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。

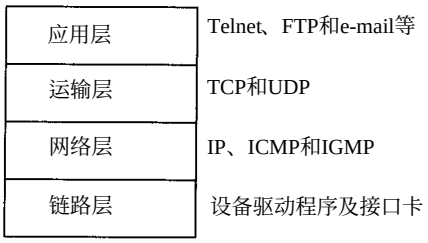

图1-1 TCP/IP协议族的四个层次

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：

- 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。

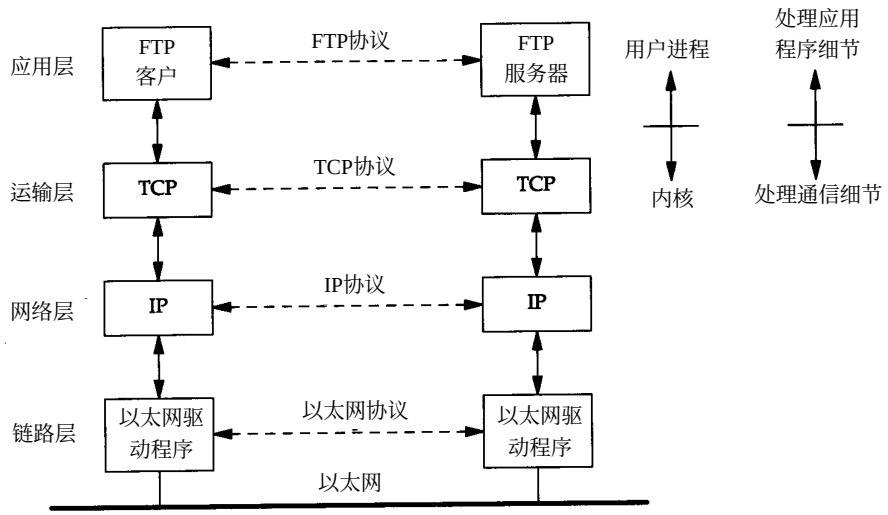

图1-2 局域网上运行FTP的两台主机

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。

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。

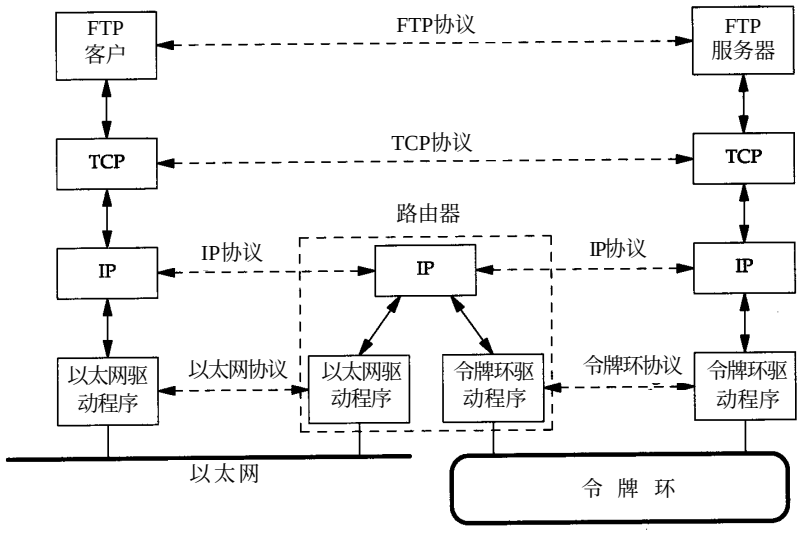

图1-3 通过路由器连接的两个网络

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）。

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。

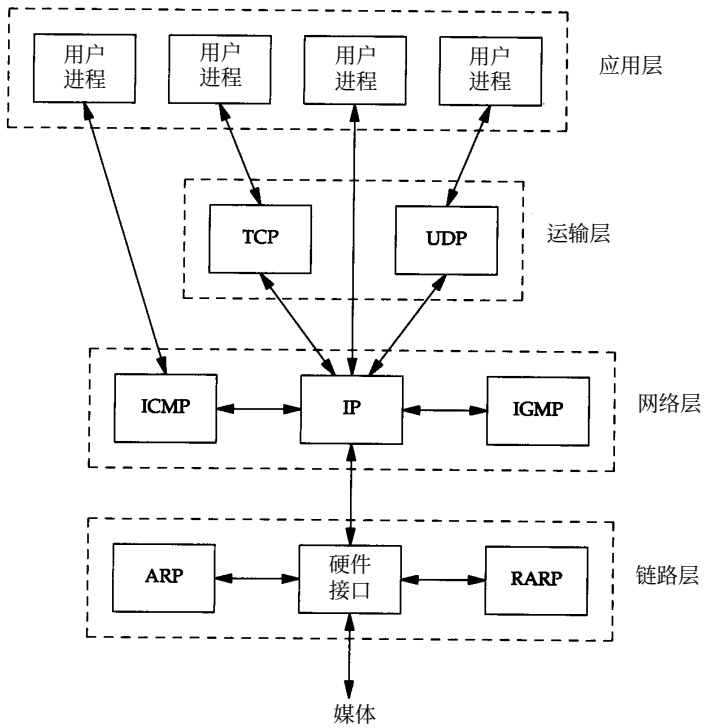

图1-4 TCP/IP协议族中不同层次的协议

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。

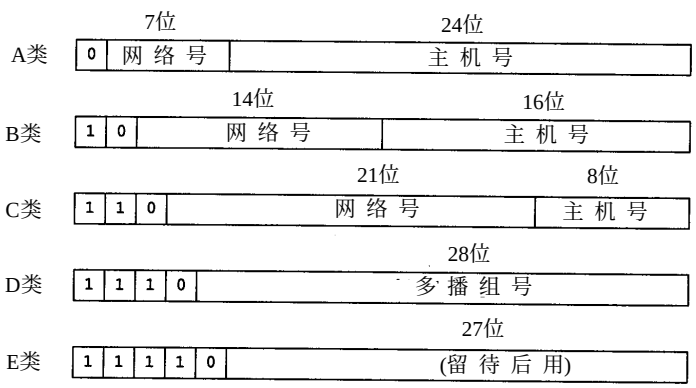

图1-5 五类互联网地址

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)。

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。

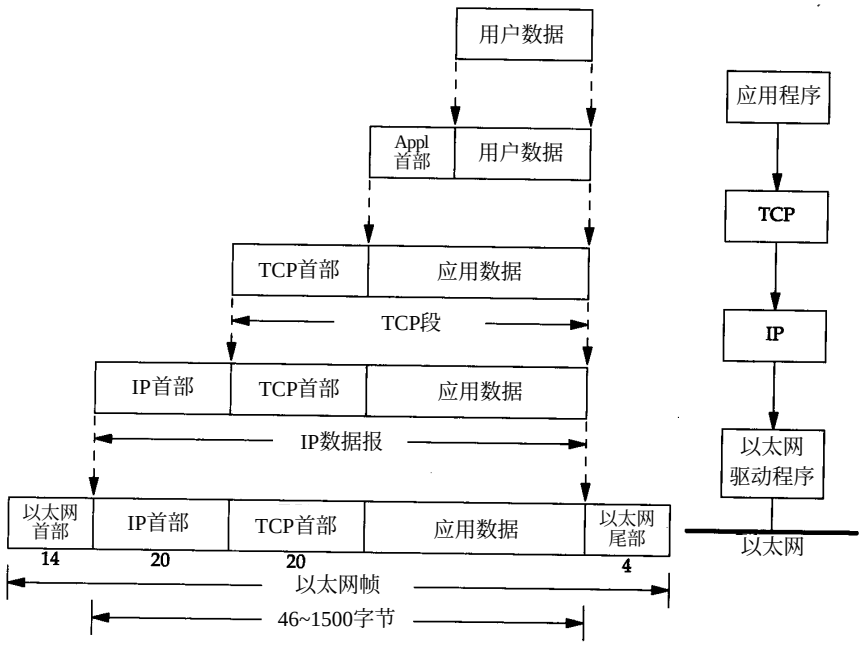

图1-7 数据进入协议栈时的封装过程

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

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。

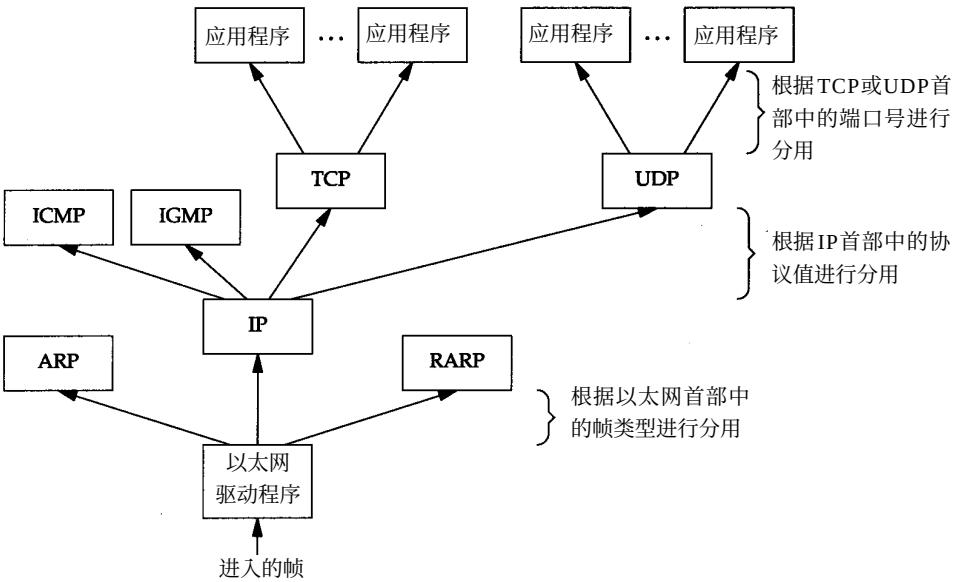

图1-8 以太网数据帧的分用过程

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：

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

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：

```
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

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。

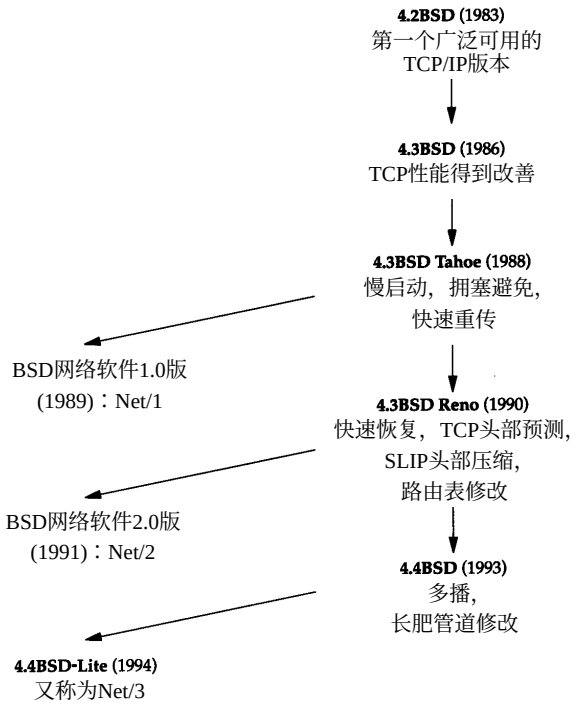

图1-10 不同的BSD版及其重要的TCP/IP特性

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

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。

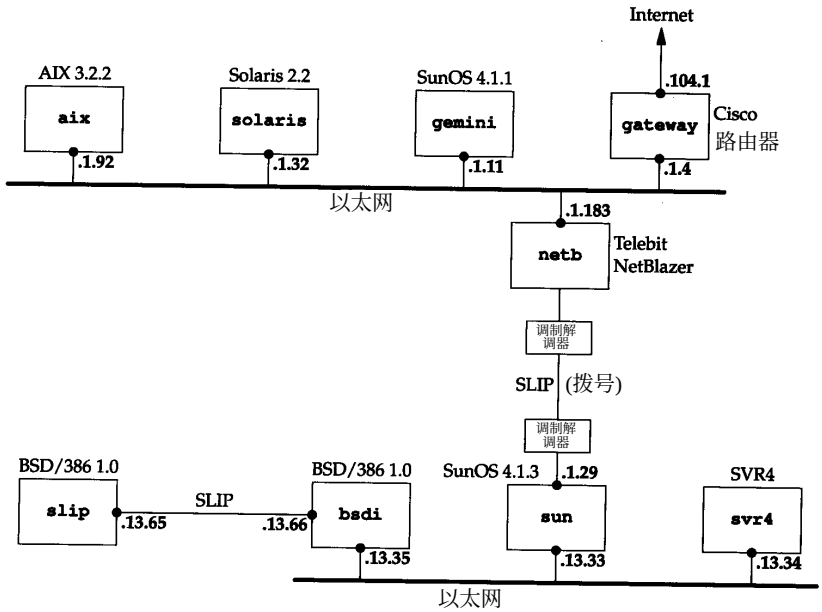

图1-11 本书中所有例子运行的测试网络，所有的IP地址均从140.252开始编址

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。

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`。

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

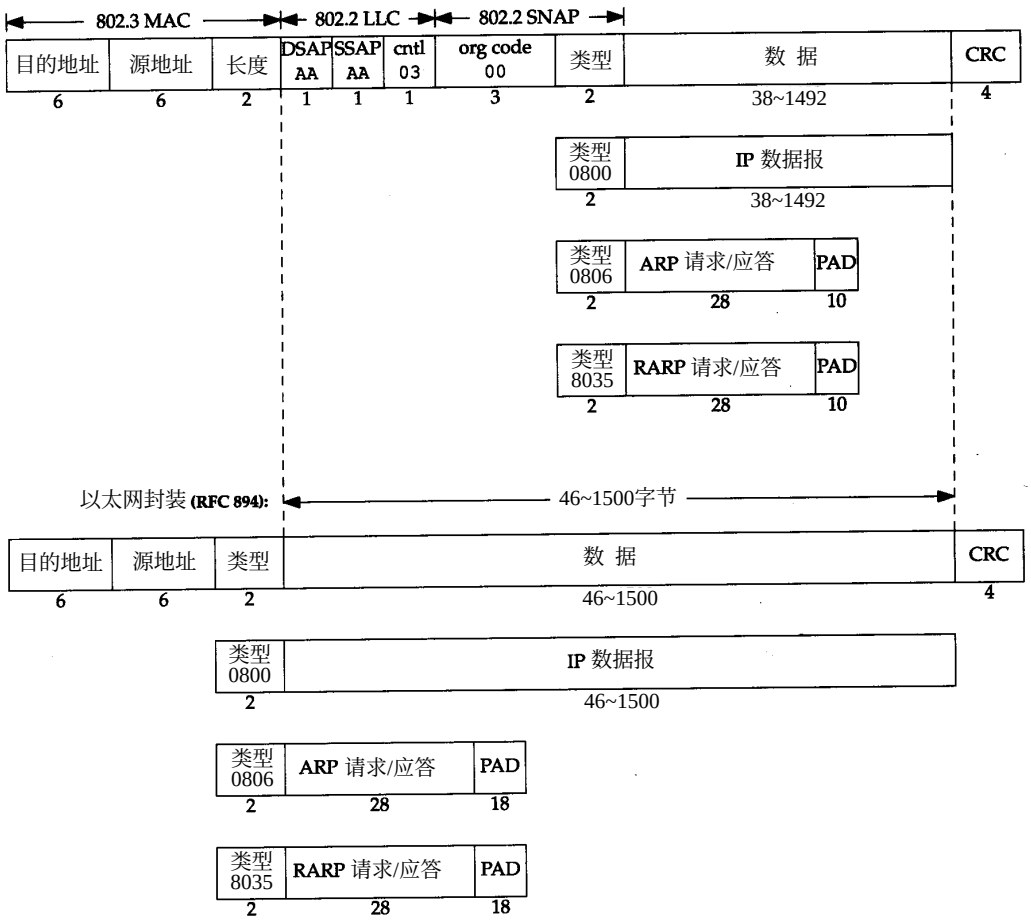

图2-1 IEEE 802.2/802.3（RFC 1042）和以太网的封装格式（RFC 894）

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。

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。

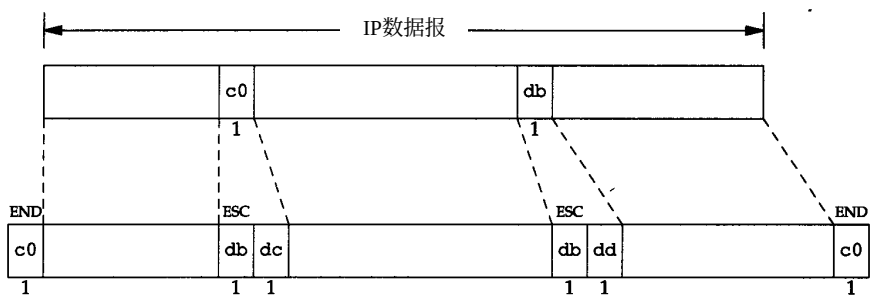

图2-2 SLIP报文的封装

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。

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。

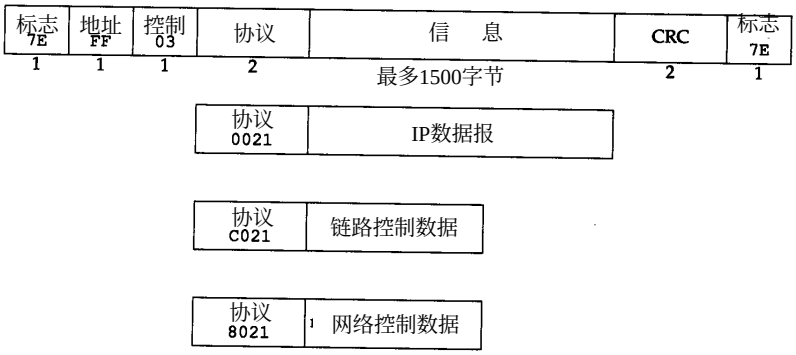

图2-3 PPP数据帧的格式

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。

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。

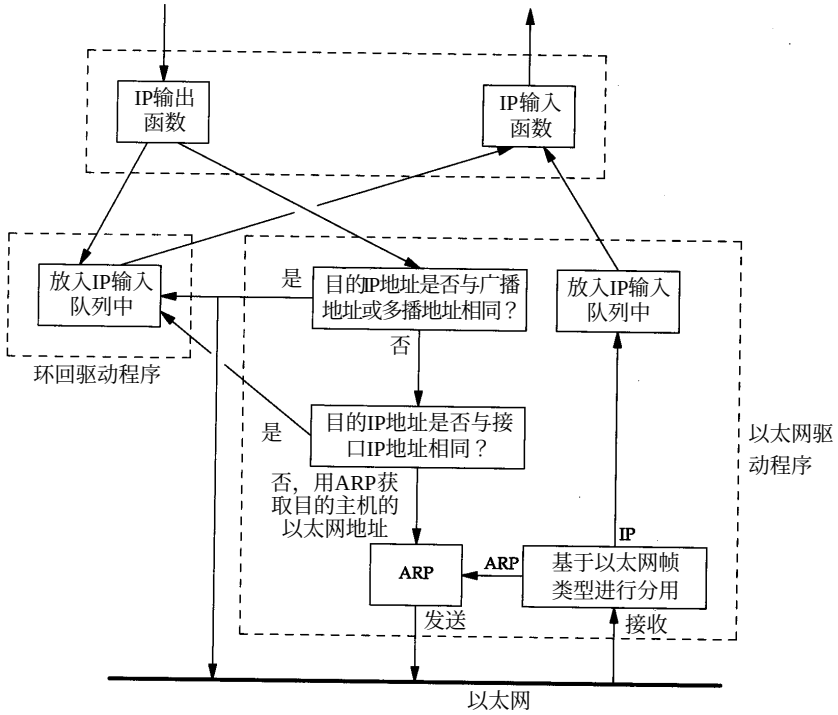

图2-4 环回接口处理IP数据报的过程

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

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

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。

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。

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。

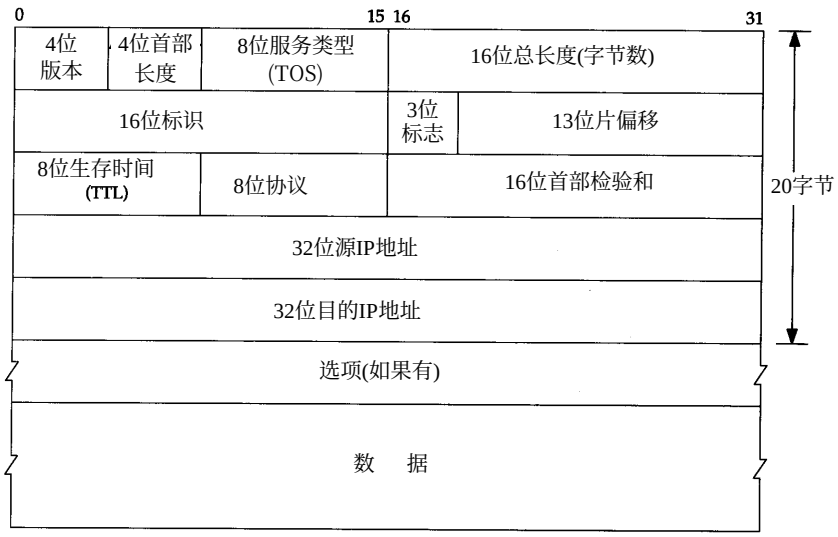

图3-1　IP数据报格式及首部中的各字段

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

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

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

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

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。

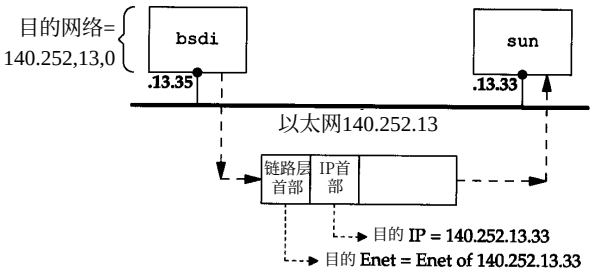

图3-3 数据报从主机bsdi到sun的传送过程

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）。

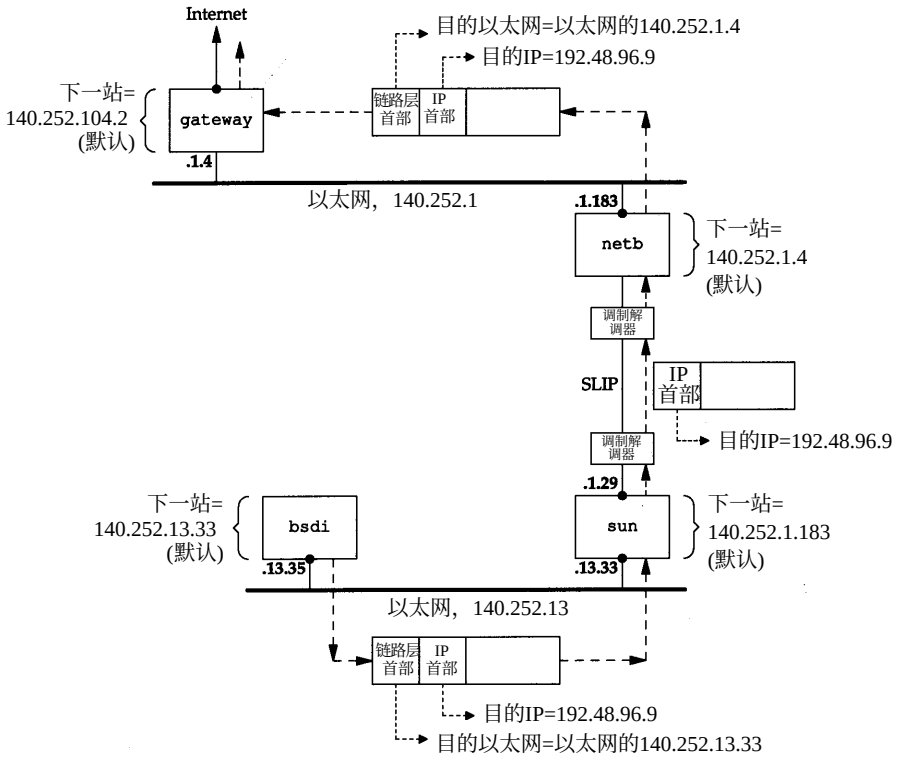

图3-4 从bsdi到ftp.uu.net (192.48.96.9)的初始路径

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。

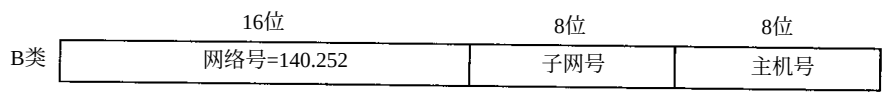

图3-5 B类地址的一种子网编址

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。

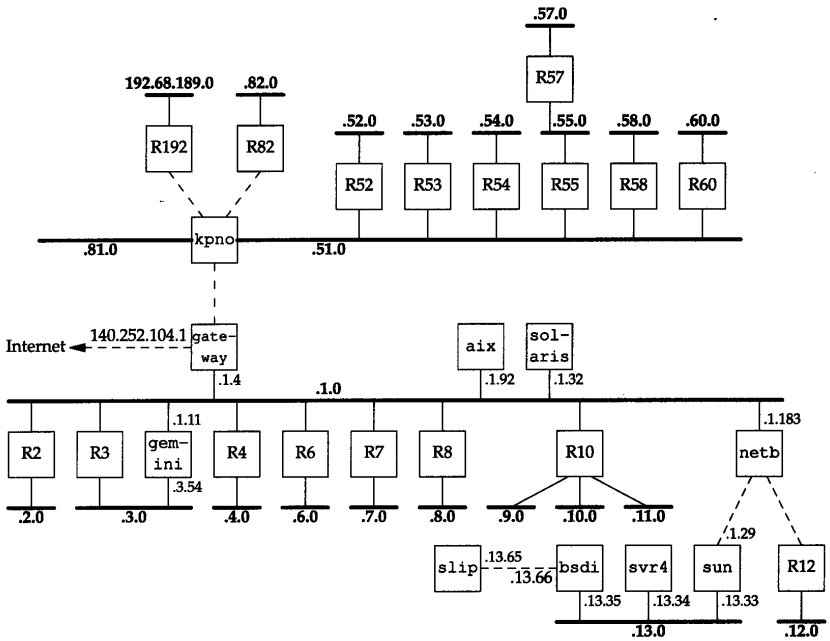

图3-6 网络noao.edu（140.252）中的大多数子网安排

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。

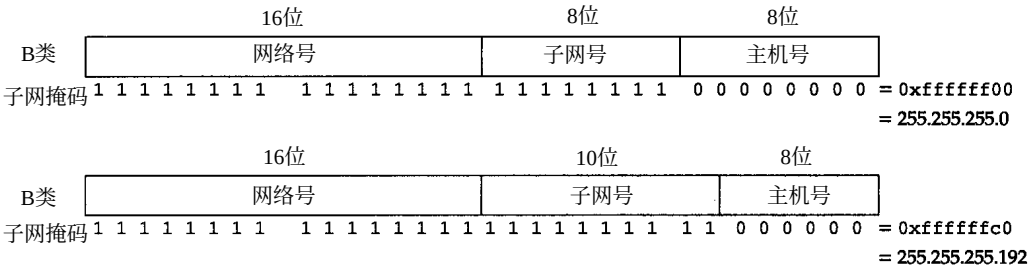

图3-7 两种不同的B类地址子网掩码的例子

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。

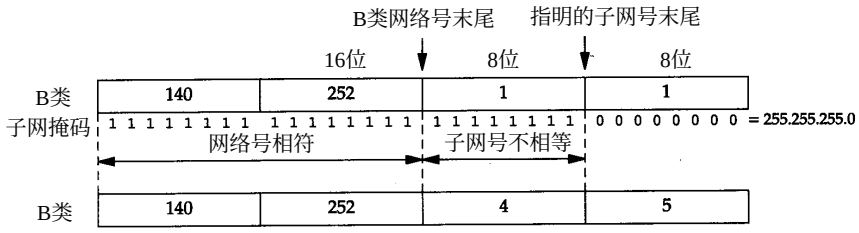

图3-8 使用子网掩码的两个B类地址之间的比较

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。

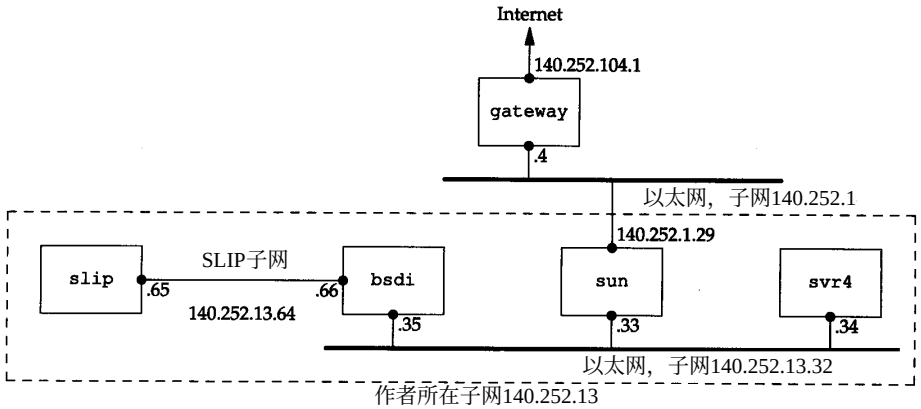

图3-10 作者所在子网中的主机和网络安排

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

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。

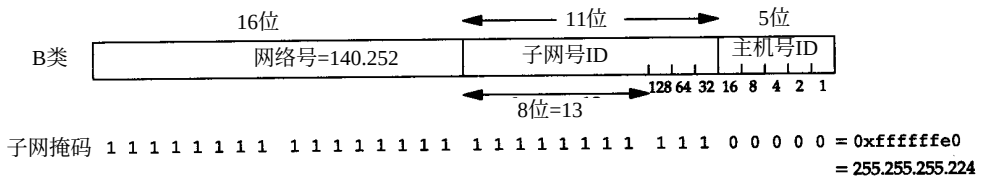

图3-11 变长子网

140.252.13子网中的所有接口的子网掩码是255.255.255.224，或0xfffffffe0。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	255.255.255.0	140.252.1	29	在子网1上
	140.252.13.33	255.255.255.224	140.252.13.32	1	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	255.255.255.224	140.252.13.32	3	在以太网上
	140.252.13.66	255.255.255.224	140.252.13.64	2	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））。

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 fffffe0 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 fffffe0 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 fffffe0
```

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 fffffe0
```

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 fffffe0 broadcast 140.252.13.63
```

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。

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。

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）。

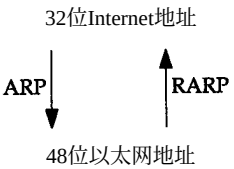

图4-1 地址解析协议：ARP和RARP

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。

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。”

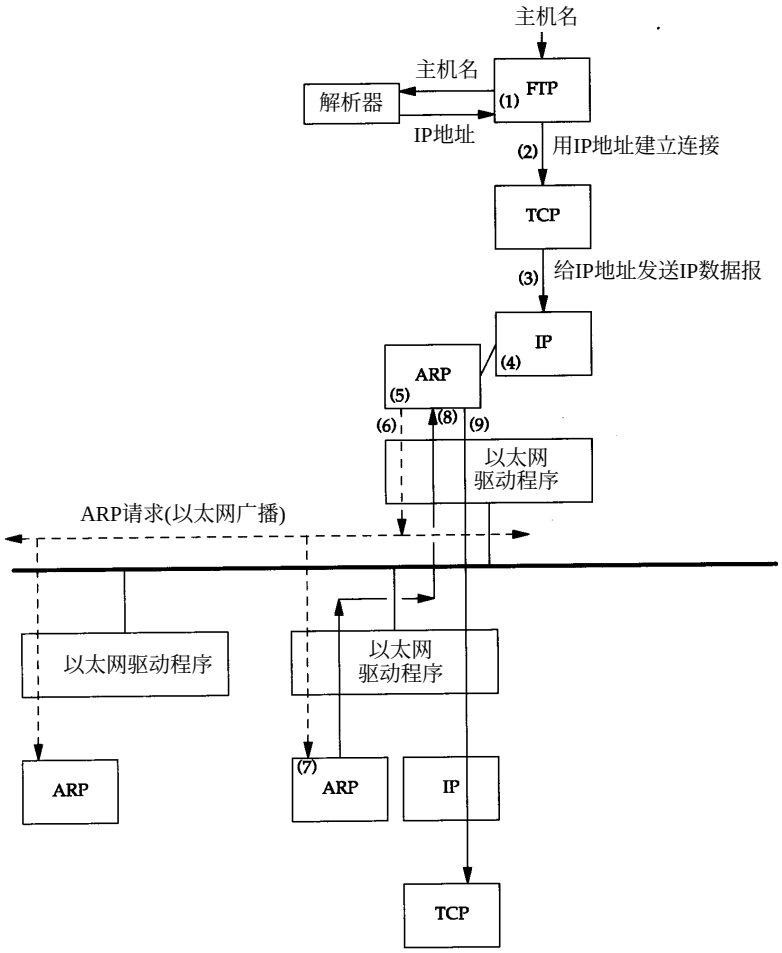

图4-2 当用户输入命令“ ftp 主机名 ”时ARP的操作

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

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）。

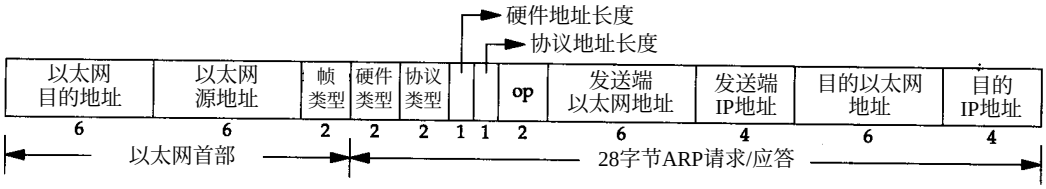

图4-3 用于以太网的ARP请求或应答分组格式

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

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

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

检查ARP高速缓存

```
bsdi % arp -a
? (140.252.13.36) at (incomplete)
```

tcpdump命令的输出如图4-5所示。

```
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

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。

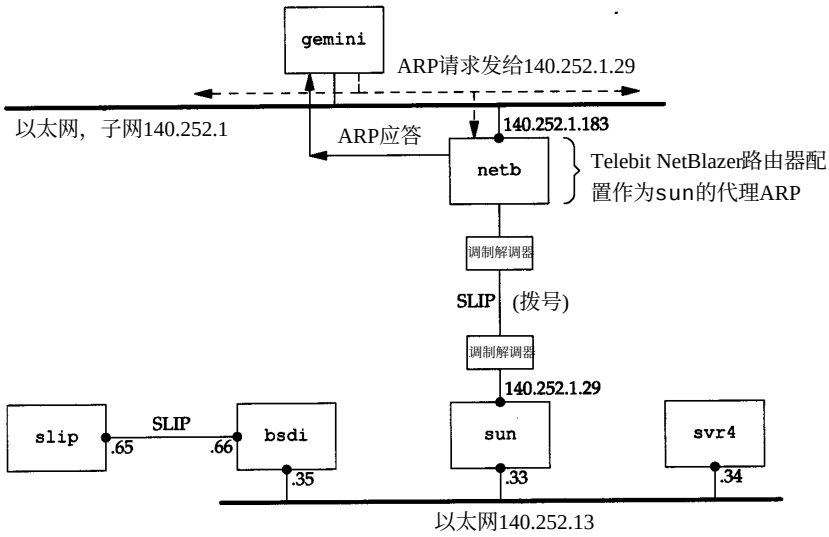

图4-6 ARP代理的例子

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

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

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？

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

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

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？

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。

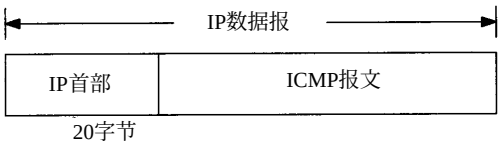

图6-1　ICMP封装在IP数据报内部

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。

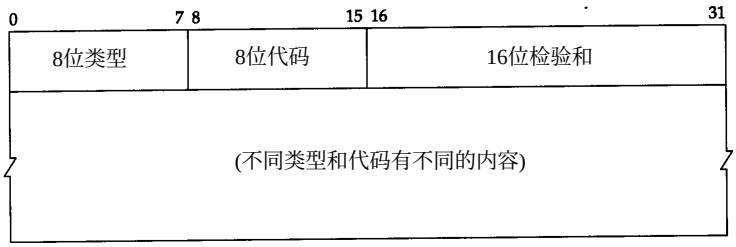

图6-2　ICMP报文

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

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。

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。

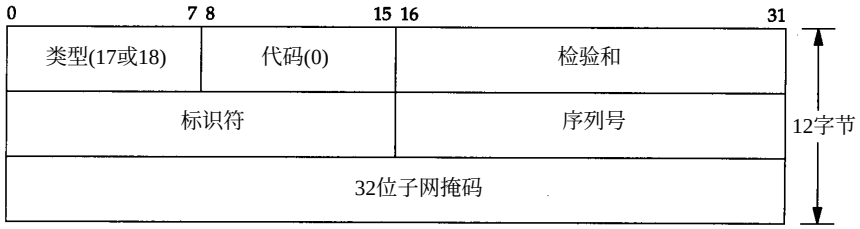

图6-4 ICMP地址掩码请求和应答报文

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 140.252.13.33 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

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。

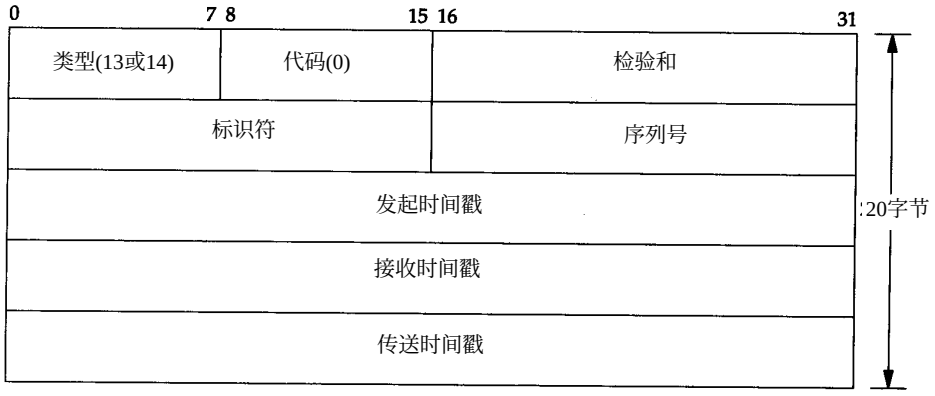

图6-6 ICMP时间戳请求和应答报文

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。

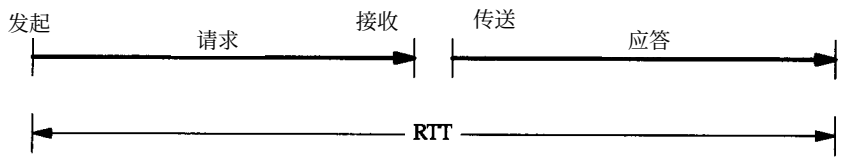

图6-7 icmptime 程序输出的值之间的关系

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

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），

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

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。

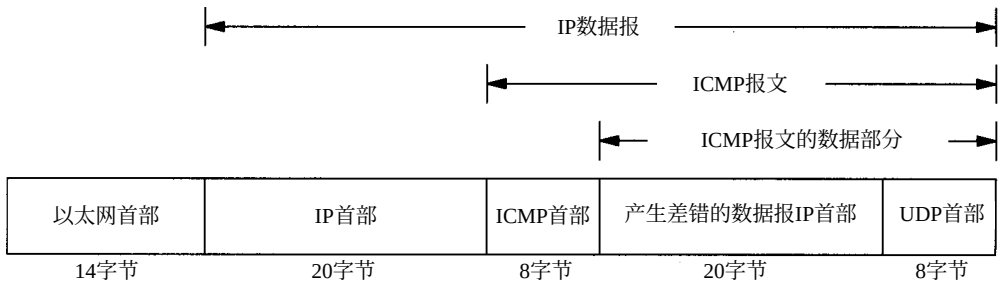

图6-9 “UDP端口不可达”例子中返回的ICMP报文

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。

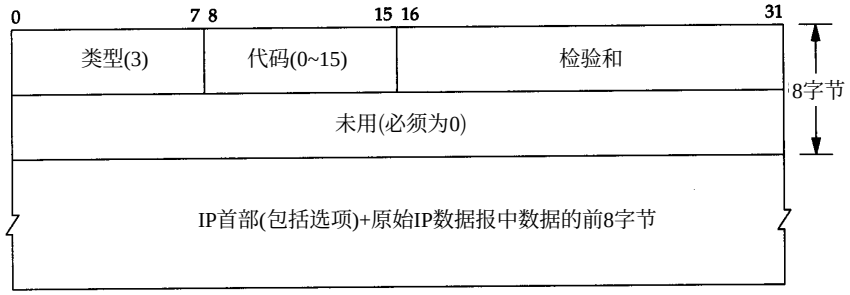

图6-10 ICMP不可达报文

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

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。

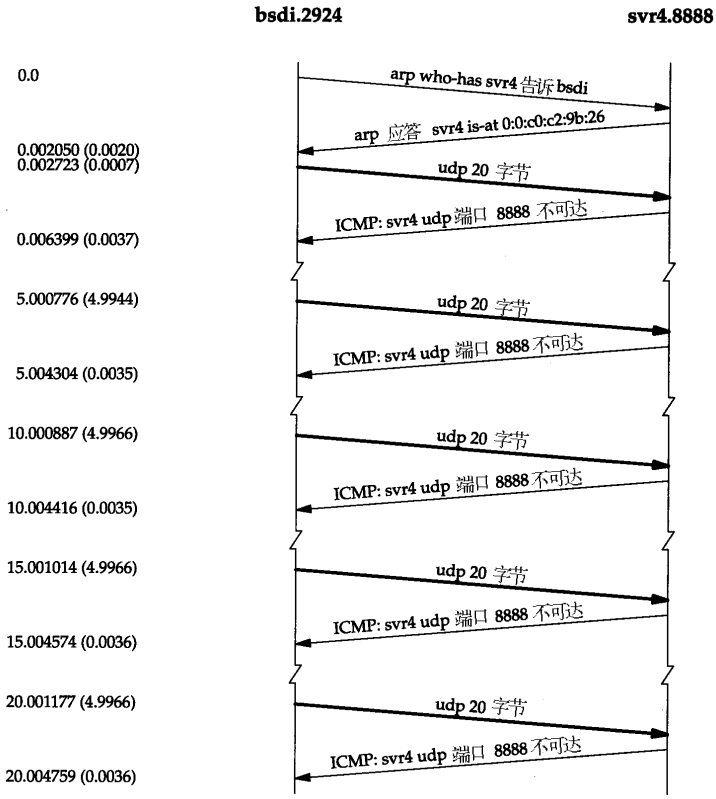

图6-11 发送到无效端口的TFTP请求的时间系列

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。

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

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。

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。

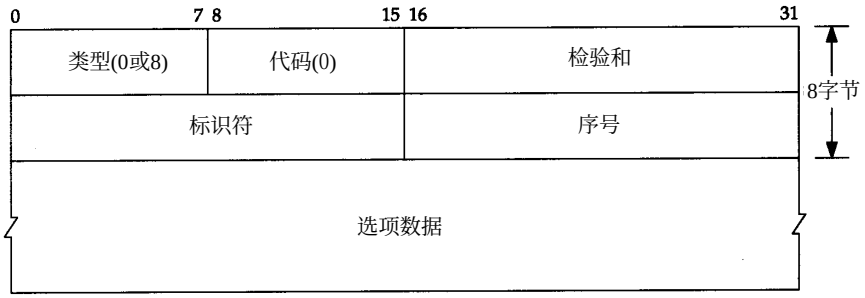

图7-1 ICMP回显请求和回显应答报文格式

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。

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

```
 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

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

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

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。

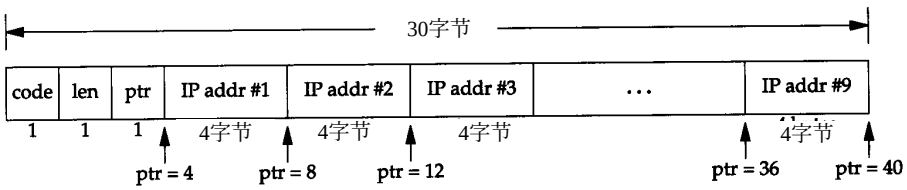

图7-3 IP首部中的记录路由选项的一般格式

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：

```
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。

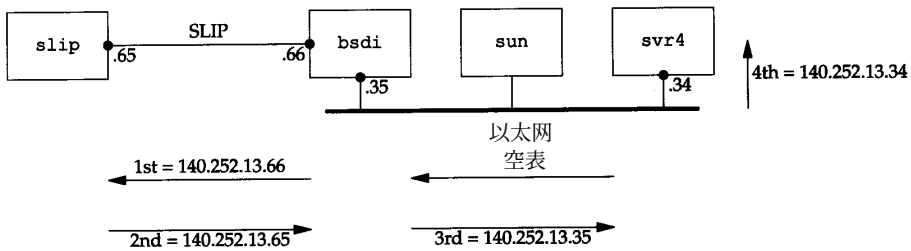

图7-4 带有记录路由选项的ping程序

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

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

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。

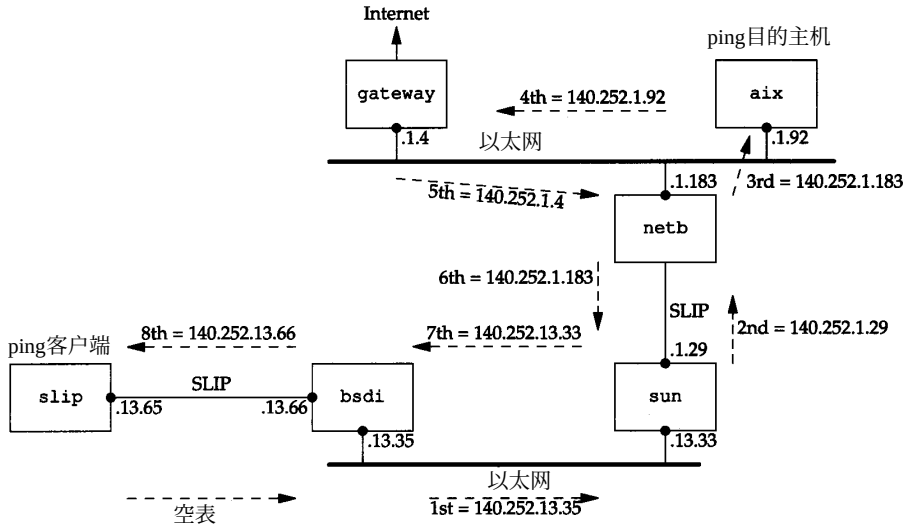

图7-6 运行带有记录路由选项的ping 程序，显示IP选路的特点

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）。

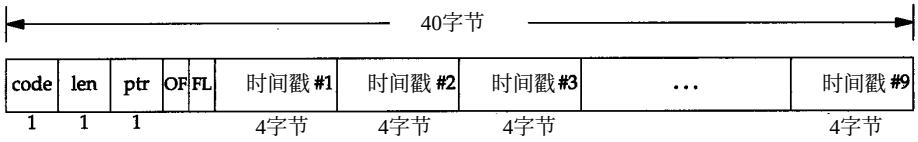

图7-7 IP首部中时间戳选项的一般格式

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。

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？

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ITL看成一个跳站计数器。

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

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 [ttl 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。

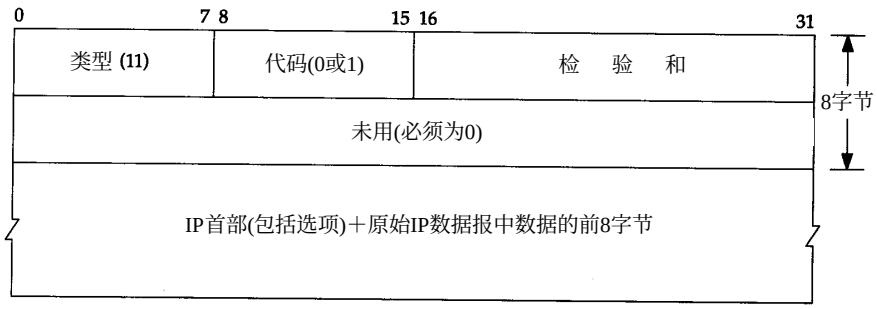

图8-2 ICMP超时报文

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。

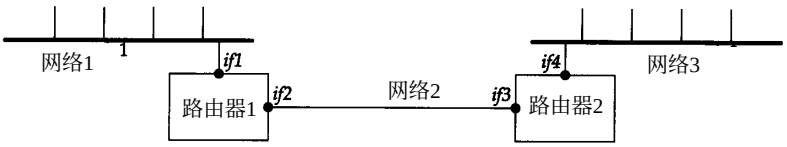

图8-3 traceroute 程序打印出的接口标识

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

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

（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。

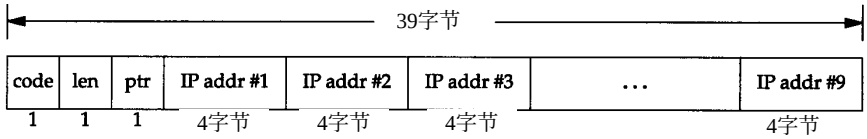

图8-6 IP首部源站路由选项的通用格式

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。

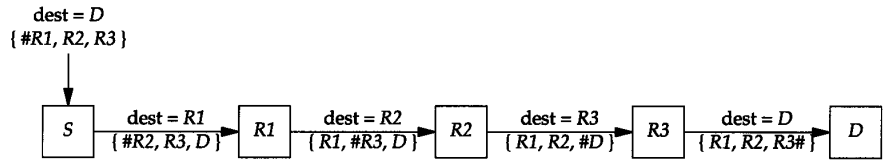

图8-7 IP源路由示例

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。

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

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

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。

```
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。

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？

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。

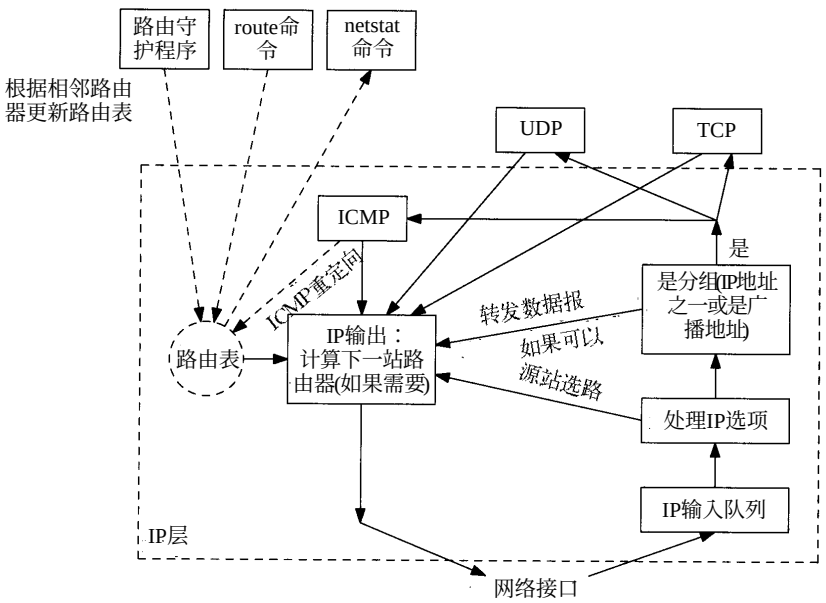

图9-1 IP层工作流程

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。

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（0xfffffe0，3.7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。

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。

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

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

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。

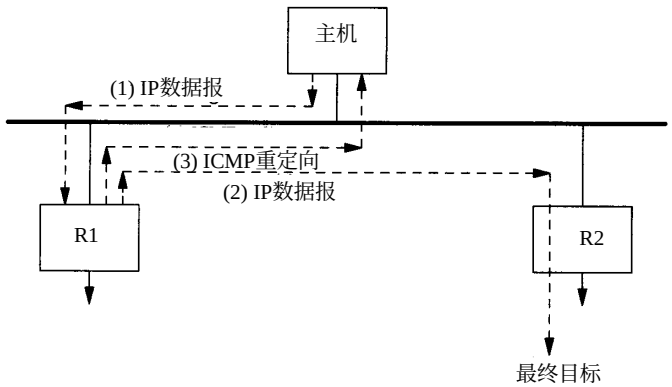

图9-3 ICMP重定向的例子

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

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。

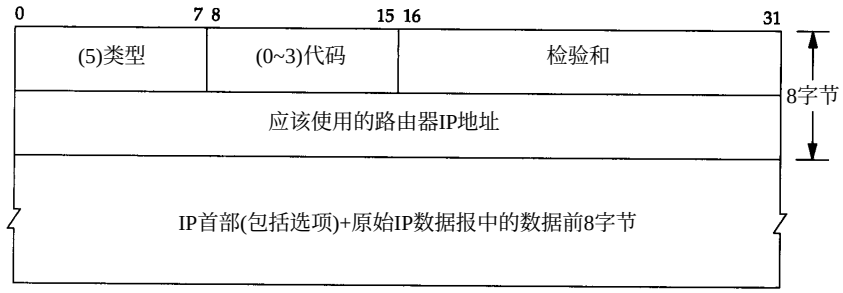

图9-4 ICMP重定向报文

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

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）。

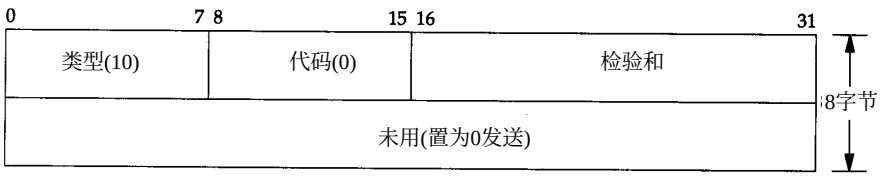

图9-6 ICMP路由器请求报文格式

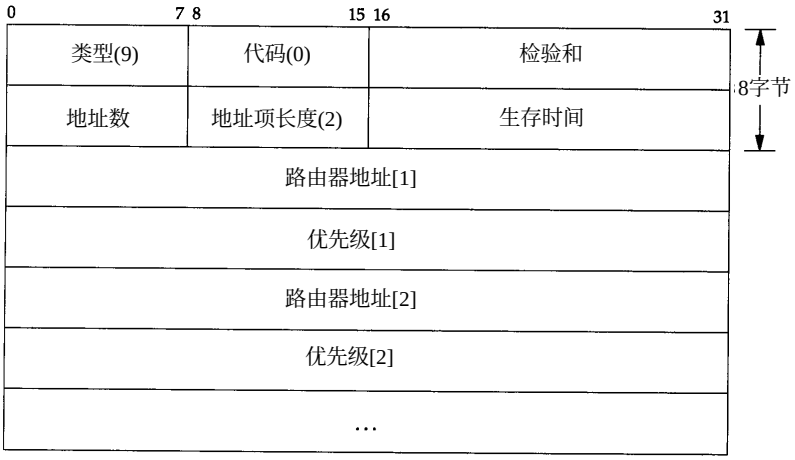

图9-7 ICMP路由器通告报文格式

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

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。

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）。

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

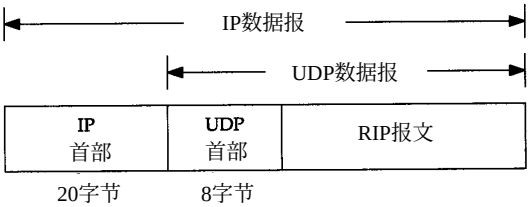

图10-2 封装在UDP数据报中的RIP报文

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。

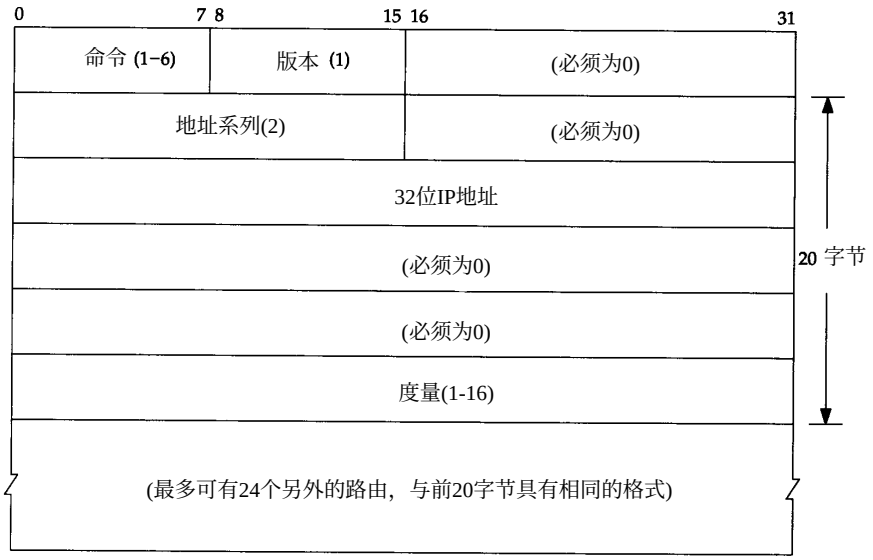

图 10-3

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。

• 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。

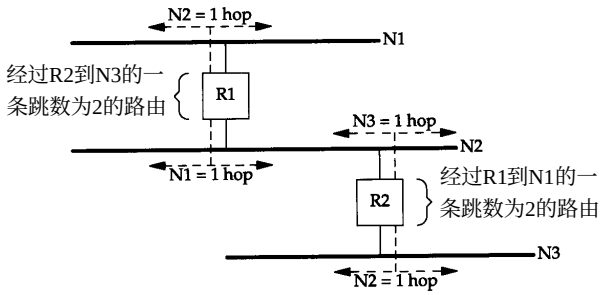

图10-4 路由器和网络示例

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。

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。

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。

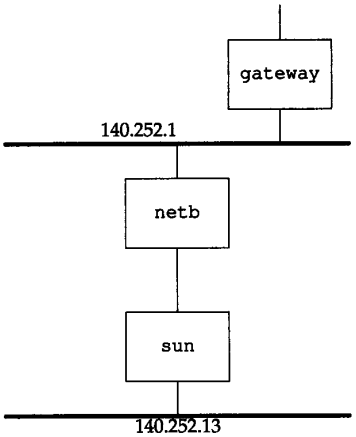

图10-5 查询其路由表内容的两个路由器netb和gateway

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：

```
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。

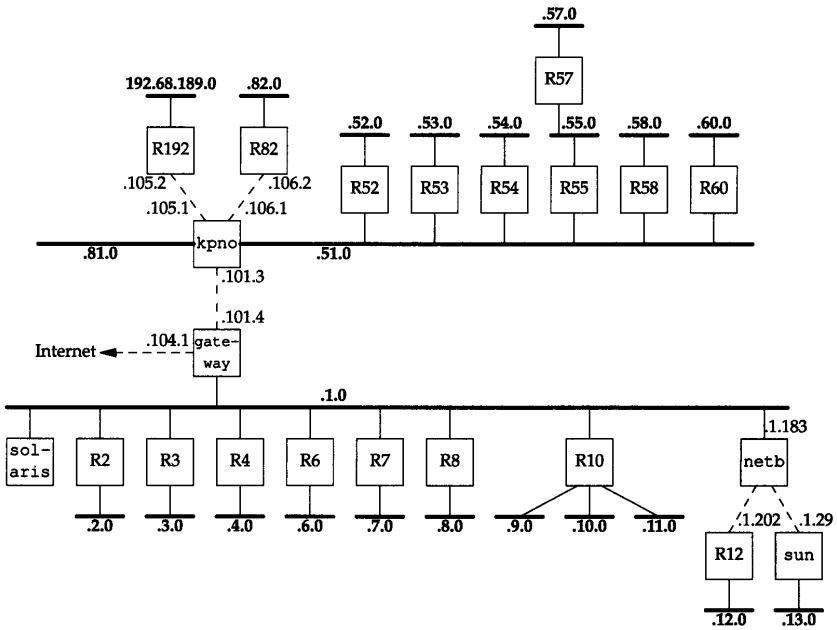

图10-7 noao.edu 140.252的多个网络

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。

```
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

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

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

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 [Rekhter 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。

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。

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）？

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。

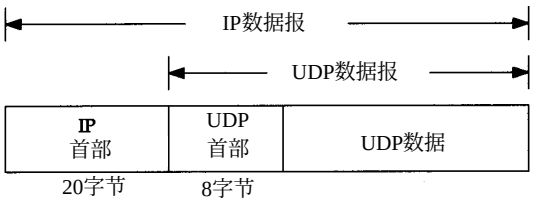

图11-1 UDP封装

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。

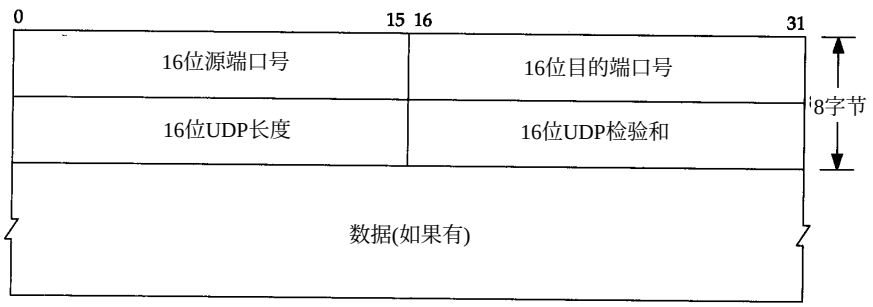

图11-2 UDP首部

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。

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。

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，

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

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。

```
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。

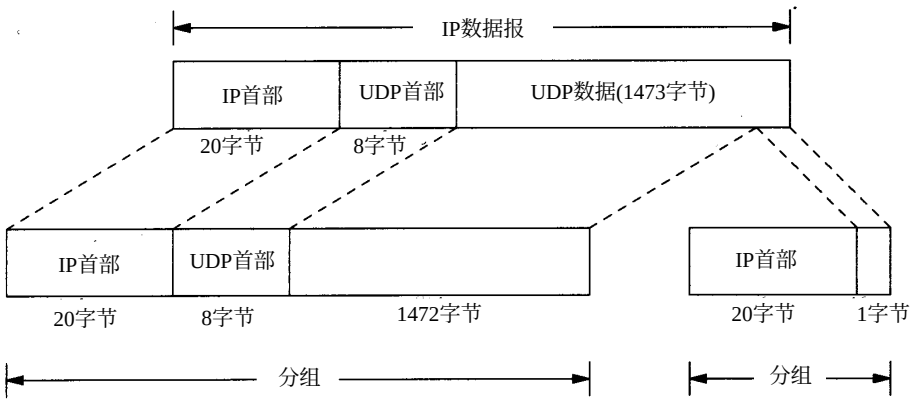

图11-8 UDP分片举例

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。

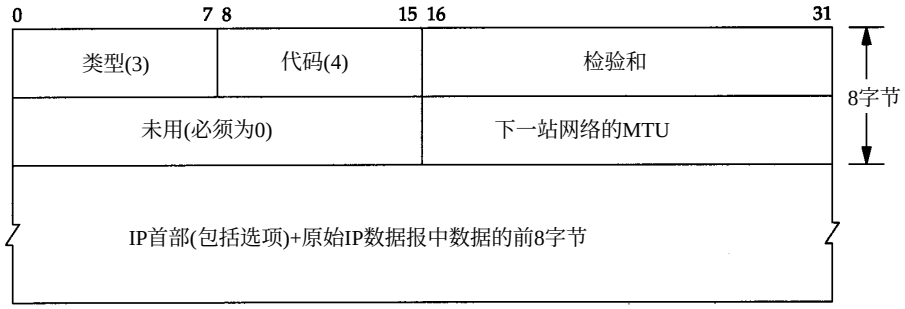

图11-9 需要分片但又设置不分片标志比特时的ICMP不可达差错报文格式

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。

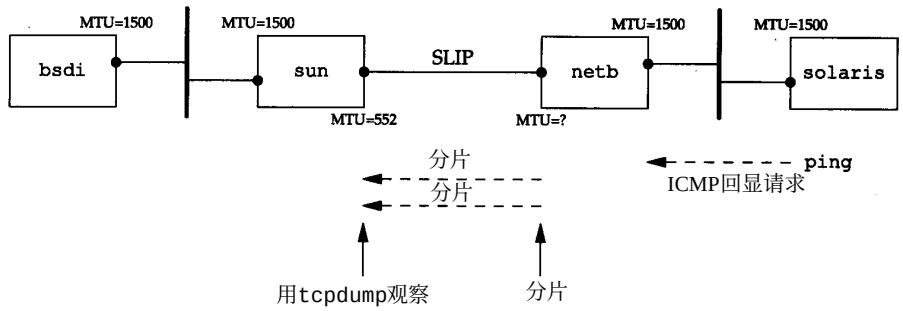

图11-10 用来判断从netb到sun的SLIP链路MTU的系统

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

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。

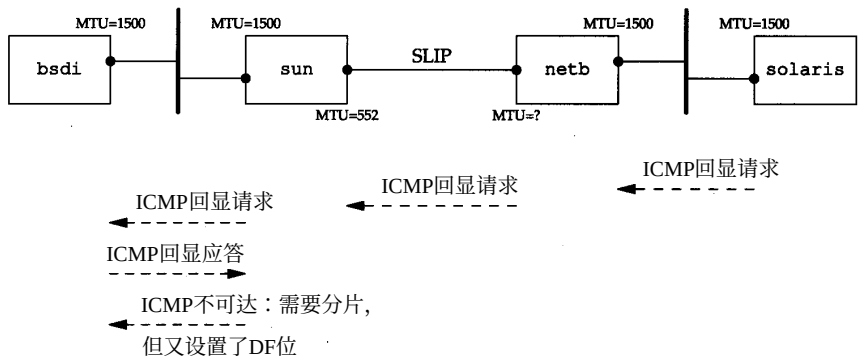

图11-12 例子中的分组交换

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

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。

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。

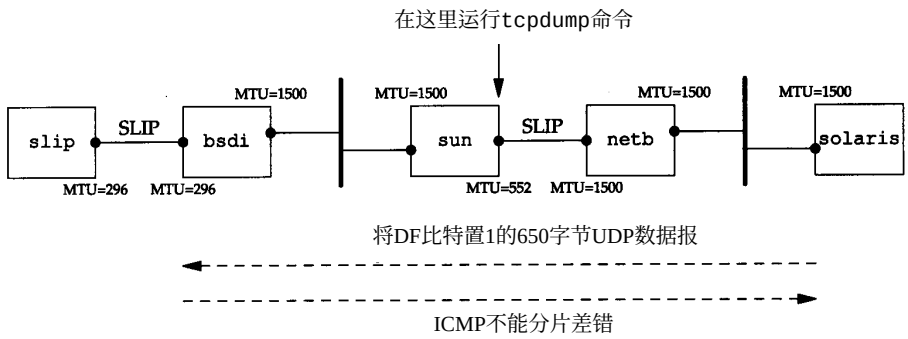

图11-13 使用UDP进行路径MTU发现的系统

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

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。

```
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，

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

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

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。

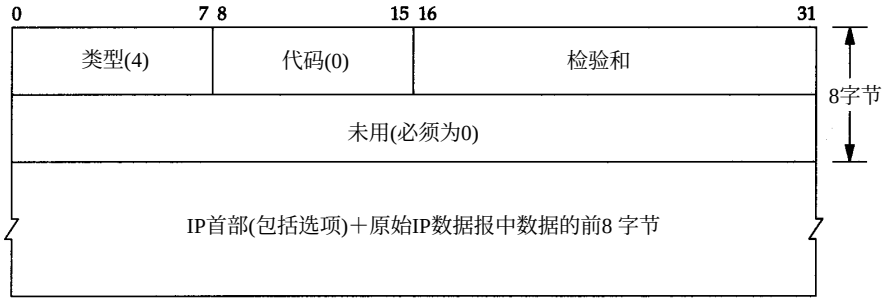

图11-18 ICMP源站抑制差错报文格式

```
 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

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）。

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

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)
```

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

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

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？

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。

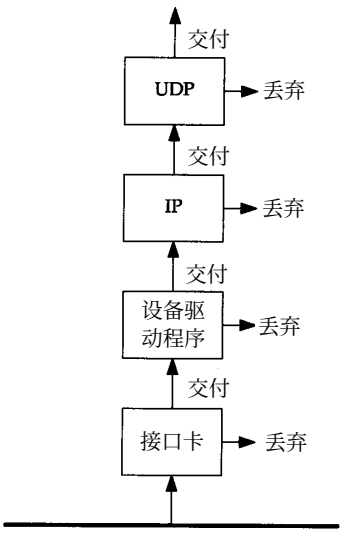

图12-1　协议栈各层对收到帧的过滤过程

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

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

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

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

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。

12.4.1 多播组地址

图12-2显示了D类IP地址的格式。

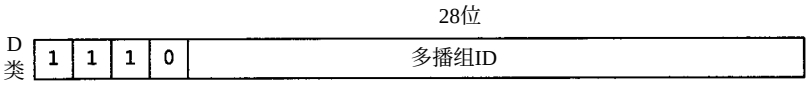

图12-2 D类IP地址格式

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。

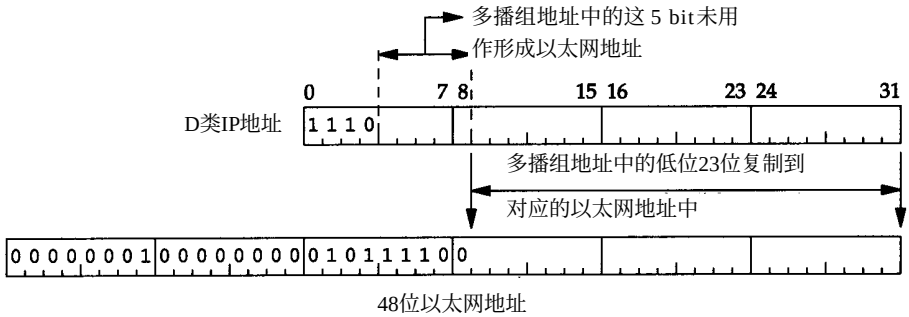

图12-3 D类IP地址到以太网多播地址的映射

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。

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？

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]。

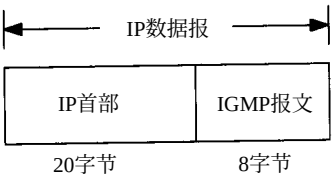

图13-1 IGMP报文封装在IP数据报中

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。

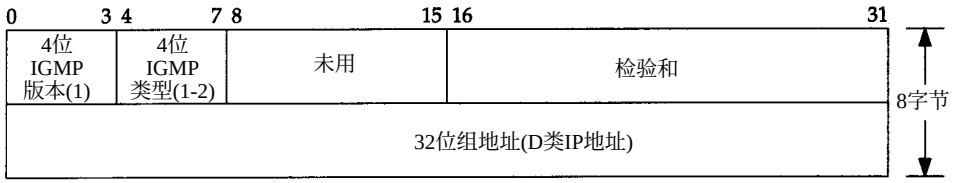

图13-2 IGMP报文的字段格式

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

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。

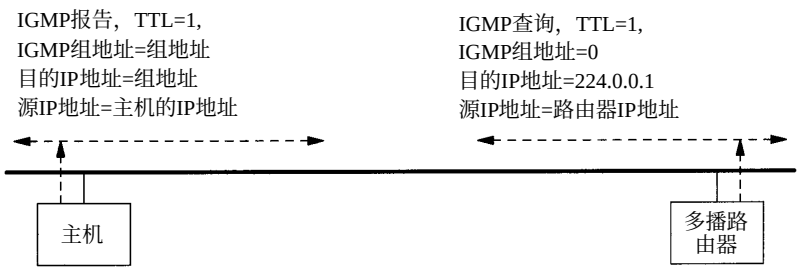

图13-3 IGMP的报告和查询

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

（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

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。

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

(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？

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，

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。

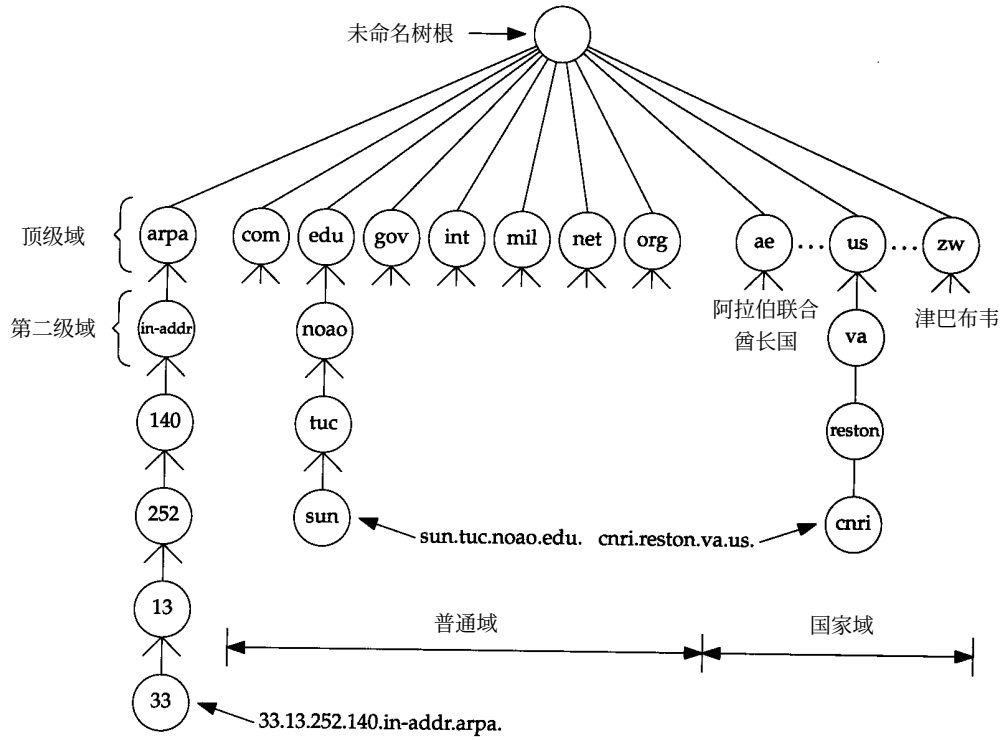

图14-1 DNS的层次组织

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。

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。

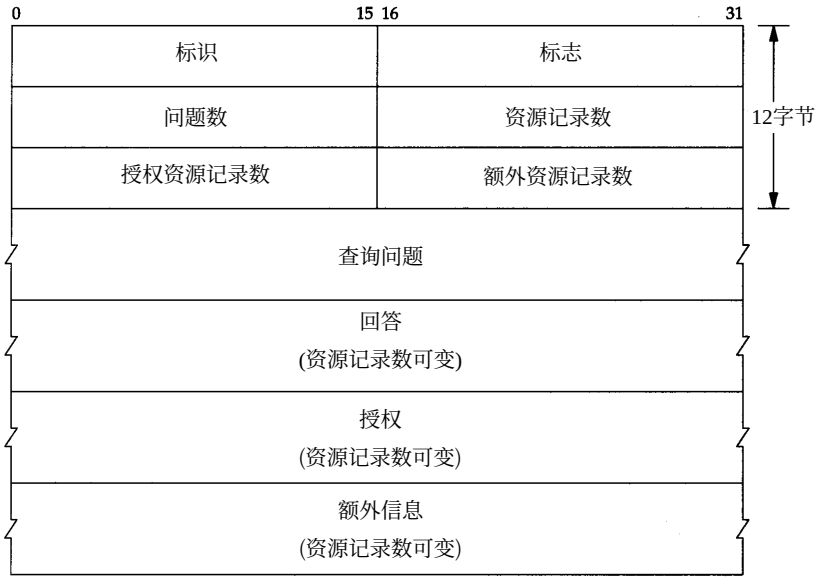

图14-3 DNS查询和响应的一般格式

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。

• 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。

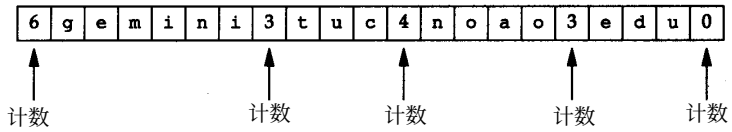

图14-6 域名gemini.tuc.noao.edu 的表示

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

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。

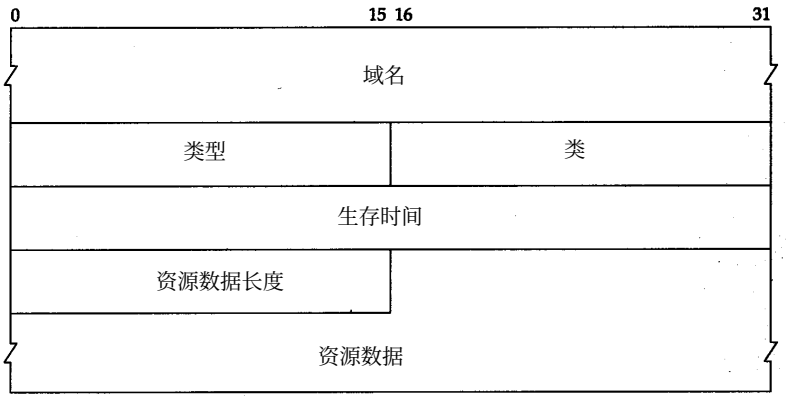

图14-8 DNS资源记录格式

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。

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。

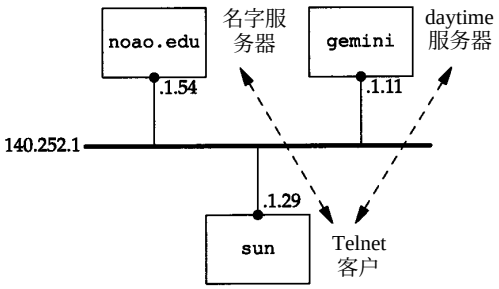

图14-9 用于简单DNS例子的系统

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。

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
```

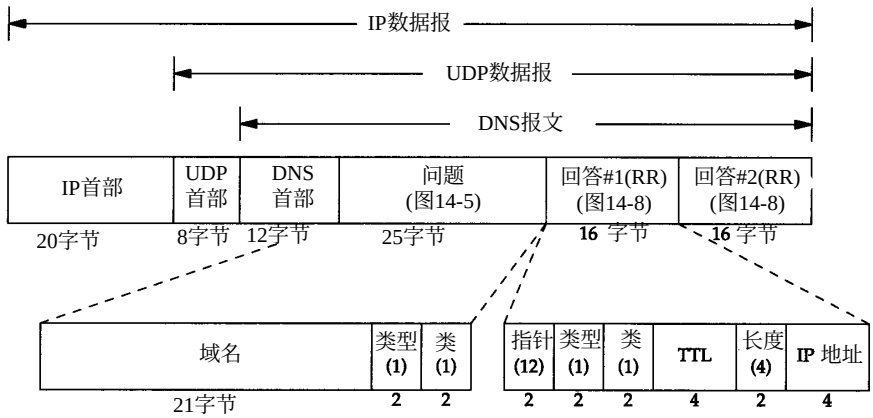

图14-11 对应于图14-10中第2行DNS应答的格式

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。

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

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：

```
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。

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
```

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nsl.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。

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。

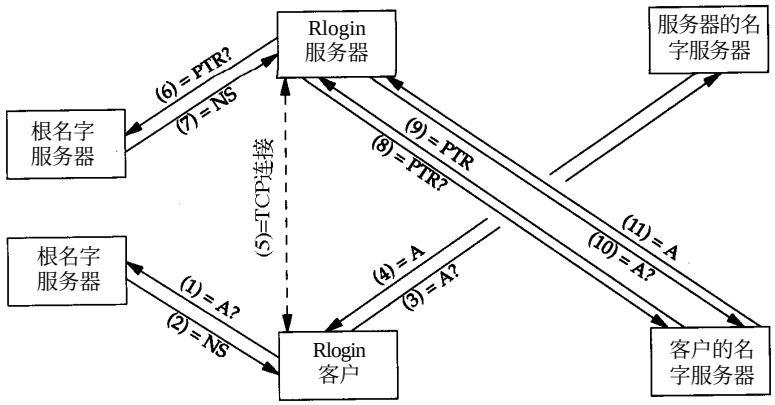

图14-16 启动Rlogin客户和服务器的分组交换过程

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

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？

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

单而不是高的系统吞吐量。

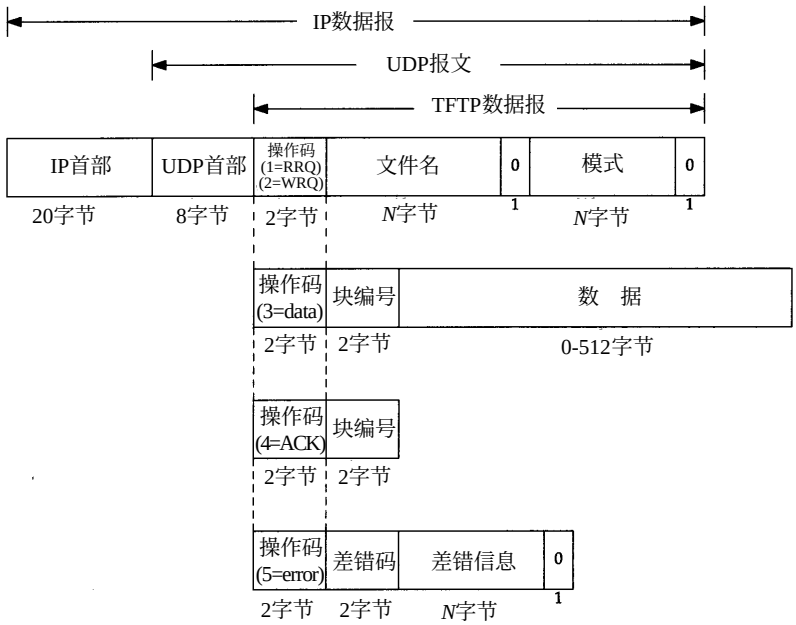

图15-1 5种TFTP报文格式

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，

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。

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？

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。

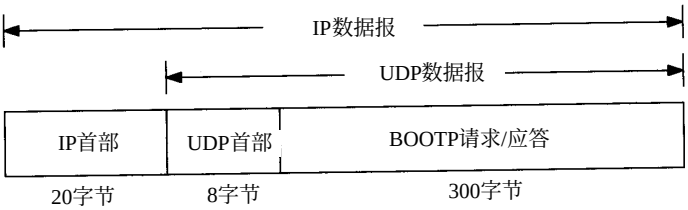

图16-1 BOOTP 请求和应答封装在一个UDP数据报内

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

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)。

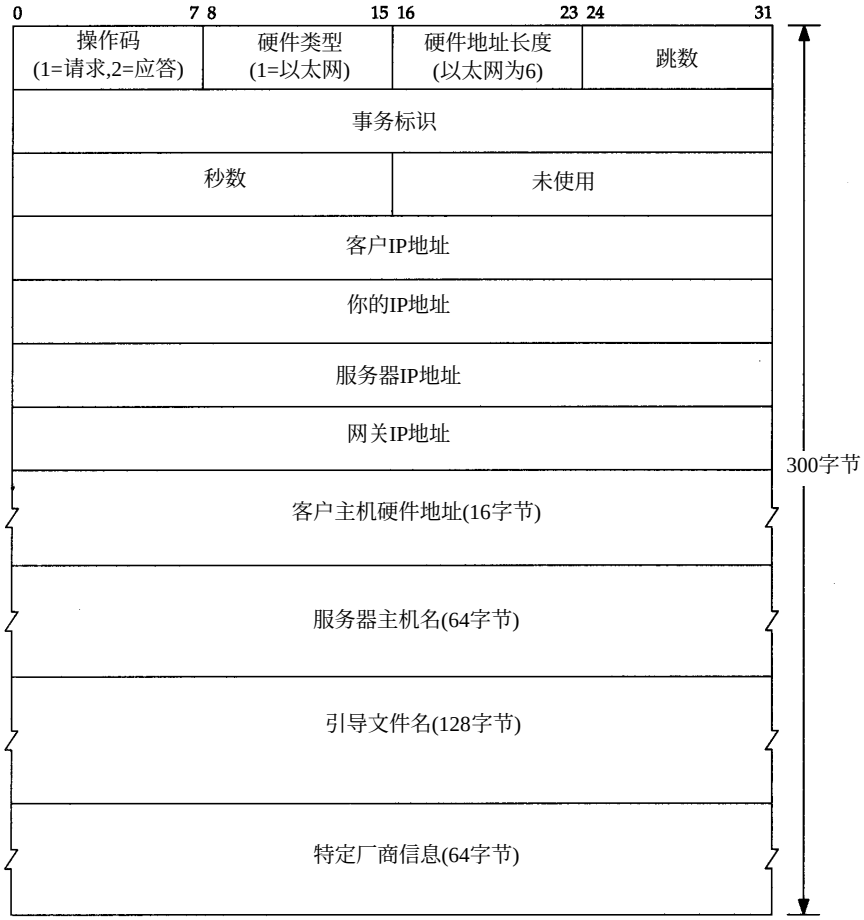

图16-2 BOOTP请求和应答的格式

“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

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，

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)。

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

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。

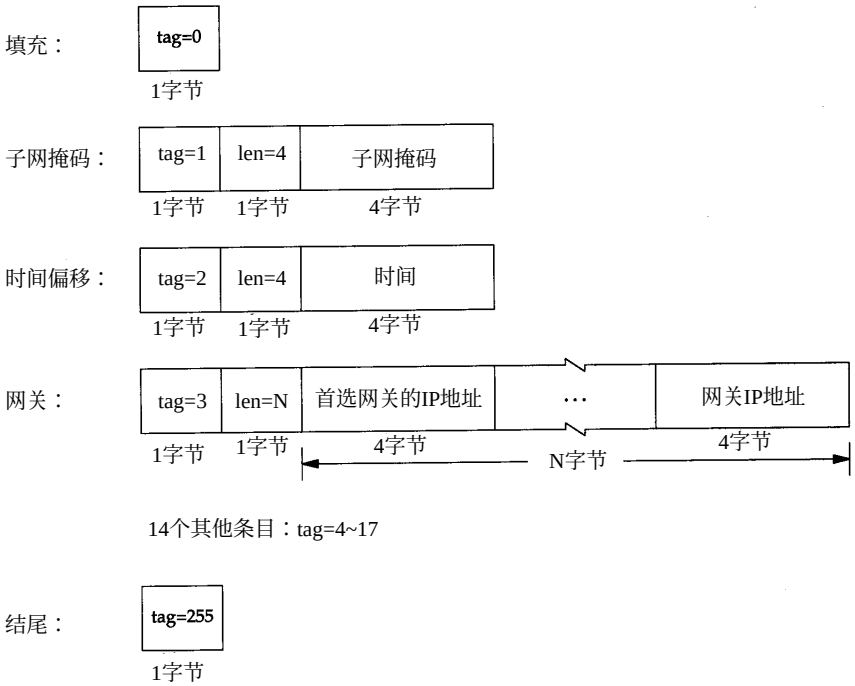

图16-4 厂商说明区域中一些条目的格式

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。

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。

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

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。

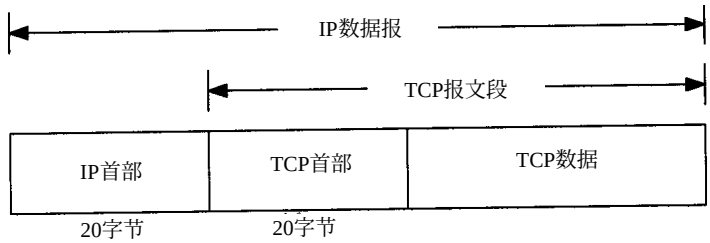

图17-1 TCP数据在IP数据报中的封装

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。

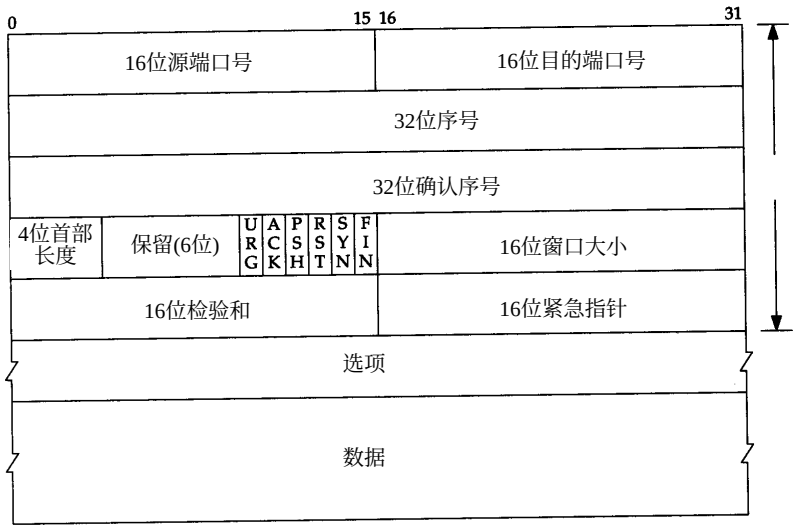

图17-2 TCP包首部

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。

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？

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                                    键入Ctrl和右括号，使Telnet客户进程终止连接
Connection closed.
```

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：

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。

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）。

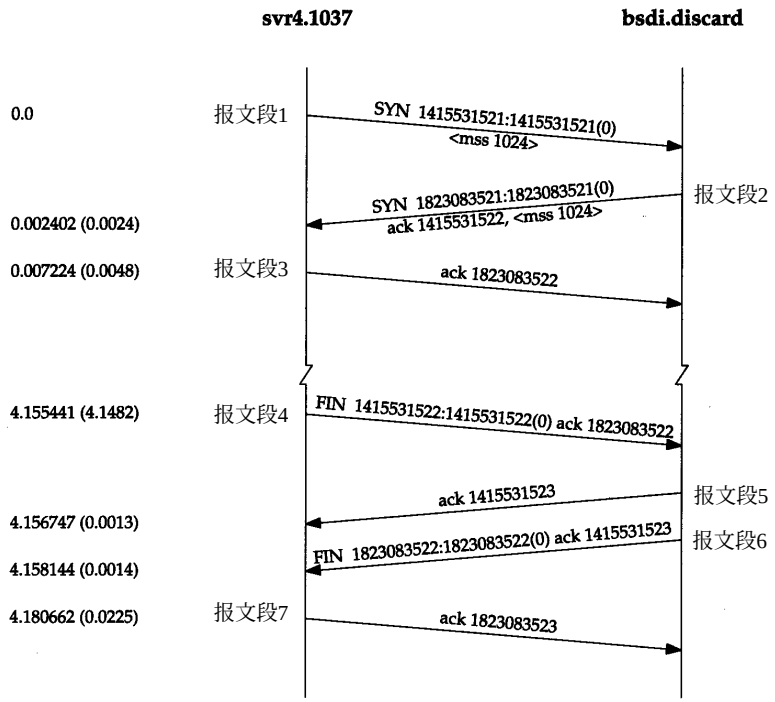

图18-3 连接建立与终止的时间系列

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。

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）。

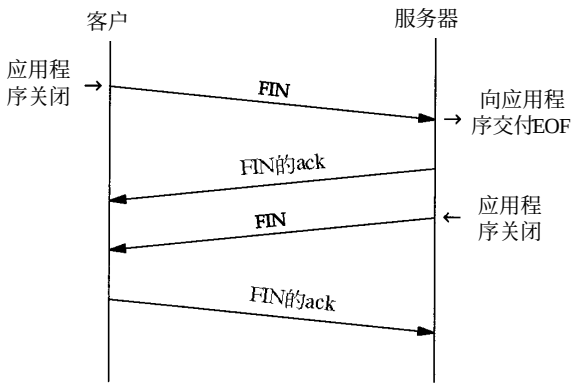

图18-4 连接终止期间报文段的正常交换

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。

```
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

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）。

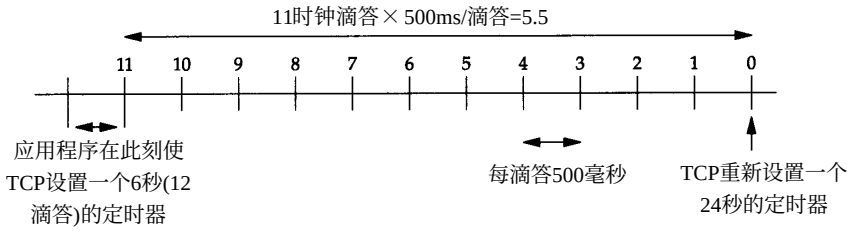

图18-7 TCP的500 ms定时器

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

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。

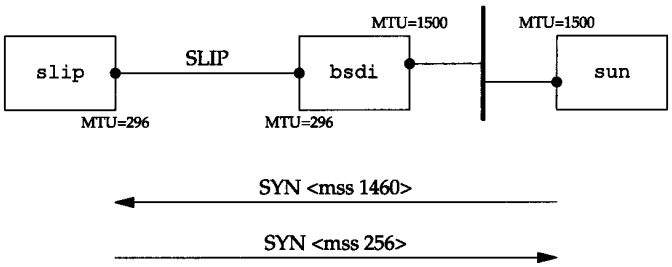

图18-8 显示sun与slip间TCP连接的MSS值

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

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。

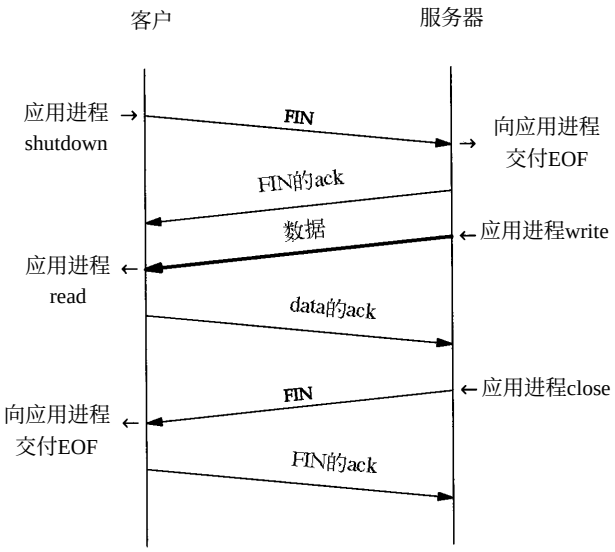

图18-10 TCP半关闭的例子

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）。

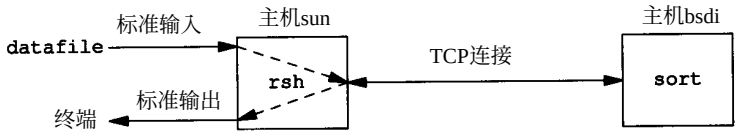

图18-11 命令：rsh bsdi sort < datafile

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

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。

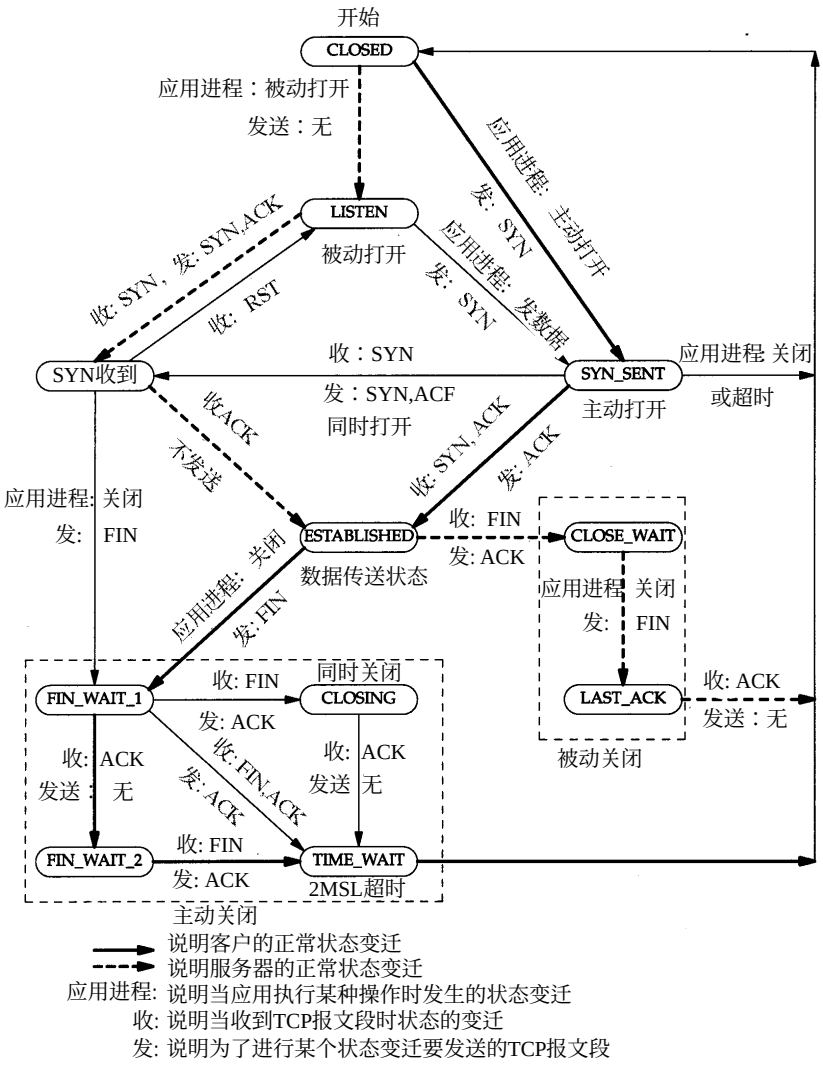

图18-12 TCP的状态变迁图

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。

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。

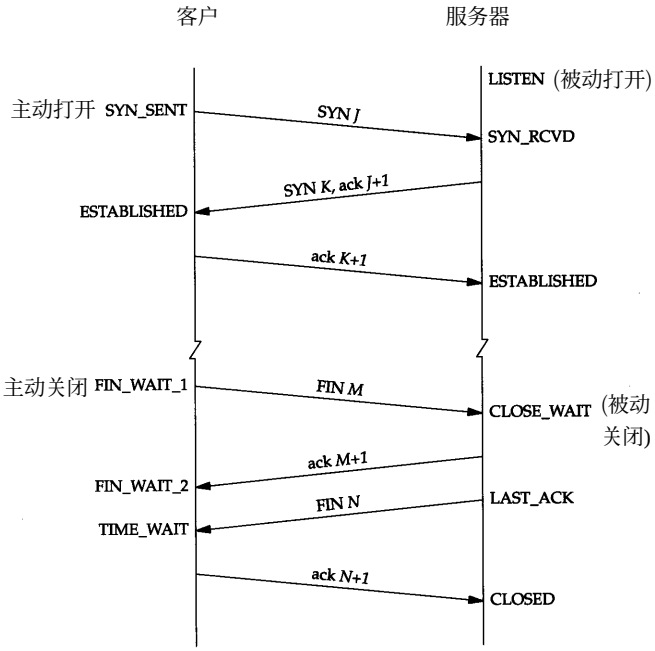

图18-13 TCP正常连接建立和终止所对应的状态

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

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

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
```

```
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）。

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

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。

```
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

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。

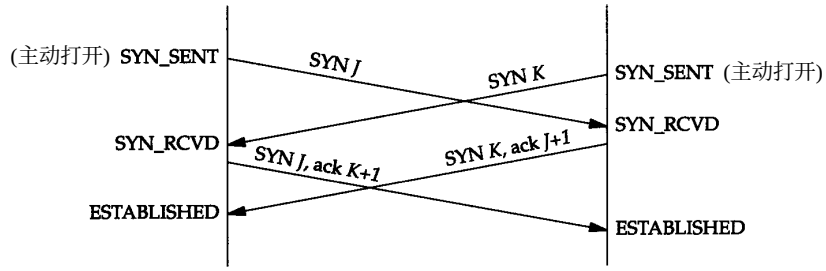

图18-17 同时打开期间报文段的交换

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

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。

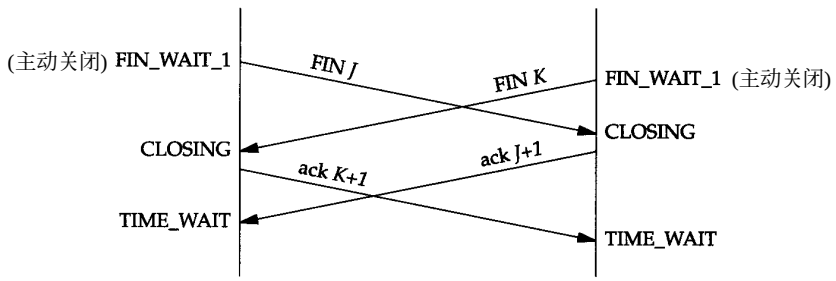

图18-19 同时关闭期间的报文段交换

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，

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。

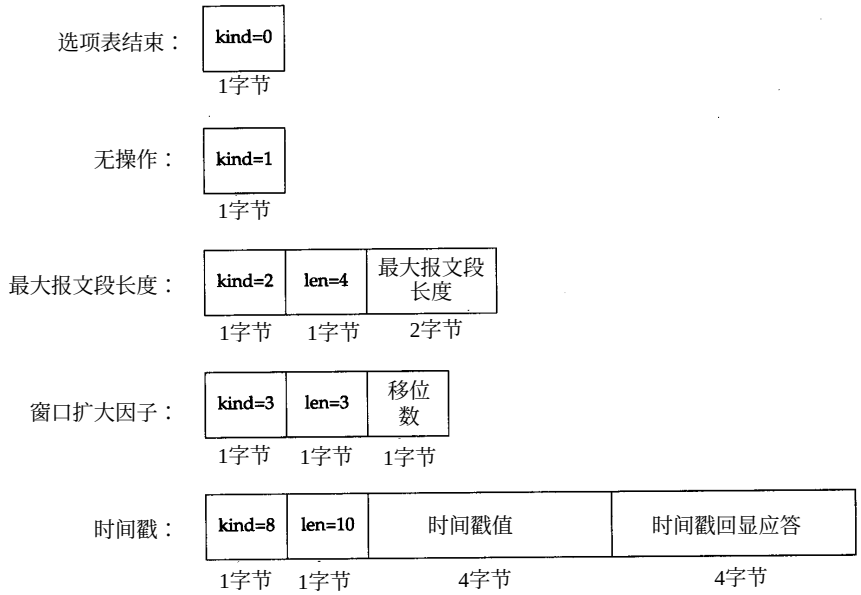

图18-20 TCP选项

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

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

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。

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。

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

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。

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

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

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

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。

```
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？

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。

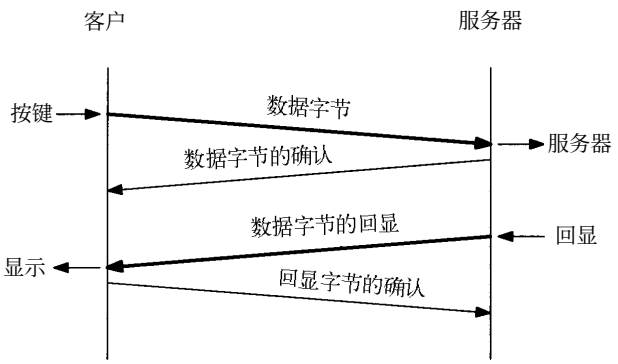

图19-1 一种可能的处理远程交互按键回显的方法

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）。

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、

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。

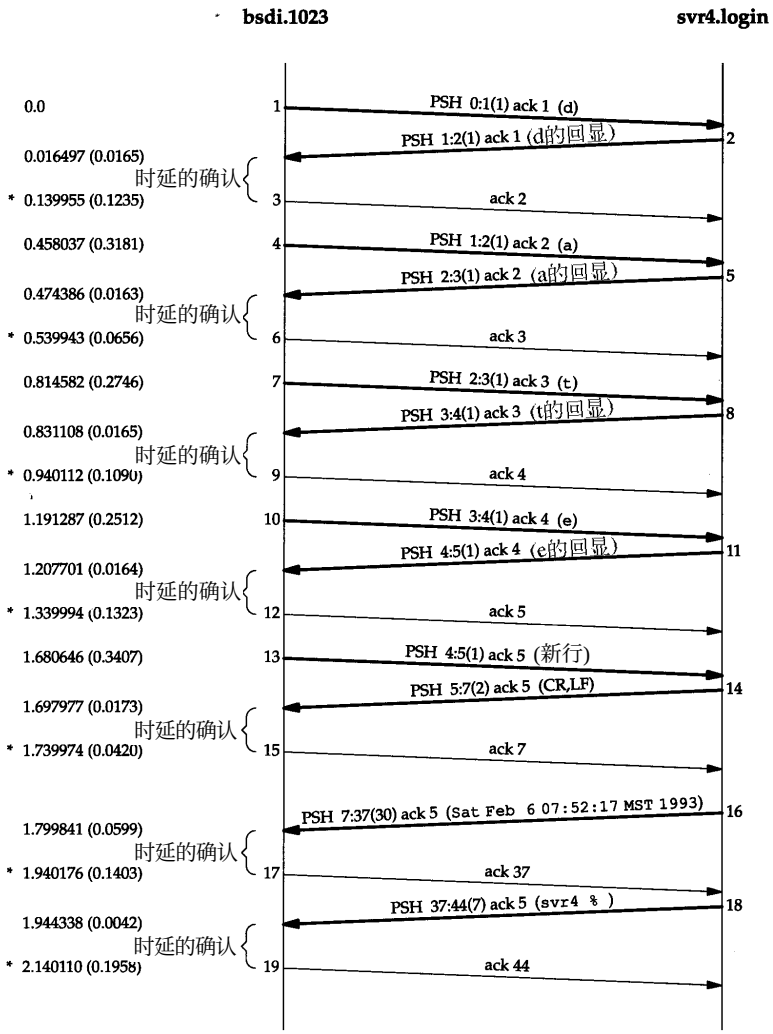

图19-3　在rlogin连接上键入date命令时的数据流时间系列

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

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

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。

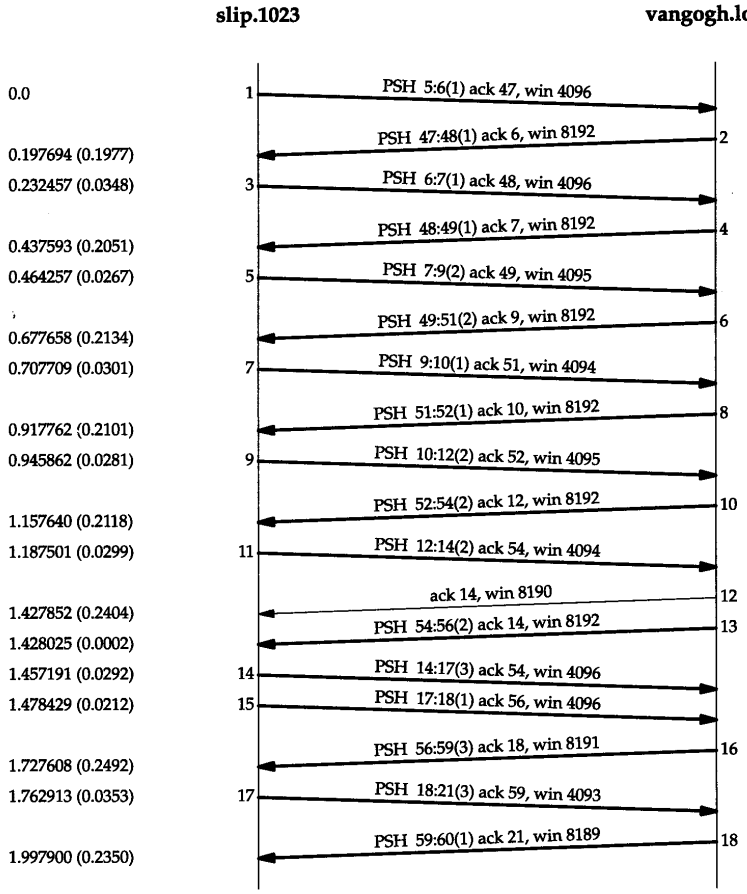

图19-4 在slip 和vangogh.cs.berkeley.edu 之间使用rlogin 时的数据流

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。

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。

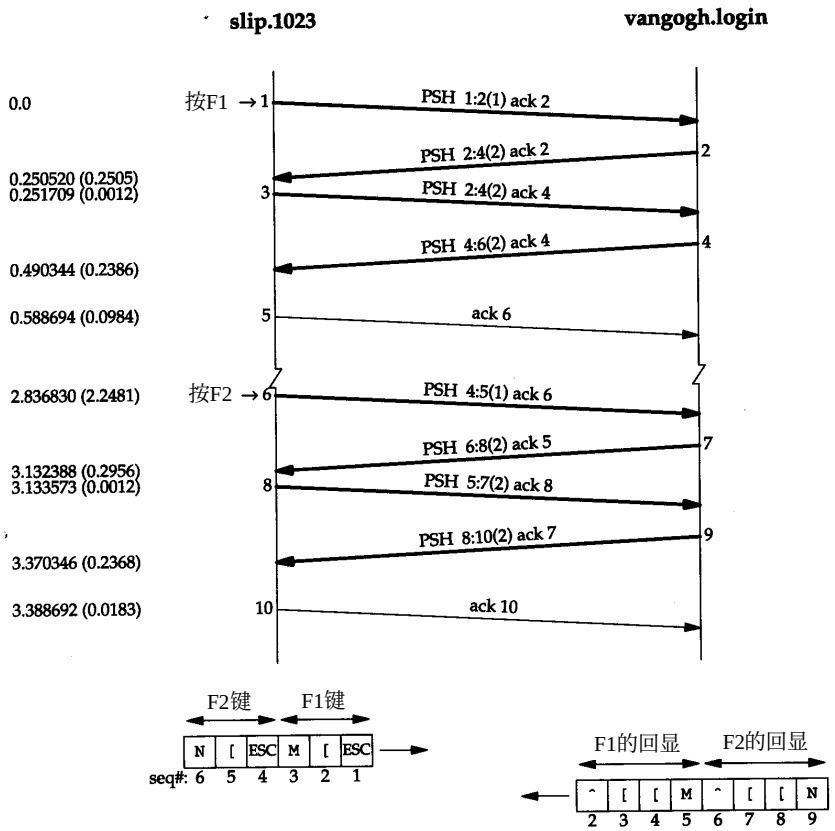

图19-6 图19-5的时间系列（Nagle算法的观察结果）

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。

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。

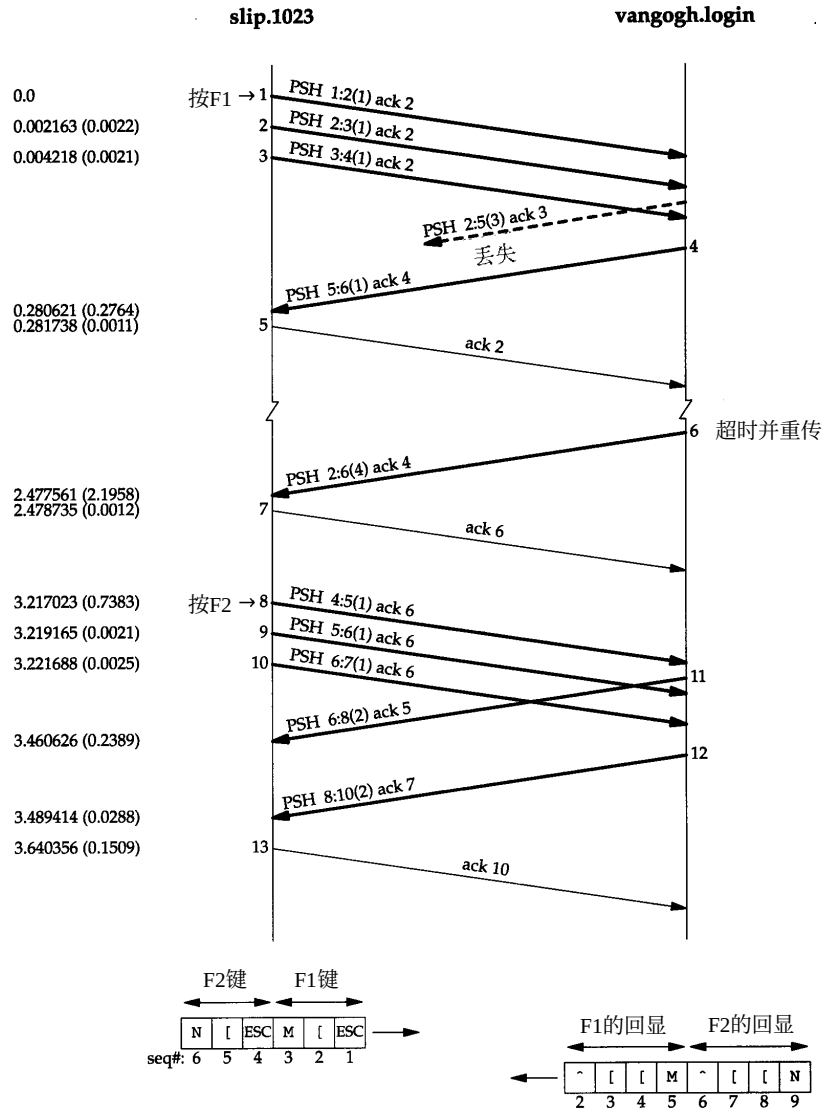

图19-8 图19-7的时间系列（关闭Nagle算法）

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。

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。

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。

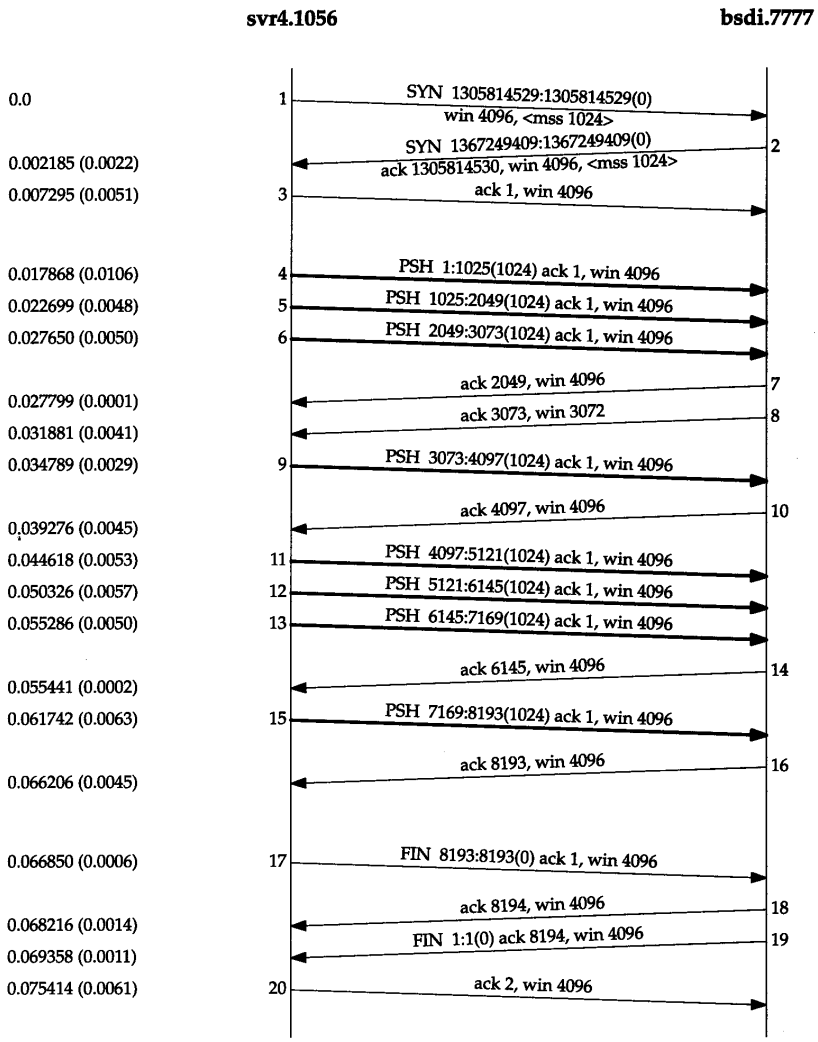

图20-1 从svr4 传输8192个字节到bsdi

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

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）。

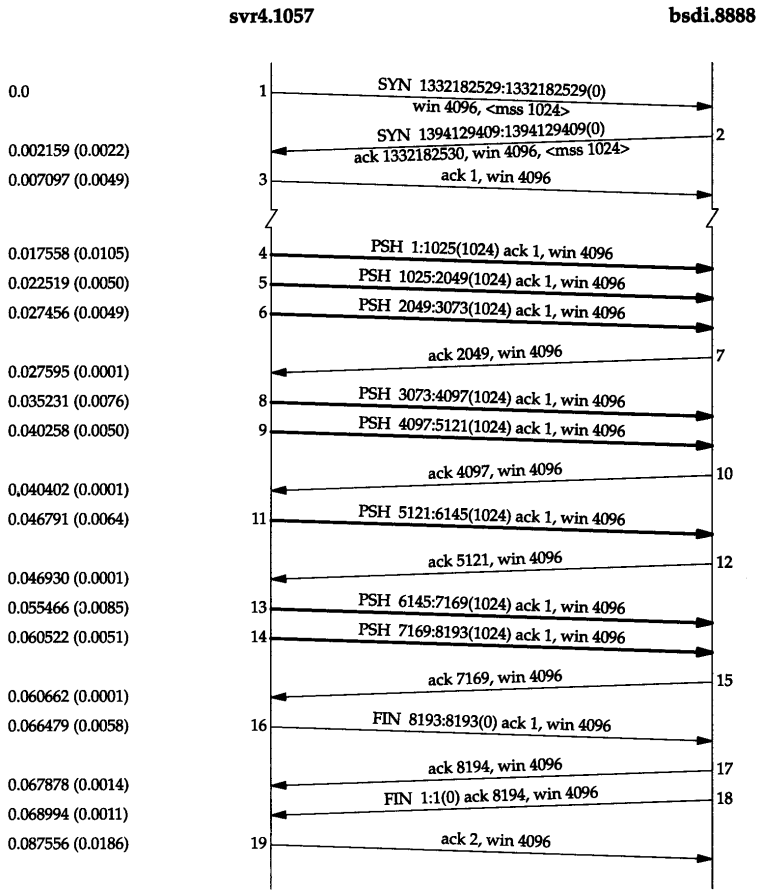

图20-2 从svr4 到bsdi 的另外8192字节数据的传输过程

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）。

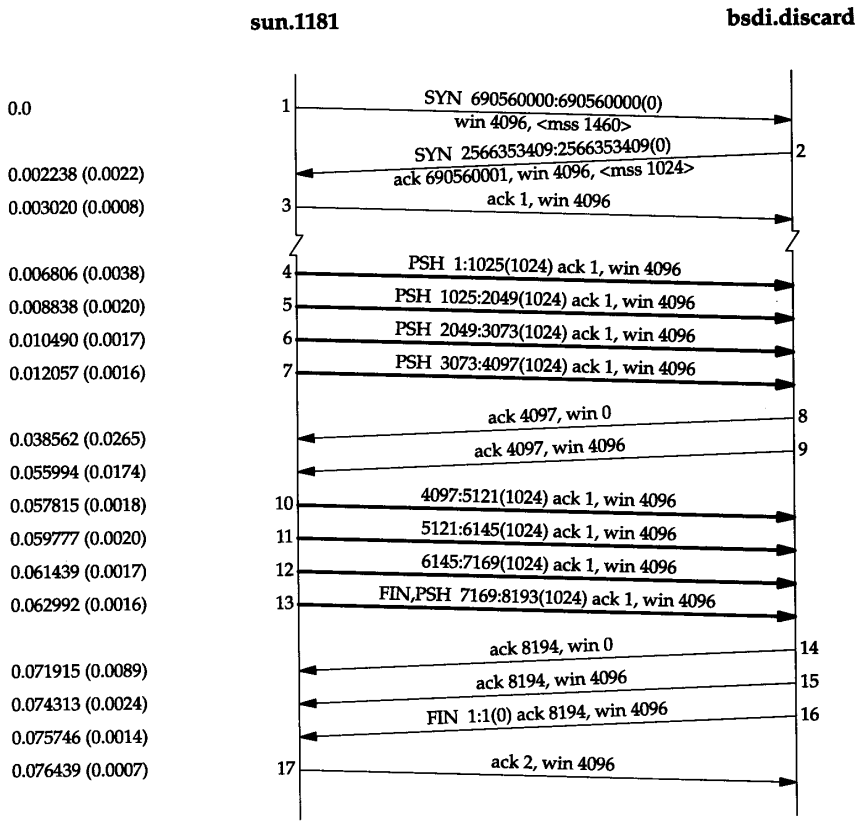

图20-3 从一个快发送方发送8192字节的数据到一个慢接收方

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。

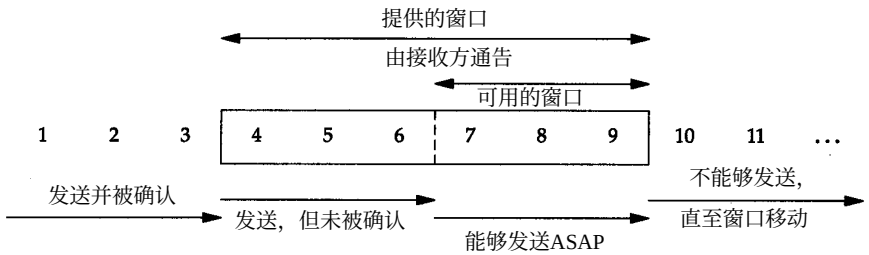

图20-4 TCP滑动窗口的可视化表示

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：

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。

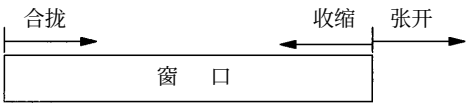

图20-5 窗口边沿的移动

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。

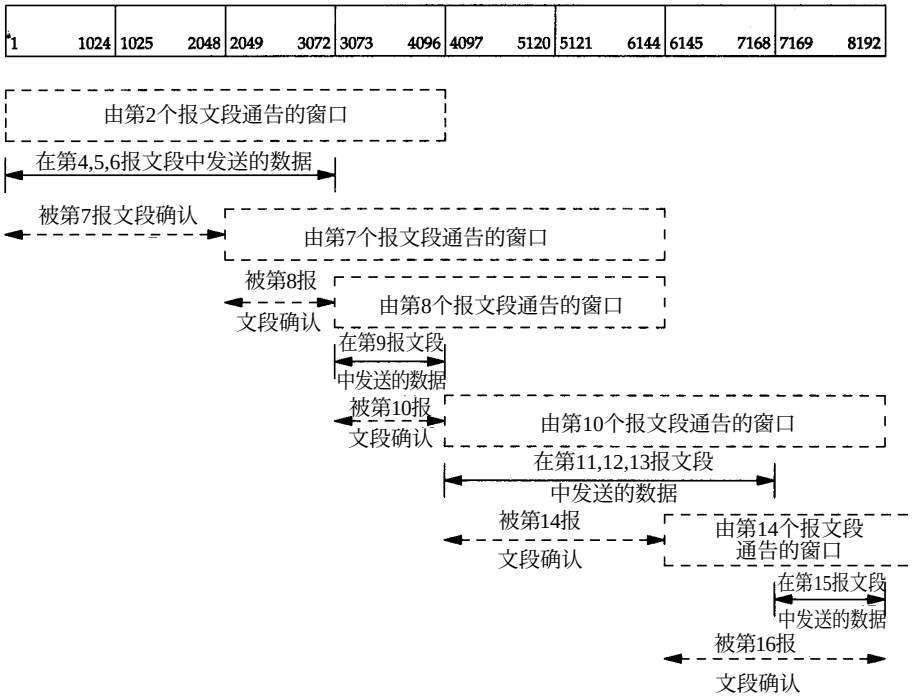

图20-6 图20-1的滑动窗口协议

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。

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。

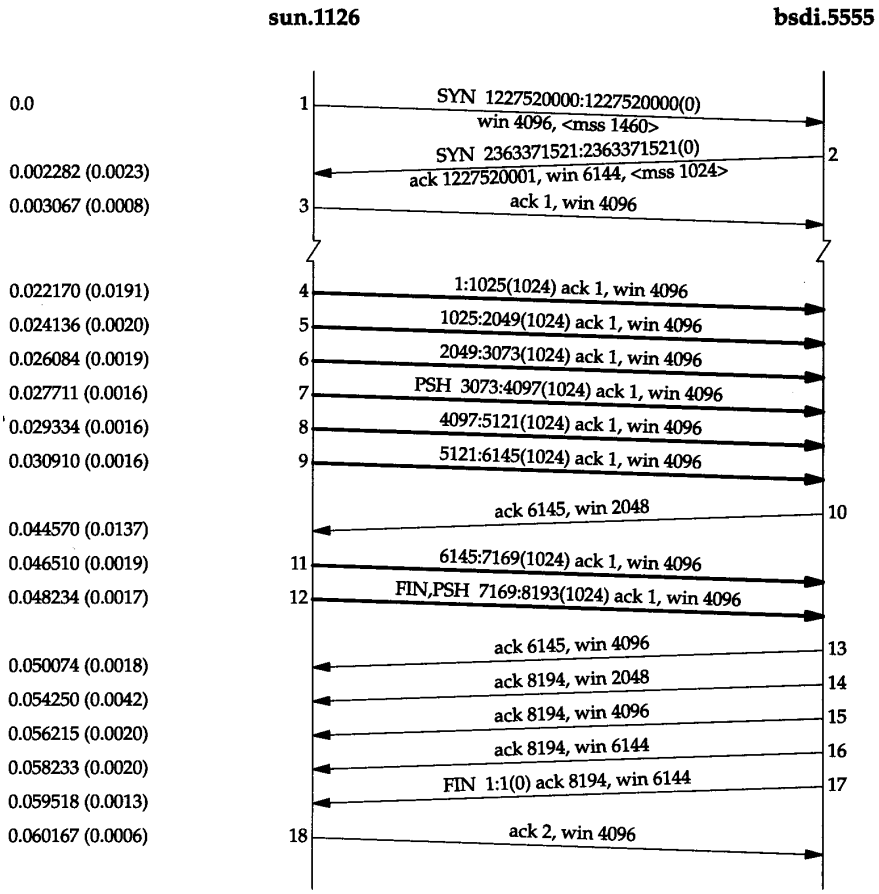

图20-7 接收方提供一个6144字节的接收窗口的情况下的数据传输

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

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

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）。

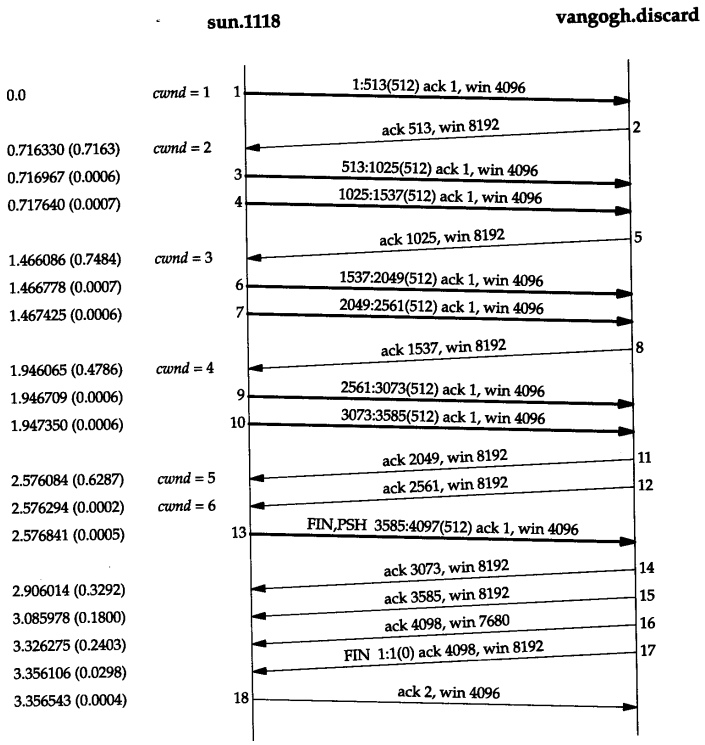

图20-8 慢启动的例子

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

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。

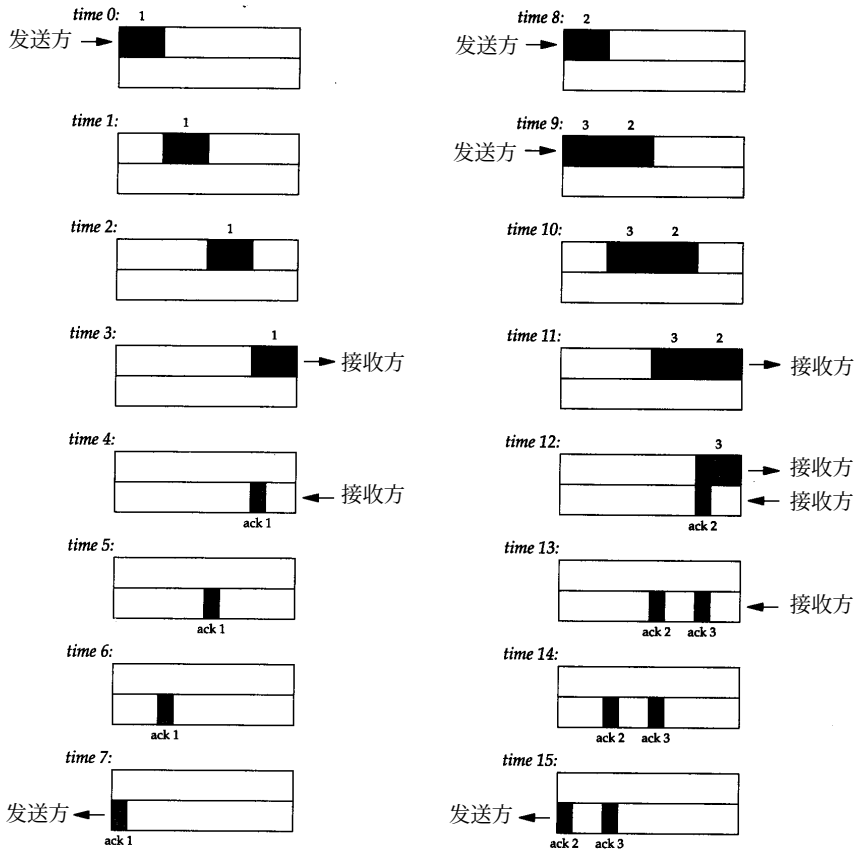

图20-9 时间0~15的成块数据吞吐量举例

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。

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。

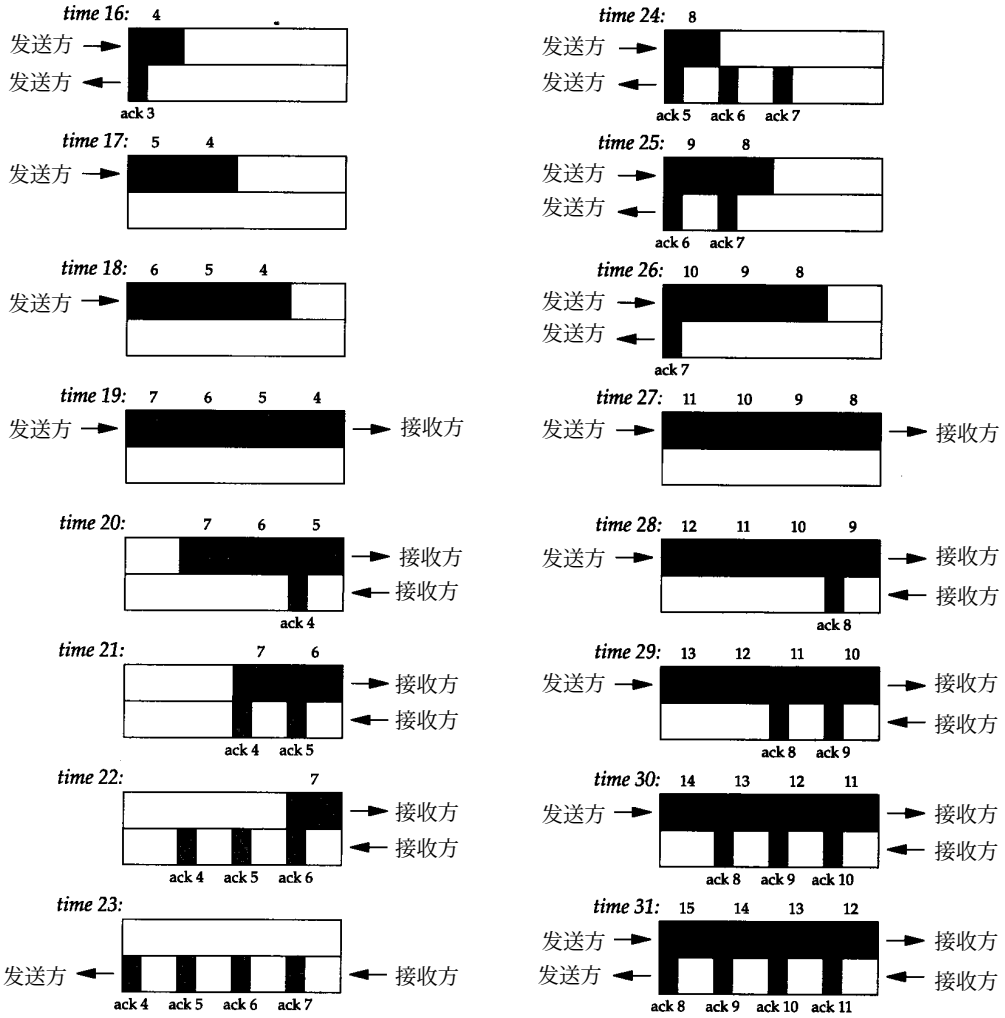

图20-10 时间16~31的成块数据吞吐量举例

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。

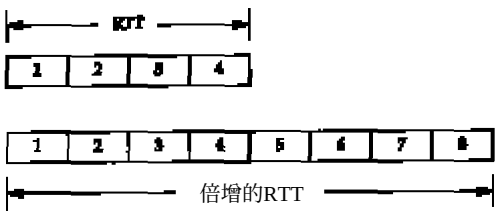

图20-11 RTT加倍可使管道容量增加一倍

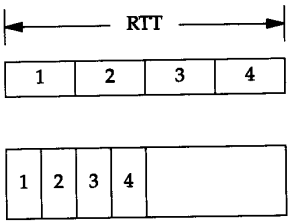

图20-12 带宽加倍可使管道容量增加一倍

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

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）。

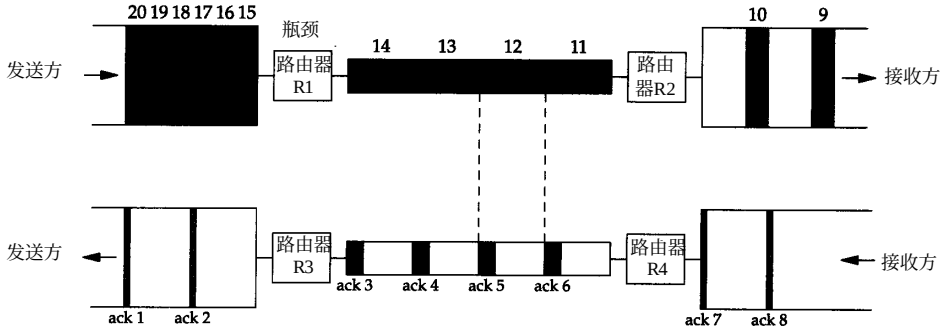

图20-13 从较大管道向较小管道发送分组引起的拥塞

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。

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：

```
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

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。

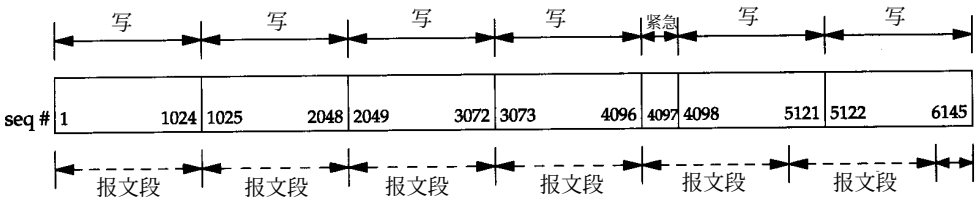

图20-15 紧急方式例子中，应用进程的写操作和TCP的一些报文段

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

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？

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
```

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

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

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。

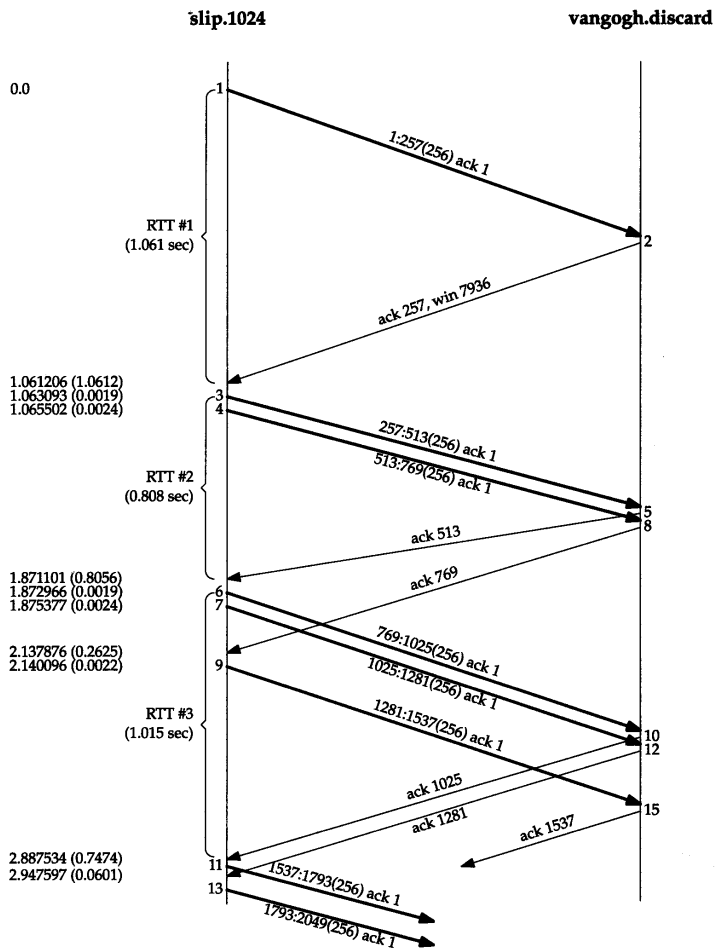

图21-2 分组交换和RTT测量

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

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。

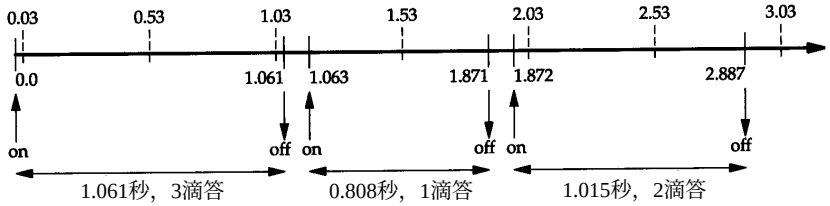

图21-3 RTT测量和时钟滴答

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。

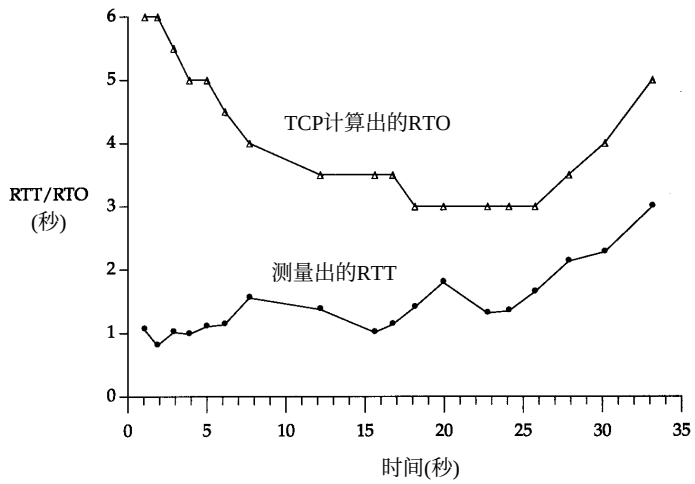

图21-4 测量出的RTT和TCP计算的*RTO*的例子

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

$$RTO = A + 2D = 0 + 2 \times 3 = 6 \text{ s}$$

（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

$$RTO = A + 4D = 0 + 4 \times 3 = 12 \text{ s}$$

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

$$RTO = A + 4D = 2 + 4 \times 1 = 6 \text{ s}$$

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。

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。

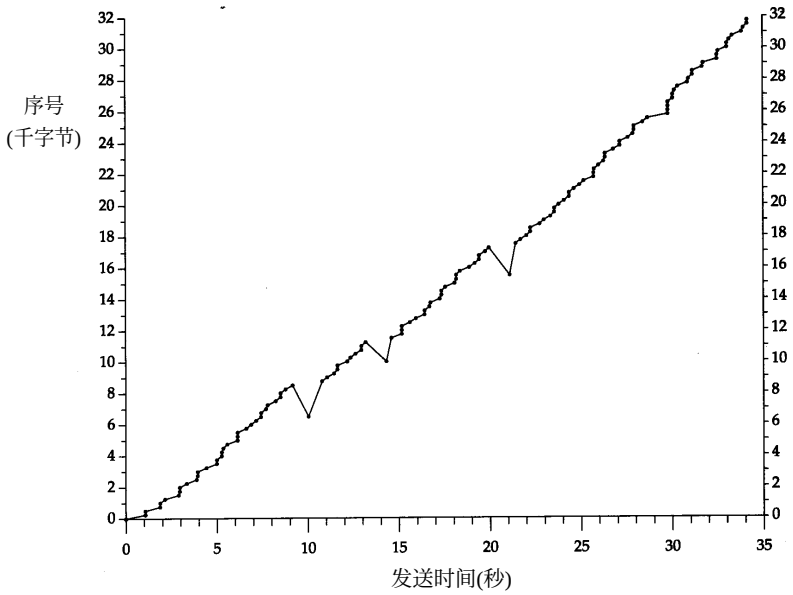

图21-6 从slip 发送32768个字节的数据到vangogh

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

关的段（第44, 47和49以及所有来自vangogh的ACK）。

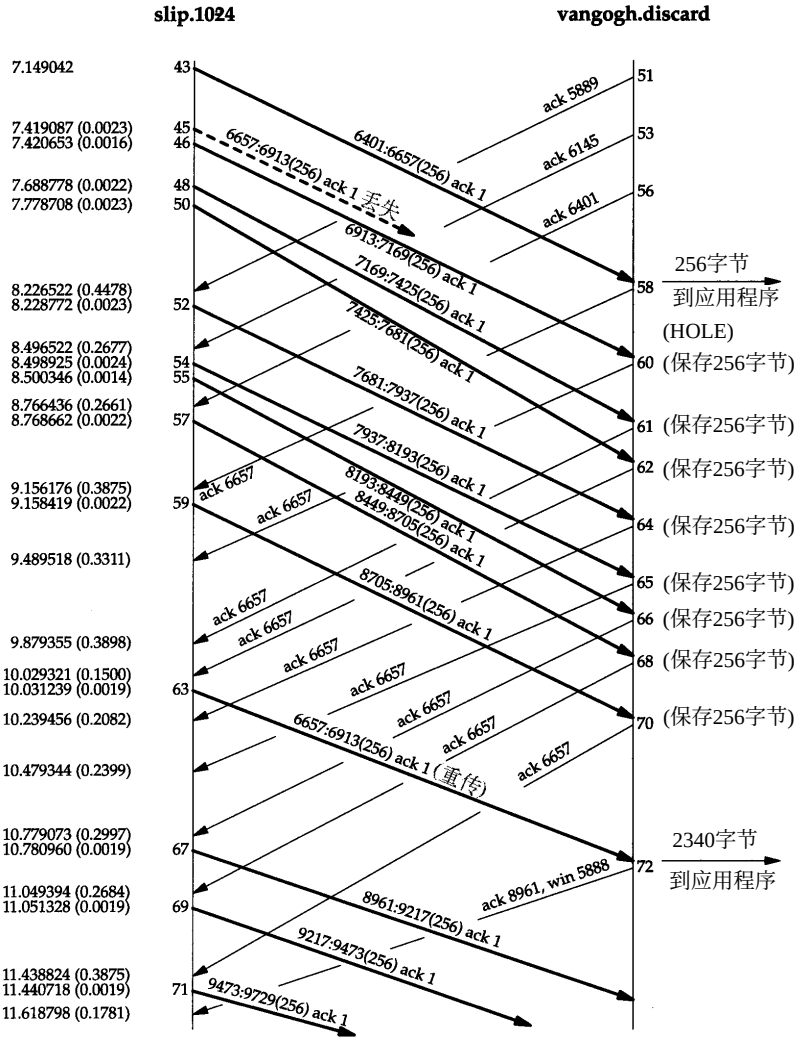

图21-7　10秒标记处附近重传的分组交换

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

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

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。

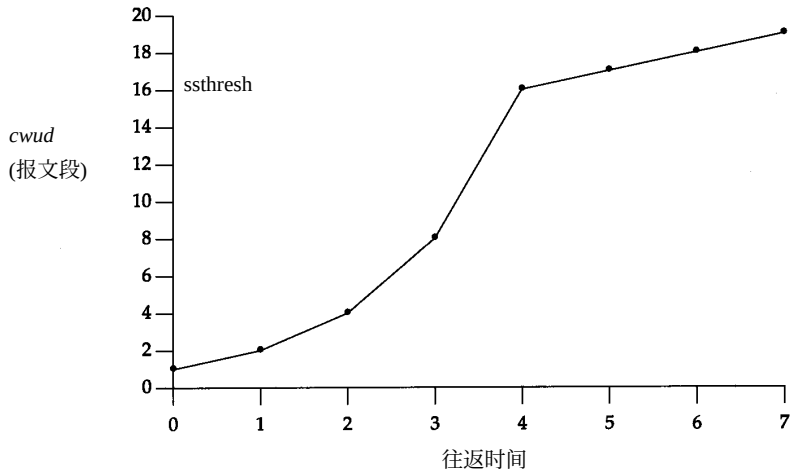

图21-8 慢启动和拥塞避免的可视化描述

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

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）。

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。

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。

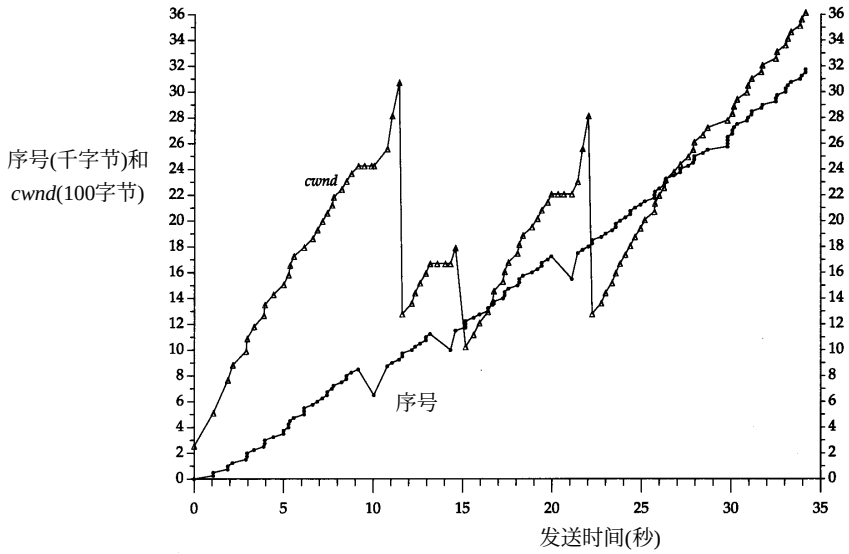

图21-10 当数据被发送时的发送序号和*cwnd*的取值

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	1792	1024
63	6657:6913(256)		重传		
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

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

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。

```
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。

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。

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？