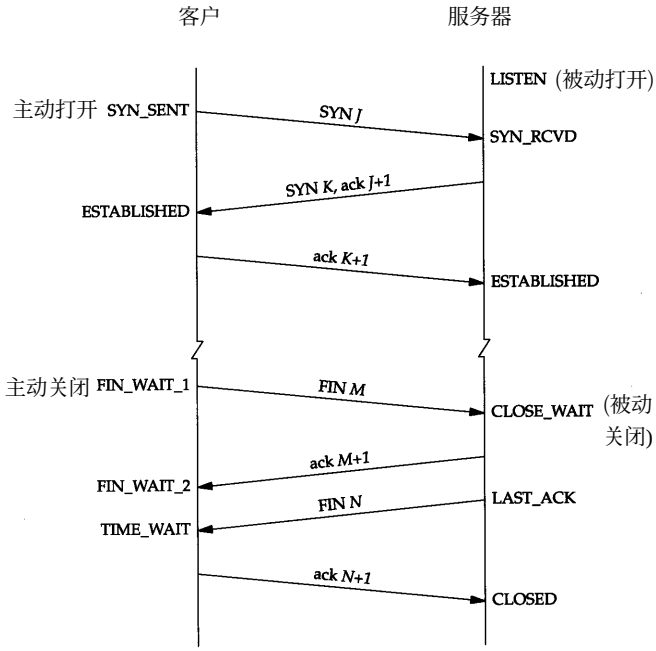


图18-13 TCP正常连接建立和终止所对应的状态

从LISTEN到SYN_SENT的变迁是正确的，但伯克利版的TCP软件并不支持它。

只有当SYN_RCVD状态是从LISTEN状态（正常情况）进入，而不是从SYN_SENT状态（同时打开）进入时，从SYN_RCVD回到LISTEN的状态变迁才是有效的。这意味着如果我们执行被动关闭（进入LISTEN），收到一个SYN，发送一个带ACK的SYN（进入SYN_RCVD），然后收到一个RST，而不是一个ACK，便又回到LISTEN状态并等待另一个连接请求的到来。

图18-13显示了在正常的TCP连接的建立与终止过程中，客户与服务器所经历的不同状态。它是图18-3的再现，不同的是仅显示了一些状态。

假定在图18-13中左边的客户执行主动打开，而右边的服务器执行被动打开。尽管图中显示出由客户端执行主动关闭，但和早前我们提到的一样，另一端也能执行主动关闭。

可以使用图18-12的状态图来跟踪图18-13的状态变化过程，以便明白每个状态的变化。

18.6.1 2MSL等待状态

TIME_WAIT状态也称为2MSL等待状态。每个具体TCP实现必须选择一个报文段最大生存时间MSL（Maximum Segment Lifetime）。它是任何报文段被丢弃前在网络内的最长时间。我们知道这个时间是有限的，因为TCP报文段以IP数据报在网络内传输，而IP数据报则有限制其生存时间的TTL字段。

> RFC 793 [Postel 1981c] 指出MSL为2分钟。然而，实现中的常用值是30秒，1分钟，或2分钟。

从第8章我们知道在实际应用中，对IP数据报TTL的限制是基于跳数，而不是定时器。

对一个具体实现所给定的MSL值，处理的原则是：当TCP执行一个主动关闭，并发回最

Linux公社 www.linuxidc.com

后一个ACK，该连接必须在TIME_WAIT状态停留的时间为2倍的MSL。这样可让TCP再次发送最后的ACK以防这个ACK丢失（另一端超时并重发最后的FIN）。

这种2MSL等待的另一个结果是这个 TCP连接在2MSL等待期间，定义这个连接的插口（客户的IP地址和端口号，服务器的 IP地址和端口号）不能再被使用。这个连接只能在 2MSL结束后才能再被使用。

遗憾的是，大多数TCP实现（如伯克利版）强加了更为严格的限制。在 2MSL等待期间，插口中使用的本地端口在默认情况下不能再被使用。我们将在下面看到这个限制的例子。

某些实现和API提供了一种避开这个限制的方法。使用插口API时，可说明其中的SO_REUSEADDR选项。它将让调用者对处于2MSL等待的本地端口进行赋值，但我们将看到TCP原则上仍将避免使用仍处于2MSL连接中的端口。

在连接处于2MSL等待时，任何迟到的报文段将被丢弃。因为处于 2MSL等待的、由该插口对(socket pair)定义的连接在这段时间内不能被再用，因此当要建立一个有效的连接时，来自该连接的一个较早替身（ incarnation）的迟到报文段作为新连接的一部分不可能不被曲解（一个连接由一个插口对来定义。一个连接的新的实例（ instance）称为该连接的替身）。

我们说图18-13中客户执行主动关闭并进入 TIME_WAIT是正常的。服务器通常执行被动关闭，不会进入TIME_WAIT状态。这暗示如果我们终止一个客户程序，并立即重新启动这个客户程序，则这个新客户程序将不能重用相同的本地端口。这不会带来什么问题，因为客户使用本地端口，而并不关心这个端口号是什么。

然而，对于服务器，情况就有所不同，因为服务器使用熟知端口。如果我们终止一个已经建立连接的服务器程序，并试图立即重新启动这个服务器程序，服务器程序将不能把它的这个熟知端口赋值给它的端点，因为那个端口是处于 2MSL连接的一部分。在重新启动服务器程序前，它需要在1~4分钟。

可以通过sock程序看到这一切。我们启动服务器程序，从一个客户程序进行连接，然后停止这个服务器程序。

```
sun % sock -v -s 6666                    启动服务器进程，在端口6666监听(在bsdi上执行客
                                         户进程与该端口进行连接)
connection on 140.252.13.33.6666 from 140.252.13.35.1081
^?                                       键入中断键停止服务器进程
sun % sock -s 6666                       并立即在同一端口重启服务器进程
can't bind local address: Address already in use
sun % netstat                            检测连接状态
Active Internet connections
Proto Recv-Q Send-Q  Local Address   Foreign Address     (state)
tcp        0      0  sun.6666        bsdi.1081          TIME_WAIT
                                         删除了许多其他行
```

当重新启动服务器程序时，程序报告一个差错信息说明不能绑定它的熟知端口，因为该端口已被使用（即它处于2MSL等待）。

运行netstat程序来查看连接的状态，以证实它的确处于 2MSL等待状态。

如果我们一直试图重新启动服务器程序，并测量它直到成功所需的时间，我们就能确定出2MSL值。对于SunOS 4.1.3、SVR4、BSD/386和AIX 3.2.2，它需要1分钟才能重新启动服务器程序，这意味着它们的MSL值为30秒。而对于Solaris 2.2 ，它需要4分

Linux公社 www.linuxidc.com

钟才能重新启动服务器程序，这表示它的MSL值为2分钟。

如果一个客户程序试图申请一个处于 2MSL等待的端口（客户程序通常不会这么做），就会出现同样的差错。

```
sun % sock -v bsdi echo                    启动客户进程，与回显服务器进程连接
connected on 140.252.13.33.1162 to 140.252.13.35.7
hello there                                键入这一行
hello there                                这一行应被服务器进程回显
^D                                         键入文件结束符终止客户进程

sun % sock -b1162 bsdi echo
can't bind local address: Address already in use
```

我们在第1次执行客户程序时采用 -v选项来查看它使用的本地端口为（ 1162）。第2次执行客户程序时则采用 - b选项来选择端口 1162为它的本地端口。正如我们所预料的那样，客户程序无法那么做，因为那个端口是一个还处于 2MSL等待连接的一部分。

需要再次强调2MSL等待的一个效果，因为我们将在第 27章的文件传输协议FTP中遇到它。和以前介绍的一样，一个插口对（即包含本地 IP地址、本地端口、远端 IP地址和远端端口的 4元组）在它处于2MSL等待时，将不能再被使用。尽管许多具体的实现中允许一个进程重新使用仍处于2MSL等待的端口（通常是设置选项 SO_REUSEADDR），但TCP不能允许一个新的连接建立在相同的插口对上。可通过下面的试验来看到这一点：

```
sun % sock -v -s 6666                      启动服务器进程，在端口6666监听(在bsdi上执行
                                           客户进程与该端口进行连接)

connection on 140.252.13.33.6666 from 140.252.13.35.1098
^?                                         键入中断键停止服务器进程

sun % sock -b6666 bsdi 1098                尝试在本地端口6666启动客户进程
can't bind local address: Address already in use

sun % sock -A -b6666 bsdi 1098             再次尝试，加上 - A选项
active open error: Address already in use
```

在第1次运行sock程序中，我们将它作为服务器程序，端口号为 6666，并从主机bsdi上的一个客户程序与它连接，这个客户程序使用的端口为 1098。我们终止服务器程序，因此它将执行主动关闭。这将导致 4元组 140.252.13.33（本地 IP地址）、6666（本地端口号）、140.252.13.35（另一端IP地址）和1098（另一端的端口号）在服务器主机进入 2MSL等待。

在第2次运行sock程序时，我们将它作为客户程序，并试图将它的本地端口号指明为 6666，同时与主机bsdi在端口1098上进行连接。但这个程序在试图将它的本地端口号赋值为 6666时产生了一个差错，因为这个端口是处于 2MSL等待4元组的一部分。

为了避免这个差错，我们再次运行这个程序，并使用选项 - A来设置前面提到的 SO_REUSEADDR。这将让sock程序能将它的本地端口号设置为 6666，但当我们试图进行主动打开时，又出现了一个差错。即使它能将它的本地端口设置为 6666，但它仍不能和主机bsdi在端口1098上进行连接，因为定义这个连接的插口对仍处于 2MSL等待状态。

如果我们试图从其他主机来建立这个连接会如何？首先我们必须在 sun上以 - A标记来重新启动服务器程序，因为它需要的端口（ 6666）是还处于2MSL等待连接的一部分。

```
sun % sock -A -s 6666                      启动服务器程序，在端口6666监听
```

接着，在2MSL等待结束前，我们在bsdi上启动客户程序：

```
bsdi % sock -b1098 sun 6666
```

Linux公社 www.linuxidc.com

```
connected on 140.252.13.35.1098 to 140.252.13.33.6666
```

不幸的是它成功了！这违反了 TCP规范，但被大多数的伯克利版实现所支持。这些实现允许一个新的连接请求到达仍处于 TIME_WAIT状态的连接，只要新的序号大于该连接前一个替身的最后序号。在这个例子中，新替身的 ISN被设置为前一个替身最后序号与 128 000的和。附录的RFC 1185 [Jacobsan、Braden和Zhang 1990] 指出了这项技术仍可能存在缺陷。

对于同一连接的前一个替身，这个具体实现中的特性让客户程序和服务器程序能连续地重用每一端的相同端口号，但这只有在服务器执行主动关闭才有效。我们将在图 27-8中使用FTP时看到这个2MSL等待条件的另一个例子。也见习题 18.5。

18.6.2 平静时间的概念

对于来自某个连接的较早替身的迟到报文段， 2MSL等待可防止将它解释成使用相同插口对的新连接的一部分。但这只有在处于 2MSL等待连接中的主机处于正常工作状态时才有效。

如果使用处于2MSL等待端口的主机出现故障，它会在 MSL秒内重新启动，并立即使用故障前仍处于2MSL的插口对来建立一个新的连接吗？如果是这样，在故障前从这个连接发出而迟到的报文段会被错误地当作属于重启后新连接的报文段。无论如何选择重启后新连接的初始序号，都会发生这种情况。

为了防止这种情况， RFC 793指出TCP在重启动后的MSL秒内不能建立任何连接。这就称为平静时间(quiet time)。

> 只有极少的实现版遵守这一原则，因为大多数主机重启动的时间都比MSL秒要长。

18.6.3 FIN_WAIT_2 状态

在FIN_WAIT_2状态我们已经发出了 FIN，并且另一端也已对它进行确认。除非我们在实行半关闭，否则将等待另一端的应用层意识到它已收到一个文件结束符说明，并向我们发一个FIN来关闭另一方向的连接。只有当另一端的进程完成这个关闭，我们这端才会从FIN_WAIT_2状态进入TIME_WAIT状态。

这意味着我们这端可能永远保持这个状态。另一端也将处于 CLOSE_WAIT状态，并一直保持这个状态直到应用层决定进行关闭。

> 许多伯克利实现采用如下方式来防止这种在FIN_WAIT_2状态的无限等待。如果执行主动关闭的应用层将进行全关闭，而不是半关闭来说明它还想接收数据，就设置一个定时器。如果这个连接空闲10分钟75秒，TCP将进入CLOSED状态。在实现代码的注释中确认这个实现代码违背协议的规范。

18.7 复位报文段

我们已经介绍了TCP首部中的RST比特是用于“复位”的。一般说来，无论何时一个报文段发往基准的连接（referenced connection）出现错误， TCP都会发出一个复位报文段（这里提到的“基准的连接”是指由目的 IP地址和目的端口号以及源 IP地址和源端口号指明的连接。这就是为什么RFC 793称之为插口）。

Linux公社 www.linuxidc.com

18.7.1 到不存在的端口的连接请求

产生复位的一种常见情况是当连接请求到达时，目的端口没有进程正在听。对于 UDP，我们在6.5节看到这种情况，当一个数据报到达目的端口时，该端口没在使用，它将产生一个ICMP端口不可达的信息。而TCP则使用复位。

产生这个例子也很容易，我们可使用 Telnet客户程序来指明一个目的端口没在使用的情况：

```
bsdi % telnet svr4 20000            端口20000未使用
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection refused
```

Telnet客户程序会立即显示这个差错信息。图 18-14显示了对应这个命令的分组交换过程。

```
1  0.0              bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                        win 4096 <mss 1024>
                                        [tos 0x10]
2  0.003771 (0.0038)    svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

图18-14 试图在不存在的端口上打开连接而产生的复位

在这个图中需要注意的值是复位报文段中的序号字段和确认序号字段。因为 ACK比特在到达的报文段中没有被设置为1，复位报文段中的序号被置为0，确认序号被置为进入的ISN加上数据字节数。尽管在到达的报文段中没有真正的数据，但 SYN比特从逻辑上占用了1字节的序号空间；因此，在这个例子中复位报文段中确认序号被置为 ISN与数据长度（0）、SYN比特所占的1的总和。

18.7.2 异常终止一个连接

我们在18.2节中看到终止一个连接的正常方式是一方发送 FIN。有时这也称为有序释放（orderly release），因为在所有排队数据都已发送之后才发送 FIN，正常情况下没有任何数据丢失。但也有可能发送一个复位报文段而不是 FIN来中途释放一个连接。有时称这为异常释放（abortive release）。

异常终止一个连接对应用程序来说有两个优点：（1）丢弃任何待发数据并立即发送复位报文段；（2）RST的接收方会区分另一端执行的是异常关闭还是正常关闭。应用程序使用的API必须提供产生异常关闭而不是正常关闭的手段。

使用sock程序能够观察这种异常关闭的过程。 Socket API通过“ linger on close ”选项（SO_LINGER）提供了这种异常关闭的能力。我们加上 -L选项并将停留时间设为0。这将导致连接关闭时进行复位而不是正常的 FIN。我们连接到处于服务器上的 sock程序，并键入一输入行：

```
bsdi %sock -L0 svr4 8888      这是客户程序，服务器程序显示后面
hello, world                  键入一行输入，它被发往到另一端
^D                            键入文件结束符，终止客户程序
```

图18-15是这个例子的tcpdump输出显示（在这个图中我们已经删除了所有窗口大小的说明，因为它们与讨论无关）。

第1~3行显示出建立连接的正常过程。第 4行发送我们键入的数据行（ 12个字符和Unix换

Linux公社 www.linuxidc.com

行符），第5行是对收到数据的确认。

```
1  0.0                  bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                 <mss 1024>
2  0.004975 (0.0050)    svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                 ack 671112194 <mss 1024>
3  0.006656 (0.0017)    bsdi.1099 > svr4.8888: . ack 1

4  4.833073 (4.8264)    bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)    svr4.8888 > bsdi.1099: . ack 14

6  9.527634 (4.5014)    bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

图18-15 使用复位（RST）而不是FIN来异常终止一个连接

第6行对应为终止客户程序而键入的文件结束符（Control_D）。由于我们指明使用异常关闭而不是正常关闭（命令行中的-L0选项），因此主机bsdi端的TCP发送一个RST而不是通常的FIN。RST报文段中包含一个序号和确认序号。需要注意的是 RST报文段不会导致另一端产生任何响应，另一端根本不进行确认。收到RST的一方将终止该连接，并通知应用层连接复位。

我们在服务器上得到下面的差错信息：

```
svr4 %sock -s 8888                    作为服务器进程运行，在端口8888监听
hello, world                          这行是客户端发送的
read error: Connection reset by peer
```

这个服务器程序从网络中接收数据并将它接收的数据显示到其标准输出上。通常，从它的TCP上收到文件结束符后便将结束，但这里我们看到当收到 RST时，它产生了一个差错。这个差错正是我们所期待的：连接被对方复位了。

18.7.3 检测半打开连接

如果一方已经关闭或异常终止连接而另一方却还不知道，我们将这样的 TCP连接称为半打开（Half-Open）的。任何一端的主机异常都可能导致发生这种情况。只要不打算在半打开连接上传输数据，仍处于连接状态的一方就不会检测另一方已经出现异常。

半打开连接的另一个常见原因是当客户主机突然掉电而不是正常的结束客户应用程序后再关机。这可能发生在使用 PC机作为Telnet的客户主机上，例如，用户在一天工作结束时关闭PC机的电源。当关闭PC机电源时，如果已不再有要向服务器发送的数据，服务器将永远不知道客户程序已经消失了。当用户在第二天到来时，打开 PC机，并启动新的Telnet客户程序，在服务器主机上会启动一个新的服务器程序。这样会导致服务器主机中产生许多半打开的TCP连接（在第23章中我们将看到使用TCP的keepalive选项能使TCP的一端发现另一端已经消失）。

能很容易地建立半打开连接。在bsdi上运行Telnet客户程序，通过它和svr4上的丢弃服务器建立连接。我们键入一行字符，然后通过 tcpdump进行观察，接着断开服务器主机与以太网的电缆，并重启服务器主机。这可以模拟服务器主机出现异常（在重启服务器之前断开以太网电缆是为了防止它向打开的连接发送 FIN，某些TCP在关机时会这么做）。服务器主机重启后，我们重新接上电缆，并从客户向服务器发送另一行字符。由于服务器的 TCP已经重新启动，它将丢失复位前连接的所有信息，因此它不知道数据报文段中提到的连接。 TCP的处理原则是接收方以复位作为应答。

Linux公社 www.linuxidc.com

```
bsdi % telnet svr4 discard          启动客户进程
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hi there                            运行已正确发送
                                    重新启动服务器主机
another line                        导致连接复位
Connection closed by foreign host.
```

图18-16是这个例子的tcpdump输出显示（已从这个输出中删除了窗口大小的说明、服务类型信息和MSS声明，因为它们与讨论无关）。

```
1    0.0                  bsdi.1102 > svr4.discard: S 1591752193:1591752193(0)
2    0.004811 (  0.0048)  svr4.discard > bsdi.1102: S 26368001:26368001(0)
                                                      ack 1591752194
3    0.006516 (  0.0017)  bsdi.1102 > svr4.discard: . ack 1

4    5.167679 (  5.1612)  bsdi.1102 > svr4.discard: P 1:11(10) ack 1
5    5.201662 (  0.0340)  svr4.discard > bsdi.1102: . ack 11

6  194.909929 (189.7083)  bsdi.1102 > svr4.discard: P 11:25(14) ack 1
7  194.914957 (  0.0050)  arp who-has bsdi tell svr4
8  194.915678 (  0.0007)  arp reply bsdi is-at 0:0:c0:6f:2d:40
9  194.918225 (  0.0025)  svr4.discard > bsdi.1102: R 26368002:26368002(0)
```

图18-16 复位作为半打开连接上数据段的应答

第1~3行是正常的连接建立过程。第 4行向丢弃服务器发送字符行“ hithere ”，第5行是确认。

然后是断开svr4的以太网电缆，重新启动svr4，并重新接上电缆。这个过程几乎需要 190秒。接着从客户端输入下一行（即“ another line ”），当我们键入回车键后，这一行被发往服务器（图 18-16的第6行）。这导致服务器产生一个响应，但要注意的是由于服务器主机经过重新启动，它的 ARP高速缓存为空，因此需要一个 ARP请求和应答（第 7、8行）。第9行表示RST被发送出去。客户收到复位报文段后显示连接已被另一端的主机终止（ Telnet客户程序发出的最后信息不再有什么价值）。

18.8 同时打开

两个应用程序同时彼此执行主动打开的情况是可能的，尽管发生的可能性极小。每一方必须发送一个SYN，且这些SYN必须传递给对方。这需要每一方使用一个对方熟知的端口作为本地端口。这又称为同时打开（ simultaneous open）。

例如，主机A中的一个应用程序使用本地端口 7777，并与主机B的端口8888执行主动打开。主机B中的应用程序则使用本地端口 8888，并与主机 A的端口7777执行主动打开。

这与下面的情况不同：主机A中的Telnet客户程序和主机B中Telnet的服务器程序建立连接，与此同时，主机 B中的Telnet客户程序与主机 A的Telnet服务器程序也建立连接。在这个 Telnet例子中，两个 Telnet服务器都执行被动打开，而不是主动打开，并且 Telnet客户选择的本地端口不是另一端Telnet服务器进程所熟悉的端口。

TCP是特意设计为了可以处理同时打开，对于同时打开它仅建立一条连接而不是两条连接（其他的协议族，最突出的是OSI运输层，在这种情况下将建立两条连接而不是一条连接）。

当出现同时打开的情况时，状态变迁与图 18-13所示的不同。两端几乎在同时发送 SYN，并进入SYN_SENT状态。当每一端收到SYN时，状态变为SYN_RCVD（如图18-12），同时它

Linux公社 www.linuxidc.com

们都再发SYN并对收到的SYN进行确认。当双方都收到SYN及相应的ACK时，状态都变迁为ESTABLISHED。图18-17显示了这些状态变迁过程。

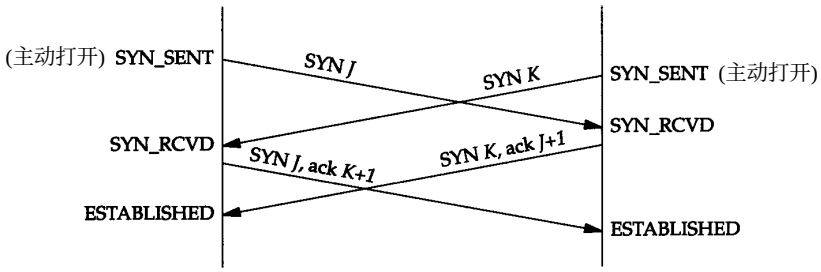


图18-17 同时打开期间报文段的交换

一个同时打开的连接需要交换4个报文段，比正常的三次握手多一个。此外，要注意的是我们没有将任何一端称为客户或服务器，因为每一端既是客户又是服务器。

一个例子

尽管很难，但仍有可能产生一个同时打开的连接。两端必须几乎在同时启动，以便收到彼此的SYN。只要两端有较长的往返时间就能保证这一点。这样我们将一端设置在主机 bsdi 上，另一端则设置在主机 vangogh.cs.berkeley.edu上。由于两端之间有一条拨号链路SLIP，它的往返时间对保证双方同步收到SYN是足够长的（几百毫秒）。

一端（bsdi）将本地端口设置为8888（使用命令行选项-b），并对另一端主机端口7777执行主动打开。

```
bsdi % sock -v -b8888 vangogh.cs.berkeley.edu 7777
connected on 140.252.13.35.8888 to 128.32.130.2.7777
TCP_MAXSEG = 512
hello, world                          键入该行
and hi there                          在另一端键入这一行
connection closed by peer             当收到FIN时的输出显示
```

另一端也几乎在同一时间将本地端口设置为7777，并对端口8888执行主动打开。

```
vangogh % sock -v -b7777 bsdi.tuc.noao.edu 8888
connected on 128.32.130.2.7777 to 140.252.13.35.8888
TCP_MAXSEG = 512
hello, world                          这是另一端键入的行
and hi there                          键入这行
^D                                    键入文件结束符EOF
```

我们指明带-v标志的sock程序来验证连接两端的IP地址和端口号。这个选项也显示每一端的MSS值。为证实两端确实在相互交谈，我们在每一端还输入一行字符，看它们是否会被送到另一端并显示出来。

图18-18显示了这个连接的段交换过程（我们删除了出现在来自 vangogh第一个SYN中的一些新的TCP选项，因为vangogh使用4.4BSD系统。将在18.10节介绍这些较新的选项）。注意两个SYN（第1~2行）后跟着两个带ACK的SYN（第3~4行）。它们将执行同时打开。

第5行显示了由bsdi发送给vangogh的输入行“ hello, world ”，第6行对此进行确认。第7~8行对应另一方向的输入行“ and hi there ”和确认。第9~12行显示正常的连接关闭。

许多伯克利版的TCP实现都不能正确地支持同时打开。在这些系统中，如果能够

Linux公社 www.linuxidc.com

进行SYN的同步接收，你将经历极多的报文段交换过程才能关闭它们。每个报文段交换过程包括每个方向上的一个 SYN和一个ACK。图18-12中从SYN_SENT到状态SYN_RCVD的变迁在许多TCP实现中很少测试过。

```
 1   0.0                 bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                     win 4096 <mss 512>
 2   0.213782 (0.2138)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                     win 8192 <mss 512>
 3   0.215399 (0.0016)   bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                     ack 1058199042 win 4096
                                                     <mss 512>
 4   0.340405 (0.1250)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                     ack 91904002 win 8192
                                                     <mss 512>
 5   5.633142 (5.2927)   bsdi.8888 > vangogh.7777: P 1:14(13) ack 1 win 4096
 6   6.100366 (0.4672)   vangogh.7777 > bsdi.8888: . ack 14 win 8192
 7   9.640214 (3.5398)   vangogh.7777 > bsdi.8888: P 1:14(13) ack 14 win 8192
 8   9.796417 (0.1562)   bsdi.8888 > vangogh.7777: . ack 14 win 4096
 9  13.060395 (3.2640)   vangogh.7777 > bsdi.8888: F 14:14(0) ack 14 win 8192
10  13.061828 (0.0014)   bsdi.8888 > vangogh.7777: . ack 15 win 4096
11  13.079769 (0.0179)   bsdi.8888 > vangogh.7777: F 14:14(0) ack 15 win 4096
12  13.299940 (0.2202)   vangogh.7777 > bsdi.8888: . ack 15 win 8192
```

图18-18 同时打开期间的报文段交换过程

18.9 同时关闭

我们在以前讨论过一方（通常但不总是客户方）发送第一个 FIN执行主动关闭。双方都执行主动关闭也是可能的，TCP协议也允许这样的同时关闭（simultaneous close）。

在图18-12中，当应用层发出关闭命令时，两端均从 ESTABLISHED变为FIN_WAIT_1。这将导致双方各发送一个 FIN，两个FIN经过网络传送后分别到达另一端。收到 FIN后，状态由FIN_WAIT_1变迁到CLOSING，并发送最后的 ACK。当收到最后的 ACK时，状态变化为TIME_WAIT。图18-19总结了这些状态的变化。

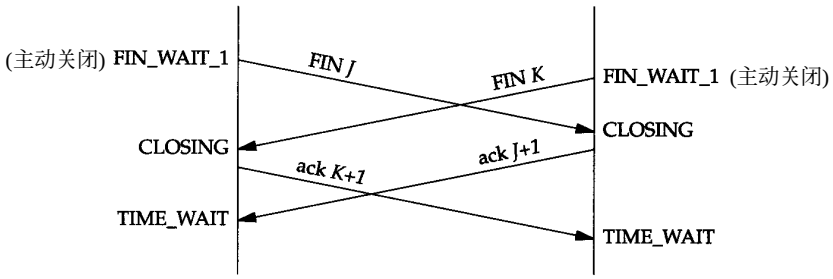


图18-19 同时关闭期间的报文段交换

同时关闭与正常关闭使用的段交换数目相同。

18.10 TCP 选项

TCP首部可以包含选项部分（图 17-2）。仅在最初的TCP规范中定义的选项是选项表结束、无操作和最大报文段长度。在我们的例子中，几乎每个 SYN报文段中我们都遇到过MSS选项。

新的RFC，主要是RFC 1323 [Jacobson, Braden和Borman 1992]，定义了新的TCP选项，

Linux公社 www.linuxidc.com

这些选项的大多数只在最新的 TCP实现中才能见到（我们将在第 24章介绍这些新选项）。图18-20显示了当前TCP选项的格式，这些选项的定义出自于 RFC 793和RFC 1323。

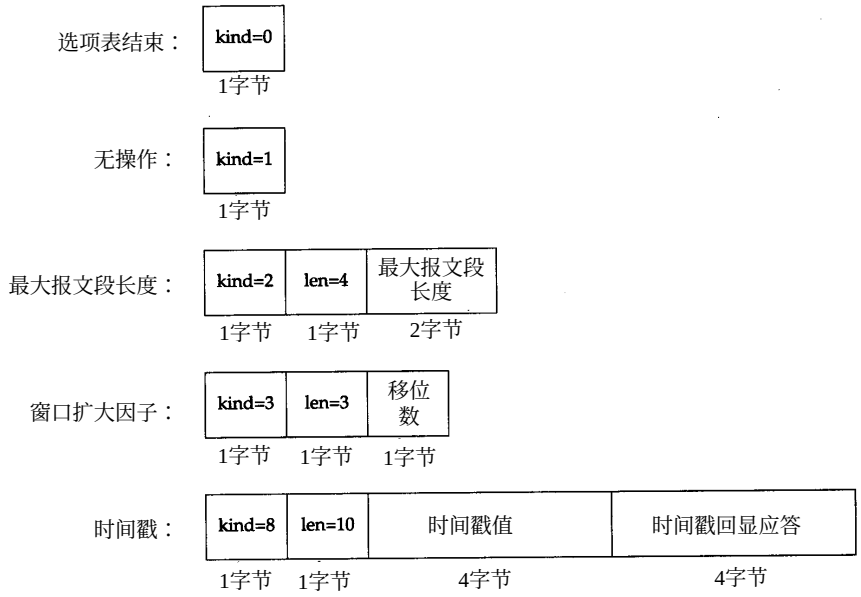


图18-20 TCP选项

每个选项的开始是1字节kind字段，说明选项的类型。kind字段为0和1的选项仅占1个字节。其他的选项在kind字节后还有len字节。它说明的长度是指总长度，包括 kind字节和len字节。

设置无操作选项的原因在于允许发方填充字段为 4字节的倍数。如果我们使用 4.4BSD系统进行初始化TCP连接，tcpdump将在初始的SYN上显示下面TCP选项：

<mss 512, nop, wscale 0, nop, nop, timestamp 146647 0>

MSS选项设置为512，后面是NOP，接着是窗口扩大选项。第一个 NOP用来将窗口扩大选项填充为4字节的边界。同样， 10字节的时间戳选项放在两个 NOP后，占12字节，同时使两个 4字节的时间戳满足4字节边界。

其他kind值为4、5、6和7的四个选项称为选择ACK及回显选项。由于回显选项已被时间戳选项取代，而目前定义的选择 ACK选项仍未定论，并未包括在 RFC 1323中，因此图18-20没有将它们列出。另外，作为 TCP事务（第24.7节）的T/TCP建议也指明kind为11, 12和 13的三个选项。

18.11 TCP 服务器的设计

我们在1.8节说过大多数的TCP服务器进程是并发的。当一个新的连接请求到达服务器时，服务器接受这个请求，并调用一个新进程来处理这个新的客户请求。不同的操作系统使用不同的技术来调用新的服务器进程。在 Unix系统下，常用的技术是使用 `fork`函数来创建新的进程。如果系统支持，也可使用轻型进程，即线程（ thread）。

我们感兴趣的是 TCP与若干并发服务器的交互作用。需要回答下面的问题：当一个服务器进程接受一来自客户进程的服务请求时是如何处理端口的？如果多个连接请求几乎同时到

Linux公社 www.linuxidc.com

达会发生什么情况？

18.11.1 TCP服务器端口号

通过观察任何一个 TCP服务器，我们能了解 TCP如何处理端口号。我们使用 netstat 命令来观察 Telnet服务器。下面是在没有 Telnet连接时的显示（只留下显示 Telnet服务器的行）。

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  *.23                   *.*                    LISTEN
```

-a标志将显示网络中的所有主机端，而不仅仅是处于 ESTABLISHED的主机端。-n标志将以点分十进制的形式显示 IP地址，而不是通过 DNS将地址转化为主机名，同时还要求显示端口号（例如为 23）而不是服务名称（如 Telnet）。-f inet选项则仅要求显示使用 TCP或 UDP的主机。

显示的本地地址为 *.23，星号通常又称为通配符。这表示传入的连接请求（即 SYN）将被任何一个本地接口所接收。如果该主机是多接口主机，我们将制定其中的一个 IP地址为本地IP地址，并且只接收来自这个接口的连接（在本节后面我们将看到这样的例子）。本地端口为23，这是Telnet的熟知端口号。

远端地址显示为 *.*，表示还不知道远端 IP地址和端口号，因为该端还处于 LISTEN状态，正等待连接请求的到达。

现在我们在主机 slip（140.252.13.65）启动一个 Telnet客户程序来连接这个 Telnet服务器。以下是netstat程序的输出行：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23    140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

端口为23的第1行表示处于ESTABLISHED 状态的连接。另外还显示了这个连接的本地 IP地址、本地端口号、远端 IP地址和远端端口号。 本地IP地址为该连接请求到达的接口（以太网接口，140.252.13.33）。

处于LISTEN状态的服务器进程仍然存在。这个服务器进程是当前 Telnet服务器用于接收其他的连接请求。当传入的连接请求到达并被接收时，系统内核中的 TCP模块就创建一个处于ESTABLISHED状态的进程。另外，注意处于 ESTABLISHED状态的连接的端口不会变化：也是23，与处于LISTEN状态的进程相同。

现在我们在主机slip上启动另一个Telnet客户进程，并仍与这个 Telnet服务器进行连接。以下是netstat程序的输出行：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23    140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23    140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

现在我们有两条从相同主机到相同服务器的处于 ESTABLISHED的连接。它们的本地端口号均为23。由于它们的远端端口号不同，这不会造成冲突。因为每个 Telnet客户进程要使用一个外

Linux公社 www.linuxidc.com

设端口，并且这个外设端口会选择为主机（slip）当前未曾使用的端口，因此它们的端口号肯定不同。

这个例子再次重申TCP使用由本地地址和远端地址组成的4元组：目的IP地址、目的端口号、源IP地址和源端口号来处理传入的多个连接请求。TCP仅通过目的端口号无法确定那个进程接收了一个连接请求。另外，在三个使用端口23的进程中，只有处于LISTEN的进程能够接收新的连接请求。处于ESTABLISHED的进程将不能接收SYN报文段，而处于LISTEN的进程将不能接收数据报文段。

下面我们从主机solaris上启动第3个Telnet客户进程，这个主机通过SLIP链路与主机sun相连，而不是以太网接口。

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.23       140.252.1.32.34603     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                  *.*                    LISTEN
```

现在第一个ESTABLISHED连接的本地IP地址对应多地址主机sun中的SLIP链路接口地址（140.252.1.29）。

18.11.2 限定的本地IP地址

我们来看看当服务器不能任选其本地IP地址而必须使用特定的IP地址时的情况。如果我们为sock程序指明一个IP地址（或主机名），并将它作为服务器，那么该IP地址就成为处于LISTEN服务器的本地IP地址。例如

```
sun % sock -s 140.252.1.29 8888
```

使这个服务器程序的连接仅局限于来自SLIP接口（140.252.1.29）。netstat的显示说明了这一点：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

如果我们从主机solaris通过SLIP链路与这个服务器相连接，它将正常工作。

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     140.252.1.32.34614     ESTABLISHED
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

但如果我们试图从以太网（140.252.13）中的主机与这个服务器进行连接，连接请求将被TCP模块拒绝。如果使用tcpdump来观察这一切，对连接请求SYN的响应是一个如图18-21所示的RST。

```
1  0.0                  bsdi.1026 > sun.8888: S 3657920001:3657920001(0)
                                                win 4096 <mss 1024>

2  0.000859 (0.0009)    sun.8888 > bsdi.1026: R 0:0(0) ack 3657920002 win 0
```

图18-21 具有限定本地IP地址服务器对连接请求的拒绝

这个连接请求将不会到达服务器的应用程序，因为它根据应用程序中指定的本地IP地址被内核中的TCP模块拒绝。

Linux公社 www.linuxidc.com

18.11.3 限定的远端IP地址

在11.12节，我们知道UDP服务器通常在指定IP本地地址和本地端口外，还能指定远端 IP地址和远端端口。RFC 793中显示的接口函数允许一个服务器在执行被动打开时，可指明远端插口（等待一个特定的客户执行主动打开），也可不指明远端插口（等待任何客户）。

遗憾的是，大多数API都不支持这么做。服务器必须不指明远端插口，而等待连接请求的到来，然后检查客户端的IP地址和端口号。

图18-22总结了TCP服务器进行连接时三种类型的地址绑定。在三种情况中， lport是服务器的熟知端口，而localIP必须是一个本地接口的IP地址。表中行的顺序正是TCP模块在收到一个连接请求时确定本地地址的顺序。最常使用的绑定（第 1行，如果支持的话）将最先尝试，最不常用的（最后一行两端的IP地址都没有制定）将最后尝试。

本地地址	远端地址	描　述
localIP.lport	*foreignIP.fport*	限制到一个客户进程(通常不支持)
localIP.lport	*.*	限制为到达一个本地接口：Local IP的连接
*.*lport*	*.*	接收发往Lport的所有连接

图18-22 TCP服务器本地和远端IP地址及端口号的规范

18.11.4 呼入连接请求队列

一个并发服务器调用一个新的进程来处理每个客户请求，因此处于被动连接请求的服务器应该始终准备处理下一个呼入的连接请求。那正是使用并发服务器的根本原因。但仍有可能出现当服务器在创建一个新的进程时，或操作系统正忙于处理优先级更高的进程时，到达多个连接请求。当服务器正处于忙时， TCP是如何处理这些呼入的连接请求？

在伯克利的TCP实现中采用以下规则：

1) 正等待连接请求的一端有一个固定长度的连接队列，该队列中的连接已被 TCP接受（即三次握手已经完成），但还没有被应用层所接受。
   注意区分TCP接受一个连接是将其放入这个队列，而应用层接受连接是将其从该队列中移出。
2) 应用层将指明该队列的最大长度，这个值通常称为积压值 (backlog)。它的取值范围是0~5之间的整数，包括0和5（大多数的应用程序都将这个值说明为 5）。
3) 当一个连接请求（即SYN）到达时， TCP使用一个算法，根据当前连接队列中的连接数来确定是否接收这个连接。我们期望应用层说明的积压值为这一端点所能允许接受连接的最大数目，但情况不是那么简单。图18-23显示了积压值与传统的伯克利系统和 Solaris 2.2所能允许的最大接受连接数之间的关系。注意，积压值说明的是 TCP监听的端点已被TCP接受而等待应用层接受的最大连接数。这个积压值对系统所允许的最大连接数，或者并发服务器所能并发处理的客户数，并无影响。

积压值	最大排队的连接数	
	传统的BSD	Solaris 2.2
0	1	0
1	2	1
2	4	2
3	5	3
4	7	4
5	8	5

图18-23 对正在听的端点所允许接受的最大连接数

在这个图中，Solaris系统规定的值正如我们所期望的。而传统的BSD系统，将这个

Linux公社 www.linuxidc.com

值（由于某些原因）设置为积压值乘3除以2，再加1。

4) 如果对于新的连接请求，该 TCP监听的端点的连接队列中还有空间（基于图 18-23），TCP模块将对SYN进行确认并完成连接的建立。但应用层只有在三次握手中的第三个报文段收到后才会知道这个新连接时。另外，当客户进程的主动打开成功但服务器的应用层还不知道这个新的连接时，它可能会认为服务器进程已经准备好接收数据了（如果发生这种情况，服务器的TCP仅将接收的数据放入缓冲队列）。

5) 如果对于新的连接请求，连接队列中已没有空间， TCP将不理会收到的SYN。也不发回任何报文段（即不发回 RST）。如果应用层不能及时接受已被 TCP接受的连接，这些连接可能占满整个连接队列，客户的主动打开最终将超时。

通过sock程序能了解这种情况。我们调用它，并使用新的选项（ -O）。让它在创建一个新的服务器进程后而没有接受任何连接请求之前暂停下来。如果在它暂停期间又调用了多个客户进程，它将导致接受连接队列被填满，通过 tcpdump能够看到这一切。

```
bsdi % sock -s -v -q1 -O30 5555
```

-q1选项将服务器端的积压值置 1。在这种情况下，传统的 BSD系统中的队列允许接受两个连接请求（图 18-23）。-O30选项使程序在接受任何客户连接之前暂停 30秒。在这30秒内，我们可启动其他客户进程来填充这个队列。在主机 sun上启动4个客户进程。

图18-24显示了tcpdump的输出，首先是第 1个客户进程的第 1个SYN（省略窗口大小和MSS声明。当TCP连接建立时，将客户进程的端口号用粗体标出）。

端口为1090的第一个客户连接请求被 TCP接受（报文段1~3）。端口为1091的第2个客户连接请求也被 TCP接受（报文段 4~6）。而服务器的应用仍处于休眠状态，还未接受任何连接。目前的一切工作都由内核中的 TCP模块完成。另外，两个客户进程已经成功地完成了它们的主动打开，因为它们建立连接的三次握手已经完成。

```
 1   0.0                  sun.1090 > bsdi.7777: S 1617152000:1617152000(0)
 2   0.002310 ( 0.0023)   bsdi.7777 > sun.1090: S 4164096001:4164096001(0)
                                                  ack 1617152001
 3   0.003098 ( 0.0008)   sun.1090 > bsdi.7777: . ack 1

 4   4.291007 ( 4.2879)   sun.1091 > bsdi.7777: S 1617792000:1617792000(0)
 5   4.293349 ( 0.0023)   bsdi.7777 > sun.1091: S 4164672001:4164672001(0)
                                                  ack 1617792001
 6   4.294167 ( 0.0008)   sun.1091 > bsdi.7777: . ack 1

 7   7.131981 ( 2.8378)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
 8  10.556787 ( 3.4248)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
 9  12.695916 ( 2.1391)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
10  16.195772 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
11  24.695571 ( 8.4998)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)

12  28.195454 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
13  28.197810 ( 0.0024)   bsdi.7777 > sun.1093: S 4167808001:4167808001(0)
                                                  ack 1618688001
14  28.198639 ( 0.0008)   sun.1093 > bsdi.7777: . ack 1

15  48.694931 (20.4963)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
16  48.697292 ( 0.0024)   bsdi.7777 > sun.1092: S 4170496001:4170496001(0)
                                                  ack 1618176001
17  48.698145 ( 0.0009)   sun.1092 > bsdi.7777: . ack 1
```

图18-24 积压值例子的tcpdump 输出

Linux公社 www.linuxidc.com

我们接着在报文段7（端口1092）和报文段8（端口1093）启动第3和第4个客户进程。由于服务器的连接队列已满，TCP将不理会两个SYN。这两个客户进程在报文段9, 10, 11, 12, 15重发它们的SYN。第4个客户进程的第3个SYN重传被接受了，因为服务器程序的30秒休眠结束后，它将已接受的两个连接从队列中移出，使连接队列变空（服务器程序接收连接的时间是28.19，小于30的原因在于启动服务器程序后它需要几秒的时间来启动第1个客户进程(报文段1，显示的就是启动时间))。第3个客户进程的第4个SYN重传这时将被接受（报文段15~17）。服务器程序先接受第4个客户连接（端口1093）的原因是服务器程序30秒休眠与客户程序重传之间的定时交互作用。

我们期望接收连接队列按先进先出顺序传递给应用层。如TCP接受了端口为1090和1091的连接，我们希望应用层先接受端口为1090的连接，然后再接受端口为1091的连接。但许多伯克利的TCP实现都出现按后进先出的传递顺序，这个错误已存在了多年。产商最近已开始改正这个错误，但在如SunOS 4.13等系统中仍存在这个问题。

当队列已满时，TCP将不理会传入的SYN，也不发回RST作为应答，因为这是一个软错误，而不是一个硬错误。通常队列已满是由于应用程序或操作系统忙造成的，这样可防止应用程序对传入的连接进行服务。这个条件在一个很短的时间内可以改变。但如果服务器的TCP以系统复位作为响应，客户进程的主动打开将被废弃（如果服务器程序没有启动我们就会遇到）。由于不应答SYN，服务器程序迫使客户TCP随后重传SYN，以等待连接队列有空间接受新的连接。

这个例子中有一个巧妙之处，这在大多TCP/IP的具体实现中都能见到，就是如果服务器的连接队列未满时，TCP将接受传入的连接请求（即SYN），但并不让应用层了解该连接源于何处（即不告知源IP地址和源端口）。这不是TCP所要求的，而只是共同的实现技术（如伯克利源代码通常都这么做）。如果一个API如TLI（见1.15节）向应用程序提供了解连接请求的到来的方法，并允许应用程序选择是否接受连接。当应用程序假定被告知连接请求已经到来时，TCP的三次握手已经结束！其他运输层的实现可能将连接请求的到达与接受分开（如OSI的运输层），但TCP不是这样。

Solaris 2.2提供了一个选项使TCP只有在应用程序说可以接受（`tcp_eager_listeners`见E.4），才允许接受传入的连接请求。

这种行为也意味着TCP服务器无法使客户进程的主动打开失效。当一个新的客户连接传递给服务器的应用程序时，TCP的三次握手就结束了，客户的主动打开已经完全成功。如果服务器的应用程序此时看到客户的IP地址和端口号，并决定是否为该客户进行服务，服务器所能做的就是关闭连接（发送FIN），或者复位连接（发送RST）。无论哪种情况，客户进程都认为一切正常，因为它的主动打开已经完成，并且已经向服务器程序发送过请求。

18.12 小结

两个进程在使用TCP交换数据之前，它们之间必须建立一条连接。完成后，要关闭这个连接。本章已经详细介绍了如何使用三次握手来建立连接以及使用4个报文段来关闭连接。

我们用`tcpdump`程序显示了TCP首部中的各个字段。也了解了连接建立是如何超时，连

Linux公社 www.linuxidc.com

接复位是如何发送，使用半打开连接发生的情况以及 TCP是如何提供半关闭、同时打开和同时关闭。

弄清TCP操作的关键在于它的状态变迁图。我们跟踪了连接建立与关闭的步骤以及它们的状态变迁过程。还讨论了在设计TCP并发服务器时TCP连接建立的具体实现方法。

一个TCP连接由一个4元组唯一确定：本地IP地址、本地端口号、远端IP地址和远端端口号。无论何时关闭一个连接，一端必须保持这个连接，我们看到 TIME_WAIT状态将处理这个问题。处理的原则是执行主动打开的一端在进入这个状态时要保持的时间为 TCP实现中规定的MSL值的两倍。

习题

18.1 在18.2节我们说初始序号（ISN）正常情况下由1开始，并且每0.5秒增加64000，每次执行一个主动打开。这意味着 ISN的最低三位通常总是 001。但在图18-3中，两个方向上ISN中的最低三位都是521。究竟是怎么回事？

18.2 在图18-15中，我们键入12个字符，看到TCP发送了13个字节。在图18-16中我们键入8个字符，但TCP发送了10个字符。为什么在第1种情况下增加1个字节，而在第2种情况下增加2个字节？

18.3 半打开连接和半关闭连接的区别是什么？

18.4 如果启动sock程序作为一个服务器程序，然后终止它（还没有客户进程与它相连接），我们能立即重新启动这个服务器程序。这意味着它没有经历 2MSL等待状态。用状态变迁来解释这一切。

18.5 在18.6节我们知道一个客户进程不能重新使用同一个本地端口，如果该端口是仍处于2MSL等待连接的一部分。但如果 sock程序作为客户程序连续运行两次，并且连接到daytime服务器上，我们就能重新使用同一本地端口。另外，对一个仍处于 2MSL等待的连接，也能为它创建一个替身。这将如何做？

```
sun % sock -v bsdi daytime
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:54:51 1993
connection closed by peer
sun % sock -v -b1163 bsdi daytime      重用相同的本地端口号
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:55:01 1993
connection closed by peer
```

18.6 在18.6节的最后，我们介绍了 FIN_WAIT_2状态，提到如果应用程序仅过 11分钟后实行完全关闭（不是半关闭），许多具体的实现都将一个连接由这个状态转移到 CLOSED状态。如果另一端（处于CLOSE_WAIT状态）在宣布关闭（即发送 FIN）之前等待了12分钟，这一端的TCP将如何响应这个FIN？

18.7 对于一个电话交谈，哪一方是主动打开，哪一方是被动打开？是否允许同时打开？是否允许同时关闭？

18.8 在图18-6中，我们没有见到一个ARP请求或一个ARP应答。显然主机svr4的硬件地址一定在bsdi的ARP高速缓存中。如果这个ARP高速缓存不存在，这个图会有什么变化？

18.9 解释如下的tcpdump输出，并和图18-13进行比较。

Linux公社 www.linuxidc.com

```
1  0.0                 solaris.32990 > bsdi.discard: S 40140288:40140288(0)
                                                     win 8760 <mss 1460>
2  0.003295 (0.0033)   bsdi.discard > solaris.32990: S 4208081409:4208081409(0)
                                                     ack 40140289 win 4096
                                                     <mss 1024>
3  0.419991 (0.4167)   solaris.32990 > bsdi.discard: P 1:257(256) ack 1 win 9216
4  0.449852 (0.0299)   solaris.32990 > bsdi.discard: F 257:257(0) ack 1 win 9216
5  0.451965 (0.0021)   bsdi.discard > solaris.32990: . ack 258 win 3840
6  0.464569 (0.0126)   bsdi.discard > solaris.32990: F 1:1(0) ack 258 win 4096
7  0.720031 (0.2555)   solaris.32990 > bsdi.discard: . ack 2 win 9216
```

18.10 为什么图18-4中的服务器不将对客户FIN的ACK与自己的FIN合并，从而将报文段数减少为3个？

18.11 在图18-16中，RST的序号为什么是26368002?

18.12 TCP向链路层查询MTU是否违反分层的规则？

18.13 假定在图14.16中，每个DNS使用TCP而不是UDP进行查询，试问需要交换多少个报文段？

18.14 假定MSL为120秒，试问系统能够初始化一个新连接然后进行主动关闭的最大速率是多少？

18.15 阅读RFC 793，分析处于TIME_WAIT状态的主机收到使其进入此状态的重复的 FIN时所发生的情况。

18.16 阅读RFC 793，分析处于TIME_WAIT状态的主机收到一个RST时所发生的情况。

18.17 阅读Host Requirements RFC并找出半双工TCP关闭的定义。

18.18 在图1-8中，我们曾提到到来的TCP报文段可根据其目的端口号进行分用，请问这种说法是否正确？

Linux公社 www.linuxidc.com

第19章　TCP的交互数据流

19.1　引言

前一章我们介绍了 TCP连接的建立与释放，现在来介绍使用 TCP进行数据传输的有关问题。

一些有关TCP通信量的研究如[Caceres et al. 1991]发现，如果按照分组数量计算，约有一半的TCP报文段包含成块数据（如 FTP、电子邮件和 Usenet新闻），另一半则包含交互数据（如Telnet和Rlogin）。如果按字节计算，则成块数据与交互数据的比例约为 90%和10%。这是因为成块数据的报文段基本上都是满长度（ full-sized）的（通常为512字节的用户数据），而交互数据则小得多（上述研究表明 Telnet和Rlogin分组中通常约90%左右的用户数据小于 10个字节）。

很明显，TCP需要同时处理这两类数据，但使用的处理算法则有所不同。本章将以 Rlogin应用为例来观察交互数据的传输过程。将揭示经受时延的确认是如何工作的以及 Nagle算法怎样减少了通过广域网络传输的小分组的数目，这些算法也同样适用于 Telnet应用。下一章我们将介绍成块数据的传输问题。

19.2　交互式输入

首先来观察在一个 Rlogin连接上键入一个交互命令时所产生的数据流。许多 TCP/IP的初学者很吃惊地发现通常每一个交互按键都会产生一个数据分组，也就是说，每次从客户传到服务器的是一个字节的按键（而不是每次一行）。而且，Rlogin需要远程系统（服务器）回显我们（客户）键入的字符。这样就会产生4个报文段：（1）来自客户的交互按键；（2）来自服务器的按键确认；（3）来自服务器的按键回显；（ 4）来自客户的按键回显确认。图 19-1表示了这个数据流。

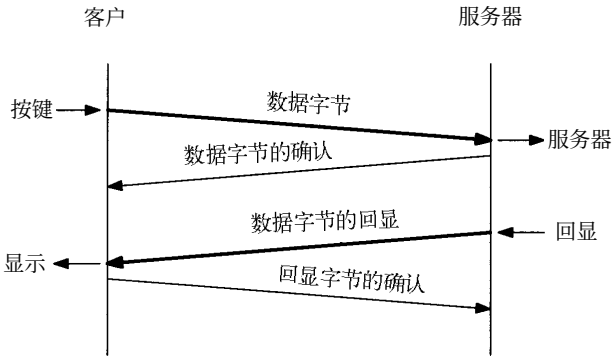


图19-1　一种可能的处理远程交互按键回显的方法

然而，我们一般可以将报文段 2和3进行合并 —— 按键确认与按键回显一起发送。下一节将描述这种合并的技术（称为经受时延的确认）。

本章我们特意使用 Rlogin作为例子，因为它每次总是从客户发送一个字节到服务器。在第 26章讲到Telnet的时候，将会发现它有一个选项允许客户发送一行到服务器，通过使用这个选项可以减少网络的负载。

图19-2显示的是当我们键入 5个字符`date\n`时的数据流（我们没有显示连接建立的过程，并且去掉了所有的服务类型输出。 BSD/386通过设置一个Rlogin连接的TOS来获得最小时延）。

Linux公社 www.linuxidc.com

第1行客户发送字符d到服务器。第2行是该字符的确认及回显（也就是图 19-1的中间两部分数据的合并）。第3行是回显字符的确认。与字符a有关的是第4~6行，与字符t有关的是第7~9行，第10~12行与字符e有关。第3~4、6~7、9~10和12~13行之间半秒左右的时间差是键入两个字符之间的时延。

注意到13~15行稍有不同。从客户发送到服务器的是一个字符（按下 RETURN键后产生的UNIX系统中的换行符），而回显的则是两个字符。这两个字符分别是回车和换行字符（CR/LF），它们的作用是将光标回移到左边并移动到下一行。

第16行是来自服务器的date命令的输出。这30个字节由28个字符与最后的CR/LF组成。紧接着从服务器发往客户的7个字符（第18行）是在服务器主机上的客户提示符： svr4 % 。第19行确认了这7个字符。

```
1  0.0                bsdi.1023 > svr4.login: P 0:1(1) ack 1 win 4096
2  0.016497 (0.0165)  svr4.login > bsdi.1023: P 1:2(1) ack 1 win 4096
3  0.139955 (0.1235)  bsdi.1023 > svr4.login: . ack 2 win 4096

4  0.458037 (0.3181)  bsdi.1023 > svr4.login: P 1:2(1) ack 2 win 4096
5  0.474386 (0.0163)  svr4.login > bsdi.1023: P 2:3(1) ack 2 win 4096
6  0.539943 (0.0656)  bsdi.1023 > svr4.login: . ack 3 win 4096

7  0.814582 (0.2746)  bsdi.1023 > svr4.login: P 2:3(1) ack 3 win 4096
8  0.831108 (0.0165)  svr4.login > bsdi.1023: P 3:4(1) ack 3 win 4096
9  0.940112 (0.1090)  bsdi.1023 > svr4.login: . ack 4 win 4096

10 1.191287 (0.2512)  bsdi.1023 > svr4.login: P 3:4(1) ack 4 win 4096
11 1.207701 (0.0164)  svr4.login > bsdi.1023: P 4:5(1) ack 4 win 4096
12 1.339994 (0.1323)  bsdi.1023 > svr4.login: . ack 5 win 4096

13 1.680646 (0.3407)  bsdi.1023 > svr4.login: P 4:5(1) ack 5 win 4096
14 1.697977 (0.0173)  svr4.login > bsdi.1023: P 5:7(2) ack 5 win 4096
15 1.739974 (0.0420)  bsdi.1023 > svr4.login: . ack 7 win 4096

16 1.799841 (0.0599)  svr4.login > bsdi.1023: P 7:37(30) ack 5 win 4096
17 1.940176 (0.1403)  bsdi.1023 > svr4.login: . ack 37 win 4096
18 1.944338 (0.0042)  svr4.login > bsdi.1023: P 37:44(7) ack 5 win 4096
19 2.140110 (0.1958)  bsdi.1023 > svr4.login: . ack 44 win 4096
```

图19-2 当在Rlogin连接上键入date时的数据流

注意TCP是怎样进行确认的。第 1行以序号0发送数据字节，第 2行通过将确认序号设为 1，也就是最后成功收到的字节的序号加 1，来对其进行确认（也就是所谓的下一个期望数据的序号）。在第2行中服务器还向客户发送了一序号为 1的数据，客户在第 3行中通过设置确认序号为2来对该数据进行确认。

19.3 经受时延的确认

在图19-2中有一些与本节将要论及的时间有关的细微之处。图 19-3表示了图19-2中数据交换的时间系列（在该时间系列中，去掉了所有的窗口通告，并增加了一个记号来表明正在传输何种数据）。

把从bsdi发送到srv4的7个ACK标记为经受时延的ACK。通常TCP在接收到数据时并不立即发送ACK；相反，它推迟发送，以便将 ACK与需要沿该方向发送的数据一起发送（有时称这种现象为数据捎带 ACK）。绝大多数实现采用的时延为 200 ms，也就是说， TCP将以最大 200 ms的时延等待是否有数据一起发送。

如果观察bsdi接收到数据和发送ACK之间的时间差，就会发现它们似乎是随机的： 123.5、

Linux公社 www.linuxidc.com

65.6、109.0、132.2、42.0、140.3和195.8 ms。相反，观察到发送ACK的实际时间（从0开始）为：139.9、539.3、940.1、1339.9、1739.9、1940.1和2140.1 ms（在图19-3中用星号标出）。这些时间之间的差则是200 ms的整数倍，这里所发生的情况是因为TCP使用了一个200 ms的定时器，该定时器以相对于内核引导的200 ms固定时间溢出。由于将要确认的数据是随机到达的（在时刻16.4, 474.3, 831.1等），TCP在内核的200 ms定时器的下一次溢出时得到通知。这有可能是将来1~200 ms中的任何一刻。

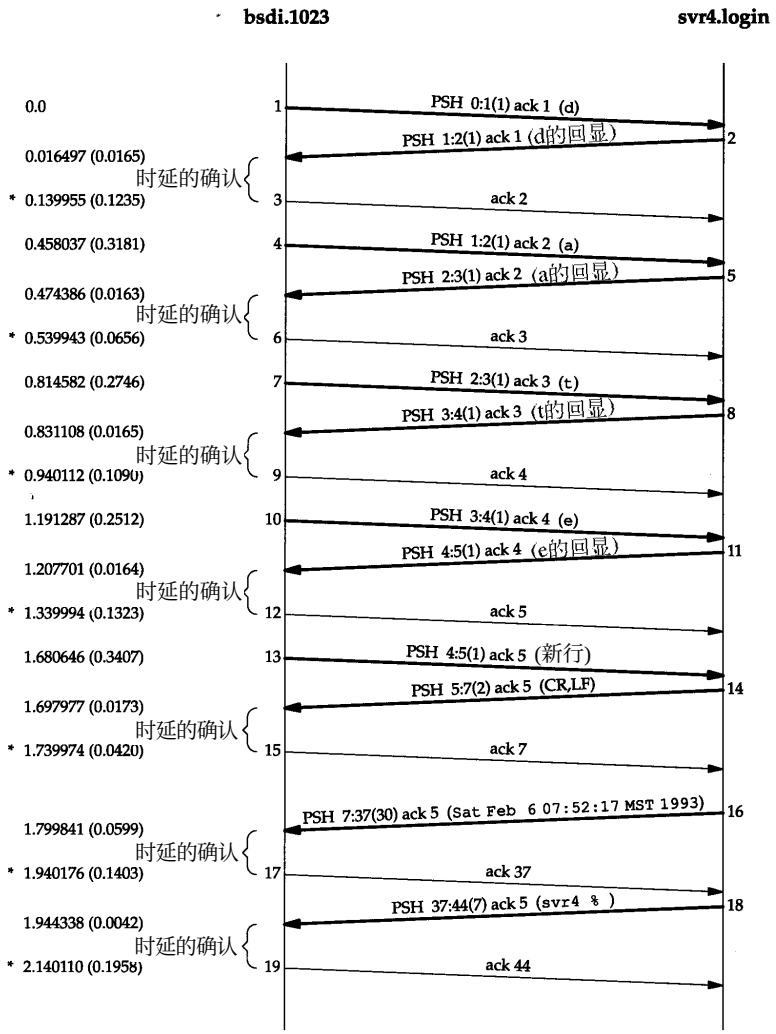


图19-3　在rlogin连接上键入date命令时的数据流时间系列

如果观察svr4为产生所收到的每个字符的回显所使用的时间，则这些时间分别为 16.5、16.3、16.5、16.4和17.3 ms。由于这个时间小于200 ms，因此我们在另一端从来没有观察到一个经受时延的ACK。在经受时延的定时器溢出前总是有数据需要发送（如果有一个约为16 ms等待时间越过了内核的200 ms时钟滴答的边界，则仍可以看到一个经受时延的ACK。在本例中我们一个也没有看到）。

在图18-7中，当为检测超时而使用500 ms的TCP定时器时，我们会看到同样的情况。这两

Linux公社 www.linuxidc.com

个200 ms和500 ms的定时器都在相对于内核引导的时间处溢出。不论 TCP何时设置一个定时器，该定时器都可能在将来1~200 ms和1~500 ms的任一处溢出。

Host Requirements RFC声明TCP需要实现一个经受时延的ACK，但时延必须小于500 ms。

19.4 Nagle算法

在前一节我们看到，在一个Rlogin连接上客户一般每次发送一个字节到服务器，这就产生了一些41字节长的分组：20字节的IP首部、20字节的TCP首部和1个字节的数据。在局域网上，这些小分组（被称为微小分组（tinygram））通常不会引起麻烦，因为局域网一般不会出现拥塞。但在广域网上，这些小分组则会增加拥塞出现的可能。一种简单和好的方法就是采用RFC 896 [Nagle 1984]中所建议的Nagle算法。

该算法要求一个TCP连接上最多只能有一个未被确认的未完成的小分组，在该分组的确认到达之前不能发送其他的小分组。相反， TCP收集这些少量的分组，并在确认到来时以一个分组的方式发出去。该算法的优越之处在于它是自适应的：确认到达得越快，数据也就发送得越快。而在希望减少微小分组数目的低速广域网上，则会发送更少的分组（我们将在22.3节看到“小”的含义是小于报文段的大小）。

在图19-3中可以看到，在以太网上一个字节被发送、确认和回显的平均往返时间约为 16 ms。为了产生比这个速度更快的数据，我们每秒键入的字符必须多于 60个。这表明在局域网环境下两个主机之间发送数据时很少使用这个算法。

但是，当往返时间（RTT）增加时，如通过一个广域网，情况就会发生变化。看一下在主机`slip`和主机`vangogh.cs.berkeley.edu`之间的Rlogin连接工作的情况。为了从我们的网络中出去（参看原书封面内侧），需要使用两个SLIP链路和Internet。我们希望获得更长的往返时间。图19-4显示了当在客户端快速键入字符（像一个快速打字员一样）时一些数据流的时间系列（去掉了服务类型信息，但保留了窗口通告）。

比较图19-4与图19-3，我们首先注意到从`slip`到`vangogh`不存在经受时延的ACK。这是因为在时延定时器溢出之前总是有数据等待发送。

其次，注意到从左到右待发数据的长度是不同的，分别为：1、1、2、1、2、2、3、1和3个字节。这是因为客户只有收到前一个数据的确认后才发送已经收集的数据。通过使用 Nagle算法，为发送16个字节的数据客户只需要使用9个报文段，而不再是16个。

报文段14和15看起来似乎是与Nagle算法相违背的，但我们需要通过检查序号来观察其中的真相。因为确认序号是54，因此报文段14是报文段12中确认的应答。但客户在发送该报文段之前，接收到了来自服务器的报文段13，报文段15中包含了对序号为56的报文段13的确认。因此即使我们看到从客户到服务器有两个连续返回的报文段，客户也是遵守了 Nagle算法的。

在图19-4中可以看到存在一个经受时延的 ACK，但该ACK是从服务器到客户的（报文段12），因为它不包含任何数据，因此我们可以假定这是经受时延的 ACK。服务器当时一定非常忙，因此无法在服务器的定时器溢出前及时处理所收到的字符。

最后看一下最后两个报文段中数据的数量以及相应的序号。客户发送 3个字节的数据（18, 19和20），然后服务器确认这3个字节（最后的报文段中的 ACK 21），但是只返回了一个字节（标号为59）。这是因为当服务器的TCP一旦正确收到这3个字节的数据，就会返回对该数据的确

Linux公社 www.linuxidc.com

认，但只有当Rlogin服务器发送回显数据时，它才能够发送这些数据的回显。这表明 TCP可以在应用读取并处理数据前发送所接收数据的确认。TCP确认仅仅表明TCP已经正确接收了数据。最后一个报文段的窗口大小为8189而非8192，表明服务器进程尚未读取这三个收到的数据。

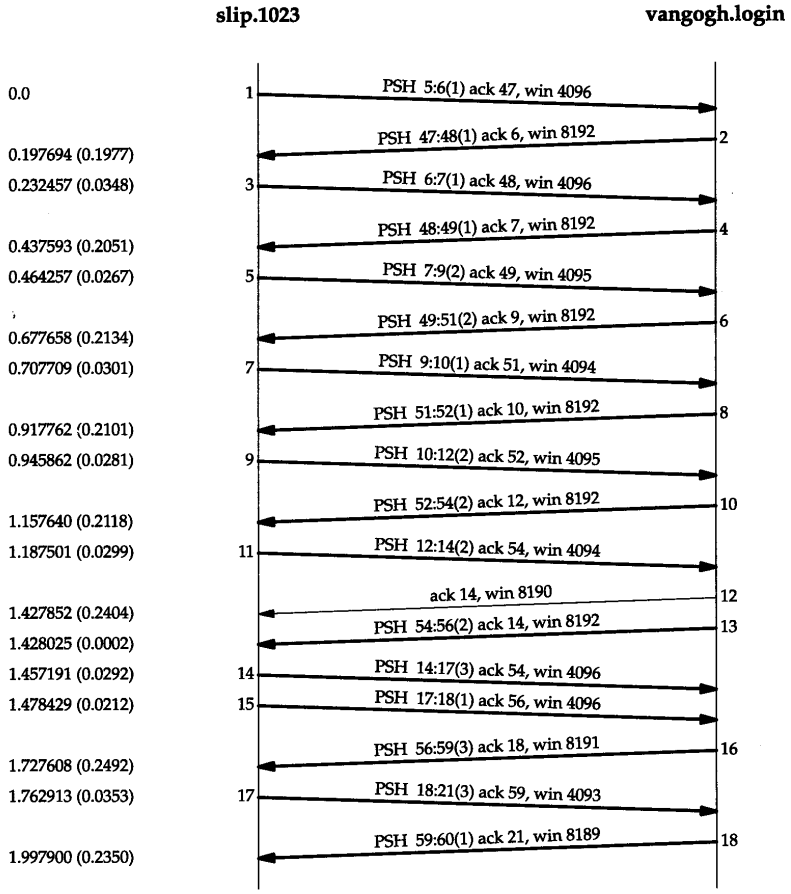


图19-4 在slip 和vangogh.cs.berkeley.edu 之间使用rlogin 时的数据流

19.4.1 关闭Nagle算法

有时我们也需要关闭Nagle算法。一个典型的例子是X窗口系统服务器（见30.5节）：小消息（鼠标移动）必须无时延地发送，以便为进行某种操作的交互用户提供实时的反馈。

这里将举另外一个更容易说明的例子——在一个交互注册过程中键入终端的一个特殊功能键。这个功能键通常可以产生多个字符序列，经常从 ASCII码的转义(escape)字符开始。如果TCP每次得到一个字符，它很可能会发送序列中的第一个字符（ ASCII码的ESC），然后缓存其他字符并等待对该字符的确认。但当服务器接收到该字符后，它并不发送确认，而是继续等待接收序列中的其他字符。这就会经常触发服务器的经受时延的确认算法，表示剩下的字符没有在200 ms内发送。对交互用户而言，这将产生明显的时延。

插口API用户可以使用TCP_NODELAY选项来关闭Nagle算法。

Host Requirements RFC声明TCP必须实现Nagle算法，但必须为应用提供一种方法来关闭该算法在某个连接上执行。

Linux公社 www.linuxidc.com

19.4.2 一个例子

可以在Nagle算法和产生多个字符的按键之间看到这种交互的情况。在主机 `slip`和主机`vangogh.cs.berkeley.edu`之间建立一个Rlogin连接，然后按下F1功能键，这将产生3个字节：一个escape、一个左括号和一个M。然后再按下F2功能键，这将产生另外3个字节。图19-5表示的是`tcpdump`的输出结果（我们去掉了其中的服务类型和窗口通告）。

```
                                按F1键
 1  0.0                         slip.1023 > vangogh.login: P 1:2(1) ack 2
 2  0.250520 (0.2505)           vangogh.login > slip.1023: P 2:4(2) ack 2
 3  0.251709 (0.0012)           slip.1023 > vangogh.login: P 2:4(2) ack 4
 4  0.490344 (0.2386)           vangogh.login > slip.1023: P 4:6(2) ack 4
 5  0.588694 (0.0984)           slip.1023 > vangogh.login: . ack 6
                                按F2键
 6  2.836830 (2.2481)           slip.1023 > vangogh.login: P 4:5(1) ack 6
 7  3.132388 (0.2956)           vangogh.login > slip.1023: P 6:8(2) ack 5
 8  3.133573 (0.0012)           slip.1023 > vangogh.login: P 5:7(2) ack 8
 9  3.370346 (0.2368)           vangogh.login > slip.1023: P 8:10(2) ack 7
10  3.388692 (0.0183)           slip.1023 > vangogh.login: . ack 10
```

图19-5 当键入能够产生多个字节数据的字符时Nagle算法的观察情况

图19-6表示了这个交互过程的时间系列。在该图的下面部分我们给出了从客户发送到服务器的6个字节和它们的序号以及将要返回的8个字节的回显。

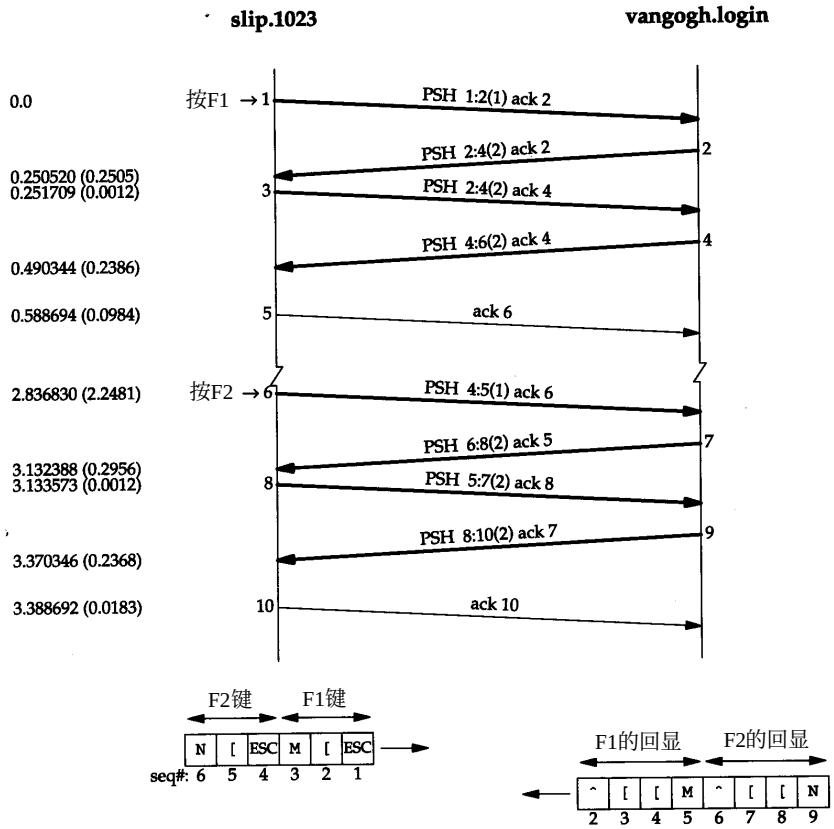


图19-6 图19-5的时间系列（Nagle算法的观察结果）

Linux公社 www.linuxidc.com

当rlogin客户读取到输入的第1个字节并向TCP写入时，该字节作为报文段1被发送。这是F1键所产生的3个字节中的第1个。它的回显在报文段2中被返回，此时剩余的2个字节才被发送（报文段3）。这两个字节的回显在报文段4被接收，而报文段5则是对它们的确认。

第1个字节的回显为2个字节（报文段2）的原因是因为在ASCII码中转义符的回显是2个字节：插入记号和一个左括号。剩下的两个输入字节：一个左括号和一个 M，分别以自身作为回显内容。

当按下下一个特殊功能键（报文段 6~10）时，也会发生同样的过程。正如我们希望的那样，在报文段5和10（slip发送回显的确认）之间的时间差是 200 ms的整数倍，因为这两个ACK被进行时延。

现在我们使用一个修改后关闭了 Nagle算法的rlogin版本重复同样的实验。图 19-7显示了tcpdump的输出结果（同样去掉了其中的服务类型和窗口通告）。

```
                                  按F1键
 1  0.0                           slip.1023 > vangogh.login: P 1:2(1) ack 2
 2  0.002163 (0.0022)             slip.1023 > vangogh.login: P 2:3(1) ack 2
 3  0.004218 (0.0021)             slip.1023 > vangogh.login: P 3:4(1) ack 2
 4  0.280621 (0.2764)             vangogh.login > slip.1023: P 5:6(1) ack 4
 5  0.281738 (0.0011)             slip.1023 > vangogh.login: . ack 2
 6  2.477561 (2.1958)             vangogh.login > slip.1023: P 2:6(4) ack 4
 7  2.478735 (0.0012)             slip.1023 > vangogh.login: . ack 6
                                  按F2键
 8  3.217023 (0.7383)             slip.1023 > vangogh.login: P 4:5(1) ack 6
 9  3.219165 (0.0021)             slip.1023 > vangogh.login: P 5:6(1) ack 6
10  3.221688 (0.0025)             slip.1023 > vangogh.login: P 6:7(1) ack 6
11  3.460626 (0.2389)             vangogh.login > slip.1023: P 6:8(2) ack 5
12  3.489414 (0.0288)             vangogh.login > slip.1023: P 8:10(2) ack 7
13  3.640356 (0.1509)             slip.1023 > vangogh.login: . ack 10
```

图19-7 在一个Rlogin会话中关闭Nagle算法

在已知某些报文段在网络上形成交叉的情况下，以该结果构造时间系列则更具有启发性和指导意义。这个例子同样也需要随着数据流对序号进行仔细的检查。在图 19-8中显示这个结果。用图19-7中tcpdump输出的号码对报文段进行了相应的编号。

我们注意到的第1个变化是当3个字节准备好时它们全部被发送（报文段 1、2和3）。没有时延发生——Nagle算法被禁止。

在tcpdump输出中的下一个分组（报文段 4）中带有来自服务器的第 5个字节及一个确认序号为4的ACK。这是不正确的，因为客户并不希望接收到第 5个字节，因此它立即发送一个确认序号为2而不是6的响应（没有被延迟）。看起来一个报文段丢失了，在图 19-8中我们用虚线表示。

如何知道这个丢失的报文段中包含第 2、3和4个字节，且其确认序号为 3呢？这是因为正如在报文段5中声明的那样，我们希望的下一个字节是第 2个字节（每当TCP接收到一个超出期望序号的失序数据时，它总是发送一个确认序号为其期望序号的确认）。也正是因为丢失的分组中包含第2、3和4个字节，表明服务器必定已经接收到报文段 2，因此丢失的报文段中的确认序号一定为3（服务器期望接收的下一个字节号）。最后，注意到重传的报文段6中包含有丢失的报文段中的数据和报文段 4，这被称为重新分组化。我们将在 22.11节对其进行更多的介绍。

Linux公社 www.linuxidc.com

现在回到禁止Nagle算法的讨论中来。可以观察到键入的下一个特殊功能键所产生的 3个字节分别作为单独的报文段（报文段8、9和10）被发送。这一次服务器首先回显了报文段8中的字节（报文段11），然后回显了报文段9和10中的字节（报文段12）。

在这个例子中，我们能够观察到的是在跨广域网运行一个交互应用的环境下，当进行多字节的按键输入时，默认使用Nagle算法会引起额外的时延。

在第21章我们将进行有关时延和重传方面的讨论。

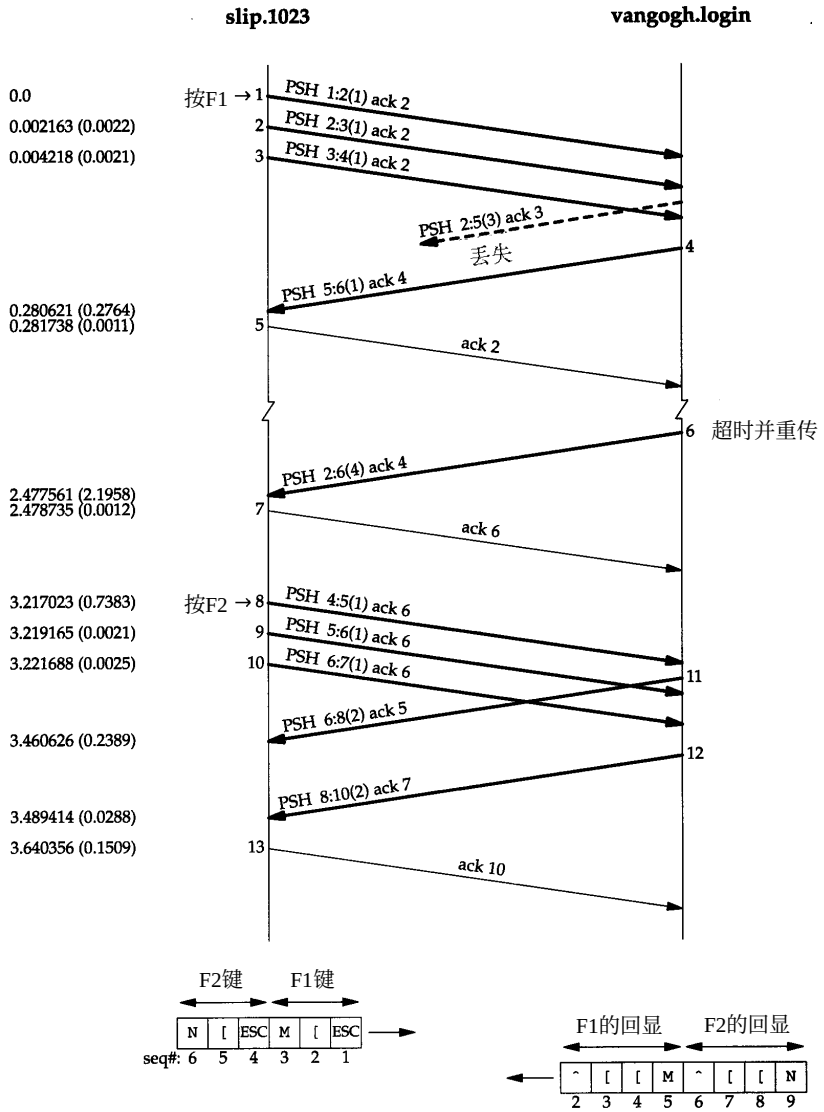


图19-8 图19-7的时间系列（关闭Nagle算法）

19.5 窗口大小通告

在图19-4中，我们可以观察到slip通告窗口大小为4096字节，而vangogh通告其窗口大小为8192个字节。该图中的大多数报文段都包含这两个值中的一个。

Linux公社 www.linuxidc.com

然而，报文段5通告的窗口大小为4095个字节，这意味着在TCP的缓冲区中仍然有一个字节等待应用程序（Rlogin客户）读取。同样，来自客户的下一个报文段声明其窗口大小为4094个字节，这说明仍有两个字节等待读取。

服务器通常通告窗口大小为8192个字节，这是因为服务器在读取并回显接收到的数据之前，其TCP没有数据发送。当服务器已经读取了来自客户的输入后，来自服务器的数据将被发送。

然而，在ACK到来时，客户的TCP总是有数据需要发送。这是因为它在等待ACK的过程中缓存接收到的字符。当客户TCP发送缓存的数据时，Rlogin客户没有机会读取来自服务器的数据，因此，客户通告的窗口大小总是小于4096。

19.6 小结

交互数据总是以小于最大报文段长度的分组发送。在Rlogin中通常只有一个字节从客户发送到服务器。Telnet允许一次发送一行输入数据，但是目前大多数实现仍然发送一个字节。

对于这些小的报文段，接收方使用经受时延的确认方法来判断确认是否可被推迟发送，以便与回送数据一起发送。这样通常会减少报文段的数目，尤其是对于需要回显用户输入字符的Rlogin会话。

在较慢的广域网环境中，通常使用Nagle算法来减少这些小报文段的数目。这个算法限制发送者任何时候只能有一个发送的小报文段未被确认。但我们给出的一个例子也表明有时需要禁止Nagle算法的功能。

习题

19.1 考虑一个TCP客户应用程序，它发送一个小应用程序首部（8个字节）和一个小请求（12个字节），然后等待来自服务器的一个应答。比较以下两种方式发送请求时的处理情况：先发送8个字节再发送12个字节和一次发送20个字节。

19.2 图19-4中我们在路由器sun上运行tcpdump。这意味着从右至左的箭头中的数据也需要经过bsdi，同时从左至右的箭头中的数据已经流经bsdi。当观察一个送往slip的报文段及下一个来自slip的报文段时，我们发现它们之间的时间差分别为：34.8、26.7、30.1、28.1、29.9和35.3 ms。现给定在sun和slip之间存在两条链路（一个以太链路和一个9600 b/s的CSLIP链路），试问这些时间差的含义（提示：重新阅读2.10节）。

19.3 比较在使用Nagle算法（图19-6）和禁止Nagle算法（图19-8）的情况下发送一个特殊功能键并等待其应答所需要的时间。

Linux公社 www.linuxidc.com

第20章　TCP的成块数据流

20.1　引言

在第15章我们看到TFTP使用了停止等待协议。数据发送方在发送下一个数据块之前需要等待接收对已发送数据的确认。本章我们将介绍 TCP所使用的被称为滑动窗口协议的另一种形式的流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。

我们还将介绍TCP的PUSH标志，该标志在前面的许多例子中都出现过。此外，我们还要介绍慢启动，TCP使用该技术在一个连接上建立数据流，最后介绍成块数据流的吞吐量。

20.2　正常数据流

我们以从主机svr4单向传输8192个字节到主机bsdi开始。在bsdi上运行sock程序作为服务器：

```
bsdi %sock -i -s 7777
```

其中，标志-i和-s指示程序作为一个“吸收（sink）”服务器运行（从网络上读取并丢弃数据），服务器端口指明为7777。相应的客户程序运行为：

```
svr4 %sock -i -n8 bsdi 7777
```

该命令指示客户向网络发送8个1024字节的数据。图20-1显示了这个过程的时间系列。我们在输出的前3个报文段中显示了每一端MSS的值。

发送方首先传送3个数据报文段（4~6）。下一个报文段（7）仅确认了前两个数据报文段，这可以从其确认序号为2048而不是3073看出来。

报文段7的ACK的序号之所以是2048而不是3073是由以下原因造成的：当一个分组到达时，它首先被设备中断例程进行处理，然后放置到 IP的输入队列中。三个报文段4、5和6依次到达并按接收顺序放到IP的输入队列。IP将按同样顺序将它们交给 TCP。当TCP处理报文段4时，该连接被标记为产生一个经受时延的确认。TCP处理下一报文段（5），由于TCP现在有两个未完成的报文段需要确认，因此产生一个序号为 2048的ACK（报文段7），并清除该连接产生经受时延的确认标志。TCP处理下一个报文段（6），而连接又被标志为产生一个经受时延的确认。在报文段9到来之前，由于时延定时器溢出，因此产生一个序号为3073的ACK（报文段8）。报文段8中的窗口大小为3072，表明在TCP的接收缓存中还有1024个字节的数据等待被应用程序读取。

报文段11~16说明了通常使用的“隔一个报文段确认”的策略。报文段 11、12和13到达并被放入IP的接收队列。当报文段11被处理时，连接被标记为产生一个经受时延的确认。当报文段12被处理时，它们的ACK（报文段14）被产生且连接的经受时延的确认标志被清除。报文段13使得连接再次被标记为产生经受时延。但在时延定时器溢出之前，报文段 15处理完毕，因此该确认立刻被发送。

Linux公社 www.linuxidc.com

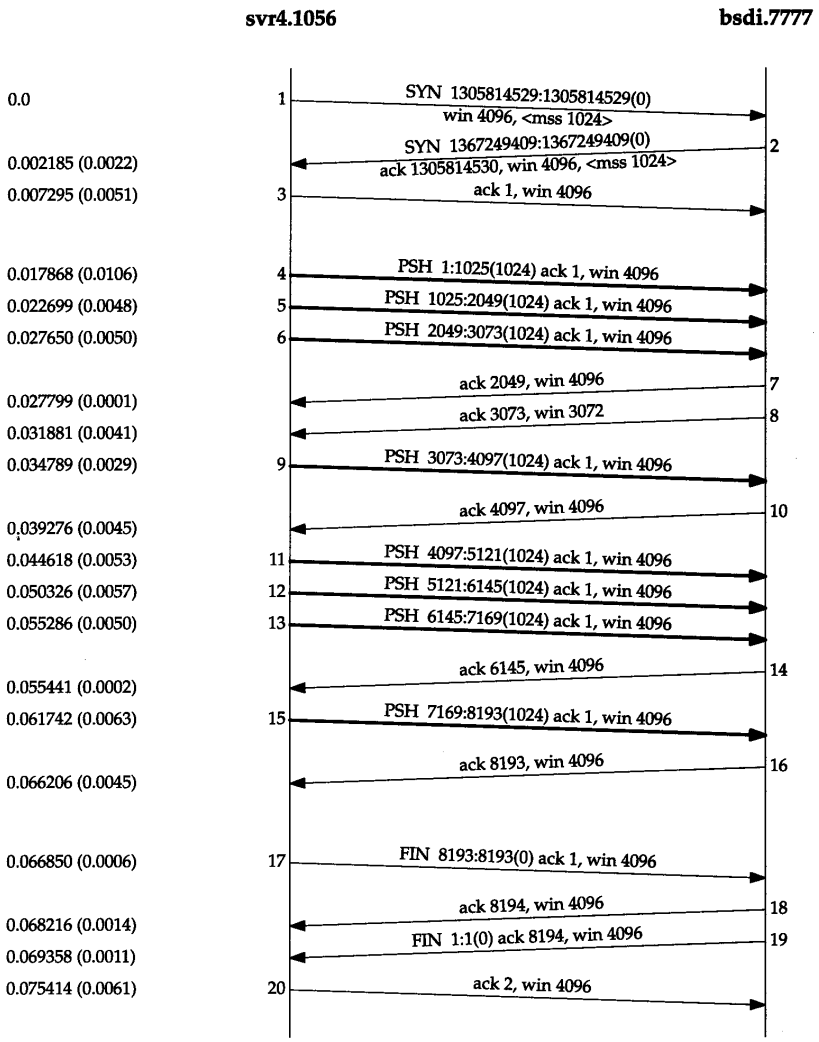


图20-1 从svr4 传输8192个字节到bsdi

注意到报文段7、14和16中的ACK确认了两个收到的报文段是很重要的。使用 TCP的滑动窗口协议时，接收方不必确认每一个收到的分组。在 TCP中，ACK是累积的——它们表示接收方已经正确收到了一直到确认序号减 1的所有字节。在本例中，三个确认的数据为 2048字节而两个确认的数据为1024字节（忽略了连接建立和终止中的确认）。

用tcpdump看到的是TCP的动态活动情况。我们在线路上看到的分组顺序依赖于许多无法控制的因素：发送方TCP的实现、接收方TCP的实现、接收进程读取数据（依赖于操作系统的调度）和网络的动态性（如以太网的冲突和退避等）。对这两个TCP而言，没有一种单一的、正确的方法来交换给定数量的数据。

为显示情况可能怎样变化，图 20-2显示了在同样两个主机之间交换同样数据时的另一个时间系列，它们是在图20-1所示的几分钟之后截获的。

一些情况发生了变化。这一次接收方没有发送一个序号为 3073的ACK，而是等待并发送序号为4097的ACK。接收方仅发送了 4个ACK（报文段7、10、12和15）：三个确认了 2048字

Linux公社 www.linuxidc.com

节，另一个确认了1024字节。最后1024字节数据的ACK出现在报文段17中，它与FIN的ACK一道发送（比较该图中的报文段17与图20-1中的报文段16和18）。

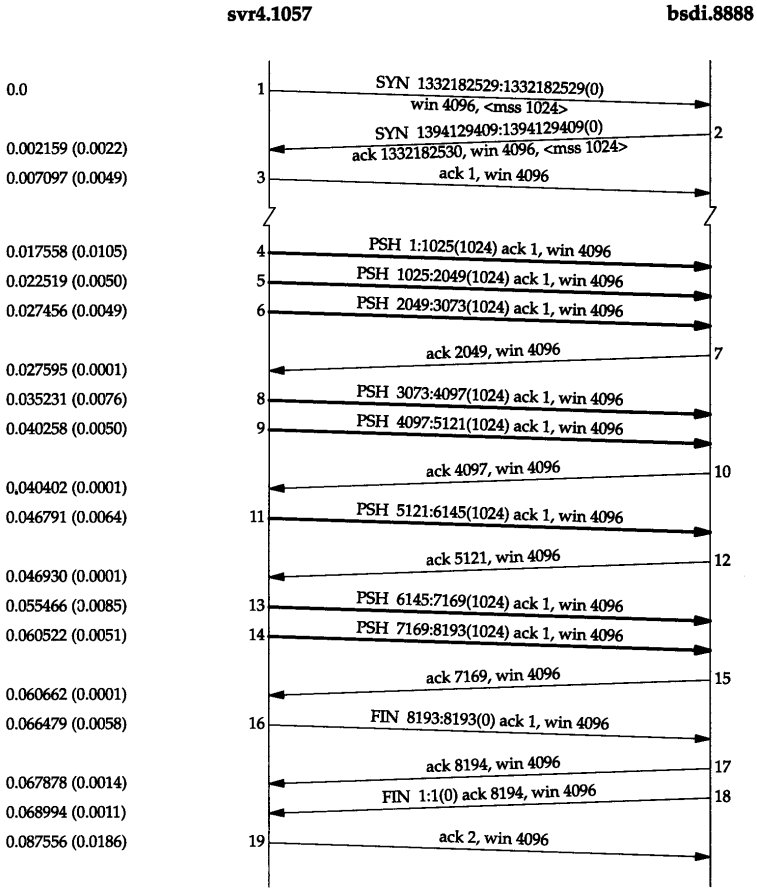


图20-2 从svr4 到bsdi 的另外8192字节数据的传输过程

快的发送方和慢的接收方

图20-3显示了另外一个时间系列。这次是从一个快的发送方（一个 Sparc工作站）到一个慢的接收方（配有慢速以太网卡的80386机器）。它的动态活动情况又有所不同。

发送方发送4个背靠背（back-to-back）的数据报文段去填充接收方的窗口，然后停下来等待一个ACK。接收方发送ACK（报文段8），但通告其窗口大小为0，这说明接收方已收到所有数据，但这些数据都在接收方的 TCP缓冲区，因为应用程序还没有机会读取这些数据。另一个ACK（称为窗口更新）在17.4 ms后发送，表明接收方现在可以接收另外的4096个字节的数据。虽然这看起来像一个ACK，但由于它并不确认任何新数据，只是用来增加窗口的右边沿，因此被称为窗口更新。

发送方发送最后4个报文段（10~13），再次填充了接收方的窗口。注意到报文段13中包括两个比特标志：PUSH和FIN。随后从接收方传来另外两个ACK，它们确认了最后的4096字节的数据（从4097到8192字节）和FIN（标号为8192）。

Linux公社 www.linuxidc.com

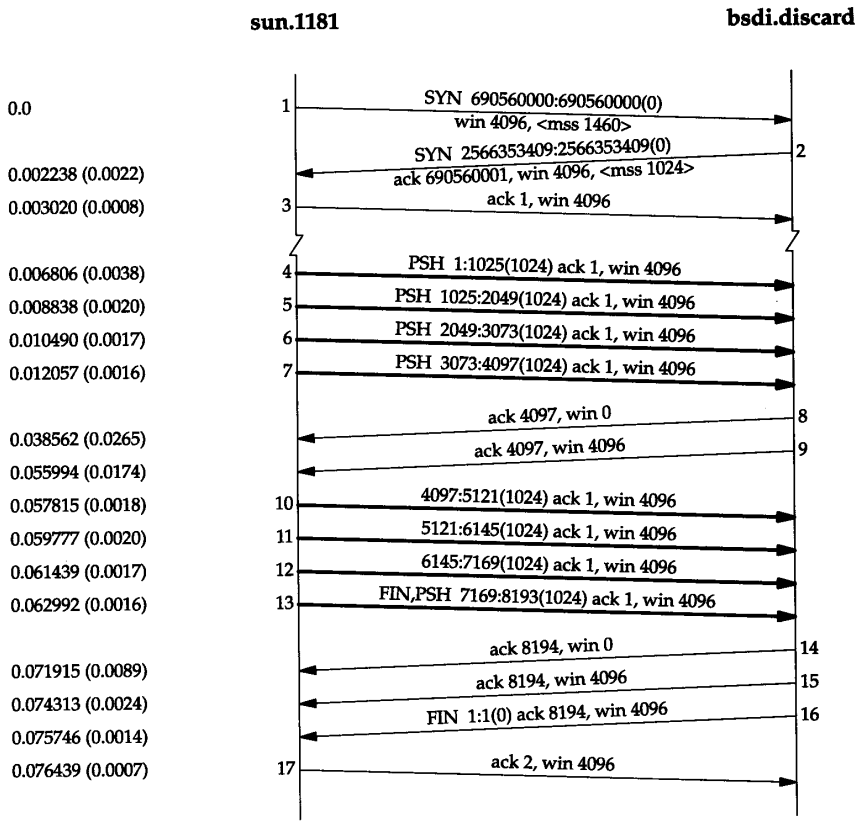


图20-3 从一个快发送方发送8192字节的数据到一个慢接收方

20.3 滑动窗口

图20-4用可视化的方法显示了我们在前一节观察到的滑动窗口协议。

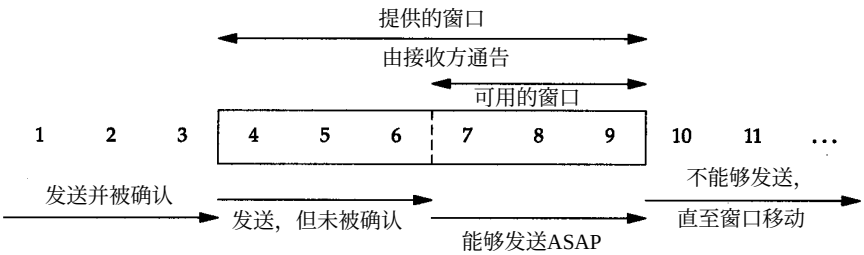


图20-4 TCP滑动窗口的可视化表示

在这个图中，我们将字节从1至11进行标号。接收方通告的窗口称为提出的窗口（offered window），它覆盖了从第4字节到第9字节的区域，表明接收方已经确认了包括第3字节在内的数据，且通告窗口大小为6。回顾第17章，我们知道窗口大小是与确认序号相对应的。发送方计算它的可用窗口，该窗口表明多少数据可以立即被发送。

当接收方确认数据后，这个滑动窗口不时地向右移动。窗口两个边沿的相对运动增加或减少了窗口的大小。我们使用三个术语来描述窗口左右边沿的运动：

Linux公社 www.linuxidc.com

1) 称窗口左边沿向右边沿靠近为窗口合拢。这种现象发生在数据被发送和确认时。

2) 当窗口右边沿向右移动时将允许发送更多的数据，我们称之为窗口张开。这种现象发生在另一端的接收进程读取已经确认的数据并释放了 TCP的接收缓存时。

3) 当右边沿向左移动时，我们称之为窗口收缩。 Host Requirements RFC强烈建议不要使用这种方式。但TCP必须能够在某一端产生这种情况时进行处理。第 22.3节给出了这样的一个例子，一端希望向左移动右边沿来收缩窗口，但没能够这样做。

图20-5表示了这三种情况。因为窗口的左边沿受另一端发送的确认序号的控制，因此不可能向左边移动。如果接收到一个指示窗口左边沿向左移动的 ACK，则它被认为是一个重复 ACK，并被丢弃。

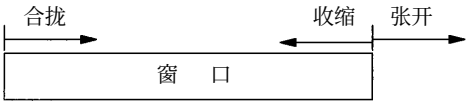


图20-5 窗口边沿的移动

如果左边沿到达右边沿，则称其为一个零窗口，此时发送方不能够发送任何数据。

一个例子

图20-6显示了在图20-1所示的数据传输过程中滑动窗口协议的动态性。

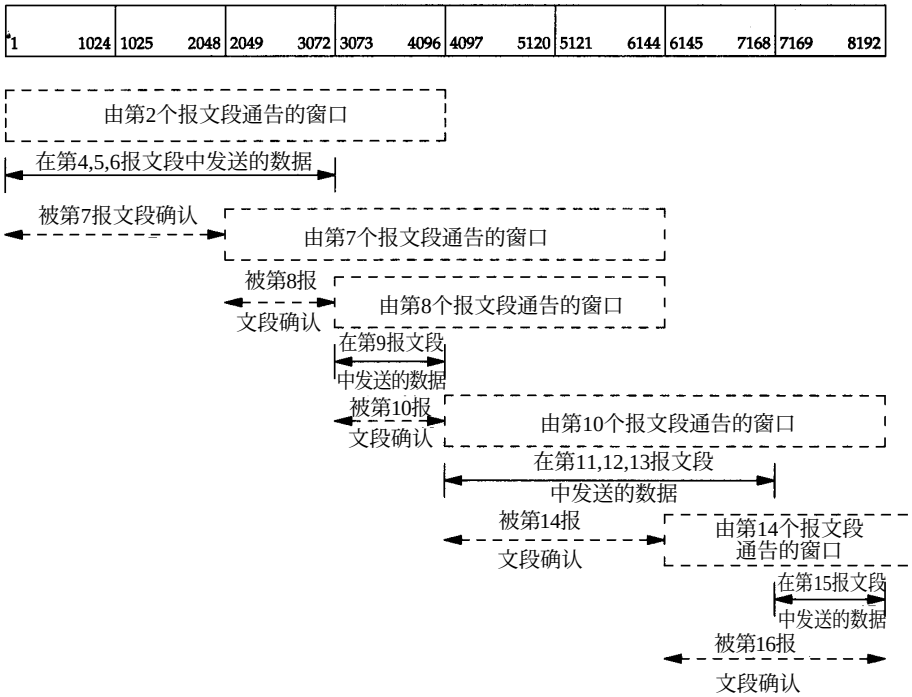


图20-6 图20-1的滑动窗口协议

以该图为例可以总结如下几点：

1) 发送方不必发送一个全窗口大小的数据。

2) 来自接收方的一个报文段确认数据并把窗口向右边滑动。这是因为窗口的大小是相对于确认序号的。

Linux公社 www.linuxidc.com

3) 正如从报文段7到报文段8中变化的那样，窗口的大小可以减小，但是窗口的右边沿却不能够向左移动。

4) 接收方在发送一个ACK前不必等待窗口被填满。在前面我们看到许多实现每收到两个报文段就会发送一个ACK。

下面我们可以看到更多的滑动窗口协议动态变化的例子。

20.4 窗口大小

由接收方提供的窗口的大小通常可以由接收进程控制，这将影响 TCP的性能。

> 4.2BSD默认设置发送和接受缓冲区的大小为2048个字节。在4.3BSD中双方被增加为4096个字节。正如我们在本书中迄今为止所看到的例子一样， SunOS 4.1.3、BSD/386和SVR4仍然使用4096字节的默认大小。其他的系统，如Solaris 2.2、4.4BSD和AIX3.2则使用更大的默认缓存大小，如8192或16384等。
>
> 插口API允许进程设置发送和接收缓存的大小。接收缓存的大小是该连接上所能够通告的最大窗口大小。有一些应用程序通过修改插口缓存大小来增加性能。

[Mogul 1993]显示了在改变发送和接收缓存大小（在单向数据流的应用中，如文件传输，只需改变发送方的发送缓存和接收方的接收缓存大小）的情况下，位于以太网上的两个工作站之间进行文件传输时的一些结果。它表明对以太网而言，默认的 4096字节并不是最理想的大小，将两个缓存增加到 16384个字节可以增加约 40%左右的吞吐量。在 [Papadopoulos和Parulkar 1993]中也有相似的结果。

在20.7节中，我们将看到在给定通信媒体带宽和两端往返时间的情况下，如何计算最小的缓存大小。

一个例子

可以使用sock程序来控制这些缓存的大小。我们以如下方式调用服务器程序：

```
bsdi %sock -i -s -R6144 5555
```

该命令设置接收缓存为6144个字节（-R选项）。接着我们在主机sun上启动客户程序并使之发送8192个字节的数据：

```
sun %sock -i -n1 -w8192 bsdi 5555
```

图20-7显示了结果。

首先注意到的是在报文段2中提供的窗口大小为6144字节。由于这是一个较大的窗口，因此客户立即连续发送了6个报文段（4~9），然后停止。报文段10确认了所有的数据（从第1到6144字节），但提供的窗口大小却为2048，这很可能是接收程序没有机会读取多于 2048字节的数据。报文段11和12完成了客户的数据传输，且最后一个报文段带有 FIN标志。

报文段13包含与报文段10相同的确认序号，但通告了一个更大的窗口大小。报文段 14确认了最后的2048字节的数据和FIN，报文段15和16仅用于通告一个更大的窗口大小。报文段17和18完成通常的关闭过程。

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

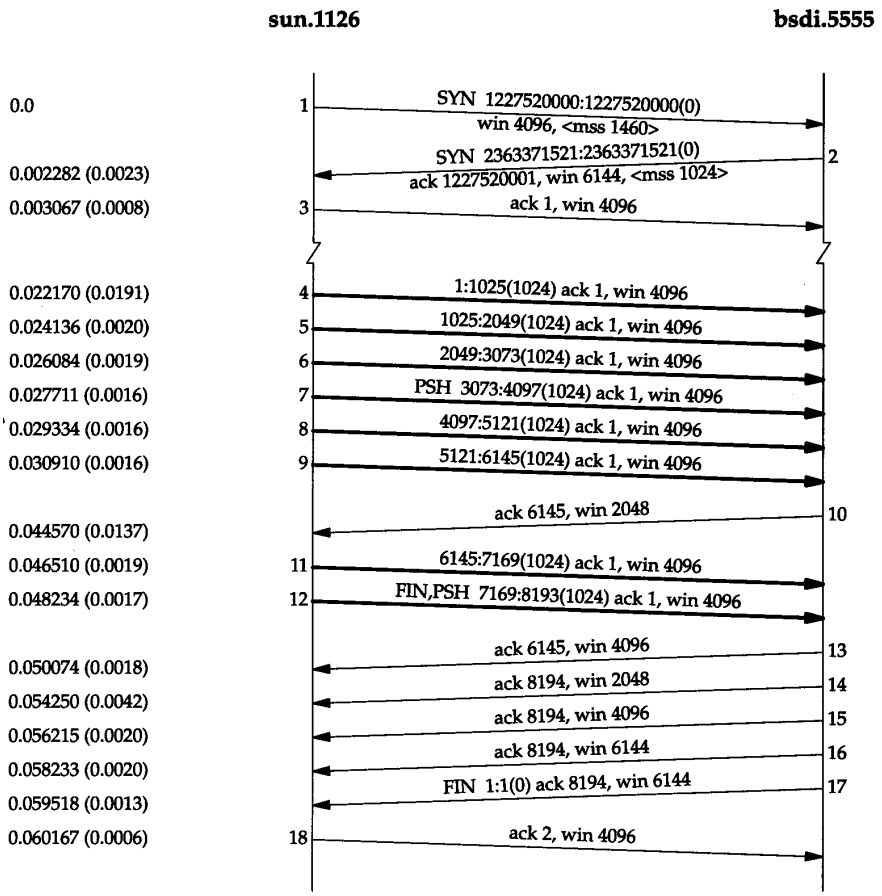


图20-7 接收方提供一个6144字节的接收窗口的情况下的数据传输

20.5 PUSH标志

在每一个TCP例子中，我们都看到了PUSH标志，但一直没有介绍它的用途。发送方使用该标志通知接收方将所收到的数据全部提交给接收进程。这里的数据包括与 PUSH一起传送的数据以及接收方TCP已经为接收进程收到的其他数据。

在最初的TCP规范中，一般假定编程接口允许发送进程告诉它的 TCP何时设置PUSH标志。例如，在一个交互程序中，当客户发送一个命令给服务器时，它设置 PUSH标志并停下来等待服务器的响应（在习题 19.1中我们假定当发送 12字节的请求时客户设置 PUSH标志）。通过允许客户应用程序通知其TCP设置PUSH标志，客户进程通知TCP在向服务器发送一个报文段时不要因等待额外数据而使已提交数据在缓存中滞留。类似地，当服务器的 TCP接收到一个设置了PUSH标志的报文段时，它需要立即将这些数据递交给服务器进程而不能等待判断是否还会有额外的数据到达。

然而，目前大多数的 API没有向应用程序提供通知其 TCP设置PUSH标志的方法。的确，许多实现程序认为PUSH标志已经过时，一个好的TCP实现能够自行决定何时设置这个标志。

如果待发送数据将清空发送缓存，则大多数的源于伯克利的实现能够自动设置 PUSH标志。这意味着我们能够观察到每个应用程序写的数据均被设置了 PUSH标志，因为数据在写的时候

Linux公社 www.linuxidc.com

就立即被发送。

代码中的注释表明该算法对那些只有在缓存被填满或收到一个PUSH标志时才向应用程序提交数据的TCP实现有效。

使用插口API通知TCP设置正在接收数据的PUSH标志或得到该数据是否被设置PUSH标志的信息是不可能的。

由于源于伯克利的实现一般从不将接收到的数据推迟交付给应用程序，因此它们忽略所接收的PUSH标志。

举例

在图20-1中我们观察到所有8个数据报文段（4~6、9、11~13和15）的PUSH标志均被置1，这是因为客户进行了8次1024字节数据的写操作，并且每次写操作均清空了发送缓存。

再次观察图20-7，我们预计报文段12中的PUSH标志被置1，因为它是最后一个报文段。为什么发送方知道有更多的数据需要发送还设置报文段 7中的PUSH标志呢？这是因为虽然我们指定写的是8192个字节的数据，但发送方的发送缓存却是4096个字节。

值得注意的另外一点是在图20-7中的第14、15和16这三个连续的确认报文段。在图20-3中我们也观察到了两个连续的ACK，但那是因为接收方已经通告其窗口为0（使发送方停止）。当窗口张开时，需要发送另一个窗口非0的ACK来使发送方重新启动。可是，在图20-7中，窗口的大小从来没有达到过0。然而，当窗口大小增加了2048个字节的时候，另一个ACK(报文段15和16)被发送以通知对方窗口被更新（在报文段15和16中，这两个窗口更新是不需要的，因为已经收到了对方的FIN，表明它不会再发送任何数据）。许多TCP实现在窗口大小增加了两个最大报文段长度（本例中为2048字节，因为MSS为1024字节）或者最大可能窗口的50%（本例中为2048字节，因为最大窗口大小为4096字节）时发送这个窗口更新。在第22.3节详细考察糊涂窗口综合症的时候，我们还会看到这种现象。

作为PUSH标志的另一个例子，再次回到图20-3。我们之所以看到前4个报文段（4~7）的标志被设置，是因为它们每一个均使TCP产生了一个报文段并提交给IP层。但是随后，TCP停下来等待一个确认来移动4096字节的窗口。在此期间，TCP又得到了应用程序的最后4096个字节的数据。当窗口张开时（报文段9），发送方TCP知道它有4个可立即发送的报文段，因此它只设置了最后一个报文段（13）的PUSH标志。

20.6 慢启动

迄今为止，在本章所有的例子中，发送方一开始便向网络发送多个报文段，直至达到接收方通告的窗口大小为止。当发送方和接收方处于同一个局域网时，这种方式是可以的。但是如果在发送方和接收方之间存在多个路由器和速率较慢的链路时，就有可能出现一些问题。一些中间路由器必须缓存分组，并有可能耗尽存储器的空间。[Jacobson 1988]证明了这种连接方式是如何严重降低了TCP连接的吞吐量的。

现在，TCP需要支持一种被称为“慢启动(slow start)”的算法。该算法通过观察到新分组进入网络的速率应该与另一端返回确认的速率相同而进行工作。

慢启动为发送方的TCP增加了另一个窗口：拥塞窗口(congestion window)，记为cwnd。当

Linux公社 www.linuxidc.com

与另一个网络的主机建立TCP连接时，拥塞窗口被初始化为1个报文段（即另一端通告的报文段大小）。每收到一个ACK，拥塞窗口就增加一个报文段（cwnd以字节为单位，但是慢启动以报文段大小为单位进行增加）。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。拥塞窗口是发送方使用的流量控制，而通告窗口则是接收方使用的流量控制。

发送方开始时发送一个报文段，然后等待ACK。当收到该ACK时，拥塞窗口从1增加为2，即可以发送两个报文段。当收到这两个报文段的ACK时，拥塞窗口就增加为4。这是一种指数增加的关系。

在某些点上可能达到了互联网的容量，于是中间路由器开始丢弃分组。这就通知发送方它的拥塞窗口开得过大。当我们在下一章讨论TCP的超时和重传机制时，将会看到它们是怎样对拥塞窗口起作用的。现在，我们来观察一个实际中的慢启动。

一个例子

图20-8表示的是将从主机sun发送到主机vangogh.cs.berkeley.edu的数据。这些数据将通过一个慢的SLIP链路，该链路是TCP连接上的瓶颈（我们已经在时间系列上去掉了连接建立的过程）。

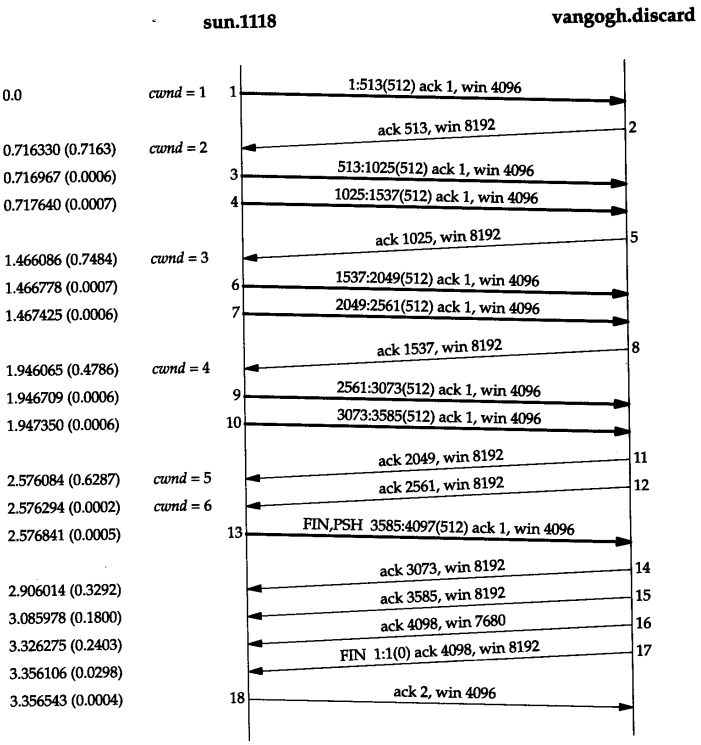


图20-8 慢启动的例子

我们观察到发送方发送一个长度为512字节的报文段，然后等待ACK。该ACK在716 ms后收到。这个时间是一个往返时间的指示。于是拥塞窗口增加了2个报文段，且又发送了两个报文段。当收到报文段5的ACK后，拥塞窗口增加为3。此时尽管可发送多达3个报文段，可是在下一个ACK收到之前，只发送了2个报文段。

在21.6节中我们将再次讨论慢启动，并介绍怎样采用另一种被称为“拥塞避免”的技术来

Linux公社 www.linuxidc.com

作为通常的实现。

20.7 成块数据的吞吐量

让我们看一看窗口大小、窗口流量控制以及慢启动对传输成块数据的 TCP连接的吞吐量的相互作用。

图20-9显示了左边的发送方和右边的接收方之间的一个 TCP连接上的时间系列，共显示了16个时间单元。为简单起见，本图只显示离散的时间单元。每个粗箭头线的上半部分显示的是从左到右的携带数据的报文段，标记为 1, 2, 3，等等。在粗线箭头下面表示的是反向传输的ACK。我们把ACK用细箭头线表示，并标注了被确认的报文段号。

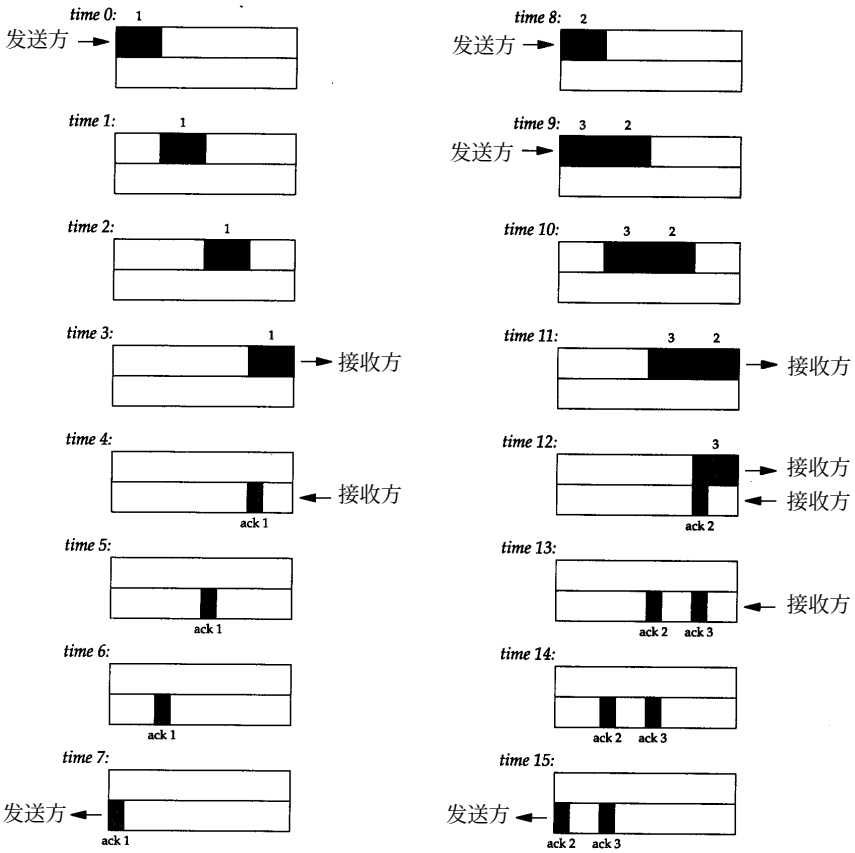


图20-9 时间0~15的成块数据吞吐量举例

在时间0，发送方发送了一个报文段。由于发送方处于慢启动中（其拥塞窗口为 1个报文段），因此在继续发送以前它必须等待该数据段的确认。

在时间1, 2和3，报文段从左向右移动一个时间单元。在时间 4接收方读取这个报文段并产生确认。经过时间 5、6和7，ACK移动到左边的发送方。我们有了一个 8个时间单元的往返时间RTT（Round-Trip Time）。

我们有意把 ACK报文段画得比数据报文段小，这是因为它通常只有一个 IP首部和一个TCP首部。这里显示仅仅是一个单向的数据流动，并且假定 ACK的移动速率与数据报文段的移动速率相等。实际上并不总是这样。

Linux公社 www.linuxidc.com

通常发送一个分组的时间取决于两个因素：传播时延（由光的有限速率、传输设备的等待时间等引起）和一个取决于媒体速率（即媒体每秒可传输的比特数）的发送时延。对于一个给定的两个接点之间的通路，传播时延一般是固定的，而发送时延则取决于分组的大小。在速率较慢的情况下发送时延起主要作用（例如，在习题 7.2中我们甚至没有考虑传播时延），而在千兆比特速率下传播时延则占主要地位（见图24-6）。

当发送方收到ACK后，在时间8和9发送两个报文段（我们标记为2和3）。此时它的拥塞窗口为2个报文段。这两个报文段向右传送到接收方，在时间12和13接收方产生两个ACK。这两个返回到发送方的ACK之间的间隔与报文段之间的间隔一致，被称为TCP的自计时(self-clocking)行为。由于接收方只有在数据到达时才产生ACK，因此发送方接收到的ACK之间的间隔与数据到达接收方的间隔是一致的（然而在实际中，返回路径上的排队会改变ACK的到达率）。

图20-10表示的是后面16个时间单位。2个ACK的到达使得拥塞窗口从2个报文段增加为4个，而这4个报文段在时间16~19时被发送。第1个ACK在时间23到达。4个ACK的到达使得拥塞窗口从4个报文段增加为8个，并在时间24~31发送8个报文段。

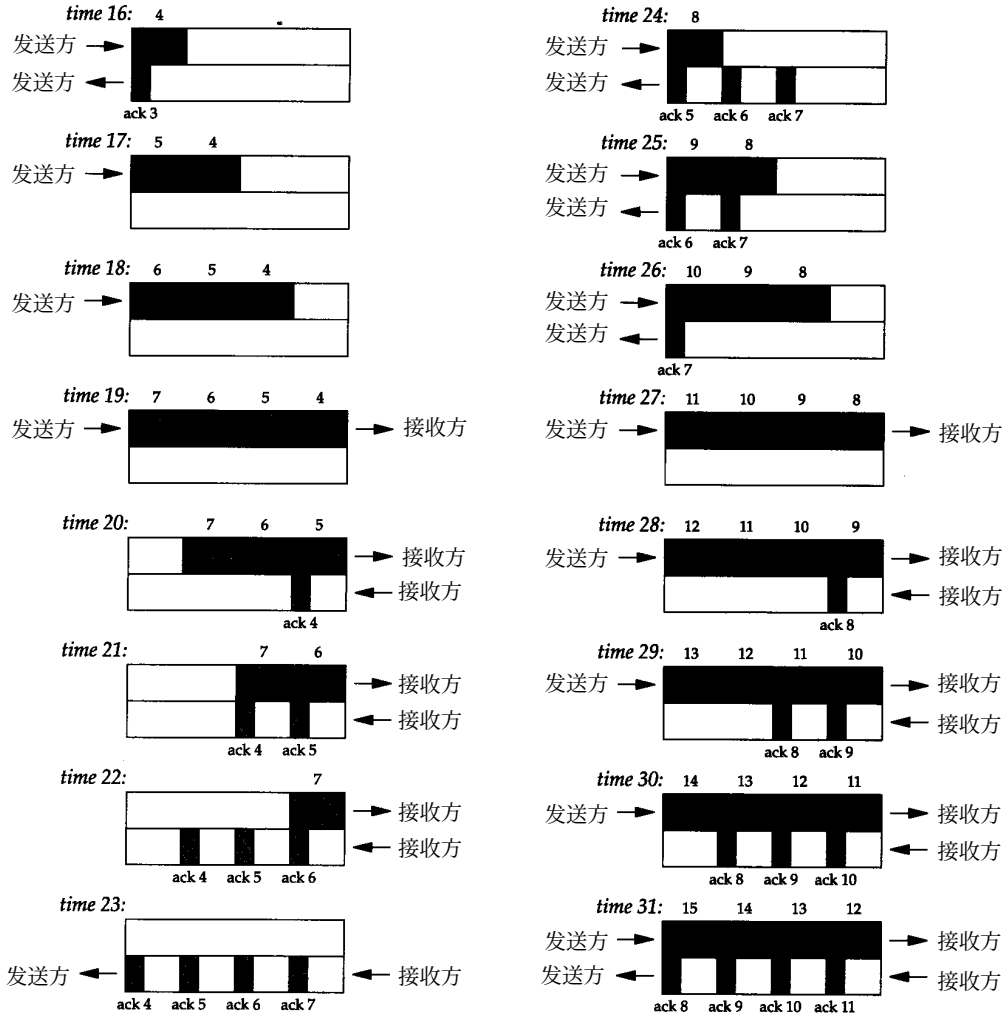


图20-10 时间16~31的成块数据吞吐量举例

Linux公社 www.linuxidc.com

在时间31及其后续时间，发送方和接收方之间的管道 (pipe)被填满。此时不论拥塞窗口和通告窗口是多少，它都不能再容纳更多的数据。每当接收方在某一个时间单位从网络上移去一个报文段，发送方就再发送一个报文段到网络上。但是不管有多少报文段填充了这个管道，返回路径上总是具有相同数目的 ACK。这就是连接的理想稳定状态。

20.7.1 带宽时延乘积

现在来回答窗口应该设置为多大的问题。在我们的例子中，作为最大的吞吐量，发送方在任何时候有8个已发送的报文段未被确认。接收方的通告窗口必须不小于这个数目，因为通告窗口限制了发送方能够发送的段的数目。

可以计算通道的容量为：

capacity (bit) = *bandwidth* (b/s) × *round-trip time* (s)

一般称之为带宽时延乘积。这个值依赖于网络速度和两端的 RTT，可以有很大的变动。例如，一条穿越美国（RTT约为60 ms）的T1的电话线路（1 544 000 b/s）的带宽时延乘积为11 580字节。对于20.4节中讨论的缓存大小而言，这个结果是合理的。但是一条穿越美国的 T3电话线路（45 000 000 b/s）的带宽时延乘积则为337 500字节，这个数值超过了最大所允许的TCP通告窗口的大小（65535字节）。在24.4节我们将讨论能够避免当前TCP限制的新的TCP窗口大小选项。

> T1电话线的1 544 000 b/s是原始比特率。由于每193个bit使用1个作为帧同步，因此实际数据率为1 536 000 b/s。一个T3电话线的原始比特率实际上是44 736 000 b/s，其数据率可达到44 210 000 b/s。在讨论中我们使用1.544 Mb/s和45 Mb/s。

不论是带宽还是时延均会影响发送方和接收方之间通路的容量。在图 20-11中我们显示了一个增加了一倍的RTT会使通路容量也增加一倍。

在图20-11底下的说明部分，通过使用一个较长的 RTT，这个管道能够容纳8个报文段而不是4个。

类似地，图20-12表示了增加一倍的带宽也可使该管道的容量增加一倍。

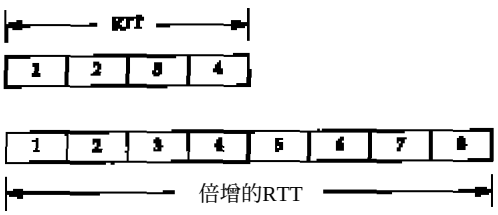


图20-11 RTT加倍可使管道容量增加一倍

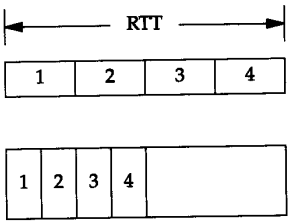


图20-12 带宽加倍可使管道容量增加一倍

在图20-12的下部，假定网络速率已经加倍，使得我们能够只使用上面一半的时间来发送4个报文段。这样，该管道的容量再次加倍（假定该图的上半部分与下半部分中的报文段具有同样大小，即具有相同的比特数）。

20.7.2 拥塞

当数据到达一个大的管道（如一个快速局域网）并向一个较小的管道（如一个较慢的广

Linux公社 www.linuxidc.com

域网）发送时便会发生拥塞。当多个输入流到达一个路由器，而路由器的输出流小于这些输入流的总和时也会发生拥塞。

图20-13显示了一个典型的大管道向小管道发送报文的情况。之所以说它典型，是因为大多数的主机都连接在局域网上，并通过一个路由器与速率相对较低的广域网相连（我们再次假定图中上半部分的报文段（9~20）都是相同的，而图中下半部分的ACK也都是相同的）。

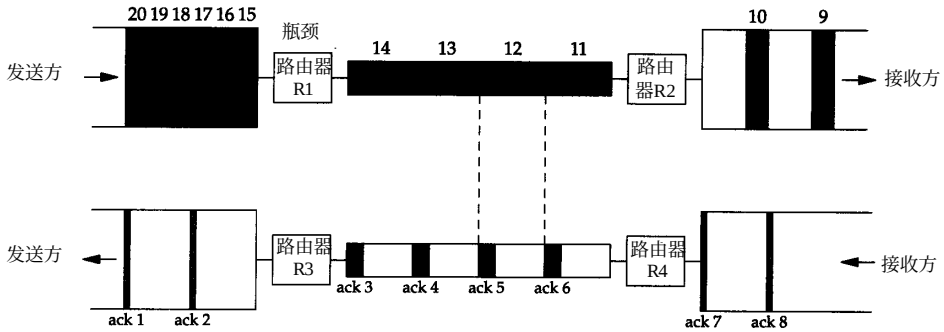


图20-13 从较大管道向较小管道发送分组引起的拥塞

在该图中，我们已经标记路由器 R1为“瓶颈”，因为它是拥塞发生的地方。它从左侧速率较高的局域网接收数据并向右侧速率较低的广域网发送（通常 R1与R3是同样的路由器，如同R2与R4一样。但这并不是必需的，有时也会使用不对称的路径）。当路由器R2将所接收到的分组发送到右侧的局域网时，这些分组之间维持与其左侧广域网上同样的间隔，尽管局域网具有更高的带宽。类似地，返回的确认之间的间隔也与其在路径中最慢的链路上的间隔一致。

在图20-13中已经假定发送方不使用慢启动，它按照局域网的带宽尽可能快地发送编号为1~20的报文段（假定接收方的通告窗口至少为 20个报文段）。正如我们看到的那样， ACK之间的间隔与在最慢链路上的一致。假定瓶颈路由器具有足够的容纳这 20个分组的缓存。如果这个不能保证，就会引起路由器丢弃分组。在 21.6节讨论避免拥塞时会看到怎样避免这种情况。

20.8 紧急方式

TCP提供了“紧急方式 (urgent mode)”，它使一端可以告诉另一端有些具有某种方式的“紧急数据”已经放置在普通的数据流中。另一端被通知这个紧急数据已被放置在普通数据流中，由接收方决定如何处理。

可以通过设置TCP首部（图17-2）中的两个字段来发出这种从一端到另一端的紧急数据已经被放置在数据流中的通知。 URG比特被置1，并且一个16bit的紧急指针被置为一个正的偏移量，该偏移量必须与 TCP首部中的序号字段相加，以便得出紧急数据的最后一个字节的序号。

仍有许多关于紧急指针是指向紧急数据的最后一个字节还是指向紧急数据最后一个字节的下一个字节的争论。最初的 TCP规范给出了两种解释，但Host Requirements RFC确定指向最后一个字节是正确的。

然而，问题在于大多数的实现（包括源自伯克利的实现）继续使用错误的解释。

Linux公社 www.linuxidc.com

所有符合Host Requirements RFC的实现都是可兼容的，但很有可能无法与其他大多数主机正确通信。

TCP必须通知接收进程，何时已接收到一个紧急数据指针以及何时某个紧急数据指针还不在此连接上，或者紧急指针是否在数据流中向前移动。接着接收进程可以读取数据流，并必须能够被告知何时碰到了紧急数据指针。只要从接收方当前读取位置到紧急数据指针之间有数据存在，就认为应用程序处于“紧急方式”。在紧急指针通过之后，应用程序便转回到正常方式。

TCP本身对紧急数据知之甚少。没有办法指明紧急数据从数据流的何处开始。TCP通过连接传送的唯一信息就是紧急方式已经开始（TCP首部中的URG比特）和指向紧急数据最后一个字节的指针。其他的事情留给应用程序去处理。

不幸的是，许多实现不正确地称TCP的紧急方式为带外数据(out-of-band data)。如果一个应用程序确实需要一个独立的带外信道，第二个TCP连接是达到这个目的的最简单的方法（许多运输层确实提供许多人认为的那种真正的带外数据：使用同一个连接的独立的逻辑数据通道作为正常的数据通道。这是TCP所没有提供的）。

TCP的紧急方式与带外数据之间的混淆，也是因为主要的编程接口（插口API）将TCP的紧急方式映射为称为带外数据的插口。

紧急方式有什么作用呢？两个最常见的例子是Telnet和Rlogin。当交互用户键入中断键时，我们在第26章将看到使用紧急方式来完成这个功能的例子。另一个例子是FTP，当交互用户放弃一个文件的传输时，我们将在第27章看到这样的一个例子。

Telnet和Rlogin从服务器到客户使用紧急方式是因为在这个方向上的数据流很可能要被客户的TCP停止（也即，它通告了一个大小为0的窗口）。但是如果服务器进程进入了紧急方式，尽管它不能够发送任何数据，服务器TCP也会立即发送紧急指针和URG标志。当客户TCP接收到这个通知时就会通知客户进程，于是客户可以从服务器读取其输入、打开窗口并使数据流动。

如果在接收方处理第一个紧急指针之前，发送方多次进入紧急方式会发生什么情况呢？在数据流中的紧急指针会向前移动，而其在接收方的前一个位置将丢失。接收方只有一个紧急指针，每当对方有新的值到达时它将被覆盖。这意味着如果发送方进入紧急方式时所写的内容对接收方非常重要，那么这些字节数据必须被发送方用某种方式特别标记。我们将看到Telnet通过在数据流中加入一个值为255的字节作为前缀来标记它所有的命令。

一个例子

让我们观察一下即使是在接收方窗口关闭的情况下，TCP是如何发送紧急数据的。在主机bsdi上启动sock程序，并使之在连接建立后和从网络读取前暂停10秒种（通过使用-P选项），这将使另一端填满发送窗口：

```
bsdi %sock -i -s -P10 5555
```

接着我们在主机sun上启动客户，使之使用一个8192字节的发送缓存（使用-S选项）并进行6个向网络写1024字节数据的操作（使用-n选项）。还指明-U5选项，告知它向网络写第5个缓存之前要写1个字节的数据，并进入紧急数据方式。我们指明详细标志来观察写的顺序：

Linux公社 www.linuxidc.com

```
sun % sock -v -i -n6 -S8192 -U5 bsdi 5555
connected on 140.252.13.33.1305 to 140.252.13.35.5555
SO_SNDBUF = 8192
TCP_MAXSEG = 1024
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1 byte of urgent data
wrote 1024 bytes
wrote 1024 bytes
```

我们设置发送缓存为8192个字节，以便让发送应用程序能够立即写所有的数据。图 20-14显示了tcpdump输出的这个交换过程的结果（删去了连接建立的过程）。第1~5行表示发送方用4个1024字节的报文段去填充接收方的窗口。然后由于接收方的窗口被填满（第 4行的ACK确认了数据，但并没有移动窗口的右边沿），所以发送方停止发送。

在写了第4个正常数据之后，应用进程写了1个字节并进入紧急方式。第6行是该应用进程写的结果，紧急指针被设置为4098。尽管发送方不能发送任何数据，但紧急指针和URG标志一起被发送。

5个这样的ACK在13 ms内被发送（第6~10行）。第1个ACK在应用进程写1个字节并进入紧急方式时被发送，后面两个在应用进程写最后两个 1024字节的数据时被发送（尽管 TCP不能发送这2048个字节的数据，可每次当应用程序执行写操作的时候， TCP的输出功能被调用。当TCP看到正处于紧急方式时，它会发送其他的紧急通知）。第4个ACK在应用进程关闭其TCP连接时被发送（TCP的输出功能再次被调用）。发送应用程序在启动几毫秒后终止 —— 在接收方应用进程已经发出其第一个写操作之前。 TCP将所有的数据进行排队，并在可能时发送出去（这就是为何指明发送缓存为 8192字节的原因，因此只有这样才能够把所有的数据都放置在缓存中）。第5个ACK很可能是在接收第 4行的ACK时产生的。发送 TCP很可能在这个ACK到达前便已将其第 4个报文段放入队列以便输出（第 5行）。另一端接收到这个 ACK也会引起TCP输出例程被调用。

```
 1  0.0                  sun.1305 > bsdi.5555: P 1:1025(1024) ack 1 win 4096
 2  0.073743 (0.0737)    sun.1305 > bsdi.5555: P 1025:2049(1024) ack 1 win 4096
 3  0.096969 (0.0232)    sun.1305 > bsdi.5555: P 2049:3073(1024) ack 1 win 4096
 4  0.157514 (0.0605)    bsdi.5555 > sun.1305: . ack 3073 win 1024
 5  0.164267 (0.0068)    sun.1305 > bsdi.5555: P 3073:4097(1024) ack 1 win 4096

 6  0.167961 (0.0037)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 7  0.171969 (0.0040)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 8  0.176196 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 9  0.180373 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
10  0.180768 (0.0004)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

11  0.367533 (0.1868)    bsdi.5555 > sun.1305: . ack 4097 win 0
12  0.368478 (0.0009)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

13  9.829712 (9.4612)    bsdi.5555 > sun.1305: . ack 4097 win 2048
14  9.831578 (0.0019)    sun.1305 > bsdi.5555: . 4097:5121(1024) ack 1 win 4096
                                                  urg 4098
15  9.833303 (0.0017)    sun.1305 > bsdi.5555: . 5121:6145(1024) ack 1 win 4096

16  9.835089 (0.0018)    bsdi.5555 > sun.1305: . ack 4097 win 4096
17  9.835913 (0.0008)    sun.1305 > bsdi.5555: FP 6145:6146(1) ack 1 win 4096
18  9.840264 (0.0044)    bsdi.5555 > sun.1305: . ack 6147 win 2048
19  9.842386 (0.0021)    bsdi.5555 > sun.1305: . ack 6147 win 4096
20  9.843622 (0.0012)    bsdi.5555 > sun.1305: F 1:1(0) ack 6147 win 4096
21  9.844320 (0.0007)    sun.1305 > bsdi.5555: . ack 2 win 4096
```

图20-14 tcpdump 对TCP紧急方式的输出结果

Linux公社 www.linuxidc.com

接着，接收方确认最后的1024字节的数据（第11行），但同时通告窗口为0。发送方用一个包含紧急通知的报文段进行了响应。

在第13行，当应用进程被唤醒、并从接收缓存读取一些数据时，接收方通告窗口为 2048字节。于是后面又发送了两个1024字节的报文段（第14和15行）。其中，由于紧急指针在第1个报文段的范围内，因此这个报文段被设置了紧急通知标志，而第 2个报文段则关闭了该标志。

当接收方再次打开窗口（第16行）时，发送方传输最后的数据（序号为6145）并发起正常的连接关闭。

图20-15显示了发送的6145个字节数据的序号。可以看到当进入紧急方式时所发送的字节的序号是4097，但在图20-14中紧急指针指向4098，这证明了该实现（SunOS 4.1.3）将紧急指针设置为紧急数据最后字节的下一个字节。

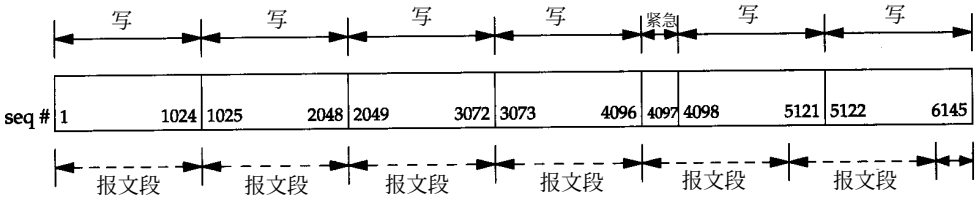


图20-15 紧急方式例子中，应用进程的写操作和TCP的一些报文段

该图还可以让我们观察TCP是如何对应用进程写的数据进行重新分组化的。当进入紧急方式时待输出的1个字节是与在缓存中的后面1023个字节一同发送的。下一个报文段也包含1024字节的数据，而最后一个报文段则只包含一个字节。

20.9 小结

正如我们在本章一开始时讲的那样，没有一种单一的方法可以使用 TCP进行成块数据的交换。这是一个依赖于许多因素的动态处理过程，有些因素我们可以控制（如发送和接收缓存的大小），而另一些我们则没有办法控制（如网络拥塞、与实现有关的特性等）。在本章，我们已经考察了许多TCP的传输过程，介绍了所有我们能够看到的特点和算法。

进行成块数据有效传输的最重要的方法是TCP的滑动窗口协议。我们考察了TCP为使发送方和接收方之间的管道充满来获得最可能快的传输速度而采用的方法。我们用带宽时延乘积衡量管道的容量，并分析了该乘积与窗口大小之间的关系。在24.8节介绍TCP性能的时候将再次涉及这个概念。

我们还介绍了TCP的PUSH标志，因为在跟踪结果中总是观察到它，但我们无法对它的设置与否进行控制。本章最后一个主题是TCP的紧急数据，人们常常错误地称其为“带外数据”。TCP的紧急方式只是一个从发送方到接收方的通知，该通知告诉接收方紧急数据已被发送，并提供该数据最后一个字节的序号。应用程序使用的有关紧急数据部分的编程接口常常都不是最佳的，从而导致更多的混乱。

习题

20.1 在图20-6中，我们可以看到一个序号为0的字节和一个序号为8193的字节，试问这两个

Linux公社 www.linuxidc.com

字节的含义是什么？

20.2 提前观察图22-1，并解释主机bsdi设置PUSH标志的含义。

20.3 在一个Usenet记录中，有人抱怨说美国和日本之间的一个128 ms时延、速率为256 000 b/s的链路吞吐量为120 000 b/s（利用率为47%），而当链路通过卫星时其吞吐量则为33 000 b/s（利用率为13%）。试问在这两种情况下窗口大小各为多少（假定卫星链路的时延为500 ms）？卫星链路的窗口大小应该如何调整？

20.4 如果API提供一种方法，使得发送方可以告诉其TCP打开PUSH标志，而接收方可以查询一个接收的报文段是否被设置了PUSH标志，试问该标志能否被用作一个记录标记？

20.5 在图20-3中为什么没有合并报文段15和16？

20.6 在图20-13中，我们假定对应数据报文段之间的间隔，返回的ACK之间的间隔被分隔得很好。如果在链路某处进行缓存并使许多ACK同时到达发送方，试问会发生什么情况？

Linux公社 www.linuxidc.com

第21章　TCP的超时与重传

21.1　引言

TCP提供可靠的运输层。它使用的方法之一就是确认从另一端收到的数据。但数据和确认都有可能会丢失。 TCP通过在发送时设置一个定时器来解决这种问题。如果当定时器溢出时还没有收到确认，它就重传该数据。对任何实现而言，关键之处就在于超时和重传的策略，即怎样决定超时间隔和如何确定重传的频率。

我们已经看到过两个超时和重传的例子：（1）在6.5节的ICMP端口不能到达的例子中，看到TFTP客户使用UDP实现了一个简单的超时和重传机制：假定 5秒是一个适当的时间间隔，并每隔5秒进行重传；（2）在向一个不存在的主机发送 ARP的例子中（第4.5节），我们看到当TCP试图建立连接的时候，在每个重传之间使用一个较长的时延来重传 SYN。

对每个连接，TCP管理4个不同的定时器。

1) 重传定时器使用于当希望收到另一端的确认。在本章我们将详细讨论这个定时器以及一些相关的问题，如拥塞避免。

2) 坚持(persist)定时器使窗口大小信息保持不断流动，即使另一端关闭了其接收窗口。第22章将讨论这个问题。

3) 保活(keepalive)定时器可检测到一个空闲连接的另一端何时崩溃或重启。第 23章将描述这个定时器。

4) 2MSL定时器测量一个连接处于 TIME_WAIT状态的时间。我们在 18.6节对该状态进行了介绍。

本章以一个简单的 TCP超时和重传的例子开始，然后转向一个更复杂的例子。该例子可以使我们观察到TCP时钟管理的所有细节。可以看到 TCP的典型实现是怎样测量 TCP报文段的往返时间以及 TCP如何使用这些测量结果来为下一个将要传输的报文段建立重传超时时间。接着我们将研究TCP的拥塞避免 —— 当分组丢失时TCP所采取的动作 —— 并提供一个分组丢失的实际例子，我们还将介绍较新的快速重传和快速恢复算法，并介绍该算法如何使　TCP检测分组丢失比等待时钟超时更快。

21.2　超时与重传的简单例子

首先观察TCP所使用的重传机制，我们将建立一个连接，发送一些分组来证明一切正常，然后拔掉电缆，发送更多的数据，再观察 TCP的行为。

```
bsdi % telnet svr4 discard
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hello, world                          正常发送本行
and hi                                在发送本行前断连
Connection closed by foreign host.    9分钟后TCP放弃时输出
```

Linux公社 www.linuxidc.com

图21-1表示的是tcpdump的输出结果（已经去掉了bsdi设置的服务类型信息）。

```
 1    0.0                bsdi.1029 > svr4.discard: S 1747921409:1747921409(0)
                                                     win 4096 <mss 1024>
 2    0.004811 ( 0.0048) svr4.discard > bsdi.1029: S 3416685569:3416685569(0)
                                                     ack 1747921410
                                                     win 4096 <mss 1024>
 3    0.006441 ( 0.0016) bsdi.1029 > svr4.discard: . ack 1 win 4096

 4    6.102290 ( 6.0958) bsdi.1029 > svr4.discard: P 1:15(14) ack 1 win 4096
 5    6.259410 ( 0.1571) svr4.discard > bsdi.1029: . ack 15 win 4096

 6   24.480158 (18.2207) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
 7   25.493733 ( 1.0136) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
 8   28.493795 ( 3.0001) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
 9   34.493971 ( 6.0002) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
10   46.484427 (11.9905) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
11   70.485105 (24.0007) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
12  118.486408 (48.0013) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
13  182.488164 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
14  246.489921 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
15  310.491678 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
16  374.493431 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
17  438.495196 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
18  502.486941 (63.9917) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
19  566.488478 (64.0015) bsdi.1029 > svr4.discard: R 23:23(0) ack 1 win 4096
```

图21-1 TCP超时和重传的简单例子

第1、2和3行表示正常的TCP连接建立的过程，第4行是“hello, world”（12个字符加上回车和换行）的传输过程，第5行是其确认。接着我们从svr4拔掉了以太网电缆，第6行表示“and hi”将被发送。第7~18行是这个报文段的12次重传过程，而第19行则是发送方的TCP最终放弃并发送一个复位信号的过程。

现在检查连续重传之间不同的时间差，它们取整后分别为1、3、6、12、24、48和多个64秒。在本章的后面，我们将看到当第一次发送后所设置的超时时间实际上为1.5秒（它在首次发送后的1.0136秒而不是精确的1.5秒后，发生的原因我们已在图18-7中进行了解释），此后该时间在每次重传时增加1倍并直至64秒。

这个倍乘关系被称为“指数退避(exponential backoff)”。可以将该例子与6.5节中的TFTP例子比较，在那里每次重传总是在前一次的5秒后发生。

首次分组传输（第6行，24.480秒）与复位信号传输（第19行，566.488秒）之间的时间差约为9分钟，该时间在目前的TCP实现中是不可变的。

> 对于大多数实现而言，这个总时间是不可调整的。Solaris 2.2允许管理者改变这个时间（E.4节中的tcp_ip_abort_interval变量），且其默认值为2分钟，而不是最常用的9分钟。

21.3 往返时间测量

TCP超时与重传中最重要的部分就是对一个给定连接的往返时间（RTT）的测量。由于路由器和网络流量均会变化，因此我们认为这个时间可能经常会发生变化，TCP应该跟踪这些变化并相应地改变其超时时间。

首先TCP必须测量在发送一个带有特别序号的字节和接收到包含该字节的确认之间的RTT。在上一章中，我们曾提到在数据报文段和ACK之间通常并没有一一对应的关系。在图

Linux公社 www.linuxidc.com

20.1中，这意味着发送方可以测量到的一个 RTT，是在发送报文段4（第1~1024字节）和接收报文段7（对1~1024字节的ACK）之间的时间，用M表示所测量到的RTT。

最初的TCP规范使TCP使用低通过滤器来更新一个被平滑的 RTT估计器（记为O）。

$$R \leftarrow \alpha R + (1-\alpha)M$$

这里的α是一个推荐值为 0.9的平滑因子。每次进行新测量的时候，这个被平滑的 RTT将得到更新。每个新估计的90%来自前一个估计，而10%则取自新的测量。

该算法在给定这个随RTT的变化而变化的平滑因子的条件下， RFC 793推荐的重传超时时间*RTO*（Retransmission TimeOut）的值应该设置为

$$RTO = R\beta$$

这里的β是一个推荐值为2的时延离散因子。

[Jacobson 1988] 详细分析了在RTT变化范围很大时，使用这个方法无法跟上这种变化，从而引起不必要的重传。正如 Jacobson记述的那样，当网络已经处于饱和状态时，不必要的重传会增加网络的负载，对网络而言这就像在火上浇油一样。

除了被平滑的RTT估计器，所需要做的还有跟踪RTT的方差。在往返时间变化起伏很大时，基于均值和方差来计算 *RTO*，将比作为均值的常数倍数来计算 *RTO*能提供更好的响应。在[Jacobson 1988]中的图5和图6中显示了根据RFC 793计算的某些实际往返时间的*RTO*和下面考虑了往返时间的方差所计算的*RTO*的比较结果。

正如Jacobson所描述的，均值偏差是对标准偏差的一种好的逼近，但却更容易进行计算（计算标准偏差需要一个平方根）。这就引出了下面用于每个RTT测量M的公式。

$$Err = M - A$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + h(|Err| - D)$$

$$RTO = A + 4D$$

这里的A是被平滑的RTT（均值的估计器）而D则是被平滑的均值偏差。*Err*是刚得到的测量结果与当前的RTT估计器之差。A和D均被用于计算下一个重传时间（*RTO*）。增量g起平均作用，取为1/8（0.125）。偏差的增益是h，取值为0.25。当RTT变化时，较大的偏差增益将使 *RTO*快速上升。

[Jacobson 1988]指明在计算 *RTO*时使用 2D，但经过后来更深入的研究，[Jacobson1990c]将该值改为4D，也就是在BSD Net/1的实现中使用的那样。

Jacobson指明了一种使用整数运算来计算这些公式的方法，并被许多实现所采用（这也就是g, h和倍数4均是2的乘方的一个原因，这样一来计算均可只通过移位操作而不需要乘、除运算来完成）。

将Jacobson与最初的方法比较，我们发现被平滑的均值计算公式是类似的（ α是1减去增益g），而增益可使用不同的值。而且 Jacobson计算*RTO*的公式依赖于被平滑的RTT和被平滑的均值偏差，而最初的方法则使用了被平滑的 RTT的一个倍数。

在看完下一节中的例子时，我们将看到这些估计器是如何被初始化的。

Karn算法

在一个分组重传时会产生这样一个问题：假定一个分组被发送。当超时发生时， *RTO*正

Linux公社 www.linuxidc.com

如21.2节中显示的那样进行退避，分组以更长的*RTO*进行重传，然后收到一个确认。那么这个ACK是针对第一个分组的还是针对第二个分组呢？这就是所谓的重传多义性问题。

[Karn and Partridge 1987]规定，当一个超时和重传发生时，在重传数据的确认最后到达之前，不能更新RTT估计器，因为我们并不知道ACK对应哪次传输（也许第一次传输被延迟而并没有被丢弃，也有可能第一次传输的ACK被延迟）。

并且，由于数据被重传，*RTO*已经得到了一个指数退避，我们在下一次传输时使用这个退避后的*RTO*。对一个没有被重传的报文段而言，除非收到了一个确认，否则不要计算新的*RTO*。

21.4 往返时间RTT的例子

在本章中，我们将使用以下这些例子来检查TCP的超时和重传、慢启动以及拥塞避免等方方面面的实现细节。

使用`sock`程序和如下的命令来将32768字节的数据从主机slip发送到主机`vangogh.cs.berkeley.edu`上的丢弃服务。

```
slip %sock -D -i -n32 vangogh.cs.berkeley.edu discard
```

在扉页前图中，可以看到`slip`通过两个SLIP链路与140.252.1以太网相连，并从这里通过Internet到达目的地。通过使用两个9600 b/s的SLIP链路，我们期望能够得到一些可测量的时延。

该命令执行32个写1024字节的操作。由于`slip`和`bsdi`之间的MTU为296字节，因此这些操作会产生128个报文段，每个报文段包含256字节的用户数据。整个传输过程的时间约为45秒，我们观察到了一个超时和三次重传。

当该传输过程进行时，我们在`slip`上使用`tcpdump`来截获所有的发送和接收的报文段，并通过使用`-D`选项来打开插口排错功能（见A.6节），这样便可以通过运行一个修改后的`trpt`(8)程序来打印出连接控制块中与RTT、慢启动及拥塞避免等有关的多个变量。

对于给出的跟踪结果，我们不能够完全进行显示，相反，我们将在介绍本章时看到它的各个部分。图21-2显示的是前5秒中的数据和确认的传输过程。与前面`tcpdump`的输出相比，我们已对其显示稍微进行了修改。虽然我们仅能够在运行`tcpdump`的主机上测量分组发送和接收的时间，但在本图中我们希望显示出分组正在网络中传输（它们确实存在，因为这个局域网连接与共享式的以太网并不一样）以及接收主机何时可能产生ACK（在本图中去掉了所有的窗口大小通告。主机`slip`总是通告窗口大小为4096，而`vangogh`则总是通告窗口大小为8192）。

还需要注意的是在本图中我们已经将报文段按照在主机`slip`上发送和接收的序号记为1~13和15。这与在这个主机上所收集的`tcpdump`的输出结果有关。

21.4.1 往返时间RTT的测量

在图21-2左边的时间轴上有三个括号，它们表明为进行RTT计算对哪些报文段进行了计时，并不是所有的报文段都被计时。

大多数源于伯克利的TCP实现在任何时候对每个连接仅测量一次RTT值。在发送一个报文段时，如果给定连接的定时器已经被使用，则该报文段不被计时。

Linux公社 www.linuxidc.com

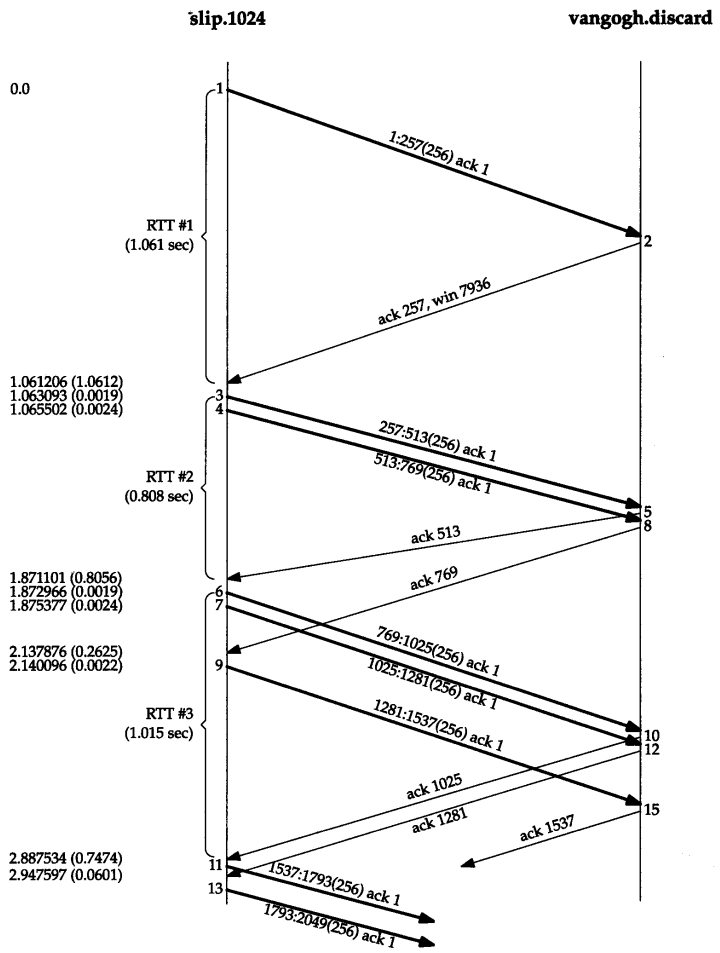


图21-2 分组交换和RTT测量

在每次调用500 ms的TCP的定时器例程时，就增加一个计数器来完成计时。这意味着，如果一个报文段的确认在它发送550 ms后到达，则该报文段的往返时间RTT将是1个滴答（即500 ms）或是2个滴答(即1000 ms)。

对每个连接而言，除了这个滴答计数器，报文段中数据的起始序号也被记录下来。当收到一个包含这个序号的确认后，该定时器就被关闭。如果ACK到达时数据没有被重传，则被平滑的RTT和被平滑的均值偏差将基于这个新测量进行更新。

图21-2中连接上的定时器在发送报文段1时启动，并在确认（报文段2）到达时终止。尽管它的RTT是1.061秒（tcpdump的输出），但插口排错的信息显示该过程经历了3个TCP时钟滴答，即RTT为1500 ms。

下一个被计时的是报文段3。当2.4 ms后传输报文段4时，由于连接的定时器已经被启动，因此该报文段不能被计时。当报文段5到达时，确认了正在被计时的数据。虽然我们从tcpdump的输出结果可以看到其RTT是0.808秒，但它的RTT被计算为1个滴答（500 ms）。

定时器在发送报文段6时再次被启动，并在1.015秒后接收到它的确认（报文段10）时终止。测量到的RTT是2个滴答。报文段7和9不能被计时，因为定时器已经被使用。而且，当收

Linux公社 www.linuxidc.com

到报文段8（第769字节的确认）时，由于该报文段不是正在计时的数据的确认，因此什么也没有进行更新。

图21-3显示了本例中通过tcpdump的输出所得到的实际RTT与时钟滴答计数之间的关系。

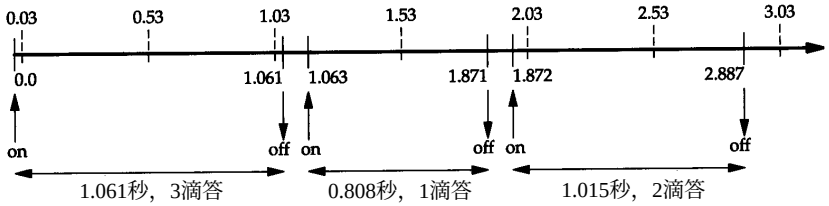


图21-3 RTT测量和时钟滴答

在图的上端表示间隔为500 ms的时钟滴答，图的下端表示tcpdump的输出时间及定时器何时被启动和关闭。在发送报文段 1和接收到报文段 2之间经历了 3个滴答，时间为 1.061秒，因此假定第1个滴答发生在0.03秒处（第1个滴答一定在0~0.061秒之间）。接着该图表示了第 2个被测量的RTT为什么被记为1个滴答，而第3个被记为2个滴答。

在这个完整的例子中， 128个报文段被传送，并收集了 18个RTT采样。图21-4表示了测量的RTT（取自tcpdump的输出）和TCP为超时所使用的 *RTO*（取自插口排错的输出）。在图21-2中，*x*轴从时间0开始，表示的是传输报文段1的时刻，而不是传输第1个SYN的时刻。

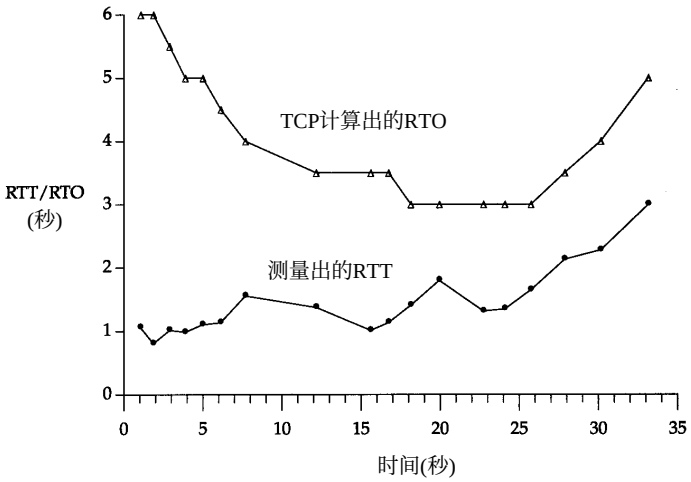


图21-4 测量出的RTT和TCP计算的*RTO*的例子

测量出RTT的前3个数据点对应图21-2所示的3个RTT。在时间10, 14和21处的间隔是由在这些时刻附近发生的重传（将在本章后面给出）引起的。 Karn算法在另一个报文段被发送和确认之前阻止我们更新估计器。同样注意到在这个实现中， TCP计算的*RTO*总是500 ms的倍数。

21.4.2 RTT估计器的计算

让我们来看一下RTT估计器（平滑的RTT和平滑的均值偏差）是如何被初始化和更新，以及每个重传超时是怎样计算的。

变量*A*和*D*分别被初始化为0和3秒。初始的重传超时使用下面的公式进行计算

Linux公社 www.linuxidc.com

$$RTO = A + 2D = 0 + 2 \times 3 = 6\ s$$

（因子2*D*只在这个初始化计算中使用。正如前面提到的，以后使用4*D*和*A*相加来计算*RTO*）。这就是传输初始SYN所使用的*RTO*。

结果是这个初始SYN丢失了，然后超时并引起了重传。图 21-5给出了tcpdump输出文件中的前4行。

```
1  0.0                  slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                       win 4096 <mss 256>

2  5.802377 (5.8024)    slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                       win 4096 <mss 256>

3  6.269395 (0.4670)    vangogh.discard > slip.1024: S 1365512705:1365512705(0)
                                                       ack 35648002
                                                       win 8192 <mss 512>

4  6.270796 (0.0014)    slip.1024 > vangogh.discard: . ack 1 win 4096
```

图21-5 初始SYN的超时和重传

当超时在5.802秒后发生时，计算当前的*RTO*值为

$$RTO = A + 4D = 0 + 4 \times 3 = 12\ s$$

因此，应用于*RTO*的指数退避取为 12。由于这是第 1次超时，我们使用倍数 2，因此下一个超时时间取值为24秒。再下一个超时时间的倍数为 4，得出值为48秒（这些初始*RTO*，对于一个连接上的最初的SYN，取值为6秒，接下来为24秒，正是我们在图4-5中看到的）。

ACK在重传后467ms到达。*A*和*D*的值没有被更新，这是因为 Karn算法对重传的处理比较模糊。下一个发送的报文段是第 4行的ACK，但它只是一个 ACK，所以没有被计时（只有数据报文段才会被计时）。

当发送第1个数据报文段时（图 21-2中的报文段1），*RTO*没有改变，这同样是由于 Karn算法。当前的24秒一直被使用，直到进行一个 RTT测量。这意味着图 21-4中时间0的*RTO*并不真的是24，但我们没有画出那个点。

当第1个数据报文段的ACK（图21-2中的报文段2）到达时，经历了3个时钟滴答，估计器被初始化为

$$A = M + 0.5 = 1.5 + 0.5 = 2$$

$$D = A/2 = 1$$

（因为经历3个时钟滴答，因此，*M*取值为1.5）。在前面，*A*和*D*初始化为0，*RTO*的初始计算值为3。这是使用第1个RTT的测量结果M对估计器进行首次计算的初始值。计算的 *RTO*值为

$$RTO = A + 4D = 2 + 4 \times 1 = 6\ s$$

当第2个数据报文段的ACK（图21-2中的报文段5）到达时，经历了1个时钟滴答（0.5秒），估计器按如下更新：

$$Err = M - A = 0.5 - 2 = -1.5$$

$$A = A + gErr = 2 - 0.125 \times 1.5 = 1.8125$$

$$D = D + h(|Err| - D) = 1 + 0.25 \times (1.5 - 1) = 1.125$$

$$RTO = A + 4D = 1.8125 + 4 \times 1.125 = 6.3125$$

Err、*A*和*D*的定点表示与实际使用的定点计算（在简化浮点计算中表示过）有一些微小的差别。这些不同使*RTO*取值为6秒（而非6.3125秒），正如我们在图 21-4中的时间1.871处所画的那样。

Linux公社 www.linuxidc.com

21.4.3 慢启动

我们在第20.6节介绍了慢启动算法，在图21-2中可再次看到它的工作过程。

连接上最初只允许传输一个报文段，然后在发送下一个报文段之前必须等待接收它的确认。当报文段2被接收后，就可以再发送两个报文段。

21.5 拥塞举例

现在观察一下数据报文段的传输过程。图 21-6显示了报文段中数据的起始序号与该报文段发送时间的对比图。它提供了一种较好的数据传输的可视化方法。通常代表数据的点将向上和向右移动，这些点的斜率就表示传输速率。当这些点向下和向右移动则表示发生了重传。

在21.4节开始时，我们曾提到整个传输的时间约为 45秒，但在本图中只显示了 35秒钟。这35秒只是数据报文段发送的时间。因为第 1个SYN看来是丢失了并被重传（见图 21-5），因此第1个数据报文段是在第1个SYN发送6.3秒后才发送的。而且，在发送最后一个数据报文段和FIN（图21-6中的34.1秒）之后，在接收方的FIN到达之前，又花费了另外的4.0秒接收来自接收方的最后14个ACK。

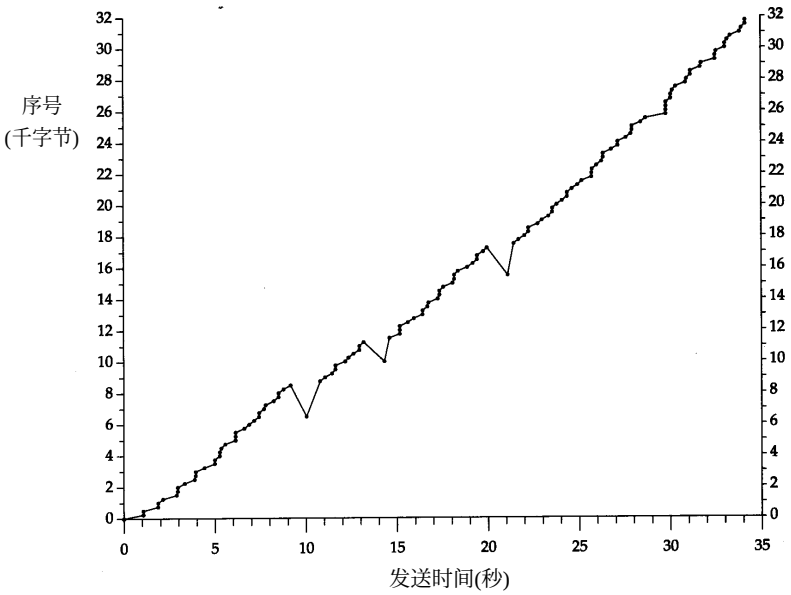


图21-6 从slip 发送32768个字节的数据到vangogh

可以立即看到图21-6中发生在时刻10，14和21附近的3个重传。我们还可以看到在这 3个点中只进行了一次报文段的重传，因为只有一个点下垂低于向上的斜率。

仔细检查一下这几个下垂点中的第 1个点（在10秒标记处的附近）。整理tcpdump的输出结果可以得到图21-7。

在这个图中，除了下面将要讨论的报文段 72，已经去掉了其他所有的窗口通告。主机slip总是通告窗口大小为4096，而主机vangogh则通告窗口为8192。该图中报文段的编号可以看作是图21-2的延续，在那里报文段的编号从 1开始。与图21-2一样，报文段根据在slip上发送和接收的顺序进行编号， tcpdump在主机slip上运行。我们还去掉了一些与讨论无

Linux公社 www.linuxidc.com

关的段（第44, 47和49以及所有来自vangogh的ACK）。

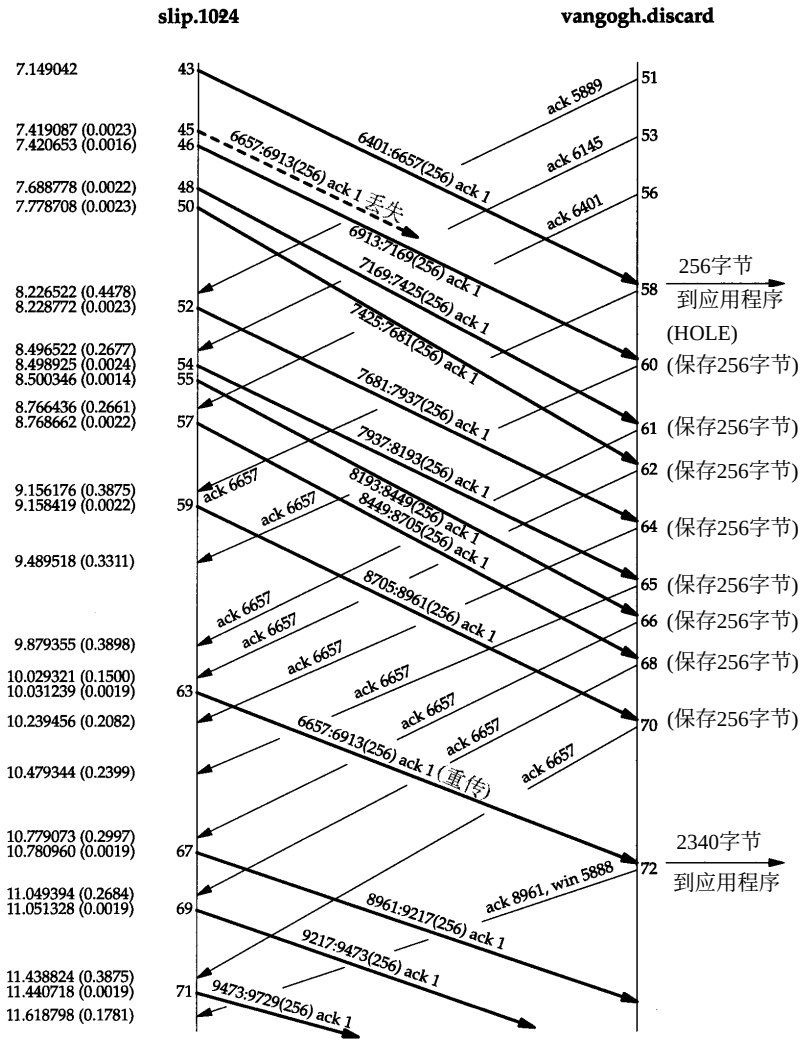


图21-7 10秒标记处附近重传的分组交换

看来报文段45丢失或损坏了，这一点无法从该输出上进行辨认。能够在主机 slip上看到的是对第6657字节（报文段58）以前数据的确认（不包括字节6657在内）。紧接着的是带有相同序号的8个ACK。正是接收到报文段62，也就是第3个重复ACK，才引起自序号6657开始的数据报文段（报文段63）进行重传。的确，源于伯克利的TCP实现对收到的重复ACK进行计数，当收到第3个时，就假定一个报文段已经丢失并重传自那个序号起的一个报文段。这就是Jacobson的快速重传算法，该算法通常与他的快速恢复算法一起配合使用。我们在第21.7节中介绍这两个算法。

注意到在重传后（报文段63），发送方继续正常的数据传输（报文段67、69和71）。TCP不需要等待对方确认重传。

现在检查一下在接收端发生了什么。当按序收到正常数据（报文段43）后，接收TCP将255个字节的数据交给用户进程。但下一个收到的报文段（报文段46）是失序的：数据的开始

Linux公社 www.linuxidc.com

序号（6913）并不是下一个期望的序号（6657）。TCP保存256字节的数据，并返回一个已成功接收数据的最大序号加1（6657）的ACK。被vangogh接收到的后面7个报文段（48, 50, 52, 54, 55, 57和59）也是失序的，接收方TCP保存这些数据并产生重复ACK。

目前TCP尚无办法告诉对方缺少一个报文段，也无法确认失序数据。此时主机 vangogh 所能够做的就是继续发送确认序号为6657的ACK。

当缺少的报文段（报文段63）到达时，接收方TCP在其缓存中保存第6657~8960字节的数据，并将这2304字节的数据交给用户进程。所有这些数据在报文段 72中进行确认。请注意此时该ACK通告窗口大小为5888（8192-2304），这是因为用户进程没有机会读取这些已准备好的2304字节的数据。

如果仔细检查图21-6 中tcpdump的输出中第14和21秒附近的下垂点，我们会看到它们也是由于收到了3个重复ACK引起的，这表明一个分组已经丢失。在这些例子中只有一个分组被重传。

在介绍完拥塞避免算法后，将在第21.8节中继续讨论这个例子。

21.6 拥塞避免算法

在第20.6节介绍的慢启动算法是在一个连接上发起数据流的方法，但有时我们会达到中间路由器的极限，此时分组将被丢弃。拥塞避免算法是一种处理丢失分组的方法。该方法的具体描述见 [Jacobson 1988]。

该算法假定由于分组受到损坏引起的丢失是非常少的（远小于 1%），因此分组丢失就意味着在源主机和目的主机之间的某处网络上发生了拥塞。有两种分组丢失的指示：发生超时和接收到重复的确认（我们在 21.5节看到这种现象。如果使用超时作为拥塞指示，则需要使用一个好的RTT算法，正如在21.3节中描述的那样）。

拥塞避免算法和慢启动算法是两个目的不同、独立的算法。但是当拥塞发生时，我们希望降低分组进入网络的传输速率，于是可以调用慢启动来作到这一点。在实际中这两个算法通常在一起实现。

拥塞避免算法和慢启动算法需要对每个连接维持两个变量：一个拥塞窗口 *cwnd*和一个慢启动门限*ssthresh*。这样得到的算法的工作过程如下：

1) 对一个给定的连接，初始化*cwnd*为1个报文段，*ssthresh*为65535个字节。

2) TCP输出例程的输出不能超过*cwnd*和接收方通告窗口的大小。拥塞避免是发送方使用的流量控制，而通告窗口则是接收方进行的流量控制。前者是发送方感受到的网络拥塞的估计，而后者则与接收方在该连接上的可用缓存大小有关。

3) 当拥塞发生时（超时或收到重复确认），*ssthresh*被设置为当前窗口大小的一半（*cwnd*和接收方通告窗口大小的最小值，但最少为 2个报文段）。此外，如果是超时引起了拥塞，则*cwnd*被设置为1个报文段（这就是慢启动）。

4) 当新的数据被对方确认时，就增加 *cwnd*，但增加的方法依赖于我们是否正在进行慢启动或拥塞避免。如果*cwnd*小于或等于*ssthresh*，则正在进行慢启动，否则正在进行拥塞避免。慢启动一直持续到我们回到当拥塞发生时所处位置的半时候才停止（因为我们记录了在步骤 2中给我们制造麻烦的窗口大小的一半），然后转为执行拥塞避免。

慢启动算法初始设置 *cwnd*为1个报文段，此后每收到一个确认就加 1。正如20.6节描述的

Linux公社 www.linuxidc.com

那样，这会使窗口按指数方式增长：发送1个报文段，然后是2个，接着是4个……。

拥塞避免算法要求每次收到一个确认时将*cwnd*增加1/*cwnd*。与慢启动的指数增加比起来，这是一种加性增长(additive increase)。我们希望在一个往返时间内最多为*cwnd*增加1个报文段（不管在这个RTT中收到了多少个ACK），然而慢启动将根据这个往返时间中所收到的确认的个数增加*cwnd*。

所有的4.3BSD版本和4.4BSD都在拥塞避免中将增加值不正确地设置为1个报文段的一小部分（即一个报文段的大小除以8），这是错误的，并在以后的版本中不再使用[Floyd 1994]。但是，为了和（不正确的）实现的结果对应，我们在将来的计算中给出了这个细节。

在[Leffler et al. 1989]中介绍的4.3BSD Tahoe版本仅在对方处于一个不同的网络上时才进行慢启动。而4.3BSD Reno版本改变了这种做法，因此，慢启动总是被执行。

图21-8是慢启动和拥塞避免的一个可视化描述。我们以段为单位来显示 *cwnd*和*ssthresh*，但它们实际上都是以字节为单位进行维护的。

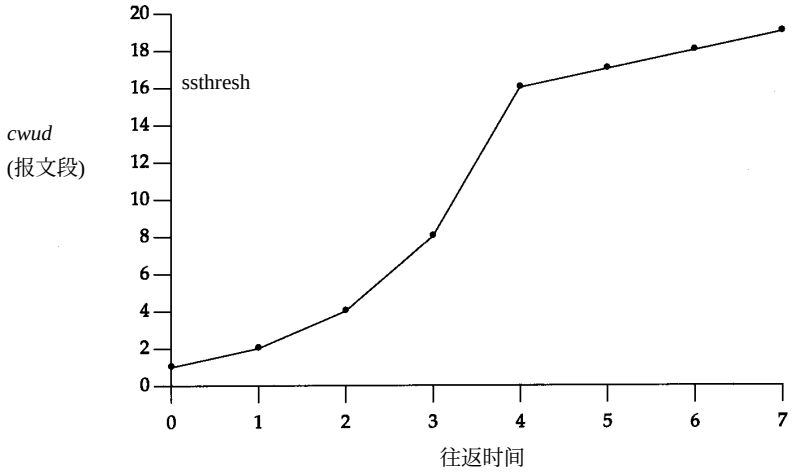


图21-8 慢启动和拥塞避免的可视化描述

在该图中，假定当*cwnd*为32个报文段时就会发生拥塞。于是设置 *ssthresh*为16个报文段，而*cwnd*为1个报文段。在时刻0发送了一个报文段，并假定在时刻1接收到它的ACK，此时*cwnd*增加为2。接着发送了2个报文段，并假定在时刻2接收到它们的ACK，于是*cwnd*增加为4（对每个ACK增加1次）。这种指数增加算法一直进行到在时刻3和4之间收到8个ACK后*cwnd*等于*ssthresh*时才停止，从该时刻起，*cwnd*以线性方式增加，在每个往返时间内最多增加1个报文段。

正如我们在这个图中看到的那样，术语“慢启动”并不完全正确。它只是采用了比引起拥塞更慢些的分组传输速率，但在慢启动期间进入网络的分组数增加的速率仍然是在增加的。只有在达到*ssthresh*拥塞避免算法起作用时，这种增加的速率才会慢下来。

21.7 快速重传与快速恢复算法

拥塞避免算法的修改建议1990年提出[Jacobson 1990b]。在我们的例子（见21.5节）中已

Linux公社 www.linuxidc.com

经可以看到这些实施中的修改。

在介绍修改之前，我们认识到在收到一个失序的报文段时， TCP立即需要产生一个ACK（一个重复的ACK）。这个重复的ACK不应该被迟延。该重复的ACK的目的在于让对方知道收到一个失序的报文段，并告诉对方自己希望收到的序号。

由于我们不知道一个重复的ACK是由一个丢失的报文段引起的，还是由于仅仅出现了几个报文段的重新排序，因此我们等待少量重复的ACK到来。假如这只是一些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的ACK之前，只可能产生1~2个重复的ACK。如果一连串收到3个或3个以上的重复ACK，就非常可能是一个报文段丢失了（我们在21.5节中见到过这种现象）。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。接下来执行的不是慢启动算法而是拥塞避免算法。这就是快速恢复算法。

在图21-7中可以看到在收到3个重复的ACK之后没有执行慢启动。相反，发送方进行重传，接着在收到重传的ACK以前，发送了3个新的数据的报文段（报文段67, 69和71）。

在这种情况下没有执行慢启动的原因是由于收到重复的ACK不仅仅告诉我们一个分组丢失了。由于接收方只有在收到另一个报文段时才会产生重复的ACK，而该报文段已经离开了网络并进入了接收方的缓存。也就是说，在收发两端之间仍然有流动的数据，而我们不想执行慢启动来突然减少数据流。

这个算法通常按如下过程进行实现：

1) 当收到第3个重复的ACK时，将*ssthresh*设置为当前拥塞窗口*cwnd*的一半。重传丢失的报文段。设置*cwnd*为*ssthresh*加上3倍的报文段大小。

2) 每次收到另一个重复的ACK时，*cwnd*增加1个报文段大小并发送1个分组（如果新的*cwnd*允许发送）。

3) 当下一个确认新数据的ACK到达时，设置*cwnd*为*ssthresh*（在第1步中设置的值）。这个ACK应该是在进行重传后的一个往返时间内对步骤1中重传的确认。另外，这个ACK也应该是对丢失的分组和收到的第1个重复的ACK之间的所有中间报文段的确认。这一步采用的是拥塞避免，因为当分组丢失时我们将当前的速率减半。

在下一节中我们将看到变量*cwnd*和*ssthresh*的计算过程。

快速重传算法最早出现在4.3BSD Tahoe版本中，但它随后错误地使用了慢启动。

快速恢复算法出现在4.3BSD Reno版本中。

21.8 拥塞举例(续)

通过使用`tcmdump`和插口排错选项（在第21.4节进行了介绍）来观察一个连接，就会在发送每一个报文段时看到*cwnd*和*ssthresh*的值。如果MSS为256字节，则*cwnd*和*ssthresh*的初始值分别为256和65535字节。每当收到一个ACK时，我们可以看到*cwnd*增加了一个MSS，取值分别为512, 768, 1024, 1280等。假定不会发生拥塞，则最终拥塞窗口将超过接收方的通告窗口，意味着通告窗口将对数据流进行限制。

一个更有趣的例子是观察在拥塞发生时的情况。使用与21.4节同样的例子。当这个例子运行时发生了4次拥塞。为建立连接而发送的初始SYN有一个因超时而引起的重传（见图21-5），接着在数据传输过程中有3个分组丢失（见图21-6）。

Linux公社 www.linuxidc.com

图21-9显示了当初始SYN重传并接着发送了前7个数据报文段时变量*cwnd*和*ssthresh*的值（在图21-2中显示了最初的数据报文段及其ACK之间的交换过程）。使用`tcpdump`的记号来表示数据字节：1:257(256)表示第1~256字节。

当SYN的超时发生时，*ssthresh*被置为其最小取值（512字节，在本例中表示2个报文段）。为进入慢启动阶段，*cwnd*被置为1个报文段（256字节，与当前值一致）。

当收到SYN和ACK时，没有对这两个变量做任何修改，因为新的数据还没有被确认。

当ACK 257到达时，因为*cwnd*小于等于*ssthresh*，因此仍然处于慢启动阶段，于是将*cwnd*增加256字节。当收到ACK 513时，进行同样的处理。

当ACK 769到达时，我们不再处于慢启动状态，而是进入了拥塞避免状态。新的*cwnd*值按以下方法计算：

$$cwnd \leftarrow cwnd + \frac{segsize \times segsize}{cwnd} + \frac{segsize}{8}$$

考虑到*cwnd*实际上以字节而非以报文段来维护，因此这就是我们前面提到的增加1/*cwnd*。在这个例子中我们计算

$$cwnd \leftarrow 768 + \frac{256 \times 256}{768} + \frac{256}{8}$$

为885字节（使用整数算法）。当下一个ACK 1025到达时，我们计算

$$cwnd \leftarrow 885 + \frac{256 \times 256}{885} + \frac{256}{8}$$

为991字节（在这些表达式中包括了不正确的256/8项来匹配实现计算的数值，正如我们在前面标注的那样）。

报文段号(图21-2)	行为			变量	
	发送	接收	注释	*cwnd*	*ssthresh*
	SYN		初始化	256	65535
	SYN		超时重传	256	512
	ACK	SYN, ACK			
1	1:257(256)				
2		ACK 257	慢启动	512	512
3	257:513(256)				
4	513:769(256)				
5		ACK 513	慢启动	768	512
6	769:1025(256)				
7	1025:1281(256)				
8		ACK 769	cong. avoid	885	512
9	1281:1537(256)				
10		ACK 1025	cong. avoid	991	512
11	1537:1793(256)				
12		ACK 1281	cong. avoid	1089	512

图21-9 拥塞避免的例子

这个*cwnd*持续增加一直到在图21-6所示的发生在10秒左右的第1次重传。图21-10是使用与图21-6相同数据得到的图表，并给出了*cwnd*增加的数值。

本图中*cwnd*的前6个值就是我们为图21-9所计算的数值。在这个图中，要想直观分辨出在慢启动过程中的指数增加和在拥塞避免过程中的线性增加之间的区别是不可能的，因为慢启动的过程太快。

Linux公社 www.linuxidc.com

我们需要解释在重传的3个点上所发生的情况。回想起每个重传都是因为收到3个重复的ACK，表明1个分组丢失了。这就是21.7节的快速重传算法。*ssthresh*立即设置为当重传发生时正在起作用的窗口大小的一半，但是在接收到重复ACK的过程中*cwnd*允许保持增加，这是因为每个重复的ACK表示1个报文段已离开了网络（接收TCP已缓存了这个报文段，等待所缺数据的到达）。这就是快速恢复算法。

与图20-9类似，图21-10表示了*cwnd*和*ssthresh*的数值。第一列上的报文段编号与图21-7对应。

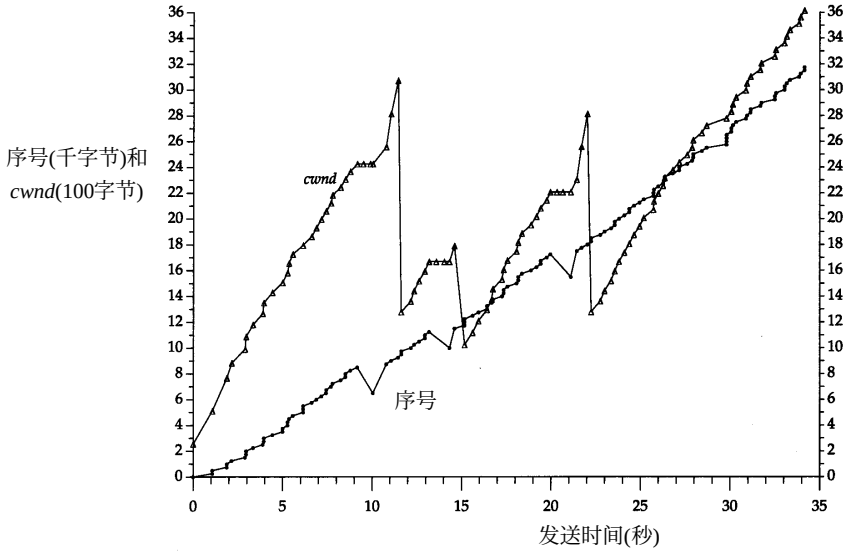


图21-10 当数据被发送时的发送序号和*cwnd*的取值

报文段号(图21-7)	行为			变量	
	发送	接收	注释	*cwnd*	*ssthresh*
58		ACK 6657	新数据的确认	2426	512
59	8705:8961(256)				
60		ACK 6657	重复 ACK #1	2426	512
61		ACK 6657	重复 ACK #2	2426	512
62		ACK 6657	重复 ACK #3 重传	1792	1024
63	6657:6913(256)				
64		ACK 6657	重复 ACK #4	2048	1024
65		ACK 6657	重复 ACK #5	2304	1024
66		ACK 6657	重复 ACK #6	2560	1024
67	8961:9217(256)				
68		ACK 6657	重复 ACK #7	2816	1024
69	9217:9473(256)				
70		ACK 6657	重复 ACK #8	3072	1024
71	9473:9729(256)				
72		ACK 8961	新数据的确认	1280	1024

图21-11 拥塞避免的例子

*cwnd*的值一直持续增加，从图21-9中对应于报文段12的最终取值（1089）到图21-11中对应于报文段58的第一个取值（2426），而*ssthresh*的值则保持不变（512），这是因为在此过程中没有出现过重传。

当最初的2个重复的ACK（报文段60和61）到达时它们被计数，而*cwnd*保持不变（也就是图21-10中处理重传之前的平坦的一段）。然而，当第3个重复的ACK到达时，*ssthresh*被置

Linux公社 www.linuxidc.com

为*cwnd*的一半（四舍五入到报文段大小的下一个倍数），而*cwnd*被置为*ssthresh*加上所收到的重复的ACK数乘以报文段大小（也即1024加上3倍的256），然后发送重传数据。

又有5个重复的ACK到达（报文段64~66, 68和70），每次*cwnd*增加1个报文段长度。最后一个新的ACK（报文段72段）到达时，*cwnd*被置为*ssthresh*（1024）并进入正常的拥塞避免过程。由于*cwnd*小于等于*ssthresh*（现在相等），因此报文段的大小增加到 *cwnd*，取值为1280。当下一个新的ACK到达（没有在图21-11中表示出来）时，*cwnd*大于*ssthresh*，取值为1363。

在快速重传和快速恢复阶段，我们收到报文段 66、68和70中的重复的ACK后才发送新的数据，而不是在接收到报文段 64和65中重复的ACK之后就发送。这是*cwnd*的取值与未被确认的数据大小比较的结果。当报文段 65到达时，*cwnd*为2048，但未被确认的数据有 2304字节（9个报文段：46, 48, 50, 52, 54, 55, 57, 59和63），因此不能发送任何数据。当报文段65到达后，*cwnd*被置为2304，此时我们仍不能进行发送。但是当报文段 66到达时，*cwnd*为2560，所以我们可以发送1个新的数据报文段。类似地，当报文段 68到达时，*cwnd*等于2816，该数值大于未被确认的2560字节的数据大小，因此我们可以发送另 1个新的数据报文段。报文段 70到达时也进行了类似的处理。

在图21-10中的时刻14.3发生下一个重传，也是因为收到了 3个重复的ACK。因此当另一个ACK到达时，可以看到*cwnd*以同样的方式增长，之后降低到 1024。

图21-10中的时刻21.1也是因为收到了重复的 ACK而引起了重传。在重传后收到了 3个重复的ACK，因此观察到*cwnd*增加3个，之后降低到 1280。在传输的后面部分，*cwnd*以线性方式增加到最终值3615。

21.9 按每条路由进行度量

较新的TCP实现在路由表项中维持许多我们在本章已经介绍过的指标。当一个 TCP连接关闭时，如果已经发送了足够多的数据来获得有意义统计资料，且目的结点的路由表项不是一个默认的表项，那么下列信息就保存在路由表项中以备下次使用：被平滑的 RTT、被平滑的均值偏差以及慢启动门限。所谓“足够多的数据”是指 16个窗口的数据，这样就可得到 16个RTT采样，从而使被平滑的RTT过滤器能够集中在正确结果的 5%以内。

而且，管理员可以使用 route(8)命令来设置给定路由的度量：前一段中给出的三个指标以及MT、输出的带宽时延乘积（见第 20.7节）和输入的带宽时延乘积。

当建立一个新的连接时，不论是主动还是被动，如果该连接将要使用的路由表项已经有这些度量的值，则用这些度量来对相应的变量进行初始化。

21.10 ICMP的差错

让我们来看一下TCP是怎样处理一个给定的连接返回的 ICMP的差错。TCP能够遇到的最常见的ICMP差错就是源站抑制、主机不可达和网络不可达。

当前基于伯克利的实现对这些错误的处理是：

- 一个接收到的源站抑制引起拥塞窗口 *cwnd*被置为1个报文段大小来发起慢启动，但是慢启动门限*ssthresh*没有变化，所以窗口将打开直至它或者开放了所有的通路（受窗口大小和往返时间的限制）或者发生了拥塞。
- 一个接收到的主机不可达或网络不可达实际上都被忽略，因为这两个差错都被认为是

Linux公社 www.linuxidc.com

短暂现象。这有可能是由于中间路由器被关闭而导致选路协议要花费数分钟才能稳定到另一个替换路由。在这个过程中就可能发生这两个 ICMP差错中的一个，但是连接并不必被关闭。相反， TCP试图发送引起该差错的数据，尽管最终有可能会超时（回想图21-1中TCP在9分钟内没有放弃的情况）。当前基于伯克利的实现记录发生的ICMP差错，如果连接超时， ICMP差错被转换为一个更合适的的差错码而不是“连接超时”。

早期的BSD实现在任何时候收到一个主机不可达或网络不可达的ICMP差错时会不正确的放弃连接。

一个例子

可以通过在连接中拨号SLIP链路的断开来观察一个ICMP主机不可达的差错是如何被处理的。建立一个从主机slip到主机aix的连接（从扉页前的图中可以看到这个连接经过了我们的拨号SLIP链路）。在建立连接并发送一些数据之后，在路由器 sun和netb之间的SLIP链路被断开，这引起sun上的默认路由表项（见9.2节）被移去。我们希望sun对目的为140.252.1以太网的IP数据报响应ICMP主机不可达。希望观察TCP如何处理这些ICMP差错。

下面是主机slip的交互会话：

```
slip % sock aix echo                    运行sock程序
test line                               键入本行
test line                               和它的回显
                                        此时挂断SLIP链路
another line                            然后键入本行并观察其行为
                                        SLIP链路此时重新建立，
another line                            该行及其回显被交换
line number 3
line number 3
the last line
                                        此时挂断SLIP链路，且没有重新建立
read error: No route to host             TCP最终放弃
```

图21-12显示了在路由器bsdi上截获的tcpdump的相应输出（去掉了连接建立和所有的窗口通告）。我们连接到在主机 aix上的回显服务器并键入“ test line ”（第1行），它被回显（第2行）且回显被确认（第3行），接着我们断开了SLIP链路。

我们键入“ another line ”（第3行之后）并希望看到TCP超时和重传报文。的确，这一行在收到应答前被发送了6次。第4~13行显示了第1次传输和接着的4次重传，每个都产生了一个来自路由器sun的ICMP主机不可达。这正是我们所希望的：从 slip来的IP数据报发往路由器bsdi（这是一个指向sun的默认路由器），并到达检测到链路中断的sun。

在发生这些重传时， SLIP链路又被连通，在第14行的重传被交付。第15行是来自aix的回显，而第16行是对这个回显的确认。

这表明TCP忽略ICMP主机不可达的差错并坚持重传。我们也可以观察到所预期的在每一次重传超时中的指数退避：第1次约为2.5秒，接着乘2（约5秒），乘4（约10秒），乘8（约20秒），乘14（约40秒）。

接着我们键入输入的第3行（“ line number 3 ”）并看到它在第17行被发送，在第18行回显，并在第19行对回显进行确认。

Linux公社 www.linuxidc.com

```
1     0.0                    slip.1035 > aix.echo: P 1:11(10) ack 1
2     0.212271 (  0.2123)    aix.echo > slip.1035: P 1:11(10) ack 11
3     0.310685 (  0.0984)    slip.1035 > aix.echo: . ack 11
```

SLIP链路此时被挂断

```
4   174.758100 (174.4474)    slip.1035 > aix.echo: P 11:24(13) ack 11
5   174.759017 (  0.0009)    sun > slip: icmp: host aix unreachable

6   177.150439 (  2.3914)    slip.1035 > aix.echo: P 11:24(13) ack 11
7   177.151271 (  0.0008)    sun > slip: icmp: host aix unreachable

8   182.150200 (  4.9989)    slip.1035 > aix.echo: P 11:24(13) ack 11
9   182.151189 (  0.0010)    sun > slip: icmp: host aix unreachable

10  192.149671 (  9.9985)    slip.1035 > aix.echo: P 11:24(13) ack 11
11  192.150608 (  0.0009)    sun > slip: icmp: host aix unreachable

12  212.148783 ( 19.9982)    slip.1035 > aix.echo: P 11:24(13) ack 11
13  212.149786 (  0.0010)    sun > slip: icmp: host aix unreachable
```

SLIP链路此时被建立

```
14  252.146774 ( 39.9970)    slip.1035 > aix.echo: P 11:24(13) ack 11
15  252.439257 (  0.2925)    aix.echo > slip.1035: P 11:24(13) ack 24
16  252.505331 (  0.0661)    slip.1035 > aix.echo: . ack 24

17  261.977246 (  9.4719)    slip.1035 > aix.echo: P 24:38(14) ack 24
18  262.158758 (  0.1815)    aix.echo > slip.1035: P 24:38(14) ack 38
19  262.305086 (  0.1463)    slip.1035 > aix.echo: . ack 38
```

SLIP链路此时被挂断

```
20  458.155330 (195.8502)    slip.1035 > aix.echo: P 38:52(14) ack 38
21  458.156163 (  0.0008)    sun > slip: icmp: host aix unreachable

22  461.136904 (  2.9807)    slip.1035 > aix.echo: P 38:52(14) ack 38
23  461.137826 (  0.0009)    sun > slip: icmp: host aix unreachable

24  467.136461 (  5.9986)    slip.1035 > aix.echo: P 38:52(14) ack 38
25  467.137385 (  0.0009)    sun > slip: icmp: host aix unreachable

26  479.135811 ( 11.9984)    slip.1035 > aix.echo: P 38:52(14) ack 38
27  479.136647 (  0.0008)    sun > slip: icmp: host aix unreachable

28  503.134816 ( 23.9982)    slip.1035 > aix.echo: P 38:52(14) ack 38
29  503.135740 (  0.0009)    sun > slip: icmp: host aix unreachable
```

在这里14行输出结果被删除

```
44  1000.219573 ( 64.0959)   slip.1035 > aix.echo: P 38:52(14) ack 38
45  1000.220503 (  0.0009)   sun > slip: icmp: host aix unreachable

46  1064.201281 ( 63.9808)   slip.1035 > aix.echo: R 52:52(0) ack 38
47  1064.202182 (  0.0009)   sun > slip: icmp: host aix unreachable
```

图21-12 TCP对接收到的ICMP主机不可达差错的处理

现在我们希望观察在接收到ICMP主机不可达后，TCP重传并放弃的情况。于是再次断开SLIP链路，之后键入“the last line”，并观察到在TCP放弃之前该行被发送了13次（我们已经从结果中删除了第30~43行，它们是额外的重传）。

然而，我们所观察到的现象是sock程序在最终放弃时打印出来的差错信息：“没有到达主机的路由”。这与Unix的ICMP主机不可达的差错类似（图6-12）。这表明TCP保存了它在连接上收到的ICMP差错，并在最终放弃时打印出该差错，而不是“连接超时”。

最后，注意到第22~46行与第6~14行不同的重传间隔。看起来我们键入的第3行在第17~19行被发送和确认时（无任何重传），TCP更新了它的估计器。最初的重传超时时间现在是3秒，后续取值为6, 12, 24, 48，直至上限64。

Linux公社 www.linuxidc.com

21.11 重新分组

当TCP超时并重传时，它不一定要重传同样的报文段。相反， TCP允许进行重新分组而发送一个较大的报文段，这将有助于提高性能（当然，这个较大的报文段不能够超过接收方声明的MSS）。在协议中这是允许的，因为 TCP是使用字节序号而不是报文段序号来进行识别它所要发送的数据和进行确认。

在实际中，可以很容易地看到这一点。我们使用 sock程序连接到丢弃服务器并键入一行。接着拔掉以太网电缆并再键入一行。当这一行被重传时，键入第 3行。我们预期下一个重传包含第2次和第3次键入的数据。

```
bsdi %sock svr4 discard
hello there                          第一行发送成功
                                     接着我们断开以太网电缆
line number 2                        本行被重传
and 3                                在第2行发送成功之前键入本行
                                     接着重新连接以太网电缆
```

图21-13显示了tcpdump的输出（去掉了连接建立、连接终止以及所有的窗口通告）。

```
1   0.0                  bsdi.1032 > svr4.discard: P 1:13(12) ack 1
2   0.140489 ( 0.1405)   svr4.discard > bsdi.1032: . ack 13

                         此时断开以太网电缆

3  26.407696 (26.2672)   bsdi.1032 > svr4.discard: P 13:27(14) ack 1
4  27.639390 ( 1.2317)   bsdi.1032 > svr4.discard: P 13:27(14) ack 1
5  30.639453 ( 3.0001)   bsdi.1032 > svr4.discard: P 13:27(14) ack 1

                         此时键入第3行

6  36.639653 ( 6.0002)   bsdi.1032 > svr4.discard: P 13:33(20) ack 1
7  48.640131 (12.0005)   bsdi.1032 > svr4.discard: P 13:33(20) ack 1

                         此时重新连接以太网电缆

8  72.640768 (24.0006)   bsdi.1032 > svr4.discard: P 13:33(20) ack 1
9  72.719091 ( 0.0783)   svr4.discard > bsdi.1032: . ack 33
```

图21-13 TCP对数据的重新分组

第1行和第2行显示了头一行（“ hello there ”）被发送及其ACK。接着我们拔掉以太网电缆并键入“ line number 2 ”（14字节，包括换行）。这些数据在第3行被发送，并在第4和第5行被重传。

在第6行重传前，我们键入“ and 3 ”（6个字节，包括换行），并观察到这个重传包括 20个字节：键入的两行。当ACK在第9行到达时，它确认了这20字节的数据。

21.12 小结

本章提供了对TCP超时和重传机制的详细研究。使用的第 1个例子是一个丢失的建立连接的SYN，并观察了在随后的重传和超时中怎样使用指数退避方式。

TCP计算往返时间并使用这些测量结果来维护一个被平滑的 RTT估计器和被平滑的均值偏差估计器。这两个估计器用来计算下一个重传时间。许多实现对每个窗口仅测量一次 RTT。Karn算法在分组丢失时可以不测量 RTT就能解决重传的二义性问题。

Linux公社 www.linuxidc.com

详细例子包括3个丢失的分组，使我们看到TCP的许多实际算法：慢启动、拥塞避免、快速重传和快速恢复。我们也能够使用拥塞窗口和慢启动门限来手工计算 TCP RTT估计器，并将这些值与跟踪输出的实际数据进行比较。

以多种ICMP差错对TCP连接的影响以及TCP怎样允许对数据进行重新分组来结束本章。我们观察到“软”的ICMP差错没有引起TCP连接终止，但这些差错被保存以便在连接非正常中止时能够报告这些软差错。

习题

21.1 在图21-5中第1个超时时间计算为6秒而第2个为12秒。如果初始SYN的确认在12秒超时溢出时还没有到达，则下一次超时在什么时候发生？

21.2 在图21-5后面的讨论中，我们提到计算的超时间隔分别为图 4-5中表示的6、24和48秒。但是如果观察一个从SVR4系统到一个不存在的主机的连接，则超时间隔分别为 6, 12, 24和48秒。请问发生了什么情况？

21.3 按下面的描述比较TCP滑动窗口协议与TFTP的停止等待协议的性能。在本章中，我们在35秒（图21-6）内传输32768字节的数据，其中链路的平均RTT是1.5秒（图21-4）。计算在同样条件下TFTP需要多长时间？

21.4 在第21.7节，我们提到过收到一个重复的 ACK是因为一个报文段丢失或重新进行排序。在21.5节我们看到1个丢失的报文段产生一些重复的 ACK。请画图表示重新排序也会产生一些重复的ACK。

21.5 在图21-6中的时刻28.8和29.8之间有一个显而易见的点，请问这是不是一个重传？

21.6 在21.6节我们提到过，如果目的地址位于一个不同的网络上， 4.3BSD Tahoe版本只执行慢启动。你认为在这里“不同的网络”是由什么决定的？（提示：参看附录 E）。

21.7 在20.2节我们提到过，在正常情况下， TCP每隔一个报文段进行一次确认，但是在图 21-2中，我们看到接收方对每个报文段都进行了确认，请解释其中的原因？

21.8 如果默认路由占优势，那么每路由（per-route）的度量是否真的有用？

Linux公社 www.linuxidc.com

第22章　TCP的坚持定时器

22.1　引言

我们已经看到TCP通过让接收方指明希望从发送方接收的数据字节数（即窗口大小）来进行流量控制。如果窗口大小为0会发生什么情况呢？这将有效地阻止发送方传送数据，直到窗口变为非0为止。

可以在图20-3中看到这种情况。当发送方接收到报文段 9时，它打开被报文段8关闭的窗口并立即开始发送数据。TCP必须能够处理打开此窗口的ACK（报文段9）丢失的情况。ACK的传输并不可靠，也就是说，TCP不对ACK报文段进行确认，TCP只确认那些包含有数据的ACK报文段。

如果一个确认丢失了，则双方就有可能因为等待对方而使连接终止：接收方等待接收数据（因为它已经向发送方通告了一个非 0的窗口），而发送方在等待允许它继续发送数据的窗口更新。为防止这种死锁情况的发生，发送方使用一个坚持定时器 (persist timer)来周期性地向接收方查询，以便发现窗口是否已增大。这些从发送方发出的报文段称为窗口探查 (window probe)。在本章中，我们将讨论窗口探查和坚持定时器，还将讨论与坚持定时器有关的糊涂窗口综合症。

22.2　一个例子

为了观察到实际中的坚持定时器，我们启动一个接收进程。它监听来自客户的连接请求，接受该连接请求，然后在从网上读取数据前休眠很长一段时间。

`sock`程序可以通过指定一个暂停选项`-P`使服务器在接受连接和进行第一次读动作之间进入休眠。我们以这种方式调用服务器：

```
svr4 %sock -i -s -P100000 5555
```

该命令在从网络上读数据之前休眠 100 000秒（27.8小时）。客户运行在主机`bsdi`上，并向服务器的5555端口执行1024字节的写操作。图22-1给出了`tcpdump`的输出结果（我们已经在结果中去掉了连接的建立过程）。

报文段1~13显示的是从客户到服务器的正常的数据传输过程，有 9216个字节的数据填充了窗口。服务器通告窗口大小为 4096字节，且默认的插口缓存大小为 4096字节。但实际上它一共接收了9216字节的数据，这是在SVR4中TCP代码和流子系统(stream subsystem)之间某种形式交互的结果。

在报文段13中，服务器确认了前面 4个数据报文段，然后通告窗口为 0，从而使客户停止发送任何其他的数据。这就引起客户设置其坚持定时器。如果在该定时器时间到时客户还没有接收到一个窗口更新，它就探查这个空的窗口以决定窗口更新是否丢失。由于服务器进程处于休眠状态，所以TCP缓存9216字节的数据并等待应用进程读取。

请注意客户发出的窗口探查之间的时间间隔。在收到一个大小为 0的窗口通告后的第1个

Linux公社 www.linuxidc.com

（报文段14）间隔为4.949秒，下一个（报文段16）间隔是4.996秒，随后的间隔分别约为6, 12, 24, 48和60秒。

```
 1   0.0                 bsdi.1027 > svr4.5555: P 1:1025(1024) ack 1 win 4096
 2   0.191961 ( 0.1920)  svr4.5555 > bsdi.1027: . ack 1025 win 4096
 3   0.196950 ( 0.0050)  bsdi.1027 > svr4.5555: . 1025:2049(1024) ack 1 win 4096
 4   0.200340 ( 0.0034)  bsdi.1027 > svr4.5555: . 2049:3073(1024) ack 1 win 4096
 5   0.207506 ( 0.0072)  svr4.5555 > bsdi.1027: . ack 3073 win 4096
 6   0.212676 ( 0.0052)  bsdi.1027 > svr4.5555: . 3073:4097(1024) ack 1 win 4096
 7   0.216113 ( 0.0034)  bsdi.1027 > svr4.5555: P 4097:5121(1024) ack 1 win 4096
 8   0.219997 ( 0.0039)  bsdi.1027 > svr4.5555: P 5121:6145(1024) ack 1 win 4096
 9   0.227882 ( 0.0079)  svr4.5555 > bsdi.1027: . ack 5121 win 4096
10   0.233012 ( 0.0051)  bsdi.1027 > svr4.5555: P 6145:7169(1024) ack 1 win 4096
11   0.237014 ( 0.0040)  bsdi.1027 > svr4.5555: P 7169:8193(1024) ack 1 win 4096
12   0.240961 ( 0.0039)  bsdi.1027 > svr4.5555: P 8193:9217(1024) ack 1 win 4096
13   0.402143 ( 0.1612)  svr4.5555 > bsdi.1027: . ack 9217 win 0

14   5.351561 ( 4.9494)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
15   5.355571 ( 0.0040)  svr4.5555 > bsdi.1027: . ack 9217 win 0

16  10.351714 ( 4.9961)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
17  10.355670 ( 0.0040)  svr4.5555 > bsdi.1027: . ack 9217 win 0

18  16.351881 ( 5.9962)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
19  16.355849 ( 0.0040)  svr4.5555 > bsdi.1027: . ack 9217 win 0

20  28.352213 (11.9964)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
21  28.356178 ( 0.0040)  svr4.5555 > bsdi.1027: . ack 9217 win 0

22  52.352874 (23.9967)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
23  52.356839 ( 0.0040)  svr4.5555 > bsdi.1027: . ack 9217 win 0

24 100.354224 (47.9974)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
25 100.358207 ( 0.0040)  svr4.5555 > bsdi.1027: . ack 9217 win 0

26 160.355914 (59.9977)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
27 160.359835 ( 0.0039)  svr4.5555 > bsdi.1027: . ack 9217 win 0

28 220.357575 (59.9977)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
29 220.361668 ( 0.0041)  svr4.5555 > bsdi.1027: . ack 9217 win 0

30 280.359254 (59.9976)  bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
31 280.363315 ( 0.0041)  svr4.5555 > bsdi.1027: . ack 9217 win 0
```

图22-1 坚持定时器探查一个0大小窗口的例子

为什么这些间隔总是比5、6、12、24、48和60小一个零点几秒呢？因为这些探查被TCP的500 ms定时器超时例程所触发。当定时器时间到时，就发送窗口探查，并大约在4 ms之后收到一个应答。接收到应答使得定时器被重新启动，但到下一个时钟滴答之间的时间则约为500减4 ms。

计算坚持定时器时使用了普通的 TCP指数退避。对一个典型的局域网连接，首次超时时间算出来是1.5秒，第2次的超时值增加一倍，为 3秒，再下次乘以4为6秒，之后再乘以8为12秒等。但是坚持定时器总是在5~60秒之间，这与我们在图22-1中观察到的现象一致。

窗口探查包含一个字节的数据（序号为 9217）。TCP总是允许在关闭连接前发送一个字节的数据。请注意，尽管如此，所返回的窗口为 0的ACK并不是确认该字节（它们确认了包括9216在内的所有数据），因此这个字节被持续重传。

坚持状态与第21章中介绍的重传超时之间一个不同的特点就是 TCP从不放弃发送窗口探查。这些探查每隔60秒发送一次，这个过程将持续到或者窗口被打开，或者应用进程使用的连接被终止。

22.3 糊涂窗口综合症

基于窗口的流量控制方案，如 TCP所使用的，会导致一种被称为“糊涂窗口综合症 SWS

Linux公社 www.linuxidc.com

(Silly Window Syndrome）”的状况。如果发生这种情况，则少量的数据将通过连接进行交换，而不是满长度的报文段[Clark 1982]。

该现象可发生在两端中的任何一端：接收方可以通告一个小的窗口（而不是一直等到有大的窗口时才通告），而发送方也可以发送少量的数据（而不是等待其他的数据以便发送一个大的报文段）。可以在任何一端采取措施避免出现糊涂窗口综合症的现象。

1) 接收方不通告小窗口。通常的算法是接收方不通告一个比当前窗口大的窗口（可以为0），除非窗口可以增加一个报文段大小（也就是将要接收的 MSS）或者可以增加接收方缓存空间的一半，不论实际有多少。

2) 发送方避免出现糊涂窗口综合症的措施是只有以下条件之一满足时才发送数据： (a)可以发送一个满长度的报文段；(b)可以发送至少是接收方通告窗口大小一半的报文段；(c)可以发送任何数据并且不希望接收 ACK（也就是说，我们没有还未被确认的数据）或者该连接上不能使用Nagle算法（见第19.4节）。

条件(b)主要对付那些总是通告小窗口（也许比 1个报文段还小）的主机，条件 (c)使我们在有尚未被确认的数据（正在等待被确认）以及在不能使用 Nagle算法的情况下，避免发送小的报文段。如果应用进程在进行小数据的写操作（例如比该报文段还小），条件(c)可以避免出现糊涂窗口综合症。

这三个条件也可以让我们回答这样一个问题：在有尚未被确认数据的情况下，如果 Nagle算法阻止我们发送小的报文段，那么多小才算是小呢？从条件 (a)中可以看出所谓“小”就是指字节数小于报文段的大小。条件(b)仅用来对付较老的、原始的主机。

步骤2中的条件(b)要求发送方始终监视另一方通告的最大窗口大小，这是一种发送方猜测对方接收缓存大小的企图。虽然在连接建立时接收缓存的大小可能会减小，但在实际中这种情况很少见。

一个例子

现在我们通过仔细查看一个详细的例子来观察实际避免出现糊涂窗口综合症的情况，该例子也包括了坚持定时器。我们将在发送主机 sun上运行sock程序，并向网络写6个1024字节的数据。

```
sun % sock -i -n6 bsdi 7777
```

但是在主机bsdi的接收过程中我们加入一些暂停。在第1次读数据前暂停4秒，之后每次读之前暂停2秒。而且，接收方进行的是256字节的读操作：

```
bsdi %sock -i -s -P4 -p2 -r256 7777
```

最初的暂停是为了让接收缓存被填满，迫使发送方停止发送。随后由于接收方从网络上进行了一些小数目的读取，我们预期能看到接收方采取的避免糊涂窗口综合症的措施。

图22-2是传输6144字节数据的时间系列（我们去掉了连接建立过程）。

我们还需要跟踪在每个时间点上读取数据时应用程序的运行情况、当前正在接收缓存中的数据的序号以及接收缓存中可用空间的大小。图 22-3显示了所发生的每件事情。

图22-3中的第1列是每个行为的相对时间点。那些带有3位小数点的时间是从tcpdump的输出结果（图22-2）中得到的，而小数点部分为 99的则是在接收服务器上产生行为的估计时间（使这些在接收方的估计时间包含一秒的99%仅与图22-2中的报文段20和22有关，它们是我们能

Linux公社 www.linuxidc.com

够从tcpdump的输出结果中看到的由接收主机超时引起的仅有的两个事件。而在主机 bsdi上观察到的其他分组，则是由接收到来自发送方的一个报文段所引起的。这同样是有意义的，因为这就使我们可以将最初的4秒暂停刚好放置在发送方发送第1个数据报文段的时间0前面。这是接收方在连接建立过程中收到它的SYN的ACK之后将要获得控制权的大致时间）。

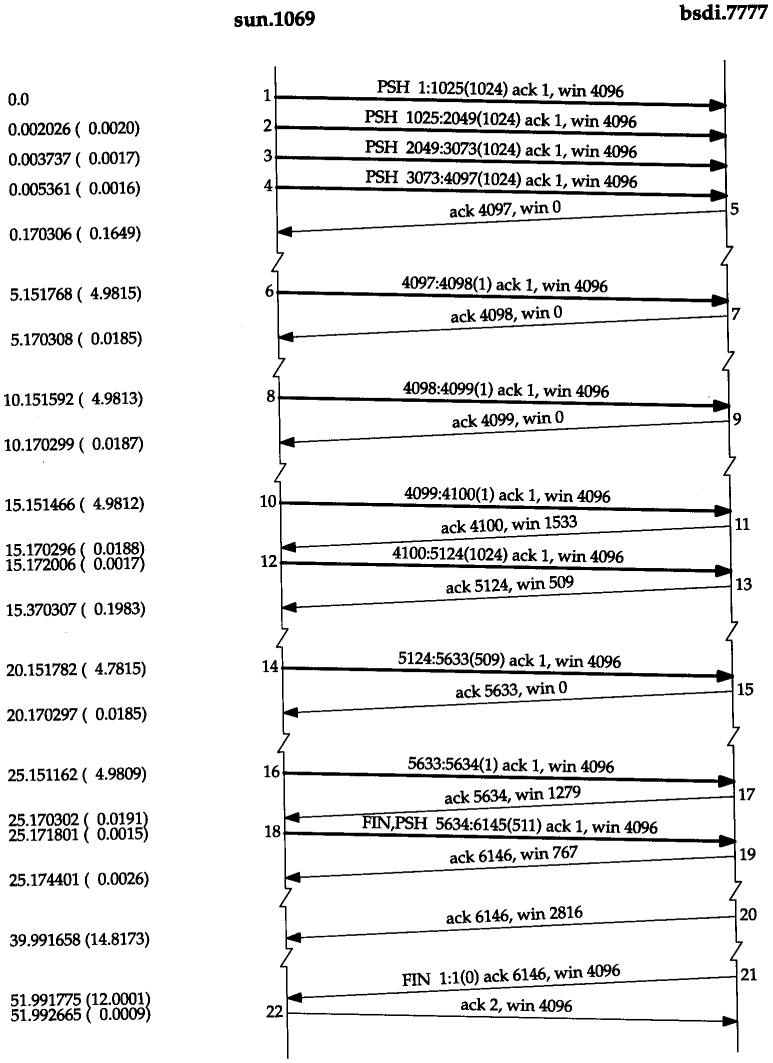


图22-2 显示接收方避免出现糊涂窗口综合症的时间系列

当接收到来自发送方的数据时，接收方缓存中的数据增加，而当应用进程从缓存中读取数据时，数据就减少。接下来我们关注的是接收方发给发送方的窗口通告以及这些窗口通告是什么。这样就可以使我们看到接收方是如何避免糊涂窗口综合症的。

前4个数据报文段及其ACK（报文段1~5）表示发送方正在填充接收方的缓存。在那个时刻发送方停止了发送，但仍然有更多的数据需要发送。它将自己的坚持定时器置为最小值5分钟。

当坚持定时器时间到时，就发送出1个字节的数据（报文段6）。接收的应用进程已经从接收缓存中读取了256字节的数据（在时刻3.99），因此这个字节被接受并被确认（报文段7段）。但是通告窗口仍为0，由于接收方仍然没有足够的空间来接收一个满长度的报文，或者不能腾

Linux公社 www.linuxidc.com

出缓存空间的一半。这就是接收方的糊涂窗口避免措施。

时间	报文段号(图22-2)	行为			接收缓冲区	
		发送TCP	接收TCP	应用	数据	可用的
0.000	1	1:1025(1024)			1024	3072
0.002	2	1025:2049(1024)			2048	2048
0.003	3	2049:3073(1024)			3072	1024
0.005	4	3073:4097(1024)			4096	0
0.170	5		ACK 4097, win 0			
3.99				读取256	3840	256
5.151	6	4097:4098(1)			3841	255
5.17	7		ACK 4098, win 0			
5.99				读取256	3585	511
7.99				读取256	3329	767
9.99				读取256	3073	1023
10.151	8	4098:4099(1)			3074	1022
10.170	9		ACK 4099, win 0			
11.99				读取256	2818	1278
13.99				读取256	2562	1534
15.151	10	4099:4100(1)			2563	1533
15.170	11		ACK 4100, win 1533			
15.172	12	4100:5124(1024)			3587	509
15.370	13		ACK 5124, win 509			
15.99				读取256	3331	765
17.99				读取256	3075	1021
19.99				读取256	2819	1277
20.151	14	5124:5633(509)			3328	768
20.170	15		ACK 5633, win 0			
21.99				读取256	3072	1024
23.99				读取256	2816	1280
25.151	16	5633:5634(1)			2817	1279
25.170	17		ACK 5634, win 1279			
25.171	18	5634:6145(511)			3328	768
25.174	19		ACK 6146, win 767			
25.99				读取256	3072	1024
27.99				读取256	2816	1280
29.99				读取256	2560	1536
31.99				读取256	2304	1792
33.99				读取256	2048	2048
35.99				读取256	1792	2304
37.99				读取256	1536	2560
39.99				读取256	1280	2816
39.99	20		ACK 6146, win 2816			
41.99				读取256	1024	3072
43.99				读取256	768	3328
45.99				读取256	512	3584
47.99				读取256	256	3840
49.99				读取256	0	4096
51.99				读取256(EOF)	0	4096
51.991	21		ACK 6146, win 4096			
51.992	22	ACK 2				

图22-3 接收方避免出现糊涂窗口综合症的事件序列

发送方的坚持定时器被复位，并在5秒后再次到时（在时刻10.151）。然后又发送一个字节并被确认（报文段8和9），而接收方的缓存空间还不够用（1022字节），使得通告窗口为0。

发送方的坚持定时器在时刻15.151再次时间到，又发送了另一个字节并被确认（报文段10和11）。这一次由于接收方有1533字节的有效缓存空间，因此通告了一个非0窗口。发送方立即利用这个窗口发送了1024字节的数据（报文段12）。对这1024字节数据的确认（报文段13）通告其窗口为509字节。这看起来与我们在前面看到的小窗口通告相抵触。

在这里之所以发生这种情况，是因为报文段11段通告了一个大小为1533字节的窗口，而发送方只使用了其中的1024字节。如果在报文段13中的ACK通告其窗口为0，就会违反窗口的右边沿不能向左边沿移动而导致窗口收缩的TCP原则（见第20.3节）。这就是为什么必须通告

Linux公社 www.linuxidc.com

一个509字节的窗口的原因。

接下来我们看到发送方没有立即向这个小窗口发送数据。这就是发送方采取的糊涂窗口避免策略。相反，它等待另一个坚持定时器在时刻 20.151到时间，并在该时刻发送 509字节的数据。尽管它最终还是发送了一个长度为 509字节的小数据段，但在发送前它等待了 5秒钟，看是否会有一个ACK到达，以便可以将窗口开得更大。这 509字节的数据使得接收缓存仅剩下768字节的有效空间，因此接收方通告窗口为 0（报文段15）。

坚持定时器在时刻25.151再次到时间，发送方发送 1个字节，于是接收缓存中有 1279字节的可用空间，这就是在报文段 17所通告的窗口大小。

发送方只有另外的511个字节的数据需要发送，因此在收到 1279的窗口通告后立刻发送了这些数据（报文段 18）。这个报文段也带有 FIN标志。接收方确认数据和 FIN，并通告窗口大小为767（见习题22.2）。

由于发送应用进程在执行完 6个1024字节的写操作后发出关闭命令，发送方的连接从ESTABLISHED状态转变到FIN_WAIT_1状态，再到FIN_WAIT_2状态（见图18-12）。它一直处于这个状态，直到收到对方的 FIN。在这个状态上没有设置定时器（回忆我们在 18.6节结束时的讨论），因为它在报文段 18中发送的FIN被报文段19确认。这就是为什么我们看到发送方直到接收到FIN（报文段21）为止没有发送其他任何数据的原因。

接收应用进程继续每隔 2秒从接收缓存区中读取 256个字节的数据。为什么在时刻 39.99发送ACK（报文段20）呢？这是因为应用进程在时刻 39.99读取数据时，接收缓存中的可用空间已经从原来通告的 767（报文段19）变为2816，这相当于接收缓存中增加了额外的 2049字节的空间。回忆本节开始讲的第 1个规则，因为现在接收缓存已经增加了其空间的一半，因此接收方现在发送窗口更新。这意味着每次当应用进程从 TCP的接收缓存中读取数据时，接收的 TCP将检查是否需要更新发送窗口。

应用进程在时间51.99发出最后一个读操作，然后收到一个文件结束标志，因为缓存已经变空。这就导致了最后两个完成连接终止的报文段（报文段 21和22）的发送。

22.4 小结

在连接的一方需要发送数据但对方已通告窗口大小为0时，就需要设置TCP的坚持定时器。发送方使用与第21章类似的重传间隔时间，不断地探查已关闭的窗口。这个探查过程将一直持续下去。

当运行一个例子来观察坚持定时器时，我们还观察到了 TCP的避免出现糊涂窗口综合症的现象。这就是使 TCP避免通告小的窗口大小或发送小的报文段。在我们的例子中，可以观察到发送方和接收方为避免糊涂窗口综合症所使用的策略。

习题

22.1 在图22-3中注意到所有确认（报文段5、7、9、11、13、15和17）的发送时刻为：0.170, 5.170、10.170、15.170、20.170和25.170，还注意到在接收数据和发送ACK之间的时间差分别为：164.5、18.5、18.7、18.8、198.3、18.5和19.1 ms。试解释可能会发生的情况。

22.2 在图22-3中的时刻25.174，发送出一个767字节的通告窗口，而在接收缓存中有 768字节的可用空间。为什么相差1个字节？

Linux公社 www.linuxidc.com

第23章　TCP的保活定时器

23.1　引言

许多TCP/IP的初学者会很惊奇地发现可以没有任何数据流通过一个空闲的 TCP连接。也就是说，如果TCP连接的双方都没有向对方发送数据，则在两个TCP模块之间不交换任何信息。例如，没有可以在其他网络协议中发现的轮询。这意味着我们可以启动一个客户与服务器建立一个连接，然后离去数小时、数天、数个星期或者数月，而连接依然保持。中间路由器可以崩溃和重启，电话线可以被挂断再连通，但是只要两端的主机没有被重启，则连接依然保持建立。

这意味着两个应用进程 —— 客户进程或服务器进程 —— 都没有使用应用级的定时器来检测非活动状态，而这种非活动状态可以导致应用进程中的任何一个终止其活动。回想在第10.7节末尾曾提到过的BGP每隔30秒就向对端发送一个应用的探查，就是独立于 TCP的保活定时器之外的应用定时器。

然而，许多时候一个服务器希望知道客户主机是否崩溃并关机或者崩溃又重新启动。许多实现提供的保活定时器可以提供这种能力。

> 保活并不是TCP规范中的一部分。Host Requirements RFC提供了3个不使用保活定时器的理由：(1) 在出现短暂差错的情况下，这可能会使一个非常好的连接释放掉；(2) 它们耗费不必要的带宽；(3) 在按分组计费的情况下会在互联网上花掉更多的钱。然而，许多实现提供了保活定时器。

保活定时器是一个有争论的功能。许多人认为如果需要，这个功能不应该在 TCP中提供，而应该由应用程序来完成。这是应当认真对待的一些问题之一，因为在这个论题上有些人表达出了很大的热情。

在连接两个端系统的网络出现临时故障的时候，保活选项会引起一个实际上很好的连接终止。例如，如果在一个中间路由器崩溃并重新启动时发送保活探查，那么 TCP会认为客户的主机已经崩溃，而实际上所发生的并非如此。

保活功能主要是为服务器应用程序提供的。服务器应用程序希望知道客户主机是否崩溃，从而可以代表客户使用资源。许多版本的 Rlogin和Telnet服务器默认使用这个选项。

一个说明现在需要使用保活功能的常见例子是当个人计算机用户使用 TCP/IP向一个使用Telnet的主机注册时。如果在一天结束时，他们仅仅关闭了电源而没有注销，那么便会留下一个半开放的连接。在图18-16中，我们看到通过一个半开放连接发送数据会导致返回一个复位，但那是在来自正在发送数据的客户端。如果客户已经消失了，使得在服务器上留下一个半开放连接，而服务器又在等待来自客户的数据，则服务器将永远等待下去。保活功能就是试图在服务器端检测到这种半开放的连接。

Linux公社 www.linuxidc.com

23.2 描述

在这个描述中，我们称使用保活选项的一端为服务器，而另一端则为客户。并没有什么使客户不能使用这个选项，但通常都是服务器设置这个功能。如果双方都特别需要了解对方是否已经消失，则双方都可以使用这个选项（在 29章我们将看到NFS使用TCP时，客户和服务器都设置了这个选项。但在第 26章讲到Telnet和Rlogin时，只有服务器设置了这个选项，而客户则没有）。

如果一个给定的连接在两个小时之内没有任何动作，则服务器就向客户发送一个探查报文段（我们将在随后的例子中看到这个探查报文段看起来像什么）。客户主机必须处于以下 4个状态之一。

1) 客户主机依然正常运行，并从服务器可达。客户的 TCP响应正常，而服务器也知道对方是正常工作的。服务器在两小时以后将保活定时器复位。如果在两个小时定时器到时间之前有应用程序的通信量通过此连接，则定时器在交换数据后的未来 2小时再复位。

2) 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的 TCP都没有响应。服务器将不能够收到对探查的响应，并在 75秒后超时。服务器总共发送 10个这样的探查，每个间隔 75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

3) 客户主机崩溃并已经重新启动。这时服务器将收到一个对其保活探查的响应，但是这个响应是一个复位，使得服务器终止这个连接。

4) 客户主机正常运行，但是从服务器不可达。这与状态 2相同，因为TCP不能够区分状态4与状态2之间的区别，它所能发现的就是没有收到探查的响应。

服务器不用关注客户主机被关闭和重新启动的情况（这指的是一个操作员的关闭，而不是主机崩溃）。当系统被操作员关闭时，所有的应用进程也被终止（也就是客户进程），这会使客户的TCP在连接上发出一个 FIN。接收到FIN将使服务器的TCP向服务器进程报告文件结束，使服务器可以检测到这个情况。

在第1种情况下，服务器的应用程序没有感觉到保活探查的发生。 TCP层负责一切。这个过程对应用程序都是透明的，直至第 2、3或4种情况发生。在这三种情况下，服务器应用程序将收到来自它的 TCP的差错报告（通常服务器已经向网络发出了读操作请求，然后等待来自客户的数据。如果保活功能返回一个差错，则该差错将作为读操作的返回值返回给服务器）。在第2种情况下，差错是诸如“连接超时”之类的信息，而在第 3种情况则为“连接被对方复位”。第4种情况看起来像是连接超时，也可根据是否收到与连接有关的 ICMP差错来返回其他的差错。在下一节中我们将观察这 4种情况。

> 一个被人们不断讨论的关于保活选项的问题就是两个小时的空闲时间是否可以改变。通常他们希望该数值可以小得多，处在分钟的数量级。正如我们在附录 E看到的，这个值通常可以改变，但是在该附录所描述的所有系统中，保活间隔时间是系统级的变量，因此改变它会影响到所有使用该功能的用户。
>
> Host Requirements RFC提到一个实现可提供保活的功能，但是除非应用程序指明要这样，否则就不能使用该功能。而且，保活间隔必须是可配置的，但是其默认值必须不小于两个小时。

Linux公社 www.linuxidc.com

23.3 保活举例

现在详细讨论前一节提到的第 2、3和4种情况。我们将在使用这个选项的情况下检查所交换的分组。

23.3.1 另一端崩溃

首先观察另一端崩溃且没有重新启动的情况下所发生的现象。为模拟这种情况，我们采用如下步骤：

- 在客户（主机bsdi上运行的sock程序）和主机svr4上的标准回显服务器之间建立一个连接。客户使用-K选项使能保活功能。
- 验证数据可以通过该连接。
- 观察客户TCP每隔2小时发送保活分组，并观察被服务器的 TCP确认。
- 将以太网电缆从服务器上拔掉直到这个例子完成，这会使客户认为服务器主机已经崩溃。
- 我们预期服务器在断定连接已中断前发送 10个间隔为75秒的保活探查。

这里是客户端的交互输出结果：

```
bsdi %sock -K svr4 echo          -K是保活选项
hello, world                     开始时键入本行以验证连接有效
hello, world                     和看到回显
                                 4小时后断开以太网电缆
read error: Connection timed out这发生在启动后约6小时10分钟
```

图23-1显示的是tcpdump的输出结果（已经去掉了连接建立和窗口通告）。

```
 1      0.0                  bsdi.1055 > svr4.echo: P 1:14(13) ack 1
 2      0.006105 (   0.0061) svr4.echo > bsdi.1055: P 1:14(13) ack 14
 3      0.093140 (   0.0870) bsdi.1055 > svr4.echo: . ack 14

 4   7199.972793 (7199.8797) arp who-has svr4 tell bsdi
 5   7199.974878 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
 6   7199.975741 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
 7   7199.979843 (   0.0041) svr4.echo > bsdi.1055: . ack 14

 8  14400.134330 (7200.1545) arp who-has svr4 tell bsdi
 9  14400.136452 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
10  14400.137391 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
11  14400.141408 (   0.0040) svr4.echo > bsdi.1055: . ack 14

12  21600.318309 (7200.1769) arp who-has svr4 tell bsdi
13  21675.320373 (  75.0021) arp who-has svr4 tell bsdi
14  21750.322407 (  75.0020) arp who-has svr4 tell bsdi
15  21825.324460 (  75.0021) arp who-has svr4 tell bsdi
16  21900.436749 (  75.1123) arp who-has svr4 tell bsdi
17  21975.438787 (  75.0020) arp who-has svr4 tell bsdi
18  22050.440842 (  75.0021) arp who-has svr4 tell bsdi
19  22125.432883 (  74.9920) arp who-has svr4 tell bsdi
20  22200.434697 (  75.0018) arp who-has svr4 tell bsdi
21  22275.436788 (  75.0021) arp who-has svr4 tell bsdi
```

图23-1 决定一个主机已经崩溃的保活分组

客户在第1、2和3行向服务器发送“ Hello, world ”并得到回显。第4行是第一个保活探查，发生在两个小时以后（ 7200秒）。在第6行的TCP报文段能够发送之前，首先观察到的是一个ARP请求和一个ARP应答。第6行的保活探查引出来自另一端的响应（第 7行）。两个小时以后，在第7和8行发生了同样的分组交换过程。

Linux公社 www.linuxidc.com

如果能够观察到第6和第10行的保活探查中的所有字段，我们就会发现序号字段比下一个将要发送的序号字段小1（在本例中，当下一个为14时，它就是13）。但是因为报文段中没有数据，tcpdump不能打印出序号字段（它仅能够打印出设置了SYN、FIN或RST标志的空数据的序号）。正是接收到这个不正确的序号，才导致服务器的TCP对保活探查进行响应。这个响应告诉客户，服务器下一个期望的序号是14。

> 一些基于4.2BSD的旧的实现不能够对这些保活探查进行响应，除非报文段中包含数据。某些系统可以配置成发送一个字节的无用数据来引出响应。这个无用数据是无害的，因为它不是所期望的数据（这是接收方前一次接收并确认的数据），因此它会被接收方丢弃。其他一些系统在探查的前半部分发送4.3BSD格式的报文段（不包含数据），如果没有收到响应，在后半部分则切换为4.2BSD格式的报文段。

接着我们拔掉电缆，并期望两个小时的再一次探查失败。当这下一个探查发生时，注意到从来没有看到电缆上出现TCP报文段，这是因为主机没有响应ARP请求。在放弃之前，我们仍可以观察到客户每隔75秒发送一个探查，一共发送了10次。从交互式脚本可以看到返回给客户进程的差错码被TCP转换为“连接超时”，这正是实际所发生的。

23.3.2 另一端崩溃并重新启动

在这个例子中，我们可以观察到当客户崩溃并重新启动时发生的情况。最初的环境与前一个例子相似，但是在我们验证连接有效之后，我们将服务器从以太网上断开，重新启动，然后再连接到网络上。我们希望看到下一个保活探查产生一个来自服务器的复位，因为现在服务器不知道关于这个连接的任何信息。这是交互会话的过程：

```
bsdi %sock -K svr4 echo          -K使保活选项有效
hi, there                        键入这行以验证连接有效
hi, there                        这是来自另一端的回显
                                 从以太网断连后，服务器这时重新启动
read error: Connection reset by peer
```

图23-2显示的是tcpdump的输出结果（已经去掉了连接建立和窗口通告）。

```
 1     0.0                    bsdi.1057 > svr4.echo: P 1:10(9) ack 1
 2     0.006406 (   0.0064)   svr4.echo > bsdi.1057: P 1:10(9) ack 10
 3     0.176922 (   0.1705)   bsdi.1057 > svr4.echo: . ack 10

 4  7200.067151 (7199.8902)   arp who-has svr4 tell bsdi
 5  7200.069751 (   0.0026)   arp reply svr4 is-at 0:0:c0:c2:9b:26
 6  7200.070468 (   0.0007)   bsdi.1057 > svr4.echo: . ack 10
 7  7200.075050 (   0.0046)   svr4.echo > bsdi.1057: R 1135563275:1135563275(0)
```

图23-2 另一端崩溃并重启时保活的例子

我们建立了连接，并从客户发送9个字节的数据到服务器（第1~3行）。两个小时之后，客户发送第1个保活探查，其响应是一个来自服务器的复位。客户应用进程打印出“连接被对端复位”的差错，这是有意义的。

23.3.3 另一端不可达

在这个例子中，客户没有崩溃，但是在保活探查发送后的10分钟内无法到达，可能是一个中间路由器已经崩溃，或一条电话线临时出现故障，或发生了其他一些类似的情况。

Linux公社 www.linuxidc.com

为了仿真这个例子，我们从主机 slip经过一个拨号 SLIP链路与主机 vangogh.cs.berkeley.edu建立一个连接，然后断掉链路。这里是交互输出的结果：

```
bsdi %sock -K vangogh.cs.berkeley.edu echo
testing                                          我们键入这行
testing                                          看到这行的回显
                                                 在某个时刻这条SLIP链路被断开
read error: No route to host
```

图23-3显示了在路由器bsdi上收集到的tcpdump输出结果（已经去掉了连接建立和窗口通告）。

```
 1      0.0                  slip.1056 > vangogh.echo: P 1:9(8) ack 1
 2      0.277669 (   0.2777) vangogh.echo > slip.1056: P 1:9(8) ack 9
 3      0.424423 (   0.1468) slip.1056 > vangogh.echo: . ack 9

 4   7200.818081 (7200.3937) slip.1056 > vangogh.echo: . ack 9
 5   7201.243046 (   0.4250) vangogh.echo > slip.1056: . ack 9

 6  14400.688106 (7199.4451) slip.1056 > vangogh.echo: . ack 9
 7  14400.689261 (   0.0012) sun > slip: icmp: net vangogh unreachable

 8  14475.684360 (  74.9951) slip.1056 > vangogh.echo: . ack 9
 9  14475.685504 (   0.0011) sun > slip: icmp: net vangogh unreachable

                                   删除14行
24  15075.759603 (  75.1008) slip.1056 > vangogh.echo: R 9:9(0) ack 9
25  15075.760761 (   0.0012) sun > slip: icmp: net vangogh unreachable
```

图23-3 当另一端不可达时的保活例子

我们与以前一样开始讨论这个例子：第 1~3行证实连接是有效的。两个小时之后的第 1个保活探查是正常的（第4、5行），但是在两个小时后发生下一个探查之前，我们断开在路由器sun和netb之间的SLIP连接（拓扑结构参见封）。

第6行的保活探查引发一个来自路由器 sun的ICMP网络不可达的差错。正如我们在第21.10节描述的那样，对于主机slip上接收的TCP而言，这只是一个软差错。它报告收到了一个ICMP差错，但是差错的接收者并没有终止这个连接。在发送主机最终放弃之前，一共发送了9个保活探查，间隔为 75秒。这时返回给应用进程的差错产生了一个不同的报文：“没有到达主机的路由”。我们在图6-12看到这对应于ICMP网络不可达的差错。

23.4 小结

正如我们在前面提到的，对保活功能是有争议的。协议专家继续在争论该功能是否应该归入运输层，或者应当完全由应用层来处理。

在连接空闲两个小时后，在一个连接上发送一个探查分组来完成保活功能。可能会发生 4种不同的情况：对端仍然运行正常、对端已经崩溃、对端已经崩溃并重新启动以及对端当前无法到达。我们使用一个例子来观察每一种情况，并观察到在最后三个条件下返回的不同差错。

在前两个例子中，如果没有提供这种功能，并且也没有应用层的定时器，则客户将永远无法知道对端已经崩溃或崩溃并重新启动。可是在最后一个例子中，两端都没有发生差错，只是它们之间的连接临时中断。我们在使用保活时必须关注这个限制。

习题

23.1 列出保活功能的一些优点。

23.2 列出保活功能的一些缺点。

Linux公社 www.linuxidc.com

第24章　TCP的未来和性能

24.1　引言

TCP已经在从1200 b/s的拨号SLIP链路到以太数据链路上运行了许多年。在 80年代和90年代初期，以太网是运行 TCP/IP最主要的数据链路方式。虽然 TCP在比以太网速率高的环境（如T2电话线、FDDI及千兆比网络）中也能够正确运行，但在这些高速率环境下，　TCP的某些限制就会暴露出来。

本章讨论TCP的一些修改建议，这些建议可以使 TCP在高速率环境中获得最大的吞吐量。首先要讨论前面已经碰到过的路径 MTU发现机制，本章主要关注它如何与 TCP协同工作。这个机制通常可以使TCP为非本地的连接使用大于536字节的MTU，从而增加吞吐量。

接着介绍长肥管道(long fat pipe)，也就是那些具有很大的带宽时延乘积的网络，以及 TCP在这些网络上所具有的局限性。为处理长肥管道，我们描述两个新的　TCP选项：窗口扩大选项（用来增加 TCP的最大窗口，使之超过 65535字节）和时间戳选项。后面这个选项可以使TCP对报文段进行更加精确的 RTT测量，还可以在高速率下对可能发生的序号回绕提供保护。这两个选项在RFC 1323 [Jacobson, Braden, and Borman 1992]中进行定义。

我们还将介绍建议的T/TCP，这是为增加事务功能而对 TCP进行的修改。通信的事务模式以客户的请求将被服务器应答的响应为主要特征。这是客户服务器计算的常见模型。　T/TCP的目的就是减少两端交换的报文段数量，避免三次握手和使用 4个报文段进行连接的关闭，从而使客户可以在一个RTT和处理请求所必需的时间内收到服务器的应答。

这些新选项（路径 MTU发现、窗口扩大选项、时间戳选项和 T/TCP）中令人印象最深刻的就是它们与现有的 TCP实现能够向后兼容，即包括这些新选项的系统仍然可以与原有的旧系统进行交互。除了在一个 ICMP报文中为路径MTU发现增加了一个额外字段之外，这些新的选项只需要在那些需要使用它们的端系统中进行实现。

我们以介绍近来发表的有关 TCP性能的图例作为本章的结束。

24.2　路径MTU发现

在2.9节我们描述了路径MTU的概念。这是当前在两个主机之间的路径上任何网络上的最小MTU。路径MTU发现在IP首部中继承并设置“不要分片（DF）”比特，来发现当前路径上的路由器是否需要对正在发送的 IP数据报进行分片。在 11.6节我们观察到如果一个待转发的 IP数据报被设置 DF比特，而其长度又超过了 MTU，那么路由器将返回 ICMP不可达的差错。在11.7节我们显示了某版本的 `traceroute`程序使用该机制来决定目的地的路径 MTU。在11.8节我们看到 UDP是怎样处理路径 MTU发现的。在本节我们将讨论这个机制是如何按照　RFC 1191 [Mogul and Deering 1990]中规定的那样在TCP中进行使用的。

在本书的多种系统（参看序言）中只有Solaris 2.x支持路径MTU发现。

Linux公社 www.linuxidc.com

TCP的路径MTU发现按如下方式进行：在连接建立时， TCP使用输出接口或对端声明的MSS中的最小MTU作为起始的报文段大小。路径 MTU发现不允许TCP超过对端声明的MSS。如果对端没有指定一个MSS，则默认为536。一个实现也可以按21.9节中讲的那样为每个路由单独保存路径MTU信息。

一旦选定了起始的报文段大小，在该连接上的所有被 TCP发送的IP数据报都将被设置DF比特。如果某个中间路由器需要对一个设置了 DF标志的数据报进行分片，它就丢弃这个数据报，并产生一个我们在11.6节介绍的ICMP的“不能分片”差错。

如果收到这个ICMP差错， TCP就减少段大小并进行重传。如果路由器产生的是一个较新的该类ICMP差错，则报文段大小被设置为下一跳的 MTU减去IP和TCP的首部长度。如果是一个较旧的该类ICMP差错，则必须尝试下一个可能的最小 MTU（见图2-5）。当由这个ICMP差错引起的重传发生时，拥塞窗口不需要变化，但要启动慢启动。

由于路由可以动态变化，因此在最后一次减少路径 MTU的一段时间以后，可以尝试使用一个较大的值（直到等于对端声明的 MSS或输出接口MTU的最小值）。RFC 1191推荐这个时间间隔为10分钟（我们在11.8节看到Solaris 2.2使用一个30分钟的时间间隔）。

在对非本地目的地，默认的 MSS通常为536字节，路径MTU发现可以避免在通过 MTU小于576（这非常罕见）的中间链路时进行分片。对于本地目的主机，也可以避免在中间链路（如以太网）的MTU小于端点网络（如令牌环网）的情况下进行分片。但为了能使路径 MTU更加有用和充分利用MTU大于576的广域网，一个实现必须停止使用为非本地目的制定的 536的MTU默认值。MSS的一个较好的选择是输出接口的MTU（当然要减去IP和TCP的首部大小）（在附录E中，我们将看到大多数的实现都允许系统管理员改变这个默认的 MSS值）。

24.2.1 一个例子

在某个中间路由器的MTU比任一个端点接口MTU小的情况下，我们能够观察路径MTU发现是如何工作的。图24-1显示了这个例子的拓扑结构。

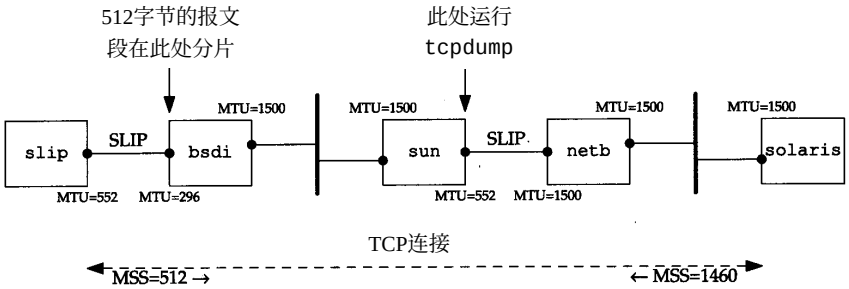


图24-1 路径MTU例子的拓扑结构

我们从主机solaris（支持路径MTU发现机制）到主机slip建立一个连接。这个建立过程与UDP的路径MTU发现（图11-13）中的一个例子相同，但在这里我们已经把 slip接口的MTU设置为552，而不是通常的296。这使得slip通告一个512的MSS。但是在bsdi上的SLIP链路上的 MTU为296，这就引起超过 256的TCP报文段被分片。于是就可以观察在solaris上的路径MTU发现是如何进行处理的。

我们在solaris上运行sock程序并向slip上的丢弃服务器进行一个512字节的写操作：

Linux公社 www.linuxidc.com

```
solaris %sock -i -n1 -w512 slip discard
```

图24-2是在主机sun的SLIP接口上收集的tcpdump的输出结果。

```
 1  0.0                 solaris.33016 > slip.discard: S 1171660288:1171660288(0)
                                                        win 8760 <mss 1460> (DF)
 2  0.101597 (0.1016)   slip.discard > solaris.33016: S 137984001:137984001(0)
                                                        ack 1171660289 win 4096
                                                        <mss 512>
 3  0.630609 (0.5290)   solaris.33016 > slip.discard: P 1:513(512)
                                                        ack 1 win 9216 (DF)
 4  0.634433 (0.0038)   bsdi > solaris: icmp:
                              slip unreachable - need to frag, mtu = 296 (DF)
 5  0.660331 (0.0259)   solaris.33016 > slip.discard: F 513:513(0)
                                                        ack 1 win 9216 (DF)
 6  0.752664 (0.0923)   slip.discard > solaris.33016: . ack 1 win 4096
 7  1.110342 (0.3577)   solaris.33016 > slip.discard: P 1:257(256)
                                                        ack 1 win 9216 (DF)
 8  1.439330 (0.3290)   slip.discard > solaris.33016: . ack 257 win 3840
 9  1.770154 (0.3308)   solaris.33016 > slip.discard: FP 257:513(256)
                                                         ack 1 win 9216 (DF)
10  2.095987 (0.3258)   slip.discard > solaris.33016: . ack 514 win 3840
11  2.138193 (0.0422)   slip.discard > solaris.33016: F 1:1(0) ack 514 win 4096
12  2.310103 (0.1719)   solaris.33016 > slip.discard: . ack 2 win 9216 (DF)
```

图24-2 路径MTU发现的tcpdump 输出结果

在第1和第2行的MSS值是我们所期望的。接着我们观察到 solaris发送一个包含512字节的数据和对SYN的确认报文段（第 3行）（在习题18.9中可以看到这种把SYN的确认与第一个包含数据的报文段合并的情况）。这就在第4行产生了一个ICMP差错，我们看到路由器bsdi产生较新的、包含输出接口MTU的ICMP差错。

看来在这个差错回到solaris之前，就发送了FIN（第5行）。由于slip从没有收到被路由器bsdi丢弃的512字节的数据，因此并不期望接收这个序号（ 513），所以在第6行用它期望的序号（1）进行了响应。

在这个时候， ICMP差错返回到了solaris，solaris用两个256字节的报文段（第7和第9行）重传了512字节的数据。因为在bsdi后面可能还有具有更小的MTU的路由器，因此这两个报文段都设置了DF比特。

接着是一个较长的传输过程（持续了大约 15分钟），在最初的512字节变为256字节以后，solaris没有再尝试使用更大的报文段。

24.2.2 大分组还是小分组

常规知识告诉我们较大的分组比较好 [Mogul 1993, 15.2.8节]，因为发送较少的大分组比发送较多的小分组“花费”要少（假定分组的大小不足以引起分片，否则会引起其他方面的问题）。这些减少的花费与网络（分组首部负荷）、路由器（选路的决定）和主机（协议处理和设备中断）等有关。但并非所有的人都同意这种观点 [Bellovin 1993]。

考虑下面的例子。我们通过 4个路由器发送8192个字节，每个路由器与一个 T1电话线（1 544 000b/s）相连。首先我们使用两个4096字节的分组，如图24-3所示。

基本问题在于路由器是存储转发设备。它们通常接收整个输入分组，检验包含 IP检验和的IP首部，进行选路判决，然后开始发送输出分组。在这个图中，我们可以假定在理想情况下这些在路由器内部进行的操作不花费时间（水平点状线）。然而，从R1到R4它需要花费4个

Linux公社 www.linuxidc.com

单位时间来发送所有的8192字节。每一跳的时间为

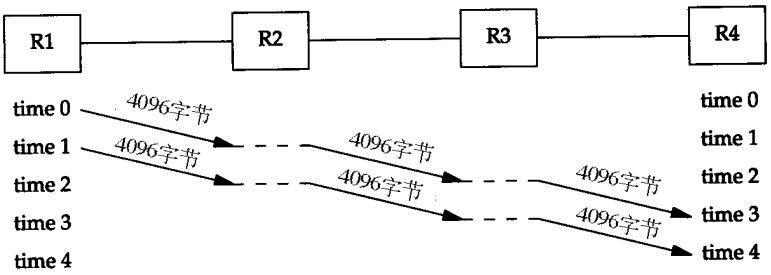


图24-3 通过4个路由器发送两个4096字节的分组

$$\frac{(4096字节+40字节)\times 8\ b/字节}{1\ 544\ 000\ b/s}=21.4\ ms/跳$$

(将TCP和IP的首部算为40字节)。发送数据的整个时间为分组个数加上跳数减 1，从图中可以看到是4个单位时间，或85.6秒。每个链路空闲2个单位时间，或42.8秒。

图24-4显示了当我们发送16个512字节的分组时所发生的情况。

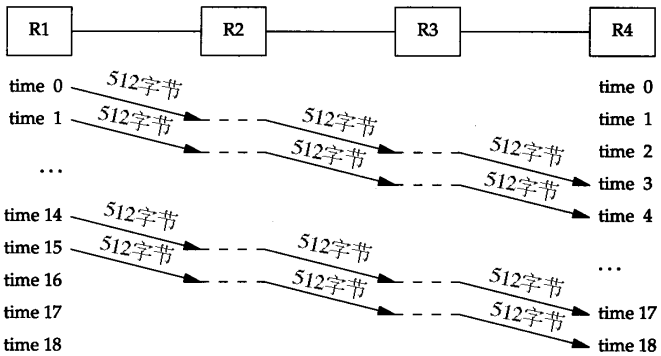


图24-4 通过4个路由器发送16个512字节的分组

这将花费更多的单位时间，但是由于发送的分组较短，因此每个单位时间较小。

$$\frac{(512字节+40字节)\times 8\ b/字节}{1\ 544\ 000\ b/s}=2.9\ ms/跳$$

现在总时间为（18×2.9）=52.2 ms。每个链路也空闲2个单位的时间，即5.8 ms。

在这个例子中，我们忽略了确认返回所需要的时间、连接建立和终止以及链路可能被其他流量共享等的影响。然而，在 [Bellovin 1993]中的测量表明，分组并不一定是越大越好。我们需要在更多的网络上对该领域进行更多的研究。

24.3 长肥管道

在20.7节，我们把一个连接的容量表示为

$$capacity\ (b) = bandwidth\ (b/s) \times round\text{-}trip\ time\ (s)$$

并称之为带宽时延乘积。也可称它为两端的管道大小。

当这个乘积变得越来越大时，TCP的某些局限性就会暴露出来。图 24-5显示了多种类型的网络的某些数值。

Linux公社 www.linuxidc.com

网 络	带宽(b/s)	RTT(ms)	带宽时延乘积(字节)
以太局域网	10 000 000	3	3 750
横跨大陆的T1电话线	1 544 000	60	11 580
卫星T1电话线	1 544 000	500	96 500
横跨大陆的T3电话线	45 000 000	60	337 500
横跨大陆的gigabit线路	1 000 000 000	60	7 500 000

图24-5 多种网络的带宽时延乘积

可以看到带宽时延乘积的单位是字节，这是因为我们用这个单位来测量每一端的缓存大小和窗口大小。

具有大的带宽时延乘积的网络被称为长肥网络（Long Fat Network，即LFN，发音为“elefan(t)s”），而一个运行在LFN上的TCP连接被称为长肥管道。回顾图20-11和图20-12，管道可以被水平拉长（一个长的RTT），或被垂直拉高（较高的带宽），或向两个方向拉伸。使用长肥管道会遇到多种问题。

1) TCP首部中窗口大小为16 bit，从而将窗口限制在65535个字节内。但是从图24-5的最后一列可以看到，现有的网络需要一个更大的窗口来提供最大的吞吐量。在24.4节介绍的窗口扩大选项可以解决这个问题。

2) 在一个长肥网络LFN内的分组丢失会使吞吐量急剧减少。如果只有一个报文段丢失，我们需要利用21.7节介绍的快速重传和快速恢复算法来使管道避免耗尽。但是即使使用这些算法，在一个窗口内发生的多个分组丢失也会典型地使管道耗尽（如果管道耗尽了，慢启动会使它渐渐填满，但这个过程将需要经过多个RTT）。

   在RFC 1072 [Jacobson and Braden 1988]中建议使用有选择的确认（SACK）来处理在一个窗口发生的多个分组丢失。但是这个功能在RFC 1323中被忽略了，因为作者觉得在把它们纳入TCP之前需要先解决一些技术上的问题。

3) 我们在第21.4节看到许多TCP实现对每个窗口的RTT仅进行一次测量。它们并不对每个报文段进行RTT测量。在一个长肥网络LFN上需要更好的RTT测量机制。我们将在24.5节介绍时间戳选项，它允许更多的报文段被计时，包括重传。

4) TCP对每个字节数据使用一个32 bit无符号的序号来进行标识。如果在网络中有一个被延迟一段时间的报文段，它所在的连接已被释放，而一个新的连接在这两个主机之间又建立了，怎样才能防止这样的报文段再次出现呢？首先回想起IP首部中的TTL为每个IP段规定了一个生存时间的上限——255跳或255秒，看哪一个上限先达到。在18.6节我们定义了最大的报文段生存时间（MSL）作为一个实现的参数来阻止这种情况的发生。推荐的MSL的值为2分钟（给出一个240秒的2MSL），但是我们在18.6节看到许多实现使用的MSL为30秒。

   在长肥网络LFN上，TCP的序号会碰到一个不同的问题。由于序号空间是有限的，在已经传输了4 294 967 296个字节以后序号会被重用。如果一个包含序号*N*字节数据的报文段在网络上被迟延并在连接仍然有效时又出现，会发生什么情况呢？这仅仅是一个相同序号*N*在MSL期间是否被重用的问题，也就是说，网络是否足够快以至于在不到一个MSL的时候序号就发生了回绕。在一个以太网上要发送如此多的数据通常需要60分钟左右，因此不会发生这种情况。但是在带宽增加时，这个时间将会减少：一个T3的电话线（45 Mb/s）在12分钟内会发生回绕，FDDI（100 Mb/s）为5分钟，而一个千兆比网络（1000 Mb/s）则为34秒。这时问题不再是带宽时延乘积，而在于带宽本身。

Linux公社 www.linuxidc.com

在24.6节，我们将介绍一种对付这种情况的办法：使用 TCP的时间戳选项的 PAWS (Protection Against Wrapped Sequence numbers)算法（保护回绕的序号）。

4.4BSD包含了我们将要在下面介绍的所有选项和算法：窗口扩大选项、时间戳选项和保护回绕的序号。许多供应商也正在开始支持这些选项。

千兆比网络

当网络的速率达到千兆比的时候，情况就会发生变化。[Partridge 1994]详细介绍了千兆比网络。在这里我们看一下在时延和带宽之间的差别 [Kleinrock 1992]。

考虑通过美国发送一个100万字节的文件的情况，假定时延为30 ms。图24-6显示了两种情况：上图显示了使用一个T1电话线（1 544 000 b/s）的情况，而下图则是使用一个1 Gb/s网络的情况。*x*轴显示的是时间，发送方在图的左侧，而接收方则在图的右侧，y轴为网络容量。两幅图中的阴影区域表示发送的100万字节。

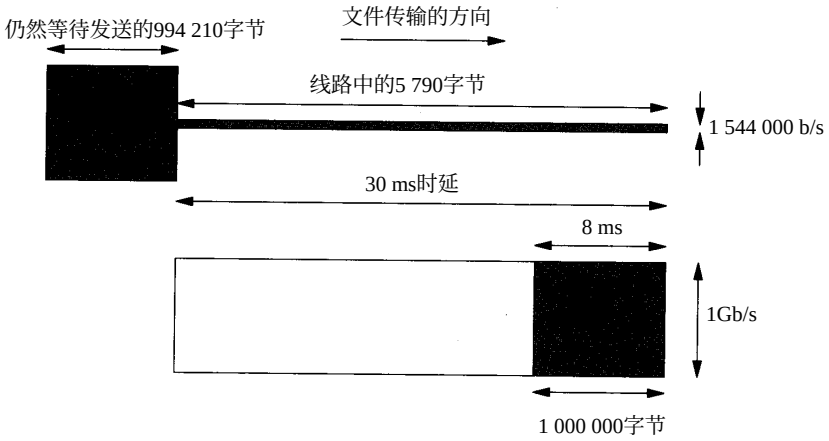


图24-6 以30 ms的延时通过网络发送100万字节的文件

图24-6显示了30 ms后这两个网络的状态。经过30 ms（延时）以后数据的第1个比特都已到达对端。但对T1网络而言，由于管道容量仅为5 790字节，因此发送方仍然有994 210个字节等待发送。而千兆比网络的容量则为3 750 000字节，因此，整个文件仅使用了25%左右的带宽，此时文件的最后一个比特已经到达第1个字节后8 ms处。

经过T1网络传输文件的总时间为5.211秒。如果增加更多的带宽，使用一个T3网络(45 000 000 b/s)，则总时间减少到0.208秒。增加约29倍的带宽可以将总时间减小到约25分之一。

使用千兆比网络传输文件的总时间为0.038秒：30 ms的时延加上8 ms的真正传输文件的时间。假定能够将带宽增加为2000 Mb/s，我们只能够将总时间减小为0.304 ms：同样30 ms的时延和4ms的真正传输时间。现在使带宽加倍仅能够将时间减少约10%。在千兆比速率下，时延限制占据了主要地位，而带宽不再成为限制。

时延主要是由光速引起的，而且不能够被减小（除非爱因斯坦是错误的）。当我们考虑到分组需要建立和终止一个连接时，这个固定时延起的作用就更糟糕了。千兆比网络会引起一些需要不同看待的连网观点。

Linux公社 www.linuxidc.com

24.4 窗口扩大选项

窗口扩大选项使TCP的窗口定义从16 bit增加为32 bit。这并不是通过修改TCP首部来实现的，TCP首部仍然使用16 bit，而是通过定义一个选项实现对16 bit的扩大操作(scaling operation)来完成的。于是TCP在内部将实际的窗口大小维持为32 bit的值。

在图18-20可以看到关于这个选项的例子。一个字节的移位记数器取值为0（没有扩大窗口的操作）和14。这个最大值14表示窗口大小为1 073 725 440字节（65535×2^{14}）。

这个选项只能够出现在一个SYN报文段中，因此当连接建立起来后，在每个方向的扩大因子是固定的。为了使用窗口扩大，两端必须在它们的SYN报文段中发送这个选项。主动建立连接的一方在其SYN中发送这个选项，但是被动建立连接的一方只能够在收到带有这个选项的SYN之后才可以发送这个选项。每个方向上的扩大因子可以不同。

如果主动连接的一方发送一个非零的扩大因子，但是没有从另一端收到一个窗口扩大选项，它就将发送和接收的移位记数器置为0。这就允许较新的系统能够与较旧的、不理解新选项的系统进行互操作。

> Host Requirements RFC要求TCP接受在任何报文段中的一个选项（只有前面定义的一个选项，即最大报文段大小，仅在SYN报文段中出现）。它还进一步要求TCP忽略任何它不理解的选项。这就使事情变得容易，因为所有新的选项都有一个长度字段（图8-20）。

假定我们正在使用窗口扩大选项，发送移位记数为S，而接收移位记数则为R。于是我们从另一端收到的每一个16 bit的通告窗口将被左移R位以获得实际的通告窗口大小。每次当我们向对方发送一个窗口通告的时候，我们将实际的32 bit窗口大小右移S比特，然后用它来替换TCP首部中的16 bit的值。

TCP根据接收缓存的大小自动选择移位计数。这个大小是由系统设置的，但是通常向应用进程提供了修改途径（我们在20.4节中讨论了这个缓存）。

一个例子

如果在4.4BSD的主机`vangogh.cs.berkeley.edu`上使用`sock`程序来初始化一个连接，我们可以观察到它的TCP计算窗口扩大因子的情况。下面的交互输出显示的是两个连续运行的程序，第1个指定接收缓存为128 000字节，而第2个的缓存则为220 000字节。

```
vangogh % sock -v -R128000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 128000
connected on 128.32.130.2.4107 to 140.252.13.35.7
TCP_MAXSEG = 512
hello, world                          我们键入这一行
hello, world                          此处是它的回显
^D                                    键入文件结束字符以终止

vangogh % sock -v -R220000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 220000
connected on 128.32.130.2.4108 to 140.252.13.35.7
TCP_MAXSEG = 512
bye, bye                              我们键入这一行
bye, bye                              此处是它的回显
^D                                    键入文件结束字符以终止
```

图24-7显示了这两个连接的`tcpdump`输出结果（去掉了第2个连接的最后8行，因为没有

Linux公社 www.linuxidc.com

什么新内容)。

```
 1   0.0                    vangogh.4107 > bsdi.echo: S 462402561:462402561(0)
                              win 65535
                              <mss 512,nop,wscale 1,nop,nop,timestamp 995351 0>
 2   0.003078 ( 0.0031)     bsdi.echo > vangogh.4107: S 177032705:177032705(0)
                              ack 462402562 win 4096 <mss 512>
 3   0.300255 ( 0.2972)     vangogh.4107 > bsdi.echo: . ack 1 win 65535

 4  16.920087 (16.6198)     vangogh.4107 > bsdi.echo: P 1:14(13) ack 1 win 65535
 5  16.923063 ( 0.0030)     bsdi.echo > vangogh.4107: P 1:14(13) ack 14 win 4096
 6  17.220114 ( 0.2971)     vangogh.4107 > bsdi.echo: . ack 14 win 65535

 7  26.640335 ( 9.4202)     vangogh.4107 > bsdi.echo: F 14:14(0) ack 14 win 65535
 8  26.642688 ( 0.0024)     bsdi.echo > vangogh.4107: . ack 15 win 4096
 9  26.643964 ( 0.0013)     bsdi.echo > vangogh.4107: F 14:14(0) ack 15 win 4096
10  26.880274 ( 0.2363)     vangogh.4107 > bsdi.echo: . ack 15 win 65535

11  44.400239 (17.5200)     vangogh.4108 > bsdi.echo: S 468226561:468226561(0)
                              win 65535
                              <mss 512,nop,wscale 2,nop,nop,timestamp 995440 0>
12  44.403358 ( 0.0031)     bsdi.echo > vangogh.4108: S 182792705:182792705(0)
                              ack 468226562 win 4096 <mss 512>
13  44.700027 ( 0.2967)     vangogh.4108 > bsdi.echo: . ack 1 win 65535
```

该连接的其余部分被删除

图24-7 窗口扩大选项的例子

在第1行，vangogh通告一个65535的窗口，并通过设置移位计数为 1来指明窗口扩大选项。这个通告的窗口是比接收窗口（128 000）还小的一个最大可能取值，因为在一个 SYN报文段中的窗口字段从不进行扩大运算。

扩大因子为1表示vangogh发送窗口通告一直到131 070（65535×2^1）。这将调节我们的接收缓存的大小（12 8000）。因为bsdi在它的SYN（第2行）中没有发送窗口扩大选项，因此这个选项没有被使用。注意到vangogh在随后的连接阶段继续使用最大可能的窗口（65535）。

对于第2个连接vangogh请求的移位计数为2，表明它希望发送窗口通告一直为 262 140（65535×2^2），这比我们的接收缓存（220 000）大。

24.5 时间戳选项

时间戳选项使发送方在每个报文段中放置一个时间戳值。接收方在确认中返回这个数值，从而允许发送方为每一个收到的 ACK计算RTT（我们必须说“每一个收到的 ACK”而不是“每一个报文段”，是因为TCP通常用一个ACK来确认多个报文段）。我们提到过目前许多实现为每一个窗口只计算一个 RTT，对于包含8个报文段的窗口而言这是正确的。然而，较大的窗口大小则需要进行更好的RTT计算。

> RFC 1323的3.1节给出了需要为较大窗口进行更好的RTT计算的信号处理的理由。通常RTT通过对一个数据信号（包含数据的报文段）以较低的频率（每个窗口一次）进行采样来进行计算，这就将别名引入了被估计的RTT中。当每个窗口中有8个报文段时，采样速率为数据率的1/8，这还是可以忍受的。但是如果每个窗口中有100个报文段时，采样速率则为数据速率的1/100，这将导致被估计的RTT不精确，从而引起不必要的重传。如果一个报文段被丢失，则会使情况变得更糟。

图18-20显示了时间戳选项的格式。发送方在第 1个字段中放置一个32 bit的值，接收方在应答字段中回显这个数值。包含这个选项的 TCP首部长度将从正常的20字节增加为32字节。

Linux公社 www.linuxidc.com

时间戳是一个单调递增的值。由于接收方只需要回显收到的内容，因此不需要关注时间戳单元是什么。这个选项不需要在两个主机之间进行任何形式的时钟同步。 RFC 1323推荐在1毫秒和1秒之间将时间戳的值加1。

4.4BSD在启动时将时间戳始终设置为0，然后每隔500 ms将时间戳时钟加1。

在图24-7中，如果观察在报文段1和报文段11的时间戳，它们之间的差（89个单元）对应于每个单元500 ms的规定，因为实际时间差为44.4秒。

在连接建立阶段，对这个选项的规定与前一节讲的窗口扩大选项类似。主动发起连接的一方在它的SYN中指定选项。只有在它从另一方的 SYN中收到了这个选项之后，该选项才会在以后的报文段中进行设置。

我们已经看到接收方 TCP不需要对每个包含数据的报文段进行确认，许多实现每两个报文段发送一个ACK。如果接收方发送一个确认了两个报文段的 ACK，那么哪一个收到的时间戳应当放入回显应答字段中来发回去呢？

为了减少任一端所维持的状态数量，对于每个连接只保持一个时间戳的数值。选择何时更新这个数值的算法非常简单：

1) TCP跟踪下一个ACK中将要发送的时间戳的值（一个名为 *tsrecent*的变量）以及最后发送的ACK中的确认序号（一个名为 *lastack*的变量）。这个序号就是接收方期望的序号。

2) 当一个包含有字节号 *lastack*的报文段到达时，则该报文段中的时间戳被保存在 *tsrecent*中。

3) 无论何时发送一个时间戳选项， *tsrecent*就作为时间戳回显应答字段被发送，而序号字段被保存在 *lastack*中。

这个算法能够处理下面两种情况：

1) 如果ACK被接收方迟延，则作为回显值的时间戳值应该对应于最早被确认的报文段。例如，如果两个包含1~1024和1025~2048字节的报文段到达，每一个都带有一个时间戳选项，接收方产生一个ACK 2049来对它们进行确认。此时， ACK中的时间戳应该是包含字节1~1024的第1个报文段中的时间戳。这种处理是正确的，因为发送方在进行重传超时时间的计算时，必须将迟延的ACK也考虑在内。

2) 如果一个收到的报文段虽然在窗口范围内但同时又是失序，这就表明前面的报文段已经丢失。当那个丢失的报文段到达时，它的时间戳（而不是失序的报文段的时间戳）将被回显。例如，假定有3个各包含1024字节数据的报文段，按如下顺序接收：包含字节 1~1024的报文段1，包含字节2049~4072的报文段3和包含字节1025~2048的报文段2。返回的ACK应该是带有报文段1的时间戳的ACK 1025（一个正常的所期望的对数据的 ACK）、带有报文段1的时间戳的ACK 1025（一个重复的、响应位于窗口内但却是失序的报文段的 ACK），然后是带有报文段2的时间戳的ACK 3073（不是报文段3中的较后的时间戳）。这与当报文段丢失时的对RTT估计过高具有同样的效果，但这比估计过低要好些。而且，如果最后的 ACK含有来自报文段3的时间戳，它可以包括重复的 ACK返回和报文段2被重传所需要的时间，或者可以包括发送方的报文段2的重传超时定时器到期的时间。无论在哪一种情况下，回显报文段 3的时间戳将引起发送方的RTT计算出现偏差。

尽管时间戳选项能够更好地计算 RTT，它还为发送方提供了一种方法，以避免接收到旧的报文段，并认为它们是现在的数据的一部分。下一节将对此进行描述。

Linux公社 www.linuxidc.com

24.6 PAWS：防止回绕的序号

考虑一个使用窗口扩大选项的 TCP连接，其最大可能的窗口大小为 1千兆字节（2^{30}）（最大的窗口是65535×2^{14}，而不是$2^{16}\times2^{14}$，但只比这个数值小一点点，并不影响这里的讨论）。还假定使用了时间戳选项，并且由发送方指定的时间戳对每个将要发送的窗口加 1（这是保守的方法。通常时间戳比这种方式增加得快）。图24-8显示了在传输6 千兆字节的数据时，在两个主机之间可能的数据流。为了避免使用许多 10位的数字，我们使用G来表示1 073 741 824的倍数。我们还使用了tcpdump的记号，即用*J:K*来表示通过了*J*字节的数据，且包括字节*K*-1。

时间	发送字节	发送序号	发送时间戳	接　　收
A	0G:1G	0G:1G	1	正确
B	1G:2G	1G:2G	2	正确，但有一个段丢失并重发
C	2G:3G	2G:3G	3	正确
D	3G:4G	3G:4G	4	正确
E	4G:5G	0G:1G	5	正确
F	5G:6G	1G:2G	6	正确，但重发的段又出现了

图24-8 在6个1千兆字节的窗口中传输6千兆字节的数据

32 bit的序号在时间D和时间E之间发生了回绕。假定一个报文段在时间 B丢失并被重传。还假定这个丢失的报文段在时间E重新出现。

这假定了在报文段丢失和重新出现之间的时间差小于 MSL，否则这个报文段在它的 TTL到期时会被某个路由器丢弃。正如我们前面提到的，这种情况只有在高速连接上才会发生，此时旧的报文段重新出现，并带有当前要传输的序号。

我们还可以从图 24-8中观察到使用时间戳可以避免这种情况。接收方将时间戳视为序列号的一个32 bit的扩展。由于在时间E重新出现的报文段的时间戳为2，这比最近有效的时间戳小（5或6），因此PAWS算法将其丢弃。

PAWS算法不需要在发送方和接收方之间进行任何形式的时间同步。接收方所需要的就是时间戳的值应该单调递增，并且每个窗口至少增加 1。

24.7 T/TCP：为事务用的TCP扩展

TCP提供的是一种虚电路方式的运输服务。一个连接的生存时间包括三个不同的阶段：建立、数据传输和终止。这种虚电路服务非常适合诸如远程注册和文件传输之类的应用。

但是，还有出现其他的应用进程被设计成使用事务服务。一个事务 (transaction)就是符合下面这些特征的一个客户请求及其随后的服务器响应。

1) 应该避免连接建立和连接终止的开销，在可能的时候，发送一个请求分组并接收一个应答分组。

2) 等待时间应当减少到等于RTT与SPT之和。其中RTT (Round-Trip Time)为往返时间，而SPT (Server Processing Time)则是服务器处理请求的时间。

3) 服务器应当能够检测出重复的请求，并且当收到一个重复的请求时不重新处理事务（避免重新处理意味着服务器不必再次处理请求，而是返回保存的、与该请求对应的应答）。

我们已经看到的一个使用这种类型服务的应用就是域名服务（第 14章），尽管DNS与服务器重新处理重复的请求无关。

Linux公社 www.linuxidc.com

如今一个应用程序设计人员面对的一种选择是使用 TCP还是UDP。TCP提供了过多的事务特征，而UDP提供的则不够。通常应用程序使用UDP来构造（避免TCP连接的开销），而许多需要的特征（如动态超时和重传、拥塞避免等）被放置在应用层，一遍又一遍的重新设计和实现。

一个较好的解决方法是提供一个能够提供足够多的事务处理功能的运输层。我们在本节所介绍的事务协议被称为 T/TCP。我们从它的定义，即 RFC 1379 [Braden 1992b]和[Braden 1992c]，开始介绍。

大多数的TCP需要使用7个报文段来打开和关闭一个连接（见图 18-13）。现在增加三个报文段：一个对应于请求，一个对应于应答和对请求的确认，第三个对应于对应答的确认。如果额外的控制比特被追加到报文段上 —— 也就是，第1个报文段带有 SYN、客户请求和一个 FIN —— 客户仍然能够看到一个 2倍的RTT与SPT之和的最小开销（与数据一起发送一个 SYN和FIN是合法的；当前的TCP是否能够正确处理它们是另外一个问题）。

另一个与TCP有关的问题是TIME_WAIT状态和它需要的2MSL的等待时间。正如在习题18.14中看到的，这使两个主机之间的事务率降低到每秒 268个。

TCP为处理事务而需要进行的两个改动是避免三次握手和缩短 WAIT_TIME状态。T/TCP通过使用加速打开来避免三次握手：

1) 它为打开的连接指定一个32 bit的连接计数CC (Connection Count)，无论主动打开还是被动打开。一个主机的CC值从一个全局计数器中获得，该计数器每次被使用时加 1。
2) 在两个使用T/TCP的主机之间的每一个报文段都包括一个新的 TCP选项CC。这个选项的长度为6个字节，包含发送方在该连接上的32 bit的CC值。
3) 一个主机维持一个缓存，该缓存保留每个主机上一次的 CC值，这些值从来自这个主机的一个可接受的SYN报文段中获得。
4) 当在一个开始的SYN中收到一个CC选项的时候，接收方比较收到的值与为该发送方缓存的CC值。如果接收到的 CC比缓存的大，则该 SYN是新的，报文段中的任何数据被传递给接收应用进程（服务器）。这个连接被称为半同步。
   如果接收的CC比缓存的小，或者接收主机上没有对应这个客户的缓存 CC，则执行正常的TCP三次握手过程。
5) 为响应一个开始的SYN，带有SYN和ACK的报文段在另一个被称为 CCECHO的选项中回显所接收到的CC值。
6) 在一个非SYN报文段中的CC值检测和拒绝来自同一个连接的前一个替身的任何重复的报文段。

这种“加速打开”避免了使用三次握手的要求，除非客户或者服务器已经崩溃并重新启动。这样做的代价是服务器必须记住从每个客户接收的最近的 CC值。

基于在两个主机之间测量 RTT来动态计算 TIME_WAIT的延时，可以缩短 TIME_WAIT状态。TIME_WAIT时延被设置为8倍的重传超时值*RTO*（见21.3节）。

通过使用这些特征，最小的事务序列是交换三个报文段：

1) 由一个主动打开引起的客户到服务器：客户的 SYN、客户的数据（请求）、客户的FIN以及客户的CC。当被动的服务器TCP接收到这个报文段的时候，如果客户的 CC比为这个客户缓存的CC要大，则客户的数据被传送给服务器应用程序进行处理。
2) 服务器到客户：服务器的 SYN、服务器的数据（应答）、服务器的FIN、对客户的FIN

Linux公社 www.linuxidc.com

的ACK、服务器的CC以及客户的CC的CCECHO。由于TCP的确认是累积的，这个对客户的FIN的ACK也对客户的SYN、数据及FIN进行了确认。

当客户TCP接收到这个报文段，就将其传送给客户应用进程。

3) 客户到服务器：对服务器的FIN的ACK，它也确认了服务器的SYN、数据和FIN。

客户对它的请求的响应时间为RTT与SPT的和。

在参考资料中有许多关于实现这个TCP选项的很好的地方。我们在这里将它们归纳如下：

- 服务器的SYN和ACK（第2个报文段）必须被迟延，从而允许应答与它一起捎带发送（通常对SYN的ACK是不迟延的）。但它也不能迟延得太多，否则客户将超时并引起重传。
- 请求可以需要多个报文段，但是服务器必须对它们可能失序达到的情况进行处理（通常当数据在SYN之前到达时，该数据被丢弃并产生一个复位。通过使用 T/TCP，这些失序的数据将放入队列中处理）。
- API必须使服务器进程用一个单一的操作来发送数据和关闭连接，从而允许第二个报文段中的FIN与应答一起捎带发送（通常应用进程先写应答，从而引起发送一个数据报文段，然后关闭连接，引起发送FIN）。
- 在收到来自服务器的MSS通告之前，客户在第1个报文段中正在发送数据。为避免限制客户的MSS为536，一个给定主机的MSS应该与它的CC值一起缓存。
- 客户在没有接收到来自服务器的窗口通告之前也可以向服务器发送数据。 T/TCP建议默认的窗口为4096，并且也为服务器缓存拥塞门限。
- 使用最小3个报文段交换，在每个方向上只能计算一个 RTT。加上包括了服务器处理时间的客户测量RTT。这意味着被平滑的RTT及其方差的值也必须为服务器缓存起来，这与我们在21.9节描述的类似。

T/TCP的特征中吸引人的地方在于它对现有协议进行了最小的修改，同时又兼容了现有的实现。它还利用了TCP中现有的工程特征（动态超时和重传、拥塞避免等），而不是迫使应用进程来处理这些问题。

一个可作为替换的事务协议是通用报文事务协议 VMTP（Versatile Message Transaction Protocol），该协议在RFC 1045 [Cheriton 1988]中进行了描述。与T/TCP是现有协议的一个小的扩充不同， VMTP是使用IP的一个完整的运输层。 VMTP处理差错检测、重传和重复压缩。它还支持多播通信。

24.8 TCP的性能

在80年代中期出版的数值显示出 TCP在一个以太网上的吞吐量在每秒 100 000~200 000字节之间（[Stevens 1990]的17.5节给出了参考文献）。从那时起事情已经发生了许多改变。现在通常使用的硬件（工作站和更快的个人电脑）每秒可以传输 800 000字节或者更快。

在10 Mb/s的以太网上计算我们能够观察到的理论上的TCP最大吞吐量是一件值得做的练习[Warnock

字 段	数据(字节)	ACK(字节)
以太网前导	8	8
以太网目的地址	6	6
以太网源地址	6	6
以太类型字段	2	2
IP首部	20	20
TCP首部	20	20
用户数据	1460	0
填充字符	0	6
以太网CRC检验	4	4
分组间隙(9.6ms)	12	12
总计	1538	84

图24-9 计算以太网理论上最大吞吐量的字段大小

Linux公社 www.linuxidc.com

1991]。我们可以在图24-9中看到这个计算的基础。这个图显示了满长度的数据报文段和一个ACK交换的全部的字节。

我们必须计及所有的开销：前同步码、加到确认上的填充字节、循环冗余检验 CRC以及分组之间的最小间隔（9.6ms，相当在10 Mb/s速率下的12个字节）。

首先假定发送方传输两个背对背、满长度的数据报文段，然后接收方为这两个报文段发送一个ACK。于是最大的吞吐量（用户数据）为：

$$throughput=\frac{2\times 1460\ \text{B}}{22\times 1538\ \text{B}+84\ \text{B}}\times\frac{10\ 000\ 000\ \text{b/s}}{8\ \text{b/B}}=1\ 555\ 063\ \text{B/S}$$

如果TCP窗口开到它的最大值（65535，不使用窗口扩大选项），这就允许一个窗口容纳44个1460字节的报文段。如果接收方每个报文段发送一个ACK，则计算变为：

$$throughput=\frac{22\times 1460\ \text{B}}{22\times 1538\ \text{B}+84\ \text{B}}\times\frac{10\ 000\ 000\ \text{b/s}}{8\ \text{b/B}}=1\ 183\ 667\ \text{B/S}$$

这就是理论上的限制，并做出某些假定：接收方发送的一个 ACK没有和发送方的报文段之一在以太网上发生冲突；发送方可按以太网的最小间隔时间来发送两个报文段；接收方可以在最小的以太网间隔时间内产生一个 ACK。不论在这些数字上多么乐观，[Warnock 1991]在一个以太网上使用标准的多用户工作站（即使是快的工作站）测量到了一个连续的 1 075 000字节/秒的速率，这个值在理论值的90%之内。

当移到更快的网络上时，如FDDI（100 Mb/s），[Schryver 1993]指出三个商业厂家已经演示了在FDDI上的TCP在80 Mb/s~90 Mb/s之间。即使在有更多带宽的环境下，[Borman 1992]报告说两个Gray Y-MP计算机在一个800 Mb/s的HIPPI通道上最大值为781 Mb/s，而运行在一个Gray Y-MP上的使用环回接口的两个进程间的速率为907 Mb/s。

下面这些实际限制适用于任何的实际情况 [Borman 1991]。

1) 不能比最慢的链路运行得更快。

2) 不能比最慢的机器的内存运行得更快。这假定实现是只使用一遍数据。如果不是这样（也就是说，实现使用一遍数据是将它从用户空间复制到内核中，而使用另一遍数据是计算TCP的检验和），那么将运行得更慢。[Dalton et al. 1993]描述了将数据复制数目减少从而使一个标准伯克利源程序的性能得到改进。[Partridge and Pink 1993]将类似的“复制与检验和”的改变与其他性能改进措施一道应用于UDP，从而将UDP的性能提高了约30%。

3) 不能够比由接收方提供的窗口大小除以往返时间所得结果运行得更快（这就是带宽时延乘积公式，使用窗口大小作为带宽时延乘积，并解出带宽）。如果使用24.4节的最大窗口扩大因子14，则窗口大小为1.073 千兆字节，所以这除以RTT的结果就是带宽的极限。

所有这些数字的重要意义就是TCP的最高运行速率的真正上限是由TCP的窗口大小和光速决定的。正如[Partridge and Pink 1993]中计算的那样，许多协议性能问题在于实现中的缺陷而不是协议所固有的一些限制。

24.9 小结

本章已经讨论了五个新的TCP特征：路径MTU发现、窗口扩大选项、时间戳选项、序号回绕保护以及使用改进的TCP事务处理。我们观察到中间的三个特征是为在长肥管道——具有大的带宽时延乘积的网络——上优化性能所需要的。

Linux公社 www.linuxidc.com

路径MTU发现在MTU较大时，对于非本地连接，允许 TCP使用比默认的536大的窗口。这样可以提高性能。

窗口扩大选项使最大的TCP窗口从65535增加到1千兆字节以上。时间戳选项允许多个报文段被精确计时，并允许接收方提供序号回绕保护（ PAWS）。这对于高速连接是必须的。这些新的TCP选项在连接时进行协商，并被不理解它们的旧系统忽略，从而允许较新的系统与旧的系统进行交互。

为事务用的TCP扩展，即T/TCP，允许一个客户/服务器的请求-应答序列在通常的情况下只使用三个报文段来完成。它避免使用三次握手，并缩短了 TIME_WAIT状态，其方法是为每个主机高速缓存少量的信息，这些信息曾用来建立过一个连接。它还在包含数据报文段中使用SYN和FIN标志。

由于还有许多关于TCP能够运行多快的不精确的传闻，因此我们以对 TCP性能的分析来结束本章。对于一个使用本章介绍的较新特征、协调得非常好的实现而言， TCP的性能仅受最大的1千兆字节窗口和光速（也就是往返时间）的限制。

习题

24.1 当一个系统发送一个开始的SYN报文段，其窗口扩大因子为0，这是什么含义？

24.2 如果在图24-7中的主机`bsdi`支持窗口扩大选项，则来自`vangogh`的报文段3的16 bit窗口大小字段中的期望值是多少？类似地，如果在该图的第 2个连接中也使用这个选项，那么报文段13中的窗口通告应该是多少？

24.3 与在建立连接时的固定窗口扩大因子不同，已经定义过的窗口扩大因子能否在扩大因子变化时也出现呢？

24.4 假定MSL为2分钟，那么在什么速率下序号回绕会成为一个问题呢？

24.5 PAWS被定义为只在一个单独的连接中进行。为了使 TCP将PAWS来替换2MSL等待(即TIME_WAIT状态)，需要进行什么改动？

24.6 在24.4节最后的例子中，为什么`sock`程序在紧接着（具有IP地址和端口）后面的一行之前，将接收缓存的大小来输出呢？

24.7 假定MSS为1024，重新计算24.8节中的吞吐量。

24.8 时间戳选项是如何影响Karn算法（见21.3节）的？

24.9 如果主动建立连接的TCP发送带有SYN标志的报文段（没有使用我们在 24.7节介绍的扩展），那么接收TCP应该怎样处理这些数据呢？

24.10 在24.7节我们提到如果没有使用T/TCP扩展，即使主动开启方发送带有FIN的数据，客户在接收服务器的响应的时延仍然是两倍的RTT再加上SPT。给出符合这种情况的报文段。

24.11 假定支持T/TCP，且源自伯克利系统的最小*RTO*为0.5秒，重做习题18.14。

24.12 如果我们实现了T/TCP，并测量两个主机之间的事务时间，那么可以通过比较什么指标来确定它的有效性？

Linux公社 www.linuxidc.com

第25章　SNMP：简单网络管理协议

25.1　引言

随着网络技术的飞速发展，网络的数量也越来越多。而网络中的设备来自各个不同的厂家，如何管理这些设备就变得十分重要。本章的内容就是介绍管理这些设备的标准。

基于TCP/IP的网络管理包含两个部分：网络管理站（也叫管理进程， manager）和被管的网络单元（也叫被管设备）。被管设备种类繁多，例如：路由器、 X 终端、终端服务器和打印机等。这些被管设备的共同点就是都运行 TCP/IP协议。被管设备端和管理相关的软件叫做代理程序(agent)或代理进程。管理站一般都是带有彩色监视器的工作站，可以显示所有被管设备的状态(例如连接是否掉线、各种连接上的流量状况等)。

管理进程和代理进程之间的通信可以有两种方式。一种是管理进程向代理进程发出请求，询问一个具体的参数值（例如：你产生了多少个不可达的 ICMP端口？）。另外一种方式是代理进程主动向管理进程报告有某些重要的事件发生（例如：一个连接口掉线了）。当然，管理进程除了可以向代理进程询问某些参数值以外，它还可以按要求改变代理进程的参数值（例如：把默认的IP TTL值改为64）。

基于TCP/IP的网络管理包含3个组成部分：

1) 一个管理信息库MIB（Management Information Base）。管理信息库包含所有代理进程的所有可被查询和修改的参数。 RFC 1213 [McCloghrie and Rose 1991]定义了第二版的MIB，叫做MIB-II。

2) 关于MIB的一套公用的结构和表示符号。叫做管理信息结构　SMI（Structure of Management Information）。这个在RFC 1155 [Rose and McCloghrie 1990] 中定义。例如：SMI定义计数器是一个非负整数，它的计数范围是 0~4 294 967 295，当达到最大值时，又从0开始计数。

3) 管理进程和代理进程之间的通信协议，叫做简单网络管理协议 SNMP（Simple Network Management Protocol）。在RFC 1157 [Case et al. 1990]中定义。SNMP包括数据报交换的格式等。尽管可以在运输层采用各种各样的协议，但是在 SNMP中，用得最多的协议还是UDP。

上面提到的RFC所定义的SNMP叫做SNMP v1，或者就叫做SNMP，这也是本章的主要内容。到1993年为止，又有一些新的关于SNMP的 RFC发表。在这些RFC中定义的SNMP叫做第二版SNMP（SNMP v2），这将在25.12章节中讨论。

本章首先介绍管理进程和代理进程之间的协议，然后讨论参数的数据类型。在本章中将用到前面已经出现过的名词，如： IP、UDP和TCP等。我们在叙述中将举一些例子来帮助读者理解，这些例子和前面的某些章节相关。

25.2　协议

关于管理进程和代理进程之间的交互信息， SNMP定义了5种报文：

Linux公社 www.linuxidc.com

1) get-request操作：从代理进程处提取一个或多个参数值。

2) get-next-request操作：从代理进程处提取一个或多个参数的下一个参数值（关于“下一个（next）”的含义将在后面的章节中介绍）。

3) set-request操作：设置代理进程的一个或多个参数值。

4) get-response操作：返回的一个或多个参数值。这个操作是由代理进程发出的。它是前面3中操作的响应操作。

5) trap 操作：代理进程主动发出的报文，通知管理进程有某些事情发生。

前面的3个操作是由管理进程向代理进程发出的。后面两个是代理进程发给管理进程的（为简化起见，前面3个操作今后叫做get、get-next和set操作）。图25-1描述了这5种操作。

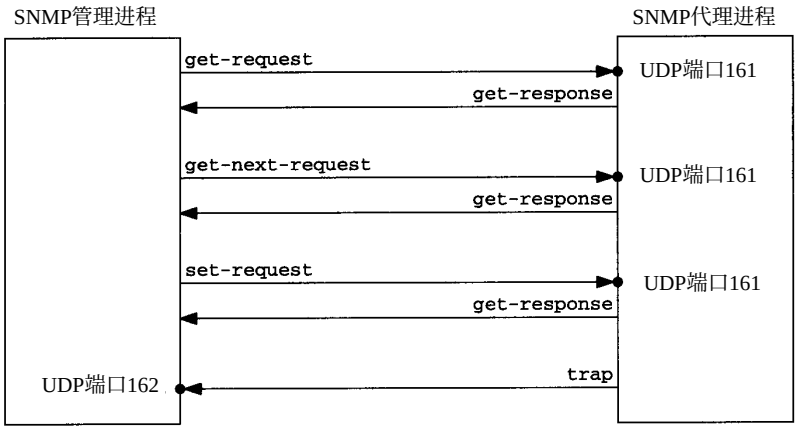


图25-1 SNMP的5种操作

既然这些操作中的前 4种操作是简单的请求 -应答方式（也就是管理进程发出请求，代理进程应答响应），而且在SNMP中往往使用UDP协议，所以可能发生管理进程和代理进程之间数据报丢失的情况。因此一定要有超时和重传机制。

管理进程发出的前面 3种操作采用UDP的161端口。代理进程发出的 Trap操作采用UDP的162端口。由于收发采用了不同的端口号，所以一个系统可以同时为管理进程和代理进程（参见习题25.1）。

图25-2是封装成UDP数据报的5种操作的SNMP报文格式。

在图中，我们仅仅对IP和UDP的首部长度进行了标注。这是由于：SNMP报文的编码采用了ASN.1和BER，这就使得报文的长度取决于变量的类型和值。关于 ASN.1和BER的内容将在后面介绍。在这里介绍各个字段的内容和作用。

版本字段是0。该字段的值是通过SNMP版本号减去1得到的。显然0代表SNMP v1。

图25-3显示各种PDU对应的值（PDU即协议数据单元，也就是分组）。

共同体字段是一个字符串。这是管理进程和代理进程之间的口令，是明文格式。默认的值是public。

对于get、get-next和set操作，请求标识由管理进程设置，然后由代理进程在 get-response中返回。这种类型的字段我们在其他UDP应用中曾经见过（回忆一下在图14-3中DNS的标识字段，或者是图16-2中的事务标识字段）。这个字段的作用是使客户进程（在目前情况下是管理进程）能够将服务器进程（即代理进程）发出的响应和客户进程发出的查询进行匹配。这个

Linux公社 www.linuxidc.com

字段允许管理进程对一个或多个代理进程发出多个请求，并且从返回的众多 应答中进行分类。

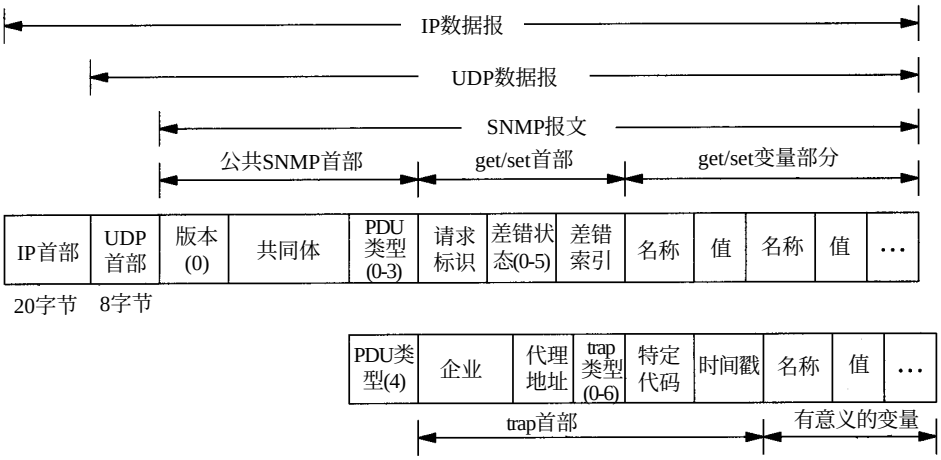


图25-2 SNMP报文的格式

差错状态字段是一个整数，它是由代理进程标注的，指明有差错发生。图25-4是参数值、名称和描述之间的对应关系。

差错索引字段是一个整数偏移量，指明当有差错发生时，差错发生在哪个参数。它是由代理进程标注的，并且只有在发生noSuchName、readOnly和badValue差错时才进行标注。

PDU类型	名　　称
0	get-request
1	get-next-request
2	get-response
3	set-request
4	trap

图25-3 SNMP报文中的PDU类型

差错状态	名　称	描　　述
0	noError	没有错误
1	tooBig	代理进程无法把响应放在一个SNMP消息中发送
2	noSuchName	操作一个不存在的变量
3	badValue	set操作的值或语义有错误
4	readOnly	管理进程试图修改一个只读变量
5	genErr	其他错误

图25-4 SNMP差错状态的值

在get、get-next和set的请求数据报中，包含变量名称和变量值的一张表。对于 get和get-next操作，变量值部分被忽略，也就是不需要填写。

对于trap操作符（PDU类型是4），SNMP报文格式有所变化。我们将在 25.10节中当讨论到trap时再详细讨论。

25.3 管理信息结构

SNMP中，数据类型并不多。在本节，我们就讨论这些数据类型，而不关心这些数据类型在实际中是如何编码的。

- INTEGER。一个变量虽然定义为整型，但也有多种形式。有些整型变量没有范围限制，有些整型变量定义为特定的数值（例如，IP的转发标志就只有允许转发时的1或者不允许转发时的2这两种），有些整型变量定义为一个特定的范围（例如，UDP和TCP的端口号就从0到65535）。
- OCTER STRING。0或多个8 bit字节，每个字节值在0~255之间。对于这种数据类型和

Linux公社 www.linuxidc.com

下一种数据类型的 BER编码，字符串的字节个数要超过字符串本身的长度。这些字符串不是以NULL结尾的字符串。

- DisplayString。0或多个8 bit字节，但是每个字节必须是 ASCII码（26.4中有ASCII字符集）。在MIB-II中，所有该类型的变量不能超过 255个字符（0个字符是可以的）。
- OBJECT IDENTIFIER将在下一节中介绍。
- NULL。代表相关的变量没有值。例如，在 get或get-next操作中，变量的值就是NULL，因为这些值还有待到代理进程处去取。
- IpAddress。4字节长度的OCTER STRING，以网络序表示的 IP地址。每个字节代表IP地址的一个字段。
- PhysAddress。OCTER STRING类型，代表物理地址（例如以太网物理地址为 6个字节长度）。
- Counter。非负的整数，可从0递增到 $2^{32}-1$（4 294 976 295）。达到最大值后归0。
- Gauge。非负的整数，取值范围为从 0到4 294 976 295(或增或减)。达到最大值后锁定，直到复位。例如， MIB中的tcpCurrEstab就是这种类型的变量的一个例子，它代表目前在ESTABLISHED或CLOSE_WAIT状态的TCP连接数。
- TimeTicks。时间计数器，以0.01秒为单位递增，但是不同的变量可以有不同的递增幅度。所以在定义这种类型的变量的时候，必须指定递增幅度。例如， MIB中的sysUpTime变量就是这种类型的变量，代表代理进程从启动开始的时间长度，以多少个百分之一秒的数目来表示。
- SEQUENCE。这一数据类型与C程序设计语言中的“ structure ”类似。一个SEQUENCE包括 0个或多个元素，每一个元素又是另一个 ASN.1数据类型。例如， MIB中的UdpEntry就是这种类型的变量。它代表在代理进程侧目前“ 激活 ”的 UDP数量（“ 激活 ”表示目前被应用程序所用）。在这个变量中包含两个元素：
  1) IpAddress类型中的udpLocalAddress，表示IP地址。
  2) INTEGER类型中的udpLocalPort，从0到65535，表示端口号。
- SEQUENDE OF。这是一个向量的定义，其所有元素具有相同的类型。如果每一个元素都具有简单的数据类型，例如是整数类型，那么我们就得到一个简单的向量（一个一维向量）。但是我们将看到， SNMP在使用这个数据类型时，其向量中的每一个元素是一个SEQUENCE（结构）。因而可以将它看成为一个二维数组或表。

  例如，名为udpTable的UDP监听表(listener)就是这种类型的变量。它是一个二元的SEQUENCE变量。每个二元组就是一个UdpEntry。如图25-5所示。

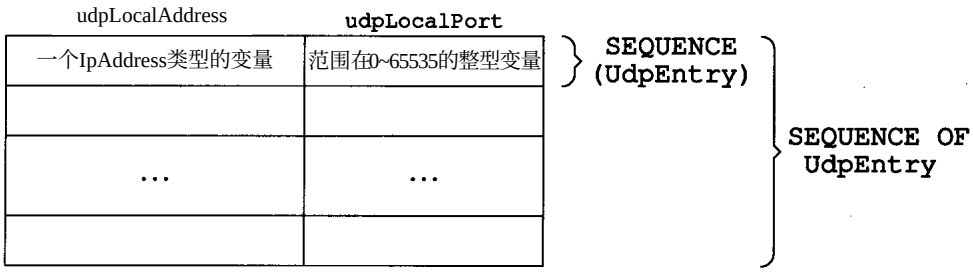


图25-5 表格形式的udpTable 变量

Linux公社 www.linuxidc.com

在SNMP中，对于这种类型的表格并没有标注它的列数。但在 25.7节中，我们将看到get-next操作是如何判断已经操作到最后一列的情况。同时，在 25.6节中，我们还将介绍管理进程如何表示它对某一行数据进行get或set操作。

25.4 对象标识符

对象标识是一种数据类型，它指明一种“授权”命名的对象。“授权”的意思就是这些标识不是随便分配的，它是由一些权威机构进行管理和分配的。

对象标识是一个整数序列，以点（“.”）分隔。这些整数构成一个树型结构，类似于 DNS（图14-1）或Unix的文件系统。对象标识从树的顶部开始，顶部没有标识，以 root表示（这和Unix中文件系统的树遍历方向非常类似）。

图25-6显示了在SNMP中用到的这种树型结构。所有的 MIB变量都从1.3.6.1.2.1这个标识开始。

树上的每个结点同时还有一个文字名。例如标识 1.3.6.1.2.1就和`iso.org.dod.internet.memt.mib`对应。这主要是为了人们阅读方便。在实际应用中，也就是说在管理进程和代理进程进行数据报交互时，MIB变量名是以对象标识来标识的，当然都是以1.3.6.1.2.1开头的。

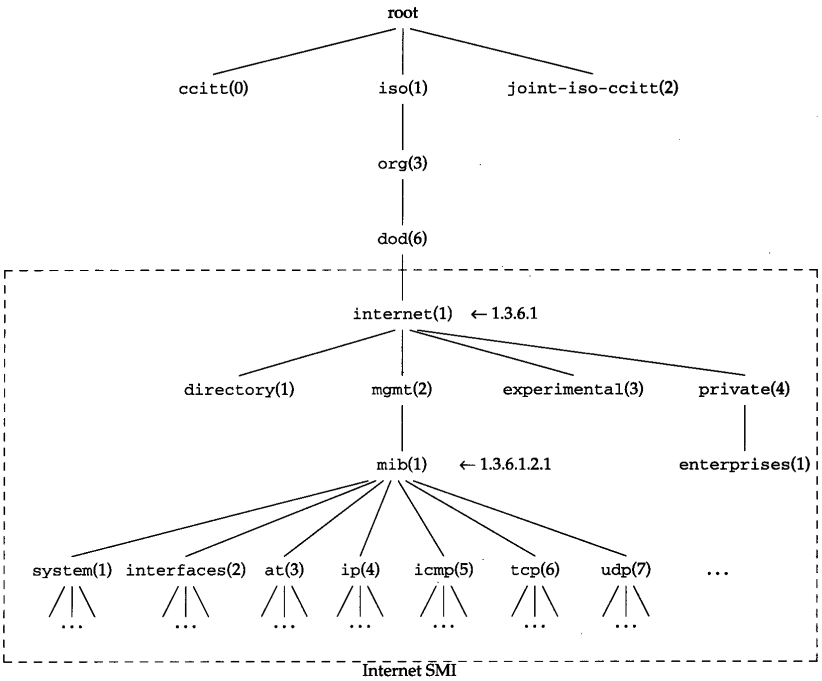


图25-6 管理信息库中的对象标识

在图25-6中，我们除了给出了`mib`对象标识外，还给出了`iso.org.dod.internet.private.enterprises`（1.3.6.1.4.1）这个标识。这是给厂家自定义而预留的。在 Assigned Number RFC中列出了在该结点下大约400个标识。

25.5 管理信息库介绍

所谓管理信息库，或者MIB，就是所有代理进程包含的、并且能够被管理进程进行查询和设

Linux公社 www.linuxidc.com

置的信息的集合。我们在前面已经提到了在RFC 1213 [McColghrie 和Rose 1991]中定义的MIB-II。

如图25-6所示，MIB被划分为若干个组，如system、interfaces、at（地址转换）和ip组等。

在本节，我们仅仅讨论UDP组中的变量。这个组比较简单，它包含几个变量和一个表格。在下一节，我们将以UDP组为例，详细讲解什么是实例标识（instance identification），什么是字典式排序(lexicographic ordering)以及和这些概念有关的一些简单例子。在这些例子之后，在25.8节我们继续回到MIB，描述MIB中的其他一些组。

在图25-6中我们画出了udp组在mib的下面。图25-7就显示了UDP组的结构。

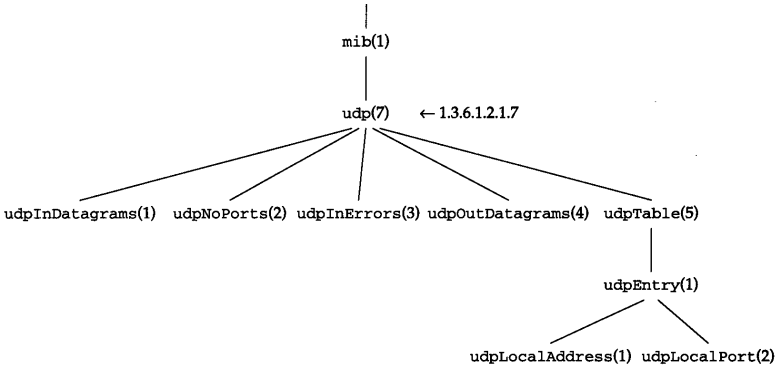


图25-7 UDP组的结构

在该组中，包含4个简单变量和1个由两个简单变量组成的表格。图 25-8描述了这4个简单变量。

名　　称	数据类型	R/W	描　述
udpInDatagrams	Counter		UDP数据报输入数
udpNoPorts	Counter		没有发送到有效端口的UDP数据报个数
udpInErrors	Counter		接收到的有错误的UDP数据报个数(例如检验错误)
udpOutDatagrams	Counter		UDP数据报输出数

图25-8 UDP组下的简单变量

在本章中，我们就以图 25-8的格式来描述所有的 MIB变量。“ R/W ”列如果为空，则代表该变量是只读的；如果变量是可读可写的，则以“ · ”符号来表示。哪怕整个组中的变量都是只读的，我们也将列出“ R/W ”列，以提示读者管理进程只能对这些变量进行查询操作(上图UDP组我们就是这样做的)。同样，如果变量类型是 INTEGET类型并且有范围约束，我们也将标明它的下限和上限，就如我们在下图中描述 UDP端口号所做的一样。

图25-9描述了在udpTable中的两个简单变量。

UDP监听表，索引=<udpLocalAddress>.<udpLocalPort>			
名　　称	数据类型	R/W	描　　述
udpLocalAddress	IpAddress		监听进程的本地IP地址。0.0.0.0代表接收任何接口的数据报
udpLocalPort	[0..65535]		监听进程的本地端口号

图25-9 udpTable 中的变量

每次当我们以SNMP表格形式来描述MIB变量时，表格的第1行都表示索引的值，它是表

Linux公社 www.linuxidc.com

格中的每一列的参考。在下一节中读者将看到的一些例子也是这样做的。

Case图

在图25-8中，前3个计数器是有相互关系的。Case图真实地描述了一个给出的MIB组中变量之间的相互关系。图25-10就是UDP组的Case图。

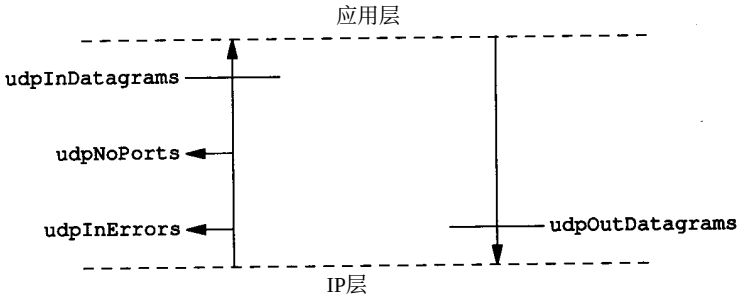


图25-10 UDP组的Case图

这张图表明，发送到应用层的UDP数据报的数量（udpInDatagrams）就是从IP层送到UDP层的UDP数据报的数量，当然udpInError和udpNoPorts也类似。同样，发送到IP层的UDP数据报的数量（udpOutDatagrams）就是从应用层发出的UDP数据报的数量。这表明udpInDatagram不包括udpInError和udpNoPorts。

在深入讲解MIB的时候，这些Case图被用来验证：分组的所有数据路径都是被计数的。[Rose 1994] 中显示了所有MIB组的Case图。

25.6 实例标识

当对MIB变量进行操作，如查询和设置变量的值时，必须对MIB的每个变量进行标识。首先，只有叶子结点是可操作的。SNMP没法处理表格的一整行或一整列。回到图25-7，在图25-8和图25-9中描述过的变量就是叶子结点，而mib、udp、udpTable和udpEntry就不是叶子结点。

25.6.1 简单变量

对于简单变量的处理方法是通过在其对象标识后面添加“ .0 ”来处理的。例如图25-8中的计数器udpInDatagrams，它的对象标识是1.3.6.1.2.1.7.1，它的实例标识是1.3.6.1.2.1.7.1.0，相对应的文字名称是iso.org.dod.internet.mgmt.mib.udp.udpInDatagrams.0。

虽然这个变量处理后通常可以缩写为 udpInDatagrams.0，但我们还是要提醒读者在SNMP报文中（图25-2）该变量的名称是其对象的标识1.3.6.1.2.1.7.1.0。

25.6.2 表格

表格的实例标识就要复杂得多。回顾一下图25-8中的UDP 监听表。

每个MIB中的表格都指明一个以上的索引。对于 UDP监听表来说，MIB定义了包含两个变量的联合索引，这两个变量是：udpLocalAddress，它是一个IP地址；udpLocalPort，它是一个整数（在图25-9中的第1行就显示了这个索引）。

假设在UDP监听表中有3行具体成员：第1行的IP地址是0.0.0.0，端口号是67；第2行的IP

Linux公社 www.linuxidc.com

地址是0.0.0.0，端口号是161；第3行的IP地址是0.0.0.0，端口号是520。如图25-11所示。

这意味着系统将从端口 67（BOOTP服务器）、端口161（SNMP）和端口520（RIP）接受来自任何接口的UDP数据报。表格中的这 3行经过处理后的结果在图 25-12中显示。

udpLocalAddress	udpLocalPort
0.0.0.0	67
0.0.0.0	161
0.0.0.0	520

图25-11 UDP监听表

25.6.3 字典式排序

MIB中按照对象标识进行排序时有一个隐含的排序规则。 MIB表格是根据其对象标识按照字典的顺序进行排序的。这就意味着图 25-12中的6个变量排序后的情况如图 25-13所示。从这种字典式排序中可以得出两个重要的结论。

行	对象标识	简　　称	值
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67 1.3.6.1.2.1.7.5.1.2.0.0.0.0.67	udpLocalAddress.0.0.0.0.67 udpLocalPort.0.0.0.0.67	0.0.0.0 67
2	1.3.6.1.2.1.7.5.1.1.0.0.0.0.161 1.3.6.1.2.1.7.5.1.2.0.0.0.0.161	udpLocalAddress.0.0.0.0.161 udpLocalPort.0.0.0.0.161	0.0.0.0 161
3	1.3.6.1.2.1.7.5.1.1.0.0.0.0.520 1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalAddress.0.0.0.0.520 udpLocalPort.0.0.0.0.520	0.0.0.0 520

图25-12 UDP监听表中行的实例标识

列	对象标识(字典序)	简　　称	值
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67 1.3.6.1.2.1.7.5.1.1.0.0.0.0.161 1.3.6.1.2.1.7.5.1.1.0.0.0.0.520	udpLocalAddress.0.0.0.0.67 udpLocalAddress.0.0.0.0.161 udpLocalAddress.0.0.0.0.520	0.0.0.0 0.0.0.0 0.0.0.0
2	1.3.6.1.2.1.7.5.1.2.0.0.0.0.67 1.3.6.1.2.1.7.5.1.2.0.0.0.0.161 1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalPort.0.0.0.0.67 udpLocalPort.0.0.0.0.161 udpLocalPort.0.0.0.0.520	67 161 520

图25-13 UDP监听表的字典式排序

1) 在表格中，一个给定变量（在这里指udpLocalAddress）的所有实例都在下个变量（这里指udpLocalPort）的所有实例之前显示。这暗示表格的操作顺序是“先列后行”的次序。这是由于对对象标识进行字典式排序所得到的，而不是按照人们的阅读习惯而排列的。

2) 表格中对行的排序和表格中索引的值有关。在图25-13中，67的字典序小于161，同样161的字典序小于520。

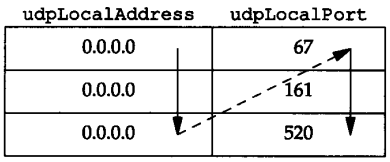


图25-14 按“先列后行”次序显示的UDP监听表

图25-14描述了例子中UDP监听表的这种“先列后行”的次序。

在下节中，讲述到get-next操作时，同样还会遇到这种“先列后行”的次序。

25.7 一些简单的例子

在本节中，我们将介绍如何从SNMP代理进程处获取变量的值。对代理进程进行查询的软件属于ISODE系统，叫做snmpi。两者在[Rose 1994]中有详细的介绍。

Linux公社 www.linuxidc.com

25.7.1 简单变量

对一个路由器取两个UDP组的简单变量值：

```
sun % snmpi -a gateway -c secret

snmpi> get udpInDatagrams.0 udpNoPorts.0
udpInDatagrams.0=616168
udpNoPorts.0=33

snmpi> quit
```

其中，-a选项代表要和之通信的代理进程名称，-c选项表示SNMP的共同体名。所谓共同体名，就是客户进程（在这里指 snmpi）提供、同时能被服务器进程（这里指代理进程gateway）所识别的一个口令，共同体名称是管理进程请求的权限标志。代理进程允许客户进程用只读共同体名对变量进行读操作，用读写共同体名对变量进行读和写操作。

Snmpi程序的输出提示符是snmpi>，在后面可以键入如get这样的命令，该软件将把它转化为SNMP中的get-request报文。当结束时，键入quit就退出（在后面的例子中，我们将省略掉quit的操作）。

图25-15显示的是对于这个例子tcpdump的两行输出结果。

```
1  0.0                  sun.1024 > gateway.161: GetRequest(42)
                        1.3.6.1.2.1.7.1.0 1.3.6.1.2.1.7.2.0

2  0.348875 (0.3489)    gateway.161 > sun.1024: GetResponse(46)
                        1.3.6.1.2.1.7.1.0=616168
                        1.3.6.1.2.1.7.2.0=33
```

图25-15 简单SNMP查询操作tcpdump 的输出结果

对这两个变量的查询请求是封装在一个UDP数据报中的，而响应也在一个UDP数据报中。

显示的变量是以其对象标识的形式显示的，这是在SNMP报文中实际传输的内容。我们必须指定这两个变量的实例是0。注意，变量的名称（它的对象标识）同样也在响应中返回。在下面我们将看到对于get-next操作这是必需的。

25.7.2 get-next操作

get-next操作是基于MIB的字典式排序的。在下面的例子中，首先向代理进程询问UDP后的下一个对象标识（由于不是一个叶子对象，没有指定任何实例）。代理进程将返回UDP组中的第1个对象，然后我们继续向代理进程取该对象的下一个对象标识，这时候第 2个对象将被返回。重复上面的步骤直到取出所有的对象为止。

```
sun % snmpi -a gateway -c secret

snmpi> next udp
udpInDatagrams.0=616318

snmpi> next udpInDatagrams.0
udpNoPorts.0=33

snmpi> next udpNoPorts.0
udpInErrors.0=0
```

这个例子解释了为什么get-next操作总是返回变量的名称，这是因为我们向代理进程询问下一个变量，代理进程就把变量值和名称一起返回了。

采用这种方式进行get-next操作，我们可以想象管理进程只要做一个简单的循环程序，

Linux公社 www.linuxidc.com

就可以从MIB树的顶点开始，对代理进程一步步地进行查询，就可以得出代理进程处所有的变量值和标识。该方式的另外一个用处就是可以对表格进行遍历。

25.7.3 表格的访问

对于“先列后行”次序的UDP监听表，只要采用前面的简单查询程序一步一步地进行操作，就可以遍历整个表格。只要从询问代理进程 udpTable的下一个变量开始就可以了。由于udpTable不是叶子对象，我们不能指定一个实例，但是get-next操作依然能够返回表格中的下一个对象。然后就可以以返回的结果为基础进行下一步的操作，代理进程也会以“先列后行”的次序返回下一个变量，这样就可以遍历整个表格。我们可以看到返回的次序和图25-14相同。

```
sun % snmpi -a gateway -c secret

snmpi> next udpTable
udpLocalAddress.0.0.0.0.67=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.67
udpLocalAddress.0.0.0.0.161=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.161
udpLocalAddress.0.0.0.0.520=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.520
udpLocalPort.0.0.0.0.67=67

snmpi> next udpLocalPort.0.0.0.0.67
udpLocalPort.0.0.0.0.161=161

snmpi> next udpLocalPort.0.0.0.0.161
udpLocalPort.0.0.0.0.520=520

snmpi> next udpLocalPort.0.0.0.0.520
snmpInPkts.0=59
```

我们已完成了对UDP监听表的操作

但是管理进程如何知道已经到达表格的最后一行呢？既然get-next操作返回结果中包含表格中的下一个变量的值和名称，当返回的结果是超出表格之外的下一个变量时，管理进程就可以发现变量的名称发生了较大的变化。这样就可以判断出已经到达表格的最后一行。例如在我们的例子中，当返回的是snmpInPkts变量的时候就代表已经到了UDP监听表的最后一个变量了。

25.8 管理信息库（续）

现在继续讨论MIB。我们仅仅介绍下列MIB组：system（系统标识）、if（接口）、at（地址转换）、ip、icmp和tcp。

25.8.1 system组

system组非常简单，它包含7个简单变量（例如，没有表格）。图25-16列出了system组的名称、数据类型和描述。

可以对netb路由器查询一些简单变量：

```
sun % snmpi -a netb -c secret

snmpi> get sysDescr.0 sysObjectID.0 sysUpTime.0 sysServices.0
sysDescr.0="Epilogue Technology SNMP agent for Telebit NetBlazer"
sysObjectID.0=1.3.6.1.4.1.12.42.3.1
sysUpTime.0=22 days, 11 hours, 23 minutes, 2 seconds (194178200 timeticks)
sysServices.0=0xc<internet,transport>
```

Linux公社 www.linuxidc.com

名　称	数据类型	R/W	描　　述
sysDescr	DisplayString		系统的文字描述
sysObjectID	ObjectID		在子树1.3.6.1.4.1中的厂商标识
sysUpTime	TimeTicks		从系统的网管部分启动以来运行的时间（以百分之一秒为计算单位）
sysContact	DisplayString	•	联系人的名字及联系方式
sysName	DisplayString	•	结点的完全合格的域名(FQDN)
sysLocation	DisplayString	•	结点的物理位置
sysServices	[0…127]	•	指示结点提供的服务的值。该值是此结点所支持的OSI模型中层次的和。根据所提供的服务，将下面的一些值相加：0x01(物理层)、0x02（数据链路层）、0x04（互联网层）、0x08（端到端的运输层）和 0x40（应用层）

图25-16　system组中的简单变量

回到图25-6中，system的对象标识符在internet.private.enterprises组（1.3.6.1.4.1）中，在Assigned Numbers RFC文档中可以确定下一个对象标识符（12）肯定是指派给了厂家（Epilogue）。

同时还可以看出，sysServices变量的值是4与8的和，它支持网络层（例如选路）和运输层的应用（例如端到端）。

25.8.2　interface组

在本组中只定义了一个简单变量，那就是系统的接口数量，如图25-17所示。

名　称	数据类型	R/W	描述
ifNumber	INTEGER		系统上的网络接口数

图25-17　if组中的简单变量

在该组中，还有一个表格变量，有 22列。表格中的每行定义了接口的一些特征参数。如图25-18所示。

可以向主机sun查询所有这些接口的变量。如3.8节中所示，我们还是希望访问三个接口，如果SLIP接口已经启动：

```
sun % snmpi -a sun                                  首先看第一个接口的接口索引

snmpi> next ifTable
ifIndex.1=1

snmpi> get ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.1="le0"
ifType.1=ethernet-csmacd(6)
ifMtu.1=1500
ifSpeed.1=10000000
ifPhysAddress.1=0x08:00:20:03:f6:42

snmpi> next ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.2="sl0"
ifType.2=propPointToPointSerial(22)
ifMtu.2=552
ifSpeed.2=0
ifPhysAddress.2=0x00:00:00:00:00:00

snmpi> next ifDescr.2 ifType.2 ifMtu.2 ifSpeed.2 ifPhysAddress.2
ifDescr.3="lo0"
ifType.3=softwareLoopback(24)
ifMtu.3=1536
ifSpeed.3=0
ifPhysAddress.3=0x00:00:00:00:00:00
```

Linux公社 www.linuxidc.com

接口表，索引＝<*IfIndex*>			
名称	数据类型	R/W	描　述
ifIndex	INTEGER		接口索引，介于1和ifNumber之间
ifDescr	DisplayString		接口的文字描述
ifType	INTEGER		类型，例如：6 = 以太网，7 = 802.3以太网，9 = 802.5令牌环，23 = PPP，28 = SLIP,还有其他一些值
ifMtu	INTEGER		接口的MTU
ifSpeed	Gauge		以b/s为单位的速率
ifPhysAddress	PhysAddress		物理地址，对无物理地址的接口，以一串0表示（例如，串行链路）
ifAdminStatus	[1…3]	•	所希望的接口状态：1= 工作，2 = 不工作，3 = 测试
ifOperStatus	[1…3]	•	当前接口的状态：1= 工作，2 = 不工作，3 = 测试
ifLastChange	TimeTicks		当接口进入目前运行状态时sysUpTime的值
ifInOctets	Counter		收到的字节总数，包括组帧字符
ifInUcastPkts	Counter		交付给高层的单播分组数
ifInNUcastPkts	Counter		交付给高层的非单播（例如，广播或多播）分组数
ifInDiscards	Counter		收到的被丢弃的分组数，即使在分组中无差错（例如，无缓存空间）
ifInErrors	Counter		收到的由于差错被丢弃的分组数
ifInUnknownProtos	Counter		收到的由于未知的协议被丢弃的分组数
ifOutOctets	Counter		发送的字节总数，包括组帧字符
ifOutUcastPkts	Counter		从高层接收到的单播分组数
ifOutNUcastPkts	Counter		从高层接收到的非单播（如广播或多播）分组数
ifOutDiscards	Counter		发出的被丢弃的分组数，即使在分组中无差错（如无缓存空间）
ifOutErrors	Counter		发出的由于差错被丢弃的分组数
ifOutQLen	Gauge		在输出队列中的分组数
ifSpecific	ObjectID		对这种特定媒体类型的MIB定义的引用

图25-18　在接口表中的变量：ifTable

对于第1个接口，采用get操作提取5个变量值，然后用get-next操作提取第2个接口的相同的5个参数。对于第3个接口，同样采用get-next操作。

对于SLIP链路的接口类型，所报告的是一个点到点的专用串行链路，而不是 SLIP链路。此外，SLIP链路的速率没有报告。

这个例子对我们理解get-next操作和“先列后行”次序之间的关系十分重要。如果我们键入命令“next ifDescr.1”，则系统返回的是表格中的下一行所对应的变量，而不是同一行中的下个变量。如果表格是按照“先行后列”次序存放，我们就不能通过一个给定变量来读取下一个变量。

25.8.3　at组

地址转换组对于所有的系统都是必需的，但是在 MIB-II中已经没有这个组。从 MIB-II开

Linux公社 www.linuxidc.com

始，每个网络协议组（如IP组）都包含它们各自的网络地址转换表。例如对于IP组，网络地址转换表就是ipNetToMediaTable。

在该组中，仅有一个由3列组成的表格变量。如图25-19所示。

我们将用snmpi程序中的一个新命令来转储(dump)整个表格。向一个叫做kinetics的路由器（该路由器连接了一个TCP/IP网络和一个AppleTalk网络）查询其整个ARP高速缓存。命令的输出是字典式排序的整个表格内容。

地址转换表，索引＝<atIfIndex>.1.<atNetAddress>			
名　称	数据类型	R/W	描　　述
atIfIndex	INTEGER	•	接口数：ifIndex
atPhysAddress	PhysAddress	•	物理地址。若设置为长度为0的字符串，则表示无效表项
atNetAddress	NetworkAddress	•	IP地址

图25-19　网络地址转换表：atTable

```
sun % snmpi -a kinetics -c secret dump at

atIfIndex.1.1.140.252.1.4=1
atIfIndex.1.1.140.252.1.22=1
atIfIndex.1.1.140.252.1.183=1
atIfIndex.2.1.140.252.6.4=2
atIfIndex.2.1.140.252.6.6=2

atPhysAddress.1.1.140.252.1.4=0xaa:00:04:00:f4:14
atPhysAddress.1.1.140.252.1.22=0x08:00:20:0f:2d:38
atPhysAddress.1.1.140.252.1.183=0x00:80:ad:03:6a:80
atPhysAddress.2.1.140.252.6.4=0x00:02:16:48
atPhysAddress.2.1.140.252.6.6=0x00:02:3c:48

atNetAddress.1.1.140.252.1.4=140.252.1.4
atNetAddress.1.1.140.252.1.22=140.252.1.22
atNetAddress.1.1.140.252.1.183=140.252.1.183
atNetAddress.2.1.140.252.6.4=140.252.6.4
atNetAddress.2.1.140.252.6.6=140.252.6.6
```

让我们来分析一下用tcpdump命令时的分组交互情况。当snmpi要转储整个表格时，首先发出一条get-next命令以取得表格的名称（在本例中是at），该名称就是要获取的第一个表项。然后在屏幕上显示的同时生成另一条get-next命令。直到遍历完整个表格的内容后才终止。

图25-20显示了在路由器中实际表格的内容。

atIfIndex	atPhysAddress	atNetAddress
1	0xaa:00:04:00:f4:14	140.252.1.4
1	0x08:00:20:0f:2d:38	140.252.1.22
1	0x00:80:ad:03:6a:80	140.252.1.183
2	0x00:02:16:48	140.252.6.4
2	0x00:02:3c:48	140.252.6.6

图25-20　at表举例（ARP高速缓存）

注意图中，接口2的AppleTalk协议的物理地址是32 bit的数值，而不是我们所熟悉的以太网的48 bit物理地址。同时请注意，正如我们所希望的那样，在图中有一条记录和netb路由器（其IP地址是140.252.1.183）有关。这是因为netb路由器和kinetics路由器在同一个以太网中（140.252.1），而且kinetics路由器必需采用ARP来回送SNMP响应。

25.8.4　ip组

ip组定义了很多简单变量和3个表格变量。图25-21显示了所有的简单变量。

Linux公社 www.linuxidc.com

名称	数据类型	R/W	描　述
ipForwarding	[1…2]	•	1代表系统正在转发IP数据报，2则代表不在转发
ipDefaultTTL	INTEGER	•	当运输层不提供TTL值时的默认TTL值
ipInReceives	Counter		从所有接口收到的IP数据报的总数
ipInHdrErrors	Counter		由于首部差错被丢弃的数据报数（例如，检验和差错，版本不匹配，TTL超过等）
ipInAddrErrors	Counter		由于不正确的目的地址被丢弃的IP数据报数
ipForwDatagrams	Counter		曾进行过一次转发尝试的IP数据报数
ipInUnknownProtos	Counter		具有无效协议字段的发往本地的IP数据报数
ipInDiscards	Counter		由于缓存空间不足被丢弃的收到的数据报数
ipInDelivers	Counter		交付到适当的协议模块的IP数据报数
ipOutRequests	Counter		传递给IP层来传输的IP数据报总数。不包括已经在ipForwDatagrams中计入的那些
ipOutDiscards	Counter		由于缓存空间不足被丢弃的输出数据报数
ipOutNoRoutes	Counter		由于找不到路由被丢弃的数据报数
ipReasmTimeout	INTEGER		在等待重装时已收到的数据报片被保留的最大秒数
ipReasmReqds	Counter		收到的需要进行重装的IP数据报片的数目
ipReasmOKs	Counter		已成功重装的IP数据报数
ipReasmFails	Counter		IP重装算法失败次数
ipFragOKs	Counter		被成功分片的IP数据报数
ipFragFails	Counter		需要进行分片但由于设置了“不分片”标志而不能分片的IP数据报数
ipFragCreates	Counter		由分片而产生的IP数据报片的数目
ipRoutingDiscards	Counter		所选择的选路表项即使是有效的但也要丢弃的数目

图25-21　ip组中的简单变量

ip组中的第一个表格变量是IP地址表。系统的每个IP地址都对应该表格中的一行。每行中包含了5个变量，如图25-22所示。

IP地址表，索引＝<*ipAdEntAddr*>			
名　称	数据类型	R/W	描　述
ipAdEntAddr	IpAddress		这一行的IP地址
ipAdEntIfIndex	INTEGER		对应的接口数：ifIndex
ipAdEntNetMask	IpAddress		对这个IP地址的子网掩码
ipAdEntBcastAddr	[0…1]		IP广播地址中的最低位的值。通常为1
ipAdEntReasmMaxSize	[0…65535]		在这个接口上收到的、能够进行重装的、最长的IP数据报

图25-22　IP地址表：ipAddrTable

同样可以向主机sun查询整个IP地址表：

```
sun % snmpi -a sun dump ipAddrTable

ipAdEntAddr.127.0.0.1=127.0.0.1
ipAdEntAddr.140.252.1.29=140.252.1.29
ipAdEntAddr.140.252.13.33=140.252.13.33
```

Linux公社 www.linuxidc.com

```
ipAdEntIfIndex.127.0.0.1=3                环回接口
ipAdEntIfIndex.140.252.1.29=2             SLIP接口
ipAdEntIfIndex.140.252.13.33=1            以太网接口

ipAdEntNetMask.127.0.0.1=255.0.0.0
ipAdEntNetMask.140.252.1.29=255.255.255.0
ipAdEntNetMask.140.252.13.33=255.255.255.224

ipAdEntBcastAddr.127.0.0.1=1              所有这三个都使用一个比特进行广播
ipAdEntBcastAddr.140.252.1.29=1
ipAdEntBcastAddr.140.252.13.33=1

ipAdEntReasmMaxSize.127.0.0.1=65535
ipAdEntReasmMaxSize.140.252.1.29=65535
ipAdEntReasmMaxSize.140.252.13.33=65535
```

输出的接口号码可以和图25-18中的输出进行比较，同样IP地址和子网掩码可以和3.8节中采用ifconfig命令时的输出进行比较。

ip组中的第二个表是IP路由表（请回忆一下我们在9.2节中讲到的路由表），如图25-23所示。访问该表中每行记录的索引是目的IP地址。

名　称	数据类型	R/W	描　述
ipRouteDest	IpAddress	•	目的IP地址。值0.0.0.0表示一个默认的表项
ipRouteIfIndex	INTEGER	•	接口数：ifIndex
ipRouteMetric1	INTEGER	•	主要的选路度量。这个度量的意义取决于选路协议（ipRouteProto）。-1表示未使用
ipRouteMetric2	INTEGER	•	可选的选路度量
ipRouteMetric3	INTEGER	•	可选的选路度量
ipRouteMetric4	INTEGER	•	可选的选路度量
ipRouteNextHop	IpAddress	•	下一跳路由器的IP地址
ipRouteType	INTEGER	•	路由类型：1 = 其他，2 = 无效路由，3 = 直接，4= 间接
ipRouteProto	INTEGER		选路协议：1 = 其他，4 = ICMP重定向，8 = RIP, 13 = OSPF, 14 = BGP, 以及其他
ipRouteAge	INTEGER	•	自从路由上次被更新或确定是正确的以后所经历的秒数
ipRouteMask	IpAddress	•	在和ipRouteDest相比较之前，掩码要与目的IP地址进行逻辑“与”
ipRouteMetric5	INTEGER	•	其他的选路度量
ipRouteInfo	ObjectID	•	对这种特定选路协议的MIB定义的引用

图25-23　IP路由表：ipRouteTable

图25-24显示的是用snmpi程序采用dump ipRouteTable命令从主机sun得到的路由表。在这张表中，已经删除了所有5个路由度量，那是由于这5条记录的度量都是-1。在列的标题中，对每个变量名称已经删除了ipRoute这样的前缀。

Dest	IfIndex	NextHop	Type	Proto	Mask
0.0.0.0	2	140.252.1.183	间接(4)	其他(1)	0.0.0.0
127.0.0.1	3	127.0.0.1	直接(3)	其他(1)	255.255.255.255
140.252.1.183	2	140.252.1.29	直接(3)	其他(1)	255.255.255.255
140.252.13.32	1	140.252.13.33	直接(3)	其他(1)	255.255.0.0
140.252.13.65	1	140.252.13.35	间接(4)	其他(1)	255.255.255.255

图25-24　路由器sun上的IP路由表

Linux公社 www.linuxidc.com

为比较起见，下面的内容是我们用netstat命令（在9.2节中曾经讨论过）格式显示的路由表信息。图25-24是按字典序显示的，这和netstat命令显示格式不同：

```
sun % netstat -rn
Routing tables
Destination          Gateway              Flags    Refcnt Use       Interface
140.252.13.65        140.252.13.35        UGH      0      115       le0
127.0.0.1            127.0.0.1            UH       1      1107      lo0
140.252.1.183        140.252.1.29         UH       0      86        sl0
default              140.252.1.183        UG       2      1628      sl0
140.252.13.32        140.252.13.33        U        8      68359     le0
```

ip组的最后一个表是地址转换表，如图 25-25所示。正如我们前面所说的， at组已经被删除了，在这里已经用IP表来代替了。

IP地址转换表，索引＝<ipNetToMediaIfIndex>.<ipNetToMediaNetAddress>			
名　　称	数 据 类 型	R/W	描　　述
ipNetToMediaIfIndex	INTEGER	•	对应的接口：ifIndex
ipNetToMediaPhysAddress	PhysAddress	•	物理地址
ipNetToMediaNetAddress	IpAddress	•	IP地址
ipNetToMediaType	[1…4]	•	映射的类型：1 = 其他，2 = 无效的，3 = 动态的，4 = 静态的。

图25-25 IP地址转换表：ipNetToMediaTable

这里显示的是系统sun上的ARP高速缓存信息：

```
sun % arp -a
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

相应的SNMP输出：

```
sun % snmpi -a sun dump ipNetToMediaTable
ipNetToMediaIfIndex.1.140.252.13.34=1
ipNetToMediaIfIndex.1.140.252.13.35=1
ipNetToMediaPhysAddress.1.140.252.13.34=0x00:00:c0:c2:9b:26
ipNetToMediaPhysAddress.1.140.252.13.35=0x00:00:c0:6f:2d:40
ipNetToMediaNetAddress.1.140.252.13.34=140.252.13.34
ipNetToMediaNetAddress.1.140.252.13.35=140.252.13.35
ipNetToMediaType.1.140.252.13.34=dynamic(3)
ipNetToMediaType.1.140.252.13.35=dynamic(3)
```

25.8.5 icmp组

icmp组包含4个普通计数器变量（ICMP报文的输出和输入数量以及 ICMP差错报文的输入和输出数量）和22个其他ICMP报文数量的计数器：11个是输出计数器，另外11个是输入计数器。如图25-26所示。

对于有附加代码的 ICMP报文（请回忆一下图 6-3中，有15种报文代表目的不可达），SNMP没有为它们定义专门的计数器。

25.8.6 tcp组

图25-27显示的是tcp组中的简单变量。其中的很多变量和图 18-12描述的TCP状态有关。

Linux公社 www.linuxidc.com

名　称	数据类型	R/W	描　述
icmpInMsgs	Counter		收到的ICMP报文总数
icmpInErrors	Counter		收到的有差错的ICMP报文数（例如，无效的ICMP检验和）
icmpInDestUnreachs	Counter		收到的ICMP目的站不可达报文数
icmpInTimeExcds	Counter		收到的ICMP超时报文数
icmpInParmProbs	Counter		收到的ICMP参数问题报文数
icmpInSrcQuenchs	Counter		收到的ICMP源站抑制报文数
icmpInRedirects	Counter		收到的ICMP重定向报文数
icmpInEchos	Counter		收到的ICMP回显请求报文数
icmpInEchosReps	Counter		收到的ICMP回显应答报文数
icmpInTimeStamps	Counter		收到的ICMP时间戳请求报文数
icmpInTimeStampReps	Counter		收到的ICMP时间戳应答报文数
icmpInAddrMasks	Counter		收到的ICMP地址掩码请求报文数
icmpInAddrMaskReps	Counter		收到的ICMP地址掩码应答报文数
icmpOutMsgs	Counter		输出的ICMP报文总数
icmpOutErrors	Counter		由于在ICMP报文中有一个问题（例如，缓存空间不足）而未发送的ICMP报文数
icmpOutDestUnreachs	Counter		发送的ICMP目的站不可达报文数
icmpOutTimeExcds	Counter		发送的ICMP超时报文数
icmpOutParmProbs	Counter		发送的ICMP参数问题报文数
icmpOutSrcQuenchs	Counter		发送的ICMP源站抑制报文数
icmpOutRedirects	Counter		发送的ICMP重定向报文数
icmpOutEchos	Counter		发送的ICMP回显请求报文数
icmpOutEchosReps	Counter		发送的ICMP回显应答报文数
icmpOutTimeStamps	Counter		发送的ICMP时间戳请求报文数
icmpOutTimeStampReps	Counter		发送的ICMP时间戳应答报文数
icmpOutAddrMasks	Counter		发送的ICMP地址掩码请求报文数
icmpOutAddrMaskReps	Counter		发送的ICMP地址掩码应答报文数

图25-26　icmp 组中的简单变量

名　称	数据类型	R/W	描　述
tcpRtoAlgorithm	INTEGER		用来计算重传超时值的算法：1 =除下列值以外，2 = 固定的RTO, 3 = MIL-STD-1778附件B, 4 = Van Jacobson算法
tcpRtoMin	INTEGER		以毫秒计的最小重传超时值
tcpRtoMax	INTEGER		以毫秒计的最大重传超时值
tcpMaxConn	INTEGER		最大的TCP连接数。若为动态的，则值为-1
tcpActiveOpens	Counter		从CLOSED到SYN_SENT的状态变迁数
tcpPassiveOpens	Counter		从LISTEN到SYN_RCVD的状态变迁数
tcpAttempFails	Counter		从SYN_SENT或SYN_RCVD到CLOSED的状态变迁数,加上从SYN_RCVD到LISTEN的状态变迁数
tcpEstabResets	Counter		从ESTABLISHED或CLOSE_WAIT状态到CLOSED的状态变迁数
tcpCurrEstab	Gauge		当前在ESTABLISHED或CLOSE_WAIT状态的连接数
tcpInSegs	Counter		收到的报文段的总数
tcpOutSegs	Counter		发送的报文段的总数，但将仅包含重传字节的除外
tcpRetransSegs			重传的报文段的总数
tcpInErrs	Counter		收到的具有一个差错(如无效的检验和)的报文段总数
tcpOutRsts	Counter		具有RST标志置位的报文段的总数

图25-27　tcp 组中的简单变量

Linux公社 www.linuxidc.com

现在向系统sun查询一些tcp组变量：

```
sun % snmpi -a sun

snmpi> get tcpRtoAlgorithm.0 tcpRtoMin.0 tcpRtoMax.0 tcpMaxConn.0
tcpRtoAlgorithm.0=vanj(4)
tcpRtoMin.0=200
tcpRtoMax.0=12800
tcpMaxConn.0=-1
```

本系统（指SunOS4.1.3）使用的是Van Jacobson超时重传算法，超时定时器的范围在200 ms~12.8 s之间，并且对TCP连接数量没有特定的限制（这里的超时上限 12.8 s恐怕有错，因为我们在21章中曾经介绍大多数应用的超时上限是 64 s）。

tcp组还包括一个表格变量，即TCP连接表，如图25-28所示。对于每个TCP连接，都对应表格中的一条记录。每条记录包含5个变量：连接状态、本地IP地址、本地端口号、远端IP地址以及远端端口号。

索引＝<tcpConnLocalAddress>.<tcpConnLocalPort>.<tcpConnRemAddress>.<tcpConnRemPort>			
名　称	数据类型	R/W	描　述
tcpConnState	[1…12]	•	连接状态：1 = CLOSED, 2 = LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED, 6 = FIN_WAIT, 7 = FIN_WAIT, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT, 12 = 删除TCB。管理进程对此变量可以设置的唯一值就是12（例如，立即终止此连接）
tcpConnLocalAddress	IpAddress		本地IP地址。0.0.0.0代表监听进程愿意在任何接口接受连接
tcpConnLocalPort	[1…65535]		本地端口号
tcpConnRemAddress	IpAddress		远程IP地址
tcpConnRemPort	[1…65535]		远程端口号

图25-28 TCP连接表：tcpConnTable

让我们看一看在系统sun上的这个表。由于有许多服务器进程在监听这些连接，所以我们只显示该表的一部分内容。在转储全部表格的变量之前，我们必需先建立两条 TCP连接：

```
sun %rlogin gemini              gemini的IP地址是140.252.1.11
```

和

```
sun %rlogin localhost           IP地址应该是127.0.0.1
```

在所有的监听服务器进程中，我们仅仅列出了FTP服务器进程的情况，它使用21号端口。

```
sun % snmpi -a sun dump tcpConnTable

tcpConnState.0.0.0.0.21.0.0.0.0.0=listen(2)
tcpConnState.127.0.0.1.23.127.0.0.1.1415=established(5)
tcpConnState.127.0.0.1.1415.127.0.0.1.23=established(5)
tcpConnState.140.252.1.29.1023.140.252.1.11.513=established(5)

tcpConnLocalAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnLocalAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnLocalAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnLocalAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.29
```

Linux公社 www.linuxidc.com

```
tcpConnLocalPort.0.0.0.0.21.0.0.0.0.0=21
tcpConnLocalPort.127.0.0.1.23.127.0.0.1.1415=23
tcpConnLocalPort.127.0.0.1.1415.127.0.0.1.23=1415
tcpConnLocalPort.140.252.1.29.1023.140.252.1.11.513=1023

tcpConnRemAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnRemAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnRemAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnRemAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.11

tcpConnRemPort.0.0.0.0.21.0.0.0.0.0=0
tcpConnRemPort.127.0.0.1.23.127.0.0.1.1415=1415
tcpConnRemPort.127.0.0.1.1415.127.0.0.1.23=23
tcpConnRemPort.140.252.1.29.1023.140.252.1.11.513=513
```

对于rlogin到gemini，只显示一条记录，这是因为gemini是另外一个主机。而且我们仅仅能够看到连接的客户端信息（端口号是1023）。但是Telnet连接，客户端和服务器端都显示（客户端口号是1415，服务器端口号是23），这是因为这种连接通过环回接口。同时我们还可以看到，FTP监听服务器程序的本地IP地址是0.0.0.0。这表明它可以接受通过任何接口的连接。

25.9 其他一些例子

现在开始回答前面一些没有回答的问题，我们将用SNMP的知识进行解释。

25.9.1 接口MTU

回忆一下在11.6节的实验中，我们试图得出一条从netb到sun的SLIP连接的MTU。现在可以采用SNMP得到这个MTU。首先从IP路由表中取到SLIP连接（140.252.1.29）的接口号（ipRouteIfIndex），然后就可以用这个数值进入接口表并且取得想要的SLIP连接的MTU（通过SLIP的描述和数据类型）。

```
sun % snmpi -a netb -c secret

snmpi> get ipRouteIfIndex.140.252.1.29
ipRouteIfIndex.140.252.1.29=12
snmpi> get ifDescr.12 ifType.12 ifMtu.12
ifDescr.12="Telebit NetBlazer dynamic dial virtual interface"
ifType.12=other(1)
ifMtu.12=1500
```

可以看到，即使连接的类型是SLIP连接，但是MTU仍设置为以太网，其值为1500，目的可能是为了避免分片。

25.9.2 路由表

回忆一下在14.4节中，我们讨论了DNS如何进行地址排序的问题。当时我们介绍了从域名服务器返回的第1个IP地址是和客户有相同子网掩码的情况。还介绍了用其他的IP地址也会正常工作，但是效率比较低。现在我们从SNMP的角度来查阅路由表的入口，在这里将用到前面章节中和IP路由有关的很多相关知识。

路由器gemini是一个多接口主机，有两个以太网接口。首先确认一下两个接口都可以Telnet登录：

```
sun % telnet 140.252.1.11 daytime
Trying 140.252.1.11 ...
Connected to 140.252.1.11.
```

Linux公社 www.linuxidc.com

```
Escape character is '^]'.
Sat Mar 27 09:37:24 1993
Connection closed by foreign host.

sun % telnet 140.252.3.54 daytime
Trying 140.252.3.54 ...
Connected to 140.252.3.54.
Escape character is '^]'.
Sat Mar 27 09:37:35 1993
Connection closed by foreign host.
```

可以看出这两个地址的连接没有什么区别。现在我们采用 traceroute命令来看一下对于每个地址，是否有选路方面的不同：

```
sun % traceroute 140.252.1.11
traceroute to 140.252.1.11 (140.252.1.11), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  299 ms  234 ms  233 ms
 2  gemini (140.252.1.11)  233 ms  228 ms  234 ms

sun % traceroute 140.252.3.54
traceroute to 140.252.3.54 (140.252.3.54), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  245 ms  212 ms  234 ms
 2  swnrt (140.252.1.6)  233 ms  229 ms  234 ms
 3  gemini (140.252.3.54)  234 ms  233 ms  234 ms
```

可以看到：如果采用属于 140.252.3子网的地址，就多了额外的一跳。下面解释造成这个额外一跳的原因。

图25-29是系统的连接关系图。从traceroute命令的输出结果可以看出主机 gemini和路由器swnrt都连接了两个网段：140.252.3子网和140.252.1子网。

回忆一下在图4-6中，我们解释了路由器netb采用ARP代理进程，使得sun工作站好象是直接连接到140.252.1子网上的情况。我们忽略了 sun和netb之间SLIP连接的调制解调器，因为这和我们这里的讨论不相关。

在图25-29中，我们用虚线箭头画出了当 Telnet到140.252.3.54时的路径。返回的数据报怎么知道直接从gemini到netb，而不是从原路返回呢？我们采用在 8.5节中介绍过的，带有宽松选路特性的traceroute版本来解释：

```
sun % traceroute -g 140.252.3.54 sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  244 ms  256 ms  234 ms
 2  * * *
 3  gemini (140.252.3.54)  285 ms  227 ms  234 ms
 4  netb (140.252.1.183)  263 ms  259 ms  294 ms
 5  sun (140.252.13.33)  534 ms  498 ms  504 ms
```

当在命令中指明是宽松源站选路时， swnrt路由器就不再有响应。看一下前面没有指明源站选路的traceroute命令输出，可以看出swnrt路由器是事实上的第 2跳。超时数据必须这样设置的原因是：当数据报指定了宽松源站选路选项时，该路由器没有发生ICMP超时差错。所以在traceroute命令的输出中可以得出，返回路径是从gemini(TTL 3, 4和5)路由器直接到达netb路由器，而不通过swnrt路由器。

还剩下一个需要用SNMP来解释的问题就是：在netb路由器的路由表中，哪条信息代表寻径到140.252.3？该信息表示netb路由器把分组发送给swnrt而不是直接发送给gemini？用get命令来取下一跳路由器的值。

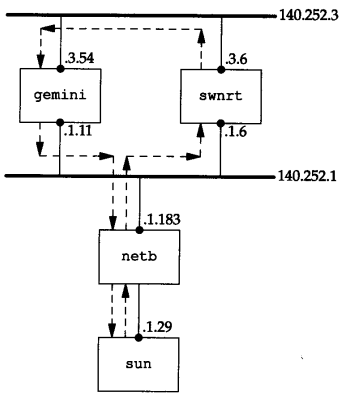


图25-29 例子中的网络拓扑结构

Linux公社 www.linuxidc.com

```
sun %snmpi -a netb -c secret get ipRouteNextHop.140.252.3.0
ipRouteNextHop.140.252.3.0=140.252.1.6
```

正如我们所看到发生的那样，路由表设置使得 netb路由器把分组发送到swnrt路由器。

为什么gemini路由器直接把分组回送给netb路由器？那是因为在gemini路由器端，它要回送的分组目的地址是140.252.1.29，而子网140.252.1是直接连接到gemini路由器上的。

从上面这个例子可以看出选路的策略。由于gemini是打算作一个多接口主机而不是路由器，所以默认的到140.253.3子网的路由器是swnrt。这是多接口主机和路由器之间差异的一个典型例子。

25.10 Trap

本章我们看到的例子都是从管理进程到代理进程的。当然代理进程也可以主动发送 trap到管理进程，以告诉管理进程在代理进程侧有某些管理进程所关心的事件发生，如图 25-1所示。trap发送到管理进程的162号端口。

在图25-2中，我们已经描述了 trap PDU的格式。在下面关于 tcpdump输出内容中我们将再一次用到这些字段。

现在已经定义了6种特定的trap类型，第7种trap类型是由供应商自己定义的特定类型。图25-30给出了trap报文中trap类型字段的内容。

trap类型	名　称	描　　述
0	coldStart	代理进程对自己初始化
1	warmStart	代理进程对自己重新初始化
2	linkDown	一个接口已经从工作状态改变为故障状态（图 25-18）。报文中的第一个变量标识此接口
3	linkUp	一个接口已经从故障状态改变为工作状态（图 25-18）。报文中的第一个变量标识此接口
4	authenticationFailure	从SNMP管理进程收到无效共同体的报文
5	egpNeighborLoss	一个EGP邻站已变为故障状态。报文中的第一个变量包含此邻站的IP地址
6	enterpriseSpecific	在这个特定的代码字段中查找trap信息

图25-30　trap的类型

用tcpdump命令来看看trap的情况。我们在系统sun上启动SNMP代理进程，然后让它产生coldStart类型的trap（我们告诉代理进程把trap信息发送到bsdi主机。虽然在该主机上并没有运行处理trap的管理进程，但是可以用tcpdump来查看产生了什么样的分组。回忆一下在图25-1中，trap是从代理进程发送到管理进程的，而管理进程不需要给代理进程发送确认。所以我们不需要trap的处理程序）。然后我们用snmpi程序发送一个请求，但该请求的共同体名称是无效的。这将产生一个authenticationFailure类型的trap。图25-31显示了命令的输出结果。

```
1  0.0              sun.snmp > bsdi.snmp-trap: C=traps Trap(28)
                    E:unix.1.2.5 [140.252.13.33] coldStart 20

2  18.86 (18.86)    sun.snmp > bsdi.snmp-trap: C=traps Trap(29)
                    E:unix.1.2.5 [140.252.13.33] authenticationFailure 1907
```

图25-31　tcpdump 输出的由SNMP代理进程产生的trap

首先注意一下两个 UDP数据报都是从 SNMP代理进程（端口是 161，图中显示的名称是snmp）发送到目的端口号是162的服务器进程上的（图中显示的名称是 snmp-trap）。

再注意一下C=traps是trap报文的共同体名称。这是ISODE SNMP代理进程的配置选项。

Linux公社 www.linuxidc.com

下一个要注意的是：第1行中的Trap（28）和第2行中的Trap（29）是PDU类型和长度。

两个输出行的第一项内容都是相同的 E:unix.1.2.5。这代表企业名字段和代理进程的sysObjiectID。它是图25-6中1.3.6.1.4.1（iso.org.dod.internet.private.enterparses）结点下面的某个结点，所以代理进程的对象标识是1.3.6.1.4.1.4.1.2.5。它的简称是：unix.agents.fourBSD-isode.5。最后一个数字“5”代表ISODE代理进程软件的版本号。这些企业名的值代表了产生trap的代理进程软件信息。

输出的下一项是代理进程的IP地址（140.252.13.33）。

在第1行中，trap的类型显示的是coldStart，第2行中，显示的是authenticationFailure。与之相对应的trap类型值是0和4（如图25-30所示）。由于这些都不是厂家自定义的trap，所以特定代码必须是0，在图中没有显示。

输出的最后分别是20和1907，这是时间戳字段。它是TimeTicks类型的值，表示从代理进程初始化开始到trap发生所经历了多少个百分之一秒。在冷启动 trap的情况下，在代理进程初始化后到trap的产生共经历了200 ms。同样，tcpdump的输出表明第2个trap在第1个trap产生的18.86s后出现，这对应于打印出的1907个百分之一秒减去200 ms所得到的值。

图25-2表明trap报文还包含很多代理进程发送给管理进程的变量，但在这些在例子中没有再讨论。

25.11 ASN.1和BER

在正式的SNMP规范中都是采用ASN.1（Abstract Syntax Notation 1）语法，并且在SNMP报文中比特的编码采用BER（Basic Encoding Rule）。和其他介绍SNMP的书不同，我们有目的地把ASN.1和BER的讨论放到最后。因为如果放在前面讨论，有可能使读者产生混淆而忽略了SNMP的真正目的是进行网络管理。在这里我们也只是对这两个概念简单地进行解释，[Rose 1990]的第8章详细讨论了ASN.1和BER。

ASN.1是一种描述数据和数据特征的正式语言。它和数据的存储及编码无关。 MIB和SNMP报文中的所有的字段都是用 ASN.1描述的。例如：对于 SMI中的ipAddress数据类型，ASN.1是这样描述的：

```
IpAddress ::=
     [APPLICATION 0]              -- in network-byte order
          IMPLICIT OCTET STRING (SIZE (4))
```

同样，在MIB中，简单变量的定义是这样描述的：

```
udpNoPorts OBJECT-TYPE
    SYNTAX  Counter
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The total number of received UDP datagrams for which
            there was no application at the destination port."
    ::= { udp 2 }
```

用SEQUENCE和SEQUENCE OF来定义表格的描述更加复杂。

当有了这样的ASN.1定义，可以有多种编码方法把数据编码为传输的比特流。 SNMP使用的编码方法是BER。例如，对于一个简单的整数如64，在BER中需要用3个字节来表示。第一个字节说明类型是一个整数，下个字节说明用了多少个字节来存储该整数（在这里是 1），最后一个字节才是该整数的值。

幸运的是，ASN.1和BER这两个繁琐的概念仅仅在实现SNMP的时候才重要，对我们理解

Linux公社 www.linuxidc.com

网络管理的概念和流程并没有太大的关系。

25.12 SNMPv2

在1993年，发表了定义新版本SNMP的11个RFC。RFC 1441 [Case et al. 1993]是其中的第一个，它系统地介绍了SNMPv2。同样，有两本书 [Stallings 1993; Rose 1994]也对SNMPv2进行了介绍。现在已经有两个SNMPv2的基本模型（参见附录 B.3中的[Rose 1994]），但是厂家的实现到1994年才能广泛使用。

在本节中，我们主要介绍SNMPv1和SNMPv2之间的重要区别。

1) 在SNMPv2中定义了一个新的分组类型`get-bulk-request`，它高效率地从代理进程读取大块数据。

2) 另的一个新的分组类型是`inform-request`，它使一个管理进程可以向另一个管理进程发送信息。

3) 定义了两个新的MIB，它们是：SNMPv2 MIB和SNMPv2-M2M MIB（管理进程到管理进程的MIB）。

4) SNMPv2的安全性比SNMPv1大有提高。在SNMPv1中，从管理进程到代理进程的共同体名称是以明文方式传送的。而SNMP v2可以提供鉴别和加密。

厂家提供的设备支持SNMPv2的会越来越多，管理站将对两个版本的 SNMP代理进程进行管理。[Routhier 1993]中描述了如何将SNMPv1的实现扩展到支持SNMPv2。

25.13 小结

SNMP是一种简单的、SNMP管理进程和SNMP代理进程之间的请求-应答协议。MIB定义了所有代理进程所包含的、能够被管理进程查询和设置的变量，这些变量的数据类型并不多。

所有这些变量都以对象标识符进行标识，这些对象标识符构成了一个层次命名结构，由一长串的数字组成，但通常缩写成人们阅读方便的简单名字。一个变量的特定实例可以用附加在这个对象标识符后面的一个实例来标识。

很多SNMP变量是以表格形式体现的。它们有固定的栏目，但有多少条记录并不固定。对于SNMP来讲，重要的是对表格中的每一行如何进行标识（尤其当我们不知道表格中有多少条记录时）以及如何按字典方式进行排序（“先列后行”的次序）。最后要说明的一点是：SNMP的`get-next`操作符对任何SNMP管理进程来讲都是最基本的操作。

然后我们介绍了下列的SNMP变量组：system、interface、address translation、IP、ICMP、TCP和UDP。接着是两个例子，一个介绍如何确定一个接口的MTU，另外一个介绍如何获取路由器的路由信息。

在本章的后面介绍了SNMP的trap操作，它是当代理进程发生了某些重大事件后主动向管理进程报告的。最后我们简单介绍了 ASN.1和BER，这两个概念比较繁琐，但所幸的是，它对我们了解SNMP并不十分重要，仅仅在实现SNMP的时候才要用到。

习题

25.1 我们说过采用两个不同的UDP端口（161和162）可以使得一个系统既可以是管理进程，也可以是代理进程。如果对管理进程和代理进程采用一个同样的端口，会出现什么情况？

25.2 用`get-next`操作，如何列出一张路由表的完整信息？

Linux公社 www.linuxidc.com

第26章　Telnet和Rlogin：远程登录

26.1　引言

远程登录（Remote Login）是Internet上最广泛的应用之一。我们可以先登录（即注册）到一台主机然后再通过网络远程登录到任何其他一台网络主机上去，而不需要为每一台主机连接一个硬件终端（当然必须有登录帐号）。

在TCP/IP网络上，有两种应用提供远程登录功能。

1) Telnet是标准的提供远程登录功能的应用，几乎每个 TCP/IP的实现都提供这个功能。它能够运行在不同操作系统的主机之间。Telnet通过客户进程和服务器进程之间的选项协商机制，从而确定通信双方可以提供的功能特性。

2) Rlogin起源于伯克利Unix，开始它只能工作在 Unix系统之间，现在已经可以在其他操作系统上运行。

在本章中，我们将介绍Telnet和Rlogin。首先介绍Rlogin，因为Rlogin比较简单。

> Telnet是一种最老的Internet应用，起源于1969年的ARPANET。它的名字是“电信网络协议（telecommunication network protocol）”的缩写词。

远程登录采用客户 - 服务器模式。图 26-1显示的是一个Telnet客户和服务器的典型连接图（对于Rlogin的客户和服务器连接图，我们可以画得更加简单）。

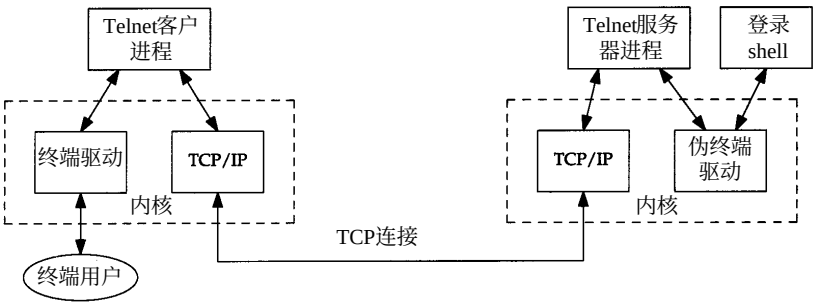


图26-1　客户-服务器模式的Telnet简图

在这张图中，有以下要点需要注意：

1) Telnet客户进程同时和终端用户和 TCP/IP协议模块进行交互。通常我们所键入的任何信息的传输是通过TCP连接，连接的任何返回信息都输出到终端上。

2) Telnet服务器进程经常要和一种叫做“伪终端设备”（pseudo-terminal device）打交道，至少在Unix系统下是这样的。这就使得对于登录外壳 (shell)进程来讲，它是被Telnet服务器进程直接调用的，而且任何运行在登录外壳进程处的程序都感觉是直接和一个终端进行交互。对于像满屏编辑器这样的应用来讲，就像直接在和终端打交道一样。实际上，如何对服务器进程的登录外壳进程进行处理，使得它好像在直接和终端交互，往往是编写远程登录服务器

Linux公社 www.linuxidc.com

进程程序中最困难的方面之一。

3) 仅仅使用了一条TCP连接。由于客户进程必须多次和服务器进程进行通信（反之亦然），这就必然需要某些方法，来描绘在连接上传输的命令和用户数据。我们在后面的内容中会介绍Telnet和Rlogin是如何处理这个问题的。

4) 注意在图 26-1中，我们用虚线框把终端驱动进程和伪终端驱动进程框了起来。在TCP/IP实现中，虚线框的内容一般是操作系统内核的一部分。 Telnet客户进程和服务器进程一般只是属于用户应用程序。

5) 把服务器进程的登录外壳进程画出来的目的是为了说明：当我们想登录到系统的时候，必须要有一个帐号， Telnet和Rlogin都是如此。

对于Telnet和Rlogin，如果比较一下它们客户进程和服务器进程源代码的数量，就可以知道这两者的复杂程度。图 26-2显示了伯克利不同版本的 Telnet和Rlogin客户进程和服务器进程源代码的数量。

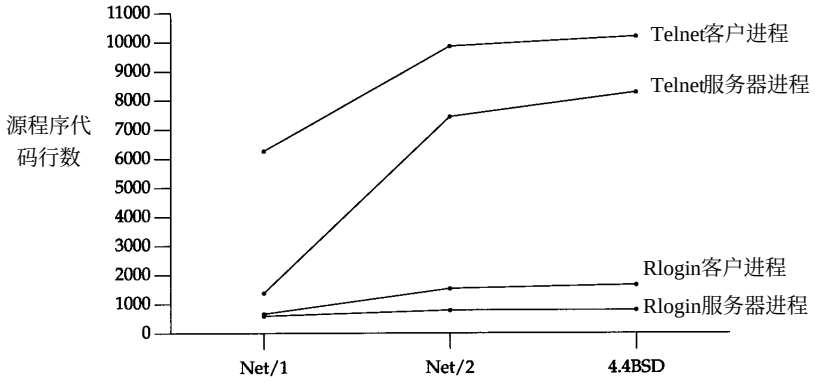


图26-2 Telnet/ Rlogin/客户进程/服务器进程的源代码数量比较

现在，不断有新的Telnet选项被添加到Telnet中去，这就使得Telnet实现的源代码数量大大增加，而Rlogin依然变化不大，还是比较简单。

远程登录不是那种有大量数据报传输的应用。正如我们前面讲到的一样，客户进程和服务器进程交互的分组大多比较小。 [Paxson 1993]发现客户进程发出的字节数（用户在终端上键入的信息）和服务器进程端发出的字节数的数量之比是 1:20。这是因为我们在终端上键入的一条短命令往往令服务器进程端产生很多输出。

26.2 Rlogin协议

Rlogin的第一次发布是在 4.2BSD中，当时它仅能实现 Unix主机之间的远程登录。这就使得Rlogin比Telnet简单。由于客户进程和服务器进程的操作系统预先都知道对方的操作系统类型，所以就不需要选项协商机制。在过去的几年中， Rlogin协议也派生出几种非 Unix环境的版本。

RFC 1282 [Kantor 1991]详细说明了Rlogin协议。类似于选路信息协议（RIP）的RFC，它是Rlogin用了许多年后才发布的。 [Stevens 1990]的第15章介绍了远程登录的客户进程及服务器进程端的编程，并且给出了 Rlogin的客户进程及服务器进程的完整源代码。 [Comer和Stevens 1993]的第25章和第26章给出了Telnet的客户进程的实现细节和源代码。

Linux公社 www.linuxidc.com

26.2.1 应用进程的启动

Rlogin的客户进程和服务器进程使用一个 TCP连接。当普通的TCP连接建立完毕之后，客户进程和服务器进程之间将发生下面所述的动作。

1) 客户进程给服务器进程发送4个字符串：(a) 一个字节的0；(b) 用户登录进客户进程主机的登录名，以一个字节的0结束；(c) 登录服务器进程端主机的登录名，以一个字节的0结束；(d) 用户终端类型名，紧跟一个正斜杠“/”，然后是终端速率，以一个字节的0结束。在这里需要两个登录名字，这是因为用户登录客户和服务器的名称有可能不一样。

   由于大多满屏应用程序需要知道终端类型，所以终端类型也必须发送到服务器进程。发送终端速率的原因是因为有些应用随着速率的改变，它的操作也有所变化。例如 vi编辑器，当速率比较小的时候，它的工作窗口也变小。所以它不能永远保持同样大小的窗口。

2) 服务器进程返回一个字节的0。

3) 服务器进程可以选择是否要求用户输入口令。这个步骤的数据交互没有什么特别的协议，而被当作是普通的数据进行传输。服务器进程给客户进程发送一个字符串（显示在客户进程的屏幕上），通常是password: 。如果在一定的限定时间内（通常是 60秒）客户进程没有输入口令，服务器进程将关闭该连接。

   通常可以在服务器进程的主目录(home directory)下生成一个文件（通常叫 .rhosts），该文件的某些行记录了一个主机名和用户名。如果从该文件中已经记录的主机上用已经记录的用户名进行登录，服务器进程将不提示我们输入口令。但是很多关于安全性的文献，如[Curry 1992]，强烈建议不要采用这种方法，因为这存在安全漏洞。

   如果提示输入口令，那么我们输入的口令将以明文的形式发送到服务器进程。我们所键入的每个字符都是以明文的格式传输的。所以某人只要能够截取网络上的原始传输的分组，他就可以截获用户口令。针对这个问题，新版本的 Rlogin客户程序，例如4.4BSD版本的客户程序，第一次采用了 Kerberos安全模型。Kerberos安全模型可以避免用户口令以明文的形式在网络上传输。当然，这要求服务器进程也支持 Kerberos（[Curry 1992]详细描述了Kerberos安全模型）。

4) 服务器进程通常要给客户进程发送请求，询问终端的窗口大小（将在后面解释）。

客户进程每次给服务器进程发送一个字节的内容，并且接收服务器进程的所有返回信息。这在19.2节中已经介绍过了。同样我们也采用了 Nagle算法（在19.4节中曾经介绍），该算法可以保证在速率较低的网络上，若干输入字节以单个 TCP报文段传输。操作其实很简单：用户键入的所有东西被发送到服务器，服务器发送给客户的任何信息返回到用户的屏幕上。

另外，服务器和客户之间还可以互相发送命令。在介绍这些命令之前，先介绍需要用到这些命令的场合。

26.2.2 流量控制

默认情况下，流量控制是由 Rlogin的客户进程完成的。客户进程能够识别用户键入的STOP和START的ASCII字符（Control_S和Control_Q），并且终止或启动终端的输出。

如果不是这样，每次我们为终止终端输出而键入的 Control_S字符将沿网络传输到服务器进程，这时服务器进程将停止往网络上写数据。但是在写操作终止之前，服务器进程可能已经往网络上写了一窗口的输出数据。也就是说，在输出停止之前，成千上万的数据字节还将

Linux公社 www.linuxidc.com

在屏幕上显示。图26-3显示了这个情况。

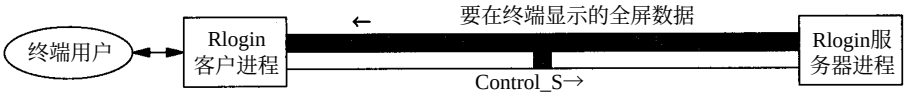


图26-3 服务器进程执行STOP/START的情况

对于一个交互式用户来讲，Control_S字符的响应延时是较大的。

有时候，服务器的应用程序需要解释输入的每个字节，但又不想让客户对它的输入内容进行处理，例如对控制字符如Control_S和Control_Q进行特殊处理（emacs编辑器就是这样的一个例子，它把Control_S和Control_C作为自己的命令）。解决这个问题的办法就是由服务器告诉客户是否要进行流量控制。

26.2.3 客户的中断键

当我们为中断服务器正在运行的进程而键入一个中断字符时（通常是DELETE或Control_C），会发生和流量控制相同的问题。这个情况和图26-3所示的类似，在一条TCP连接的管道上，从服务器进程向客户进程正在发送大量的数据，而客户进程同时在向服务器进程传输中断字符。而我们的本意是要中断字符尽快终止某个进程，使屏幕上不再有任何响应输出。

在流量控制和中断键这两种情况中，流量控制机制很少终止客户进程到服务器进程的数据流。这个方向仅仅包含我们键入的字符。所以对于从客户输出到服务器的特殊输入字符（Control_S和中断字符）不需要采用TCP的紧急方式（urgent mode）。

26.2.4 窗口大小的改变

如果是窗口风格的显示方式，当应用程序在运行的时候，我们还可以动态地改变窗口的大小。一些应用程序（典型的如那些操作整个窗口的应用程序，如全屏编辑器）需要知道窗口大小的变化。目前大多数Unix系统提供这种功能，可以告诉应用程序关于窗口大小的变化。

对于远程登录这种情况，窗口大小的变化发生在客户端，而运行在服务器端的应用程序需要知道窗口大小变化。所以Rlogin的客户需要采用某些方法来通知服务器窗口大小变化的情况以及新窗口的大小。

26.2.5 服务器到客户的命令

现在我们介绍通过TCP连接，Rlogin服务器进程可以发送给客户进程的4条命令。问题是只有一条TCP连接可供使用，所以服务器进程必须给这些命令字节做标记，使得客户进程可以从数据流中识别出这些是命令，而不是显示在终端上。所以我们将使用TCP的紧急方式（在20.8节中曾经介绍）。

当服务器要给客户发送命令时，服务器就进入紧急方式，并且把命令放在紧急数据的最后一个字节中。当客户进程收到这个紧急方式通知时，它从连接上读取数据并且保存起来，直到读到命令字节（即紧急数据的最后一个字节）。这时候客户进程根据读到的命令，再决定对于所读到并保存起来的数据是显示在终端上还是丢弃它。图26-4介绍了这4个命令。

采用TCP紧急方式发送这些命令的一个原因是第一个命令（“清仓输出(flush output)”）需要立即发送给客户，即使服务器到客户的数据流被窗口流量控制所终止。这种情况下，即服

Linux公社 www.linuxidc.com

务器到客户的输出被流量控制所终止的情况是经常发生的，这是因为运行在服务器的进程的输出速率通常大于客户终端的显示速率。另一方面，客户到服务器的数据流很少被流量控制所终止，因为这个方向的数据流仅仅包含用户所键入的字符。

字节	描　述
0x02	清仓输出。客户丢弃所有从服务器收到的数据，直到命令字节（紧急数据的最后一个字节）。客户还丢弃任何有可能被缓存的挂起输出（pending output）。当服务器收到客户发出的中断命令时，就发送此命令
0x10	客户停止执行流量控制
0x20	客户继续进行流量控制处理
0x80	客户立即响应，将当前窗口大小发送给服务器，并在今后当窗口大小变化时通知服务器。通常，当连接建立后，服务器就立即发送这个命令

图26-4　服务器到客户的Rlogin命令

回忆一下图20-14中的例子，在那里我们介绍了即使窗口大小是 0时，紧急通知通过网络进行传输的情况（在下节中，我们还将介绍一个类似的例子）。其他的3个命令实时性并不特别强，但为了简单起见，也采用了和第一个命令相同的技术。

26.2.6　客户到服务器的命令

对于客户到服务器的命令，只定义了一条命令，那就是：将当前窗口大小发送给服务器。当客户的窗口大小发生变化时，客户并不立即向服务器报告，除非收到了服务器发来的 0x80命令（图26-4中有介绍）。

同样，由于只存在一条 TCP连接，客户必须对在连接上传输的该命令字节进行标注，使得服务器可以从数据流中识别出命令，而不是把它发送到上层的应用程序中去。处理的方法就是在两个字节的0xff后面紧跟着发送两个特殊的标志字节。

对于窗口大小命令，两个标志字节是 ASCII码的字符‘ s ’。之后是4个16 bit长的数据（按网络字节顺序），分别是：行数（例如， 25），每列的字符数（例如， 80），X方向的像素数量，Y方向的像素数量。通常情况下，后两个 16bit是0，因为在Rlogin服务器进程调用的应用程序中，通常是以字符为单位来度量屏幕的，而不是像素点。

上面我们介绍的从客户进程到服务器进程的命令采用带内信令（ in-band signaling），这是因为命令字节和其他的普通数据一起传输。选择 0xff字节来表示这个带内信令的原因是：一般用户的操作不会产生0xff这个字节。所以说Rlogin是不完备的，如果我们采用某种方法，使得通过键盘就可以产生两个连续的 0xff字节，而且正好在这之前是两个 ASCII的‘ s ’字符，那么下面的8个字节就会被误认为是窗口大小了。

图26-4中介绍的是从服务器到客户的 Rlogin命令，由于大多数的 API采用的技术叫做“带外数据（out-of-band data）”，所以我们就称它为带外信令(out-of-band signaling)。但是回忆一下在20.8节中对TCP紧急方式的讨论，在那里我们说紧急方式数据不是带外数据，命令字节是按照普通数据流进行传输的，特殊之处是采用了紧急指针。

既然带内信令被用来传输从客户到服务器的命令，那么服务器进程必须检查从客户进程收到的每个字节，看看是否有两个连续的 0xff字节。但是对于采用带外信令的、从服务器传输到到客户的命令，客户进程不需要检查收到的每个字节，除非服务器进程进入了紧急方

Linux公社 www.linuxidc.com

式。即使在紧急方式下，客户进程也仅仅需要留意紧急指针所指向的字节。而且由于从客户进程到服务器的数据流量和相反方向的数据流量之比是 1:20，这就暗示带内信令适合于数据量比较小的情况（从客户到服务器），而带外信令适合于数据量比较大的情况（从服务器到客户）。

26.2.7 客户的转义符

通常情况下，我们向Rlogin客户进程键入的信息将传输到服务器进程。但是有些时候，我们并不需要把键入的信息传输到服务器，而是要和 Rlogin客户进程直接通信。方法是在一行的开头键入代字符(tilde)“ ~ ”，紧跟着是下列4个字符之一：

1) 以一个句号结束客户进程。

2) 以文件结束符（通常是Control_D）结束客户进程。

3) 以任务控制挂起符（通常是Control_Z）挂起客户进程。

4) 以任务控制延迟挂起符（通常是 Control_Y）来挂起仅仅是客户进程的输入。这时，不管客户运行什么程序，键入的任何信息将由该程序进行解释，但是从服务器发送到客户的信息还是输出到终端上。这非常适合当我们需要在服务器上运行一个长时间程序的场合，我们既想知道该程序的输出结果，同时还想在客户上运行其他程序。

只有当客户进程的Unix系统支持任务控制时，后两个命令才有效。

26.3 Rlogin的例子

在这里举两个例子：第一个是当 Rlogin会话建立的时候，客户和服务器的协议交互；从第二个例子可以看到，当用户键入中断键以取消正在服务器运行的程序时，服务器将产生很多输出。在图19-2中，我们给出了通常情况下， Rlogin会话上的数据流交互情况。

26.3.1 初始的客户-服务器协议

图26-5显示的是从主机bsdi到服务器svr4的Rlogin建立一个连接时的时间系列(在图中，去掉了通常的TCP连接的建立过程，窗口通告以及服务类型信息)。

上节介绍的协议对应图中的报文段 1~9。客户发送一个字节的 0（报文段1）之后发送3个字符串（报文段 3）。在本例中，这 3个字符串分别是： rstevens（客户的登录名）、rstevens（服务器的登录名）和 ibmpc3/9600（终端类型和速率）。当服务器确认了这些信息后回送一个字节的0（报文段5）。

然后服务器发送窗口请求命令（报文段 7）。这是采用TCP紧急方式发送的，我们又一次看到一个实现(SVR4)采用较老的但更普通的解释，即紧急指针指明的序号是紧急数据的最后一个字节加1。客户回送12字节的数据：2字节的0xff，2字节的‘ s ’，4个16 bit长度的窗口数据。

下面的4个报文段（ 10，12，14和16）是由服务器发送的，是从服务器操作系统的问候(greeting)。之后报文段18是一个7字节长度的外壳进程提示符“ svr4% ”。

客户输入的信息如图 19-2所示，每次发送一个字节。客户和服务器都可以主动中断该连接。如果我们输入一个命令，让服务器的外壳程序终止运行，那么服务器将中断该连接。如果我们给Rlogin客户键入一个转移符（通常是一个“ ~ ”），紧跟着一个句点或者是一个文件结束符号，那么客户将主动关闭该连接。

Linux公社 www.linuxidc.com

图26-5中，客户进程的端口号是 1023，这是由IANA分配的（在1.9节中介绍）。Rlogin协议要求客户进程用小于1024的端口号，术语叫做保留端口。在Unix系统中，客户进程一般不能使用保留端口号，除非客户进程具有超级用户权限。这是客户进程和服务器进程相互鉴别的一部分，这种鉴别可以使得用户不需要口令而可以登录。[Stevens 1990]详细讨论了客户进程和服务器进程相互鉴别的过程和有关保留端口号的问题。

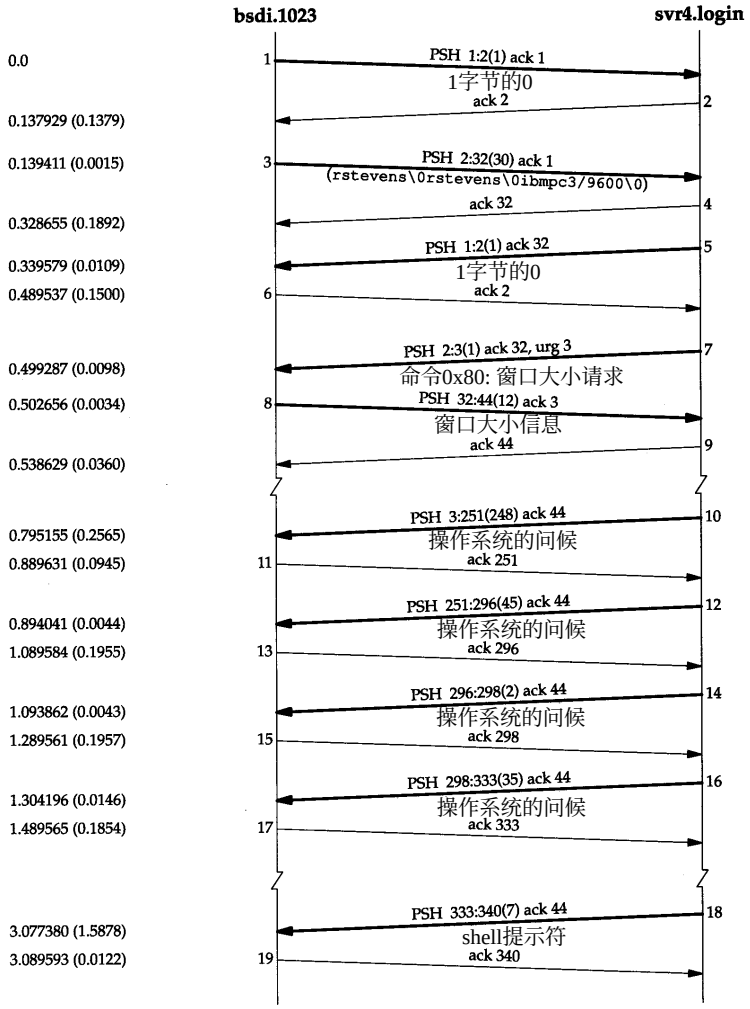


图26-5 Rlogin连接时间次序

26.3.2 客户中断键

让我们看一下另外一个例子，这个例子涉及到 TCP的紧急方式。当数据流已经终止时，我们键入中断键。这个例子要用到前面讲到的很多 TCP算法如：紧急方式、糊涂窗口避免技术、窗口流量控制和坚持计时器。在主机 sun上运行客户进程。我们登录到主机 bsdi，向终端输出一个大文本文件，然后键入 Control_S中断输出。当输出停止时，我们键入中断键（DELETE）以异常方式中止该进程。

Linux公社 www.linuxidc.com

```
sun %rlogin bsdi
                                        所有操作系统的问候
bsdi %cat /usr/share/misc/termcap       向终端输出大文件
                                        大量的终端输出
                                        键入Control_S以中断输出，
                                        然后等待直到输出停止
^?                                      键入中断键，而这被回显了，
bsdi %                                  然后输出了提示符
```

下面这些要点是关于客户、服务器和连接的状态的概述：

1) 键入Control_S以停止终端的输出。

2) 用户终端的输出缓存很快被填满，所以Rlogin的客户向终端的写操作被阻塞。

3) 此时客户也不能从网络连接上读取数据，所以客户的TCP接收缓存也将被填满。

4) 当接收缓存已满时，客户进程的TCP会向服务器进程的TCP通告现在的接收窗口是0。

5) 当服务器收到客户的窗口为0时，将停止向客户发送数据，这样，服务器的发送缓存也将被填满。

6) 由于发送缓存已满，所以Rlogin服务器进程将停止。这样，Rlogin服务器将不能从服务器运行的应用程序（cat）处读取数据。

7) 当cat程序的输出缓存也被填满时，cat也将停止。

8) 然后我们用中断键来终止服务器上的cat程序。这个命令从客户的TCP传输到服务器的TCP，这是因为该方向的数据传输没有被流量控制所终止。

9) cat应用程序收到中断命令并且终止。这使得它的输出缓存（也就是Rlogin服务器进程读取数据的地方）被清空，这将唤醒Rlogin服务器进程。然后Rlogin服务器进程进入紧急方式，向客户进程发送“清仓输出”命令（0x02）。

图26-6概括了从服务器到客户的数据流（图中的序号就是下面将介绍的图中的时间系列）。

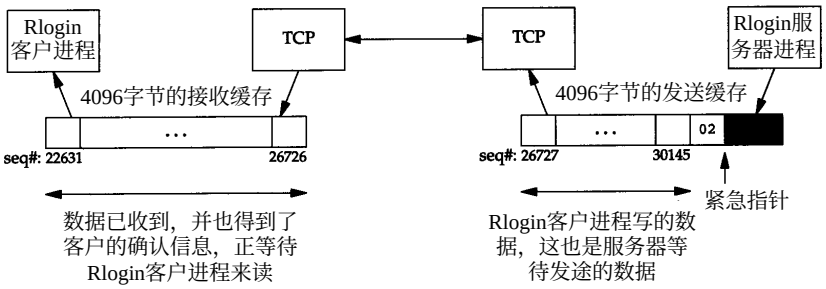


图26-6 Rlogin例子中，服务器进程到客户进程的数据流概述

发送缓存的阴影部分是4096字节的缓存中没有被使用的部分。图26-7是该例子的时间系列。

在报文段1~3，服务器进程向客户进程发送满长度（即1024字节）的TCP报文段。由于此时客户进程不能向终端写信息，客户进程也不能从网络上读数据，所以在报文段4中，客户进程向服务器进程发送ACK确认时，告诉服务器进程此时接收窗口是1024个字节。在报文段5中，服务器进程发送的数据长度就不再是满长度的了。同样，报文段6中客户进程的确认信号所带的接收窗口大小是此时接收缓存的空余字节长度。那么在报文段5中，客户进程ACK信号中为什么接收窗口大小是349而不是0呢？这是因为如果发送的是0（糊涂窗口避免技术），那么窗口指针将右边界移动到了左边界，而这是绝对不能发生的（见20.3节）。当服务器进程收到报

Linux公社 www.linuxidc.com

文段6的ACK信号后，它就不能再发送全长的数据报了，这时候它就采用糊涂窗口症避免技术，不发送任何东西，同时置一个 5秒的坚持计时器。当计时器超时，服务器进程就发送一个 349字节大小的数据（如报文段 7）。由于此时客户进程依然不能输出接收缓存的信息，所以接收缓存将被填满，客户进程将发送 ACK信号，此时接收窗口大小为0（如报文段8）。

这时候我们键入中断键并且以报文段 9显示的那样传输。此时的接收窗口大小依然为 0。当服务器进程接收到该中断键后，服务器进程把它发送给应用程序（ cat），应用程序就终止。由于应用程序被终端中断键所终止，应用程序就清空它的输出缓存。服务器进程发现该变化后就通过TCP紧急方式向客户进程发送“清仓输出”命令，这如报文段 10所示。注意命令字节0x02放在第30146字节中（紧急指针减1）。报文段10告诉客户进程在命令字节前还有3419个字节（从26727到30145）在服务器进程的发送缓存中等待发送。

报文段10采用紧急通知方式发送，包含了服务器进程向客户进程发送的下一个字节（序号是26727）。它不包含“清仓输出”命令字节。记得在 22.2节中曾经介绍过，发送进程可以发送一个字节的数据来试探对方的接收窗口是否关闭。报文段 10就是采用了这个原理。然后客户进程TCP就立即发送如报文段 11所示的数据。虽然此时接收窗口还是 0，但是在客户进程内部，由于客户进程的 TCP收到了对方的紧急通知，它把该通知告诉客户进程，客户进程就知道服务器进程已经进入了紧急方式了。

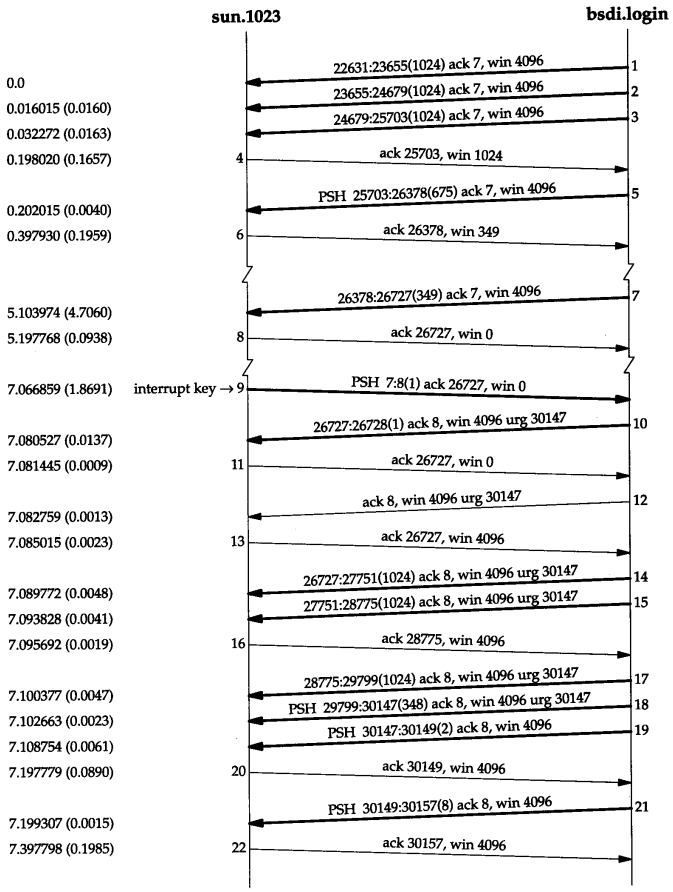


图26-7 Rlogin举例：当客户程停止输出然后终止服务器的应用程序的情况

Linux公社 www.linuxidc.com

当Rlogin客户进程从它的TCP收到了紧急通知，并且客户进程开始读取已经在输入缓存中等待被读取的数据时，接收窗口就会重新打开（报文段 13）。然后服务器进程就开始正常发送数据（报文段14, 15, 17和18）。注意报文段18的数据报中包含紧急数据的最后一个字节的数据（序号30146），该字节包含服务器进程发送给客户进程的命令字节。当客户进程收到该命令后，它就丢弃报文段14、15、17和18所收到的数据，并且清空终端的输出缓存。在报文段 19中的下两个字节是中断键的回显“ ^? ”。最后一个报文段（21）包含了客户进程的外壳提示符。

这个例子描述了当用户键入中断键后，连接的双方数据如何被存储的情况。如果这些动作仅仅丢弃在服务器的 3419个字节数据，而不丢弃已经在客户的 4096个字节的数据，那么这些已经在客户的终端输出缓存中的 4096字节数据将输出到终端上。

26.4 Telnet协议

Telnet协议可以工作在任何主机（例如，任何操作系统）或任何终端之间。 RFC 854 [Postel 和Reynolds 1983a]定义了该协议的规范，其中还定义了一种通用字符终端叫做网络虚拟终端NVT（Network Virtual Terminal）。NVT是虚拟设备，连接的双方，即客户机和服务器，都必须把它们的物理终端和 NVT进行相互转换。也就是说，不管客户进程终端是什么类型，操作系统必须把它转换为 NVT格式。同时，不管服务器进程的终端是什么类型，操作系统必须能够把NVT格式转换为终端所能够支持的格式。

NVT是带有键盘和打印机的字符设备。用户击键产生的数据被发送到服务器进程，服务器进程回送的响应则输出到打印机上。默认情况下，用户击键产生的数据是发送到打印机上的，但是我们可以看到这个选项是可以改变的。

26.4.1 NVT ASCII

术语NVT ASCII代表7比特的ASCII字符集，网间网协议族都使用 NVT ASCII。每个7比特的字符都以8比特格式发送，最高位比特为 0。

行结束符以两个字符 CR（回车）和紧接着的 LF（换行）这样的序列表示。以 \r\n来表示。单独的一个 CR也是以两个字符序列来表示，它们是 CR和紧接着的 NUL（字节 0），以 \r\0表示。

在下面的章节中可以看到， FTP, SMTP, Finger和Whois协议都以NVT ASCII来描述客户命令和服务器的响应。

26.4.2 Telnet命令

Telnet通信的两个方向都采用带内信令方式。字节 0xff（十进制的 255）叫做 IAC（interpret as command，意思是“ 作为命令来解释 ”）。该字节后面的一个字节才是命令字节。如果要发送数据255，就必须发送两个连续的字节 255（在前面一节中我们讲到数据流是 NVT ASCII，它们都是 7bit的格式，这就暗示着 255这个数据字节不能在 Telnet上传输。其实在 Telnet中有一个二进制选项，在 RFC856[Postel和Reynolds 1983b]中有定义，关于这点我们没有讨论，该选项允许数据以 8bit进行传输）。图26-8列出了所有的Telnet命令。

由于这些命令中很多命令很少用到，所以对于一些重要的命令，如果在下面章节的例子或叙述中遇到，我们再做解释。

Linux公社 www.linuxidc.com

名称	代码(十进制)	描　述
EOF	236	文件结束符
SUSP	237	挂起当前进程（作业控制）
ABORT	238	异常中止进程
EOR	239	记录结束符
SE	240	子选项结束
NOP	241	无操作
DM	242	数据标记
BRK	243	中断
IP	244	中断进程
AO	245	异常中止输出
AYT	246	对方是否还在运行？
EC	247	转义字符
EL	248	删除行
GA	249	继续进行
SB	250	子选项开始
WILL	251	选项协商（图26-9）
WONT	252	选项协商
DO	253	选项协商
DONT	254	选项协商
IAC	255	数据字节255

图26-8　当前一个字节是IAC（255）时的Telnet命令集

26.4.3　选项协商

虽然我们可以认为Telnet连接的双方都是NVT，但是实际上Telnet连接双方首先进行交互的信息是选项协商数据。选项协商是对称的，也就是说任何一方都可以主动发送选项协商请求给对方。

对于任何给定的选项，连接的任何一方都可以发送下面4种请求的任意一个请求。

1) WILL：发送方本身将激活(enable)选项。

2) DO：发送方想叫接收端激活选项。

3) WONT：发送方本身想禁止选项。

4) DON'T：发送方想让接收端去禁止选项。

	发送方		接收方	描　述
1.	WILL	→		发送方想激活选项
		←	DO	接收方说同意
2.	WILL	→		发送方想激活选项
		←	DONT	接收方说不同意
3.	DO	→		发送方想让接收方激活选项
		←	WILL	接收方说同意
4.	DO	→		发送方想让接收方激活选项
	WONT	←		接收方说不同意
5.	WONT	→		发送方想禁止选项
		←	DONT	接收方必须说同意
6.	DONT	→		发送方想让接收方禁止选项
		←	WONT	接收方必须说同意

图26-9　Telnet选项协商的6种情况

由于Telnet规则规定，对于激活选项请求（如1和2），有权同意或者不同意。而对于使选项失效请求（如3和4），必须同意。这样，4种请求就会组合出6种情况，如图26-9所示。

选项协商需要3个字节：一个IAC字节，接着一个字节是WILL，DO，WONT和DONT这四

Linux公社 www.linuxidc.com

者之一，最后一个ID字节指明激活或禁止选项。现在，有40多个选项是可以协商的。Assigned Number RFC文档中指明选项字节的值，并且一些相关的RFC文档描述了这些选项。图26-10显示了在本章中会出现的选项代码。

Telnet的选项协商机制和Telnet协议的大部分内容一样，是对称的。连接的双方都可以发起选项协商请求。但我们知道，远程登录不是对称的应用。客户进程完成某些任务，而服务器进程则完成其他一些任务。下面我们将看到，某些Telnet选项仅仅适合于客户进程（例如要求激活行模式方式），某些选项则仅仅适合于服务器进程。

选项标识(十进制)	名　称	RFC
1	回显	857
3	抑制继续进行	858
5	状态	859
6	定时标记	860
24	终端类型	1091
31	窗口大小	1073
32	终端速率	1079
33	远程流量控制	1372
34	行方式	1184
36	环境变量	1408

图26-10　本章中将讨论的Telnet选项代码

26.4.4　子选项协商

有些选项不是仅仅用“激活”或“禁止”就能够表达的。指定终端类型就是一个例子，客户进程必须发送用一个ASCII字符串来表示终端类型。为了处理这种选项，我们必须定义子选项协商机制。

在RFC1091[VanBokkelen 1989]中定义了如何表示终端类型这样的子选项协商机制。首先连接的某一方（通常是客户进程）发送3个字节的字符序列来请求激活该选项。

<IAC, WILL, 24>

这里的24（十进制）是终端类型选项的ID号。如果收端（通常是服务器进程）同意，那么响应数据是：

<IAC, DO, 24>

然后服务器进程再发送如下的字符串：

<IAC, SB, 24, 1, IAC, SE>

该字符串询问客户进程的终端类型。其中SB是子选项协商的起始命令标志。下一个字节的“24”代表这是终端类型选项的子选项（通常SB后面的选项值就是子选项所要提交的内容）。下一个字节的“1”表示“发送你的终端类型”。子选项协商的结束命令标志也是IAC，就像SB是起始命令标志一样。如果终端类型是ibmpc，客户进程的响应命令将是：

＜IAC, SB, 24, 0‘I’, ‘B’, ‘M’, ‘P’, ‘C’, IAC, SE＞

第4个字节“0”代表“我的终端类型是”（在Assigned Numbers RFC文档中有正式的关于终端类型的数值定义，但是最起码在Unix系统之间，终端类型可以用任何对方可理解的数据进行表示。只要这些数据在termcap或terminfo数据库中有定义）。在Telnet子选项协商过程中，终端类型用大写表示，当服务器收到该字符串后会自动转换为小写字符。

26.4.5　半双工、一次一字符、一次一行或行方式

对于大多数Telnet的服务器进程和客户进程，共有4种操作方式。

1. 半双工

Linux公社 www.linuxidc.com

这是Telnet的默认方式，但现在却很少使用。NVT默认是一个半双工设备，在接收用户输入之前，它必须从服务器进程获得GO AHEAD（GA）命令。用户的输入在本地回显，方向是从NVT键盘到NVT打印机，所以客户进程到服务器进程只能发送整行的数据。

虽然该方式适用于所有类型的终端设备，但是它不能充分发挥目前大量使用的支持全双工通信的终端功能。RFC 857 [Postel 和Reynolds 1983c]定义了ECHO选项，RFC 858 [Postel和Reynolds 1983d]定义了SUPPRESS GO AHEAD（抑制继续进行）选项。如果联合使用这两个选项，就可以支持下面将讨论的方式：带远程回显的一次一个字符的方式。

2. 一次一个字符方式

这和前面的Rlogin工作方式类似。我们所键入的每个字符都单独发送到服务器进程。服务器进程回显大多数的字符，除非服务器进程端的应用程序去掉了回显功能。

该方式的缺点也是显而易见的。当网络速度很慢，而且网络流量比较大的时候，那么回显的速度也会很慢。虽然如此，但目前大多数Telnet实现都把这种方式作为默认方式。

我们将看到，如果要进入这种方式，只要激活服务器进程的SUPPRESS GO AHEAD选项即可。这可以通过由客户进程发送DO SUPPRESS GO AHEAD（请求激活服务器进程的选项）请求完成，也可以通过服务器进程给客户进程发送WILL SUPPRESS GO AHEAD（服务器进程激活选项）请求来完成。服务器进程通常还会跟着发送WILL ECHO，以使回显功能有效。

3. 一次一行方式

该方式通常叫做准行方式（kludge line mode），该方式的实现是遵照RFC 858的。该RFC规定：如果要实现带远程回显的一次一个字符方式，ECHO选项和SUPPRESS GO AHEAD选项必须同时有效。准行方式采用这种方式来表示当两个选项的其中之一无效时，Telnet就是工作在一次一行方式。在下节中我们将介绍一个例子，可以看到如何协商进入该方式，并且当程序需要接收每个击键时如何使该方式失效。

4. 行方式

我们用这个术语代表实行方式选项，这是在RFC 1184[Borman 1990]中定义的。这个选项也是通过客户进程和服务器进程进行协商而确定的，它纠正了准行方式的所有缺陷。目前比较新的Telnet实现支持这种方式。

图26-11是不同的Telnet客户进程和服务器进程之间默认的操作方式。“char”表示一次一个字符方式，“kludge”表示准行方式，“linemode”表示如RFC 1184定义的实行方式。

客户端	服务器端					
	SunOS 4.1.3	Solaris 2.2	SVR4	AIX 3.2.2	BSD/386	4.4BSD
SunOS 4.1.3	char	char	char	char	kludge	kludge
Solaris 2.2	char	char	char	char	kludge	kludge
SVR4	char	char	char	char	kludge	kludge
AIX 3.2.2	char	char	char	char	kludge	kludge
BSD/386	char	char	char	char	linemode	linemode
4.4BSD	char	char	char	char	linemode	linemode

图26-11 不同的Telnet客户进程和服务器进程之间默认的操作方式

从图中可以看出，只有当客户进程和服务器进程都是BSD/386或4.4BSD的时候才支持实行方式。当服务器进程的操作系统是这两者之一时，如果客户进程不支持实行方式，才会协商进入准行方式。从图中还可以看出，其实任何类型的客户进程和服务器进程都支持准行方式，但是一般都不把它作为默认方式，除非服务器进程指定。

Linux公社 www.linuxidc.com

26.4.6 同步信号

Telnet以Data Mark命令（即图26-8中的DM）作为同步信号，该同步信号是以TCP紧急数据形式发送的。DM命令是随数据流传输的同步标志，它告诉收端回到正常的处理过程上来。Telnet的双方都可以发送该命令。

当一端收到对方已经进入了紧急方式的通知后，它将开始读数据流，一边读一边丢弃所读的数据，直到读到Telnet命令为止。紧急数据的最后一个字节就是DM字节。采用TCP紧急方式的原因就是：即使TCP数据流已经被TCP流量控制所终止，Telnet命令也可以在连接上传输。

在下节中我们将看到Telnet同步信号的使用情况。

26.4.7 客户的转义符

和Rlogin的客户进程一样，Telnet客户进程也可以使用户直接和客户进程进行交互，而不是被发送到服务器进程。通常客户的转义字符是Control_]（control键和右中括号键，通常以“^]”表示）。这使得客户进程显示它的提示符，通常是“`telnet>`”。这时候有很多命令可供用户选择，以改变连接的特性或打印某些信息。大多数的Unix客户进程提供“help”命令，该命令将显示所有可用的命令。

在下节中我们将看到客户进程转义的例子，以及此时可以输入的命令。

26.5 Telnet举例

在这里我们将介绍在三种不同的操作方式下Telnet选项协商的情况。这些方式包括：单字符方式、实行方式和准行方式。同样我们还将讨论当用户在服务器端按了中断键退出了一个正在运行的进程后，系统的运行情况。

26.5.1 单字符方式

首先介绍基本的单字符方式，该方式类似于Rlogin。用户在终端输入的每个字符都将由终端发送到服务器进程，服务器进程的响应也将以字符方式回显到终端上。在这里运行的是一个新的客户进程BSD/386，它试图激活很多新的选项，服务器进程还是运行老的SVR4，我们将看到很多选项被服务器拒绝。

为了看到服务器和客户机之间选项协商的内容，我们将激活客户进程的一个选项来显示所有的选项协商。同样我们运行`tcpdump`来获得数据报交换的时间次序。图26-12显示了这个交互会话。

在图中，我们已经对由SENT或RCVD开头的选项协商的每一步都进行了标注。关于每一步的解释如下：

1) 客户发起SUPPRESS GO AHEAD选项协商。由于GO AHEAD命令通常是由服务器发送给客户的，而且客户希望服务器激活该选项，因此该选项的请求方式是DO（由于激活这一选项将会禁止GA命令的发送，上述过程很容易让人混淆）。在第10行可以看到服务器进程同意该选项。

2) 客户进程要按照在RFC 1091[VanBokkelen 1989]中的定义发送终端类型。这对Unix类型的客户进程来讲是很普通的。因为客户进程要激活本地的选项，所以该选项的请求方式是WILL。

Linux公社 www.linuxidc.com

```
bsdi % telnet                                   调用客户进程，不带任何命令行选项

telnet> toggle options                          告诉客户进程显示所有的选项协商过程
Will show option processing.

telnet> open svr4                               现在和服务器建立连接
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.

SENT DO SUPPRESS GO AHEAD                       1. (后面将讨论的行序号)
SENT WILL TERMINAL TYPE                         2.
SENT WILL NAWS                                  3.
SENT WILL TSPEED                                4.
SENT WILL LFLOW                                 5.
SENT WILL LINEMODE                              6.
SENT WILL ENVIRON                               7.
SENT DO STATUS                                  8.
RCVD DO TERMINAL TYPE                           9.
RCVD WILL SUPPRESS GO AHEAD                     10.
RCVD DONT NAWS                                  11.
RCVD DONT TSPEED                                12.
RCVD DONT LFLOW                                 13.
RCVD DONT LINEMODE                              14.
RCVD DONT ENVIRON                               15.
RCVD WONT STATUS                                16.
RCVD IAC SB TERMINAL-TYPE SEND                  17.
SENT IAC SB TERMINAL-TYPE IS "IBMPC3"           18.
RCVD WILL ECHO                                  19.
SENT DO ECHO                                    20.
RCVD DO ECHO                                    21.
SENT WONT ECHO                                  22.

UNIX(r) System V Release 4.0 (svr4)

RCVD DONT ECHO                                  23.
login: rstevens                                 我们键入用户和口令，服务进程不回显这些数据，
Password:                                       然后操作系统问候输出，然后是外壳提示符
```

图26-12 Telnet双方选项协商的初始化过程

3) NAWS的意思是“协商窗口大小”，它在RFC 1073 [Waitzman]中有定义。如果服务器进程同意该选项（实际上不同意，见11行），客户进程就要发送终端窗口的行、列大小的子选项。而且只要窗口大小发生变化，客户进程随时都将向服务器进程发送这一子选项（这和图26-4中Rlogin的0x80命令类似）。

4) TSPEED选项允许发送方（通常是客户进程）发送它的终端速率，这在 RFC 1079 [Hedrick 1988b]中有定义。如果服务器进程同意（实际上不同意，见12行），客户进程将发送其发送速率和接收速率的子选项。

5) LFLOW代表“本地流量控制”，这在RFC1371 [Hedrick 和Borman 1992]中定义。客户进程给服务器进程发送该选项，表示客户进程希望用命令方式激活或禁止流量控制。如果服务器进程同意（实际上不同意，见13行），只要Control_S和Control_Q进程需要在客户进程和服务器进程进行切换，客户进程都要向服务器进程发送子选项（这类似于图 26-4中Rlogin的0x10和0x20命令）。正如在关于Rlogin的讨论中我们所提到的那样，由客户进程进行流量控制的效果比由服务器进程来完成要好。

6) LINEMODE代表在26.4中所说的实行方式。所有终端字符的处理由 Telnet客户进程完成（例如回格，删除行等），然后整行发送给服务器进程。在本节后面，我们将介绍一个例子。该选项同样被服务器进程拒绝，如14行所示。

7) ENVIRON选项允许客户进程把环境变量发送给服务器进程，这在 RFC 1408 [Borman

Linux公社 www.linuxidc.com

1993a]中有定义。这样就可以把客户进程的用户环境变量自动传播到服务器进程。在 15行，服务器进程拒绝该选项（Unix中的环境变量通常是大写字母，紧跟一个等号，然后是一个字符串值，当然这只是一个惯例而已）。默认情况下，BSD/386 Telnet客户进程发送两个环境变量：DISPLAY和PRINTER，前提是这两个变量已经定义并且有效。Telnet用户可以定义其他一些要发送的环境变量。

8) STATUS选项（RFC 859 [Postel 和Reynolds 1983e]中定义）允许连接的一方询问对方对Telnet选项目前状态的理解。在这个例子中，客户进程要求对方激活选项（DO）。如果服务器进程同意（实际上不同意，见16行），客户进程就可以要求服务器进程以子选项的形式发送它的状态值。

9) 这是服务器进程的第一个响应。服务器进程同意激活终端类型选项（几乎所有的 Unix类型的服务器进程都支持该选项）。但现在客户进程还不能立即发送它的终端类型。它必须要等到服务器进程用子选项的形式询问终端类型的时候才能够发送（17行）。

10) 服务器进程同意抑制发送GO AHEAD命令。

11) 服务器进程不同意客户进程发送它的窗口大小。

12) 服务器进程不同意客户进程发送它的终端速率。

13) 服务器进程不同意客户进程实施流量控制。

14) 服务器进程不同意客户进程激活行方式选项。

15) 服务器进程不同意客户进程发送环境变量。

16) 服务器进程不发送状态信息。

17) 这是服务器进程要求客户进程发送终端类型的子选项。

18) 客户进程把终端类型“IBMPC3”以6字节的字符串形式发送给服务器进程。

19) 服务器进程要求客户进程发起请求，要求服务器进程激活回显选项。这是本例中服务器进程第一次主动发起选项协商。

20) 客户进程同意由服务器进程实现回显功能。

21) 服务器进程要求客户进程实现回显功能。这个命令是多余的，它只是将前两行进行了交换。这是目前大多数Unix的Telnet服务器进程判断客户进程是否运行4.2BSD或更新的BSD版本时的一个方法。如果客户进程回送 WILL ECHO，就表明客户进程运行的是老版本的4.2BSD，不支持TCP的紧急方式（在这种情况下就不能采用TCP紧急方式）。

22) 客户进程回送WONT ECHO，表示它不是一台4.2BSD主机。

23) 对于客户进程回送的WONT ECHO，服务器进程以DONT ECHO作为响应。

图26-13显示的是本例中服务器进程和客户进程交互的时间系列（去掉了连接建立部分）。

报文段1包含了图26-12中的1~8行。该报文段中包含24个字节数据，每个选项占3个字节。这是客户进程发起的选项协商。该报文段显示多个Telnet选项可以打在一个TCP段中发送。

报文段3是图26-12中的第9行，即DO TERMINAL TYPE命令。报文段5包含下面的8个选项协商中服务器进程的响应，即图 26-12中的10~17行。该报文段的长度是27个字节，因为10~16行是常规选项，每个占3个字节，而17行的子选项部分占6个字节。报文段6包含12个字节，和18行对应，这是客户发送它的终端类型的子选项。

报文段8（53个字节）包含两个Telnet命令和47字节的输出数据。前面的两个服务器进程发送Telnet命令占6字节，：WILL ECHO和DO ECHO（19和21行）。后47个字节的数据是：

Linux公社 www.linuxidc.com

```
\r\n\r\nUNIX(r) System V Release 4.0 (svr4) \r\n\r\0\r\n\r\0
```

前面4个字节数据在字符串输出之前产生两个空行。两字节的字符序列“ \r\n ”在Telnet中被认为是换行命令，而两字节的字符序列“ \r\0 ”则被认为是回车命令。这表明数据和命令可以在一个数据段中传输。 Telnet服务器进程和客户进程必须扫描接收到的每个字符，寻找 IAC字节并执行它后续的命令。

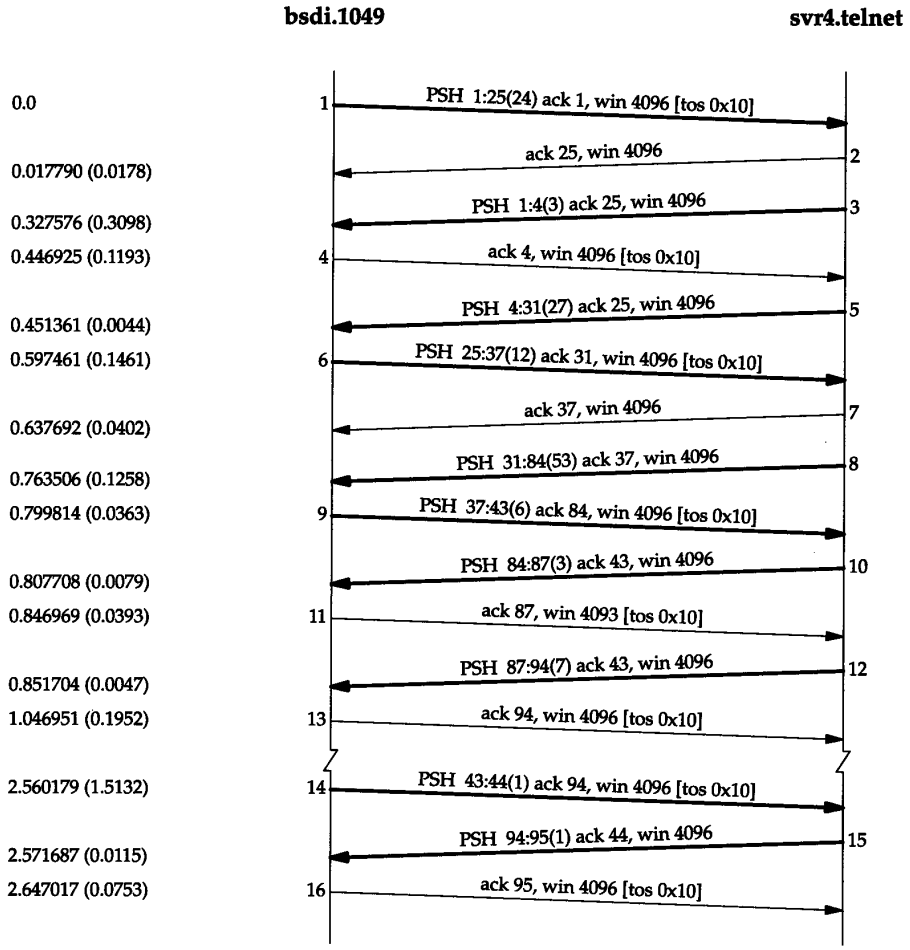


图26-13 Telnet服务器进程和客户进程选项协商初始化过程

报文段9包含客户进程发送的最后两个选项： 20和22行。23行是报文段10的响应，也是服务器进程发送的最后一个选项数据。

从现在开始，双方就可以交互数据了。当然在交互数据的过程中还可以进行选项协商，我们在该例子中就不多介绍了。报文段 12是服务器进程发送的提示符“ login: ”。报文段14是用户输入的登录用户名的第一个字符，它的回显在报文段 15中。这和我们在 19.2节中介绍的Rlogin交互类似：客户进程每次发送一个字符，服务器进程完成回显工作。

图26-12中的选项协商由客户进程初始化的，但是在本书中我们已经介绍了用Telnet客户进程连接某些标准服务器进程如：日间（daytime）服务器、回显(echo)服务器等情况。当然我们介绍这些的目的是为了描述 TCP的各种特性。但考察这些例子中

Linux公社 www.linuxidc.com

的分组交换，如图18-1，我们并没有看到客户进程发起的选项协商。为什么？这是因为在Unix系统中，除非使用标准的Telnet端口号23，否则客户进程不进行选项协商。这个特性使得Telnet客户进程可以使用标准的NVT同其他一些非Telnet服务器进程交换数据。我们已经在日间服务器、回显服务器和丢弃(discard)服务器中使用了这个特性，在后面章节介绍FTP和SMTP服务器的时候我们还将使用该特性。

26.5.2 行方式

为了描述Telnet的行方式选项协商过程，我们在主机 bsdi运行客户进程，服务器是位于`vangogh.vs.berkeley.edu`节点运行4.4BSD操作系统的一台主机。BSD/386和4.4BSD都支持这个选项。

我们不详细讨论所有的报文、选项和子选项协商过程，因为这个过程和前面的例子类似，而且对于行方式选项我们已经论述得比较清楚。下面我们仅仅讨论在选项协商中的一些区别。

1) 对于BSD/386希望协商的选项例如：窗口大小、本地流量控制、状态、环境变量和终端速率等，4.4BSD服务器进程都支持。
2) 4.4BSD服务器进程将协商一个BSD/386客户进程不支持的新选项：鉴别（为避免以明文形式在网络上传输用户口令）。
3) 和上个例子一样，客户进程发送WILL LINEMODE选项，由于服务器进程支持该选项，所以服务器进程发送DO LINEMODE。此时客户进程以子选项形式给服务器进程发送16个特定字符。这些字符是能影响客户进程的特定终端字符值：如中断字符，文件结束符等。服务器进程给客户进程发送一个子选项，让客户进程处理所有的输入，执行所有的编辑功能（删除字符，删除行等）。客户进程把除控制字符以外的字符以行的形式发送给服务器进程。服务器进程还要求客户进程把所有中断键和信号键转换为相应的 Telnet字符。例如中断键是Control_C，我们可以按Control_C来中断服务器端的某个进程。客户进程必须把Control_C转换为Telnet的IP命令（<IAC, IP>）传输给服务器进程。
4) 当用户输入口令时情况也有所不同。在 Rlogin和一次一字符方式的Telnet中，都是由服务器进程负责回显，所以当服务器进程读到口令时，它并不回显这些字符。但在行方式中由客户进程负责回显。下面这些交互过程将处理这种情况：
   (a) 服务器进程发送WILL ECHO，以告诉客户进程：服务器进程将处理回显。
   (b) 客户进程回送DO ECHO。
   (c) 服务器进程向客户进程发送字符串`Password:`，客户进程把它发送到终端上。
   (d) 然后用户输入口令，当用户按下RETURN键的时候，客户进程把口令发送给服务器进程。此时口令不回显，因为客户进程认为服务器进程将回显它。
   (e) 由于口令的结束符RETURN没有回显，服务器进程发送两字节字符序列 CR和LF以移动光标。
   (f) 服务器进程发送WONT ECHO。
   (g) 客户进程回送DONT ECHO。然后继续由客户进程负责回显。

一旦登录完成，客户进程将把数据以整行的方式发送给服务器进程。这就是行方式选项的目的。行方式大大地减少了客户进程和服务器进程之间的数据交互数量，而且对于用户的

Linux公社 www.linuxidc.com

击键（也就是回显和编辑）提供更快的响应。图 26-14显示的是当我们输入命令时，在行方式连接下分组交换的情况。

```
Vangogh %date
```

（去掉了业务种类信息和窗口通告信息）。

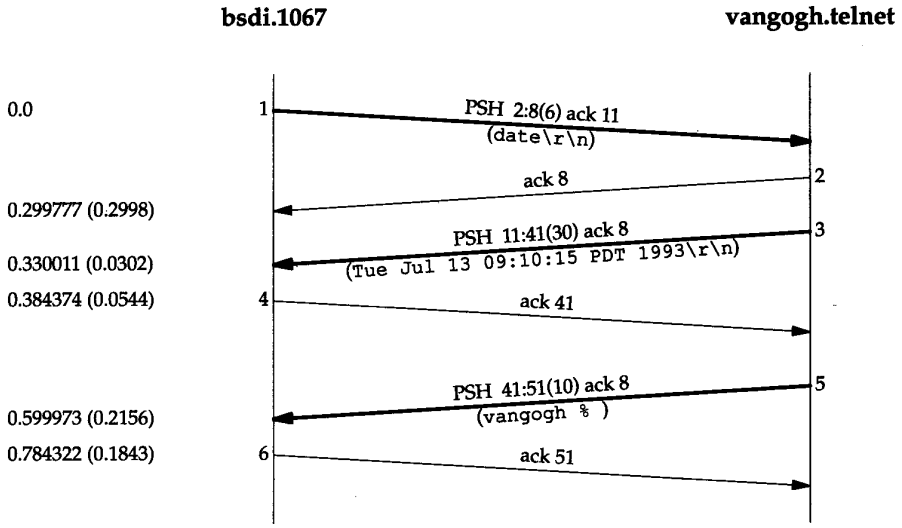


图26-14 Telnet行方式下客户进程向服务器进程发送命令的情况

把它和在Rlogin中输入同样命令（图 19-2）时的情况进行一下比较。我们看到在 Telnet行方式下只需要 2个报文段（一个包含数据，另一个用于 ACK，连同IP和TCP首部共86字节），而在Rlogin中要发送15个报文段（5个有键入的数据， 5个有回显的数据， 5个是ACK，共611字节）。可见节省的数据量是非常可观的。

如果在服务器端运行一个需要进入单个字符方式的应用程序（例如 vi编辑器）会怎么样呢？实际上将发生如下的一些交互：

1) 当服务器的应用程序启动了，并改变其伪终端方式时， Telnet服务器进程被通告需要进入单个字符方式。然后服务器发送 WILL ECHO命令和行方式子选项，以告知客户不要再以行方式工作，转而进入单个字符方式。

2) 客户响应以DO ECHO，并确认行方式子选项。

3) 应用程序在服务器上运行。我们键入的每个字符将发送到服务器（当然要强制使用Nagle算法），此时服务器将处理必要的回显工作。

4) 当应用程序终止时，就恢复其伪终端方式，并通告 Telnet服务器。服务器将向客户发送WONT ECHO命令，同时发送行方式子选项，告诉客户恢复进入行方式。

5) 客户响应DONT ECHO，确认进入行方式。

上述情况同我们键入口令之间的区别表明：回显功能和单个字符方式与一次一行方式没有依赖关系。当我们键入口令时，回显功能必须失效，但一次一行方式有效。对于一个全屏应用来讲，例如编辑器，回显必须失效而单个字符方式必须有效。

图26-15概括了Rlogin和Telnet不同方式之间的差异。

Linux公社 www.linuxidc.com

应用程序	客户进程发送		客户进程回显？	例　子
	一次一字符	一次一行		
Rlogin	•		否	
Telnet	•		否	
Telnet，行方式		•	是	正常命令
Telnet，行方式		•	否	键入我们的口令
Telnet，行方式	•		否	vi编辑器

图26-15　Rlogin和不同方式的Telnet之间的比较

26.5.3　一次一行方式（准行方式）

从图26-11可以看出，如果客户不支持行方式，那么较新的服务器支持行方式选项，它也将转入准行方式(Kludge line mode)。我们同时指出所有的客户进程和服务器进程都支持准行方式，但它不是默认方式，必须由客户进程或用户特地激活它。让我们来看看如何用 Telnet选项激活准行方式。

首先介绍当客户进程不支持行方式时， BSD/386服务器进程如何协商进入该方式。

1) 当客户进程不同意服务器进程激活行方式的请求时，服务器进程发送 DO TIMING MARK选项。RFC 860 [Postel 和Reynolds 1983f]定义了这个Telnet选项。它的作用是让收发双方同步，关于这个问题将在本节的后面讲到用户键入中断键时讨论。该选项只是用来判断客户进程是否支持准行方式。

2) 客户响应WILL TIMING MARK，表明支持准行方式。

3) 服务器发送WONT SUPPRESS GO AHEAD和WONT ECHO选项，告诉客户它希望禁止这两个选项。我们在前面已经强调：单个字符方式下是假定 SUPPRESS GO AHEAD和ECHO选项同时有效的，所以禁止两个选项就进入了准行方式。

4) 客户响应DONT SUPPRESS GO AHEAD和DONT ECHO命令。

5) 服务器发送`login:`提示符，然后用户键入用户名。用户名是以整行的方式发送给服务器，回显由客户进程在本地处理。

6) 服务器发送`Password:`提示符和WILL ECHO命令。这将使客户进程的回显失效，因为此时客户进程认为服务器进程将处理回显工作，所以用户键入的口令就不回显到屏幕上。客户响应DO ECHO命令。

7) 我们键入口令。客户以整行方式发送到服务器。

8) 服务器发送WONT ECHO命令，使得客户重新激活回显功能，客户响应 DONT ECHO。

从此以后的普通命令处理过程就和行方式相似了。客户进程负责所有的编辑和回显，并以整行的方式发送给服务器进程。

在图26-11中，我们已经强调：所有标注为“ char ”的记录都支持准行方式，只不过默认是单个字符方式罢了。如果要客户进入行方式，我们就能很容易看到选项协商的过程：

```
svr4 %                                    客户是sun，服务器是svr 4
                                          键入Control]以和Telnet客户进程通信(无回显)
telnet> status                            检验现在是否在一次一字符方式下
Connected to svr4.tuc.noao.edu
Operating in character-at-a-time mode.
Escape character is '^]'.
```

Linux公社 www.linuxidc.com

```
telnet> toggle options                    注意选项协商过程
Will show option processing.

telnet> mode line                         切换到准行方式
SENT dont SUPPRESS GO AHEAD               客户发送这两个选项
SENT dont ECHO
RCVD wont SUPPRESS GO AHEAD               服务器把WONT做为上述两个选项协商的响应
RCVD wont ECHO
```

这将使Telnet会话进入准行方式，此时SUPPRESS GO AHEAD和ECHO选项都是失效的。

如果在服务器端运行如vi编辑器这样的应用程序，同样会有行方式下遇到的问题。当要运行这样的应用程序时，服务器进程必须告诉客户进程从准行方式切换到单字符方式。当应用程序结束时，必须告诉客户进程返回到准行方式。下面是这个过程需要用到的技术要点。

1) 当应用程序改变其伪终端方式并通知服务器进程时，服务器进程将进入单字符方式。服务器进程向客户进程发送WILL SUPPRESS GO AHEAD和WILL ECHO，这将使客户进程进入单字符方式。

2) 客户进程回送DO SUPPRESS GO AHEAD和WILL ECHO。

3) 应用程序开始在服务器端运行。

4) 当应用程序结束并改变其伪终端方式时，服务器进程发送 WONT SUPPRESS GO AHEAD和WONT ECHO命令，使得客户进程返回准行方式。

5) 客户进程回送DONT SUPPRESS GO AHEAD和DONT ECHO命令，告诉服务器进程它已经回到了准行方式。

图26-16概括了单个字符方式及准行方式中不同的SUPPRESS GO AHEAD和ECHO选项设置。

方　式	SUPPRESS GO AHEAD	ECHO	举　例
一次一字符	√	√	准行方式下的vi编辑器
准行方式	×	×	正常命令
准行方式	×	√	键入我们的口令

图26-16 准行方式下Telnet选项的设置

26.5.4 行方式：客户中断键

看一下当用户键入中断键时 Telnet将发生什么情况。假定在客户主机 bsdi和服务器cangogh.cs.berkeley.edu之间建立了一个Telnet会话。图26-17显示了当用户键入中断键后的时间系列（去掉了窗口通告和服务类型）。

报文段1中显示的是中断键（通常是Control_C或DELETE）已经转换为Telnet的IP（中断进程）命令：<IAC, IP>。下面的3个字节：<IAC, DO, TM>，组成了Telnet的DO TIMING MARK选项。这个标志由客户进程发送，必须使用WILL或WONT响应。所有在响应前收到的数据都要丢弃（除非是Telnet命令）。这是服务器进程和客户机端的同步过程。报文段1没有采用TCP紧急方式。

> Host Requirements RFC叙述了IP命令不能使用Telnet的同步信号来发送。如果可以的话，那么<IAC, IP>的后面将跟随<IAC, DM>，同时紧急指针指向DM字节。大多数的Unix Telnet 客户有一个选项来使用同步信号发送IP命令，但是这个选项默认是不用的（正如我们这里看到的）。

Linux公社 www.linuxidc.com

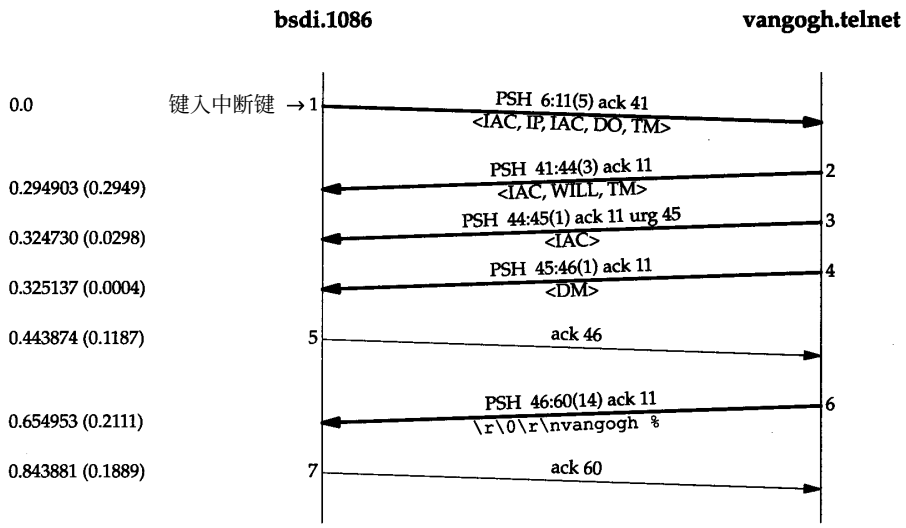


图26-17 行方式下键入中断键后的情况

报文段2是服务器进程对DO TIMING MARK选项的响应。紧随其后的是报文段3和4中Telnet的同步信号：<IAC, DM>。报文段3中的紧急指针指向将在报文段4中发送的DM字节。

如果服务器进程到客户进程的窗口已满，那么客户进程发送了如报文段1中的IP命令后就丢弃收到的所有数据。即使服务器进程被TCP流量控制所终止而不能发送如报文段2、3和4中的数据，紧急指针仍然可以发送。这和图26-7中的Rlogin类似。

为什么同步信号要分为两个数据段发送（3和4）？原因就是我们在20.8节中详细讨论TCP紧急指针时提到的情况。有关主机需求的RFC中提到紧急指针应指向紧急数据的最后一个字节，而很多衍生于伯克利的系统中，紧急指针指向紧急数据的倒数第2个字节（回忆一下在图26-6中，紧急指针指向命令字节的前一个字节）。Telnet服务器进程故意把同步信号的第1个字节作为紧急数据，它知道紧急指针将指向下1个字节（即DM字节），而IAC字节和紧急指针必须立即发送，在下一步才发送DM字节。

最后一个报文段6发送的是数据，它是服务器进程发生的提示符。

26.6 小结

本章我们介绍了Rlogin和Telnet操作。两者都提供了从客户进程远程登录到服务器进程，是我们能够在服务器端运行程序的方法。

这两个应用是不同的。Rlogin假定连接的双方都是Unix系统，所以只提供一个选项，它是1个简单的协议。Telnet则不同，它用于在不同类型的主机之间建立连接。

为了支持这种多机环境，Telnet提供客户进程和服务器进程的选项协商机制。如果连接的双方都支持这些选项，则可以增强一些功能。对于比较简单的客户进程和服务器进程，它可以提供Telnet的基本功能，而当双方都支持某些选项时，它又可以充分利用双方的新特性。

我们介绍了Telnet的选项协商机制，也介绍了3种数据传输的方式：单字符方式、准行方式和实行方式。现在的趋势是只要有可能，就尽量工作在准行方式下。这样可以减少网络上的数据量，同时为交互用户提供更好的行编辑和回显的响应。

图26-18概括并比较了Rlogin和Telnet的不同特性。

Linux公社 www.linuxidc.com

特　征	Rlogin	Telnet
运输协议	一个TCP连接。使用紧急方式	一个TCP连接。使用紧急方式。
分组方式	总是一次一字符，远程回显	通常的默认是一次一字符，远程回显。带客户回显的准行方式也支持带回显的实行模式。当服务器上的应用进程请求时，总是一次一字符的方式
流量控制	通常由客户完成，可以被服务器禁止	通常由服务器完成，选项允许客户来完成
终端类型	总是提供	选项，通常被支持
终端速率	总是提供	选项
窗口大小	大多数服务器支持此选项	选项
环境变量	不支持	选项
自动登录	默认。提示用户键入口令，口令以明文发送。较新的版本支持Kerberos方式的登录	默认是键入登录名和口令。口令以明文发送。较新的版本支持鉴别选项

图26-18　Rlogin和Telnet的不同特性

Rlogin服务器和Telnet服务器通常都将设置TCP的保活选项以检测客户主机是否崩溃（如果服务器的TCP实现支持，见第23章）。这两种应用都采用了TCP紧急方式，以便即使从服务器到客户的数据传输被流量控制所终止，服务器仍然可以向客户发送命令。

习题

26.1　在图26-5中，标出所有延迟的ACK。

26.2　在图26-7中，为什么要发送报文段12？

26.3　我们说过Rlogin客户进程必须使用保留端口号（见 1.9节）（通常Rlogin客户使用512~1023之间的保留端口）。这会给主机带来什么限制？有没有解决的办法？

26.4　阅读RFC 1097，它描述了Telnet的阈下报文(subliminal-message)选项。

Linux公社 www.linuxidc.com

第27章　FTP：文件传送协议

27.1　引言

FTP是另一个常见的应用程序。它是用于文件传输的 Internet标准。我们必须分清文件传送（file transfer）和文件存取 (file access) 之间的区别，前者是 FTP提供的，后者是如 NFS（Sun的网络文件系统，第 29章）等应用系统提供的。由 FTP提供的文件传送是将一个完整的文件从一个系统复制到另一个系统中。要使用 FTP，就需要有登录服务器的注册帐号，或者通过允许匿名FTP的服务器来使用（本章我们将给出这样的一个例子）。

与Telnet类似，FTP最早的设计是用于两台不同的主机，这两个主机可能运行在不同的操作系统下、使用不同的文件结构、并可能使用不同字符集。但不同的是，Telnet获得异构性是强制两端都采用同一个标准：使用7比特ASCII码的NVT。而FTP是采用另一种方法来处理不同系统间的差异。FTP支持有限数量的文件类型（ASCII，二进制，等等）和文件结构（面向字节流或记录）。

参考文献959 [Postel 和 Reynolds 1985] 是FTP的正式规范。该文献叙述了近年来文件传输的历史演变。

27.2　FTP协议

FTP与我们已描述的另一种应用不同，它采用两个 TCP连接来传输一个文件。

1) 控制连接以通常的客户服务器方式建立。服务器以被动方式打开众所周知的用于 FTP的端口（21），等待客户的连接。客户则以主动方式打开 TCP端口21，来建立连接。控制连接始终等待客户与服务器之间的通信。该连接将命令从客户传给服务器，并传回服务器的应答。

   由于命令通常是由用户键入的，所以IP对控制连接的服务类型就是“最大限度地减小迟延”。

2) 每当一个文件在客户与服务器之间传输时，就创建一个数据连接。（其他时间也可以创建，后面我们将说到）。

   由于该连接用于传输目的，所以IP对数据连接的服务特点就是“最大限度提高吞吐量”。

图27-1描述了客户与服务器以及它们之间的连接情况

从图中可以看出，交互式用户通常不处理在控制连接中转换的命令和应答。这些细节均由两个协议解释器来完成。标有“用户接口”的方框功能是按用户所需提供各种交互界面（全屏幕菜单选择，逐行输入命令，等等），并把它们转换成在控制连接上发送的 FTP命令。类似地，从控制连接上传回的服务器应答也被转换成用户所需的交互格式。

从图中还可以看出，正是这两个协议解释器根据需要激活文件传送功能。

27.2.1　数据表示

FTP协议规范提供了控制文件传送与存储的多种选择。在以下四个方面中每一个方面都必须作出一个选择。

Linux公社 www.linuxidc.com

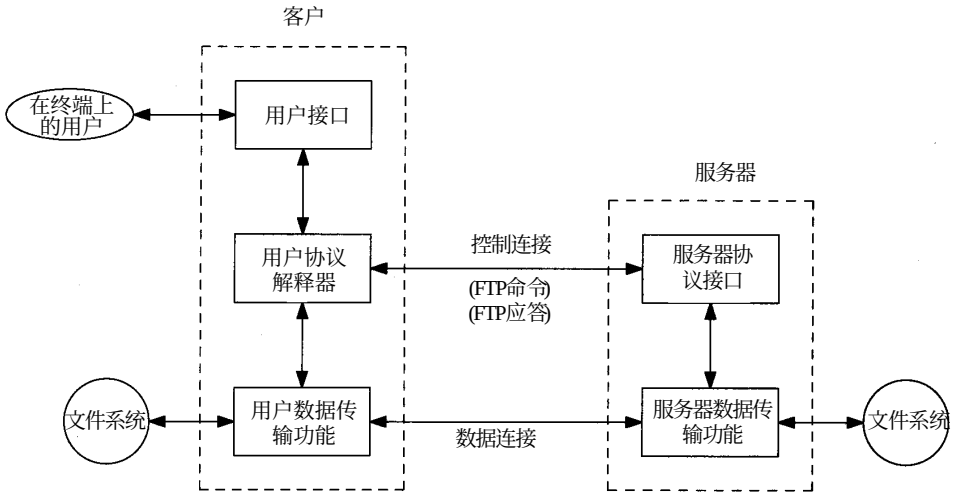


图27-1 文件传输中的处理过程

1. 文件类型

(a) ASCII码文件类型 （默认选择）文本文件以NVT ASCII码形式在数据连接中传输。这要求发方将本地文本文件转换成NVT ASCII码形式，而收方则将NVT ASCII码再还原成本地文本文件。其中，用NVT ASCII码传输的每行都带有一个回车，而后是一个换行。这意味着收方必须扫描每个字节，查找CR、LF对（我们在第15.2节见过的关于TFIP的ASCII码文件传输情况与此相同）。

(b) EBCDIC文件类型 该文本文件传输方式要求两端都是EBCDIC系统。

(c) 图像文件类型（也称为二进制文件类型） 数据发送呈现为一个连续的比特流。通常用于传输二进制文件。

(d) 本地文件类型 该方式在具有不同字节大小的主机间传输二进制文件。每一字节的比特数由发方规定。对使用8 bit字节的系统来说，本地文件以8 bit字节传输就等同于图像文件传输。

2. 格式控制

该选项只对ASCII和EBCDIC文件类型有效。

(a) 非打印 （默认选择）文件中不含有垂直格式信息。

(b) 远程登录格式控制 文件含有向打印机解释的远程登录垂直格式控制。

(c) Fortran 回车控制 每行首字符是Fortran格式控制符。

3. 结构

（a） 文件结构 （默认选择）文件被认为是一个连续的字节流。不存在内部的文件结构。

（b） 记录结构 该结构只用于文本文件（ASCII或EBCDIC）。

（c） 页结构 每页都带有页号发送，以便收方能随机地存储各页。该结构由 TOPS-20操作系统提供（主机需求RFC不提倡采用该结构）。

4. 传输方式

它规定文件在数据连接中如何传输。

（a） 流方式 （默认选择）文件以字节流的形式传输。对于文件结构，发方在文件尾提示关闭数据连接。对于记录结构，有专用的两字节序列码标志记录结束和文件结束。

（b） 块方式 文件以一系列块来传输，每块前面都带有一个或多个首部字节。

（c） 压缩方式 一个简单的全长编码压缩方法，压缩连续出现的相同字节。在文本文件

Linux公社 www.linuxidc.com

中常用来压缩空白串，在二进制文件中常用来压缩 0字节（这种方式很少使用，也不受支持。现在有一些更好的文件压缩方法来支持 FTP）。

如果算一下所有这些选择的排列组合数，那么对传输和存储一个文件来说就有 72种不同的方式。幸运的是，其中很多选择不是废弃了，就是不为多数实现环境所支持，所以我们可以忽略掉它们。

通常由Unix实现的FTP 客户和服务器把我们的选择限制如下：

- 类型：ASCII或图像。
- 格式控制：只允许非打印。
- 结构：只允许文件结构。
- 传输方式：只允许流方式。

这就限制我们只能取一、两种方式：ASCII或图像（二进制）。

该实现满足主机需求 RFC的最小需求（该 RFC也要求能支持记录结构，但只有操作系统支持它才行，而Unix不行）。

很多非Unix的实现提供了处理它们自己文件格式的 FTP功能。主机需求RFC指出“ FTP协议有很多特征，虽然其中一些通常不实现，但对 FTP中的每一个特征来说，都存在着至少一种实现 ”。

27.2.2 FTP命令

命令和应答在客户和服务器的控制连接上以 NVT ASCII码形式传送。这就要求在每行结尾都要返回CR、LF对（也就是每个命令或每个应答）。

从客户发向服务器的 Telnet命令（以IAC打头）只有中断进程（<IAC,IP>）和Telnet的同步信号（紧急方式下 <IAC,DM>）。我们将看到这两条 Telnet命令被用来中止正在进行的文件传输，或在传输过程中查询服务器。另外，如果服务器接受了客户端的一个带选项的 Telnet命令（WILL，WONT，DO或DONT），它将以DONT 或WONT响应。

这些命令都是3或4个字节的大写ASCII字符，其中一些带选项参数。从客户向服务器发送的FTP命令超过30种。图27-2给出了一些常用命令，其中大部分将在本章再次遇到。

命　令	说　明
ABOR	放弃先前的FTP命令和数据传输
LIST *filelist*	列表显示文件或目录
PASS *password*	服务器上的口令
PORT *n*1,*n*2,*n*3,*n*4,*n*5,*n*6	客户端IP地址（*n*1.*n*2.*n*3.*n*4）和端口（*n*5×256+*n*6）
QUIT	从服务器注销
RETR *filename*	检索（取）一个文件
STOR *filename*	存储（放）一个文件
SYST	服务器返回系统类型
TYPE *type*	说明文件类型：A表示ASCII码，I表示图像
USER *username*	服务器上用户名

图27-2　常用的FTP命令

下节我们将通过一些例子看到，在用户交互类型和控制连接上传送的 FTP命令之间有时是一对一的。但也有些操作下，一个用户命令产生控制连接上多个 FTP命令。

Linux公社 www.linuxidc.com

27.2.3 FTP应答

应答都是ASCII码形式的3位数字，并跟有报文选项。其原因是软件系统需要根据数字代码来决定如何应答，而选项串是面向人工处理的。由于客户通常都要输出数字应答和报文串，一个可交互的用户可以通过阅读报文串（而不必记忆所有数字回答代码的含义）来确定应答的含义。

应答3位码中每一位数字都有不同的含义（我们将在第 28章看到简单邮件传送输协议，SMTP，使用相同的命令和应答约定）。

图27-3给出了应答代码第1位和第2位的含义。

应答	说　　明
1yz	肯定预备应答。它仅仅是在发送另一个命令前期待另一个应答时启动
2yz	肯定完成应答。一个新命令可以发送
3yz	肯定中介应答。该命令已被接受，但另一个命令必须被发送
4yz	暂态否定完成应答。请求的动作没有发生，但差错状态是暂时的，所以命令可以过后再发
5yz	永久性否定完成应答。命令不被接受，并且不再重试
x0z	语法错误
x1z	信息
x2z	连接。应答指控制或数据连接
x3z	鉴别和记帐。应答用于注册或记帐命令
x4z	未指明
x5z	文件系统状态

图27-3 应答代码3位数字中第1位和第2位的含义

第3位数字给出差错报文的附加含义。例如，这里是一些典型的应答，都带有一个可能的报文串。

- 125 数据连接已经打开；传输开始。
- 200 就绪命令。
- 214 帮助报文（面向用户）。
- 331 用户名就绪，要求输入口令。
- 425 不能打开数据连接。
- 452 错写文件。
- 500 语法错误（未认可的命令）。
- 501 语法错误（无效参数）。
- 502 未实现的MODE(方式命令)类型。

通常每个FTP命令都产生一行回答。例如， QUIT命令可以产生如下应答：

```
221 Goodbye.
```

如果需要产生一条多行应答，第1行在3位数字应答代码之后包含一个连字号，而不是空格，最后一行包含相同的3位数字应答代码，后跟一个空格符。例如，HELP命令可以产生如下应答：

```
214- The following commands are recognized (* =>'s unimplemented).
   USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP
   PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP
   ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU
   SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE
```

Linux公社 www.linuxidc.com

```
   REIN*   MODE    MSND*   REST    XCWD    HELP    PWD     MDTM
   QUIT    RETR    MSOM*   RNFR    LIST    NOOP    XPWD
214 Direct comments to ftp-bugs@bsdi.tuc.noao.edu.
```

27.2.4 连接管理

数据连接有以下三大用途：

1) 从客户向服务器发送一个文件。

2) 从服务器向客户发送一个文件。

3) 从服务器向客户发送文件或目录列表。

FTP服务器把文件列表从数据连接上发回，而不是控制连接上的多行应答。这就避免了行的有限性对目录大小的限制，而且更易于客户将目录列表以文件形式保存，而不是把列表显示在终端上。

我们已说过，控制连接一直保持到客户-服务器连接的全过程，但数据连接可以根据需要随时来，随时走。那么需要怎样为数据连接选端口号，以及谁来负责主动打开和被动打开？

首先，我们前面说过通用传输方式（Unix环境下唯一的传输方式）是流方式，并且文件结尾是以关闭数据连接为标志。这意味着对每一个文件传输或目录列表来说都要建立一个全新的数据连接。其一般过程如下：

1) 正由于是客户发出命令要求建立数据连接，所以数据连接是在客户的控制下建立的。

2) 客户通常在客户端主机上为所在数据连接端选择一个临时端口号。客户从该端口发布一个被动的打开。

3) 客户使用PORT命令从控制连接上把端口号发向服务器。

4) 服务器在控制连接上接收端口号，并向客户端主机上的端口发布一个主动的打开。服务器的数据连接端一直使用端口20。

图27-4给出了第3步执行时的连接状态。假设客户用于控制连接的临时端口是 1173，客户用于数据连接的临时端口是1174。客户发出的命令是PORT命令，其参数是6个ASCII中的十进制数字，它们之间由逗点隔开。前面4个数字指明客户上的IP地址，服务器将向它发出主动打开（本例中是140.252.13.34），而后两位指明16 bit端口地址。由于16 bit端口地址是从这两个数字中得来，所以其值在本例中就是4×256+150 = 1174。

图27-5给出了服务器向客户所在数据连接端发布主动打开时的连接状态。服务器的端点是端口20。

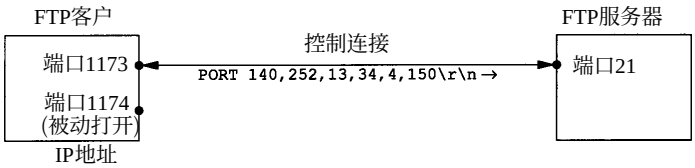


图27-4 在FTP控制连接上通过的PORT命令

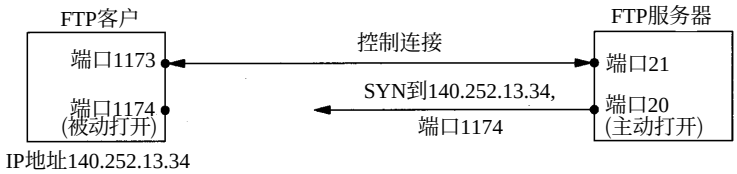


图27-5 主动打开数据连接的FTP服务器

Linux公社 www.linuxidc.com

服务器总是执行数据连接的主动打开。通常服务器也执行数据连接的主动关闭，除非当客户向服务器发送流形式的文件时，需要客户来关闭连接（它给服务器一个文件结束的通知）。

客户也有可能不发出PORT命令，而由服务器向正被客户使用的同一个端口号发出主动打开，来结束控制连接。这是可行的，因为服务器面向这两个连接的端口号是不同的：一个是20，另一个是21。不过，下节我们将看到为什么现有实现通常不这样做。

27.3 FTP的例子

现在看一些使用FTP的例子：它对数据连接的管理，采用NVT ASCII码的文本文件如何发送，FTP使用Telnet同步信号来中止进行中的文件传输，最后是常用的“匿名FTP”。

27.3.1 连接管理：临时数据端口

先看一下FTP的连接管理，它只在服务器上用简单FTP会话显示一个文件。我们用-d标志（debug）来运行svr4主机上的客户。这告诉它要打印控制连接上变换的命令和应答。所有前面冠以--->的行是从客户上发向服务器的，所有以3位数字开头的行都是服务器的应答。客户的交互提示是ftp>。

```
svr4 %ftp -d bsdi                                  -d 选项用作排错输出
Connected to bsdi.                                 客户执行控制连接的主动打开
220 bsdi FTP server (Version 5.60) ready           服务器响应就绪
Name (bsdi:rstevens):                              客户提示我们输入
---> USER rstevens                                 键入RETURN，客户发送默认信息
331 Password required for rstevens.
Password:                                          键入口令；它不需要回显
---> PASS XXXXXXXX                                     客户以明文发送它
230 User rstevens logged in.
ftp> dir hello.c                                   要求列出一个文件的目录
---> PORT 140,252,13,34,4,150                          见图27-4
200 PORT Command successful.
---> LIST hello.c
150 Opening ASCII mode data connection for /bin/ls.
-rw-r--r--   1 rstevens  staff  38 Jul 17 12:47 hello.c
226 Transfer complete.
remote: hello.c                                    客户输出
56 bytes received in 0.03 seconds (1.8 Kbytes/s)
ftp> quit                                          我们已完成
---> QUIT
221 Goodbye
```

当FTP客户提示我们注册姓名时，它打印了默认值（我们在客户上的注册名）。当我们敲RETURN键时，默认值被发送出去。

对一个文件列出目录的要求引发一个数据连接的建立和使用。本例体现了我们在图27-4和图27-5中给出的程序。客户要求TCP为其数据连接的终端提供一个临时端口号，并用PORT命令发送这个端口号（1174）给服务器。我们也看到一个交互用户命令（dir）成为两个FTP命令（PORT和LIST）。

Linux公社 www.linuxidc.com

图27-6是控制连接上分组交换的时间系列（已除去了控制连接的建立和结束，以及所有窗口大小的通知）。我们关注该图中数据连接在哪儿被打开、使用和过后的关闭。

图27-7是数据连接的时间系列。图中的起始时间与图 27-6中的相同。已除去了所有窗口大小通知，但留下服务类型字段，以说明数据连接使用另一个服务类型（最大吞吐量），而不同于控制连接（最小时延）（服务类型(TOS)值在图3-2中）。

在时间系列上，FTP服务器执行数据连接的主动打开，从端口 20（称为`ftp-data`）到来自PORT命令的端口号（1174）。本例中还可以看到服务器在哪儿向数据连接上执行写操作，服务器对数据连接执行主动的关闭，这就告诉客户列表已完成。

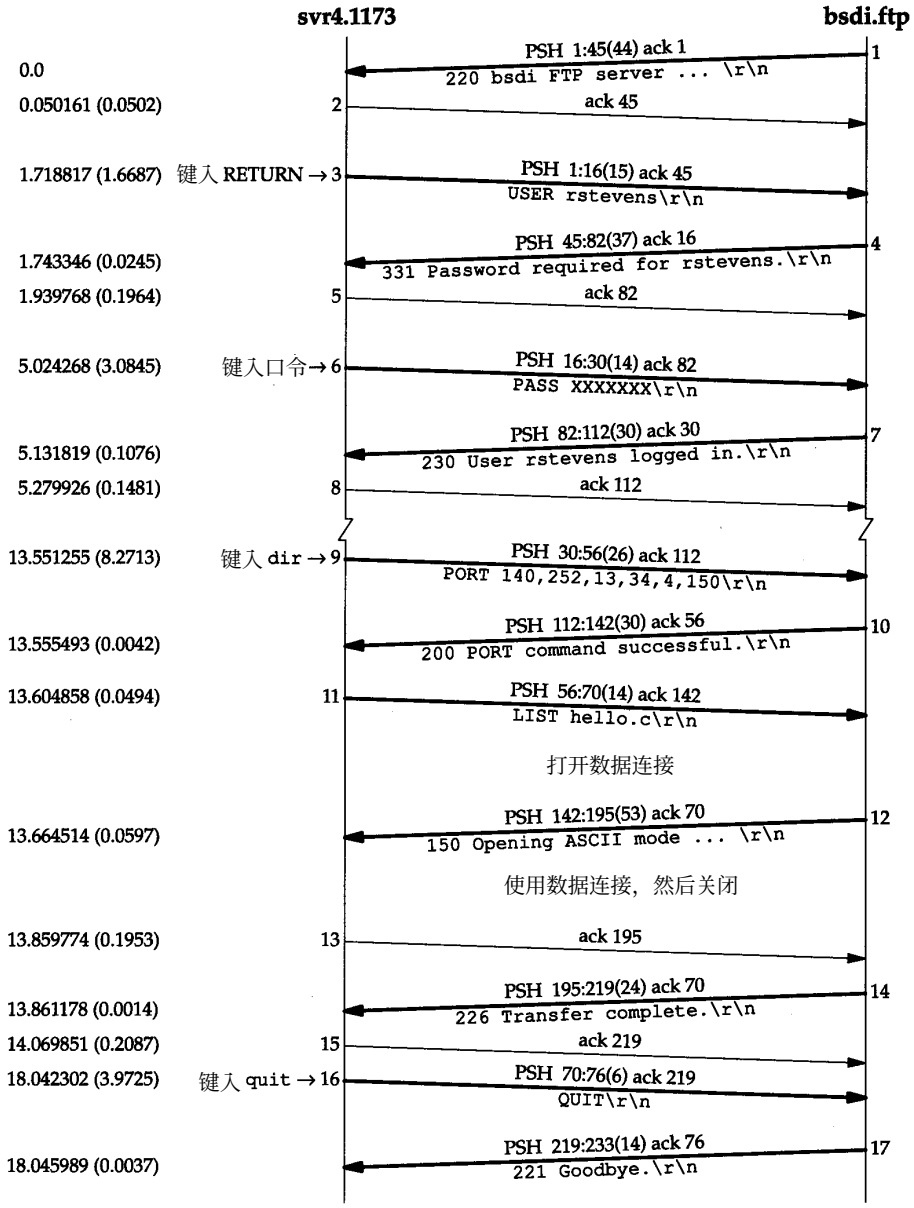


图27-6 FTP控制连接示例

Linux公社 www.linuxidc.com

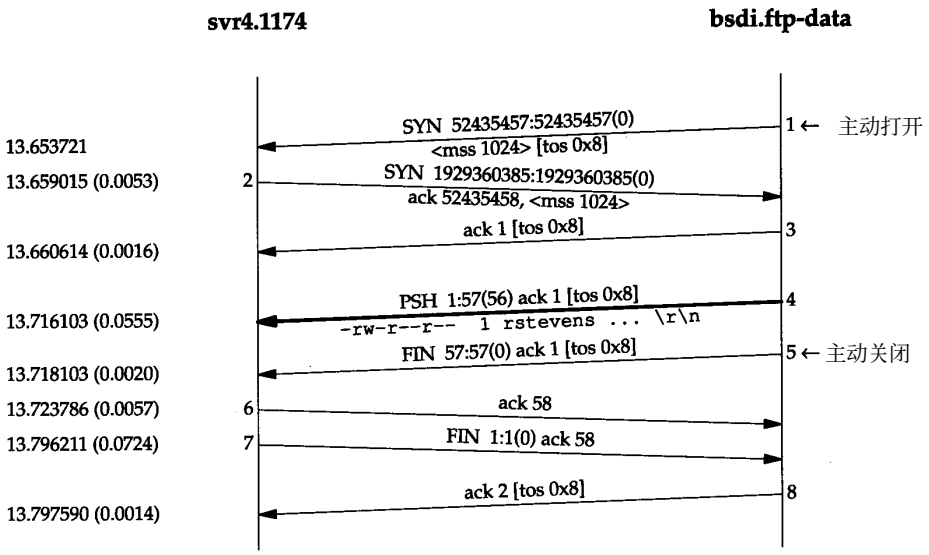


图27-7 FTP数据连接示例

27.3.2 连接管理：默认数据端口

如果客户没有向服务器发出PORT命令，来指明客户数据连接端的端口号，服务器就用与控制连接正在用的相同的端口号给数据连接。这会给使用流方式（Unix FTP客户和服务器一直使用）的客户带来一些问题。正如下面所示：

> Host Requirements RFC建议使用流方式的FTP客户在每次使用数据连接前发一个PORT命令来启用一个非默认的端口号。

回到先前的例子（图27-6），如果我们要求在列出第1个目录后几秒钟再列出另一个目录，那该怎么办？客户将要求其内核选择另一个临时端口号（可能是1175），下一个数据连接将建立在svr4端口1175和bsdi端口20之间。但在图27-7中服务器执行数据连接的主动打开，我们在18.6节说明了服务器将不把端口20分配给新的数据连接，这是因为本地端口号已被更早的连接使用，而且还处于2MSL等待状态。

服务器通过指明我们在18.6节中提到的SO_REUSEADDR选项，来解决这个问题。这让它把端口20分配给新连接，而新连接将从处于2MSL等待状态的端口（1174）处得到一个不一样的外部端口号（1175），这样一切都解决了。

如果客户不发送PORT命令，而在客户上指明一个临时端口号，那么情况将改变。我们可以通过执行用户命令sendport给FTP来使之发生。Unix FTP客户用这个命令在每个数据连接使用之前关闭向服务器发送PORT命令。

图27-8给出了用于两个连续LIST命令的数据连接时间系列。控制连接起自主机svr4上的端口1176，所以在没有PORT命令的情况下，客户和服务器给数据连接使用相同的端口号（除去了窗口通知和服务类型值）。

事件序列如下：

1) 控制连接是建立在客户端口1176到服务器端口21上的（这里我们不展示）。

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com

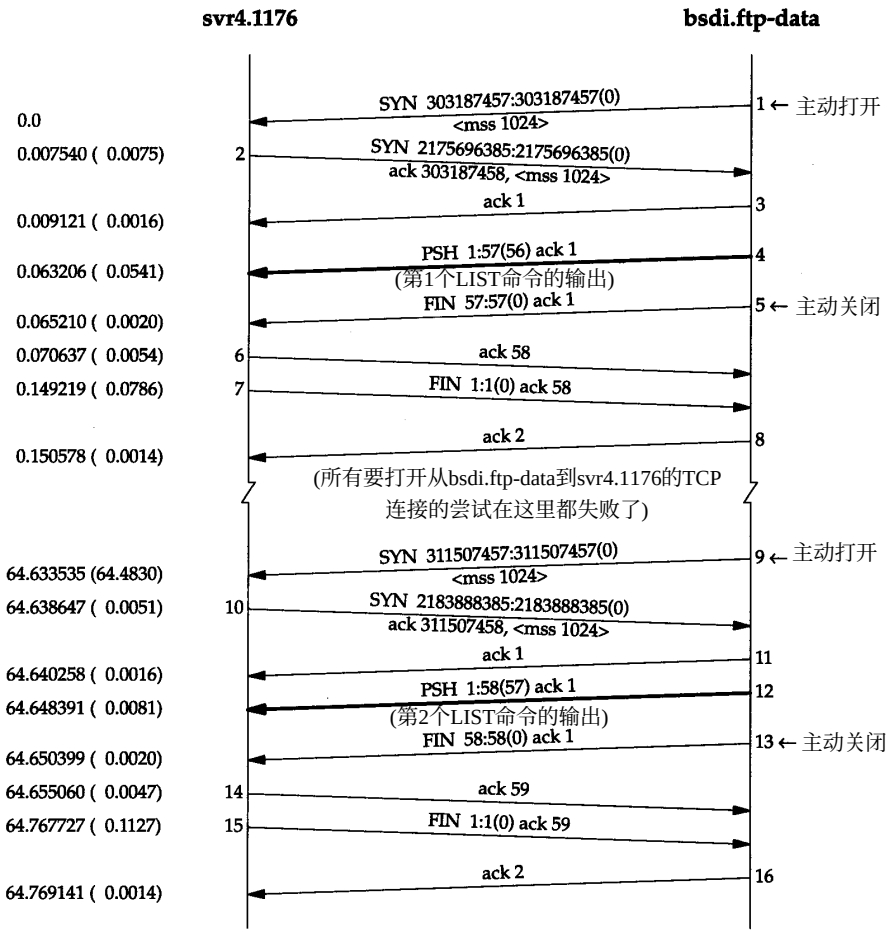


图27-8 两个连续LIST命令的数据连接

2) 当客户为端口1176上的数据连接做被动打开时，由于该端口已被客户上的控制连接使用，所以必须确定SO_REUSEADDR选项。

3) 服务器给端口20到端口1176的数据连接（报文段1）做主动打开。即便端口1176已在客户上被使用，客户仍会接受它（报文段2），这是因为下面这一对插口是不同的：

```
<svr4, 1176, bsdi, 21>
<svr4, 1176, bsdi, 20>
```

（在bsdi上的端口号是不同的）。TCP通过查看源IP地址、源端口号、目的IP地址、目的端口号分用各呼入报文段，只要这4个元素中的一个不同，就行。

4) 服务器对数据连接（报文段 5）做主动的关闭，即把这对插口置入服务器上的一个2MSL等待。

```
<svr4, 1176, bsdi, 20>
```

5) 客户在控制连接上发送另一个 LIST命令（这里我们不展示）。在此之前，客户在端口1176上为其数据连接端做一个被动打开。客户必须再一次指明 SO_REUSEADDR，这是因为端口号1176已在使用。

Linux公社 www.linuxidc.com

6) 服务器给从端口20到端口1176的数据连接发出一个主动打开。在此之前，服务器必须指明SO_REUSEADDR，这是因为本地端口（20）与处于2MSL等待状态的连接是相关联的，但从18.6节所示可知，该连接将不成功。其原由是这个连接用插口对 (socket pair)与步骤4中的仍处于2MSL等待状态的插口对相同。TCP规定禁止服务器发送同步信息（SYN）。这样就没办法让服务器跨过插口对的2MSL等待状态来重用相同的插口对。

在这一步伯克利软件分发（BSD）服务器每隔5秒就重试一次连接请求，直到满18次，总共90秒。我们看到报文段9将在大约1分钟后成功（我们在第18章提到过，SVR4使用一个30秒的MSL，以两个MSL来达到持续1分钟的等待）。我们没看到在这个时间系列上的这些失败有任何同步（SYN）信息，这是因为主动打开失败，服务器的TCP不再发送一个SYN。

Host Requirements RFC建议使用PORT命令的原因是在两个相继使用数据连接之间避免出现这个2MSL。通过不停地改变某一端的端口号，我们所说的这个问题就不会出现。

27.3.3 文本文件传输：NVT ASCII表示还是图像表示

让我们查证一下默认的文本文件传输使用 NVT ASCII码。这次不指定 -d标志，所以不看客户命令，但注意到客户还将打印服务器的响应：

```
sun % ftp bsdi
Connected to bsdi.
220 bsdi FTP server (Version 5.60) ready.
Name (bsdi:retevens);                                   键入RETURN
331 Password required for rstevens.
Passord :                                               键入口令
230 User rstevens logged in.
ftp> get hello.c                                        取一个文件
200 PORT command successful.
150 Opening ASCII mode data connection for hello.c (38 bytes).
226 Transfer complete.                                  服务器说明文件含有38字节
local: hello.c remote: hello.c                          由客户输出
42 bytes received in 0.0037 seconds (11 Kbytes/s)       字节传过数据连接
ftp> quit
221 Goodbye.
Sun % ls -l hello.c
-rw-rw-r--   1 rstevens     38 Jul 18  08:48 hello.c   但文件还含有38字节
sun % wc -l hello.c                                     在文件中记行数
    4 hello.c
```

因为文件有4行，所以从数据连接上传输42个字节。Unix下的每一新行符（\n）被服务器转换成NVT ASCII码的2字节行结尾序列（\r\n）来传输，然后再由客户转换成原先形式来存储。

新客户试图确定服务器是否是相同类型的系统，一旦相同，就可以用二进制码（图像文件类型）来传输文件，而不是ASCII码。这可以获得两个方面的好处：

1) 发方和收方不必查看每一字节（很大的节约）。

2) 如果主机操作系统使用比2字节的NVT ASCII码序列更少的字节来作行尾，就会传输更少的字节数（很小的节约）。

我们可以看到使用一个BSD/386客户和服务器的最优效果。启动排错（ debug）方式来看

Linux公社 www.linuxidc.com

客户FTP命令：

```
bsdi &ftp -d slip                                  指明 -d来看客户命令
Connected to slip.
220 slip FTP server (Version 5.60) ready.
Name (slip:rstevens):                              我们键入RETURN
---> USER rstevns
331 Password required for rstevens.
Password :                                         我们键入自己的口令
---> PASS XXXX
230 User rstevns logged in .
---> SYST                                          这由客户服务器的应答自动发送
215 UNIX Type: L8 Version : BSD-199103
Remote system type is UNIX.                        由客户发出的信息
Using binary mode to transfer files.               由客户发出的信息
ftp> get hello.c                                   取一个文件
---> TYPE I                                        由客户自动发送
200 Type set to I.
---> PORT  140,252,13,66,4,84                      端口号=4×256+84=1108
200 PORT command successful.
---> RETR hello.c
150 Opening BINARY mode data connection for hello.c (38 bytes) .
226 Transefer complete .
38 bytes received in 0.035 seconds (1.1 Kbytes/这时只有38个字节
ftp> quit
---> QUIT
221 Goodbye.
```

注册到服务器后，客户FTP自动发出SYST命令，服务器将用自己的系统类型来响应。如果应答起自字符串“215 UNIX Type : L8”，并且如果客户在每字节为8 bit的Unix系统上运行，那么二进制方式（图像）将被所有文件传输所使用，除非被用户改变。

当我们取文件hello.c时，客户自动发出命令TYPE I把文件类型定成图像。这样在数据连接上只有38字节被传输。

> Host Requirements RFC指出一个FTP服务器必须支持SYST命令（这曾是RFC 959中的一个选项）。但支持它的使用文本的系统（见封 2）仅仅是BSD/386和AIX 3.2.2。SunOS 4.1.3和Solaris 2.x 用500（不能理解的命令）来应答。SVR4采用极不大众化的应答行为500，并关闭控制连接！

27.3.4 异常中止一个文件的传输：Telnet 同步信号

现在看一下FTP客户是怎样异常中止一个来自服务器的文件传输。异常中止从客户传向服务器的文件很容易——只要客户停止在数据连接上发送数据，并发出 ABOR命令到控制连接上的服务器即可。而异常中止接收就复杂多了，这是因为客户要告知服务器立即停止发送数据。我们前面提到要使用Telnet同步信号，下面的例子就是这样。

我们先发起一个接收，并在它开始后键入中断键。这里是交互会话，其中初始注册被略去：

Linux公社 www.linuxidc.com

```
ftp> get a.out                                  取一个大文件
---> TYPE I                                     客户和服务器都是8 bit字节的Unix系统
200 Type set to I.
---> PORT 140,252,13,66,4,99
200 PORT command successful.
---> RETR a.out
150 Opening BINARY mode data connection for a.out (28672 bytes).
^?                                              键入的中断键
receive aborted                                 由客户输出
waiting for remote to finish abort              由客户输出
426 Transfer aborted. Data connection closed.
226 Abort successful
1536 bytes received in 1.7 seconds (0.89 Kbytes/s)
```

在我们键入中断键之后，客户立即告知我们它将发起异常中止，并正在等待服务器完成。服务器发出两个应答：426和226。这两个应答都是由Unix 服务器在收到来自客户的紧急数据和ABOR命令时发出的。

图27-9和图27-10展示了会话时间系列。我们已把控制连接（实线）和数据连接（虚线）合在一起来说明它们之间的关系。

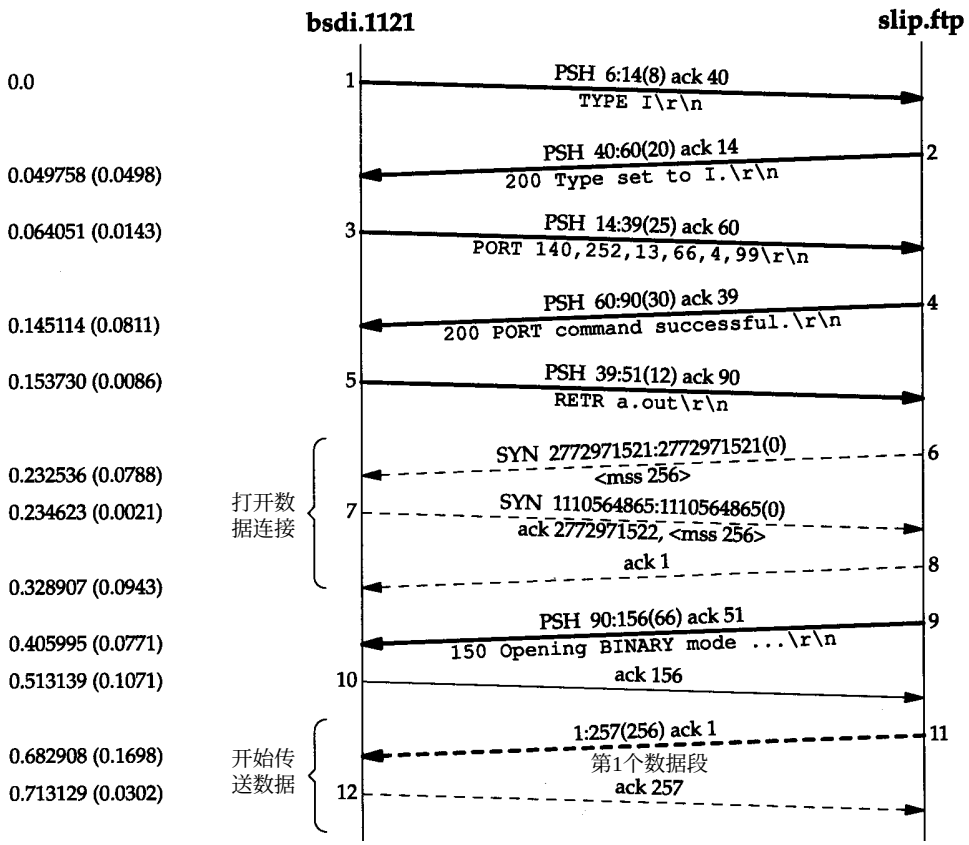


图27-9 异常中止一个文件的传输（前半部）

图27-9的前面12个报文段是我们所期望的。通过控制连接的命令和应答建立起文件传输，数据连接被打开，第1个报文段的数据从服务器发往客户。

Linux公社 www.linuxidc.com

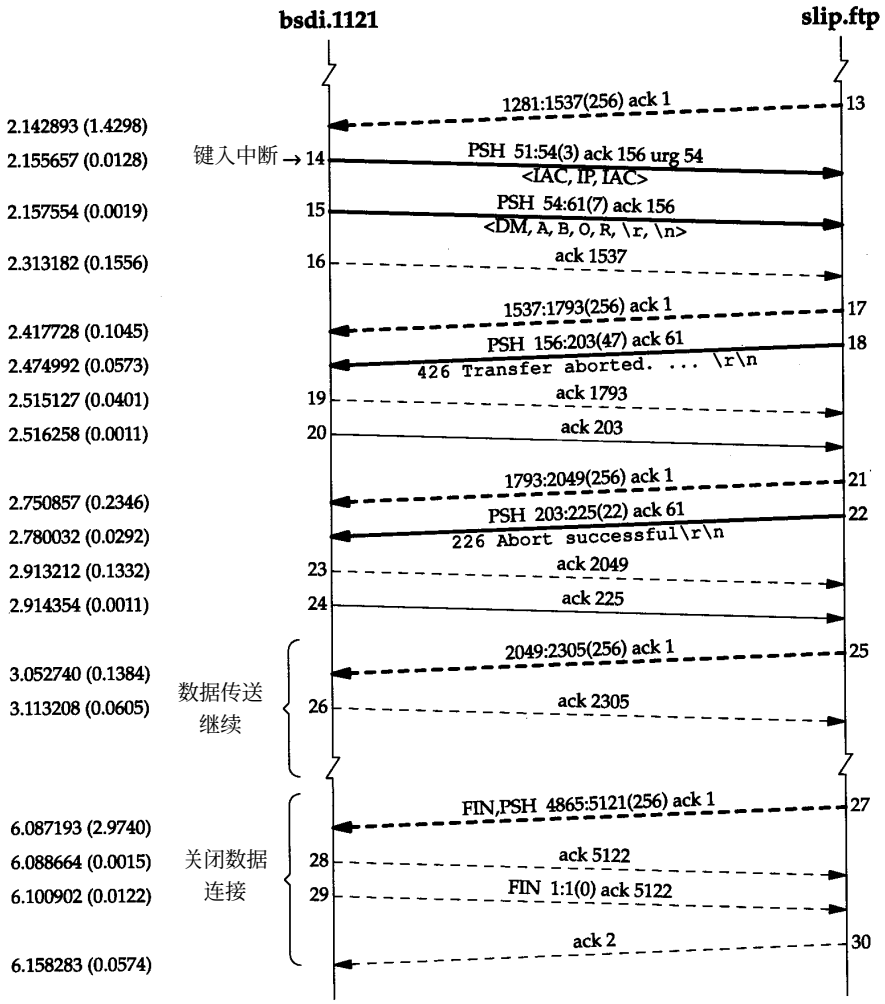


图27-10 异常中止一个文件的传输（后半部）

在图27-10中，报文段13是数据连接上来自服务器的第 6个数据报文段，后跟由我们键入的中断键产生的报文段14。客户发出10个字节来异常中止传输：

```
<IAC,IP,IAC,DM,A,B,O,R,\r,\n>
```

由于20.8节中详细讨论过这个问题，我们看到有两个报文段（ 14和15）涉及到TCP的紧急指针（我们在图26-17中看过对Telnet问题也做相同的处理）。Host Requirements RFC指出紧急指针应指向紧急数据的最后一个字节，而多数伯克利的派生实现使之指向紧急数据最后一个字节后面的第一个字节。了解到紧急指针将（错误地）指向下一个要写的字节（数据标志，DM。在序号为54处），FTP客户进程特意写前3个字节作为紧急数据。首先写下的 3字节紧急数据与紧急指针一起被立即发送，紧接着是后面 7个字节（BSD FTP 服务器不会出现由客户使用的紧急指针的解释问题。当服务器收到控制连接上的紧急数据时，它读下一个 FTP命令，寻找ABOR或STAT，忽略嵌入的Telnet命令）。

注意到尽管服务器指出传输被异常中止（报文段 18，在控制连接上），客户进程还要在数据连接上再接收14个报文段的数据（序列号是 1537~5120）。这些报文段可能在收到异常中止

Linux公社 www.linuxidc.com

时，还在服务器上的网络设备驱动器中排队，但客户打印“收到 1536字节”，意思是在发出异常中止后（报文段14和15），略去收到的所有数据报文段。

一旦Telnet用户键入中断键，我们在图 26-17中看到Unix客户在默认情况下不发出中断进程命令作为紧急数据。因为几乎没有机会用流控制来中止从客户进程到服务器进程的数据流，所以我们说这样就行了。FTP的客户进程也通过控制连接发送一个中断进程命令，因为两个连接正在被使用，因此没有机会用流控制来中止控制连接。为什么 FTP发送中断进程命令作为紧急数据而Telnet不呢？答案在于FTP使用两个连接，而Telnet只使用一个，在某些操作系统上要求一个进程同时监控两个连接的输入是困难的。FTP假设这些临界的操作系统至少提供紧急数据在控制连接上已到达的通知，而后让服务器从处理数据连接切换到控制连接上来。

27.3.5 匿名FTP

FTP的一种形式很常用，我们下面给出它的例子。它被称为匿名 FTP，当有服务器支持时，允许任何人注册并使用FTP来传输文件。使用这个技术可以提供大量的自由信息。

怎样找出你正在搜寻的站点是一个完全不同的问题。我们将在 30.4节简要介绍。

我们将把匿名FTP用在站点`ftp.uu.net`上（一个常用的匿名FTP站点）来取本书的勘误表文件。要使用匿名FTP，须使用“anonymous”（复习数遍就能正确地拼写）用户名来注册。当提示输入口令时，我们键入自己的电子邮箱地址。

```
sun % ftp ftp.uu.net
Connected to ftp.uu.net
220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready
Name (ftp.uu.net:rstevens): anonymous
331 Guest login ok, send your complete e-mail addraess as password.
Password :                                  键入rstevens@noao.edu；它没有回显
230-
230-                    Welcome to the UUNET archive.
230-    A service of UUNET Technologies Inc, Falls Church, Virginia
230-    For information about UUNET, call +1 703 204 8000, or see the files
230-    in /uunet-info
                                            还有一些问候行
230 Guest login ok, access restrictions apply.
ftp> cd published/books                     换成需要的目录
250 CWD command successful.
ftp> binary                                 我们将传送一个二进制文件
200 Type set to I.
ftp> get stevens.tcpipivl.errata.Z          取文件
200 PORT command successful.
150 Opening BINARY mode data connection for stevens.tcpipivl.errata.Z (150 bytes).
226 Transfer complete.                      (你可能得到一个不同的文件大小)
local: stevens.tcpipivl.errata.Z remote: stevens.tcpipivl.errata.Z
105 bytes received in 4.1 seconds (0.83 Kbytes/s)
ftp> quit
221 Goodbye.
sun % uncompress stevens.tcpipivl.errata.Z
sun % more stevens.tcpipivl.errata
```

Linux公社 www.linuxidc.com

不压缩是因为很多现行匿名FTP文件是用Unix compress(1)程序压缩的，这样导致文件带有.Z的扩展名。这些文件必须使用二进制文件类型来传输，而不是ASCII码文件类型。

27.3.6 来自一个未知IP地址的匿名FTP

可以把一些使用匿名FTP的域名系统（DNS）特征和选路特征结合在一起。在14.5节中我们谈到DNS中指针查询现象——取一个IP地址并返回其主机名。不幸的是并非所有系统管理员都能正确地创立涉及指针查询的名服务器。他们经常记得把新主机加入名字到地址匹配的文件中，却忘了把他们加入到地址到名字匹配的文件中。对此，可用traceroute经常看到这种现象，即它打印一个IP地址，而不是主机名。

有些匿名FTP服务器要求客户有一个有效域名。这就允许服务器来记录正在执行传输的主机域名。由于服务器在来自客户IP数据报中收到的关于客户的唯一标识是客户的IP地址，所以服务器能叫DNS来做指针查询，并获得客户的域名。如果负责客户主机的名服务器没有正确地创立，指针查询将失败。

要看清这个错误，我们来做以下诸步骤：

1) 把主机slip（见封2的图）的IP地址换成140.252.13.67。这是给作者子网的一个有效IP地址，但没有涉及到noao.edu域的域名服务器。

2) 把在bsdi上SLIP连接的目的IP地址换成140.252.13.67。

3) 把将数据报引向140.252.13.67的sun上的路由表入口加入路由器bsdi（回忆一下我们在9.2节中关于这个选路表的讨论）。

从Internet上仍然可以访问我们的主机slip，这是因为在10.4节中路由器gateway和netb正好把所有目的是子网140.252.13的所有数据报都发送给路由器sun。路由器sun知道利用我们在上述第3步建立的路由表入口来如何处理这些数据报。我们所创建的是拥有完整Internet连接性的主机，但没有有效的域名。结果，指针查询IP地址140.252.13.67将失败。

现在给一个我们所知的服务器使用匿名FTP，需要一个有效的域名：

```
slip % ftp ftp.uu.net
Connected to ftp.uu.net.

220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready.
Name (ftp.uu.net:rstevens): anonymous

530- Sorry, we're unable to map your IP address 140.252.13.67 to a hostname
530- in the DNS.  This is probably because your nameserver does not have a
530- PTR record for your address in its tables, or because your reverse
530- nameservers are not registered.  We refuse service to hosts whose
530- names we cannot resolve.  If this is simply because your nameserver is
530- hard to reach or slow to respond then try again in a minute or so, and
530- perhaps our nameserver will have your hostname in its cache by then.
530- If not, try reaching us from a host that is in the DNS or have your
530- system administrator fix your servers.
530 User anonymous access denied..

Login failed.
Remote system type is UNIX.
Using binary mode to transfer files.

ftp> quit
221 Goodbye.
```

来自服务器的出错应答是无需加以说明的。

Linux公社 www.linuxidc.com

27.4 小结

FTP是文件传输的Internet标准。与多数其他TCP应用不同，它在客户进程和服务器进程之间使用两个TCP连接——一个控制连接，它一直持续到客户进程与服务器进程之间的会话完成为止；另一个按需可以随时创建和撤消的数据连接。

FTP使用的关于数据连接的连接管理让我们更详细地了解 TCP连接管理需求。我们看到TCP在不发出PORT命令的客户进程上对2MSL等待状态的作用。

FTP使用NVT ASCII码做跨越控制连接的所有远程登录命令和应答。数据传输的默认方式通常也是NVT ASCII码。我们看到较新的Unix客户进程会自动发送命令来查看服务器是否是8 bit字节的Unix主机，并且如果是，那么就使用二进制方式来传输所有文件，那将带来更高的效率。

我们也展示了匿名FTP的一个例子，它是在Internet上分发软件的常用形式。

习题

27.1 图27-8中，如果客户对第2个数据连接做一次主动打开，而不是由服务器来做，那将发生什么变化？

27.2 在本章FTP客户例子中，我们加入诸如由客户输出行的行注释。如果不看源代码，我们如何确定这些不是来自服务器？

```
local: hello.c remote: hello.c
42 bytes received in 0.0037 seconds (11 Kbytes/s)
```

Linux公社 www.linuxidc.com

第28章　SMTP: 简单邮件传送协议

28.1　引言

电子邮件（e-mail）无疑是最流行的应用程序。[Caceres et al.1991]说明，所有TCP连接中大约一半是用于简单邮件传送协议 SMTP (Simple Mail Transfer Protocol)的（以比特计算为基础，FTP连接传送更多的数据）。[Paxson 1993] 发现，平均每个邮件中包含大约1500字节的数据，但有的邮件中包含兆比特的数据，因为有时电子邮件也用于发送文件。

图28-1显示了一个用TCP/IP交换电子邮件的示意图。

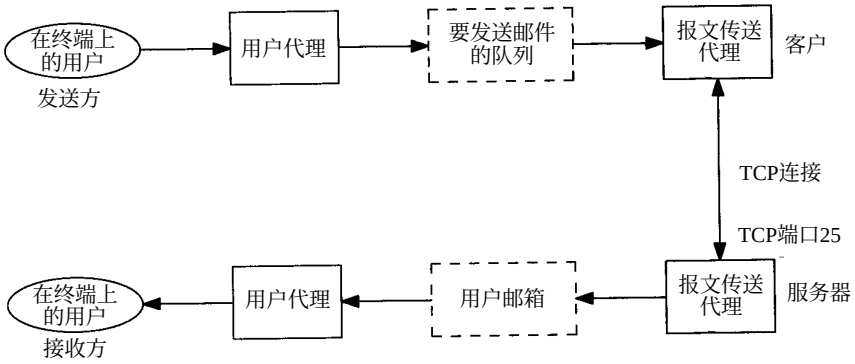


图28-1　Internet电子邮件示意图

用户与用户代理（user agent）打交道，可能会有多个用户代理可供选择。常用的 Unix上的用户代理包括MH，Berkeley Mail, Elm和Mush。

用TCP进行的邮件交换是由报文传送代理 MTA（Message Transfer Agent）完成的。最普通的Unix系统中的MTA是Sendmail。用户通常不和MTA打交道，由系统管理员负责设置本地的MTA。通常，用户可以选择它们自己的用户代理。

本章研究在两个MTA之间如何用TCP交换邮件。我们不考虑用户代理的运行或实现。

RFC 821 [Postel 1982] 规范了SMTP协议，指定了在一个简单TCP连接上，两个MTA如何进行通信。RFC 822 [Crocker 1982] 指定了在两个MTA之间用RFC 821 发送的电子邮件报文的格式。

28.2　SMTP协议

两个MTA之间用NVT ASCII进行通信。客户向服务器发出命令，服务器用数字应答码和可选的人可读字符串进行响应。这与上一章的 FTP类似。

客户只能向服务器发送很少的命令：不到 12个（相比较而言， FTP超过40个）。我们用简单的例子说明发送邮件的工作过程，并不仔细描述每个命令。

28.2.1　简单例子

我们将发送一个只有一行的简单邮件，并观察 SMTP连接。我们用 -v 标志调用用户代理，

Linux公社 www.linuxidc.com

它被传送给邮件传送代理（本例中是 Sendmail）。当设置该标志时，该MTA显示在SMTP连接上发送和接收的内容。以 >>>开始的行是SMTP客户发出的命令，以3位数字的应答码开始的行是从SMTP服务器来的。以下就是交互会话：

```
sun % mail -v rstevens@noao.edu                 调用我们的代理
To: rstevens@noao.edu                           这是用户代理的输出
Subject: testing                                然后指示我们键入主题
                                                用户代理在首部和正文之间加上一行空行
1, 2, 3.                                        这是我们键入的正文
.                                               我们在一行上输入一个句点，说明完成了
Sending letter ... rstevens@noao.edu...         用户代理上详细的输出

Connecting to mailhost via ether...             以下是MTA(Sendmail)的输出
Trying 140.252.1.54...  connected.
220 noao.edu Sendmail 4.1/SAG-Noao.G89 ready at Mon, 19 Jul 93 12:47:34 MST

>>> HELO sun.tuc.noao.edu.
250 noao.edu Hello sun.tuc.noao.edu., pleased to meet you

>>> MAIL From:<rstevens@sun.tuc.noao.edu>
250 <rstevens@sun.tuc.noao.edu>... Sender ok
>>> RCPT To:<rstevens@noao.edu>
250 <rstevens@noao.edu>... Recipient ok

>>> DATA
354 Enter mail, end with "." on a line by itself

>>> .
250 Mail accepted

>>> QUIT
221 noao.edu delivering mail

rstevens@noao.edu... Sent
sent.                                           这是用户代理的输出
```

只有5个SMTP命令用于发送邮件：HELO，MAIL，RCTP，DATA和QUIT。

我们键入`mail`启动用户代理，然后键入主题（subject）的提示；键入后，再键入报文的正文。在一行上键入一个句点结束报文，用户代理把邮件传给 MTA，由MTA进行交付。

客户主动打开TCP端口25。返回时，客户等待从服务器来的问候报文（应答代码为 220）。该服务器的应答必须以服务器的完全合格的域名开始：本例中为 `noao.edu`（通常，跟在数字应答后面的文字是可选的。这里需要域名。以`Sendmail`打头的文字是可选的）。

下一步客户用HELO命令标识自己。参数必须是完全合格的的客户主机名： `sun.tuc.noao.edu`。

MAIL命令标识出报文的发起人。下一个命令， RCPT，标识接收方。如果有多个接收方，可以发多个RCPT命令。

邮件报文的内容由客户通过DATA命令发送。报文的末尾由客户指定，是只有一个句点的一行。最后的命令QUIT，结束邮件的交换。

图28-2是在发送方SMTP（客户端）与接收方SMTP（服务器）之间的一个SMTP连接。

我们键入到用户代理的数据是一行报文（“1，2，3”），但在报文段12中共发送了393字节的数据。下面的12行组成了客户发送的393字节数据：

```
Received: by sun.tuc.noao.edu. (4.1/SMI-4.1)
        id AA00502; Mon, 19 Jul 93 12:47:32 MST
Message-Id: <9307191947.AA00502@sun.tuc.noao.edu.>
From: rstevens@sun.tuc.noao.edu (Richard Stevens)
Date: Mon, 19 Jul 1993 12:47:31 -0700
```

Linux公社 www.linuxidc.com

```
Reply-To: rstevens@noao.edu
X-Phone: +1 602 676 1676
X-Mailer: Mail User's Shell (7.2.5 10/14/92)
To: rstevens@noao.edu
Subject: testing

1, 2, 3.
```

前三行，Received:和Message-Id: 由MTA加上；下一行由用户代理生成。

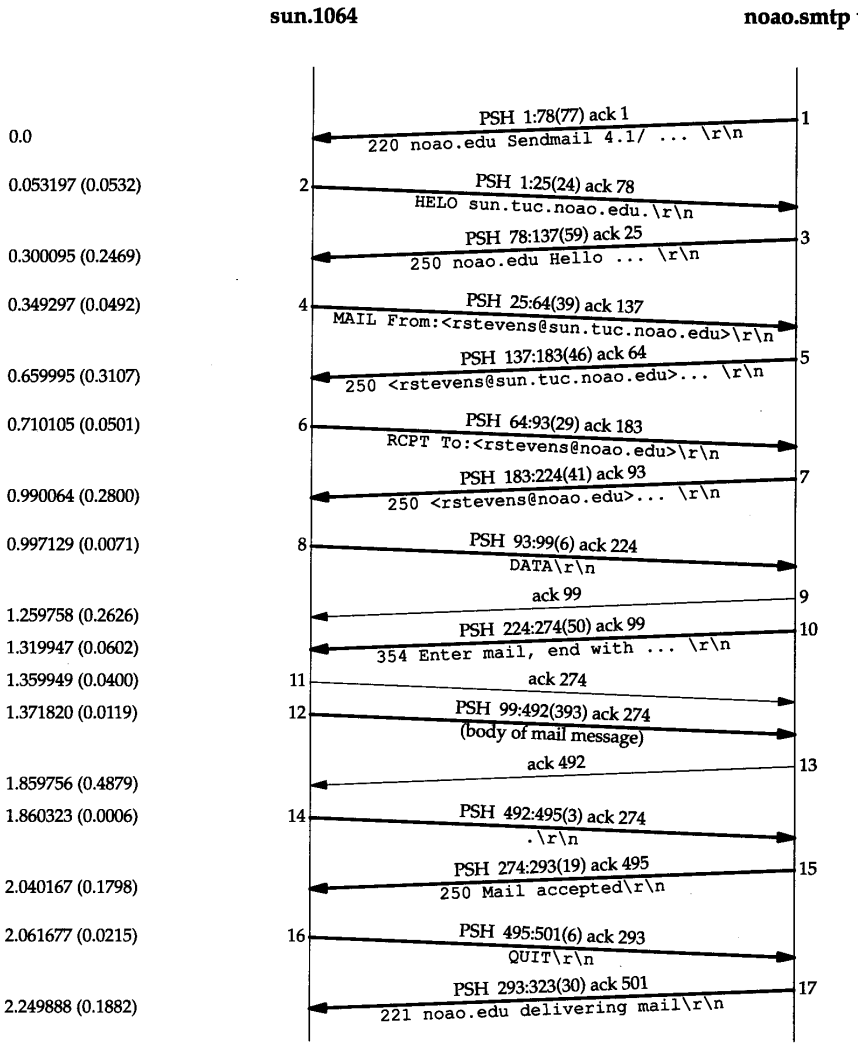


图28-2 基本SMTP邮件交付

28.2.2 SMTP命令

最小SMTP实现支持8种命令。我们在前面的例子中遇到 5个：HELO，MAIL，RCPT，DATA和QUIT。

RSET命令异常中止当前的邮件事务并使两端复位。丢掉所有有关发送方、接收方或邮件的存储信息。

Linux公社 www.linuxidc.com

VRFY命令使客户能够询问发送方以验证接收方地址，而无需向接收方发送邮件。通常是系统管理员在查找邮件交付差错时手工使用的。我们将在下一节中给出这方面的例子。

NOOP命令除了强迫服务器响应一个OK应答码（200）外，不做任何事情。

还有附加和可选命令。EXPN扩充邮件表，与VRFY类似，通常是由系统管理员使用的。事实上，许多Sendmail的版本都把这两者等价地处理。

> 4.4BSD 中的Sendmail版本8不再将两者等同处理。VRFY不扩充别名也不接受.forward文件。

TURN命令使客户和服务器交换角色，无需拆除TCP连接并建立新的连接就能以相反方向发送邮件（Sendmail不支持这个命令）。其他还有三个很少被实现的命令（SEND、SOML和SAML）取代MAIL命令。这三个命令允许邮件直接发送到客户终端（如果已注册）或发送到接收方的邮箱。

28.2.3 信封、首部和正文

电子邮件由三部分组成：

1) 信封（envelope）是MTA用来交付的。在我们的例子中信封由两个SMTP命令指明：

```
MAIL From: <rstevens@sun.tuc.noao.edu>
RCPT To: <estevens@noao.edu>
```

RFC 821指明了信封的内容及其解释，以及在一个TCP连接上用于交换邮件的协议。

2) 首部由用户代理使用。在我们的例子中可以看到9个首部字段：`Received`、`Message-Id`、`From`、`Data`、`Reply-To`、`X-Phone`、`X-Mailer`、`To`和`Subject`。每个首部字段都包含一个名，紧跟一个冒号，接着是字段值。RFC 822指明了首部字段的格式的解释（以X-开始的首部字段是用户定义的字段，其他是由RFC 822定义的）。长首部字段，如例子中的Received，被折在几行中，多余行以空格开头。

3) 正文（body）是发送用户发给接收用户报文的内容。RFC 822指定正文为NVT ASCII文字行。当用DATA命令发送时，先发送首部，紧跟一个空行，然后是正文。用DATA命令发送的各行都必须小于1000字节。

用户接收我们指定为正文的部分，加上一些首部字段，并把结果传到MTA。MTA加上一些首部字段，加上信封，并把结果发送到另一个MTA。

内容（content）通常用于描述首部和正文的结合。内容是客户用DATA命令发送的。

28.2.4 中继代理

在我们的例子中本地MTA的信息输出的第1行是："Connecting to mailhost via ether"（即"通过以太网连接到邮件主机"）。这是因为作者的系统已被配置成把所有非本地的向外的邮件发送到一台中继机上进行转发。

这样做的原因有两个。首先，简化了除中继系统MTA外的其他所有MTA的配置（所有曾使用过Sendmail的人都能证明，配置一个MTA并不简单）。第二，它允许某个机构中的一个系统作为邮件集线器，从而可能把其他所有系统隐藏起来。

在这个例子中，中继系统在本地域（.tuc.noao.edu）中有一个mailhost的主机名，而其他所有系统都被配置成把它们的邮件发往该主机。我们可以执行host命令来看看在DNS中这个名

Linux公社 www.linuxidc.com

是如何定义的：

```
sun %host mailhost
mailhost.tuc.noao.edu     CNAME     noao.edu          规范名
noao.edu                    A       140.252.1.54      它的真实IP地址
```

如果将来用于中继的主机改变了，只需改变它的 DNS名——其他所有单个系统的邮箱配置都无需改变。

目前许多机构都采用中继系统。图 28-3是修改后的Internet邮件图（图28-2），考虑发送主机和最后的接收主机都可能使用中继主机。

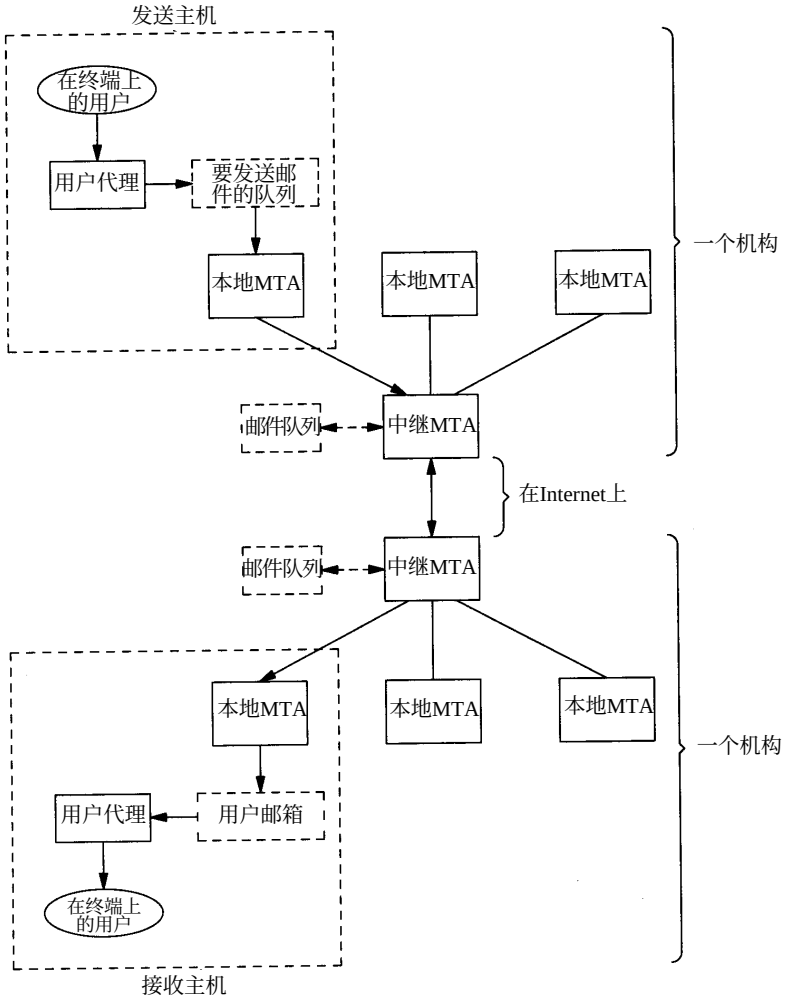


图28-3 在两端都有一个中继系统的Internet电子邮件

在这种情况下，在发送方和接收方之间有 4个MTA。发送方主机上的本地 MTA只把邮件交给它自己的中继 MTA（该中继MTA可能在该机构的域中有一个 mailhost的主机名）。这个通信就在该机构的本地互联网上用 SMTP。然后，发送方机构的中继 MTA就在Internet上把邮件发送到接收方机构的中继 MTA上，而这个中继 MTA就通过与接收方主机上的本地 MTA通信，把邮件交给接收方主机。尽管可能存在其他协议，但这个例子中所有 MTA均使用SMTP协议。

Linux公社 www.linuxidc.com

28.2.5 NVT ASCII

SMTP的一个特色是它用NVT ASCII表示一切：信封、首部和正文。正如我们在26.4节中谈到的，这是一个7 bit的字符码，以8 bit字节发送，高位比特被置为0。

在28.4节中，我们讨论了Internet邮件的一些新特性、允许发送和接收诸如音频和视频数据的扩充SMTP和多媒体邮件（MIME）。我们将看到，MIME和NVT ASCII一起表示信封、首部和正文，只需对用户代理作一些改变。

28.2.6 重试间隔

当用户把一个新的邮件报文传给它的MTA时，通常立即试图交付。如果交付失败，MTA必须把该报文放入队列中以后再重试。

Host Requirements RFC推荐初始时间间隔至少为30分钟。发送方至少4~5天内不能放弃。而且，因为交付失败通常是透明的（接收方崩溃或临时网络连接中断），所以当报文在队列中等待的第1个小时内，尝试两次连接是有意义的。

28.3 SMTP的例子

上面我们说明了普通邮件发送，在这里我们将说明MX记录如何用于邮件发送，以及VRFY和EXPN命令的用法。

28.3.1 MX记录：主机非直接连到Internet

在14.6节中我们提到DNS中的一种资源记录类型是邮件交换记录，称为MX记录。在下面的例子中我们将说明如何用MX记录向不直接连到Internet的主机发送邮件。RFC 974 [Partridge 1986] 描述了MTA对MX记录的处理。

主机mlfarm.com不是直接连到Internet的，但是有一个MX记录指向Internet上的一个邮件转发器。

```
sun % host -a -v -t mx mlfarm.com
The following answer is not authoritative:
mlfarm.com              86388   IN   MX   10 mercury.hsi.com
mlfarm.com              86388   IN   MX   15 hsi86.hsi.com
Additional information:
mercury.hsi.com         86388   IN    A   143.122.1.91
hsi86.hsi.com          172762   IN    A   143.122.1.6
```

有两个MX记录，各有不同的优先级。我们希望MTA从优先级数值低的开始。

```
sun % mail -v ron@mlfarm.com                 -v标志看MTA在做什么
To: ron@mlfarm.com
Subject: MX test message
                                             在这里键入报文的正文(没显示出来)
.                                            一行中的一个句号，结束报文
Sending letter ... ron@mlfarm.com...
Connecting to mlfarm.com via tcp...
mail exchanger is mercury.hsi.com            找到MX记录
Trying 143.122.1.91...  connected.           先试优先级低的那一个
220 mercury.hsi.com ...
                                             下面是正常的SMTP邮件传送
```

Linux公社 www.linuxidc.com

从输出中我们看到，MTA发现目的主机有一个MX记录，并使用具有低优先级数值的MX记录。

在主机sun运行这个例子之前，它被配置成不使用本地中继主机，所以我们会看到与目的主机的邮件交换。主机sun还被配置成可使用主机noao.edu（通过拨号SLIP链路）上的域名服务器，所以我们能用tcpdump捕获在SLIP链路上进行的邮件发送和DNS通信。图28-4显示了tcpdump输出的开始部分。

```
1  0.0                 sun.1624 > noao.edu.53: 2+ MX? mlfarm.com. (28)
2  0.445572 (0.4456)   noao.edu.53 > sun.1624: 2* 2/0/2 MX
                                                mercury.hsi.com. 10 (113)

3  0.505739 (0.0602)   sun.1143 > mercury.hsi.com.25: S 1617536000:1617536000(0)
                                                        win 4096
4  0.985428 (0.4797)   mercury.hsi.com.25 > sun.1143: S 1832064000:1832064000(0)
                                                        ack 1617536001 win 16384
5  0.986003 (0.0006)   sun.1143 > mercury.hsi.com.25: . ack 1 win 4096
6  1.735360 (0.7494)   mercury.hsi.com.25 > sun.1143: P 1:90(89) ack 1 win 16384
```

图28-4 向一个使用MX记录的主机发送邮件

在第1行，MTA向它的域名服务器查询mlfarm.com的MX记录。跟在2后面的加号“+”意思是设置要求递归的标志位。第2行的响应置位授权比特（跟在2后面的星号“*”），并包含两个回答RR（两个MX主机名），0个授权RR，以及两个附加的RR（两个主机的IP地址）。

第3~5行与主机mercury.hsi.com上的SMTP建立了一个TCP连接。服务器的初始响应220显示在第6行。

由于某种原因，主机mercury.hsi.com必须把这个邮件报文交付给目的地，mlfarm.com。对于没有连接到Internet上与它的MX站点交换邮件的系统，UUCP协议是一种常用的办法。

在这个例子中，MTA要求一个MX记录，得到一个肯定的结果，然后发送邮件。但不幸的是，MTA与DNS之间的交互随不同的实现而不同。RFC 974指定MTA必须首先要求MX记录，如果没有，就尝试提交给目的主机（也就是说，向DNS要主机的记录和IP地址）。MTA也必须处理DNS中的CNAM记录（规范的名）。

作为一个例子，如果我们从一个BSD/386主机上向rstevens@mailhost.tuc.noao.edu发送邮件，则MTA（Sendmail）执行以下步骤：

1) Sendmail 向DNS询问主机mailhost.tuc.noao.edu的CNAME记录。我们看到存在一个CNAME记录：

```
sun %host -t cname mailhost.tuc.noao.edu
mailhost.tuc.noao.edu     CNAME    noao.edu
```

2) 发布一个要求noao.edu的CNAME记录的DNS查询，回答是不存在。

3) Sendmail向DNS寻求noao.edu的MX记录并得到一个记录：

```
sun %host -t mx noao.edu
noao.edu                  MX       noao.edu
```

4) Sendmail 向DNS查询noao.edu的A记录（IP地址），并得到返回值140.252.1.54（这个A记录大概是由域名服务器为noao.edu返回的，作为第3步中MX应答的一个附加的RR）。

5) 启动一个到140.252.1.54的SMTP连接并发送邮件。

CNAME查询不是为MX记录（noao.edu）中返回的数据做的。MX记录中的数据不能是

Linux公社 www.linuxidc.com

别名 —— 必须是具有一个A记录的主机名。

与只用DNS 的SunOS 4.1.3一起发布的Sendmail版本查询MX记录，并且如果没有找到MX记录就放弃。

28.3.2 MX记录：主机出故障

MX记录的另一个用途是在目的主机出故障时可提供另一个邮件接收器。如果看一下主机sun的DNS入口，我们就会看到它有两个MX记录：

```
sun % host -a -v -t mx sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX       0 sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX      10 noao.edu
Additional information:
sun.tuc.noao.edu        86400     IN      A      140.252.1.29
sun.tuc.noao.edu        86400     IN      A      140.252.13.33
noao.edu                86400     IN      A      140.252.1.54
```

最低优先级的MX记录表明应该首先尝试直接发送到主机本身，下一个优先级是把邮件发送到主机noao.edu。

在下面的描述中，在关掉目的SMTP服务器后，我们从主机vangogh.cs.berkeley.edu向位于主机sun.tuc.noao.edu的我们自己发送邮件。当端口25上的连接请求到达时，TCP应该响应一个RST，因为没有被动打开的进程为等待该端口而挂起。

```
vangogh % mail -v rstevens@sun.tuc.noao.edu
A test to a host that's down.
.
EOT

rstevens@sun.tuc.noao.edu... Connecting to sun.tuc.noao.edu. (smtp)...
rstevens@sun.tuc.noao.edu... Connecting to noao.edu. (smtp)...

220 noao.edu ...
```

下面是正常的*SMTP*邮件传送

我们看到MTA尝试联系sun.tuc.noao.edu，然后放弃，并转而联系noao.edu。

图28-5显示了TCP用一个RST向到来的SYN响应的tcpdump输出。

```
1 0.0                vangogh.3873 > 140.252.1.29.25: S 2358303745:2358303745(0) ...
2 0.000621 (0.0006) 140.252.1.29.25 > vangogh.3873: R 0:0(0) ack 2358303746 win 0

3 0.300203 (0.2996) vangogh.3874 > 140.252.13.33.25: S 2358367745:2358367745(0) ...
4 0.300620 (0.0004) 140.252.13.33.25 > vangogh.3874: R 0:0(0) ack 2358367746 win 0
```

图28-5 尝试连接一个不在运行的SMTP服务器

第1行vangogh向sun的第1个IP地址140.252.1.29 的端口25 发送一个SYN。在第2行它被拒绝。然后，vangogh上的SMTP客户尝试sun的第2个IP地址140.252.13.33（第3行），也产生一个RST的返回（第4行）。

SMTP客户不区分第1行它主动打开时所返回的不同差错，而这是导致它在第 2行尝试其他IP地址的原因。如果第1次的差错是类似“ host unreachable(主机不可达)”，那么第2次尝试或许可行。

如果SMTP客户的主动打开失败的原因是因为服务器主机出故障了，我们将看到客户会向IP地址140.252.1.29重传SYN总共75秒（类似于图18-6）。然后客户向IP地址140.252.13.33发送另一个75秒的其他3个SYN。150秒后客户会移到下一个具有更高优先级的 MX记录。

Linux公社 www.linuxidc.com

28.3.3 VRFY和EXPN命令

VRFY命令无需发送邮件而验证某个接收方地址是否OK。EXPN的目的是无需向邮件表发送邮件就可以扩充该表。许多SMTP实现（如Sendmail）把两者看成一个，但我们提到新的Sendmail区分这两者。

作为一个简单测试，我们可以连到一个新的Sendmail版本，并看到不同之处（已经删除了无关的Telnet客户输出）。

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Tue, 3 Aug 1993 14:
59:12 -0700
220 ESMTP spoken here

helo bsdi.tuc.noao.edu
250 vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you

vrfy nosuchname
550 nosuchname... User unknown

vrfy rstevens
250 Richard Stevens <rstevens@vangogh.CS.Berkeley.EDU>

expn rstevens
250 Richard Stevens <rstevens@noao.edu>
```

首先注意到我们故意在HELO命令中键入错误的主机名：bsdi，而不是sun。许多SMTP服务器得到客户的IP地址，完成一个DNS指针查询（14.5节）并比较主机名。这样允许服务器基于IP地址注册到客户的连接，而不是基于用户可能错误键入的名。某些服务器会用幽默的报文回答，如“你是一个骗子”，或“为什么叫你自己……”。在这个例子中我们看到，这个服务器通过指针查询只打印出我们的真实域名以及我们的IP地址。

然后我们用一个无效的名字键入VRFY命令，服务器就响应550差错。下一步我们键入一个有效的名字，服务器用本地主机上的用户名回答。然后我们试试EXPN命令，并得到一个不同的回答。EXPN命令决定到该用户的邮件是否被转发，并打印出转发的地址。

许多站点禁止VRFY和EXPN命令，有时是因为隐私，有时因为相信这是安全漏洞。例如，我们可以向白宫的SMTP服务器试试下面的命令：

```
sun % telnet whitehouse.gov 25
220 whitehouse.gov SMTP/smap Ready.

helo sun.tuc.noao.edu
250 (sun.tuc.noao.edu) pleased to meet you.

vrfy clinton
500 Command unrecognized

expn clinton
500 Command unrecognized
```

28.4 SMTP的未来

Internet邮件发生了很多改变。应当记得Internet邮件的三个组成部分：信封、首部和正文。新加入的SMTP命令影响了信封，首部中可以使用非ASCII字母，正文（MIME）中也加入了结构。本节中我们依次对这三部分的扩充进行讨论。

Linux公社 www.linuxidc.com

28.4.1 信封的变化：扩充的SMTP

RFC 1425 [Klensin等，1993a] 定义了扩充的SMTP的框架，其结果被称为扩充的 SMTP（ESMTP）。与其他我们已经讨论过的新特性一样，这些变化以向后兼容的方式被加入，所以不影响已有的实现。

如果客户想使用新的特性，首先通过发布一个 EHLO而不是HELO命令启动一个与服务器的会话。相兼容的服务器用 250应答码响应。这个应答通常有好几行，每行都包含一个关键字和一个可选的参数。这些关键字指定了该服务器支持的 SMTP扩充。新的扩充将在一个RFC中描述并以IANA注册（在一个多行应答中，各行数字应答码的后面都要有一个连字符。最后一行的数字应答码后面跟一个空行）。

我们将给出到4个SMTP服务器的初始连接，其中 3个支持扩充的SMTP。我们用Telnet和它们连接，但删掉了不必要的Telnet客户输出。

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Mon, 2 Aug 1993 15:
47:48 -0700
220 ESMTP spoken here

ehlo sun.tuc.noao.edu
250-vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
250-EXPN
250-SIZE
250 HELP
```

这个服务器用一个多行220应答作为它的欢迎报文。对EHLO命令的250应答中列出的扩充命令是EXPN、SIZE和HELP。第一个和最后一个来自原来的 RFC 821规范，但它们是可选命令。ESMTP服务器说明除了新命令外，它们还支持哪些可选的 RFC 821命令。

这个服务器支持的SIZE关键字是在RFC 1427 [Klensin, Freed和Moore 1993] 中定义的。它让客户在MAIL FROM命令行中以字节的多少指定报文的大小，这样服务器就可以在客户开始发送该报文之前，验证它是否接收该长度的报文。增加这个命令的原因在于，随着对非 ASCII码（如图像、音频等）内容的支持， Internet邮件报文的长度在不断增大。

下一个主机也支持ESMTP，注意250应答指明支持包含一个可选参数的 SIZE关键字。这表明该服务器将接受长度不超过461兆字节的报文。

```
sun % telnet ymir.claremont.edu 25
220 ymir.claremont.edu -- Server SMTP (PMDF V4.2-13 #4220)

ehlo sun.tuc.noao.edu
250-ymir.claremont.edu
250-8BITMIME
250-EXPN
250-HELP
250-XADR
250 SIZE 461544960
```

关键字8BITMIME来自于RFC 1426 [Klensin等，1993a]。它允许客户把关键字BODY加到MAIL FROM命令中，指定正文中是否包含NVT ASCII字符（默认的）或8 bit数据。除非客户收到服务器应答EHLO命令发来的8BITMIME关键字，否则禁止客户发送任何非 NVT ASCII字符（当我们在本节中谈到MIME时，我们将看到MIME不要求8 bit传送）。

该服务器也通告了XADR关键字。任何以X开头的关键字都指的是本地SMTP扩充。

Linux公社 www.linuxidc.com

另一个服务器也支持ESMTP，通知了我们已经看到的HELP和SIZE关键字。它也支持三个以X开头的本地扩充。

```
sun % telnet dbc.mtview.ca.us 25
220 dbc.mtview.ca.us Sendmail 5.65/3.1.090690, it's Mon, 2 Aug 93 15:48:50
-0700

ehlo sun.tuc.noao.edu
250-Hello sun.tuc.noao.edu, pleased to meet you
250-HELP
250-SIZE
250-XONE
250-XVRB
250 XQUE
```

最后，我们将看到当客户试图通过向一个不支持 EHLO的服务器发布EHLO命令来使用ESMTP时将发生什么。

```
sun % telnet relay1.uu.net 25
220 relay1.UU.NET Sendmail 5.61/UUNET-internet-primary ready at Mon, 2 Aug
93 18:50:27 -0400

ehlo sun.tuc.noao.edu
500 Command unrecognized

rset
250 Reset state
```

对EHLO命令，客户收到一个500应答而不是250应答。客户应发布RSET命令，并跟着一个HELO命令。

28.4.2 首部变化：非ASCII字符

RFC 1522 [Moore 1993] 指明了一个在RFC 822报文首部中如何发送非ASCII字符的方法。这样做的主要用途是为了允许在发送方名、接收方名以及主题中使用其他的字符。

首部字段中可以包含编码字(coded word)。它们具有以下格式：

=?*charset*?*encoding*?*encoded-text*?=

charset是字符集规范。有效值是两个字符串us-ascii和iso-8859-x，其中x是一个单个数字，例如在iso-8859-1中的数字“1”。

encoding是一个单个字符用来指定编码方法，支持两个值。

1) Q编码意思是引号中可打印的（quoted-printable），目的是用于拉丁字符集。大多数字符是作为NVT ASCII（当然最高位比特置0）发送的。任何要发送的字符若其第8比特置1则被作为3个字符发送：第1个是字符是“=”，跟着两个十六进制数。例如，字符é（它的二进制8 bit值为0xe9）作为三个字符发送：=E9。空格通常作为下划线或三个字符=20发送。这种编码的目的在于，某些文本中除了大多数ASCII字符外，还有几个特殊字符。

2) B意思是以64为基数的编码。文本中的3个连续字节（24bit）被编码成4个6 bit值。用于表示所有可能的6bit值的64个NVT ASCII字符如图28-6所示。当要编码的个数不是3的倍数时，等号符“=”被用作填充符。

下面两种编码方式的例子取自RFC 1522：

```
From: =?US-ASCII?Q?Keith_Moore?= <moore@cs.utk.edu>
To: =?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?= <keld@dkuug.dk>
CC: =?ISO-8859-1?Q?Andr=E9_?= Pirard <PIRARD@vm1.ulg.ac.be>
Subject: =?ISO-8859-1?B?SWYgeW91IGNhbiByZWFkIHRoaXMgeW8=?=
 =?ISO-8859-2?B?dSB1bmRlcnN0YW5kIHRoZSBleGFtcGxlLg==?=
```

Linux公社 www.linuxidc.com

6 bit 值	ASCII 字符	6 bit 值	ASCII 字符	6 bit 值	ASCII 字符	6 bit 值	ASCII 字符
0	A	10	Q	20	g	30	w
1	B	11	R	21	h	31	x
2	C	12	S	22	i	32	y
3	D	13	T	23	j	33	z
4	E	14	U	24	k	34	0
5	F	15	V	25	l	35	1
6	G	16	W	26	m	36	2
7	H	17	X	27	n	37	3
8	I	18	Y	28	o	38	4
9	J	19	Z	29	p	39	5
a	K	1a	a	2a	q	3a	6
b	L	1b	b	2b	r	3b	7
c	M	1c	c	2c	s	3c	8
d	N	1d	d	2d	t	3d	9
e	O	1e	e	2e	u	3e	+
f	P	1f	f	2f	v	3f	/

图28-6 6 bit值的编码（以64为基数编码）

能处理这些首部的用户代理将输出：

```
From: Keith Moore <moore@cs.utk.edu>
To: Keld Jørn Simonsen <keld@dkuug.dk>
CC: André  Pirard <PIRARD@vm1.ulg.ac.be>
Subject: If you can read this you understand the example.
```

为说明以64为基数的编码方法是如何工作的，我们看一下主题行中前面 4个编码的字符：SWYg。按照图28-6写出这4个字符的6 bit值（S=0x12,W=0x16,Y=0x18以及g=0x20）的二进制码：

```
010010  010110  011000  100000
```

然后把这24 bit重新分组成3个8 bit字节：

```
01001001  01100110  00100000
 =0x49     =0x66     =0x20
```

它们是I、f和空格的ASCII表示。

28.4.3 正文变化：通用Internet邮件扩充

我们已经提到RFC 822指定正文是NVT ASCII文本行，没有结构。RFC 1521 [Borenstein和Freed 1993] 把扩充定义为允许把结构置入正文。这被称为MIME，即通用Internet邮件扩充。

MIME不要求任何扩充，我们在本节前面已作了说明（扩充的 SMTP或非ASCII标题）。MIME正好加入了一些告知收件者正文结构的新标题（与 RFC 822相一致）。正文仍可以用NVT ASCII码来发送，而不考虑邮件内容。虽然我们前面所述的一些扩充可能会和 MIME合在一起产生好的效果——扩充的SMTP SIZE命令，因为MIME报文能变得很长，以及非ASCII标题——这些扩充并不是MIME所要求的。与另一方交换MIME报文所需的一切，就是双方都要有一个能够理解MIME的用户代理。在任何一个MTA中不需要做任何改变。

MIME定义这5个新标题字段如下：

```
Mime-Version:
Content-Type:
Content-Transfer-Encoding:
```

Linux公社 www.linuxidc.com

```
Content-ID:
Content-Description:
```

作为例子，下面两个标题行可以出现在一个 Internet邮件报文中：

```
Mime-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
```

当前MIME版本是1.0，内容类型是无格式 ASCII码文本，即Internet邮件的默认选择。PLAIN这个字被认为是内容类型（TEXT）的一个子类型，字符串 charset=US-ASCII是一个参数。

Text是MIME的7个被定义的内容类型之一。图 28-7总结了RFC 1521中定义的16个不同的内容类型和子类型。对具体的内容类型和子类型来说都有指定的很多参数。

内容类型	子类型	描　　述
text	plain	无格式文本
	richtext	简单格式文本，如粗体、斜体或下划线等
	enriched	richtext的简化和改进
multipart	mixed	多个正文部分，串行处理
	parallel	多个正文部分，可并行处理
	digest	一个电子邮件的摘要
	altenative	多个正文部分，具有相同的语义内容
message	rfc822	内容是另一个RFC 822邮件报文
	partial	内容是一个邮件报文的片断
	external-body	内容是指向实际报文的指针
application	octet-stream	任意二进制数据
	postscript	一个PostScript程序
image	jpeg	ISO 10918格式
	gif	CompuServe的图形交换格式
audio	basic	用8 bit ISDN μ律格式编码
video	mpeg	ISO 11172格式

图28-7　MIME 内容类型和子类型

内容类型和用于内容的传送编码是相互独立的。前者由首部字段 Content-Type指明，后者由首部字段Content-Transfer-Encoding指明。在RFC 1521中定义了5种不同的编码格式。

1) 7bit，是默认的NVT ASCII；

2) quoted-printable，我们在前面的一个例子中看到有非 ASCII首部。当字符中只有很少一部分的第8 bit置1时非常有用；

3) base64，如图28-6所示；

4) 8bit，包含字符行，其中某些为非ASCII字符且第8bit置1；

5) binary编码，无需包含多行的8 bit数据。

对RFC 821 MTA，以上5种编码格式中只有前 3种是有效的。因为这 3种产生只包含NVT ASCII字符的正文。使用有8BITMIME支持的扩充SMTP允许使用8bit编码。

尽管内容类型和编码是独立的，RFC1521推荐有非ASCII数据的text使用quoted-printable，而image、audio、video和octet-stream application使用base64。这样允许与符合 RFC 821的MTA保持最大的互操作性。而且，multipart和message内容类型必须以7bit编码。

Linux公社 www.linuxidc.com

作为一个multipart内容类型的例子，图28-8显示了一个来自RFC发布清单的邮件报文。子类型是mixed，意思是各部分是顺序处理的，各部分的边界是字符串 NextPart，其前面是行首的两个连字符。

每个边界上可跟一行用于指明下一部分首部字段。忽略报文中第 1个边界之前和最后一个边界之后的所有内容。

因为在第一个边界后面跟着一个空行，而不是首部，所以在第 1个和第2个边界之间的数据的内容类型被假定为具有us-ascii字符集的text/plain。这是新RFC的文字描述。

但是第2个边界后面跟着首部字段。它指定了另一个multipart报文，具有边界OtherAccess。子类型为alternative，有两种不同的选择。第1种OtherAccess选项是用电子邮件获取RFC，第2种选项是用匿名FTP获取。MIME用户代理将列出这两种选项，允许我们选择一个，然后自动地用电子邮件或匿名FTP获取一份复制的RFC。

```
To: rfc-dist@nic.ddn.mil
Subject: RFC1479 on IDPR Protocol
Mime-Version: 1.0
Content-Type: Multipart/Mixed; Boundary="NextPart"
Date: Fri, 23 Jul 93 12:17:43 PDT
From: "Joyce K. Reynolds" <jkrey@isi.edu>


--NextPart                                  第1个边界

A new Request for Comments is now available in online RFC libraries.

. . .                                       这里的细节在新的RFC中

Below is the data which will enable a MIME compliant Mail Reader
implementation to automatically retrieve the ASCII version
of the RFCs.

--NextPart                                  第2个边界
Content-Type: Multipart/Alternative; Boundary="OtherAccess"
                                            一个具有新边界的嵌套的多部分报文

--OtherAccess
Content-Type:  Message/External-body;
        access-type="mail-server";
        server="mail-server@nisc.sri.com"

Content-Type: text/plain

SEND rfc1479.txt

--OtherAccess
Content-Type:   Message/External-body;
        name="rfc1479.txt";
        site="ds.internic.net";
        access-type="anon-ftp";
        directory="rfc"

Content-Type: text/plain


--OtherAccess--
--NextPart--                                最后的边界
```

图28-8 MIME multipart报文的例子

Linux公社 www.linuxidc.com

这一部分是MIME的一个简要概述。MIME的详细细节和例子，见RFC 1521和[Rose 1993]。

28.5 小结

电子邮件包括在两端（发送方和接收方）都有的一个用户代理以及两个或多个报文传送代理。可以把一个邮件报文分成三个部分：信封、首部和正文。我们已经看到这三个部分用SMTP和Internet标准是如何进行交换的。所有都作为NVT ASCII字符进行交换。

我们也看到了一些新的扩充：用于信封和非ASCII首部的扩充SMTP，以及使用MIME的正文增加了结构。MIME的结构和编码允许使用已有的7bit SMTP MTA交换任意二进制数据。

习题

28.1 读RFC 822，找到域文字（domain literal）的意思。试试用其中一个给自己发送邮件。

28.2 除了连接建立和终止外，要发送一个小的邮件报文的最小网络往返次数是多少？

28.3 TCP是一个全双工协议，但是SMTP用半双工的形式使用TCP。客户发送一个命令后停止等待应答。为什么客户不一次发送多个命令，如一行中包括HELO、MAIL、RCPT、DATA和QUIT命令（假定正文不是太大）？

28.4 当网络在接近其容量运行时，SMTP的这种半双工操作如何欺骗缓慢的启动机制？

28.5 当存在多个具有相同优先值的MX记录时，名服务器是否总能以相同的顺序返回它们？

Linux公社 www.linuxidc.com

第29章　网络文件系统

29.1　引言

本章中我们要讨论另一个常用的应用程序： NFS（网络文件系统），它为客户程序提供透明的文件访问。NFS的基础是Sun RPC：远程过程调用。我们首先必须描述一下 RPC。

客户程序使用 NFS不需要做什么特别的工作，当 NFS内核检测到被访问的文件位于一个NFS服务器时，就会自动产生一个访问该文件的 RPC调用。

我们对NFS如何访问文件的细节并不感兴趣，只对它如何使用 Internet的协议，尤其是UDP协议，感兴趣。

29.2　Sun远程过程调用

大多数的网络程序设计都是编写一些调用系统提供的函数来完成特定的网络操作的应用程序。例如，一个函数完成 TCP的主动打开，另一个完成 TCP的被动打开，一个函数在一个TCP连接上发送数据，另一个设置特定的协议选项（如激活 TCP的keepalive定时器）。在1.15节我们提到过两个常用的用于网络编程的函数集（ API）：插口(socket)和TLI。正像客户端和服务器端运行的操作系统可能会不相同一样，双方使用的 API也可能会不相同。由通信协议和应用协议决定一对客户和服务器是否可以彼此通信。如果两台主机连接在一个网络上，并且都有一个TCP/IP的实现，那么一台主机上的一个使用 C语言编写的、使用插口和 TCP的Unix客户程序可以和另一台主机上的一个使用 COBOL语言编写的、使用其他 API和TCP的大型机服务器进行通信。

一般来说，客户发送命令给服务器，服务器向客户发送应答。目前为止，我们讨论过的所有应用程序 —— Ping，Traceroute，选路守护程序、以及 DNS、TFTP、BOOTP、SNMP、Telnet、FTP和SMTP的客户和服务器 —— 都是采用这种方式实现的。

远程过程调用RPC (Remote Procedure Call)是一种不同的网络程序设计方法。客户程序编写时只是调用了服务器程序提供的函数。这只是程序员所感觉到的，实际上发生了下面一些动作。

1) 当客户程序调用远程的过程时，它实际上只是调用了一个位于本机上的、由 RPC程序包生成的函数。这个函数被称为客户残桩（ stub）。客户残桩将过程的参数封装成一个网络报文，并且将这个报文发送给服务器程序。

2) 服务器主机上的一个服务器残桩负责接收这个网络报文。它从网络报文中提取参数，然后调用应用程序员编写的服务器过程。

3) 当服务器函数返回时，它返回到服务器残桩。服务器残桩提取返回值，把返回值封装成一个网络报文，然后将报文发送给客户残桩。

4) 客户残桩从接收到的网络报文中取出返回值，将其返回给客户程序。

网络程序设计是通过残桩和使用诸如插口或 TLI的某个API的RPC库例程来实现的，但是

Linux公社 www.linuxidc.com

用户程序——客户程序和被客户程序调用的服务器过程——不会和这个API打交道。客户应用程序只是调用服务器的过程，所有网络程序设计的细节都被 RPC程序包、客户残桩和服务器残桩所隐藏。

一个RPC程序包提供了很多好处。

1) 程序设计更加容易，因为很少或几乎没有涉及网络编程。应用程序设计员只需要编写一个客户程序和客户程序调用的服务器过程。

2) 如果使用了一个不可靠的协议，如 UDP，像超时和重传等细节就由 RPC程序包来处理。这就简化了用户应用程序。

3) RPC库为参数和返回值的传输提供任何需要的数据转换。例如，如果参数是由整数和浮点数组成的，RPC程序包处理整数和浮点数在客户机和服务器主机上存储的不同形式。这个功能简化了在异构环境中的客户和服务器的编码问题。

RPC程序设计的细节可以参看参考文献 [Stevens 1990]的第18章。两个常用的RPC程序包是Sun RPC和开放软件基金（OSF）分布式计算环境（DCE）的RPC程序包。我们对于RPC的兴趣在于想了解Sun RPC中过程调用和过程返回报文的形式，因为本章中讨论的网络文件系统使用了它们。Sun RPC的第2版定义在RFC 1057 [Sun Microsystems 1988a]中。

Sun RPC

Sun RPC有两个版本。一个版本建立在插口API基础上，和TCP和UDP打交道。另一个称为 TI-RPC的（独立于运输层），建立在TLI API基础上，可以和内核提供的任何运输层协议打交道。尽管本章中我们只讨论 TCP和UDP，从讨论的观点来看，两者是一样的。

图29-1显示的是使用 UDP时，一个RPC过程调用报文的格式。IP首部和UDP首部是标准的首部，我们已经在图 3-1和图11-2中显示过。UDP首部以下是 RPC程序包定义的部分。

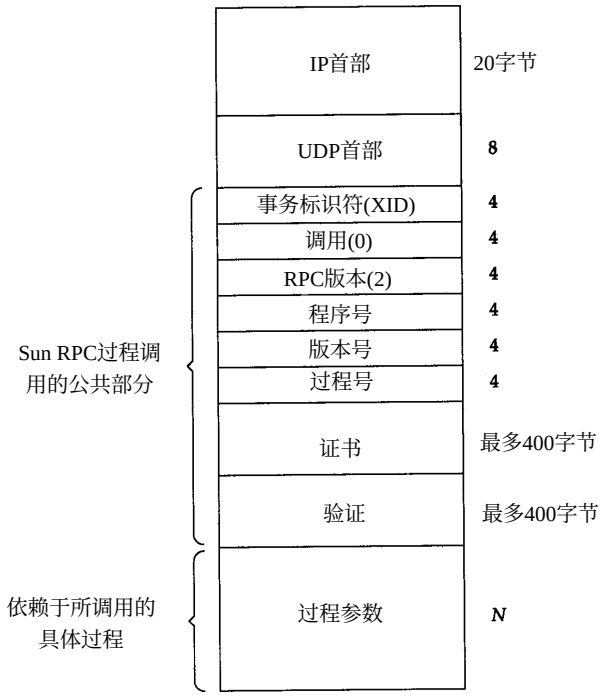


图29-1 RPC过程调用报文作为一个UDP数据报的格式

事务标识符（XID）由客户程序设置，由服务器程序返回。当客户收到一个应答，它将服务器返回的 XID与它发送的请求的XID相比较。如果不匹配，客户就放弃这个报文，等待从服务器返回的下一个报文。每次客户发出一个新的 RPC，它就会改变报文的XID。但是如果客户重传一个以前发送过的 RPC（因为它没有收到服务器的一个应答），重传报文的XID不会修改。

调用(call)变量在过程调用报文中设置为 0，在应答报文中设置为 1。当前的RPC版本是2。接下来三个变量：程序号、版本号和过程号，标识了服务器上被调用的特定过程。

Linux公社 www.linuxidc.com

证书(credential)字段标识了客户。有些情况下，证书字段设置为空值；另外一些情况下，证书字段设置为数字形式的客户的用户号和组号。服务器可以查看证书字段以决定是否执行请求的过程。验证 (verifier)字段用于使用了DES加密的安全RPC。尽管证书字段和验证字段是可变长度的字段，它们的长度也作为字段的一部分被编码。

接下来是过程参数(procedure parameter)字段。参数的格式依赖于远程过程的定义。接收者（服务器残桩）如何知道参数字段的大小呢？既然使用的是UDP协议，UDP数据报的大小减去验证字段以上所有字段的长度就是参数的大小。如果使用的不是UDP而是TCP，因为TCP是一个字节流协议，没有记录边界，所以没有固定的长度。为了解决这个问题，在TCP首部和XID之间增加了一个4字节的长度字段，告诉接收者这个RPC调用由多少字节组成。这也使得一个RPC调用报文在必要时可以用多个TCP段来传输（DNS使用了类似的技术，参见习题14-4）。

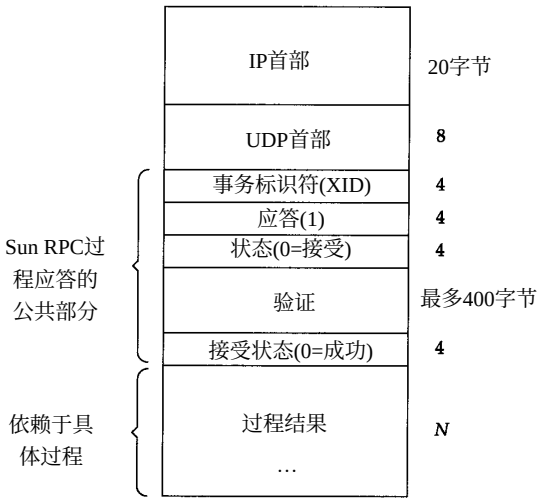


图29-2 RPC应答报文作为一个UDP数据报的格式

图29-2显示了一个RPC应答报文的格式。当远程过程返回时，服务器残桩将这个报文发送给客户残桩。

应答报文中的XID字段是从调用报文的XID字段复制而来。应答字段设置为1，以区别于调用报文。如果调用报文被接受，状态字段设置为0（如果RPC的版本号不为2，或者服务器不能鉴别客户的身份，调用报文可能被拒绝）。安全的RPC使用验证字段来标识服务器。

如果远程过程调用成功，接受状态字段置为0。一个非零的值可能表示一个不合法的版本号或者一个不合法的过程号。如果使用的不是UDP而是TCP，如同RPC调用报文一样，在TCP首部和XID字段之间插入一个4字节的长度字段。

29.3 XDR: 外部数据表示

外部数据表示XDR (eXternal Data Representation)是一个标准，用来对RPC调用报文和应答报文中的值进行编码。这些值包括RPC首部字段（XID、程序号、接受状态等）、过程参数和过程结果。采用标准化的方法对这些值进行编码使得一个系统中的客户可以调用另一个不同架构的系统中的一个过程。XDR在RFC 1014中定义[Sun Microsystems 1987]。

XDR定义了很多数据类型以及它们如何在一个RPC报文中传输的具体形式（如比特顺序，字节顺序等）。发送者必须采用XDR格式构造一个RPC报文，然后接收者将XDR格式的报文转换为本机的表示形式。例如，在图29-1和图29-2中，我们显示的所有整数值（XID、调用字段、程序号等）都是4字节的整数。在XDR中，所有的整数的确占据4个字节。XDR支持的其他数据类型包括无符号整数、布尔类型、浮点数、定长数组、可变长数组和结构。

29.4 端口映射器

包含远程过程的RPC服务器程序使用的是临时端口，而不是知名端口。这就需要某种形

Linux公社 www.linuxidc.com

式的“注册”程序来跟踪哪一个RPC程序使用了哪一个临时端口。在Sun RPC中，这个注册程序被称为端口映射器(port mapper)。

> “端口”这个词作为Internet协议族的一个特征，来自于TCP和UDP端口号。既然TI-RPC可以工作在任何运输层协议之上，而不仅仅是TCP和UDP，所以使用TI-RPC的系统中（如SVR4和Solaris 2.2），端口映射器的名字变成了rpcbind。下面我们继续使用更为常见的端口映射器的名字。

很自然地，端口映射器本身必须有一个知名端口：UDP端口111和TCP端口111。端口映射器也就是一个RPC服务器程序。它有一个程序号（100000）、一个版本号（2）、一个TCP端口111和一个UDP端口111。服务器程序使用RPC调用向端口映射器注册自身，客户程序使用RPC调用向端口映射器查询。端口映射器提供四个服务过程：

1) PMAPPROC_SET。一个RPC服务器启动时调用这个过程，注册一个程序号、版本号和带有一个端口号的协议。

2) PMAPPROC_UNSET。RPC服务器调用此过程来删除一个已经注册的映射。

3) PMAPPROC_GETPORT。一个RPC客户启动时调用此过程。根据一个给定的程序号、版本号和协议来获得注册的端口号。

4) PMAPPROC_DUMP。返回端口映射器数据库中所有的记录（每个记录包括程序号、版本号、协议和端口号）：

在一个RPC服务器程序启动，接着被一个RPC客户程序调用的过程中，进行了以下一些步骤：

1) 一般情况下，当系统引导时，端口映射器必须首先启动。它创建一个TCP端点，并且被动打开TCP端口111。它也创建一个UDP端点，并且在UDP端口111等待着UDP数据报的到来。

2) 当RPC服务器程序启动时，它为它所支持的程序的每一个版本创建一个TCP端点和一个UDP端点（一个给定的RPC程序可以支持多个版本。客户调用一个服务器过程时，说明它想要哪一个版本）。两个端点各自绑定一个临时端口（TCP端口号和UDP端口号是否一致无关紧要）。服务器通过RPC调用端口映射器的PMAPPROC_SET过程，注册每一个程序、版本、协议和端口号。

3) 当RPC客户程序启动时，它调用端口映射器的PMAPPROC_GETPORT过程来获得一个指定程序、版本和协议的临时端口号。

4) 客户发送一个RPC调用报文给第3步返回的端口号。如果使用的是UDP，客户只是发送一个包含RPC调用报文（见图29-1）的UDP数据报到服务器相应的UDP端口。服务器发送一个包含RPC应答报文（见图29-2）的UDP数据报到客户作为响应。

如果使用的是TCP，客户对服务器的TCP端口号做一个主动打开，然后在建立的TCP连接上发送一个RPC调用报文。服务器作为响应，在连接上发送一个RPC应答报文。

程序rpcinfo(8)打印了端口映射器中当前的映射记录（它调用了端口映射器的PMAPPROC_DUMP过程）。这里给出的是典型的输出：

```
sun % /usr/etc/rpcinfo -p
   program vers proto   port
    100005    1   tcp    702  mountd      NFS的安装守护程序
    100005    1   udp    699  mountd
    100005    2   tcp    702  mountd
    100005    2   udp    699  mountd
```

Linux公社 www.linuxidc.com

```
100003    2    udp    2049   nfs          NFS本身
100021    1    tcp     709   nlockmgr     NFS的加锁管理程序
100021    1    udp    1036   nlockmgr
100021    2    tcp     721   nlockmgr
100021    2    udp    1039   nlockmgr
100021    3    tcp     713   nlockmgr
100021    3    udp    1037   nlockmgr
```

可以看出一些程序确实支持多个版本。在端口映射器中，每一个程序号、版本号和协议的组合都有自己的端口号映射。

安装守护程序（mount daemon）的两个版本可以通过同样的TCP端口号（702）和同样的UDP端口号（699）来访问，而加锁管理程序（lock manager）的每个版本都有各自不同的端口号。

29.5 NFS协议

使用NFS，客户可以透明地访问服务器上的文件和文件系统。这不同于提供文件传输的FTP（第27章）。FTP会产生文件一个完整的副本。NFS只访问一个进程引用文件的那一部分，并且NFS的一个目的就是使得这种访问透明。这就意味着任何能够访问一个本地文件的客户程序不需要做任何修改，就应该能够访问一个NFS文件。

NFS是一个使用Sun RPC构造的客户服务器应用程序。NFS客户通过向一个NFS服务器发送RPC请求来访问其上的文件。尽管这一工作可以使用一般的用户进程来实现——即NFS客户可以是一个用户进程，对服务器进行显式调用。而服务器也可以是一个用户进程——因为两个理由，NFS一般不这样实现。首先，访问一个NFS文件必须对客户透明。因此，NFS的客户调用是由客户操作系统代表用户进程来完成的。第二，出于效率的考虑，NFS服务器在服务器操作系统中实现。如果NFS服务器是一个用户进程，每个客户请求和服务器应答（包括读和写的数据）将不得不在内核和用户进程之间进行切换，这个代价太大。

本节中，我们考察在RFC1094中说明的第2版的NFS [Sun Microsystems 1988b]。[X/Open 1991] 中给出了Sun RPC、XDR和NFS的一个更好的描述。[Stern 1991] 给出了使用和管理NFS的细节。第3版的NFS协议在1993年发布，我们在29.7节中对它做一个简单的描述。

图29-3显示了一个NFS客户和一个NFS服务器的典型配置，图中有很多地方需要注意。

1) 访问的是一个本地文件还是一个NFS文件对于客户来说是透明的。当文件被打开时，由内核决定这一点。文件被打开之后，内核将本地文件的所有引用传递给名为“本地文件访问”的框中，而将一个NFS文件的所有引用传递给名为“NFS客户”的框中。

2) NFS客户通过它的TCP/IP模块向NFS服务器发送RPC请求。NFS主要使用UDP，最新的实现也可以使用TCP。

3) NFS服务器在端口2049接收作为UDP数据报的客户请求。尽管NFS可以被实现成使用端口映射器，允许服务器使用一个临时端口，但是大多数的实现都是直接指定UDP端口2049。

4) 当NFS服务器收到一个客户请求时，它将这个请求传递给本地文件访问例程，后者访问服务器主机上的一个本地的磁盘文件。

5) NFS服务器需要花一定的时间来处理一个客户的请求。访问本地文件系统一般也需要一部分时间。在这段时间间隔内，服务器不应该阻止其他的客户请求得到服务。为了实现这一功能，大多数的NFS服务器都是多线程的——即服务器的内核中实际上有多个NFS服务器在

Linux公社 www.linuxidc.com

运行。具体怎么实现依赖于不同的操作系统。既然大多数的 Unix内核不是多线程的，一个共同的技术就是启动一个用户进程（常被称为 nfsd）的多个实例。这个实例执行一个系统调用，使自己作为一个内核进程保留在操作系统的内核中。

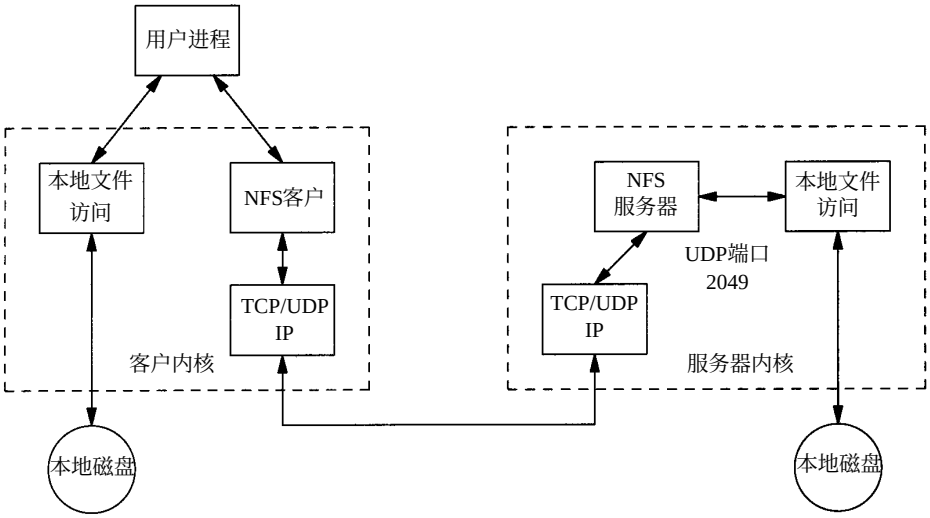


图29-3 NFS客户和NFS服务器的典型配置

6) 同样，在客户主机上，NFS客户需要花一定的时间来处理一个用户进程的请求。NFS客户向服务器主机发出一个 RPC调用，然后等待服务器的应答。为了给使用 NFS的客户主机上的用户进程提供更多的并发性，在客户内核中一般运行着多个 NFS客户。同样，具体实现也依赖于操作系统。Unix系统经常使用类似于NFS服务器的技术：一个叫作biod的用户进程执行一个系统调用，作为一个内核进程保留在操作系统的内核中。

大多数的Unix主机可以作为一个NFS客户，一个NFS服务器，或者两者都是。大多数PC机的实现（MS-DOS）只提供了NFS客户实现。大多数的IBM大型机只提供了NFS服务器功能。

NFS实际上不仅仅由NFS协议组成。图29-4显示了NFS使用的不同RPC程序。

应用程序	程序号	版本号	过程数
端口映射器	100000	2	4
NFS	100003	2	15
安装程序	100005	1	5
加锁管理程序	100021	1, 2, 3	19
状态监视器	100024	1	6

图29-4 NFS使用的不同RPC程序

在这个图中，程序的版本是在SunOS 4.1.3中使用的。更新的实现提供了其中一些程序更新的版本。例如，Solaris 2.2还支持端口映射器的第3版和第4版，以及安装守护程序的第2版。SVR4支持第3版的端口映射器。

在客户能够访问服务器上的文件系统之前，NFS客户主机必须调用安装守护程序。我们在下面讨论安装守护程序。

加锁管理程序和状态监视器允许客户锁定一个 NFS服务器上文件的部分区域。这两个程

Linux公社 www.linuxidc.com

序独立于NFS协议，因为加锁需要知道客户和服务器的状态，而 NFS本身在服务器上是无状态的（下面我们对 NFS的无状态会介绍得更多）。[X/Open 1991] 的第9, 10和11章说明了使用加锁管理程序和状态监视器进行 NFS文件锁定的过程。

29.5.1 文件句柄

NFS中一个基本概念是文件句柄(file handle)。它是一个不透明(opaque)的对象，用来引用服务器上的一个文件或目录。不透明指的是服务器创建文件句柄，把它传递给客户，然后客户访问文件时，使用对应的文件句柄。客户不会查看文件句柄的内容 —— 它的内容只对服务器有意义。

每次一个客户进程打开一个实际上位于一个 NFS服务器上的文件时， NFS客户就会从NFS服务器那里获得该文件的一个文件句柄。每次 NFS客户为用户进程读或写文件时，文件句柄就会传给服务器以指定被访问的文件。

一般情况下，用户进程不会和文件句柄打交道 —— 只有NFS客户和NFS服务器将文件句柄传来传去。在第2版的NFS中，一个文件句柄占据32个字节，第3版中增加为64个字节。

Unix服务器一般在文件句柄中存储下面的信息：文件系统标识符（文件系统最大和最小的设备号），i-node号（在一个文件系统中唯一的数值）和一个 i-node的生成码（每当一个i-node被一个不同的文件重用时就改变的数值）。

29.5.2 安装协议

客户必须在访问服务器上一个文件系统中的文件之前，使用安装协议安装那个文件系统。一般情况下，这是在客户主机引导时完成的。最后的结果就是客户获得服务器文件系统的一个文件句柄。

图29-5显示了一个Unix客户发出mount(8)命令所发生的情况，它说明一个NFS的安装过程。

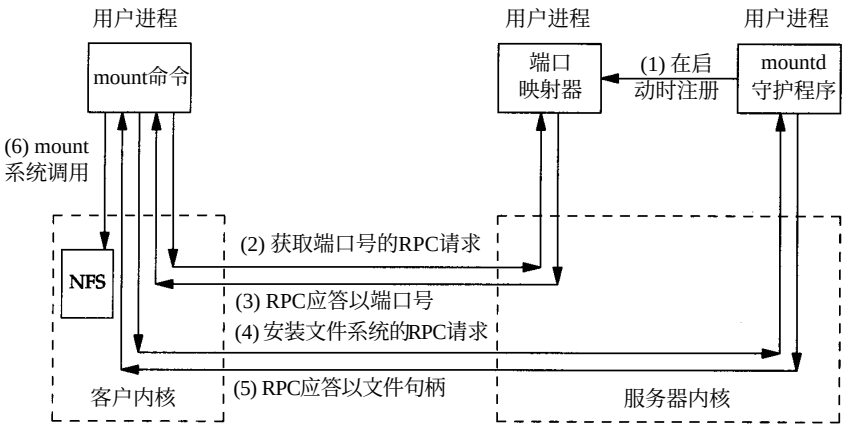


图29-5 使用Unix mount命令的安装协议

依次发生了下面的动作。

1) 服务器上的端口映射器一般在服务器主机引导时被启动。

2) 安装守护程序（mountd）在端口映射器之后被启动。它创建了一个 TCP端点和一个

Linux公社 www.linuxidc.com

UDP端点，并分别赋予一个临时的端口号。然后它在端口映射器中注册这些端口号。

3) 在客户机上执行mount命令，它向服务器上的端口映射器发出一个RPC调用来获得服务器上安装守护程序的端口号。客户和端口映射器交互既可以使用TCP也可以使用UDP，但一般使用UDP。

4) 端口映射器应答以安装守护程序的端口号。

5) mount命令向安装守护程序发出一个RPC调用来安装服务器上的一个文件系统。同样，既可以使用TCP也可以使用UDP，但一般使用UDP。服务器现在可以验证客户，使用客户的IP地址和端口号来判别是否允许客户安装指定的文件系统。

6) 安装守护程序应答以指定文件系统的文件句柄。

7) 客户机上的mount命令发出mount系统调用将第5步返回的文件句柄与客户机上的一个本地安装点联系起来。文件句柄被存储在NFS客户代码中，从现在开始，用户进程对于那个服务器文件系统的任何引用都将从使用这个文件句柄开始。

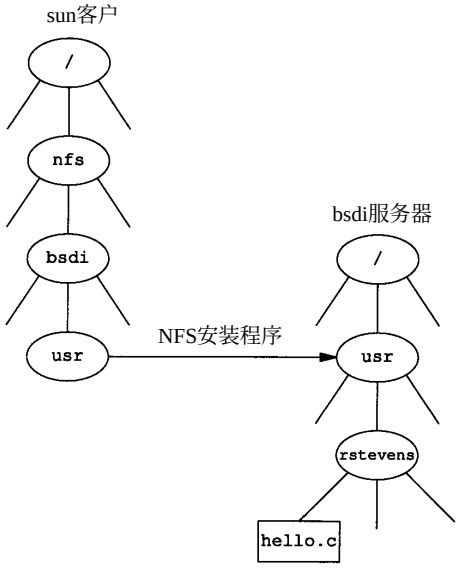


图29-6 将bsdi:/usr 目录安装成主机sun上的/nfs/bsdi/usr 目录

上述实现技术将所有的安装处理，除了客户机上的mount系统调用，都放在用户进程中，而不是放在内核中。我们显示的三个程序——mount命令、端口映射器和安装守护程序——都是用户进程。

作为一个例子，在我们的主机sun（一个NFS客户机）上执行：

```
sun  # mount -t nfs bsdi:/usr /nfs/bsdi/usr
```

这个命令将主机bsdi（一个NFS服务器）上的/usr目录安装成为本地文件系统/nfs/bsdi/usr。图29-6显示了结果。

当我们引用客户机sun上的/nfs/bsdi/usr/rstevens/hello.c文件时，实际上引用的是服务器bsdi上的文件/usr/rstevens/hello.c。

29.5.3 NFS过程

现在我们描述NFS服务器提供的15个过程（使用的个数与NFS过程的实际个数不一样，因为我们把它们按照功能分了组）。尽管NFS被设计成可以在不同的操作系统上工作，而不仅仅是Unix系统，但是一些提供Unix功能的过程可能不被其他操作系统支持（例如硬链接、符号链接、组的属主和执行权等）。[Stevens 1992]的第4章包含了Unix文件系统其他的一些信息，其中有些被NFS采用。

1) GETATTR。返回一个文件的属性：文件类型（一般文件，目录等）、访问权限、文件大小、文件的属主者及上次访问时间等信息。

2) SETATTR。设置一个文件的属性。只允许设置文件属性的一个子集：访问权限、文件

Linux公社 www.linuxidc.com

的属主、组的属主、文件大小、上次访问时间和上次修改时间。

3) STATFS。返回一个文件系统的状态：可用空间的大小、最佳传送大小等。例如 Unix的df命令使用此过程。

4) LOOKUP。查找一个文件。每当一个用户进程打开一个 NFS服务器上的一个文件时，NFS客户调用此过程。

5) READ。从一个文件中读数据。客户说明文件的句柄、读操作的开始位置和读数据的最大字节数（最多8192个字节）。

6) WRITE。对一个文件进行写操作。客户说明文件的句柄、开始位置、写数据的字节数和要写的数据。

7) CREATE。创建一个文件。

8) REMOVE。删除一个文件。

9) RENAME。重命名一个文件。

10) LINK。为一个文件构造一个硬链接。硬链接是一个 Unix的概念，指的是磁盘中的一个文件可以有任意多个目录项（即名字，也叫作硬链接）指向它。

11) SYMLINK。为一个文件创建一个符号链接。符号链接是一个包含另一个文件名字的文件。大多数引用符号链接的操作（例如，打开）实际上引用的是符号链接所指的文件。

12) READLINK。读一个符号链接。即返回符号链接所指的文件的名字。

13) MKDIR。创建一个目录。

14) RMDIR。删除一个目录。

15) READDIR。读一个目录。例如， Unix的ls命令使用此过程。

这些过程实际上有一个前缀NFSPROC_，我们把它省略了。

29.5.4 UDP还是TCP

NFS最初是用UDP写的，所有的厂商都提供了这种实现。最新的一些实现也支持 TCP。TCP支持主要用于广域网，它可以使文件操作更快。 NFS已经不再局限于局域网的使用。

当从LAN转换到WAN时，网络的动态特征变化得非常大。往返时间（ round-trip time）变动范围大，拥塞经常发生。 WAN的这些特征使得我们考虑使用具有 TCP属性的算法 —— 慢启动，但是可以避免拥塞。既然 UDP没有提供任何类似的东西，那么在 NFS客户和服务器上加进同样的算法或者使用TCP。

29.5.5 TCP上的NFS

伯克利实现的Net/2 NFS支持UDP或者TCP。[Macklem 1991]描述了这个实现。让我们看一下使用TCP有什么不同。

1) 当服务器主机进行引导时，它启动一个 NFS服务器，后者被动打开 TCP端口2049，等待着客户的连接请求。这通常是另一个 NFS服务器，正常的NFS UDP服务器在UDP端口2049等待着进入的UDP数据报。

2) 当客户使用TCP安装服务器上的文件系统时，它对服务器上的 TCP端口2049做一个主动打开。这样就为这个文件系统在客户和服务器之间形成了一个 TCP连接。如果同样的客户安装同样服务器上的另一个文件系统，就会创建另一个 TCP连接。

Linux公社 www.linuxidc.com

3) 客户和服务器在它们连接的两端都要设置 TCP的keepalive选项，这样双方都能检测到对方主机崩溃，或者崩溃然后重启动。

4) 客户方所有使用这个服务器文件系统的应用程序共享这个 TCP连接。例如，在图29-6中，如果在bsdi的/usr目录下还有另一个目录smith，那么对两个目录/nfs/bsdi/usr/rstevens和/nfs/bsdi/usr/smith下所有文件的引用将共享同样的TCP连接。

5) 如果客户检测到服务器已经崩溃，或者崩溃然后重启动（通过收到一个 TCP差错“连接超时”或者“对方复位连接”），它尝试与服务器重新建立连接。客户做另一个主动打开，为同一个文件系统请求重新建立 TCP连接。在以前连接上超时的所有客户请求在新的连接上都会重新发出。

6) 如果客户机崩溃，那么当它崩溃时正在运行的应用程序也要崩溃。当客户机重新启动时，它很可能使用TCP重新安装服务器的文件系统，这将导致和服务器的另一个连接。客户和服务器之间针对同一个文件系统的前一个连接现在打开了一半（服务器方认为它还开着），但是既然服务器设置了keepalive选项，当服务器发出下一个keepalive探查报文时，这个半开着的TCP连接就会被中止。

随着时间的流逝，另外一些厂商也计划支持TCP上的NFS。

29.6 NFS实例

我们使用tcpdump来看一下在典型的文件操作中，客户调用了哪些 NFS过程。当tcpdump检测到一个包含RPC调用（在图29-1中调用字段等于0）、目的端口是2049的UDP数据报时，它把数据报按照一个NFS请求进行解码。类似地，如果一个 UDP数据报是一个RPC应答（在图29-2中应答字段为1），源端口是2049，tcpdump就把此数据报作为一个NFS应答来解码。

29.6.1 简单的例子：读一个文件

第一个例子是使用cat(1)命令将位于一个NFS服务器上的一个文件复制到终端上：

```
sun % cat /nfs/bsdi/usr/rstevens/hello.c        把文件复制到终端
main()
{
    printf("hello, world\n");
}
```

如同图29-6所示，主机sun（NFS客户机）上的文件系统 /nfs/bsdi/usr 实际上是主机bsdi（NFS服务器）上的 /usr 文件系统。当cat打开这个文件时，sun上的内核检测到这一点，然后使用NFS去访问文件。图29-7显示了tcpdump的输出。

当tcpdump解析一个NFS请求或应答报文时，它打印客户的XID字段，而不是端口号。第1行和第2行中的XID字段值是0x7aa6。

客户内核中的打开函数一次处理文件名 /nfs/bsdi/usr/rstevens/hello.c 中的一个成员。当处理到/nfs/bsdi/usr时，它发现这是指向一个已安装的NFS文件系统的一个安装点。

在第1行中，客户调用GETATTR过程取得客户已经安装的服务器目录的属性（ /usr）。这个RPC请求，除IP首部和UDP首部之外，包含104个字节的数据。第2行中的应答返回了一个OK值，除了IP首部和UDP首部之外，包含了96个字节的数据。在这个图中，我们可以看出最小的NFS报文包含大约100个字节的数据。

Linux公社 www.linuxidc.com

```
 1  0.0                  sun.7aa6 > bsdi.nfs: 104 getattr
 2  0.003587 (0.0036)    bsdi.nfs > sun.7aa6: reply ok 96

 3  0.005390 (0.0018)    sun.7aa7 > bsdi.nfs: 116 lookup "rstevens"
 4  0.009570 (0.0042)    bsdi.nfs > sun.7aa7: reply ok 128

 5  0.011413 (0.0018)    sun.7aa8 > bsdi.nfs: 116 lookup "hello.c"
 6  0.015512 (0.0041)    bsdi.nfs > sun.7aa8: reply ok 128

 7  0.018843 (0.0033)    sun.7aa9 > bsdi.nfs: 104 getattr
 8  0.022377 (0.0035)    bsdi.nfs > sun.7aa9: reply ok 96

 9  0.027621 (0.0052)    sun.7aaa > bsdi.nfs: 116 read 1024 bytes @ 0
10  0.032170 (0.0045)    bsdi.nfs > sun.7aaa: reply ok 140
```

图29-7 读一个文件的NFS操作

在第3行中，客户调用LOOKUP过程来查看rstevens文件。在第4行中收到一个OK应答。LOOKUP过程说明了文件名rstevens和远程文件系统被安装时由内核保存的文件句柄。应答中包含了下一步要使用的一个新的文件句柄。

在第5行中，客户使用第4行中返回的文件句柄对hello.c调用LOOKUP过程。在第6行返回了另一个文件句柄。新的文件句柄就是客户在第7行和第9行中引用文件/nfs/bsdi/usr/rstevens/hello.c所使用的文件句柄。我们看到客户对于正在打开的路径名的每个成员都调用了一次LOOKUP过程。

在第7行中，客户又调用了一次GETATTR过程，接着在第9行中调用了READ过程。客户请求从偏移0开始的1024个字节，但是接收到的没有这么多（减去RPC字段和其他由READ过程返回的值的大小，在第10行中返回了38个字节的数据。这是文件hello.c的实际大小）。

在这个例子中，应用进程对于内核所做的这些RPC请求和应答一点儿也不知道。应用进程只是调用了内核的open函数，后者引起了3个RPC请求和3个应答（1~6行），然后应用进程又调用了内核的read函数，它引起了两个请求和两个应答（7~10行）。该文件位于一个NFS文件服务器，这一点对客户应用进程来说是透明的。

29.6.2 简单的例子：创建一个目录

作为另一个简单的例子，我们将当前工作目录改变为一个NFS服务器上的一个目录，然后创建一个新的目录：

```
sun  % cd  /nfs/bsdi/usr/rstevens        改变当前工作目录
sun  % mkdir  Mail                       并且创建一个目录
```

图29-8显示了tcpdump的输出。

```
 1   0.0                  sun.7ad2 > bsdi.nfs: 104 getattr
 2   0.004912 ( 0.0049)   bsdi.nfs > sun.7ad2: reply ok 96

 3   0.007266 ( 0.0024)   sun.7ad3 > bsdi.nfs: 104 getattr
 4   0.010846 ( 0.0036)   bsdi.nfs > sun.7ad3: reply ok 96

 5  35.769875 (35.7590)   sun.7ad4 > bsdi.nfs: 104 getattr
 6  35.773432 ( 0.0036)   bsdi.nfs > sun.7ad4: reply ok 96

 7  35.775236 ( 0.0018)   sun.7ad5 > bsdi.nfs: 112 lookup "Mail"
 8  35.780914 ( 0.0057)   bsdi.nfs > sun.7ad5: reply ok 28

 9  35.782339 ( 0.0014)   sun.7ad6 > bsdi.nfs: 144 mkdir "Mail"
10  35.992354 ( 0.2100)   bsdi.nfs > sun.7ad6: reply ok 128
```

图29-8 NFS的操作：cd到NFS目录，然后mkdir

Linux公社 www.linuxidc.com

改变目录引起客户调用了两次GETATTR过程（1~4行）。当我们创建新的目录时，客户调用了GETATTR过程（5~6行），接着调用LOOKUP过程（7~8行，用来验证将创建的目录不存在），跟着调用了MKDIR过程来创建目录（9-10行）。在第8行中，应答OK并不表示目录存在。它只是表示过程返回了。 tcpdump并不理解NFS过程的返回值。它一般打印OK和应答报文中数据的字节数。

29.6.3 无状态

NFS的一个特征（NFS的批评者称之为NFS的一个瑕疵，而不是一个特征）是NFS服务器是无状态的(stateless)。服务器并不记录哪个客户正在访问哪个文件。请注意一下在前面给出的NFS过程中，没有一个open操作和一个close操作。LOOKUP过程的功能与open操作有些类似，但是服务器永远也不会知道客户对一个文件调用了LOOKUP过程之后是否会引用该文件。

无状态设计的理由是为了在服务器崩溃并且重启动时，简化服务器的崩溃恢复操作。

29.6.4 例子：服务器崩溃

在下面的例子中我们从一个崩溃然后重启动的NFS服务器上读一个文件。这个例子演示了无状态的服务器是如何使得客户不知道服务器的崩溃。除了在服务器崩溃然后重启动时一个时间上的暂停外，客户并不知道发生的问题，客户应用进程没有受到影响。

在客户机sun上，我们对一个长文件（NFS服务器主机svr4上的文件/usr/share/lib/termcap）执行cat命令。在传送过程中把以太网的网线拔掉，关闭然后重启动服务器主机，再重新将网线连上。客户被配置成每个NFS read过程读1024个字节。图29-9显示了tcpdump的输出。

1~10行对应于客户打开文件，操作类似于图29-7所示。在第11行我们看到对文件的第一个READ操作，在12行返回了1024个字节的数据。这个操作一直继续到129行（读1024个字节的数据，跟着一个OK应答）。

在第130行和第131行我们看到两个请求超时，并且分别在132行和133行重传。第一个问题是这里为什么会有两个读请求，一个从偏移65536开始读，另一个从偏移73728开始读？答案是客户内核检测到客户应用进程正在进行顺序地读操作，所以试图预先取得数据块（大多数的Unix内核都采用了这种预读技术）。客户内核也正在运行多个NFS块I/O守护程序，后者试图代表客户产生多个RPC请求。一个守护程序正在从偏移65536处读8192个字节（以1024字节为一组数据块），而另一个正在从73728处预读8192个字节。

客户重传发生在130~168行。在第169行我们看到服务器已经重启动，在它对第168行的客户NFS请求做出应答之前，它发送了一个ARP请求。对168行的响应被发送在171行。客户的READ操作继续进行下去。

除了从129行到171行5分钟的暂停，客户应用进程并不知道服务器崩溃然后又重启动了。这个服务器的崩溃对于客户是透明的。

为了研究这个例子中的超时和重传时间间隔，首先要意识到这儿有两个客户守护程序，分别有它们各自的超时。第1个守护程序（在偏移65536处开始读）的间隔，四舍五入到两个十进制小数点，为0.68, 0.87, 1.74, 3.48, 6.96, 13.92, 20.0, 20.0, 20.0等等。第2个守护程序（在偏移73728处开始读）的间隔也是一样的（精确到两个小数点）。可以看出这些NFS客户使用了一个这样的超时定时器：间隔为0.875秒的倍数，上限为20秒。每次超时后，重传间隔翻倍：

Linux公社 www.linuxidc.com

0.875, 1.75, 3.5, 7.0和14.0。

```
 1   0.0                 sun.7ade > svr4.nfs: 104 getattr
 2   0.007653 ( 0.0077)  svr4.nfs > sun.7ade: reply ok 96

 3   0.009041 ( 0.0014)  sun.7adf > svr4.nfs: 116 lookup "share"
 4   0.017237 ( 0.0082)  svr4.nfs > sun.7adf: reply ok 128

 5   0.018518 ( 0.0013)  sun.7ae0 > svr4.nfs: 112 lookup "lib"
 6   0.026802 ( 0.0083)  svr4.nfs > sun.7ae0: reply ok 128

 7   0.028096 ( 0.0013)  sun.7ae1 > svr4.nfs: 116 lookup "termcap"
 8   0.036434 ( 0.0083)  svr4.nfs > sun.7ae1: reply ok 128

 9   0.038060 ( 0.0016)  sun.7ae2 > svr4.nfs: 104 getattr
10   0.045821 ( 0.0078)  svr4.nfs > sun.7ae2: reply ok 96

11   0.050984 ( 0.0052)  sun.7ae3 > svr4.nfs: 116 read 1024 bytes @ 0
12   0.084995 ( 0.0340)  svr4.nfs > sun.7ae3: reply ok 1124
```

连续地读

```
128   3.430313 ( 0.0013)  sun.7b22 > svr4.nfs: 116 read 1024 bytes @ 64512
129   3.441828 ( 0.0115)  svr4.nfs > sun.7b22: reply ok 1124

130   4.125031 ( 0.6832)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
131   4.868593 ( 0.7436)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

132   4.993021 ( 0.1244)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
133   5.732217 ( 0.7392)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

134   6.732084 ( 0.9999)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
135   7.472098 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

136  10.211964 ( 2.7399)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
137  10.951960 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

138  17.171767 ( 6.2198)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
139  17.911762 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

140  31.092136 (13.1804)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
141  31.831432 ( 0.7393)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

142  51.090854 (19.2594)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
143  51.830939 ( 0.7401)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

144  71.090305 (19.2594)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
145  71.830155 ( 0.7398)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
```

连续重传

```
167 291.824285 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
168 311.083676 (19.2594)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
```

服务器重启动

```
169 311.149476 ( 0.0658)  arp who-has sun tell svr4
170 311.150004 ( 0.0005)  arp reply sun is-at 8:0:20:3:f6:42

171 311.154852 ( 0.0048)  svr4.nfs > sun.7b23: reply ok 1124

172 311.156671 ( 0.0018)  sun.7b25 > svr4.nfs: 116 read 1024 bytes @ 66560
173 311.168926 ( 0.0123)  svr4.nfs > sun.7b25: reply ok 1124
```

连续读

图29-9 当一个NFS服务器崩溃然后重启动时，客户正在读一个文件的过程

客户要重传多久呢？客户有两个与此有关的选项。首先，如果服务器文件系统是“硬”安装的，客户就会永远重传下去。但是如果服务器文件系统是“软”安装的，客户重传了固定数目的次数之后就会放弃。在“硬”安装的情况下，客户还有一个选项决定是否允许用户中断无限制的重传。如果客户主机安装服务器文件系统时说明了中断能力，并且如果我们不想在服务器崩溃之后等 5分钟，等着服务器重启动，就可以键入一个中断键以终止客户应用

Linux公社 www.linuxidc.com

程序。

29.6.5 等幂过程

如果一个RPC过程被服务器执行多次仍然返回同样的结果，那么就把它叫作等幂过程（Idempotent Procedure）。例如，NFS的读过程是等幂的。正像我们在图29-9中看到的，客户只是重发一个特定的READ调用直到它得到一个响应。在我们的例子中，重传的原因是服务器崩溃了。如果服务器没有崩溃，而是RPC应答报文丢失了（既然UDP是不可靠的），客户只是重传请求，服务器再一次执行同样的READ过程。同一个文件的同一部分被重读一次，发送给客户。

这种方法行得通的原因在于每个READ请求指出了读操作开始的偏移位置。如果有一个NFS过程要求服务器读一个文件的下*N*个字节，这种方法就不行了。除非服务器被做成是有状态的（与无状态相反），如果一个应答丢失了，客户重发读下*N*个字节的READ请求，结果将是不一样的。这就是为什么NFS的READ和WRITE过程要求客户说明开始的偏移位置的原因。客户维护着状态（每个文件当前的偏移位置），而不是服务器。

不幸的是并不是所有的文件系统操作都是等幂的。例如，考虑下面的动作：客户NFS发出REMOVE请求来删除一个文件；服务器NFS删除了文件，并回答OK；服务器的回答丢失了；客户NFS超时，然后重传请求；服务器NFS找不到指定的文件，回答指出一个错误；客户应用程序接收到一个错误表示文件不存在。这个返回给客户应用程序的错误是不对的——该文件的确存在并且被删除了。

等幂的NFS过程是：GETATTR、STATES、LOOKUP、READ、WRITE、READLINK和READDIR。不是等幂的过程是：CREATE、REMOVE、RENAME、LINK、SYMLINK、MKDIR和RMDIR。SETATTR过程如果不用来截断文件，一般是等幂的。

既然使用UDP总会发生响应报文丢失的现象，NFS服务器需要一种方法来处理非等幂的操作。大多数的服务器实现了一个最近应答的高速缓存，用于存放非等幂操作最近的应答。每当服务器收到一个请求，它首先检查这个高速缓存，如果找到了一个匹配，就返回以前的应答而不再调用相应的NFS过程。[Juszczak 1989]提供了这种高速缓存的实现细节。

等幂服务器过程的概念可以应用于任何基于UDP的应用程序，而不仅仅是NFS。例如，DNS也提供了一个等幂服务。一个DNS的服务器可以任意多次地执行一个解析者的请求而没有任何不良的后果（如果不考虑网络资源浪费的话）。

29.7 第3版的NFS

1993年发布了第3版的NFS协议规范[Sun Microsystem 1994]。其实现有望在1994年成为可能。

我们总结一下第2版和第3版的主要区别。下面把两者分别称为V2和V3。

1) V2中的文件句柄是32字节的固定大小的数组。在V3中，它变成了一个最多为64个字节的可变长度的数组。在XDR中，一个可变长度的数组被编码为一个4字节的数组成员个数跟着实际的数组成员字节。这样在实现时减少了文件句柄的长度，例如Unix只需要12个字节，但又允许非Unix实现维护另外的信息。

2) V2将每个READ和WRITE RPC过程可以读写的数据限制为8192个字节。这个限制在V3

Linux公社 www.linuxidc.com

中取消了，这就意味着一个 UDP上的实现只受到IP数据报大小的限制（65535字节）。这样允许在更快的网络上读写更大的分组。

3) 文件大小以及READ和WRITE过程开始偏移的字节从32字节扩充到64字节，允许读写更大的文件。

4) 每个影响文件属性值的调用都返回文件的属性。这样减少了客户调用 GETATTR过程的次数。

5) WRITE过程可以是异步的，而在V2中要求同步的WRITE过程。这样可以提高WRITE过程的性能。

6) V3中删去了一个过程（STATFS），增加了七个过程：ACCESS（检查文件访问权限）、MKNOD（创建一个Unix特殊文件）、READDIRPLUS（返回一个目录中的文件名字和它们的属性）、FSINFO（返回一个文件系统的静态信息）、FSSTAT（返回一个文件系统的动态信息）、PATHCONF（返回一个文件的POSIX.1信息）和COMMIT（将以前的异步写操作提交到外存中）。

29.8 小结

RPC是构造客户-服务器应用程序的一种方式，使得看起来客户只是调用了服务器的过程。所有的网络操作细节都被隐藏在 RPC程序包为一个应用程序生成的客户和服务器残桩以及RPC库的例程中。我们显示了 RPC调用和应答报文的格式，并且提到了使用 XDR对传输的值进行编码，使得RPC客户和服务器可以运行在不同架构的机器上。

最广泛使用的RPC应用之一就是Sun的NFS，一个在各种大小的主机上广泛实现的异构的文件访问协议。我们浏览了 NFS和它使用UDP和TCP的方式。第2版的NFS协议定义了15个过程。

一个客户对一个NFS服务器的访问开始于安装协议，返回给客户一个文件句柄。客户接着可以使用那个文件句柄来访问服务器文件系统中的文件。在服务器上，一次检查文件名的一个成员，返回每个成员的一个新的文件句柄。最后的结果就是要引用的文件的一个文件句柄，它可以在随后的读写操作中被使用。

NFS试图把它的所用过程都做成等幂的，使得如果响应报文丢失了，客户只需要重发一个请求。我们看到了服务器崩溃然后又重启动时，一个客户读服务器上的一个文件的例子。

习题

29.1 在图29-7中，我们看到tcpdump将分组理解为NFS的请求和应答，打印了 XID。tcpdump可以为任何的RPC请求或者应答这样做吗？

29.2 在一个Unix系统中，你认为为什么 RPC服务器程序使用的是临时端口，而不是知名端口？

29.3 一个RPC客户调用了两个服务器过程。第1个服务器过程执行花了5秒钟的时间，第二个过程花了1秒钟。客户有一个4秒钟的超时。画出客户与服务器之间在时间轴上交互的信息（假定信息从客户传到服务器或者相反都不花时间）。

29.4 在图29-9的例子中，如果NFS服务器关机时，把它的以太网卡给换掉了，将会发生什么事情？

Linux公社 www.linuxidc.com

29.5 在图29-9中，当服务器重启动后，它处理了从偏移 65536开始的请求（168行和171行），然后处理了从偏移66560开始的下一个请求（172行和173行）。对于从偏移73728开始的请求怎么处理的呢？（167行）

29.6 当描述等幂NFS过程时，我们给出了一个REMOVE应答在网络中丢失的例子。在这种情况下，如果使用的是TCP而不是UDP会怎么样呢？

29.7 如果NFS服务器使用的是一个临时端口而不是 2049，那么当服务器崩溃然后又重启动时，一个NFS客户会发生什么情况呢？

29.8 每个主机最多只有1023个保留端口，所以保留端口是很缺乏的（ 1.9节）。如果一个NFS服务器要求它的客户拥有保留端口（公共的端口），一个NFS客户使用TCP安装了*N*个不同的服务器上的*N*个文件系统，那么客户对每个连接都需要一个不同的保留端口号吗？

Linux公社 www.linuxidc.com

第30章　其他的TCP/IP应用程序

30.1　引言

本章中我们描述了另外一些很多实现都支持的 TCP/IP应用程序。有些很简单，易于全面了解（Finger和Whois），而另一个则相当复杂（ X窗口系统）。我们只提供了这个复杂应用程序的一个简短的概述，集中介绍其对 TCP/IP协议的使用。

另外，我们提供一些 Internet上资源发现工具的概述。包括一组在 Internet上导航的工具，可以帮助寻找一些我们不知道确切位置和名字的信息。

30.2　Finger协议

Finger协议返回一个指定主机上一个或多个用户的信息。它常被用来检查某个人是否登录了，或者搞清一个人的登录名以便给他发送邮件。 RFC1288 [Zimmerman 1991] 指明了这个协议。

由于两个原因，很多站点不支持一个 Finger服务器。第一， Finger服务器的一个早期版本中的一个编程错误被 1988年声名狼藉的 Internet蠕虫病毒利用，作为进入点之一（ RFC1135 [Reynolds 1989] 和 [Curry 1992] 更详细地描述了蠕虫）。第二， Finger协议有可能会泄露一些很多管理员认为是有关用户的私有信息（登录名、电话号码，他们上次的登录时间，等等）。RFC1288的第3节给出了这个有关服务安全方面的细节。

从一个协议的角度来看， Finger服务器有一个知名的端口 79。客户对这个端口做一个主动打开，然后发送一个在线的请求。服务器处理这个请求，把输出发送回去，然后关闭连接。查询和响应都是采用NVT ASCII，类似于我们在FTP和SMTP协议中所看到的。

尽管大多数的Unix用户都是使用finger (1)客户来访问Finger服务器，我们将从使用 Telnet客户与Finger服务器直接相连开始，看看客户发出的每一条在线命令。如果客户的查询是一个空行（在NVT ASCII中，空行以一个回车符 CR跟着一个换行符 LF来传输），它就是一个请求查询所有在线用户信息的命令。

```
sun % telnet slip finger
Trying 140.252.13.65 ...
Connected to slip.
Escape character is '^]'.

Login    Name                Tty  Idle  Login Time   Office    Office Phone
rstevens Richard Stevens     *co    45  Jul 31 09:13
rstevens Richard Stevens     *c2    45  Aug  5 09:41
Connection closed by foreign host.
```

Telnet客户输出前三行

这儿我们键入回车作为Finger客户的命令

Telnet客户的输出

office和office phone的空白输出字段是从用户的口令 (password)文件记录的选项字段中取出的（在这个例子中，这两个字段的值没有提供）。

服务器必须在最后做一个主动的关闭操作，因为服务器返回的是一个可变长度的信息。

Linux公社 www.linuxidc.com

当客户收到文件结束字符时，就知道服务器的输出结束了。

当客户的请求由一个用户名组成时，服务器只以该用户的信息作为响应。下面是另一个例子，这个例子中删去了Telnet客户的输出：

```
sun % telnet vangogh.cs.berkeley.edu finger
rstevens                              这是我们键入的客户请求
Login: rstevens                       Name: Richard Stevens
Directory: /a/guest/rstevens          Shell: /bin/csh
Last login Thu Aug  5 09:55 (PDT) on ttyq2 from sun.tuc.noao.edu
Mail forwarded to: rstevens@noao.edu
No Plan.
```

当一个系统完全禁止了Finger服务时，因为没有进程被动打开端口79，所以客户的主动打开将从服务器接收到一个RST。

```
sun % finger @svr4
[svr4.tuc.noao.edu] connect: Connection refused
```

一些站点在端口79提供了一个服务器，但服务器只是向客户输出信息，而不理睬客户的任何请求：

```
sun % finger @att.com
[att.com]                这一行是Finger客户输出的；其余行是服务器输出的
-----------------------------------------------------------
There are no user accounts on the AT&T Internet gateway.
To send email to an AT&T employee, send email to their name
separated by periods at att.com. If the employee has an email
address registered in the employee database, they will receive
email - otherwise, you'll receive a non-delivery notice.
For example: John.Q.Public@att.com
sun % finger clinton@whitehouse.gov
[whitehouse.gov]

        Finger service for arbitrary addresses on whitehouse.gov is not
supported. If you wish to send electronic mail, valid addresses are
"PRESIDENT@WHITEHOUSE.GOV", and "VICE-PRESIDENT@WHITEHOUSE.GOV".
```

对一个组织来说，另一种可能就是实现一个防火墙网关：在组织内部和 Internet之间的一个路由器，负责过滤（也就是扔掉）特定的 IP数据报（[Cheswick and Bellovin 1994] 详细讨论了防火墙网关）。防火墙网关可以被配置成扔掉从Internet进来的这样一些数据报，这些数据报是目的端口为79的TCP报文段。

对于Finger的服务器和Unix的Finger客户还有其他的实现。欲知详情，请参考 RFC1288和有关finger(1)的手册。

> RFC1288指出提供了Finger服务器的、具有TCP/IP连接的自动售货机应该对客户的空行请求响应以现有产品的列表。对于由一个名字组成的客户请求，它们应该响应以一个数目或者与这个产品有关的可用项的列表。

30.3 Whois协议

Whois协议是另一种信息服务。尽管任何站点都可以提供一个 Whois服务器，在InterNIC站点（rs.internic.net）的服务器是最常使用的。这个服务器维护着所有的 DNS域和很多连接在Internet上的系统的系统管理员的信息（另一个可用的服务器在 nic.ddn.mil，不过只包含了有关MILNET的信息）。不幸的是信息有可能是过期的或不完整的。 RFC954 [Harrenstein, Stahl,

Linux公社 www.linuxidc.com

and Feinler 1985] 说明了Whois服务。

从协议的角度来看，Whois服务器有一个知名的TCP端口43。它接受客户的连接请求，客户向服务器发送一个在线的查询。服务器响应以任何可用的信息，然后关闭连接。请求和应答都以NVT ASCII来传输。除了请求和应答所包含的信息不一样，Whois服务器和Finger服务器几乎是一样的。

最常用的Unix客户程序是whois(1) 程序，尽管我们可以使用Telent自己手工键入命令。开始的命令是只包含一个问号的请求，服务器会返回所支持的客户请求的具体信息。

> 当NIC在1993年改变为InterNIC时，Whois服务器的站点也从nic.ddn.mil移到了rs.internic.net。很多厂商仍然装载了采用nic.ddn.mil版本的whois客户程序。为了和正确的服务器联系上，你可能需要指明命令行参数 -h rs.internic.net。
>
> 另外，我们可以使用Telnet登录rs.internic.net站点，登录名采用whois。

我们将使用Whois服务器来查询一下本书的作者（已经删去了无关的 Telnet客户输出）。第一个请求是查询所有匹配“ stevens ”的名字。

```
sun % telnet rs.internic.net whois
stevens                                  这是我们键入的客户命令
                                         我们省略了其他25个“ stevens ”的信息
Stevens, W. Richard (WRS28)    stevens@kohala.com      +1 602 297 9416

The InterNIC Registration Services Host ONLY contains Internet
Information (Networks, ASN's, Domains, and POC's).
Please use the whois server at nic.ddn.mil for MILNET Information.
```

名字后面的括号中的三个大写字母跟着一个数字，（WRS28），是个人的NI句柄。下一个查询包含一个感叹号和一个NIC句柄，用于获得有关这个人的进一步信息。

```
sun % telnet rs.internic.net whois
!wrs28                                   我们键入的客户请求
Stevens, W. Richard (WRS28)                 stevens@kohala.com
   Kohala Software
   1202 E. Paseo del Zorro
   Tucson, AZ 85718
   +1 602 297 9416

   Record last updated on 11-Jan-91.
```

很多有关Internet变量的其他信息也可以查找。例如，请求 net 140.252将返回有关B类地址140.252的信息。

白页

使用SMTP的VRFY命令、Finger协议以及Whois协议在Internet上查找用户类似于使用电话号码簿的白页查找一个人的电话号码。在目前阶段，诸如上述的工具已经广泛可用了，为了提高这种服务的研究正在进行当中。

[Schwartz and Tsirigotis 1991] 包含了正在Internet上试验的不同白页服务的其他信息。一个叫作Netfind的特别工具可以通过使用 Telent，以netfind登录到bruno.cs.colorado或者ds.internic.net站点来访问。

RFC1309 [Weider, Reynolds, and Heker 1992]提供了对OSI目录服务X.500的概述，并且比较了它与当前的Internet技术（Finger和Whois）的相同点和不同点。

Linux公社 www.linuxidc.com

30.4 Archie、WAIS、Gopher、Veronica和WWW

前两节我们讨论的工具 —— Finger、Whois和一个白页服务 —— 是用来查找人的信息的。还有一些工具是用来定位文件和文档的，本节中对这些工具给出了一个概述。我们只提供了一个概述，因为对每一个工具的细节的研究超出了本书的范围。我们给出了在 Internet上找到这些工具的方法，鼓励你去试一试，找找看哪些工具可以帮助你。还有一些其他的工具正在被开发。[Obraczka, Danzig, and Li1993] 概述了在Internet上的资源发现服务。

30.4.1 Archie

本书中使用的很多资源都是使用匿名 FTP得到的。问题是如何找到有我们想要的程序的FTP站点。有时候我们甚至不知道精确的文件名，但知道几个很可能在文件名中出现的关键字。

Archie提供了Internet上几千个FTP服务器的目录。我们可以通过登录进一个Archie服务器，搜索那些名字中包含了一个指定的常规表达式的文件。输出是一个与文件名匹配的 FTP服务器的列表。然后我们可以使用匿名FTP去那个站点取得想要的文件。

全世界有很多 Archie服务器。一个比较好的开始点是使用 Telnet以archie名字登录进ds.internic.net，然后执行命令servers。这个命令的输出提供了所有Archie服务器以及它们的地址的一个列表。

30.4.2 WAIS

Archie帮助我们查找名字中包含关键字的文件，但有时候我们需要查找包含一个关键字的文件或数据库。即，想查找一个内容中包含一个关键字的文件，而不是文件名字中包含关键字。

WAIS (Wide Area Information Servers广域信息服务系统)知道几百个包含了有关计算机主题的和其他一般性主题信息的数据库。为了使用 WAIS，我们要选择需要查找的数据库，指明关键字。尝试WAIS服务请使用Telnet，以wais名字登录quake.think.com站点。

30.4.3 Gopher

Gopher是其他Internet资源服务如Archie、WAIS和匿名FTP的一个菜单驱动的前端程序。Gopher是最容易使用的工具之一，因为不管它使用了哪个资源服务，它的用户界面都是一样的。

为了尝试Gopher，请使用Telnet，以gopher名字登录is.internic.net站点。

30.4.4 Veronica

就像Archie是一个匿名FTP服务器的索引一样， Veronica(Very Easy Rodent-Oriented Netwide Index to Computerized Archives)是一个Gopher标题的索引。一次Veronica搜索一般要查找几百个Gopher服务器。

我们必须通过一个 Gopher客户来访问 Veronica服务。选择 Gopher的菜单项 “ Beyond InterNIC: Virtual Treasures of the Internet ”，然后在下一个菜单中选择 Veronica。

Linux公社 www.linuxidc.com

30.4.5 万维网WWW

万维网使用一个称为超文本的工具，使得我们可以浏览一个大的 /全球范围的服务和文档。信息和关键字一起显示，不过关键字被突出显示[⊖]。我们可以通过选择关键字得到更多的信息。

为了访问WWW，请使用Telnet登录info.cern.ch站点。

30.5 X窗口系统

X窗口系统(X Window System)，或简称为X，是一种客户-服务器应用程序。它可以使得多个客户（应用）使用由一个服务器管理的位映射显示器。服务器是一个软件，用来管理显示器、键盘和鼠标。客户是一个应用程序，它与服务器在同一台主机上或者在不同的主机上。在后一种情况下，客户与服务器之间通信的通用形式是 TCP，尽管也可以使用诸如 DECNET 的其他协议。在有些场合，服务器是与其他主机上客户通信的一个专门的硬件（一个 X终端）。在另一种场合，一个独立的工作站，客户与服务器位于同一台主机，使用那台主机上的进程间通信机制进行通信，而根本不涉及任何网络操作。在这两种极端情况之间，是一台既支持同一台主机上的客户又支持不同主机上的客户的工作站。

X需要一个诸如TCP的、可靠的、双向的流协议（X不是为不可靠协议，如 UDP，而设计的）。客户与服务器的通信是由在连接上交换的 8 bit字节组成的。[Nye 1992] 给出了客户与服务器在它们的TCP连接上交换的150多个报文的格式。

在一个Unix系统中，当X客户和X服务器在同一台主机上时，一般使用 Unix系统的本地协议，而不使用 TCP协议，因为这样比使用 TCP的情况减少了协议处理时间。 Unix系统的本地协议是同一台主机上的客户和服务器之间可以使用的一种进程间通信的形式。回忆一下在图2-4中，当使用TCP作为同一台主机上进程间的通信方式时，在 IP层以下发生了这个数据的环回(loopback)，隐含着所有的TCP和IP处理都发生了。

图30-1显示了三个客户使用一个显示器的可能的脚本。一个客户与服务器在同一台主机上，使用Unix系统的本地协议。另外两个位于不同的主机上，使用 TCP。一般来说，其中一个客户是一个窗口管理程序 (window manager)，它有权限管理显示器上窗口的布局。例如，窗口管理程序允许我们在屏幕上移动窗口，或者改变窗口的大小。

在这里客户和服务器这两个词猛一看含义相反了。对于 Telnet和FTP的应用，我们把客户看作是在键盘和显示器上的交互式用户。但是对于 X，键盘和显示器是属于服务器的。服务器被认为是提供服务的一方。 X提供的服务是对窗口、键盘和鼠标的访问。对于 Telnet，服务是登录远程的主机。对于FTP，服务是服务器上的文件系统。

当X终端或工作站引导时，一般启动 X服务器。服务器创建一个 TCP端点，在端口6000 + *n*上做一个被动打开，其中 *n*是显示器号（一般是 0）。大多数的 Unix服务器也使用名字 /tmp/.X11-unix/X*n*创建一个Unix系统的插口，其中*n*还是显示器的号。

当一个客户在另一台主机上启动时，它创建一个 TCP端点，对服务器上的端口 6000+*n*做一个主动打开。每个客户都得到了一个自己与服务器的连接。服务器负责对所有的客户请求进行复用。从这点开始，客户通过 TCP连接向服务器发送请求（例如，创建一个窗口），服务

⊖ 例如通过使用不同的颜色 —— 译者注。

Linux公社 www.linuxidc.com

器返回应答，服务器也发送事件给客户（鼠标按钮按下，键盘键按下，窗口暴露，窗口大小改变，等等）。

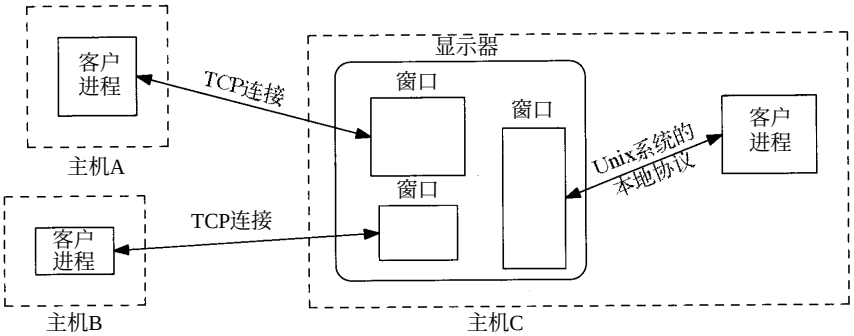


图30-1 使用一个显示器的三个X客户

图30-2将图30-1重新画，但强调了客户与X服务器进程间的通信，X服务器进程轮流管理着每个窗口。图中没有显示的是X服务器管理键盘和鼠标。

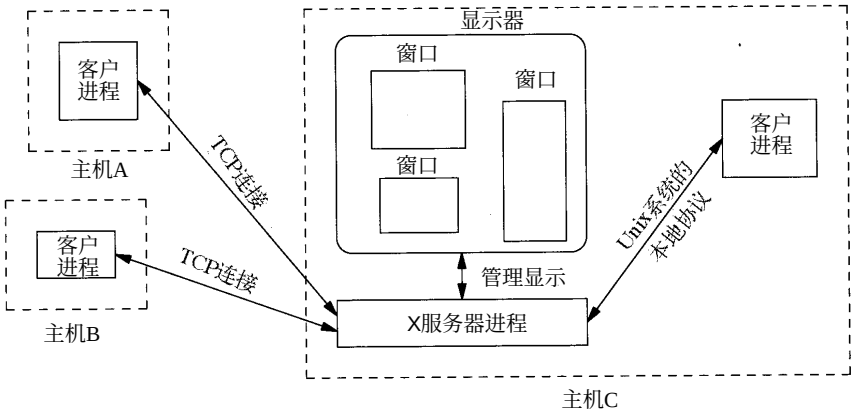


图30-2 使用一个显示器的三个客户

单个服务器处理多个客户请求的这种设计与我们在18.11节描述的正常的TCP并发服务器设计不同。例如，每次一个新的TCP连接请求到达，FTP和Telnet服务器都会产生一个新的进程，因此，每个客户都和一个不同的服务器进程通信。然而，对于X，运行在同一台主机或者在不同主机上的所有客户都和同一个服务器通信。

通过X客户和它的服务器之间的TCP连接可以交换很多数据。传输数据的数目依赖于特定的应用程序设计。例如，如果我们运行Xclock客户，Xclock在服务器的一个窗口中显示客户机当前的时间和日期。如果我们指定每隔1秒修改一次时间，那么每隔1秒，就会有一个X报文通过TCP连接从客户传输到服务器。如果我们运行X终端模拟程序，Xterm，我们敲的每一个键都会变成一个32字节的X报文（加上标准的IP和TCP首部就是72字节），在相反方向上的回送字符将是一个更大的X报文。[Droms and Dyksen 1990]检查了不同的X客户与一个特定的服务器之间的TCP流量。

30.5.1 Xscope程序

Xscope是检查X客户与它的服务器之间交换的信息的一个方便的程序。大多数的X窗口实

Linux公社 www.linuxidc.com

现都提供这个程序。它处在客户与服务器之间，双向传输所有的数据，同时解析所有的客户请求和服务器应答。图30-3显示了这种设置。

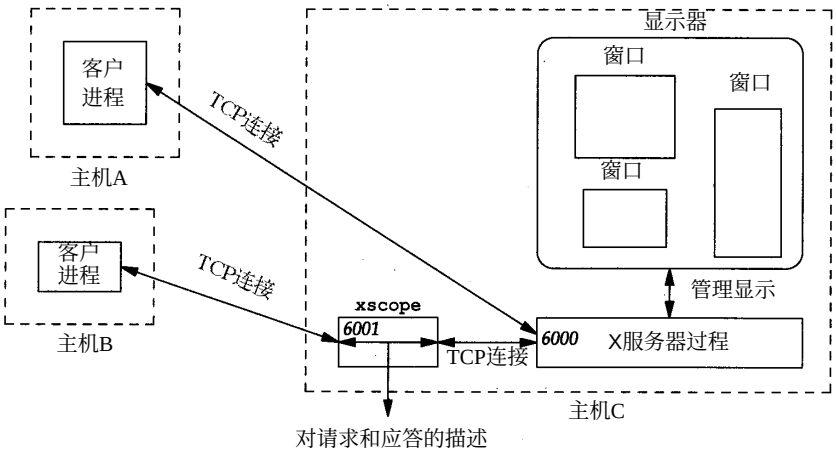


图30-3 使用xscope 监视一个X连接

首先，我们在服务器所在的主机上启动xscope进程，但是xscope不是在端口6000而是在端口6001上监听TCP的连接请求。然后我们在另一台主机上启动一个客户，指明显示器号为1，而不是0，使得客户与xscope相连，而不直接与服务器相连。当客户的连接请求到达时，xscope创建与端口6000上的真正的服务器的一个TCP连接，在客户与服务器之间复制所有的数据，同时生成请求与应答的一个可读的描述。

我们将在sun主机上启动xscope，然后在主机svr4上运行xclock客户。

```
svr4 % DISPLAY=sun:1 xclock -digital -update 5
```

这条命令在主机sun的一个窗口中以数字形式显示时间和日期。我们指明了一个每 5秒的更新时间。

```
Thu Sep 9 10:32:55 1993
```

我们对xscope指明一个-q选项以产生最小的输出。为了看到每个报文的所有字段，可以使用不同的冗长级别。下面的输出显示了前三个请求和应答。

```
sun % xscope -q
0.00: Client --> 12 bytes
0.02:                                152 bytes <-- X11 Server
0.03: Client --> 48 bytes
        ............REQUEST: CreateGC
        ............REQUEST: GetProperty
0.20:                                396 bytes <-- X11 Server
                                     ..............REPLY: GetProperty
0.30: Client -->  8 bytes
0.38: Client --> 20 bytes
        ............REQUEST: InternAtom
0.43:                                 32 bytes <-- X11 Server
                                     ..............REPLY: InternAtom
```

客户的第1个在时刻0.00的报文和服务器在时刻0.02的响应是客户与服务器之间标准的连接建立过程。客户标识它的字节顺序以及它希望的服务器版本。服务器响应以有关自己的不

Linux公社 www.linuxidc.com

同的信息。

下一个在时刻0.03的报文包含了两个客户请求。第1个请求在服务器上创建一个客户可以在其中画的图形上下文。第2个请求从服务器上得到一个属性（RESOURCE_MANAGER属性）。属性可以用于客户之间的通信，经常是在一个应用程序和窗口管理程序之间。服务器在时刻0.20的应答包含了这个属性。

下面两个在时刻0.30和0.38的客户报文形成了返回一个原子的单个请求（每个属性具有一个唯一的整型标识符称为原子）。服务器在时刻0.43的应答包含了这个原子。

如果不提供有关X窗口系统更多的细节是不可能进一步理解这个例子的，但这又不是本节的目的。在这个例子中，在窗口被显示之前，客户总共发送了1668个字节组成的12个报文段，服务器总共发送了1120个字节组成的10个报文段。耗费的时间为3.17秒。从这以后客户每5秒发送一个平均44个字节的小请求，请求更新窗口。这样一直持续到客户被终止。

30.5.2 LBX：低带宽X

为了将X用于局域网，对X协议使用的编码进行了优化，因为在局域网中花在对数据进行编码和解码的时间比最小化传输的数据更重要。尽管这种推断对以太网是适用的，但对于低速的串行线，如SLIP和PPP链路，就存在问题了（2.4节和2.6节）。

定义一个称为低带宽X（LBX）的标准的工作正在进行当中，它使用了下面的技术来减少网络流量的数目：快速缓存、只发送与前面分组的不同部分以及压缩技术。标准的规范和在第6版的X窗口系统中的一个样本实现应该会在1994年的早些时候完成。

30.6 小结

我们介绍的前两个应用，Finger和Whois，是用来获得用户信息的。Finger客户查询一个服务器，经常是为了找到某个人的登录名（以便给他们发电子邮件），或者去看一下某个人是否登录了。Whois客户一般与InterNIC运行的服务器联系，查找关于一个人、机构、域或网络号的信息。

我们简单描述了其他一些Internet资源发现服务：Archie、WAIS、Gopher、Veronica和WWW，帮助我们在Internet上定位文件和文档。还有一些资源发现工具正在被开发。

本章的最后简单浏览了另一个TCP/IP的重要客户程序，X窗口系统。我们看到X服务器管理一个显示器上的多个窗口，处理客户与其窗口的通信。每个客户都有它自己的与服务器的TCP连接，一个单个的服务器为一个给定的显示器管理着所有的客户。通过Xscope程序，我们看到怎样把一个程序放在一个客户与服务器之间，输出有关两者之间交换的报文的信息。

习题

30.1 试用Whois找到网络号为88的A类网络的拥有者。

30.2 试用Whois找到管理`whitehouse.gov`域的DNS服务器。这个应答与DNS给出的答案相匹配吗？

30.3 在图30-3中，你认为`xscope`进程必须和X服务器运行在同一台主机上吗？

Linux公社 www.linuxidc.com

附录A　tcpdump程序

tcpdump程序是由Van Jacobson、Craig Leres和Steven McCanne编写的，他们都来自加利福尼亚大学伯克利分校的劳伦斯伯克利实验室。本书中使用的是 2.2.1版（1992年6月）。

tcpdump通过将网络接口卡设置为混杂模式（ promiscuous mode）来截获经过网络接口的每一个分组。正常情况下，用于诸如以太网媒体的接口卡只截获送往特定接口地址或广播地址的链路层的帧（2.2节）。

底层的操作系统必须允许将一个接口设置成混杂模式，并且允许一个用户进程截获帧。下列的操作系统可以支持 tcpdump，或者可以加入对 tcpdump的支持：4.4BSD、BSD/386、SunOS、Ultrix和HP-UX。参考一下随着 tcpdump发布的README文件，了解它支持哪些操作系统以及哪些版本。

除了tcpdump还有其他一些选择。在图10-8中，我们使用了Solaris 2.2的程序snoop来查看一些分组。AIX 3.2.2提供了iptrace程序，该程序也提供了类似的功能。

A.1　BSD 分组过滤器

当前由BSD演变而来的 Unix内核提供了 BSD 分组过滤器 BPF (BSD Packet Filter)，tcpdump用它来截获和过滤来自一个被置为混杂模式的网络接口卡的分组。 BPF也可以工作在点对点的链路上，如 SLIP（2.4节），不需要什么特别的处理就可以截获所有通过接口的分组。BPF还可以工作在环回接口上（2.7节）。

> BPF有一个很长的历史。1980年卡耐基梅隆大学的Mike Accetta和Rick Rashid创造了Enet分组过滤程序。斯坦福的Jeffrey Mogul将代码移植到BSD，从1983年开始继续开发。从那以后，它演变为DEC的Ultrix分组过滤器、SunOS 4.1下的一个STREAMS NIT模块和BPF。劳伦斯伯克利实验室的Steven McCanne在1990年的夏天实现了BPF。其中很多设计来自于Van Jacobson。[McCanne and Jacobson 1993] 给出了最新版本的细节以及与Sun的NIT的一个比较。

图A-1显示了用于以太网的BPF的特征。

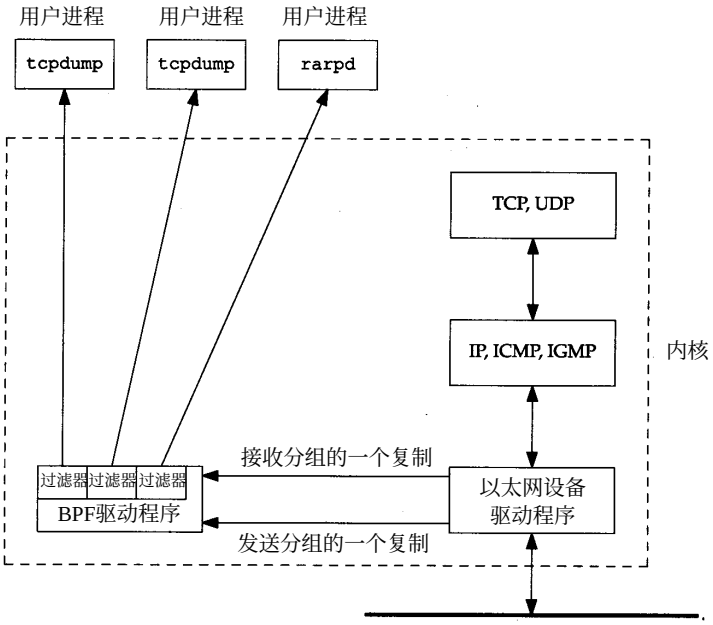
BPF将以太网设备驱动程序设置为混杂模式，然后从驱动程序那里接收每一个收到的分组和传输的分组。这些分组要通过一个用户指明的过滤器，使得只有那些用户进程感兴趣的分组才会传递给用户进程。

多个进程可以同时监视一个接口，每个进程指明了一个自己的过滤器。图　A-1显示了tcpdump的两个实例进程和一个RARP守护进程（5.4节）监视同样的以太网接口。 tcpdump的每个实例指明了一个自己的过滤器。　tcpdump的过滤器可以由用户在命令行指明，而rarpd总是使用只截获RARP请求的过滤器。

除了指明一个过滤器，BPF的每个用户还指明了一个超时定时器的值。因为网络的数据传输率可以很容易地超过 CPU的处理能力，而且一个用户进程从内核中只读小块数据的代价昂

Linux公社 www.linuxidc.com

贵，因此，BPF试图将多个帧装载进一个读缓存，只有缓存满了或者用户指明的超时到期才将读缓存保存的帧返回。tcpdump将超时定时器置为1秒，因为它一般从BPF收到很多数据。而RARP守护进程收到的帧很少，所以rarpd将超时置为0（收到一个帧就返回）。



图A-1 BSD分组过滤器

用户指明的过滤器告诉BPF用户进程对什么帧感兴趣，过滤器是对一个假想机器的一组指令。这些指令被内核中的BPF过滤器解释。在内核中过滤，而不在用户进程中，减少了必须从内核传递到用户进程的数据量。RARP守护进程总是使用绑定在程序里的、同样的过滤程序。另一方面，tcpdump在每次运行时，让用户在命令行指明一个过滤表达式。tcpdump将用户指明的表达式转换为相应的BPF的指令序列。tcpdump表达式的例子如下：

```
% tcpdump tcp port 25
% tcpdump'icmp [0] != 8 and icmp[0] != 0
```

第一个只打印源端口和目的端口为25的TCP报文段。第二个只打印不是回送请求和回送应答的ICMP报文（也就是非ping的分组）。这个表达式指明了ICMP报文的第一个字节，图6-2中的type字段，不等于8或0，即图6-3中的回送请求和回送应答。正像你所看到的，设计过滤器需要有底层分组结构的知识。第二个例子中的表达式被放在一对单引号中，防止Unix外壳程序解释特殊字符。

参考tcpdump (1) 的手册，了解用户可以指明的表达式的全部细节。bpf(4) 的手册详细描述了BPF使用的假想机器指令。[McCanne and Jacobson 1993] 比较了这个假想机器方法与其他方法的设计与性能。

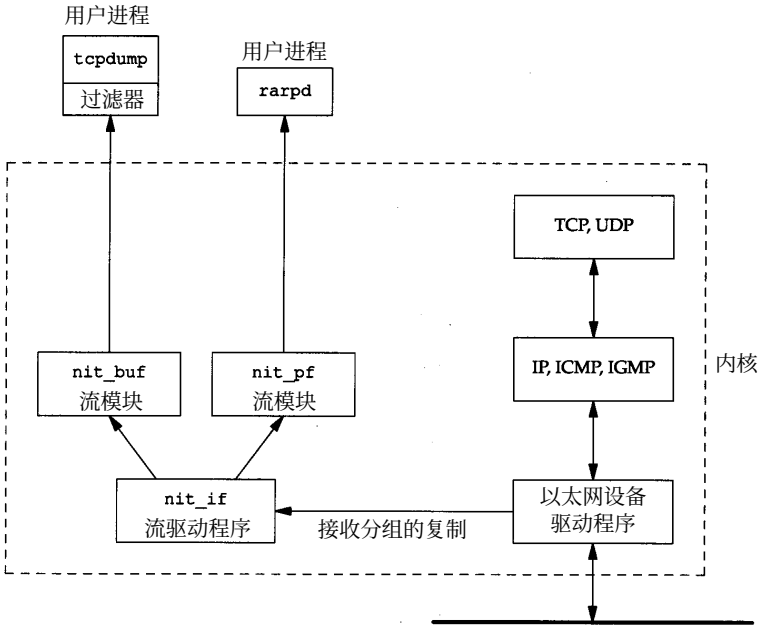
A.2 SunOS的网络接口分接头

SunOS 4.1.x提供了一个STREAMS伪设备驱动程序(pseudo-device driver)，称为网络接口分接头(Network Interface Tap)或者NIT（[Rago 1993] 包含了流设备驱动程序的其他细节。我

Linux公社 www.linuxidc.com

们把这种特征叫作“流”)。NIT类似于BSD分组过滤器，但不如后者功能强大和效率高。图A-2显示了使用NIT所用到的流模块。这个图与图A-1之间的一个不同点在于BPF可以截获网络接口收到的和传送的分组，而NIT只能截获接口收到的分组。将tcpdump与NIT结合起来意味着我们只能看见由网络中其他主机发送来的分组——即根本不可能看见我们自己主机发送的分组（尽管BPF可以工作在SunOS 4.1.x上，但它需要对以太网设备驱动程序的源代码进行改变，大多数的用户没有权限访问源代码，因而这是不可能的)。

当设备/dev/nit被打开时，流驱动程序nit_if就会被打开。既然NIT是使用流来构造的，处理模块可以放在nit_if驱动程序之上。tcpdump将模块nit_buf放在STREAM之上。这个模块将多个网络帧聚集在一个读缓存中，允许用户进程指明一个超时的值。这种情况类似于我们在BPF中所描述的。RARP守护进程没有把这个模块放在它的流之上，因为它只处理了一小部分分组。



图A-2 SunOS的网络接口分接头

用户指明的过滤由流模块nit_pf处理。在图A-2中，我们注意到这个模块被RARP守护进程所用，但没有被tcpdump使用。在SunOS操作系统中，tcpdump代之以在用户进程中完成自己的过滤操作。这么做的理由是nit_pf使用的假想机器的指令与BPF所支持的指令不同(不如BPF所支持的功能强大)。这就意味着当用户对tcpdump指明了一个过滤表达式时，与BPF相比较，使用NIT就会有更多的数据在内核与用户进程之间交换。

A.3 SVR4数据链路提供者接口

SVR4支持数据链路提供者接口DLPI (Data Link Provider Interface)，它是OSI数据链路服务定义的一个流实现。SVR4的大多数版本支持第1版的DLPI，SVR4.2同时支持第1版和第2版，Sun的Solaris 2.x支持第2版，但是增强了一些功能。

像tcpdump的网络监视程序必须使用DLPI来直接访问数据链路设备驱动程序。在Solaris 2.x中，分组过滤的流模块被改名为pfmod，缓存模块被改名为bufmod。

Linux公社 www.linuxidc.com

尽管Solaris 2.x还很新，tcpdump在其上的一个实现有一天也会出现。Sun还提供了一个叫作snoop的程序，完成的功能类似于 tcpdump（snoop代替了SunOS 4.x的程序etherfind）。作者还不知道tcpdump到vanilla SVR4上的任何端口实现。

A.4 tcpdump的输出

tcpdump的输出是“原始的”。在本书中包含它的输出时，我们对它进行了修改以便阅读。

首先，它总是输出它正在监听的网络接口的名字。我们把这一行给删去了。

其次，tcpdump输出的时间戳在一个微秒精度的系统中采用如同09:11:22.642008的格式，在一个10ms时钟精度的系统中则如同09:11:22.64一样（在附录B中，我们更多地讨论了计算机时钟的精度）。在任何一种情况下，HH:MM:SS的格式都不是我们想要的。我们感兴趣的是每个分组与开始监听的相对时间以及与下面分组的时间差。我们修改了输出以显示这两个时间差。第1个差值在微秒精度的系统中打印到十进制小数点后面6位（对于只有10 ms精度的系统打印到小数点后面2位），第2个差值打印到十进制小数点后面4位或2位（依赖于时钟精度）。

本书中大多数tcpdump的输出都是在sun主机上收集的，它提供了微秒精度。一些输出来自于运行0.9.4版BSD/386操作系统的主机bsdi，它只提供了10 ms的精度（如图5-1所示）。一些输出收集于当bsdi主机运行1.0版BSD/386时，后者提供了微秒级的精度。

tcpdump总是打印发送主机的名字，接着一个大于号，然后是目的主机的名字。这样显示很难追踪两个主机之间的分组流。尽管tcpdump输出仍然显示了数据流的方向，但我们经常把这条输出删掉，代替以产生一条时间线（在本书中的第一次出现是在图6-11）。在我们的时间线上，一个主机在左边，另一个在右边。这样很容易看出哪一边发送分组，哪一边接收分组。

我们给tcpdump的每条输出增加了行号，使得我们可以在书中引用特定的行。还在某些行之间增加了额外的空白，以区别一些不同的分组交换。

最后，tcpdump的输出可能会超出一页的宽度。我们在太长行的适当地方进行了换行。

作为一个例子，相应于图4-4的tcpdump的原始输出显示在图A-3中。这里假设了一个80列的终端窗口。

没有显示我们键入的中断键（用于中止tcpdump），也没有显示接收到的和漏掉的分组的个数（漏掉的分组是那些到达得太快，tcpdump来不及处理的分组。因为本文中的例子经常运行在另外一个空闲网络上，所以漏掉的分组个数总是0）。

```
sun % tcpdump -e
tcpdump: listening on le0
09:11:22.642008 0:0:c0:6f:2d:40 ff:ff:f f:ff:ff:ff arp 60: arp who-has svr4 tell
bsdi
09:11:22 .644182 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60: arp reply svr4 is-at 0:0
:c0:c2:9b:26
09:11:22.644839 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26  ip 60: bsdi.1030 > svr4.discard:
 S 596459521:596459521(0) win 4096 <mss 1024> [tos 0x10]
09:11:22. 649842 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60: svr4.discard > bsdi.1030:
 S 3562228225:3562228225(0) ack 596459522 win 4096 <mss 1024>
09:1 1:22.651623 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60: bsdi.1030 > svr4.discard:
 . ack 1 win 4096 [tos 0x10]

^?
9 packets received by filter
0 packets dropped by kernel
```

我们没有显示其他4个分组
键入中断字符来中断显示

图A-3 图4-4的tcpdump 的输出

Linux公社 www.linuxidc.com

A.5 安全性考虑

很明显，截获网络中传输的数据流使我们可以看到很多不应该看到的东西。例如，Telnet和FTP用户输入的口令在网络中传输的内容和用户输入的一样（与口令的加密表示相比，这称为口令的明文表示。在Unix口令文件中，一般是/etc/passwd或/etc/shadow，存储的是加密的表示）。然而，很多时候一个网络管理员需要使用一个类似于 tcpdump的工具来分析网络中出现的问题。

我们是把tcpdump作为一个学习的工具，用来查看网络中实际传输的东西。对 tcpdump以及其他厂商提供的类似工具的访问权限依赖于具体系统。例如，在 SunOS，对NIT设备的访问只限于超级用户。 BSD的分组过滤器使用了一种不同的技术：通过对 /dev/bpfXX设备的授权来控制访问。一般来说，只有属主才能读写这些设备（属主应该是超级用户），对于同组用户是可读的（经常是系统管理组）。这就是说如果系统管理员不对程序设置用户的 ID，一般的用户是不能运行类似于tcpdump的程序的。

A.6 插口排错选项

查看一个TCP连接上发生的事情的另一种方法是使能插口排错选项，当然是在支持这一特征的系统中。这个特征只能工作在 TCP上（其他协议都不行），并且需要应用程序支持（当应用程序启动时，使能一个插口排错选项）。

大多数伯克利演变的实现都支持这个特征，包括SunOS、4.4BSD和SVR4。

程序使能了一个插口选项，内核就会保留在那个连接上发生的事情的一个痕迹记录。在这之后，所有记录的信息都可以使用 trpt(8)程序打印出来。使能一个插口排错选项不需要特别的许可，但是因为trpt程序访问了内核的内存，所以运行 trpt需要特别的权限。

sock程序（附录C）的-D选项支持这个特征，但是输出的信息比相应的 tcpdump的输出更难解析和理解。然而，我们在 21.4节确实使用它查看了TCP连接上tcpdump不能访问的内核变量。

Linux公社 www.linuxidc.com

附录B　计算机时钟

既然本书中的大多数的例子都需要测量一个时间间隔，我们需要更仔细地介绍一下当前Unix系统所采用的记录时间的方法。下面的描述适用于本书中例子所使用的系统，也适用于大多数的Unix系统。[Leffler et al. 1989]的3.4节和3.5节给出了另外的细节。

硬件按照一定的频率产生一个时钟中断。对于Sun SPARC和Intel 80386，时钟中断每10 ms产生一次。

应该注意到大多数的计算机使用一种无补偿的晶体振荡器来生成这些时钟中断。正如RFC1305 [Mills 1992] 的表7指出的，你不要想知道这种振荡器一天的偏差有多少。这就意味着几乎没有计算机能维持精确的时间（即，中断并不是精确地每 10 ms发生一次）。一个0.01%的误差就会产生一个每天 8.64秒的差错。为了得到更好的时间测量需要：（1）一个更好的振荡器；（2）一个外部的更精确的时间资源（如，全球定位卫星提供的时间资源）；或者（3）通过因特网访问一个具有更精确时钟的系统。后者通过 RFC1305定义的网络时间协议实现，对该协议的描述超出了本书的范围。

Unix系统中引起时间差错的另一个公共的原因是 10 ms的中断只是引起内核给一个记录时间的变量增 1。如果内核丢失了一个中断（也就是说两个连续中断之间间隔 10 ms对于内核来说太快了），时钟将失去 10 ms。丢失这种类型的中断经常引起 Unix系统丢失时间。

尽管时间中断近似于每10 ms到达一次，更新的系统，如SPARC，提供了一个更高精度的定时器来测量时间差异。通过NIT驱动程序，tcpdump（在附录A中描述）已经访问了这个更高精度的定时器。在 SPARC上，这个定时器提供了微秒级的精度。用户进程也可以通过gettimeofday(2)函数来访问这个更高精度的定时器。

作者做了下面的试验。我们运行了一个程序，这个程序在一个循环里调用了10 000次gettimeofday函数，并将每次的返回值保存在一个数组中。在循环结束后，打印了9999个时间差。对于一个SPARC ELC，时间差的分布如图B-1所示。

微　秒	次数
36	4 914
37	4 831
38	167
39	8
其他	79

图B-1　在SPARC ELC上调用gettimeofday函数10 000次所需要的时间分布

在一个空闲的系统中，运行这个程序所花的时钟时间为 0.38秒。根据这一点，我们可以说进程调用gettimeofday所花的时间大约37微秒。既然ELC的速度是21 MIPS（MIPS表示每秒100万指令），37微秒相应于大约800个指令。这些指令对于内核处理一个用户进程的调用、执行系统调用、复制 8个字节的结果及返回给用户进程看起来是合理的（ MIPS速度是不可靠的，很难测量当前系统的指令时间。我们试图做的只是得到一个粗略的估计来评价一下上面的值是否有意义）。

从这个简单的试验，我们可以说gettimeofday返回的值确实包含了微秒级的精度。

Linux公社 www.linuxidc.com

如果我们在SVR4/386上进行类似的测试，结果是不同的。这是因为很多 386 Unix系统，如SVR4，只计数10 ms的时钟中断，而没有提供更高的精度。图 B-2是运行在 25 MHz 80386上的SVR4中9999个时间差的分布。

微 秒	次 数
0	9 871
10 000	128

图B-2 在SVR4/386上调用gettimeofday函数10000次所需要的时间

这些值是无意义的，因为时间差一般小于 10 ms，都被认为是0了。在这些系统中，我们所能做的就是在一个空闲的系统上测量时钟时间，除以循环的次数。这个结果提供了一个上界，因为它包含了调用 `printf`函数9999次的时间和将结果写入一个文件的时间（在 SPARC的情况，图B-1，时间差没有包括`printf`的时间，因为所有10 000个值都是首先获得的，然后才打印结果）。在SVR4的时钟时间为3.15秒，每个系统调用消耗了315微秒。这个大约比SPARC慢8.5倍的系统调用时间看来是正确的。

BSD/386 1.0版提供了类似于SPARC的微秒级的精度。它读8253时钟寄存器，计算从上次时钟中断以来的微秒次数。调用`gettimeofday`的进程和内核模块，如BSD 分组过滤器，可以使用这个精度。

和`tcpdump`联系起来，这些数字意味着我们可以相信在 SPARC和BSD/386系统上打印的毫秒和亚毫秒(submillisecond)的值。而在SVR4/386上，`tcpdump`打印的值总是10 ms的倍数。对于其他打印往返时间的程序，如 `ping`（第7章）和`traceroute`（第8章），在SPARC和BSD/386系统上，我们可以相信它们输出的毫秒值，但在 SVR4/386上，打印的值总是10的倍数。为了在LAN上测量某个`ping`的时间，在第 7章中我们显示的时间是 3 ms，所以需要在SPARC和BSD/386系统上运行`ping`程序。

本书中的一些例子是运行在BSD/386 0.9.4版上，它类似于SVR4，只提供了10 ms的时钟精度。在我们显示这个系统的`tcpdump`输出时，只显示到小数点后面两位，因为这就是所提供的精度。

Linux公社 www.linuxidc.com

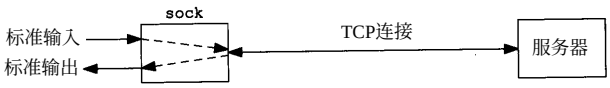
附录C　sock程序

在本书中一直使用一个称为 sock的小测试程序，用来生成 TCP和UDP数据。它既可以用作一个客户进程，也可以用作一个服务器进程。有这样一个可以从外壳程序执行的测试程序，使我们避免了为每一个我们想要研究的特征编写新的客户和服务器　C程序。因为本书的目的是了解网络互联协议，而不是网络编程，所以在这个附录中我们只描述这个程序和它不同的选项。

有很多与sock功能类似的程序。Juergen Nickelsen写了一个称为socket的程序，Dave Yost写了一个称为sockio的程序。两者都包含了很多类似的特征。sock程序的某些部分也受到了Mike Muuss和Terry Slattery所写的公开域ttcp程序的启发。

sock程序运行在以下四种模式之一：

1) 交互式客户：默认模式。程序和一个服务器相连，然后将标准输入的数据传给服务器，再将从服务器那里接收到的数据复制到标准输出。如图C-1所示。



图C-1　sock 程序作为交互式客户的默认操作

我们必须指明服务器主机的名字和想要连接的服务的名字。主机可指明为点分十进制数，服务可指明为一个整数的端口号。从 sun到bsdi与标准的echo服务器（1.12节）相连，回显我们键入的每一个字符：

```
sun  % sock  bsdi  echo
    a  test  line                    我们键入这一行
    a  test  line                    echo服务器返回一个复制行
    ^D                               键入文件结束符来中止
```

2) 交互式服务器：指明-s选项。需要指明服务名字（或端口号）：

```
sun  % sock  -s  5555                作为一个在端口5555监听的服务器
```

程序等待一个客户的连接请求，然后将标准输入复制给客户，将从客户接收到的东西复制到标准输出。在命令行中，端口号之前可以有一个因特网地址，用来指明接收哪一个本地接口上的连接：

```
sun  % sock  -s  140.252.13.33  5555        只接受来自以太网的连接
```

默认的模式是接受任何一个本地接口上的连接请求。

3) 源客户：指明-i选项。在默认情况下，将一个 1024字节的缓存写到网络中，写 1024次。-n选项和-w选项可以改变默认值。例如，

```
sun  % sock  -i  -n12  -w4096  bsdi  discard
```

把12个缓存，每个包含4096字节的数据，送给主机bsdi上的discard服务器。

4) 接收器服务器：指明-i选项和-s选项。从网络中读数据然后扔掉。

这些例子都使用了TCP（默认情况），-u选项指明使用UDP。

sock程序有许多选项，用于对程序的运行提供更好的控制。我们需要使用这些选项来产

Linux公社 www.linuxidc.com

生本书中用到的所有测试条件。

-b *n* 将*n*绑定为客户的本地端口号（在默认情况下，系统给客户分配一个临时的端口号）。

-c 将从标准输入读入的新行字符转换为一个回车符和一个换行符。类似地，当从网络中读数据时，将〈回车，换行〉序列转换为新行字符。很多因特网应用需要NVT ASCII（26.4节），它使用回车和换行来终止每一行。

-f *a.b.c.d.p* 为一个UDP端点指明远端的IP地址（*a.b.c.d*）和远端的端口号（*p*）。

-h 实现TCP的半关闭机制（18.5节）。即，当在标准输入中读到一个文件结束符时并不终止。而是在TCP连接上发送一个半关闭报文，继续从网络中读报文直到对方关闭连接。

-i 源客户或接收器服务器。向网络写数据（默认），或者如果和-s选项一起用，从网络读数据。-n选项可以指明写（或读）的缓存的数目，-w选项可以指明每次写的大小，-r选项可以指明每次读的大小。

-n *n* 当和-i选项一起使用时，*n*指明了读或写的缓存的数目。*n*的默认值是1024。

-p *n* 指明每个读或写之间暂停的秒数。这个选项可以和源客户（-i）或接收器服务器（-is）一起使用作为每次对网络读写时的延迟。参考-P选项，实现在第1次读或写之前暂停。

-q *n* 为TCP服务器指明挂起的连接队列的大小：TCP将为之进行排队的、已经接受的连接的数目（图18-23）。默认值是5。

-r *n* 和-is选项一起使用，*n*指明每次从网络中读数据的大小。默认是每次读1024字节。

-s 作为一个服务器，而不是一个客户。

-u 使用UDP，而不是TCP。

-v 详细模式。在标准差错上打印附加的细节信息（如客户和服务器的临时端口号）。

-w *n* 和-i选项一起使用，*n*指明每次从网络中写数据的大小。默认值是每次写1024字节。

-A 使能SO_REUSEADDR插口选项。对于TCP，这个选项允许进程给自己分配一个处于2MSL等待的连接的端口号。对于UDP，这个选项支持多播，它允许多个进程使用同一个本地端口来接收广播或多播的数据报。

-B 使能SO_BROADCAST插口选项，允许向一个广播IP地址发送UDP数据报。

-D 使能SO_DEBUG插口选项。这个选项使得内核为这个TCP连接维护另外的调试信息（A.6节）。以后可以运行trpt(8)程序输出这个信息。

-E 如果实现支持，使能IP_RECVDSTADDR插口选项。这个选项用于

Linux公社 www.linuxidc.com

		UDP服务器，用来打印接收到的UDP数据报的目的IP地址。
-F		指明一个并发的TCP服务器。即，服务器使用fork函数为每一个客户连接创建一个新的进程。
-K		使能TCP的SO_KEEPALIVE插口选项（第23章）。
-L	*n*	把一个TCP端点的拖延时间(linger time)（SO_LINGER）设置为*n*。一个为0的拖延时间意味着当网络连接关闭时，正在排队等着发送的任何数据都被丢弃，向对方发送一个重置报文（18.7节）。一个正的拖延时间（百分之一秒）是关闭网络连接必须等待的将所有正在排队等着发送的数据发送完并收到确认的时间。关闭网络连接时，如果这个拖延定时器超时，挂起的数据没有全部发送完并收到确认，关闭操作将返回一个差错信息。
-N		设置TCP_NODELAY插口选项来禁止Nagle算法（19.4节）。
-O	*n*	指明一个TCP服务器在接受第一个客户连接之前暂停的秒数。
-P	*n*	指明在第一次对网络进行读或写之前暂停的秒数。这个选项可以和接收器服务器（-is）一起使用，完成在接受了客户的连接请求之后但在执行从网络中第一次读之前的延迟。和接收源（-i）一起使用时，完成连接建立之后但第一次向网络写之前的延迟。参看-p选项，实现在接下来的每一次读或写之间进行暂停。
-Q	*n*	指明当一个TCP客户或服务器收到了另一端发来的一个文件结束符，在它关闭自己这一端的连接之前需要暂停的秒数。
-R	*n*	把插口的接收缓存（SO_RCVBUF插口选项）设置为*n*。这可以直接影响TCP通告的接收窗口的大小。对于UDP，这个选项指明了可以接收的最大的UDP数据报。
-S	*n*	把插口的发送缓存（SO_SNDBUF插口选项）设置为*n*。对于UDP，这个选项指明了可以发送的最大的UDP数据报。
-U	*n*	在向网络写了数字*n*后进入TCP的紧急模式。写一个字节的数据以启动紧急模式（20.8节）。

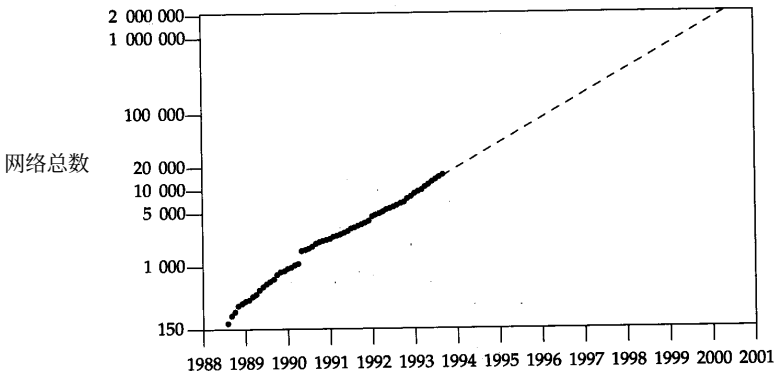
Linux公社 www.linuxidc.com

附录D　部分习题的解答

第1章

1.1　答案是：2^7-2（126）＋$2^{14}-2$（16 382）＋$2^{21}-2$（2 097 150）= 2 113 658。每一部分都减去2是因为全0或全1网络ID是非法的。

1.2　图D-1显示了直到1993年8月的有关数据。



图D-1　宣布加入NSFNET的网络数

如果网络数继续呈指数增长的话，虚线估计了 2000年可能达到的最大的网络数。

1.3　“ 自由地接收，保守地发送。”

第3章

3.1　不，任何网络ID为127的A类地址都是可行的，尽管大多数系统使用了 127.0.0.1。

3.2　`kpno`有5个接口：3个点对点链路和 2 个以太网接口。 R10有4个以太网接口。 `gateway`有3个接口：2个点对点链路和1 个以太网接口。最后， `netb`有1个以太网接口和2个点对点链路。

3.3　没有区别：作为一个没有再区分子网的 C类地址，它们都有一个 255.255.255.0的子网掩码。

3.5　它是合法的，被称为非连续的子网掩码，因为其用于子网掩码的 16位是不连续的。但是RFC建议反对使用非连续的子网掩码。

3.6　这是一个历史遗留问题。值是 1024＋512，但是打印的MTU值包含了所有需要的首部字节数。Solaris 2.2将回环接口的MTU设置为8232（8192＋40），其中包含了8192字节的用户数据加上20字节的IP首部和20字节的TCP首部。

3.7　第一，数据报降低了路由器中对于连接状态的需求。第二，数据报提供了基本的构件，在它的上面可以构造不可靠的（ UDP）和可靠的（TCP）的运输层。第三，数据报代表

Linux公社 www.linuxidc.com

了最小的网络层假定，使得可以使用很大范围的数据链路层服务。

第4章

4.1 发出一条rsh命令与另一台主机建立一个TCP连接。这样做引起在两个主机之间交换IP数据报。为此，在那台主机的ARP缓存中必须有我们这台主机的登记项。因此，即使在执行rsh命令之前，ARP缓存是空的，当rsh服务器执行arp命令时，必须保证ARP缓存中登记有我们这台主机。

4.2 保证你的主机上的ARP缓存中没有登记以太网上的某个叫作foo的主机。保证foo引导时发送一个免费ARP请求，也许是在foo引导时，在那台主机上运行tcpdump。然后关闭主机foo，使用说明了temp选项的arp命令，在你的系统的ARP缓存中为foo输入一个不正确的登记项。引导foo并在它启动好之后，察看主机的ARP缓存，看看不正确的登记项是不是已经被更正了。

4.3 阅读主机需求（Host Requirement）RFC的2.3.2.2节和本书中的11.9节。

4.4 假设当服务器关闭时，客户机保存了关于服务器的一个完整的ARP登记项。如果我们继续试图与（已关闭的）服务器联系，过了20分钟以后，ARP将超时。最后，当服务器以另一个新的硬件地址重启动。如果它没有发出一个免费ARP，旧的、不再正确的ARP登记项仍然存在于客户机上。我们将无法和在新硬件地址上的服务器联系直到我们手工删除这个ARP登记项，或者在20分钟内停止与服务器联系的尝试。

第5章

5.1 一个单独的帧类型并不是必需的，因为图4-3中的op字段对于所有的四个操作（ARP请求、ARP应答、RARP请求和RARP应答）都有一个不同的值。但是实现一个RARP服务器，独立于内核中的ARP服务器，更容易处理不同的帧类型字段。

5.2 每个RARP服务器在发送一个响应之前可以延迟一个小的随机时间。

作为一个优化，可以指定一个RARP服务器为主服务器，其他的为次服务器。主服务器发出响应不需要延迟，而次服务器发出响应则需要一个随机的延迟。

作为另一个优化，也是指定一个主RARP服务器，其他为次服务器。次服务器只对在一个短时间段内发生的重复请求进行响应。这里假设出现重复请求的原因是由于主服务器停机了。

第6章

6.1 如果在局域网线上有一百个主机，每个都可能在同一时刻发送一个ICMP端口不可达的报文。很多报文的传输都可能发生冲突（如果使用的是以太网），这将导致1秒或2秒的时间里网络不可用。

6.2 它是一个“should”。

6.3 如我们在图3-2所指出的，发送一个ICMP差错总是将TOS置为0。发送一个ICMP查询请求可以将TOS置为任何值，但是发送相应的应答必须将TOS置为相同的值。

6.4 netstat -s是查看每个协议统计数据的常用方法。在一台收到了4800万个IP数据报的SunOS 4.1.1主机（gemini）上，ICMP的统计为：

Linux公社 www.linuxidc.com

```
Output histogram:
        echo reply: 1757
        destination unreachable: 700
        time stamp reply: 1
Input histogram:
        echo reply: 211
        destination unreachable: 3071
        source quench: 249
        routing redirect: 2789
        echo: 1757
        #10: 21
        time exceeded: 56
        time stamp: 1
```

21个类型为10的报文是SunOS 4.1.1不支持的路由器的请求。

也可以使用 SNMP（图25-26），有些系统，如 Solaris 2.2，可以生成使用 SNMP变量名的netstat-s的输出。

第7章

7.2 86字节除以960字节/秒，乘以2得到179.2 ms。当以这个速率运行ping时，打印的值为180 ms。

7.3 （86＋48）除以960字节/秒，乘以2得到279.2 ms。另外的48字节是因为56字节的数据部分的最后48字节必须忽略：0xc0是SLIP END字符。

7.4 CSLIP只压缩了TCP报文段的TCP首部和IP首部。它对ping使用的ICMP报文没有作用。

7.5 在一个SPARC工作站 ELC上，对回环地址的ping操作产生一个1.310 ms的RTT，而对一个主机的以太网地址的ping操作产生一个1.460 ms的RTT。这个差值是由于以太网驱动程序需要时间来判定这个数据报的目的地址是一个本地的主机。需要一个产生微秒级输出的ping来验证这一点。

第8章

8.1 如果一个输入数据报的 TTL为0，做减一操作然后测试会将把 TTL设置为255，并且让数据报继续传输。尽管一个路由器永远不会收到一个 TTL为0的数据报，但这种情况确实会发生。

8.2 我们注意到traceroute在UDP数据报的数据部分存储了 12个字节，其中包含了数据报发送的时间。然而，从图6-9可以看出ICMP只返回了出错的IP数据报的头8个字节，实际上这是8个字节的UDP首部。因此，ICMP的差错报文没有返回traceroute存储的时间值。traceroute保存了它发送分组的时间，当收到一个ICMP应答时，取出当时的时间，把两个值相减就可以得出RTT。

回忆一下第7章中，ping在输出的ICMP回显请求中存储了时间，这个值被服务器回显了回来。这样即使分组返回时失序，ping也能打印出正确的RTT。

8.3 第1行输出是正确的，并且标识了 R1。下一个探测分组启动时将 TTL置为2，并且这个值被R1减1。当R2收到这个分组时，把TTL从1减为0，但是错误地将它传递给了 R3。R3看见进入的TTL是0就将超时的分组发送回来。这就意味着第2行输出（TTL为2）标识了R3，而不是R2。第3行输出正确地标识了 R3。这个错误所表现出来的线索就是两个连续的输

Linux公社 www.linuxidc.com

出行标识了同一个路由器。

8.4 在这种情况下，TTL为1标识了R1，TTL为2标识了R2，TTL为3标识了R3，但是当TTL为4时，UDP数据报到达了目的地，其输入的TTL为1。ICMP端口不可达报文生成了，但它的TTL是1（错误地从进入的TTL复制而来）。这个ICMP报文到了R3，在那儿TTL被减1，报文被丢弃。没有生成一个ICMP超时报文，因为被丢弃的数据报是一个ICMP的差错报文（端口不可达）。类似的现象也出现在TTL为5的探测分组，但这次输出的端口不可达报文以TTL为2开始（进入的TTL），这个报文被传给R2，在那儿被丢弃。对应于TTL为6探测分组的端口不可达报文被传递给R1，在那儿被丢弃。最后，对应于TTL为7探测分组的端口不可达报文被送回了原地，到达时它的TTL为1（traceroute认为一个TTL为0或1的到达ICMP报文是有问题的，因此它在RTT后面打印了一个惊叹号）。总之，TTL为1、2和3的行正确地标识了R1、R2和R3，接下来的三行每个都包含三个超时，再接下来的TTL为7的行标识了目的地。

8.5 它表明这些路由器都将一个ICMP报文的输出TTL设置为255，这是共同的。从netb输入的255的TTL值是我们想要的，而从butch输入的253的TTL值表明在butch和netb之间可能有一个未觉察的路由器。否则，在这个点上我们应该看到一个TTL值为254的输入报文。类似地，我们希望看到一个值为252而不是249的、来自enss142.UT.westnet.net的报文。这表明这些未觉察的路由器没有正确地处理向外输出的UDP数据报，但它们都对返回的ICMP报文正确地进行了TTL减1操作。

我们必须在查看输入的TTL时非常小心，因为有时候一个和我们想要的不同的值可能是由于返回的ICMP报文采用了一条与输出UDP数据报不同的路径。但是，在这个例子中证实了我们所怀疑的——当使用松源路由选项时，确实存在traceroute没有发现的未察觉的路由器。

8.7 ping的客户把ICMP回显请求报文（图7-1）的标识符字段设置为它的进程ID。ICMP回显应答报文包含同样值的标识符字段。每个客户都要查看这个返回的标识符字段，并且只处理那些它发送过的报文。

traceroute客户将它的UDP源端口号设置为它的进程ID和32768的逻辑或。因为返回的ICMP报文总是包含产生错误的IP数据报的前8个字节（图6-9），这8个字节包括了完整的UDP首部，所以这个源端口号在ICMP差错报文中被返回。

8.8 ping客户将ICMP回显请求报文的可选数据部分设置为分组发送的时间。这个可选的数据必须在ICMP回显应答中返回。这样使得即使分组返回时失序，客户也能计算出精确的回环时间。

traceroute客户不能这样操作，因为在ICMP差错报文中返回的只是UDP首部（图6-9），没有UDP数据。因此，traceroute必须记住它发送一个请求的时间，等待应答，然后计算两者的时间差。

这里显示了ping和traceroute的另一个不同点：ping每秒发送一个分组，而不管是否收到任何应答；traceroute发送一个请求，然后在发送下一个请求前等待一个应答或者一个超时。

8.9 因为默认情况下Solaris 2.2从32768开始使用临时的UDP端口，所以目的主机上的目的端口已经被使用的机会更大。

Linux公社 www.linuxidc.com

第9章

9.1 当ICMP标准第1次发布时，RFC 792 [Postel 1981b]所述的划分子网技术还没有使用。另外，使用一个网络重定向而不是N个主机重定向（对于目的网络中的所有*N*个主机）也节省了路由表的空间。

9.2 这一项并不需要，但是如果把它删除了，所有到`slip`的IP数据报将被发送到默认的路由器（`sun`），后者又将把它们送到路由器`bsdi`。既然`sun`将数据报从与接收数据报相同的接口转发出去，它把一个ICMP重定向到`svr4`。这样在`svr4`中又创建了我们删除过的同样的路由表项，尽管这一次是因为重定向而创建的，而不是在引导时增加的。

9.3 当那个4.2BSD主机收到目的地址是140.1.255.255的数据报，发现它有一个通往该网络（140.1）的路由，因此就试图转发数据报。它发送一个ARP广播来寻找140.1.255.255。这个ARP请求没有收到任何应答，所以这个数据报最终被丢弃。如果在网线上有很多这样的4.2BSD主机，每一个都在差不多同一时刻发送ARP这个广播，将会暂时地阻塞网络。

9.4 这次，每一个ARP请求都收到一个应答，告诉每个4.2BSD主机向一个指定的硬件地址（以太网广播）发送数据报。如果网线上有*k*个这样的4.2BSD主机，全部收到了它们自己的ARP应答，使得每一个生成了另一个广播。每个主机都收到了每一个目的地址为140.1.255.255的广播IP数据报，既然现在每个主机都有一个ARP缓存项，这个数据报又被转发给了广播地址。这个过程继续下去，就会产生一次以太网的熔毁（Ethernet meltdown）。[Manber 1990] 描述了网络中另一种形式的链式反应。

第10章

10.1 路由表中有13条来自于`kpno`：除了140.252.101.0和140.252.104.0之外的所有`gateway`直接相连的其他网络。

10.2 丢失的数据报中通告的25条路由需要60秒才能得到更新。这不成问题，因为一般来说一条路由如果连续3分钟没有得到更新，RIP才会声明它失效。

10.3 RIP运行在UDP上，而UDP提供了UDP数据报中数据部分的一个可选的检验和（11.3节）。然而，OSPF运行在IP上，IP的检验和只覆盖了IP首部，所以OSPF必须增加它自己的检验和字段。

10.4 负载平衡增加了分组被失序交付的机会，并且很可能使得运输层计算的环回时间出错。

10.5 这叫作简单的分裂范围（split horizon）。

10.6 在图12-1中，我们显示了100个主机的每一个都通过设备驱动程序、IP层和UDP层来处理这个广播的UDP数据报。当它们发现UDP端口520没有被使用时，这个广播数据报才最终被丢弃。

第11章

11.1 因为使用IEEE 802封装时，存在8个额外的首部字节，所以1465个字节的用户数据是引起分片的最小长度。

11.3 对于IP来说有8200字节的数据需要发送，8192字节的用户数据和8个字节的UDP首部。

Linux公社 www.linuxidc.com

采用tcpdump记号，第1个分片是1480@0+（1480字节的数据，偏移为0，将“更多片”比特置1）。第2个是1480@1480+，第3个是1480@2960+，第4个是1480@4440+，第5个是1480@5920+，第6个是800@7400。1480×5 + 800 = 8200，正好是要发送的字节。

11.4 每个1480字节的数据报片被分成三小片：两个528字节和一个424字节。小于532（552-20）的8的最大倍数是528。800字节的数据报片被分成两小片：一个528字节和一个272字节。这样，原来8192字节的数据报变成了SLIP链路上的17个帧。

11.5 不。问题是当应用程序超时重传时，重传产生的IP数据报有一个新的标识字段。而重新装配只针对那些具有相同标识字段的分段。

11.6 IP首部中的标识字段（47942）是一样的。

11.7 第一，从图11-4我们看到gemini没有使能输出UDP的检验和。如果输出UDP的检验和没有被使能，这个主机上的操作系统（SunOS 4.1.1）就不会验证一个进入UDP的检验和。第二，大多数的UDP通信量都是本地的，而不是WAN的，因此没有服从所有的WAN特征。

11.8 不严格的和严格的源站选路选项被复制到每一个数据报片中。时间戳选项和记录路由选项没有被复制到每一个数据报片中——它们只出现在第1个数据报片中。

11.9 不。在11.12节中，我们看到很多实现可以根据目的IP地址、源IP地址和源端口号来过滤送往一个给定UDP端口号的输入数据报。

第12章

12.1 广播本身不会增加网络通信量，但它增加了额外的主机处理时间。如果接收主机不正确地响应了诸如ICMP端口不可达之类的差错，那么广播也可能导致额外的网络通信量。路由器一般不转发广播分组，而网桥一般转发，所以在一个桥接网络上的广播分组可能比在一个路由网络上走得更远。

12.2 每个主机都收到了所有广播分组的一个副本。接口层收到了帧，把它传递给设备驱动程序。如果类型字段指的是其他协议，设备驱动程序就会丢弃该帧。

12.3 首先执行netstat-r来看一下路由表，结果显示了所有接口的名字。然后对每个接口执行ifconfig（3.8节）：标志指出了一个接口是否支持广播，如果支持，相应的广播地址也会被输出。

12.4 伯克利演变的实现不允许对一个广播数据报进行分片。当我们说明了1472字节的长度，产生的IP数据报将是1500字节，正好是以太网的MTU。不允许分片一个广播数据报是一个策略上的决定——没有技术上的原因（并不是想要减少广播分组的数目）。

12.5 依赖于100个主机上不同的以太网接口卡的多播支持，多播数据报可能被接口卡忽略，或者被设备驱动程序丢弃。

第13章

13.1 生成随机数时要使用对于主机唯一的值。IP地址和链路层地址是每个主机都应该不一样的两个值。日期时间是一个不好的选择，尤其是在所有的主机都运行了一个类似于NTP的协议来同步它们的时钟的情况下。

13.2 他们增加了一个包括一个序号和一个时间戳的应用协议首部。

Linux公社 www.linuxidc.com

第14章

14.1 一个解析器总是一个客户，但一个名字服务器既是一个客户又是一个服务器。

14.2 问题被返回，它占用了前44个字节。一个回答占用了剩下来的31个字节：2个字节指向域名的指针（即，指向问题中域名的一个指针），10字节固定长度的字段（类型、种类、TTL和资源长度），19字节的资源数据（一个域名）。注意到资源数据中的域名（`svr4.tuc.noao.edu.`）没有共享问题（`34.13.252.140.in-addr.arpa.`）中域名的后缀，所以不能使用一个指针。

14.3 将顺序颠倒意味着首先使用DNS，如果使用DNS失败，然后才将参数翻转过来作为一个点分十进制数。这就是说每次说明一个点分十进制数，都要使用DNS，涉及一个名字服务器。这是对资源的一种浪费。

14.4 RFC 1035的4.2.2节说明了在实际的DNS报文之前的两个字节长度的字段。

14.5 当一个名字服务器启动时，它一般从一个磁盘文件中读出一个根服务器列表（可能已经过时了）。然后尝试和这些根服务器中的一个联系，请求根域的名字服务器记录（一个NS的查询类型）。这个请求返回了当前最新的根服务器列表。启动磁盘文件中根服务器项中至少需要一个是有效的。

14.6 InterNIC的注册服务每一周更新三次根服务器。

14.7 就像应用是不定的一样，解析器也是不定的。如果系统配置成使用多个名字服务器，而且解析器是无状态的，那么解析器就不能记住不同的名字服务器的往返时间。这样定时太短的解析器将会超时，引起不必要的重传。

14.8 对A记录的排序应该由解析器来执行，而不是名字服务器，因为解析器一般比服务器了解更多的客户的网络拓扑（更新版本的BIND提供了解析器对A记录排序的功能）。

第15章

15.1 送往广播地址的TFTP请求应该被忽略。正像Host Requirements RFC所描述的，对一个广播请求的响应可能产生一个非常严重的安全漏洞。但是，问题是并不是所有的实现和API都对接收一个UDP数据报的进程提供了该数据报的目的地址（11.12节）。因为这个原因，很多TFTP服务器没有严格遵守这个限制。

15.2 不幸的是，RFC没有提到这个块数目环绕问题。具体实现时应该能够传输最大为33 553 920（65535×512）字节的文件。但是当文件的长度超过16 776 704（32767×512）时，很多实现都会失败，因为它们将块数目错误地表示为一个有符号的16位整数，而不是一个无符号的整数。

15.3 这样简化了编写一个适合于只读内存的TFTP客户的工作，因为服务器是引导文件的发送者，所以服务器必须实现超时和重传机制。

15.4 利用它的停止等待协议，TFTP可以在每一次客户与服务器的往返过程中最多传输512字节的数据。TFTP的最大吞吐量就是512字节除以客户与服务器之间的往返时间。在以太网上，假设一个往返时间为3 ms，那么最大的吞吐量就是大约170 000字节/秒。

第16章

16.1 一个路由器可以转发一个RARP请求到路由器连接的其他网络上的任何一台主机上。但

Linux公社 www.linuxidc.com

是发送应答就成问题了，路由器还必须转发 RARP应答。

BOOTP没有这个应答问题，因为应答的地址是路由器知道如何转发的一般 IP地址。问题是RARP只使用了链路层地址，路由器一般不知道在其他的、没有连接在路由器的网络上主机的链路层地址。

16.2 它可能使用了自己的硬件地址。该地址应该是唯一的，在请求报文中设置，在应答中返回。

第17章

17.1 除了UDP的检验和，其他都是必需的。IP检验和只覆盖了IP首部，而其他字段都紧接着IP首部开始。

17.2 源IP地址、源端口号或者协议字段可能被破坏了。

17.3 很多Internet应用使用一个回车和换行来标记每个应用记录的结束。这是 NVT ASCII采用的编码（26.4节）。另外一种技术是在每个记录之前加上一个记录的字节计数， DNS（习题14.4）和Sun RPC（29.2节）采用了这种技术。

17.4 就像我们在6.5节所看到的，一个ICMP差错报文必须至少返回引起差错的IP数据报中除了IP首部的前8 个字节。当TCP收到一个ICMP差错报文时，它需要检查两个端口号以决定差错对应于哪个连接。因此，端口号必须包含在TCP首部的前8个字节里。

17.5 TCP首部的最后有一些选项，但UDP首部中没有选项。

第18章

18.1 ISN是一个32 bit的计数器，它在系统引导大约 9.5小时后从4 294 912 000环绕到8704。再过大约9.5小时它将环绕到17 408，然后再过9.5小时是26 112，如此继续下去。因为系统引导时ISN从1开始，并且因为最低次序的数字在 4、8、2、6和0之间循环，所以ISN应该总是一个奇数。

18.2 在第1种情况下，我们使用了 sock程序。默认情况下它把Unix的新行字符不作改变地进行传输 —— 单个ASCII字符012（八进制）。在第2种情况下，我们使用了 Telnet客户，它把Unix的新行字符转变为两个 ASCII字符 —— 一个回车符（八进制 015）跟着一个换行符（八进制012）。

18.3 在一个半关闭的连接上，一个端点已经发送了一个 FIN，正等待另一端的数据或者一个FIN。一个半打开的连接是当一个端点崩溃了，而另一端还不知道的情况。

18.4 一个连接只有经过了已建立状态才能进入 2MSL等待状态。

18.5 首先，日期服务器在将时间和日期写给客户之后对 TCP连接做一个主动关闭。这可以通过sock程序打印的消息：“ connection closed by peer. ”表现出来。连接的客户端经历了被动关闭的状态。这样就把一对插口置于服务器端的 TIME_WAIT状态，而不是在客户端。

其次，正如在18.6节所示，大多数伯克利演变的实现都允许一个新的连接请求到达一个正处于TIME_WAIT状态的一对插口，这也就是这里所发生的情况。

18.6 因为在一个已经关闭的连接上到达了一个 FIN，所以相应于这个FIN发送了一个复位。

18.7 拨号的一方做主动打开，电话振铃的一方做被动打开。不允许同时打开，但允许同时关

Linux公社 www.linuxidc.com

闭。

18.8 我们将只看到ARP请求，而不是TCP SYN报文段，但ARP请求将和图中具有相同的计时。

18.9 客户在主机solaris上，服务器在主机bsdi上。客户对服务器SYN的确认（ACK）和客户的第一个数据段结合在一起（第3行）。这种处理非常符合TCP的规则，尽管大多数的实现都没有这么做。接着，客户在等待它的数据的确认之前发送了它的 FIN（第4行）。这样使得服务器可以在第5行同时确认客户数据和它的FIN。

这次交互（将一个报文段从客户发送到服务器）需要 7个报文段，而正常的连接建立和终止（图18-13），以及一个数据段和它的确认，需要9个报文段。

18.10 首先，服务器对客户的 FIN的确认一般不会被延迟（我们在 19.3节讨论延迟的确认），而是在FIN到达后立即发送。应用进程需要一些时间来接收 EOF，告诉它的TCP关闭它这一端的连接。第二，服务器收到客户的 FIN后，并不一定要关闭它这一端的连接。就像我们在18.5节中看到的，仍然可以发送数据。

18.11 如果一个产生RST的到达报文段有一个 ACK字段，那么RST的序号就是到达的 ACK字段。第6行中值为1的ACK是相对于第2行中的26368001的ISN。

18.12 参见 [Crowcroft et al. 1992]中对分层的评论。

18.13 发出了5个查询。假设有3个分组用于建立连接， 1个用于查询， 1个用于确认查询， 1个用于响应， 1个用于确认响应， 4个用于终止连接。这就是说每次查询需要 11个分组，总共需要55个分组。使用UDP则可以减少到10个分组。

如果对查询的确认和响应结合在一起，每个查询需要的分组可以减少到 10个（19.3节）。

18.14 限制是每秒268个连接：最大数目的TCP端口号（65536 - 1024 = 64512，忽略知名端口）除以TIME_WAIT状态的2MSL。

18.15 重复的FIN会得到确认， 2MSL定时器重新开始。

18.16 在TIME_WAIT状态中收到一个RST引起状态过早地终止。这就叫作 TIME_WAIT断开(assassination)。RFC1337 [Braden 1992a]仔细讨论了这个现象，并显示了潜在的问题。这个RFC提出的简单的修改就是在TIME_WAIT状态时忽略RST段。

18.17 它是在具体实现不支持半关闭连接的时候。一旦应用进程引起发送一个 FIN，应用进程就不能再从这个连接读数据了。

18.18 不。输入的数据段通过源 IP地址、源端口号、目的 IP地址和目的端口号进行区分。在18.11节中我们看到对于输入的连接请求，一个 TCP服务器一般可以通过目的 IP地址来拒绝接收。

第19章

19.1 应用程序的两个写操作，跟着一个读操作，引起了迟延，因为 Nagle算法很可能被激活。第一个报文段（包含8个字节的数据）被发送后，在发送后面 12个字节的数据之前必须等待第一个报文段的确认。如果服务器实现了延迟的确认，在收到这个确认之前，可能会有一个达到200 ms的时延（加上RTT）。

19.2 假设5个字节的CSLIP首部（IP和TCP）和两个字节的数据，这些段通过 SLIP链路的RTT大约是14.5 ms。我们要加上通过以太网的 RTT（一般是5~10 ms），加上在sun和bsdi

Linux公社 www.linuxidc.com

上的选路时间。因此，观察到的时间看起来是正确的。

19.3 在图19-6中，第6和第9报文段之间的时间是533ms。在图19-8中，第8和第12报文段之间的时间是272ms（我们测量了F2键的时间，而不是F1键，因为F1键的第1个回显在第2个图中丢掉了）。

第20章

20.1 字节号0是SYN，字节号8193是FIN。SYN和FIN在序号空间里各占用了一个字节。

20.2 应用程序的第1个写操作引起置位 PUSH标志发送第1个报文段。因为 BSD/386总是使用慢启动，它在发送更多数据之前等待第1个确认。在这段时间里，下面三个写操作发生了，发送TCP把要发送的数据缓存了起来。因为在缓存中有更多的数据要发送，下面的三个报文段没有包含PUSH标志。最后，慢启动跟上了应用程序的写操作，每个应用程序的写操作引起了发送一个报文段，并且因为那个报文段是缓存中最后的一个，所以设置PUSH标志。

20.3 为容量求解带宽迟延方程式，第一种情况是 1920字节，卫星的情况是2062字节。看起来TCP只声明了一个2048字节的窗口。

一个大于16000字节的窗口应该能够使卫星链路饱和。

20.4 不，因为TCP超时之后可能重新对数据进行分组，就像我们将在 21.11节中看到的。

20.5 作为应用进程读数据的结果，第 15报文段是TCP模块自动发送的窗口更新，它引起窗口的打开。这类似于图中的第 9报文段。但是，第 16报文段是应用进程关闭它这一端连接的结果。

20.6 这可能引起发送者以比网络实际能够处理的更快的速率向网络发送分组。这叫作确认压缩(ACK compression)或确认粉碎(ACK smashing) [Mogul 1993, 15.8.13节]。这个引用显示了在Internet上发生的ACK压缩，尽管它很少会导致拥塞。

第21章

21.1 下一个超时设定是48秒：0＋4×12。因子4是指数退避的下一个乘数。

21.2 看起来SVR4在计算*RTO*时仍然使用了因子2*D*，而不是4*D*。

21.3 TFTP使用的停止等待协议被限制为每个往返过程传送 512字节的数据。32768/512×1.5是96秒。

21.4 显示了4个报文段，编号为1、2、3和4。假设接收的顺序是1、3、2和4，接收者产生的确认将是ACK 1（一个正常的确认），ACK 1（当收到了报文段3，失序后一个重复的确认），当收到了报文段2后的ACK 3（对报文段2和报文段3的确认），然后是ACK 4。这儿产生了一个重复的确认。如果接收的顺序是 1、3、4和2，将会产生两个重复的确认。

21.5 不，因为斜率仍然是向上和向右，不是向下。

21.6 参见图E-1。

21.7 在图21-2中，报文段包含256个字节的数据，在`slip`和`bsdi`之间的9600 b/s的CSLIP链路传输大约需要250 ms。假定数据段没有在`bsdi`和`vangogh`之间的某个地方排队，它们分别需要大约250 ms到达`vangogh`。因为这个时间超过了延迟确认定时器的 200 ms时间，当下一个延迟确认定时器超时时，每个报文段得到确认。

Linux公社 www.linuxidc.com

第22章

22.1 主机bsdi上的确认很可能都要被延迟，因为没有理由立即发送它们。这就是为什么相对次数有一个0.170和0.370的小数部分。看起来bsdi上200 ms的定时器在sun上同样的定时器之后18 ms才开始计时。

22.2 FIN标志，和SYN标志一样，在序号空间占据了 1个字节。通知的窗口看起来小了一个字节，因为TCP允许FIN标志在序号空间占用1个字节的空间。

第23章

23.1 通常激活keepalive选项比显式地编写应用程序探测报文更容易；keepalive探测报文比应用程序探测报文占用更少的网络带宽（因为 keepalive探测报文和应答不包含任何数据)；如果连接不是空闲的，就不会发送探测报文。

23.2 keepalive选项可能会由于一个临时性的网络中断而引起一个非常好的连接断开；发送探测报文的间隔（2小时）一般不可以根据应用程序进行配置。

第24章

24.1 它意味着发送TCP支持窗口扩缩选项，但这个连接并不需要扩缩它的窗口。另一端（接收这个SYN的）可以说明一个窗口扩缩因子（可以是 0或非0)。

24.2 64 000：接收缓存大小（128 000）向右移1位。55 000：接收缓存（220 000）向右移2位。

24.3 不。问题是确认没有可靠地交付（除非它们被数据捎带在一起发送），因此，一个确认上的扩缩改变可能会丢失。

24.4 $2^{32}\times 8/120$等于286 Mb/s，2.86倍于FDDI数据率。

24.5 每个TCP将不得不记住从每个主机的任何一个连接上收到的上一个时间戳。阅读 RFC 1323的附录B.2以了解进一步的细节。

24.6 应用程序必须在和另一端建立连接之前设置接收缓存的大小，因为窗口扩缩选项在初始的SYN段中发送。

24.7 如果接收者每次确认第2个数据报文段，吞吐量是 1 118 881字节/秒。若使用一个62个报文段的窗口，每31个报文段确认一次，则数值是1 158 675。

24.8 使用这个选项，确认中回显的时间戳总是来自于引起确认的报文段。对哪个重传的报文段进行确认没有疑问，但仍然需要Karn算法的另一部分，即处理重传的指数退避。

24.9 接收TCP对数据进行排队，但只有完成了三次握手后，数据才能传递给应用程序：当接收TCP进入了ESTABLISH状态。

24.10 交换了5个报文段：

1) 客户到服务器：SYN，数据（请求）和FIN。服务器必须像上个习题所述的一样对数据进行排队。

2) 服务器到客户：SYN和对客户SYN的确认。

3) 客户到服务器：对服务器的 SYN的确认和客户的FIN（再次）。这样使得服务器进入已建立状态，并且来自报文段1的排队数据被传递给服务器应用程序。

4) 服务器到客户：客户 FIN的确认（它也确认了客户的数据）、数据（服务器的应答）

Linux公社 www.linuxidc.com

和服务器的FIN。这里假定了SPT足够短以允许这个延迟的确认。当客户 TCP收到这个报文段，就将应答传递给客户端的应用程序，但是整个时间是 RTT加上SPT的两倍。

5) 客户到服务器：对服务器FIN的确认。

24.11 每秒16 128次交互（64 512除以4）。

24.12 使用T/TCP的交互时间不可能比两个主机之间交换一个 UDP数据报所需的时间短。因为T/TCP涉及了UDP没有做的状态处理，所以T/TCP总是要花更多的时间。

第25章

25.1 如果一个系统运行了一个管理进程和一个代理进程，它们很可能是不同的进程。管理进程在UDP端口162监听trap告警，代理进程在 UDP端口161等待请求。如果 trap告警和SNMP请求使用了同样的端口，将很难区分管理进程和代理进程。

25.2 参考25.7节中的“表访问（Table Access）”部分。

第26章

26.1 我们预期从服务器来的第 2、4和9报文段将被延迟。第 2和第4报文段之间的时间差是190.7 ms，第2和第9报文段之间的时间差是400.7 ms。

从客户到服务器的所有确认看起来都被延迟：第 6、11、13、15、17和19报文段。从第6报文段开始的最后5个时间差是400.0、600.0、800.0、1000.0和2.600 ms。

26.2 如果连接的一个端点处于TCP的紧急模式，每次收到一个报文段，就会发送一个报文段。这个报文段没有告诉接收者任何新的东西（例如，它不是对新数据的确认），它也不包含数据，它只是重申进入了紧急模式。

26.3 只有512个这样的保留端口（ 512~1023），限制了一个主机只能有 512个远程登录的（Rlogin）客户。在实际生活中，这个限制一般小于 512个，因为在这个范围中的一些端口号被不同的服务器，如远程登录服务器，用作了知名端口。

TCP的限制是一对插口定义的一个连接（ 4元组）必须是唯一的。因为 Rlogin服务器总是使用了同样的知名端口（513），一台主机上多个Rlogin客户只有在它们和不同的服务器主机相连接时才可以使用相同的保留端口。然而， Rlogin客户没有采用重用保留端口的技术。如果使用了这种技术，理论限制就是在同一时刻最多有 512个Rlogin客户和同一个的服务器主机相连。

第27章

27.1 当一对插口处于 2MSL等待另一端状态时，理论上不能建立连接。然而，实际上，在18.6节中我们看到大多数伯克利演变的实现确实为一个处于 TIME_WAIT状态的连接接受了一个新的SYN。

27.2 这些行不是以3个数字作为应答代码开始的服务器应答的一部分，因此它们不可能来自于服务器。

第28章

28.1 一个域文字 (domain literal) 是在一对方括号里的点分十进制 IP地址。例如： `mail`

Linux公社 www.linuxidc.com

rstevens@[140.252.1.54]。

28.2 6个来回：HELO命令、MAIL、RCPT、DATA报文主体和QUIT。

28.3 这是合法的，称为流水线技术(pipelining) [Rose 1993，4.4.4节]。不幸的是，有一些脑子坏了（brain-damaged）的SMTP接收者实现，每处理完一条命令就要清除它们的输入缓存，使得这种技术不可用。如果使用了这种技术，客户自然不能丢弃报文直到所有的应答都已检查过，确信报文已被服务器接受了。

28.4 考虑习题28.2的前5个网络上的往返。每个都是一个小命令（很可能是只有一个报文段），对网络几乎没有负载。如果所有 5个都没有重传地送到了服务器，当报文主体发送时，拥塞窗口可能是6个报文段。如果报文主体很大，客户可能一次发送前 6个报文段，造成网络可能来不及处理。

28.5 更新版本的BIND使用同样的值来正移MX记录，作为平衡负载的一种方式。

第29章

29.1 不，因为tcpdump不能从其他UDP数据报中区别RPC请求或应答。它只有在源端口号或目的端口号为2049时，才将UDP数据报的内容理解为NFS分组。随机的RPC请求和应答可以使用两个端点上的一个临时的端口号。

29.2 回忆一下1.9节中，一个进程必须有超级用户权限才能给自己分配一个小于 1024的端口号（一个知名端口）。尽管这对于系统提供的服务器没问题，如 Telnet服务器、FTP服务器、和端口映射器，但我们不想给所有的 RPC服务器提供这个权限。

29.3 这个例子中的两个概念是客户忽略那些服务器应答，如果这些应答不具有客户正在等待的XID；UDP对收到的数据报进行排队（队列长度有一个上限），直到应用进程读取了数据报。另外，XID不会在超时和重传时改变，它只在调用另一个服务器过程时改变。

客户stub执行的事件为：时刻 0：发送请求1；时刻4：超时并重传请求 1；时刻5：接收服务器应答1，将应答返回给应用程序；时刻 5：发送请求2；时刻9：超时并重传请求2；时刻10：收到服务器的应答 1，但因为我们在等待应答 2，所以忽略它；时刻 11：收到服务器的应答2，将应答返回给应用程序。

服务器的事件如下：时刻 0：收到请求1，启动操作；时刻5：发送应答1；时刻5：收到请求1（来自于客户在时刻 4的重传），启动操作；时刻 10：发送应答1；时刻10：收到请求2（来自于客户在时刻 5的重传），启动操作；时刻 11：发送应答2；时刻11：收到请求2（来自于客户在时刻 9的重传），启动操作；时刻 12：发送应答2。这个最后的服务器应答仅仅被客户的 UDP排队，用于客户的下一次接收操作。当客户读了这个应答时，XID将是错误的，客户将忽略它。

29.4 改变服务器的以太网卡就改变了它的物理地址。即使我们注意到在 4.7节中SVR4没有在引导时发送一个免费 ARP，它仍然必须在能够应答 sun的NFS请求之前，向sun发送一个请求sun的物理地址的ARP请求。因为sun已经有了svr4的一个ARP登记项，它从这个ARP请求中根据发送者的（新）硬件地址更新这个登记项。

29.5 客户块I/O守护进程的第2个（在偏移 73728处读的）与第1个失去同步大约 0.74秒。即在行131~145，第2个守护进程在第 1个之后超时0.74秒。看来服务器没有看到在 167行的请求，但它看到了 168行的请求。第2个块 I/O守护进程只会在 168行之后0.74秒才会重传。

Linux公社 www.linuxidc.com

同时，第1个块I/O守护进程继续发送请求。

29.6 如果使用的是TCP，包含着服务器应答的TCP报文段在网络中丢失了，当服务器的TCP模块没有从客户的TCP模块收到一个确认时，它将超时，并重传应答。最终，这个报文段将到达客户的TCP。这里不同的是两个TCP模块完成超时和重传，而不是NFS客户和服务器（当使用UDP时，NFS客户代码完成超时和重传）。因此，NFS客户并不知道应答丢失了，需要被重传。

29.7 NFS服务器在重启动之后获得一个不同的端口号是可能的。这将使客户变得很复杂，因为它需要知道服务器崩溃了，并且在服务器重启动之后与服务器主机的端口映射器联系以找到NFS服务器的新的端口号。

这种情况，即服务器主机崩溃然后重启动时，一个服务器的RPC应用程序获得一个新的临时性端口，可能发生在任何一个没有使用知名端口的RPC应用程序上。

29.8 不。NFS客户可以为不同的服务器主机使用相同的本地的保留端口号。TCP要求由本地IP地址、本地端口、远端IP地址和远端端口组成的4元组是唯一的，对于每个服务器主机来说，远端的IP地址是不同的。

第30章

30.1 键入`whois "net 88"`。A类网络号64~95是保留的。

30.2 键入`whois whitehouse-d.o`可以使用`host`命令或者`nslookup`查询DNS。

30.3 不，`xscope`可以与服务器运行在不同的主机上。如果主机不同，`xscope`也可以使用TCP端口6000作为它的呼入连接。

Linux公社 www.linuxidc.com

附录E　配 置 选 项

我们已经看到了许多冠以“依赖于具体配置”的 TCP/IP特征。典型的例子包括是否使能UDP的检验和（11.3节），具有同样的网络号但不同的子网号的目的 IP地址是本地的还是非本地的（18.4节）以及是否转发直接的广播（12.3节）。实际上，一个特定的TCP/IP实现的许多操作特征都可以被系统管理员修改。

这个附录列举了本书中用到的一些不同的 TCP/IP实现可以配置的选项。就像你可能想到的，每个厂商都提供了与其他实现不同的方案。不过，这个附录给出的是不同的实现可以修改的参数类型。一些与实现联系紧密的选项，如内存缓存池的低水平线，没有描述。

> 这些描述的变量只用于报告的目的。在不同的实现版本中，它们的名字、默认值、或含义都可以改变。所以你必须检查你的厂商的文档（或向他们要更充分的文档）来了解这些变量实际使用的单词。

这个附录没有覆盖每次系统引导时发生的初始化工作：对每个网络接口使用 `ifconfig` 进行初始化（设置IP地址、子网掩码等等）、往路由表中输入静态路由等等。这个附录集中描述了影响TCP/IP操作的那些配置选项。

E.1　BSD/386 版本1.0

这个系统是自从4.2BSD以来使用的“经典”BSD配置的一个例子。因为源代码是和系统一起发布的，所以管理员可以指明配置选项，内核也可重编译。存在两种类型的选项：在内核配置文件中定义的常量（参见 `config(8)`手册）和在不同的C源文件中的变量初始化。大胆而又经验丰富的管理员也可以使用排错工具修改正在运行的内核或者内核的磁盘映像中这些变量的值，以避免重新构造内核。

下面列出的是在内核配置文件中可以修改的常量。

IPFORWARDING

这个常量的值初始化内核变量 `ipforwarding`。如果值为0（默认），就不转发IP数据报。如果是1，就总是使能转发功能。

GATEWAY

如果定义了这个常量，就使得 IPFORWARDING的值被置为1。另外，定义这个常量还使得特定的系统表格（ARP快速缓存表和路由表）更大。

SUBNETSARELOCAL

这个常量的值初始化内核变量 `subnetsarelocal`。如果值为1（默认），一个和发送主机具有同样网络号、但不同子网号的目的 IP地址被认为是本地的。如果是 0，只有在同一个子

Linux公社 www.linuxidc.com

网的目的IP地址才认为是本地的。图E-1总结了上述规律。

网络标识符	子网标识符	subnetsarelocal		注释
		1	0	
相同	相同	本地	本地	总是本地的
相同	不同	本地	非本地	依赖于配置
不同		非本地	非本地	总是非本地的

图E-1 对subnetarelocal内核变量的理解

这个变量的值影响了TCP选择的MSS。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS。而发送给一个非本地的地址时，TCP使用变量 `tcp_mssdflt` 作为MSS。

IPSENDREDIRECTS

这个常量的值初始化内核变量 `ipsendredirects`。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

DIRECTED_BROADCAST

如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是0，这些数据报就会被丢弃。

下面的变量也可以改变，它们在目录 /usr/src/sys/netinet中的不同文件中定义。

`tcprexmtthresh`

引起快速重传和快速恢复算法的连续ACK的数目。默认值是3。

`tcp_ttl`

TCP段的TTL字段的默认值。默认值是60。

`tcp_mssdflt`

用于非本地目的地址的默认的TCP MSS。默认值是512。

tcp_keepidle

在发送一个 `keepalive`探测报文之前必须等待的500 ms时钟间隔的次数。默认值是14 400（2个小时）。

`tcp_keepintvl`

如果没有收到响应，在两个连续的 `keepalive`探测报文之间等待的500 ms时钟间隔的次数。默认值是150（75秒）。

`tcp_sendspace`

TCP发送缓存的默认大小。默认值是4096。

`tcp_recvspace`

TCP接收缓存的默认大小。这个值影响了提供的窗口大小。默认值是4096。

`udpcksum`

如果非0，对输出的UDP数据报计算UDP检验和，并且对于包含了非0检验和的输入UDP数据报要验证它们的检验和。如果值为0，不计算输出的UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。默认值是1。

Linux公社 www.linuxidc.com

udp_ttl
UDP数据报TTL字段的默认值。默认值是30。
udp_sendspace
UDP发送缓存的默认大小。定义了可以发送最大的UDP数据报。默认值是9126。
udp_recvspace
UDP接收缓存的默认大小。默认值是41 600，允许40个1024字节的数据报。

E.2 SunOS 4.1.3

SunOS 4.1.3使用的方法类似于我们在BSD/386中看到的。因为大部分的内核源代码都没有发布，所以所有的C变量初始化都包含在一个提供的C源文件中。

管理员的内核配置文件（参见config(8)手册）可以定义下面的变量。修改了配置文件之后，需要构造一个新的内核，然后重启动。

IPFORWARDING

这个常量的值初始化内核变量ip_forwarding。如果值为-1，就不转发IP数据报，而且变量的值不能再改变。如果是0（默认），不转发IP数据报，但是如果多个接口都工作，变量的值可以修改为1。如果是1，就总是能转发IP数据报。

SUBNETSARELOCAL

这个常量的值初始化内核变量ip_subnetsarelocal。如果值为1（默认），一个和发送主机具有同样网络号，但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子网的目的IP地址才认为是本地的。图E-1总结了上述规律。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS，而发送给一个非本地的地址时，TCP使用变量tcp_default_mss作为MSS。

IPSENDREDIRECTS

这个常量的值初始化内核变量ip_sendredirects。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

DIRECTED_BROADCAST

这个常量的值初始化内核变量ip_dirbroadcast。如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是0，这些数据报就会被丢弃。

文件/usr/kvm/sys/netinet/in_proto.c定义了下面一些可以修改的变量。一旦修改了这些变量，必须构造一个新的内核，然后重启动。

tcp_default_mss
用于非本地地址的默认TCP MSS。默认值是512。
tcp_sendspace
TCP发送缓存的默认大小。默认值是4096。

Linux公社 www.linuxidc.com

`tcp_recvspace`

TCP接收缓存的默认大小。这个值影响了提供的窗口大小。默认值是 4096。

`tcp_keeplen`

一个发往4.2BSD主机的keepalive探测报文必须包含一个字节的数据来得到一个响应。把这个变量的值设置为1是为了兼容于以前的实现。默认值是 1。

`tcp_ttl`

TCP段的TTL字段的默认值。默认值是60。

`tcp_nodelack`

如果非0，对ACK不做延迟。默认值是0。

`tcp_keepidle`

在发送一个keepalive探测报文之前必须等待的 500 ms时钟间隔的次数。默认值是 14 400（2个小时）。

`tcp_keepintvl`

如果没有收到响应，在两个连续的 keepalive探测报文之间等待的500 ms时钟间隔的次数。默认值是150（75秒）。

`udp_cksum`

如果非0，对输出的UDP数据报计算UDP检验和，并且对于包含了非 0检验和的输入UDP数据报要验证它们的检验和。如果值为 0，不计算输出UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。默认值是 0。

`udp_ttl`

UDP数据报TTL字段的默认值。默认值是60。

`udp_sendspace`

UDP发送缓存的默认大小。定义了可以发送最大的 UDP数据报。默认值是9000。

`udp_recvspace`

UDP接收缓存的默认大小。默认值是 18 000，允许两个9000字节的数据报。

E.3 SRV4

SVR4的TCP/IP配置类似于前两个系统，但可用的选项更少。在文件`/etc/conf/pack.d/ip/space.c5`可以定义两个常量，然后必须重新构造内核并且重启动。

IPFORWARDING

这个常量的值初始化内核变量 `ipforwarding`。如果是0（默认），不转发IP数据报。如果是1，就总是能转发IP数据报。

IPSENDREDIRECTS

这个常量的值初始化内核变量 `ipsendredirects`。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

前两节中，我们描述的许多变量在内核中都有定义，但必须修补内核来改变它们。例如，存在一个名为`tcp_keepidle`的变量，它的值是14 400。

Linux公社 www.linuxidc.com

E.4 Solaris 2.2

Solaris 2.2是较新的Unix系统的典型代表，它为管理员提供了一个可以改变 TCP/IP系统配置选项的程序。这样可以不必通过修改源文件和重新构造内核来进行配置。

配置程序是ndd(1)。我们可以运行程序，看看在UDP模块中可以检验和修改的参数：

```
solaris % ndd /dev/udp \?
udp_wroff_extra                 读、写
udp_def_ttl                     读、写
udp_first_anon_port             读、写
udp_trust_optlen                读、写
udp_do_checksum                 读、写
udp_status                      只读
```

我们可以指明 5个模块：/dev/ip、/dev/icmp、/dev/arp、/dev/udp和/dev/tcp。问号参数（为了防止外壳程序解释问号，我们在它前面加了一个反斜线）告诉ndd程序列出那个模块的所有参数。查询一个变量的值的例子是：

```
solaris %ndd /dev/tcp tcp_mss_def
536
```

为了修改一个变量的值，我们需要有超级用户的权限，输入：

```
solaris #ndd -set /dev/ip ip_forwarding 0
```

这些变量可以划分为三种类型：

1) 系统管理员可以修改的配置变量（如， ip_forwarding）。

2) 只能显示的状态变量（如， ARP快速缓存）。这个信息一般通过命令 ifconfig，netstat和arp以一种更好理解的格式提供。

3) 用于内核源代码的排错变量。使能一些这种变量可以在运行时产生内核的排错输出，当然这会降低系统的性能。

现在我们可以描述每个模块的参数了。所有的参数如果没有注明“（只读）”，就是可读写的。只读的参数是上面第 2种情况的状态变量。我们对于第 3种情况的变量注明了“（排错）”。如果不另外说明，所有的计时变量都以毫秒指明，这和其他系统不同，其他系统一般以 500 ms时钟间隔的次数来指明时间。

/dev/ip

ip_cksum_choice

（排错）在IP检验和算法的两个独立实现之中选择一个。

ip_debug

（排错）如果大于0，使能内核打印排错信息功能。值越大输出的信息越多。默认为 0。

ip_def_ttl

如果运输层没有指明，指定输出IP数据报默认的TTL。默认值是255。

ip_forward_directed_broadcasts

如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是 0，这些数据报就会被丢弃。

ip_forward_src_routed

如果为1（默认），就转发包含一个源路由选项的接收数据报。如果为 0，这些数据报将被

Linux公社 www.linuxidc.com

丢弃。

ip_forwarding

指明系统是否转发进入的IP数据报：0表示不转发，1表示总是转发，2（默认）表示只有当两个或两个以上接口都工作时才转发。

ip_icmp_return_data_bytes

一个ICMP差错返回的除了IP首部以外的数据字节的数目，默认是64。

ip_ignore_delete_time

（排错）一个IP路由表项（IRE）最小的生命期。默认是30秒（这个参数以秒记，不是毫秒）。

ip_ill_status

（只读）显示每个IP下层数据结构的状态。每个接口存在一个下层数据结构。

ip_ipif_status

（只读）显示每个IP接口数据结构的状态（IP地址、子网掩码等等）。每个接口存在一个这种结构。

ip_ire_cleanup_interval

（排错）扫描IP路由表，删除过时表项的时间间隔。默认是30 000 ms（30秒）。

ip_ire_flush_interval

从IP路由表中无条件地刷新ARP信息的间隔。默认是1200 000 ms（20分钟）。

ip_ire_pathmtu_interval

路径MTU发现算法尝试增加MTU的间隔。默认是30 000 ms（30秒）。

ip_ire_redirect_interval

来自ICMP重定向的IP路由表项被删除的间隔。默认是60 000 ms（60秒）。

ip_ire_status

（只读）显示所有的IP路由表项。

ip_local_cksum

如果为0（默认），IP不为通过环回接口发送和接收的数据报计算IP检验和或者更高层的检验和（即TCP、UDP、ICMP或IGMP）。如果为1，就要计算这些检验和。

ip_mrtdebug

（排错）如果为1，使能内核打印多播路由的排错输出。默认是0。

ip_path_mtu_discovery

如果为1（默认），IP执行路径MTU发现。如果是0，IP不会在输出的数据报中设置“不分片”比特。

ip_respond_to_address_mask

如果为0（默认），主机不响应ICMP的地址掩码请求。如果为1，主机则响应。

ip_respond_to_echo_broadcast

如果为1（默认），主机响应发往一个广播地址的ICMP回显请求。如果为0，则不响应。

ip_respond_to_timestamp

如果为0（默认），主机不响应ICMP的时间戳请求。如果为1，则响应。

ip_respond_to_timestamp_broadcast

如果为0（默认），主机不响应发往一个广播地址的ICMP时间戳请求。如果为1，则响应。

Linux公社 www.linuxidc.com

`ip_rput_pullups`

（排错）来自于网络接口驱动程序的缓存数目的计数，它需要增长以访问整个 IP首部。引导时它被初始化为0，并且可以被复位为0。

`ip_send_redirects`

如果为1（默认），当主机作为一个路由器时，它发送ICMP重定向。如果为0，则不发送。

`ip_send_source_quench`

如果为1（默认），当输入的数据报被丢弃时，主机生成 ICMP源抑制差错。如果为 0，则不生成这种差错。

`ip_wroff_extra`

（排错）在缓存中为IP首部分配的额外空间的字节数。默认是 32。

/dev/icmp

`icmp_bsd_compat`

（排错）如果为1（默认），收到的数据报的IP首部的长度字段的值被调整为不包括 IP首部的长度。这和伯克利演变的实现是一致的，用于读原始的 IP或原始的ICMP分组的应用程序。如果为0，则不改变长度字段的值。

`icmp_def_ttl`

输出ICMP报文的默认的TTL。默认值为255。

`icmp_wroff_extra`

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

/dev/arp

`arp_cache_report`

（只读）ARP的快速缓存。

`arp_cleanup_interval`

ARP登记项从ARP快速缓存中被删除的时间间隔。默认是 300 000 ms（5分钟）（IP为完成的ARP传输维护着它自己的快速缓存；参见 `ip_ire_flush_interval`）。

`arp_debug`

（排错）如果为1，使能打印ARP驱动程序的排错输出。默认是 0。

/dev/udp

`udp_def_ttl`

输出UDP数据报的默认的TTL。默认值是255。

`udp_do_checksum`

如果为1（默认），为输出的UDP数据报计算UDP检验和。如果为0，输出的UDP数据报不包含一个检验和（和其他大多数的实现不一样，这个 UDP检验和标志并不影响进入的数据报。如果一个接收到的数据报有一个非 0的检验和，它总是要被验证）。

`udp_largest_anon_port`

可以为UDP临时端口分配的最大端口号。默认是 65535。

Linux公社 www.linuxidc.com

`udp_smallest_anon_port`

可以为UDP临时端口分配的最小端口号。默认是32768。

`udp_smallest_nonpriv_port`

一个进程需要超级用户的权限才能给自己分配一个小于这个值的端口号。默认是 1024。

`udp_status`

（只读）所有本地的UDP端点的状态：本地IP地址和端口，远端IP地址和端口。

`udp_trust_optlen`

（排错）不再使用。

`udp_wroff_extra`

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

/dev/tcp

`tcp_close_wait_interval`

2MSL的值：在TIME_WAIT状态花费的时间。默认是240 000 ms（4分钟）。

`tcp_conn_grace_period`

（排错）当发送一个SYN时，在定时器间隔上附加的时间。默认是 500 ms。

`tcp_conn_req_max`

在一个监听的端口上挂起的连接请求的最大数目。默认是 5。

`tcp_cwnd_max`

拥塞窗口的最大值。默认是32768。

`tcp_debug`

（排错）如果为1，使能打印TCP的排错输出。默认是0。

`tcp_deferred_ack_interval`

在发送一个延迟的ACK之前等待的时间。默认是50 ms。

`tcp_dupack_fast_retransmit`

引起快速重传、快速恢复算法的连续的重复 ACK的数目。默认是3。

`tcp_eager_listeners`

（排错）如果为1（默认），TCP在将一个新的连接返回给一个挂起的被动打开的应用程序之前需要进行三次握手。这是大多数的 TCP实现采用的方式。如果为 0，TCP将呼入连接请求（收到的SYN）传递给应用程序，并不完成三次握手直到该应用程序接受了这个连接（把这个值置为0可能引起很多已经存在的应用程序不能用）。

`tcp_ignore_path_mtu`

（排错）如果为1，路径MTU发现算法忽略接收到的需要 ICMP分段的报文。如果为 0（默认），使能TCP的路径MTU发现。

`tcp_ip_abort_cinterval`

当TCP进行一个主动打开时，整个重传超时的值。默认是 240 000 ms（4分钟）。

`tcp_ip_abort_interval`

一个TCP连接建立以后，整个重传超时的值。默认是 120 000 ms（2分钟）。

`tcp_ip_notify_cinterval`

Linux公社 www.linuxidc.com

当TCP正在进行一个主动打开时，TCP通知IP去寻找一条新路由超时的值。默认是10 000 ms（10秒）。

`tcp_ip_notify_interval`

TCP为一个已经建立的连接通知IP去寻找一条新路由超时的值。默认是10 000 ms（10秒）。

`tcp_ip_ttl`

用于输出TCP段的TTL。默认为255。

`tcp_keepalive_interval`

在发出一个keepalive探测报文之前，一个连接保持空闲状态的时间。默认为 7200000 ms（2小时）。

`tcp_largest_anon_port`

为TCP临时端口分配的最大端口号。默认为 65535。

`tcp_maxpsz_multiplier`

（排错）指明了报文流首部将应用程序写的数据分装成几个 MSS。默认是1。

`tcp_mss_def`

非本地的目的地址的默认的MSS。默认是536。

`tcp_mss_max`

最大的MSS。默认为65495。

`tcp_mss_min`

最小的MSS。默认为1。

`tcp_naglim_def`

（排错）每个连接的Nagle算法阈值的最大值。默认是 65535。每个连接的值以MSS的最小值或这个值开始。TCP_NODELAY插口选项将每个连接的值设置为 1，以禁止Nagle算法。

`tcp_old_urp_interpretation`

（排错）如果为1（默认），采用紧急指针的一个以前的（但更常见的） BSD的理解：它指向紧急数据最后一个字节后的一个字节。如果为 0，采用主机需求RFC理解：它指向紧急数据的最后一个字节。

`tcp_rcv_push_wait`

（排错）在把接收数据传递给应用程序之前，可以缓存的没有设置 PUSH标志的数据的最大字节数。默认是16384。

`tcp_rexmit_interval_initial`

（排错）初始的重传超时间隔。默认是 500 ms。

`tcp_rexmit_interval_max`

（排错）最大的重传超时间隔。默认是 60 000 ms（60秒）。

`tcp_rexmit_interval_min`

（排错）最小的重传超时间隔。默认是 200 ms。

`tcp_rwin_credit_pct`

（排错）在对每个接收的段进行流量控制检查之前，必须达到的接收缓存窗口的百分比。默认是50%。

`tcp_smallest_anon_port`

Linux公社 www.linuxidc.com

分配给TCP临时端口的开始端口号。默认是32768。

`tcp_smallest_nonpriv_port`

一个进程需要有超级用户的权限才能给自己分配一个小于这个值的端口号。默认是 1024。

`tcp_snd_lowat_fraction`

（排错）如果非0，发送缓存的低水平线是发送缓存的大小除以这个值。默认是 0（禁止）。

`tcp_status`

（只读）所有TCP连接的信息。

`tcp_sth_rcv_hiwat`

（排错）如果非0，把报文流首部的高水平线设置为这个值。默认为 0。

`tcp_sth_rcv_lowat`

（排错）如果非0，把报文流首部的低水平线设置为这个值。默认为 0。

`tcp_wroff_xtra`

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

E.5 AIX 3.2.2

AIX3.2.2允许在运行时使用no命令设置网络选项。它可以显示一个选项的值，设置一个选项的值，或者将一个选项的值设置为默认。例如，显示一个选项，我们键入：

```
aix % no -o udp_ttl
udp_ttl = 30
```

下面的选项可以被修改。

`arpt_killc`

在删除一个不活动的、完成的ARP项之前等待的时间（以分钟计）。默认是20。

`ipforwarding`

如果为1（默认），总是转发IP数据报。如果为0，则禁止转发。

`ipfragttl`

等待重新装配的IP数据报片的生存时间（单位为秒）。默认是60。

`ipsendredirects`

如果为1（默认），当转发IP数据报时，主机将发送 ICMP重定向。如果为 0，则不发送ICMP重定向。

`loop_check_sum`

如果为1（默认），对通过环回接口发送的数据报计算 IP检验和。如果为0，则不计算这个检验和。

`nonlocsrcroute`

如果为1（默认），就转发包含一个源路由选项的接收数据报。如果为0，就丢弃这些数据报。

`subnetsarelocal`

如果值为1（默认），一个和发送主机具有同样网络号，但不同子网号的目的 IP地址被认为是本地的。如果是 0，只有在同一个子网的目的 IP地址才认为是本地的。图 E-1总结了上述规律。当给一个本地的目的地址发送报文时， TCP选择的是基于输出接口的 MTU的MSS。当发送给一个非本地的地址时， TCP使用默认（536）作为MSS。

`tcp_keepidle`

Linux公社 www.linuxidc.com

在发送一个keepalive探测报文之前等待的500 ms时间段的倍数。默认值是14 400（2小时）。

`tcp_keepintvl`

如果没有收到响应，在发送下一个 keepalive探测报文之前等待的 500 ms时间段的倍数。默认值是150（75秒）。

`tcp_recvspace`

TCP接收缓存的默认大小。它影响了提供的窗口大小。默认值是 16 384。

`tcp_sendspace`

TCP发送缓存的默认大小。默认是 16 384。

`tcp_ttl`

TCP报文段TTL字段的默认值。默认值是60。

`udp_recvspace`

UDP接收缓存的默认大小。默认值是 41 600，允许40个1024字节数据报。

`udp_sendspace`

UDP发送缓存的默认大小。定义了可以发送的最大的 UDP数据报。默认是9216。

`udp_ttl`

UDP数据报TTL字段的默认值。默认值是30。

E.6 4.4BSD

4.4BSD是第一个为多个内核参数提供动态配置的伯克利版本。使用 sysctl（8）命令。参数的名字看起来就像SNMP的MIB变量的名字。查看一个参数，我们键入：

```
vangogh %sysctl net.inet.ip.forwarding
net.inet.ip.forwarding = 1
```

要修改一个参数的值需要有超级用户的权限，键入：

```
vangogh #sysctl -w net.inet.ip.ttl=128
```

可以修改下面的参数。

`net.inet.ip.forwarding`

如果为0（默认），就不转发IP数据报。如果为1，则使能转发功能。

`net.inet.ip.redirect`

如果为1（默认），当转发 IP数据报时，主机将发送 ICMP重定向。如果为 0，则不发送ICMP重定向。

`net.inet.ip.ttl`

TCP和UDP默认的TTL。默认值是64。

`net.inet.icmp.maskrepl`

如果为0（默认），主机不响应ICMP地址掩码请求。如果为1，则响应。

`net.inet.udp.checksum`

如果为1（默认），对输出的UDP数据报计算UDP检验和，并且对于包含了非0检验和的呼入UDP数据报要校验它们的检验和。如果值为 0，不计算输出UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。

另外，在本附录前面部分描述的许多变量分散在几个不同的源文件中（`tcp_keepidle`、`subnetarelocal`等等），它们也可以被修改。

Linux公社 www.linuxidc.com

附录F　可以免费获得的源代码

本书中使用了很多共享软件包。本附录提供了一些如何获得这些软件的细节。

用来获得共享软件的技术称为匿名 FTP，FTP是标准的Internet文件传输协议（第 27章）。27.3节显示了一个匿名FTP的例子。如果想了解Internet资源的一般背景和匿名FTP的知识，请参考任何一本关于Internet的书，如[LaQuey 1993] 或 [Krol 1992]。

这里列出的主机被认为是提供共享软件的主要站点。其他许多站点也提供这些软件。可以通过Internet的Archie服务查找其他的软件版本。下面列出的软件版本是本书中使用到的版本。

当你读到这本书时，可能已经发布了更新版本的软件。

你应该使用FTP的dir命令来看一下在指定的主机上是否存在更新的版本。

本附录按照资源在本书中出现的章节号进行排序。

RFC（1.11节）

1.11节提供了请求得到RFC站点的电子邮件地址。应答中包含了许多可以使用电子邮件或匿名FTP获得RFC的站点。

记住你要先得到一个当前的索引，在索引中查找想要的 RFC。这个RFC项中还告诉你这个RFC是否废弃了或是被一个更新的RFC替代了。

BSD Net/2 源代码（1.14节）

BSD Net/2 源代码，其中包括了TCP/IP协议的内核实现和标准的应用程序（Telnet客户和服务器、FTP客户和服务器等），从主机`ftp.uu.net`的目录树中以`system/unix/bsd-source`开始的位置可以得到。

SLIP（2.4节）

本书中使用的SLIP的版本来自于`ftp.ee.lbl.gov`。文件名以`cslip`开始，因为它提供了压缩的SLIP（2.5节）。

icmpaddrmask程序（6.3节）

参见本附录的最后一项。

icmptime程序（6.4节）

参见本附录的最后一项。

ping程序（第7章）

BSD版本的`ping`程序一般比其他厂商提供的版本具有更多的选项和特征。主机`ftp.uu.net`

Linux公社 www.linuxidc.com

的文件system/unix/bsd-sources/sbin/ping中包含了最新版本的ping程序。

traceroute程序（第8章）

traceroute程序来自于主机ftp.ee.lbl.gov。本附录的最后一项提供了8.5节使用的允许不严格的和严格的源站选路的版本。

路由器发现守护程序（9.6节）

可用的一个程序为路由器发现报文提供了主机支持和路由器支持。主机是gregorio.stanford.edu，文件是gw-discovery/nordmark-rdisc.tar。这个程序是Sun微系统公司开发的一个共享软件。

gated守护程序（10.3节）

在10.3节提到的gated路由选择守护程序在主机gated.cornell.edu上可以得到。

traceroute.pmtu程序（11.7节）

参见本附录的最后一项。

IP多播软件（第13章）

为SunOS 4.x和Ultrix提供IP多播的修改程序在主机gregorio.stanford.edu的目录vmtp-ip中可以得到。这个目录中还包含了为伯克利Unix系统提供IP多播的修改源程序。

BIND名字服务器（第14章）

BIND名字服务器，即named守护程序，在主机ftp.uu.net的文件networking/ip/dns/bind/bind.4.8.3.tar.z中。

一个更新的版本，即4.9版，从主机gatekeeper.dec.com的目录pub/BSD/bind/4.9中可以得到。

host程序（第14章）

host程序在主机nikhefh.nikhef.nl的文件host.tar.z中提供。

dig和doc程序（第14章）

第14章中提到的dig程序和doc程序在主机isi.edu的文件dig.2.0.tar.Z和doc.2.0.tar.Z中提供。

BOOTP服务器（第26章）

常用的Unix BOOTP服务器的不同版本在主机lancaster.andrew.cmu.edu的pub目录中提供。

TCP快速扩充（第24章）

TCP窗口扩缩选项、时间戳选项和PAWS算法的一个共享源代码作为BSD Net/2版的一组

Linux公社 www.linuxidc.com

修改程序在主机uxc.cso.uiuc.edu的文件pub/tcplw.shar.z中提供。

ISODE SNMP管理进程和代理进程（第25章）

25.7节中描述的SNMP管理者进程和代理进程是ISODE 8.0版本的一部分。很多站点都可以得到它，如ftp.uu.net上的目录networking/osi/isode中。

MIME软件和实例（28.4节）

一个名为MetaMail的程序为很多不同的用户代理提供了MIME的能力。这个程序可以从主机thumper.bellcore.com的pub/nsb目录中得到。在这个目录中还有MIME的其他信息。

Sun RPC（29.2节）

RPC 4.0版本的源代码（使用插口API）在主机ftp.uu.net的目录systems/sun/sextape/rpc4.0中提供。TI-RPC版本的源代码（使用了TLI API）在主机ftp.uu.net的目录networking/rpc中提供。

Sun NFS（第29章）

一个NFS客户和服务器的共享软件的实现作为BSD Net/2源代码的一部分提供，在本附录的前面部分介绍了。

tcpdump程序（附录A）

本书中使用的tcpdump的版本来自于主机ftp.ee.lbl.gov上的文件tcpdump-2.2.1.tar.z。

BSD分组过滤器（A.1节）

BSD分组过滤器是tcpdump发布的一部分。

sock程序（附录C）

参见本附录的最后一项。

ttcp程序

（这个程序本书中没有用到，但它是一个读者应该注意的常用的工具）。ttcp是测量两个系统之间TCP和UDP性能的基准工具。它是美国军事弹道研究实验室开发的，是一个共享软件。它的副本可以从很多匿名FTP站点获得，但一个增强的版本可以从主机ftp.sgi.com的目录sgi/src/ttcp中获得。

作者编写的软件

本书中用到的作者编写的软件在主机ftp.uu.net的文件published/books/stevens.tcpipiv1.tar.z中提供。

Linux公社 www.linuxidc.com

参 考 文 献

所有的RFC都可以如同在1.11节描述的那样在Internet上通过电子邮件或使用匿名FTP免费获得。

Albitz, P., and Liu, C. 1992. *DNS and BIND*. O'Reilly & Associates, Sebastopol, Calif.
配置和运行一个名字服务器所需要的管理任务的许多细节。

Alexander, S., and Droms, R. 1993. "DHCP Operations and BOOTP Vendor Extensions," RFC1533, 30 pages(Oct.).

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC1349, 28 pages (July).
怎样使用IP首部的服务类型字段。

Almquist, P., ed. 1993. "Requirements for IP Routers," Internet Draft (Mar.).
这是代替RFC 1009[Braden and Postel 1987]的一个RFC的草稿。新的RFC将很可能以四卷出现。卷1：Internet的结构、术语和一般性内容。卷2：链路层、互联网层、运输层和应用层。卷3：转发和选路协议。卷4：操作、维护和网络管理。
这个草稿可以通过匿名FTP从主机jessica.stanford.edu的目录rreq中获得。一旦最终的RFC发布就要忽略这个草稿。

Bellovin, S. M. 1993. Private Communication.

Bhide, A., Elnozahy, E. N., and Morgan, S. P. 1991. "A Highly Available Network File Server," *Proceeding of the 1991 Winter USENIX Conference*, pp. 199-205, Dallas, Tex.
描述了一个对免费ARP的使用（4.7节）。

Borenstein, N., and Freed, N. 1993. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," RFC 1521, 81 pages (Sept.).
这个RFC废止了早期的RFC 1341。这个RFC的附录H列出了与RFC 1341的不同点。

Borman, D. A., ed. 1990. "Telnet Linemode Option," RFC 1184, 23 pages (Oct.).

Borman, D. A. 1991. "IP Bandwidth Limits," Message-ID 91011437.AA17276@berserkly.cray.com>, Usenet, comp.protocols.tcp-ip Newsgroup (Jan.).
说明了我们在24.8节的最后部分列出的有关TCP性能的三个限制。

Borman, D. A. 1992. "TCP/IP Performance at Cray Research," *Proceeding of the Twenty-third Internet Engineering Task Force*, pp. 492 - 493 (Mar.), San Diego Supercomputer Center, San Diego, Calif.

Borman, D. A., ed. 1993a. "Telnet Environment Option," RFC 1408, 7 pages (Jan.).
从客户向服务器传递环境变量的Telnet选项。

Borman, D. A. 1993b. "A Practical Perspective on Host Networking," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 309-367. Addison-Wesley, Reading, Mass.
对于Host Requirements（主机需求）RFC（1122和1123）的一个实际的审视。

Linux公社 www.linuxidc.com

Braden, R. T., ed. 1989a. “Requirements for Internet Hosts —— Communication Layers,” RFC1122, 116 pages (Oct.).
主机需求RFC的前半部分。这部分覆盖了链路层，IP，TCP和UDP。

Braden, R. T., ed. 1989b. “Requirements for Internet Hosts —— Application and Support,” RFC1123, 98 pages (Oct.).
主机需求RFC的后半部分。这部分覆盖了Telnet，FTP，TFTP，SMTP和DNS。

Braden, R. T. 1989c. “Perspective on the Host Requirements RFCs,” RFC 1127, 20 pages (Oct.).
对于开发主机需求RFC的IETF工作组的讨论和结果的一个非正式的总结。

Braden, R. T. 1992a. “TIME-WAIT Assassination Hazards in TCP,” RFC1337, 11 pages (May).
显示处于TIME_WAIT状态时，接收一个RST如何导致问题。

Braden, R. T. 1993b. “Extending TCP for Transactions —— Concepts,” RFC 1379, 38 pages (Nov.).
开发T/TCP背后的概念和历史。

Braden, R. T. 1992c. “Extending TCP for Transactions —— Functional Specification,” Internet Draft, 32 pages (Dec.).
T/TCP的功能说明和有关实现的讨论。

Braden, R. T., Borman, D. A., and Partridge, C. 1988. “Computing the Internet Checksum,” RFC 1071, 24 pages (Sept.).
提供计算IP、ICMP、IGMP、UDP和TCP的检验和的技术和算法。

Braden, R. T., and Postel, J. B. 1987. “Requirements for Internet Gateways,” RFC1009, 55 pages (June).
等效于路由器的主机需求RFC。这个RFC已经被代替了；见 [Almquist 1993]。

Caceres, R., Danzig, P. B., Jamin, S., and Mitzel, D. J. 1991. “Characteristics of Wide-Area TCP/IP Conversations,” *Computer Communication Review*, vol. 21, no. 4, pp. 101-112 (Sept.).

Callon, R. 1992. “TCP and UDP with Bigger Addresses (TUBA), A Simple Proposal for Internet Addressing and Routing,” RFC 1347, 9 pages (June).

Case, J. D., Fedor, M. S., Schoffstall, M. L., and Davin, C. 1990. “Simple Network Management (SNMP),” RFC 1157, 36 pages (May).
SNMP的协议规范。

Case, J. D., McCloghrie, K., Rose, M. T., and Waldbusser, S. 1993. “An Introduction to Version 2 of the Internet-Standard Network Management Framework,” RFC 1441, 13 pages (Apr.).
一个SNMPv2的介绍和其他11个定义SNMPv2的RFC的引用。

Case, J. D., and Partridge, C. 1989. “Case Diagrams: A First Step to Diagrammed Management Information Bases,” *Computer Communication Review*, vol. 19, no. 1, pp. 13-16 (Jan.).
定义了用来可视化一个给定模块中SNMP变量之间关系的图形表示。

Casner, S., and Deering, S. E. 1992. “First IETF Internet Audiocast,” *Computer Communication Review*, vol. 22, no. 3, pp. 92-97 (July).

Linux公社 www.linuxidc.com

描述了怎样使用Internet的多播技术实时传递一个IETF会议的声音信息。这篇文章的一个PostScript的 副本可以通过匿名FTP从主机venera.isi.edu上的文件pub/ietf-autocast-article.ps中获得。另外，那台主机上的文件 mbone/faq.txt包含了有关Internet多播骨干（MBONE）常问的问题。

Cheriton, D. P. 1988. “VMTP: Versatile Message Transaction Protocol,” RFC 1045, 123 pages (Feb.).

Cheswick, W. R., and Bellovin, S. M. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, Mass.

描述了怎样建立和管理一个防火墙网关以及涉及的安全问题。

Clark, D. D. 1982. “Window and Acknowledgement Strategy in TCP,” RFC 813, 22 pages (July).

标识了糊涂窗口综合症以及如何避免它的原始RFC。

Clark, D. D. 1988. “The Design Philosophy of the DARPA Internet Protocols,” *Computer Communication Review*, vol. 18, no. 4, pp. 106-114 (Aug.).

描述了影响Internet协议的早期的推理技术。

Comer, D. E., and Stevens, D. L. 1993. *Internetworking with TCP/IP: Vol. III: Client-Server Programming and Applications, BSD Socket Version*. Prentice-Hall, Englewood Cliffs, N.J.

Cooper, A. W., and Postel, J. B. 1993. “The US Doman,” RFC 1480, 47 pages (June).

描述了DNS的.us域。

Crocker, D. H. 1982. “Standard for the Format of ARPA Internet Text Messages,” RFC 822, 47 pages (Aug.).

定义了使用SMTP传输的电子邮件的格式。

Crocker, D. H. 1993. “Evolving the System,” in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 41-76. Addison-Wesley, Reading, Mass.

讲述了开发ARPANET标准的一些历史和Internet技术共同体当前结构的细节，还定义了当前Internet标准形成的过程。

Croft, W., and Gilmore, J. 1985. “Bootstrap Protocol (BOOTP),” RFC 951, 12 pages (Sept.).

Crowcroft, J., Wakeman, I., Wang, Z., and Sirovica, D. 1992. “Is Layering Harmful?,” *IEEE Network*, vol. 6, no. 1, pp. 20-24 (Jan.).

这篇文章中漏掉的7张图刊登在下一期，卷6，第2期（三月）。

Curry, D. A. 1992. *UN LX System Security: A Guide for Users and System Administrators*. Addison-Wesley, Reading, Mass.

一本关于Unix安全性的书。第4章和第5章讲述网络安全。

Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. 1993. “After-burner,” *IEEE Network*, vol. 7, no. 4, pp. 36-43 (July).

描述了怎样通过减少数据复制的次数来加快TCP，以及一个支持这种设计的专用接口卡。

Danzig, P. B., Obraczka, K., and Kumar, A. 1992. “An Analysis of Wide-Area Name Server Traffic,” *Computer Communication Review*, vol. 22, no. 4, pp. 281-292 (Oct.).

对根名字服务器之一进行的两个 24小时流量监视的分析。显示来自于一个故障实现的

Linux公社 www.linuxidc.com

DNS流量如何消耗了20倍于正常的WAN带宽。这篇文章的一个PostScript副本通过匿名FTP在主机**caldera.usc.edu**的文件**pub/danzig/dns.ps.z**中提供。

Deering, S. E. 1989. “Host Extensions for IP Multicasting,” RFC 1112, 17 pages (Aug.).
IP多播和IGMP的规范。

Deering, S. E., ed. 1991. “ICMP Router Discovery Messages,” RFC 1256, 19 pages (Sept.).

Deering, S. E., and Cheriton, D. P. 1990. “Multicast Routing in Datagram Internetworks and Extended LANs,” *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85-110 (May).
对于常用路由技术的支持多播传输的扩充。

Dixon, T. 1993. “Comparison of Proposals for Next Version of IP,” RFC 1454, 15 pages (May).
SIP, PIP 和TUBA的比较和总结。

Droms, R. 1993. “Dynamic Host Configuration Protocol,” RFC 1541, 39 pages (Oct.).

Droms, R., and Dyksen, W. R. 1990. “Performance Measurements of the X Window System Communication Protocol,” *Software Practice & Experience*, vol. 20, pp. 119-136 (Oct.).
使用不同的X客户时涉及到的TCP通信的测量。

Fedor, M. S. 1988. “GATED: A Multi-routing Protocol Daemon for Unix,” *Proceeding of the 1988 Summer USENIX Conference*, pp. 365-376, San Francisco, Calif.

Finlayson, R. 1984. “Bootstrap Loading using TFTP,” RFC 906, 4 pages (June).

Finlayson, R., Mann, T., Mogul, J. C., and Theimer, M. 1984. “A Reverse Address Resolution Protocol,” RFC 903, 4 pages (June).

Floyd, S. 1994. Private Communication.

Ford, P. S., Rekhter, Y., and Braun, H-W. 1993. “Improving the Routing and Addressing of IP,” *IEEE Network*, vol. 7, no. 3, pp. 10-15 (May).
对于CIDR（无类型域间路由）的描述。

Fuller, V., Li, T., Yu, J.Y., and Varadhan, K. 1993. “Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy,” RFC 1519, 24 pages (Sept.).
CIDR（无类型域间选路）的规范。

Gerich, E. 1993. “Guidelines for Management of IP Address Space,” RFC 1466, 10 pages (May).
将来怎样分配IP地址的说明（即B类地址将很难获得，作为替代手段，一般将分配一组C类地址）。

Gurwitz, R., and Hinden, R. 1982. “IP —— Local Area Network Addressing Issues,” IEN 212, 11 pages (Sept.)。
对IP广播地址最早的引用之一。

Harrenstein, K., Stahl, M. K., and Feinler, E. J. 1985. “NICNAME/WHOIS,” RFC 954, 4 pages (OCT.).

Hedrick, C. L. 1988a. “Routing Information Protocol,” RFC 1058, 33 pages (June).

Hedrick, C. L. 1988b. “Telnet Terminal Speed Option,” RFC 1079, 3 pages (Dec.).

Hedrick, C. L., and Borman, D. A. 1992. “Telnet Remote Flow Control Option,” RFC 1372, 6 pages (Oct.).

Linux公社 www.linuxidc.com

Hornig, C. 1984. “Standard for the Transmission of IP Datagrams over Ethernet Networks,” RFC 894, 3 pages (Apr.).

Huitema, C. 1993. “IAB Recommendation for an Intermediate Strategy to Address the Issue of Scaling,” RFC 1481, 2 pages (July).
实现CIDR的IAB建议。

Jacobson, V. 1988. “Congrestion Avoidance and Control,” *Computer Communication Review*, vol. 18, no. 4, pp. 314-329 (Aug.).
描述TCP的慢启动和拥塞避免算法的一篇经典文章。这篇文章的一个 PostScript副本通过匿名FTP在主机`ftp.ee.lbl.gov`的文件`congavoid.ps.z`中提供。

Jacobson, V. 1990a. “Compressing TCP/IP Headers for Low-Speed Serial Links,” RFC 1144, 43 pages (Feb.).
描述CSLIP, 一个压缩TCP和IP首部的SLIP版本。

Jacobson, V. 199b. “Modified TCP Congestion Avoidance Algorithm,” April 30, 1990, end2end-interest mailing list (Apr.).
描述了快速重传和快速恢复算法。

Jacobson, V. 1990c. “Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno,” *Proceeding of the Eighteenth Internet Engineering Task Force*, p. 365 (Sept.), University of British Columbia, Vancouver, B.C.

Jacobson, V., and Braden, R. T. 1988. “TCP Extensions for Long-Delay Paths,” RFC 1072, 16 pages (Oct.).
描述了TCP的选择确认选项，在后来的RFC 1323中这个选项已经被删去了。

Jacobson, V. Braden, R. T., and Borman, D. A. 1992. “TCP Extensions for High Performance,” RFC 1323, 37 pages (May).
描述了窗口缩放选项、时间戳选项和PAWS算法，以及需要这些改变的理由。

Jacobson, V., Braden, R. T., and Zhang, L. 1990. “TCP Extensions for High-Speed Paths,” RFC 1185, 21 pages (Oct.).
尽管这个RFC已经被RFC 1323废止了，但有关防止TCP旧的重复的报文段的附录仍然值得一读。

Juszczak, C. 1989. “Improving the Performance and Correctness of an NFS Server,” *Proceedings of the 1989 Winter USENIX Conference*, pp. 53-63, San Diego, Calif.
提供了NFS服务器快速缓存的实现细节。

Kantor, B. 1991. “BSD Rlogin,” RFC 1282, 5 pages (Dec.).
Rlogin协议的规范。

Karn, P., and Partridge, C. 1987. “Improving Round-Trip Time Estimates in Reliable Transport Protocols,” *Computer Communication Review*, vol. 17, no. 5, pp. 2-7 (Aug.).
处理已经重传报文段的重传超时的 Karn算法的细节。这篇文章的一个 PostScript副本通过匿名FTP在主机sics.se的文件pub/craig/karn-partridge.ps中提供。

Katz, D. 1990. “Proposed Standard for the Transmission of IP Datagrams Over FDDI Networks,” RFC 1188, 11 pages (Oct.).

Linux公社 www.linuxidc.com

说明了在FDDI网络上如何封装IP数据报和ARP请求和应答，包括多播技术。

Kent, C. A., and Mogul, J. C. 1987. "Fragmentation Considered Harmful," *Computer Communication Review*, vol. 17, no. 5, pp. 390-401 (Aug.).

Kent, S. T. 1991. "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108, 17 pages (Nov.).

Kleinrock, L. 1992. "The Latency / Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications Magazine*, vol. 30, no. 4, pp. 36-40 (Apr.).

Klensin, J., Freed, N., and Moore, K. 1993. "SMTP Service Extension for Message Size Declaration," RFC 1427, 8 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993a. "SMTP Service Extensions," RFC 1425, 10 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993b. "SMTP Service Extension for 8bit-MIME Transport," RFC 1426, 6 pages (Feb.).

Krol, E. 1992. *The Whole Internet*. O'Reilly & Associates, Sebastopol, Calif.
关于Internet介绍的入门书。

LaQuey, T. 1993. *The Internet Companion: A Beginner's Guide to Global Networking*. Addison-Wesley, Reading, Mass.
关于Internet介绍的入门书。

Leffler, S. J., and Karels, M. J. 1984. "Trailer Encapsulations," RFC 893, 3 pages (Apr.).

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Mass.
一本完整介绍4.3BSD Unix系统的书。这本书描述了Tahoe版的4.3BSD。

Lougheed, K., and Rekhter, Y. 1991. "A Border Gateway Protocol 3 (BGP-3)," RFC 1267, 35 pages (Oct.).

Lynch, D. C. 1993. "Historical Perspective," *in Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 3-14. Addison-Wesley, Reading, Mass.
描述了Internet的前身：ARPANET。

Macklem, R. 1991. "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol," *Proceedings of the 1991 Winter USENIX Conference*, pp. 53-64, Dallas, Tex.
描述了一个同时使用TCP和UDP的NFS实现。

Malkin, G. S. 1993a. "RIP Version 2: Carrying Additional Information." RFC 1388, 7 pages (Jan.).

Malkin, G. S. 1993b. "Traceroute Using an IP Option," RFC 1393, 7 pages (Jan.).
建议用于traceroute新版本的ICMP修改。

Mallory, T., and Kullberg, A. 1990. "Incremental Updating of the Internet Checksum," RFC 1141, 2 pages (Jan.).
描述了递增改变Internet检验和的实现技术。

Manber, U. 1990. "Chain Reactions in Networks," *IEEE Computer*, vol. 23, no. 10, pp. 57-63 (Oct.).

Linux公社 www.linuxidc.com

描述了广播风暴和网络崩溃的类型，类似于习题 9.3和9.4所显示的。

McCanne, S., and Jacobson, V. 1993. “The BSD Packet Filter: A New Architecture for User-Level Packet Capture,” *Proceeding of the 1993 Winter USENIX Conference*. Pp. 259-269, San Diego, Calif.

BSD分组过滤器（BPF）的具体描述以及与 Sun的网络接口栓（NIT）的比较。这篇文章的一个PostScript副本通过匿名FTP在主机ftp.ee.lbl.gov的文件papers/bpf-usenix93.ps.Z中提供。

McCloghrie, K., and Rose, M. T. 1991. “Management Information Base for Network Management of TCP/IP-based Internets: MIB-II,” RFC 1213 (Mar.).

McGregor, G. 1992. “PPP Internet Protocol Control Protocol (IPCP),” RFC 1332, 12 pages (May).

用于TCP/IP的PPP的NCP的描述。

Mills, D. L. 1992. “Network Time Protocol (Version 3): Specification, Implementation, and Analysis,” RFC 1305, 113 pages (Mar.).

Mockapetris, P. V. 1987a. “Domain Names: Concepts and Facilities,” RFC 1034, 55 pages (Nov.).

一个DNS的介绍。

Mockapetris, P. V. 1987b. “Domain Names: Implementation and Specification,” RFC 1035, 55 pages (Nov.).

DNS规范。

Mogul, J. C. 1990. “Efficient Use of Workstations for Passive Monitoring of Local Area Networks,” *Computer Communication Review*, vol. 20, no. 4, pp. 253-263 (Sept.).

描述了如何使用工作站监视局域网，而不购买专门的网络分析硬件产品。

Mogul, J. C. 1992. “Holy Turbocharger Batman, (evil cheating), NFS async writes,” Message-ID 1992Mar2.191711.9935@PA.dec.com, Usenet, comp.protocols.nfs Newsgroup (Mar.).

在一个繁忙的NFS服务器上收集的，40天内Internet检验和错误的一些有趣的统计数据。

Mogul, J. C. 1993. “IP Network Performance,” in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 575-675. Addison-Wesley, Reading, Mass.

包含了许多TCP/IP协议栈的主题，它们可以用来获得最优的性能。

Mogul, J. C., and Deering, S. E. 1990. “Path MTU Discovery,” RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. “Internet Standard Subnetting Procedure,” RFC 950, 18 pages (Aug.).

Moore, K. 1993. “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text,” RFC 1522, 10 pages (Sept.).

描述了使用7 bit ASCII在RFC 822邮件首部发送非ASCII字符的一种方法。

Moy, J. 1991. “OSPF Version 2,” RFC 1247, 189 pages (July).

Nagle, J. 1984. “Congestion Control in IP/TCP Internetworks,” RFC 896, 9 pages(Jan.).

Nagle算法的描述。

Nye, A., ed. 1992. *The X Window System, Volume 0: X Protocol Reference Manual, Third*

Linux公社 www.linuxidc.com

Edition. O'Reilly & Associates, Sebastopol, Calif.

Obracezka, K., Danzig, P. B., and Li, S. 1993. "Internet Resource Discovery Services," *IEEE Computer*, vol. 26, no. 9, pp. 8-22 (Sept.).

对当前Internet上资源发现工具的一个综述：Alex、Archie、Gopher、Indie、Knowbot信息服务、Netfind、Prospero、WAIS、WWW和X.500。这篇文章的一个PostScript副本通过匿名FTP在主机`caldera.usc.edu`的文件`/pub/kobraczk/ieeecomputer.ps.z`中提供。

Papadopoulos, C., and Parulkar, G. M. 1993. "Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199-216 (Apr.).

测量当发送数据和接收数据时在协议族的不同层中引入的开销。

Partridge, C. 1986. "Mail Routing and the Domain System," RFC 974, 7 pages (Jan.).

怎样使用DNS的MX记录进行邮件的选路。

Partridge, C. 1994. *Gigabit Networking*. Addison-Wesley, Reading, Mass.

描述当网络速率超过1 Gb/s时产生的问题。

Partridge, C., and Pink, S. 1993. "A Faster UDP." *IEEE/ACM Transactions on Networking*, Vol. 1, no. 4, pp. 429-440 (Aug.).

描述了对伯克利实现的改进，可以将UDP的性能提高30%。

Paxson, V. 1993. "Empirically-Derived Analytic Models of Wide-Area TCP Connections: Extended Report," LBL-34086, Lawrence Berkeley Laboratory and EECS Division, University of California, Berkeley (June).

包含了在14个广域流量跟踪中250万个TCP连接的分析。这个报告的一个PostScript副本通过匿名FTP在主机`ftp.ee.lbl.gov`的文件`WAN-TCP-models.1.ps.Z`和`WAN-TCP-models.2.ps.Z`中提供。

Perlman, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Mass.

一本包含了许多详细的网络互连方法（桥和路由器）和不同的路由算法的书。

Plummer, D. C. 1982. "An Ethernet Address Resolution Protocol," RFC 826, 10 pages (Nov.).

Postel, J. B. 1980. "User Datagram Protocol," RFC 768, 3 pages (Aug.).

Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792, 21 pages (Sept.).

Postel, J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.).

Postel, J. B. 1982. "Simple Mail Transfer Protocol," RFC 821, 68 pages (Aug.).

Postel, J. B. 1987. "TCP and IP Bake Off," RFC 1025, 6 pages (Sept.).

描述了在早期开发过程中，为了测试互连性在TCP/IP的不同实现之间进行的一些测试过程和记录。

Postel, J. B., ed. 1994. "Internet Official Protocol Standards," RFC 1600, 36 pages (Mar.).

所有的Internet协议的状态。这个RFC定期改变——查看最近的RFC索引以了解当前的版本。

Postel, J. B., and Reynolds, J. K. 1983a. "Telnet Protocol Specification," RFC 854, 15 pages (May).

Linux公社 www.linuxidc.com

基本的Telnet协议规范。很多以后的RFC描述了特定的Telnet选项。

Postel, J. B., and Reynolds, J. K. 1983b. “Telnet Binary Transmission,” RFC 856, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1983c. “Telnet Echo Option,” RFC 857, 5 pages (May).

Postel, J. B., and Reynolds, J. K. 1983d. “Telnet Suppress Go Ahead Option,” RFC 858, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983e. “Telnet Status Option,” RFC 859, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983f. “Telnet Timing Mark Option,” RFC 860, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1985. “File Transfer Protocol (FTP),” RFC 959, 69 pages (Oct.).

Postel, J. B., and Reynolds, J. K. 1988. “Standard for the Transmission of IP Datagrams over IEEE 802 Networks,” RFC 1042, 15 pages (Apr.).

在IEEE 802网络上封装IP数据报和ARP请求和应答的规范。

Pusateri, T. 1993. “IP Multicast Over Token-Ring Local Area Networks,” RFC 1469, 4 pages (June).

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, Mass.

一本关于TLI和流子系统的书。

Rekhter, Y. and Gross, P. 1991. “Application of the Border Gateway Protocol in the Internet,” RFC 1268, 13 pages (Oct.).

Rekhter, Y., and Li, T. 1993. “An Architecture for IP Address Allocation with CIDR,” RFC 1518, 27 pages (Sept.).

Reynolds, J. K. 1989. “The Helminthiasis of the Internet,” RFC 1135, 33 pages (Dec.).

包含了1988年Internet蠕虫的详细讨论。

Reynolds, J. K., and Postel, J. B. 1992. “Assigned Numbers,” RFC 1340, 138 pages (July).

Internet协议族所有的编码。这个RFC定期改变——查看最近的RFC索引以了解当前的版本。

Romkey, J. L. 1988. “A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP,” RFC 1055, 6 pages (June).

Rose, M. T. 1990. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于OSI协议的书。第8章提供了ASN.1和BER的细节。

Rose, M. T. 1993. *The Internet Message: Closing the Book with Electronic Mail*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于Internet邮件的书，包含MIME的细节。

Rose, M. T. 1994. *The Simple Book: An Introduction to Internet Management, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于SNMPv2的书。这本书的第1版介绍了SNMPv1。

Rose, M. T., and McCloghrie, K. 1990. “Structure and Identification of Management

Linux公社 www.linuxidc.com

Information for TCP/IP-based Internets,” RFC 1155, 22 pages (May).
定义了SNMPv1的SMI。

Rosenberg, W., Kenney, D., and Fisher, G. 1992. *Understanding DCE*. O’Reilly & Associates, Sebastopol, Calif.
提供了OSF的分布式计算环境的概述。

Routhier, S. A. 1993. “Implementation Experience for SNMPv2,” *The Simple Times*, vol. 2, no. 4, pp. 1-4 (July-Aug.).
描述了对SNMPv1的修改以支持SNMPv2。
这个期刊是免费电子发行的。以主题“help”向st-subscriptions@simple-times.org发送一个电子邮件以获得订阅信息。

Schryver, V. J. 1993. “Info on High Speed Transport Protocols Requested,” Message-ID i0imr8g@rhyolite.wpd.sgi.com, Usenet, comp.protocols.tcp-ip Newsgroup (May).
提供了一些FDDI实现上的TCP性能数字。

Schwartz, M. F., and Tsirigotis, P. G. 1991. “Experience with a Semantically Cognizant Internet White Pages Directory Tool,” *Journal of Internetworking Research and Experience*, vol. 2, no. 1, pp. 23-50 (Mar.).
也可以通过匿名FTP在主机ftp.cs.colorado.edu的文件pub/cs/techreports/schwartz/PostScript/ White.Pages.ps.Z中得到。

Simpson, W. A. 1993. “The Point-to-Point Protocol (PPP),” RFC 1548, 53 pages (Dec.).
定义了PPP和它的链路控制协议。

Sollins, K. R. 1992. “The TFTP Protocol (Revision 2),” RFC 1350, 11 pages (July).

Stallings, W. 1987. *Handbook of Computer-Communications Standards, Volume* 2: *Local Network Standards*. Macmillan, New York.
包含了IEEE 802 局域网标准的细节。

Stallings, W. 1993. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards*. Addison-Wesley, Reading, Mass.
描述了SNMPv1和SNMPv2的差别。

Stern, H. 1991. *Managing NFS and NIS*. O’Reilly & Associates, Sebastopol, Calif.
包含了许多安装、使用和管理NFS的细节。

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, N.J.
一本详细介绍在Unix上使用插口和TLI进行网络程序设计的书。

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, Mass.
一本详细介绍Unix程序设计的书。

Sun Microsystems. 1987. “XDR: External Data Representation Standard,” RFC 1014, 20 pages (June).

Sun Microsystems. 1988a. “RFC: Remote Procedure Call, Protocol Specification, Version 2,” RFC 1057, 25 pages (June).

Sun Microsystems. 1988b. “NFS: Network File System Protocol Specification,” RFC 1094, 27 pages

Linux公社 www.linuxidc.com

(Mar.).
第2版Sun NFS的规范。

Sun Microsystems. 1994. *NFS: Network File System Version 3 Protocol Specification*. Sun Microsystems, Mountain View, Calif.
这个文档的一个 PostScript副本通过匿名 FTP在主机 ftp.uu.net的文件 networking/ip/nfs/NFS3.spec.ps.Z中提供。

Tanenbaum, A. S. 1989. *Computer Networks, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.
一本关于计算机网络的通用书。

Topolcic, C. 1993. “Status of CIDR Deployment in the Internet,” RFC 1467, 9 pages (Aug.).

Tsuchiya, P. F. 1991. “On the Assignment of Subnet Numbers,” RFC 1219, 13 pages (Apr.).
建议从最高位往下分配子网 ID，从最低位往上分配主机 ID。这样使得以后在某个点上改变子网掩码时不需要对所有的系统重新编号。

Ullmann, R. 1993. “TP/IX: The Next Internet,” RFC 1475, 35 pages (June).
下一代Internet协议的另一个建议。

VanBokkelen, J. 1989. “Telnet Terminal-Type Option,” RFC 1091, 7 pages (Feb.).

Waitzman, D. 1988. “Telnet Window Size Option,” RFC 1073, 4 pages (Oct.).

Waitzman, D., Partridge, C., and Deering, S. E. 1988. “Distance Vector Multicast Routing Protocol,” RFC 1075, 24 pages (Nov.).

Warnock, R. P. 1991. “Need Help Selecting Ethernet Cards for Very High Performance Throughput Rates,” Message-ID< lbhal10@sgi.sgi.com> Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).
提供了我们从图24-9计算的TCP的性能数据。

Weider, C., Reynolds, J. K., and Heker, S. 1992. “Technical Overview of Directory Services Using the X.500 Protocol,” RFC 1309, 16 pages (Mar.).

Wimer, W. 1993. “Clarifications and Extensions for the Bootstrap Protocol,” RFC 1542, 23 pages (Oct.).

X/Open. 1991. *Protocols for X/Open Internetworking: XNFS*. X/Open, Reading, Berkshire, U.K.
对Sun RPC、XDR和NFS的一个更好的描述。还包含了NFS锁定管理程序和状态监视协议的描述。附录中详细讨论了使用 NFS和使用本地文件访问之间的语义差别。 X/Open文档编号为XO/CAE/91/030。

Zimmerman, D. P. 1991. “Finger User Information Protocol,” RFC 1288, 12 pages (Dec.)

Linux公社 www.linuxidc.com

缩 略 语

ACK (ACKnowledgment) TCP首部中的确认标志
API (Application Programming Interface) 应用编程接口
ARP (Address Resolution Protocol) 地址解析协议
ARPANET
(Defense Advanced Research Project Agency NETwork) (美国)国防部远景研究规划局
AS (Autonomous System) 自治系统
ASCII (American Standard Code for Information Interchange) 美国信息交换标准码
ASN.1 (Abstract Syntax Notation One) 抽象语法记法1
BER (Basic Encoding Rule) 基本编码规则
BGP (Border Gateway Protocol) 边界网关协议
BIND (Berkeley Internet Name Domain) 伯克利Internet域名
BOOTP (BOOTstrap Protocol) 引导程序协议
BPF (BSD Packet Filter) BSD 分组过滤器
CIDR (Classless InterDomain Routing) 无类型域间选路
CIX (Commercial Internet Exchange) 商业互联网交换
CLNP (ConnectionLess Network Protocol) 无连接网络协议
CRC (Cyclic Redundancy Check) 循环冗余检验
CSLIP (Compressed SLIP) 压缩的SLIP
CSMA (Carrier Sense Multiple Access) 载波侦听多路存取
DCE (Data Circuit-terminating Equipment) 数据电路端接设备
DDN (Defense Data Network) 国防数据网
DF (Don’t Fragment) IP首部中的不分片标志
DHCP (Dynamic Host Configuration Protocol) 动态主机配置协议
DLPI (Data Link Provider Interface) 数据链路提供者接口
DNS (Domain Name System) 域名系统
DSAP (Destination Service Access Point) 目的服务访问点
DSLAM (DSL Access Multiplexer) 数字用户线接入复用器
DSSS (Direct Sequence Spread Spectrum) 直接序列扩频
DTS (Distributed Time Service) 分布式时间服务
DVMRP (Distance Vector Multicast Routing Protocol) 距离向量多播选路协议
EBONE (European IP BackbONE) 欧洲IP主干网
EOL (End of Option List) 选项清单结束
EGP (External Gateway Protocol) 外部网关协议
EIA (Electronic Industries Association) 美国电子工业协会

Linux公社 www.linuxidc.com

FCS (Frame Check Sequence) 帧检验序列
FDDI (Fiber Distributed Data Interface) 光纤分布式数据接口
FIFO (First In, First Out) 先进先出
FIN (FINish) TCP首部中的结束标志
FQDN (Full Qualified Domain Name) 完全合格的域名
FTP (File Transfer Protocol) 文件传送协议
HDLC (High-level Data Link Control) 高级数据链路控制
HELLO 选路协议
IAB (Internet Architecture Board) Internet体系结构委员会
IANA (Internet Assigned Numbers Authority) Internet号分配机构
ICMP (Internet Control Message Protocol) Internet控制报文协议
IDRP (InterDomain Routing Protocol) 域间选路协议
IEEE (Institute of Electrical and Electronics Engineering) （美国）电气与电子工程师协会
IEN (Internet Experiment Notes) 互联网试验注释
IESG (Internet Engineering Steering Group) Internet工程指导小组
IETF (Internet Engineering Task Force) Internet工程专门小组
IGMP (Internet Group Management Protocol) Internet组管理协议
IGP (Interior Gateway Protocol) 内部网关协议
IMAP (Internet Message Access Protocol) Internet报文存取协议
IP (Internet Protocol) 网际协议
IRTF (Internet Research Task Force) Internet研究专门小组
IS-IS (Intermediate System to Intermediate System Protocol) 中间系统到中间系统协议
ISN (Initial Sequence Number) 初始序号
ISO (International Organization for Standardization) 国际标准化组织
ISOC (Internet SOCiety) Internet协会
LAN (Local Area Network) 局域网
LBX (Low Bandwidth X) 低带宽X
LCP (Link Control Protocol) 链路控制协议
LFN (Long Fat Net) 长肥网络
LIFO (Last In, First Out) 后进先出
LLC (Logical Link Control) 逻辑链路控制
LSRR (Loose Source and Record Route) 宽松的源站及记录路由
MBONE (Multicast Backbone On the InterNEt) Internet上的多播主干网
MIB (Management Information Base) 管理信息库
MILNET (MILitary NETwork) 军用网
MIME (Multipurpose Internet Mail Extensions) 通用Internet邮件扩充
MSL (Maximum Segment Lifetime) 报文段最大生存时间
MSS (Maximum Segment Size) 最大报文段长度
MTA (Message Transfer Agent) 报文传送代理

Linux公社 www.linuxidc.com

MTU (Maximum Transmission Unit) 最大传输单元
NCP (Network Control Protocol) 网络控制协议
NFS (Network File System) 网络文件系统
NIC (Network Information Center) 网络信息中心
NIT (Network Interface Tap) 网络接口栓（Sun公司的一个程序）
NNTP (Network News Transfer Protocol) 网络新闻传送协议
NOAO (National Optical Astronomy Observatories) 国家光学天文台
NOP (No Operation) 无操作
NSFNET (National Science Foundation NETwork) 国家科学基金网络
NSI (NASA Science Internet) （美国）国家宇航局Internet
NTP (Network Time Protocol) 网络时间协议
NVT (Network Virtual Terminal) 网络虚拟终端
OSF (Open Software Foudation) 开放软件基金
OSI (Open Systems Interconnection) 开放系统互连
OSPF (Open Shortest Path First) 开放最短通路优先
PAWS (Protection Against Wrapped Sequence number) 防止回绕的序号
PDU (Protocol Data Unit) 协议数据单元
POSIX (Portable Operating System Interface) 可移植操作系统接口
PPP (Point-to-Point Protocol) 点对点协议
PSH (PuSH) TCP首部中的急迫标志
RARP (Reverse Address Resolution Protocol) 逆地址解析协议
RFC (Request For Comments) Internet的文档，其中的少部分成为标准文档
RIP (Routing Information Protocol) 路由信息协议
RPC (Remote Procedure Call) 远程过程调用
RR (Resource Record) 资源记录
RST (ReSeT) TCP首部中的复位标志
RTO (Retransmission Time Out) 重传超时
RTT (Round-Trip Time) 往返时间
SACK (Selective ACKnowledgment) 有选择的确认
SLIP (Serial Line Internet Protocol) 串行线路Internet协议
SMI (Structure of Management Information) 管理信息结构
SMTP (Simple Mail Transfer Protocol) 简单邮件传送协议
SNMP (Simple Network Management Protocol) 简单网络管理协议
SSAP (Source Service Access Point) 源服务访问点
SSRR (Strict Source and Record Route) 严格的源站及记录路由
SWS (Silly Window Syndrome) 糊涂窗口综合症
SYN (SYNchronous) TCP首部中的同步序号标志
TCP (Transmission Control Protocol) 传输控制协议
TFTP (Trivial File Transfer Protocol) 简单文件传送协议

Linux公社 www.linuxidc.com

TLI (Transport Layer Interface)　运输层接口
TTL (Time-To-Live)　生存时间或寿命
TUBA (TCP and UDP with Bigger Addresses)　具有更长地址的TCP和UDP
Telnet　远程终端协议
UA (User Agent)　用户代理
UDP (User Datagram Protocol)　用户数据报协议
URG (URGent)　TCP首部中的紧急指针标志
UTC (Coordinated Universal Time)　协调的统一时间
UUCP (Unix-to-Unix CoPy)　Unix到Unix的复制
WAN (Wide Area Network)　广域网
WWW (World Wide Web)　万维网
XDR (eXternal Data Representation)　外部数据表示
XID (transaction ID)　事务标识符
XTI (X/Open Transport Layer Interface)　X/Open运输层接口

Linux公社 www.linuxidc.com

欢迎点击这里的链接进入精彩的Linux公社 网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux公社
www.Linuxidc.com

Linux 公社微信公众号：linuxidc_com