第11章 多　　播

“多播”亦称“多点传送”（Multicasting），是一种让数据从一个成员送出，然后复制给其他多个成员的技术。采用这种技术，可有效减轻网络通信的负担，避免资源的无谓浪费。最开始的时候，设计这一技术的目的是弥补“广播”（Broadcasting）通信的不足。假如过度使用广播技术，极易造成网络带宽的大幅占用，影响整个网络的通信效率。多播通信则不同。对一个网络内的工作站来说，只有在上面运行的进程表示自己“有兴趣”，多播数据才会复制给它们。然而，并非所有协议都支持多播通信，对 Win32平台而言，仅两种可从Winsock内访问的协议（IP和ATM）才提供了对多播通信的支持。通过本章的学习，大家可掌握多播通信的一些基本知识，同时理解如何在这两种协议中实现多播通信。

我们首先探讨多播网络的基本原理，其中涉及多播通信的各种形式，以及 IP及ATM多播通信的基本特征。掌握了最基本的知识后，再分别用两个小节，讲解 IP和ATM特有的一些问题。在本章最后，介绍一些 API调用，用于实现 Winsock 1和Winsock 2中的多播通信。Winsock提供对IP多播的支持已有不短的日子，自第一个版本的 Winsock（Winsock 1）便已开始。到了Winsock 2后，多播接口得到了全面的扩展，具有了“与协议无关”的特点。

支持多播通信的平台包括Windows CE 2.1、Windows 95、Windows 98、Windows NT 4和Windows 2000。自2.1版开始，Windows CE才开始实现对IP多播的支持。在以往的版本中，则没有实现。支持多播的所有平台都能进行 IP多播通信。然而，由于只有在 Windows 98和Windows 2000中，才固化了对ATM Winsock的支持，所以到目前为止，只有这两个平台（含Win98 SE）才能提供对ATM多播通信与生俱来的支持。当然，这并不妨碍我们开发一个 IP多播应用，令其在ATM网络上运行。它对我们造成的唯一限制是：不能编写“与生俱来”或者“固有”的ATM多播代码。这个问题的详情还会在11.2节“IP多播”中讲到。对多播支持来说，另一个需要注意的问题是，某些型号较老的网卡（NIC）不能使用IP多播地址进行数据的发送及接收。1998年之后制造的大多数网卡都提供了对 IP多播的支持，但这也不是绝对的。

11.1 多播的含义

多播通信具有两个层面的重要特征：控制层面和数据层面。其中，“控制层面”（Control Plane）定义了组成员的组织方式；而“数据层面”（Data Plane）决定了在不同的成员之间，数据如何传送。这两方面的特征既可以是“有根的”（Rooted），也可以是“无根的”（Nonrooted）。在一个“有根的”控制层面内，存在着一个特殊的多播组成员，叫作 c_root（控制根，或根节点）。而剩下的每个组成员都叫作 c_leaf（控制叶，或叶节点）。大多数情况下，c_root需负责多播组的建立，其间涉及到建立同任意数量的 c_leaf的连接。而在某些特殊情况下，c_leaf则可在以后的某个时间申请加入一个特定的多播组（或者说，取得那个组的成员资格）。要注意的是，对任何一个具体的组来说，都只能存在一个根节点。 ATM协议便是“有根控制层面”的典型例子。

而对一个“无根的”控制层面来说，它则允许任何人加入一个组，其间不存在任何例外。

在这种情况下，所有组成员均为 c_leaf节点（叶节点）。每个成员都有权加入一个多播组。在一个无根控制层面内，我们可强行实施自己的组成员资格方案（实际效果类似于建立一个c_root根节点），这是通过实施自己的组成员资格协议来实现的。然而，我们的组成员资格方案实际仍然是在一个无根控制层面的基础上建立起来的。 IP多播便是无根控制层面的一个典型例子。在图 11-1中，我们向大家展示了有根与无根控制层面的区别。在位于左上部分的有根控制层面中，c_root必须明确邀请每个c_leaf都加入该组；而在位于右下部分的无根方案中，任何人都能自由加入这个组。

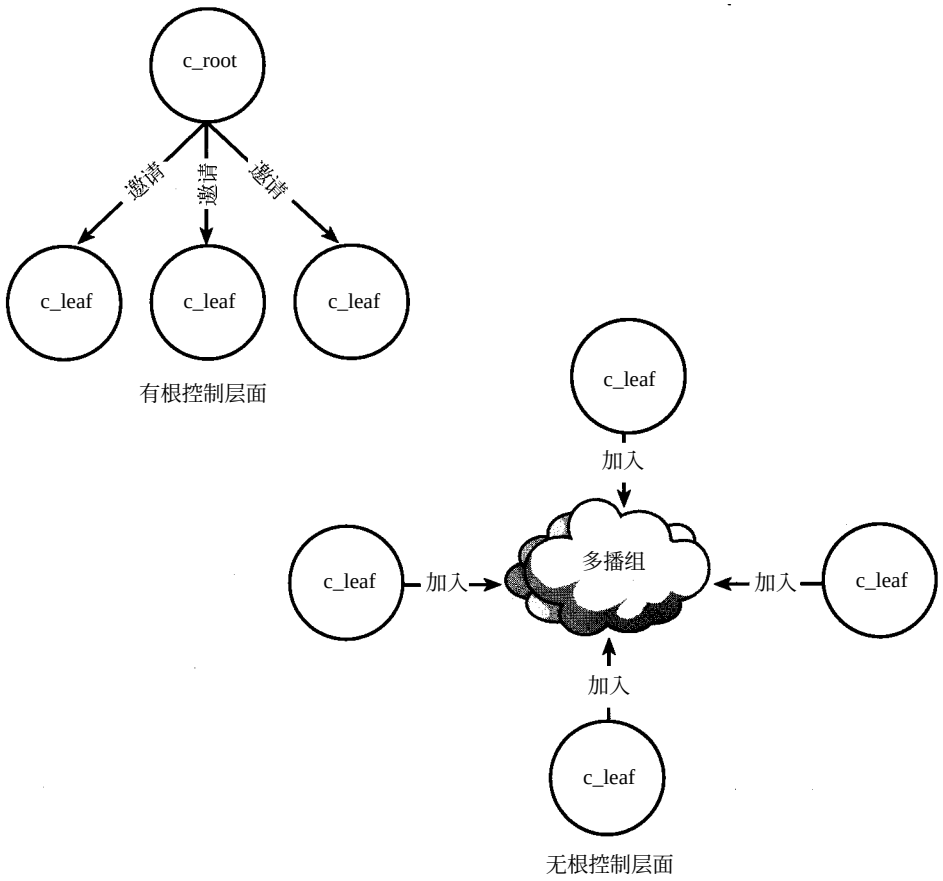

图11-1 有根和无根控制层面

数据层面也存在着“有根的”和“无根的”两种形式。对一个有根数据层面而言，它有一个参与者叫作 d_root（数据根，或根节点）。数据传输只能在 d_root和多播会话的其他所有成员之间进行。显然，那些成员是 d_leaf（数据叶，或叶节点）。这种传输既可单向进行，亦可双向进行。但既然是一个有根数据层面，便暗示着出自一个 d_leaf叶节点的数据只会被d_root根节点接收到；而自 d_root发出的数据却可由每个 d_leaf收到。ATM也是“有根数据层面”的一个典型例子。在图 11-2中，我们展示了有根及无根数据层面的区别。在位于左上部的有根数据层面中，自 d_root发出的数据abc会传送给每一个 d_leaf；而自一个d_leaf发出的数据xyz却只会由d_root接收到，不会“蔓延”到其他叶节点。与此相反的是右下部分的无根示例。其中，数据abc和xyz会“蔓延”到每个成员，无论最开始是由谁发出的数据。

最后，在一个无根数据层面上，所有组成员都能将数据发给组内的其他所有成员。从一个组成员发出的数据块会投递给其他所有成员，同时所有接收者都能回送数据。至于谁能接收或发送数据，则不存在任何限制。同样地， IP多播采用的是数据层面上的“ 无根 ”通信方式。

我们现在知道， ATM多播是在控制及数据层面的一种有根通信方式，而 IP多播在两个层面上都是“ 无根 ”的。除此以外，还有可能存在另一些组合形式。例如，我们可能有一个有根控制层面，其中一个节点决定谁能加入组内；另外还有一个无根数据层面，自任何成员发出的数据都可被其他所有成员“ 看到 ”。只不过，在Winsock中，目前尚无任何一种已经支持的协议以这种形式工作。

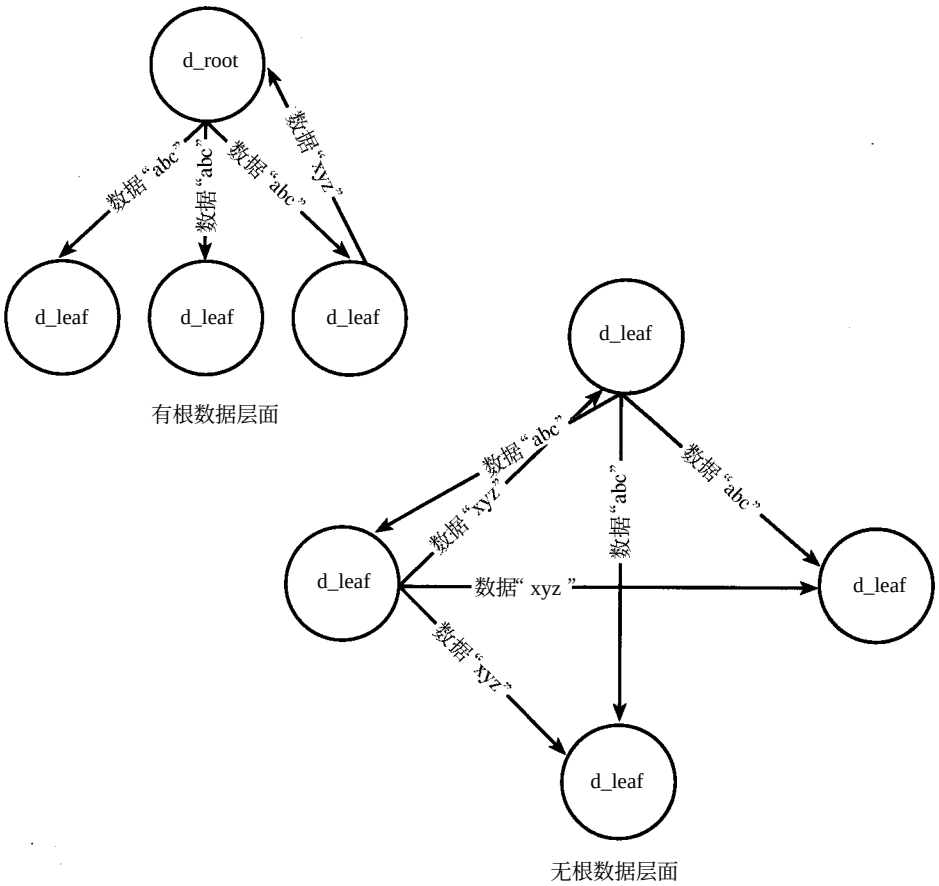

图11-2 有根和无根数据层面

调查多播属性

在第5章，我们已讨论了如何列举协议条目，以及如何决定它们的属性。对一种协议来说，与多播有关的全部信息亦可从目录的协议条目中取得。在由 WSAEnumProtocols调用返回的WSAPROTOCOL_INFO结构中， dwServiceFlags1条目包含了几个特殊的标志位，是我们所特别感兴趣的。假如设置了 XP1_MULTIPOINT_CONTROL_PLANE位，表明协议支持的是一种有根控制层面；否则便是无根的。而假如 XP1_MULTIPOINT_DATA_PLANE位已被设置，表

明协议支持的是一种有根数据层面。类似地，假如标志位设为 0，协议支持的便只有一个无根数据层面。

11.2 IP多播

IP多播通信需要依赖一个特殊的地址组，名为“多播地址”。我们正是用这个组地址对一个指定的组进行命名。举个例子来说，假定五个节点都想通过 IP多播，实现彼此间的通信，它们便可加入同一个组地址。全部加入之后，由一个节点发出的任何数据均会一模一样地复制一份，发给组内的每个成员，甚至包括始发数据的那个节点。多播IP地址是一个D类IP地址，范围在224.0.0.0到239.255.255.255之间。但是，其中还有许多地址是为特殊用途而保留的。比如，224.0.0.0根本没有使用（也不能使用），224.0.0.1代表子网内的所有系统（主机），而224.0.0.2代表子网内的所有路由器。上述最后两个特殊地址只能由 IGMP协议使用，对此我们后面还会详细探讨。在 RFC 1700文件中，提供了所有保留地址的一个详细清单。该文件是为特殊用途保留的所有资源的一个列表，大家可以找来作为参考。“Internet分配数字专家组”（IANA）负责着这个列表的维护。在表 11-1中，我们总结了目前标定为“保留”的一些地址。在实际应用中，可使用除头三个保留多播地址之外的任何地址，因为它们是网络中的路由器专用的。请参考RFC 1700，了解准确的多播地址分配情况。

表11-1 多播地址

多播地址	用 途
224.0.0.0	基本地址（保留）
224.0.0.1	这个子网上的所有系统
224.0.0.2	这个子网上的所有路由器
224.0.1.1	网络时间协议
224.0.0.9	RIP第2版本组地址
224.0.1.24	WINS服务器组地址

最开始设计TCP/IP时，由于尚未考虑多播通信，所以后来不得不进行了大量改造，使 IP能够提供对多播的支持。例如，我们已经发现 IP要求保留一系列特殊地址，专门用于多播数据的传输。除此以外，还专门引入了一个特殊的协议（ IGMP），以便管理多播客户机，以及它们在组内的成员关系。现在，请设想位于不同子网的两个工作站想加入同一个多播组。如何通过IP来做到这一点呢？此时，不能简单地将数据广播到多播地址了事，否则网络会立即由于数据泛滥而瘫痪。开发 Internet网关管理协议（Internet Gateway Management Protocol, IGMP）的目的正是为了通知路由器：网络中的一台机器对发给一个指定组的数据有兴趣。

11.2.1 Internet网关管理协议

多播主机利用IGMP通知路由器，路由器所在子网的一台计算机想加入一个特定的多播组。IGMP是IP多播方案的基础。要想使它正常工作，两个多播节点之间的所有路由器都必须提供对IGMP的支持。例如，假定机器 A和B加入了多播组224.1.2.3，两者间总共存在着三个路由器。此时，三个路由器都必须具有 IGMP能力，以保障通信的成功进行。若某个路由器不具备IGMP能力（即不支持IGMP），那么收到多播数据后，会将它草草地丢弃了事。若一个应用加入了多播组，一条 IGMP“加入”命令便会发给子网上那个特殊的“所有路由器”地址

(224.0.0.2)。该命令用于通知所有路由器，有客户机对一个特定的多播地址产生了兴趣，即它们想加入那个多播组。以后，假如路由器收到了发给那个多播地址的数据，便会将其转发给所有多播客户机。

此外，若一个端点加入多播组，便会同时指定一个“存在时间”(TTL)参数。通过该参数，我们便知道对于在端点机器上运行的多播应用程序来说，为了收发数据，中途需要经历多少个路由器。例如，假定我们编写了一个IP多播应用，令其加入组X，同时TTL值设为2。此时，一个加入命令会传给本地子网上的“所有路由器”组。子网内的路由器会根据这一命令，判断出自己以后应将多播数据转发到那个地址。随后，路由器会将TTL值减1，再将一条加入命令传给与自己相邻的各个网络。那些网络上的路由器会如法炮制，最后又将TTL值减去1。就我们的例子来说，此时的TTL值已经变成了0，所以“加入”命令不会再继续传递下去(不再传给相邻的网络)。从中可以看出，TTL实际限制了多播数据能够蔓延得多“远”。

若一个路由器拥有由工作站注册的一个或多个多播组，便会向“所有主机”组(224.0.0.1)定时发送一条“组查询”消息，查询当初通过一条加入命令通知它的每个多播地址。假如网络上的客户机仍在使用那个多播地址，便会用另一条IGMP消息作出响应，让路由器放心，以便继续转发与那个地址对应的数据。否则的话，路由器便会停止为那个地址转发任何数据。即便客户机通过任何一种Winsock方法，明确离开了那个多播组，路由器上的过滤器设置仍然不会立即消除。

不幸的是，正是由于上述原因，有时才会产生错误：客户机可能在放弃了多播组A的成员资格后，马上便加入了组B。但另一方面，除非路由器执行了一次组查询，但没有接收到响应，否则会将发给多播组A和B的数据都转发到网上。假如这两个组的传输数据总量大于网络本身允许的带宽，岂不是要糟糕？为此，我们可考虑换用IGMP协议的第2版。这是目前的最新版本，允许客户机向路由器发送一条“离开”消息，明确告诉它停止转发指定多播地址的数据。当然，针对每个特定的地址，路由器都维持着一个参考性的客户机计数。因此，除非子网上的所有客户机都脱离了一个特定的地址，否则发给那个地址的数据仍会继续“蔓延”下去。

Windows 98和Windows 2000本身便提供了对IGMP第2版的支持。而对Windows 95来说，在最新的Winsock 2升级中，也包括了IGMP第2版。在Windows NT 4中，自Service Pack 4(SP4)开始，也提供了对IGMP第2版的支持。以前的SP和操作系统本身仅支持第1版。如果想了解IGMP第1及第2版的详情情况，不妨参阅RFC 1112以及RFC 2236这两份标准文档。

11.2.2 IP叶节点

加入IP多播组的过程非常简单，因为每个节点都是一个“叶”，所以在加入一个组的时候，采取的操作步骤都是一样的。由于IP多播在Winsock 1和Winsock 2中都可以实现，所以可通过两种API调用方法，来做成同样的事情。下面是Winsock 1中，实现IP多播通信所需的基本步骤：

1) 用socket函数创建一个套接字，注意要设为AF_INET地址家族以及SOCK_DGRAM套接字类型。要想初始化一个多播套接字，不必设置任何特殊标志，因为socket函数本身便没有提供什么标志参数。

2) 如果想从组内接收数据，请将套接字同一个本地端口绑定到一起。

3) 调用setsockopt函数，同时设置IP_ADD_MEMBERSHIP选项，指定想加入的那个组的

地址结构。

如使用的是Winsock 2，那么步骤1)和2)是相同的，只是步骤3)要改一改——换成调用WSAJoinLeaf函数，将自己加入那个组。至于这两个函数（另一个是setsockopt）以及它们的差别，则将在本章后面11.4节“多播与Winsock”内，进行深入探讨。

请注意，假如一个应用程序只是打算发送数据，便不必加入一个IP多播组。向多播组发送数据时，网络中传输的数据包与普通UDP包大致相同，只是目的地址换成了一个特殊的多播地址而已。但假如想接收多播数据，便必须加入一个组。但无论如何，除了对组成员资格的要求之外，IP多播通信与普通的UDP协议通信并无什么区别：两者都是“无连接”的、“不可靠”的……等等。

11.2.3 IP多播的实施

IP多播通信得到了所有Windows平台的支持（2.1版之前的Windows CE除外）。但另一方面，在各个平台上，IP多播的具体实施方式却稍有差异。我们早先已经指出，使用的网卡首先必须支持多播通信。为此，网卡需要以硬件形式，为接口增加多播过滤器。多播IP地址采用了一个特殊的MAC地址，其中包含了编好码的IP地址，使网卡能轻易判断出进入的数据包是否属于多播数据，同时调查这个包的目标多播IP地址是哪一个。所有这些信息只需检查MAC头便可取得。在RFC 1700中，对具体的编码方案进行了讨论。本书不打算再继续深入下去，因为它与Winsock API并无多大联系。

然而，Windows 95和Windows NT 4的多播实施方式却有所不同，这是通过将网卡置于“混杂”模式来进行的。换句话说，网卡会通过信号线缆取得传来的所有数据包，再一起转交给网络驱动程序，由后者负责检查MAC头的内容，看看是否有多播数据发给一个多播组，而且工作站上的一个进程属于那个组的成员。由于这种过滤实际是由软件（驱动程序）完成的，性能当然比不上用硬件来过滤数据包。因此，Windows 98和Windows 2000采取了不同的实施方式。两者均利用了网卡本身的能力，来实现过滤器的增添。由于在硬件一级执行，所以效能当然要高出许多。目前大多数网卡都能支持16或32个硬件过滤器。对Windows 98来说，唯一的限制是一旦设置了所有硬件过滤器，机器上运行的进程便无法再加入任何新的多播组。假如超出了硬件本身的限制，多播加入操作就会失败，并返回一个WSAENOBUFS错误。在这一方面，Windows 2000显得更为“健壮”。假如所有硬件过滤器都被占用，Windows 2000就会像Windows 95和Windows NT 4那样，将网卡置为“混杂”模式，通过软件实现MAC头的解析。

11.3 ATM多播

通过Winsock实现的ATM通信也支持多播通信方式，它与IP多播在功能上有着显著的不同。前面已经说过，ATM支持“有根的”控制及数据层面。换言之，若建立了一个多播服务器（或者c_root），便能由它控制谁能加入一个多播组，以及数据在组内如何传输。

与IP多播的一项重要区别在于，在ATM网络上，可通过ATM实现对IP的支持。采用这种配置方式，ATM网络便能仿真一个IP网络，而IP地址实际上会映射成为ATM特有的地址形式。若允许通过ATM来仿真IP，那么可考虑继续使用IP多播通信。此时，IP多播会相应地转换到ATM层。当然，亦可考虑使用“正宗”的ATM多播通信机制（本节讨论的主题）。如配置成通

过ATM实现IP多播，那么IP多播通信的“行为”和在其他普通IP网络上基本一般无异。唯一的区别在于，在此不必使用IGMP，因为所有多播调用都会转换成ATM自己的命令。另外，我们或许能将一个ATM网络配置成一个或多个“LAN仿真（LANE）”网络。设计LANE最主要的目的便是让ATM表现得像一个“标准”网络，能够处理IPX/SPX、NetBEUI、TCP/IP、IGMP、ICMP……等协议。在这种情况下，IP多播与以太网上的IP多播通信真的是毫无差别；换言之，IGMP亦可正常使用。

前面已经提到，ATM支持有根控制层面和有根数据层面。这意味着在我们创建一个多播组的时候，需要建立一个根节点，由它负责“邀请”叶节点加入多播组。在新版的Windows 2000中，目前只支持由“根”发起的成员加入。换言之，“叶”不可自己请求加入一个组。除此以外，根节点（作为一个有根的数据层面）只能朝一个方向发送数据——从根到叶！

在ATM和IP多播之间，一项重大差别是ATM不需要特殊地址。唯一的要求是：作为一个“根”，它必须知道自己打算邀请的每一个叶的地址。此外，一个多播组内仅能有一个根节点存在。若另一个ATM端点发出了对相同的“叶”的邀请，便会自动与原先的多播组脱勾。两者均转移到一个单独的组内。

11.3.1 ATM叶节点

在多播组内创建一个叶节点是件轻而易举的事情。在ATM网络中，作为一个“叶”，必须随时随地监听来自一个“根”的、要自己加入一个组的邀请。下面列出必要的一系列步骤：

1) 使用WSASocket函数，创建地址家族AF_ATM的一个套接字，同时设置WSA_FLAG_MULTIPOINT_C_LEAF和WSA_FLAG_MULTIPOINT_D_LEAF这两个标志。

2) 将套接字同本地ATM地址及端口绑定到一块儿，这是用bind函数完成的。

3) 调用listen（监听）命令。

4) 用accept或WSAAccept等候邀请。至于具体用哪个命令，要取决于我们使用的是什么I/O（输入／输出）模型。要想全盘了解Winsock I/O模型，请参阅本书第8章。

建好连接后，叶节点便可开始接收来自根的数据。要记住的是，对ATM多播来说，数据流必须是单向的：由根至叶，不可逆反。

注意 目前，在任何指定时刻，Windows 98和Windows 2000仅能支持一个ATM叶节点。换言之，对任何ATM“点到多点”会话来说，在整个系统的范围之内，只能有一个进程成为它的叶成员。

11.3.2 ATM根节点

根节点的创建过程甚至还要比ATM叶的创建简单一些。基本步骤如下：

1) 使用WSASocket函数，创建地址家族AF_ATM的一个套接字，同时设置WSA_FLAG_MULTIPOINT_C_ROOT和WSA_FLAG_MULTIPOINT_D_ROOT这两个标志。

2) 针对希望邀请的每一个端点，都随ATM地址一道，调用WSAJoinLeaf。

根节点可根据自己的需要，邀请任意数量的端点加入。然而，对每个叶节点，都必须单独执行一次WSAJoinLeaf调用。

11.4 多播与Winsock

现在，大家已基本熟悉了Windows平台的两类基本多播通信方式。接下来，且让我们来看看Winsock提供了哪些API调用，以便真正实现多播。针对IP多播，Winsock提供了两种不同的实现方法，具体取决于使用的是哪个版本的Winsock。第一种方法是Winsock 1提供的，要求通过套接字选项来加入一个组。Winsock 2则引入了一个新函数，专门负责多播组的加入。这个函数便是WSAJoinLeaf，它与基层协议是无关的。下面，我们首先要讲述的是Winsock 1方法，也是应用得最广的一种方法（特别由于它是自Berkeley套接字衍生出来的）。

11.4.1 Winsock 1多播

在Winsock 1设计的方法中，IP多播组的加入和离开是用setsockopt命令来完成的，同时要用到IP_ADD_MEMBERSHIP（加入组）和IP_DROP_MEMBERSHIP（脱离组）这两个选项。使用这两个套接字选项时，必须传递一个ip_mreq结构，它的定义如下：

```
struct ip_mreq
{
    struct in_addr  imr_multiaddr;
    struct in_addr  imr_interface;
};
```

其中，imr_multiaddr字段用于指定要加入的多播组，而imr_interface指定要通过它送出多播数据的本地（或本机）接口。如果将imr_interface设为INADDR_ANY，便会自动使用默认接口；否则，请指定本地接口具体要使用哪个IP地址（假如同时有多个IP地址可选）。下面示范代码向大家展示了如何加入一个多播组234.5.6.7：

```
SOCKET            s;
struct ip_mreq    ipmr;
SOCKADDR_IN       local;
int               len = sizeof(ipmr);

s = socket(AF_INET, SOCK_DGRAM, 0);

local.sin_family = AF_INET;
local.sin_addr.s_addr = htonl(INADDR_ANY);
local.sin_port = htons(10000);
ipmr.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
ipmr.imr_interface.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&local, sizeof(local));
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&ipmr, &len);
```

要离开一个多播组，只需调用setsockopt，同时设置IP_DROP_MEMBERSHIP。注意仍然要传递一个ip_mreq结构，其中的值与加入那个组时是一样的。如下例所示：

```
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP,
    (char *)&ipmr, &len);
```

Winsock 1 IP多播示例

在程序清单11-1中，我们提供了一个简单的IP多播示范程序。该程序首先加入一个指定的组，然后扮演一个发送者或接收者（具体由命令行参数决定）。写这个程序时，我们让一个发

送者只负责数据的发送，而一个接收者只是利用一个循环结构，简单地等候进入的数据。显然，按照这一设计，我们没有展示出假如发送者将数据发给多播组；同时也作为那个组的成员，对进入数据进行监听的情况。发送的数据会返还给发送者自己的“进入数据”队列。我们称这种情况为“多播返还”（Multicast Loopback）。要注意的是，只有在我们使用IP多播通信时，才会出现这一情况。本章稍后，还会讲到如何禁止进行这种“返还”。

程序清单11-1 Winsock 1.1多播示例

```
// Module name: Mcastws1.c

#include <windows.h>
#include <winsock.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR      "234.5.6.7"
#define MCASTPORT       25000
#define BUFSIZE         1024
#define DEFAULT_COUNT   500

BOOL  bSender = FALSE,       // Act as sender?
      bLoopBack = FALSE;     // Disable loopback?

DWORD dwInterface,           // Local interface to bind to
      dwMulticastGroup,      // Multicast group to join
      dwCount;               // Number of messages to send/receive

short iPort;                 // Port number to use
//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s -s -m:str -p:int -i:str -l -n:int\n",
        progname);
    printf(" -s     Act as server (send data); otherwise\n");
    printf("          receive data.\n");
    printf(" -m:str Dotted decimal multicast IP addres to join\n");
    printf("          The default group is: %s\n", MCASTADDR);
    printf(" -p:int Port number to use\n");
    printf("          The default port is: %d\n", MCASTPORT);
    printf(" -i:str Local interface to bind to; by default \n");
    printf("          use INADDRY_ANY\n");
    printf(" -l     Disable loopback\n");
    printf(" -n:int Number of messages to send/receive\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
```

```
//
// Description:
//    Parse the command line arguments, and set some global flags,
//    depending on the values
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    iPort = MCASTPORT;
    dwCount = DEFAULT_COUNT;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':   // Sender
                    bSender = TRUE;
                    break;
                case 'm':   // Multicast group to join
                    if (strlen(argv[i]) > 3)
                        dwMulticastGroup = inet_addr(&argv[i][3]);
                    break;
                case 'i':   // Local interface to use
                    if (strlen(argv[i]) > 3)
                        dwInterface = inet_addr(&argv[i][3]);
                    break;
                case 'p':   // Port number to use
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'l':   // Disable loopback?
                    bLoopBack = TRUE;
                    break;
                case 'n':   // Number of messages to send/recv
                    dwCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: main
//
// Description:
//    Parse the command line arguments, load the Winsock library,
```

```
//    create a socket, and join the multicast group. If this program
//    is started as a sender, begin sending messages to the
//    multicast group; otherwise, call recvfrom() to read messages
//    sent to the group.
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    struct sockaddr_in  local,
                        remote,
                        from;
    struct ip_mreq      mcast;
    SOCKET              sockM;
    TCHAR               recvbuf[BUFSIZE],
                    sendbuf[BUFSIZE];
int                 len = sizeof(struct sockaddr_in),
                    optval,
                    ret;
DWORD               i=0;

// Parse the command line, and load Winsock
//
ValidateArgs(argc, argv);

if (WSAStartup(MAKEWORD(1, 1), &wsd) != 0)
{
    printf("WSAStartup failed\n");
    return -1;
}
// Create the socket. In Winsock 1, you don't need any special
// flags to indicate multicasting.
//
if ((sockM = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
{
    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
// Bind the socket to the local interface. This is done so
// that we can receive data.
//
local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sockM, (struct sockaddr *)&local,
    sizeof(local)) == SOCKET_ERROR)
{
    printf("bind failed with: %d\n", WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}

// Set up the im_req structure to indicate what group we want
```

```
// to join as well as the interface
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;
mcast.imr_multiaddr.s_addr = dwMulticastGroup;
mcast.imr_interface.s_addr = dwInterface;

if (setsockopt(sockM, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_ADD_MEMBERSHIP) failed: %d\n",
        WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}
// Set the TTL to something else. The default TTL is 1.
//
optval = 8;
if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_TTL,
    (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
        WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}
// Disable the loopback if selected. Note that in Windows NT 4
// and Windows 95 you cannot disable it.
//
if (bLoopBack)
{
    optval = 0;
    if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
            WSAGetLastError());
        closesocket(sockM);
        WSACleanup();
        return -1;
    }
}

if (!bSender)           // Client
{
    // Receive some data
    //
    for(i = 0; i < dwCount; i++)
    {
            if ((ret = recvfrom(sockM, recvbuf, BUFSIZE, 0,
                (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
            {
```

```
                printf("recvfrom failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                WSACleanup();
                return -1;
            }
            recvbuf[ret] = 0;
            printf("RECV: '%s' from <%s>\n", recvbuf,
                inet_ntoa(from.sin_addr));
        }
    }
    else                    // Server
    {
        // Send some data
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "server 1: This is a test: %d", i);
            if (sendto(sockM, (char *)sendbuf, strlen(sendbuf), 0,
                (struct sockaddr *)&remote,
                sizeof(remote)) == SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                WSACleanup();
                return -1;
            }
            Sleep(500);
        }
    }
    // Drop group membership
    //
    if (setsockopt(sockM, IPPROTO_IP, IP_DROP_MEMBERSHIP,
        (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_DROP_MEMBERSHIP) failed: %d\n",
            WSAGetLastError());
    }
    closesocket(sockM);

    WSACleanup();
    return 0;
}
```

用Winsock 1的方法来实现多播通信时，必须特别注意：使用正确的文件和库进行链接。如果装载的是Winsock 1.1库，便应将Winsock.h包括进来，并同Wsock32.lib建立链接关系。假如装载的是 Winsock 2或更高版本的库，便需同时包括 Winsock.h和Ws2tcpip.h，并同Ws2_32.lib建立链接关系。这样做是必需的，因为针对下述常数，同时存在着两套不同的值：IP_ADD_MEMBERSHIP、IP_DROP_MEMBERSHIP、IP_MULTICAST_IF和IP_MULTICAST_LOOP。而这些值原来的规定——由Stephen Deering制订——却从未正式集成到Winsock规范中来。所以到了 Winsock 2之后，这些值发生了改变。当然，假如使用的是老版本的Winsock，那么只需建立与 wsock32.lib的链接，便万事大吉。转到 Winsock 2机器上

运行时，常数会转换成正确的值。

11.4.2 Winsock 2多播

Winsock 2多播通信要比Winsock 1多播通信稍微复杂一些。但是，前者提供了一些有用的机制，可支持一些用于提供附加特性的协议，比如“服务质量”（QoS）等等。另外，它支持的一些协议还提供了“有根多播”能力。套接字选项在新版的 Winsock中不再用于对组成员关系进行初始化。相反，我们要用一个新增的函数（ WSAJoinLeaf）来做这件事情。该函数的原型定义如下：

```
SOCKET WSAJoinLeaf(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    DWORD dwFlags
);
```

其中，第一个参数s代表一个套接字句柄，是自 WSASocket返回的。传递进来的这个套接字必须使用恰当的多播标志进行创建；否则的话， WSAJoinLeaf就会失败，并返回错误WSAEINVAL。必须记住的是，我们需要指定两个多点传送标志：一个用于指出该套接字在控制层面是“有根”的，还是“无根”的；另一个用于指出该套接字在数据层面是“有根”的，还是“无根”的。用于指定控制层面的标志是 WSA_FLAG_MULTIPOINT_C_ROOT和WSA_FLAG_MULTIPOINT_C_LEAF，而用于指定数据层面的标志是 WSA_FLAG_MULTIPOINT_D_ROOT和WSA_FLAG_MULTIPOINT_D_LEAF。第二个参数是SOCKADDR（套接字地址）结构，具体内容由当前采用的协议决定。对有根控制方案来说（比如 ATM），这个地址指定了打算邀请加入的客户机；而对无根控制方案来说（比如 IP），这个地址指定的是主机打算加入的那个多播组。 namelen（名字长度）参数的意义并不难猜：用于指定 name参数的长度，以字节为单位。 lpCallerData（呼叫者数据）参数的作用是在会话建好之后，将一个数据缓冲区传输给和自己通信的对方。而 lpCalleeData（被叫者数据）参数用于初始化一个缓冲区，在会话建好之后，接收来自对方的数据。注意在当前的 Windows平台上，这两个参数并未真正实现，所以均应设为 NULL。lpSQOS参数用于指定一个FLOWSPEC结构，指出应用程序要求多大的带宽（第 12章对QoS或服务质量进行了详细讲述；简单地说，带宽越大，网络应用享受的服务质量就越高）。lpGQOS参数在此会被忽略，因为当前没有任何一种Windows平台支持所谓的“套接字组”。而最后一个参数dwFlags指出该主机是发送数据、接收数据或收发兼并。为此，该参数的可选值分别是：JL_SENDER_ONLY、JL_RECEIVER_ONLY或者JL_BOTY。

针对同多播组绑定到一起的套接字，该函数会返回一个 SOCKET描述符。假如WSAJoinLeaf调用是通过一个异步（或非锁定）套接字进行的，那么除非加入操作完成，否则返回的套接字描述符是没有用的。例如，假定套接字处于异步模式（来自 WSAAsyncSelect或WSAEventSelect调用），那么若非原来的套接字s接收到一个对应的FD_CONNECT通知，否则描述符便是无效的。注意 FD_CONNECT通知只有在有根控制方案中才会生成。在这种方案

中，name参数指定了一个特定端点的地址。在表 11-2中，我们总结了在什么样的前提下，应用程序才会接收到一个 FD_CONNECT通知。通过在原来的套接字上调用 closesocket，我们可为这些非锁定模式取消一个“ 待决 ”的加入请求。而一个多点会话中的根节点可调用 WSAJoinLeaf一次或者多次，以便增加新的叶节点。但大多数时候，一次只能有一个多点连接请求处于“ 待决 ”状态。

如先前所述，对非锁定套接字来说，一次加入请求不能立即完成。同时，除非请求完成，否则返回的套接字描述符是无法使用的。假如我们通过 ioctlsocket命令FIONBIO将套接字置入非锁定模式，对 WSAJoinLeaf的调用便不会返回 WSAEWOULDBLOCK错误，因为函数实际已返回了一个“ 成功启动 ”的指示。注意在一个异步 I/O模型中，要想接收有关成功启动的通知，唯一的方法便是通过 FD_CONNECT消息。大家可参考第 8章，了解WSAAsyncSelect和 WSAEventSelect异步I/O模型的详情。对于处在锁定模式中的套接字来说，无论 WSAJoinLeaf操作是成功还是失败，都不能向应用程序发出相应的通知。换言之，我们或许应该避免直接的非锁定套接字，因为除非在后续的 Winsock调用中实际用到套接字，否则便无法确切判断一个多播加入操作是否成功（若加入操作失败，使用时也会失败）。

WSAJoinLeaf返回的套接字描述符是各不相同的，具体取决于输入套接字是一个根节点，还是一个叶节点。如果是根节点，那么根据 name参数，我们就可知道打算邀请加入多点会话的一个特定的“ 叶 ”的地址。为了使 c_root（根节点）能够对各个叶的成员关系（成员资格）进行维护， WSAJoinLeaf会为叶返回一个新的套接字句柄。这个新套接字与邀请时使用的根套接字完全相同。其中包括通过异步 I/O模型注册的任何异步事件，比如 WSAEventSelect和 WSAAsyncSelect。然而，这些新的套接字只能用来从“ 叶 ”那里收取 FD_CLOSE通知消息。发给多播组的任何数据都应通过 c_root套接字送出。某些情况下，我们可通过由 WSAJoinLeaf返回的套接字送出数据，但只有与那个套接字对应的“ 叶 ”才会真正收到数据。 ATM协议便允许这样干。最后，要想从多播组中删去一个叶节点，那么作为“ 根 ”，只需在与那个叶对应的套接字上调用closesocket就可以了。

另一方面，如果随一个叶节点调用 WSAJoinLeaf，那么name参数对应的要么是一个根节点的地址，要么是一个多播组的地址。前一种情况指定一次由叶发起的加入，目前尚无任何一种协议提供了对它的支持（然而，以后的 ATM UNI4.0规范会提供这方面的支持）。后一种情况指定的则是 IP多播通信的工作方式。但无论在哪种情况下，自 WSAJoinLeaf返回的套接字句柄与作为s传递的套接字句柄都是相同的。如果对 WSAJoinLeaf的调用是一次由叶节点发起的加入，那么根节点会用 bind，listen和accept/WSAAccept方法来“ 监听 ”进入的连接请求。上述几种方法对一个服务器来说，应该是大家早已耳熟能详的！若应用程序想把自己从多点会话中删去，那么需要在套接字上调用 closesocket，用它终止成员关系，同时也释放出被占用的套接字资源。在表 11-2中，我们也总结了根据控制层面的类型，根据套接字传递的参数，同时根据name参数，需采取什么样的行动。这些行动包括：函数调用成功之后，是否返回一个新的套接字描述符；以及应用程序是否会收到一个 FD_CONNECT通知等。

若应用程序调用accept或者WSAAccept函数，以便等候由根发起的一次邀请（此时自己的身份是“ 叶 ”），或者等候来自叶的加入请求（此时的身份是“ 根 ”），函数便会返回一个套接字，亦即一个c_leaf套接字描述符，与WSAJoinLeaf返回的类似。若协议既允许由根发起加入，也允许由叶发起加入，那么为了与这种协议适应，对已处在“ 监听 ”模式的一个 c_root套接字

来说，完全可将其当作WSAJoinLeaf的一个输入来使用。

发出了对WSAJoinLeaf的调用之后，会返回一个新的套接字句柄。该句柄并不用于数据的收发。对于原来的套接字句柄来说（自 WSASocket返回，传递进入WSAJoinLeaf），它会在所有的数据发送及接收操作中使用。新句柄指出我们是指定多播组的一名成员 —— 只要句柄有效。若针对新句柄调用 closesocket，会使我们的应用程序脱离多播组，失去它的成员资格。或在一个c_root套接字上调用closesocket，那么凡是采用了异步 I/O模型的c_leaf节点都会收到一个FD_CLOSE通知。

表11-2 WSAJoinLeaf行动总结

控制层面	s	name参数（名字）	行 动	是否收到FD_CONNECT通知？	返回的套接字描述符
有根的	c_root	叶地址	由根邀请一个叶加入	是	用于FD_CLOSE通知，以及用于将私人数据传给那个叶s的一个副本
	c_leaf	根地址	由叶发起与根的连接	是	
无根的	c_root	N/A	不可能的一种组合	N/A	N/A
	c_leaf	组地址	叶加入一个组	否	s的一个副本

1. Winsock 2 IP多播示例

在程序清单11-2中，我们向大家展示了Mcastws2.c的源码清单。该程序实际解释了如何使用WSAJoinLeaf命令，来加入及脱离一个IP多播组。这实际是在一个简单的Winsock 1 IP多播示例基础上改编而成的。那个例子叫作 Mcastws1.c。新程序的不同之处在于，加入／离开调用已进行了修改，改为使用本节讲述的 WSAJoinLeaf。

程序清单11-2 IP多播示例

```
// Module: Mcastws2.c
//
#include <winsock2.h>
#include <ws2tcpip.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR     "234.5.6.7"
#define MCASTPORT      25000
#define BUFSIZE        1024
#define DEFAULT_COUNT  500

BOOL  bSender = FALSE,       // Act as a sender?
      bLoopBack = FALSE;     // Disable loopback?

DWORD dwInterface,           // Local interface to bind to
      dwMulticastGroup,      // Multicast group to join
      dwCount;               // Number of messages to send/receive

short iPort;                 // Port number to use

//
```

```
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s -s -m:str -p:int -i:str -l -n:int\n",
        progname);
    printf("   -s        Act as server (send data); otherwise\n");
    printf("              receive data.\n");
    printf("   -m:str    Dotted decimal multicast IP addres "
       "to join\n");
    printf("              The default group is: %s\n", MCASTADDR);
    printf("   -p:int    Port number to use\n");
    printf("              The default port is: %d\n", MCASTPORT);
    printf("   -i:str    Local interface to bind to; by default \n");
    printf("              use INADDRY_ANY\n");
    printf("   -l        Disable loopback\n");
    printf("   -n:int    Number of messages to send/receive\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags,
//    depending on the values
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    iPort = MCASTPORT;
    dwCount = DEFAULT_COUNT;

    for(i=1; i < argc ;i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':  // Sender
                    bSender = TRUE;
                    break;
                case 'm':  // Multicast group to join
                    if (strlen(argv[i]) > 3)
                        dwMulticastGroup = inet_addr(&argv[i][3]);
                    break;
                case 'i':  // Local interface to use
                    if (strlen(argv[i]) > 3)
                        dwInterface = inet_addr(&argv[i][3]);
```

```
                    break;
                case 'p':  // Port to use
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'l':  // Disable loopback
                    bLoopBack = TRUE;
                    break;
                case 'n':  // Number of messages to send/recv
                    dwCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: main
//
// Description:
//    Parse the command line arguments, load the Winsock library,
//    create a socket, and join the multicast group. If this program
//    is run as a sender, begin sending messages to the multicast
//    group; otherwise, call recvfrom() to read messages sent to the
//    group.
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    struct sockaddr_in  local,
                        remote,
                        from;
    SOCKET              sock, sockM;
    TCHAR               recvbuf[BUFSIZE],
                        sendbuf[BUFSIZE];
    int                 len = sizeof(struct sockaddr_in),
                        optval,
                        ret;
DWORD                   i=0;

// Parse the command line, and load Winsock
//
ValidateArgs(argc, argv);

if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
{
    printf("WSAStartup() failed\n");
    return -1;
}
// Create the socket. In Winsock 2, you do have to specify the
// multicast attributes that this socket will be used with.
```

```
//
if ((sock = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0,
              WSA_FLAG_MULTIPOINT_C_LEAF
            | WSA_FLAG_MULTIPOINT_D_LEAF
            | WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
// Bind to the local interface. This is done to receive data.
local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sock, (struct sockaddr *)&local,
    sizeof(local)) == SOCKET_ERROR)
{
    printf("bind failed with: %d\n", WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}
// Set up the SOCKADDR_IN structure describing the multicast
// group we want to join
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;
//
// Change the TTL to something more appropriate
//
optval = 8;
if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL,
    (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
        WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}
// Disable loopback if needed
//
if (bLoopBack)
{
    optval = 0;
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
            WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return -1;
```

```
    }
}
// Join the multicast group.  Note that sockM is not used
// to send or receive data. It is used when you want to
// leave the multicast group. You simply call closesocket()
// on it.
//
if ((sockM = WSAJoinLeaf(sock, (SOCKADDR *)&remote,
    sizeof(remote), NULL, NULL, NULL, NULL,
    JL_BOTH)) == INVALID_SOCKET)
{
    printf("WSAJoinLeaf() failed: %d\n", WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}

if (!bSender)           // Receiver
{
    // Receive data
    //
    for(i = 0; i < dwCount; i++)
    {
            if ((ret = recvfrom(sock, recvbuf, BUFSIZE, 0,
                (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
            {
                printf("recvfrom failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
            }
            recvbuf[ret] = 0;
            printf("RECV: '%s' from <%s>\n", recvbuf,
                inet_ntoa(from.sin_addr));
        }
    }
    else                    // Sender
    {
        // Send data
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "server 1: This is a test: %d", i);
            if (sendto(sock, (char *)sendbuf, strlen(sendbuf), 0,
                (struct sockaddr *)&remote,
                sizeof(remote)) == SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
```

```
            }
            Sleep(500);
        }
    }
    // Leave the multicast group by closing sock.
    // For nonrooted control and data plane schemes, WSAJoinLeaf
    // returns the same socket handle that you pass into it.
    //
    closesocket(sock);

    WSACleanup();
    return -1;
}
```

2. Winsock 2 ATM多播示例

在程序清单11-3中，我们列出了Mcastatm.c的源码清单，这是一个简单的ATM多播示范程序，用于阐释一次“由根发起的”加入。本列表并未包括文件 Support.c。在那个文件内，包括了在多播示例中用到的一些例程，但却并非多播通信所独有的，比如 GetATMAddress等等——用于返回本地接口的ATM地址。

看看程序清单11-3展示的代码，便会发现对于多播根来说，最重要的部分便是Server函数，它通过一个循环，为命令行指定的每个客户机都调用一次 WSAJoinLeaf。针对加入的每个客户机，服务器都维持了一个套接字数组。然而，在 WSASend调用中，使用的却是主套接字。假如ATM协议提供了对它的支持，而且感兴趣的是接收“秘密交谈”信息，便可选择在叶套接字上进行监听（比如自 WSAJoinLeaf返回的套接字）。换句话说，假定客户机在连好的套接字上发出数据，服务器会在自 WSAJoinLeaf返回的、相应的句柄上收到。其他客户机则收不到这种数据。

而对客户机来说，请注意观察 Client函数。可以发现，唯一的要求便是将客户机同一个本地接口绑定起来，并等候服务器通过 accept或WSAAccept调用发来的邀请。一旦邀请抵达，新建的套接字便可用来接收由“根”发出的数据。

程序清单11-3 ATM多播示例

```
// Module: Mcastatm.c

#include "Support.h"

#include <stdio.h>
#include <stdlib.h>


#define BUFSIZE                 1024
#define MAX_ATM_LEAF               4

#define ATM_PORT_OFFSET         ((ATM_ADDR_SIZE * 2) - 2)
#define MAX_ATM_STR_LEN         (ATM_ADDR_SIZE * 2)

DWORD   dwAddrCount = 0,
        dwDataCount = 20;
BOOL    bServer = FALSE,
        bLocalAddress = FALSE;
```

```
char   szLeafAddresses[MAX_ATM_LEAF][MAX_ATM_STR_LEN + 1],
       szLocalAddress[MAX_ATM_STR_LEN + 1],
       szPort[3];
SOCKET sLeafSock[MAX_ATM_LEAF];
// Module: usage
//
// Description:
//    Print usage information
//
void usage(char *progname)
{
    printf("usage: %s [-s]\n", progname);
    printf("     -s       Act as root\n");
    printf("     -l:str   Leaf address to invite (38 chars)\n");
    printf("              May be specified multiple times\n");
    printf("     -i:str   Local interface to bind to (38 chars)\n");
    printf("     -p:xx    Port number (2 hex chars)\n");
    printf("     -n:int   Number of packets to send\n");
    ExitProcess(1);
}

// Module: ValidateArgs
//
// Description:
//    Parse command line arguments
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    memset(szLeafAddresses, 0,
        MAX_ATM_LEAF * (MAX_ATM_STR_LEN + 1));
    memset(szPort, 0, sizeof(szPort));

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':     // Server
                    bServer = TRUE;
                    break;
                case 'l':     // Leaf address
                    if (strlen(argv[i]) > 3)
                    {
                        strncpy(szLeafAddresses[dwAddrCount++],
                            &argv[i][3], MAX_ATM_STR_LEN - 2);
                    }
                    break;
                case 'i':     // Local interface
                    if (strlen(argv[i]) > 3)
                    {
                        strncpy(szLocalAddress, &argv[i][3],
                            MAX_ATM_STR_LEN - 2);
```

```
                        bLocalAddress = TRUE;
                    }
                    break;
                case 'p':     // Port address to use
                    if (strlen(argv[i]) > 3)
                        strncpy(szPort, &argv[i][3], 2);
                    break;
                case 'n':     // Number of packets to send
                    if (strlen(argv[i]) > 3)
                        dwDataCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: Server
//
// Description:
//    Bind to the local interface, and then invite each leaf
//    address that was specified on the command line.
//    Once each connection is made, send some data.
//
void Server(SOCKET s, WSAPROTOCOL_INFO *lpSocketProtocol)
{
    // Server routine
    //
    SOCKADDR_ATM  atmleaf, atmroot;
    WSABUF        wsasend;
    char          sendbuf[BUFSIZE],
                  szAddr[BUFSIZE];
    DWORD         dwBytesSent,
                  dwAddrLen = BUFSIZE,
                  dwNumInterfaces,
                  i;
    int           ret;
// If no specified local interface is given, pick the
// first one
//
memset(&atmroot, 0, sizeof(SOCKADDR_ATM));
if (!bLocalAddress)
{
    dwNumInterfaces = GetNumATMInterfaces(s);
    GetATMAddress(s, 0, &atmroot.satm_number);
}
else
    AtoH(&atmroot.satm_number.Addr[0], szLocalAddress,
        ATM_ADDR_SIZE - 1);
//
// Set the port number in the address structure
```

```
//
AtoH(&atmroot.satm_number.Addr[ATM_ADDR_SIZE-1], szPort, 1);
//
// Fill in the rest of the SOCKADDR_ATM structure
//
atmroot.satm_family                 = AF_ATM;
atmroot.satm_number.AddressType     = ATM_NSAP;
atmroot.satm_number.NumofDigits     = ATM_ADDR_SIZE;
atmroot.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
atmroot.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
atmroot.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
//
// Print out what we're binding to, and bind
//
if (WSAAddressToString((LPSOCKADDR)&atmroot,
    sizeof(atmroot), lpSocketProtocol, szAddr,
    &dwAddrLen))
{
    printf("WSAAddressToString failed: %d\n",
        WSAGetLastError());
}
printf("Binding to: <%s>\n", szAddr);

if (bind(s, (SOCKADDR *)&atmroot,
    sizeof(SOCKADDR_ATM)) == SOCKET_ERROR)
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;
}
// Invite each leaf
//
for(i = 0; i < dwAddrCount; i++)
{
    // Fill in the SOCKADDR_ATM structure for each leaf
    //
    memset(&atmleaf, 0, sizeof(SOCKADDR_ATM));
    AtoH(&atmleaf.satm_number.Addr[0], szLeafAddresses[i],
        ATM_ADDR_SIZE - 1);
    AtoH(&atmleaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
        szPort, 1);

    atmleaf.satm_family                 = AF_ATM;
    atmleaf.satm_number.AddressType     = ATM_NSAP;
    atmleaf.satm_number.NumofDigits     = ATM_ADDR_SIZE;
    atmleaf.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
    atmleaf.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
    atmleaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
    //
    // Print out client's address, and then invite it
    //
    if (WSAAddressToString((LPSOCKADDR)&atmleaf,
        sizeof(atmleaf), lpSocketProtocol, szAddr,
        &dwAddrLen))
    {
        printf("WSAAddressToString failed: %d\n",
```

```
                WSAGetLastError());
        }
        printf("[%02d] Inviting: <%s>\n", i, szAddr);

        if ((sLeafSock[i] = WSAJoinLeaf(s,
            (SOCKADDR *)&atmleaf, sizeof(SOCKADDR_ATM), NULL,
            NULL, NULL, NULL, JL_SENDER_ONLY))
            == INVALID_SOCKET)
        {
            printf("WSAJoinLeaf() failed: %d\n",
                WSAGetLastError());
            WSACleanup();
            return;
        }
    }
    // Note that the ATM protocol is a bit different from TCP.
    // When the WSAJoinLeaf (or connect) call completes, the
    // peer has not necessarily accepted the connection yet,
    // so immediately sending data will result in an error;
    // therefore, we wait for a short time.
    //
    printf("Press a key to start sending.");
    getchar();
    printf("\n");

        //
        // Now send some data to the group address, which will
        // be replicated to all clients
        //
        wsasend.buf = sendbuf;
        wsasend.len = 128;
        for(i = 0; i < dwDataCount; i++)
        {
            memset(sendbuf, 'a' + (i % 26), 128);
            ret = WSASend(s, &wsasend, 1, &dwBytesSent, 0, NULL,
                NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: %d\n", WSAGetLastError());
                break;
            }
            printf("[%02d] Wrote: %d bytes\n", i, dwBytesSent);
            Sleep(500);
        }

        for(i = 0; i < dwAddrCount; i++)
            closesocket(sLeafSock[i]);
        return;
    }

    //
    // Function: Client
    //
    // Description:
    //    First the client binds to the local interface (either one
    //    specified on the command line or the first local ATM address).
```

```
//    Next it waits on an accept call for the root invitation. It
//    then waits to receive data.
//
void Client(SOCKET s, WSAPROTOCOL_INFO *lpSocketProtocol)
{
    SOCKET       sl;
    SOCKADDR_ATM atm_leaf,
                 atm_root;
    DWORD        dwNumInterfaces,
                 dwBytesRead,
                 dwAddrLen=BUFSIZE,
                 dwFlags,
                 i;
    WSABUF       wsarecv;
    char         recvbuf[BUFSIZE],
                 szAddr[BUFSIZE];
int          iLen = sizeof(SOCKADDR_ATM),
             ret;

// Set up the local interface
//
memset(&atm_leaf, 0, sizeof(SOCKADDR_ATM));
if (!bLocalAddress)
{
    dwNumInterfaces = GetNumATMInterfaces(s);
    GetATMAddress(s, 0, &atm_leaf.satm_number);
}
else
    AtoH(&atm_leaf.satm_number.Addr[0], szLocalAddress,
        ATM_ADDR_SIZE-1);
AtoH(&atm_leaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
    szPort, 1);
//
// Fill in the SOCKADDR_ATM structure
//
atm_leaf.satm_family                 = AF_ATM;
atm_leaf.satm_number.AddressType     = ATM_NSAP;
atm_leaf.satm_number.NumofDigits     = ATM_ADDR_SIZE;
atm_leaf.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
atm_leaf.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
atm_leaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
//
// Print the address we're binding to, and bind
//
if (WSAAddressToString((LPSOCKADDR)&atm_leaf,
    sizeof(atm_leaf), lpSocketProtocol, szAddr,
    &dwAddrLen))
{
    printf("WSAAddressToString failed: %d\n",
        WSAGetLastError());
}
printf("Binding to: <%s>\n", szAddr);

if (bind(s, (SOCKADDR *)&atm_leaf, sizeof(SOCKADDR_ATM))
        == SOCKET_ERROR)
```

```
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;
}
listen(s, 1);
//
// Wait for the invitation
//
    memset(&atm_root, 0, sizeof(SOCKADDR_ATM));
    if ((sl = WSAAccept(s, (SOCKADDR *)&atm_root, &iLen, NULL,
        0)) == INVALID_SOCKET)
    {
        printf("WSAAccept() failed: %d\n", WSAGetLastError());
        return;
    }
    printf("Received a connection!\n");

    // Receive some data
    //
    wsarecv.buf = recvbuf;
    for(i = 0; i < dwDataCount; i++)
    {
        dwFlags = 0;
        wsarecv.len = BUFSIZE;
        ret = WSARecv(sl, &wsarecv, 1, &dwBytesRead, &dwFlags,
            NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSARecv() failed: %d\n", WSAGetLastError());
            break;
        }
        if (dwBytesRead == 0)
            break;
        recvbuf[dwBytesRead] = 0;
        printf("[%02d] READ %d bytes: '%s'\n", i, dwBytesRead,
            recvbuf);
    }
    closesocket(sl);
    return;
}

//
// Function: main
//
// Description:
//    This function loads Winsock library, parses command line
//    arguments, creates the appropriate socket (with the right
//    root or leaf flags), and starts the client or the server
//    functions, depending on the specified flags
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    SOCKET              s;
    WSAPROTOCOL_INFO    lpSocketProtocol;
```

```
    DWORD               dwFlags;
ValidateArgs(argc, argv);
//
// Load the Winsock library
//
if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
{
    printf("WSAStartup failed\n");
    return -1;
}
// Find an ATM-capable protocol
//
if (FindProtocol(&lpSocketProtocol) == FALSE)
{
    printf("Unable to find ATM protocol entry!\n");
    return -1;
}
// Create the socket using the appropriate root or leaf flags
//
if (bServer)
    dwFlags = WSA_FLAG_OVERLAPPED
              | WSA_FLAG_MULTIPOINT_C_ROOT
              | WSA_FLAG_MULTIPOINT_D_ROOT;
else
    dwFlags = WSA_FLAG_OVERLAPPED
              | WSA_FLAG_MULTIPOINT_C_LEAF
              | WSA_FLAG_MULTIPOINT_D_LEAF;

if ((s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
        FROM_PROTOCOL_INFO, &lpSocketProtocol, 0,
        dwFlags)) == INVALID_SOCKET)
{
    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
// Start the correct driver, depending on which flags were
// supplied on the command line
//
if (bServer)
{
    Server(s, &lpSocketProtocol);
}
else
{
    Client(s, &lpSocketProtocol);
}
closesocket(s);
    WSACleanup();
    return 0;
}
```

11.4.3 常用的Winsock选项

无论 Winsock 1还是 Winsock 2，IP多播都有三个专用的套接字选项，分别是：

IP_MULTICAST_TTL，IP_MULTICAST_IF和IP_MULTICAST_LOOP。这些选项以往在Winsock 1中常常用于多播组的加入和离开。但对新版的 Winsock 2函数来说，它们仍然非常有用。当然，所有这三个套接字选项均是IP多播所独有的。

1. IP_MULTICAST_TTL

该选项用于设置多播数据的TTL（存在时间）值。在默认情况下， TTL值为1。也就是说，多播数据遇到第一个路由器，便会被它“无情”地丢弃，不允许传出本地网络之外，即只有同一个网络内的多播成员才会收到数据。假如增大 TTL的值，多播数据就可经历多个路由器传到其他网络。TTL的值是多少，最多便能经过多少个路由器。假如路由器收到一个数据包，并判断出自己应将这个包转发给相邻的网络，那么 TTL值随即会被减1。若减1之后，发现TTL的新值变成了0，路由器便会将这个包即时地丢弃，因为它的“存在时间”已经到了。若多播数据报的目标地址在224.0.0.0到224.0.0.255之间，那么多播路由器不会对其进行转发，无论它们的TTL有多大。这个地址范围是为路由协议以及其他低级拓扑查找和维护协议（比如网关查找和组成员关系报告）所保留的，不可擅用。

若调用setsockopt，level参数便是IPPROTO_IP，optname参数是IP_MULTICAST_TTL，而optval参数是一个整数，用于指定新的 TTL值。下述代码片断阐述了具体如何设置 TTL的值：

```
int    optval;

optval = 8;
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&optval,
        sizeof(int)) == SOCKET_ERROR)
{
    // Error
}
```

除了这个套接字选项之外，对于 WSAIoctl或ioctlsocket这两个函数来说，还可以使用SIO_MULTICAST_SCOPE选项，它可对套接字采取同样的操作。

注意，在ATM多播环境中， TTL参数是不必要的。因为数据在 ATM上的传送只是一个单向过程，所有接收者都是已知的。由于控制层面是“有根”的，所以 c_root节点必须明确地邀请每个叶节点加入。这就意味着，我们不必对数据传输的范围进行限制，数据不会“流窜”到没有任何多播成员的网络上。

2. IP_MULTICAST_IF

这个选项用于设置自哪个IP接口发出多播数据。正常情况下（即非多播环境），只能参照路由表来决定一个数据报自哪个接口发出。此时，依据一个特定数据报本身以及它的目的地，系统可判断出哪个接口最适合用来发出这个数据报。然而，由于多播地址可由任何人使用，所以仅仅依靠路由表的自动判断是不够的。程序员必须多少施加一些自己的影响。当然，只有打算发出多播数据的那台机器已通过网卡建立了与多个网络的连接，才需要考虑到这方面的问题。对于这个选项， optval参数指定的是一个本地接口的地址，多播数据以后便会从这个接口发出。请参考下述代码：

```
DWORD    dwInterface;

dwInterface = inet_addr("129.121.32.19");
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&dwInterface,
```

```
    sizeof(DWORD)) == SOCKET_ERROR)
{
    // Error
}
```

可以看出，在上例中，我们将本地接口设为 129.121.32.19。对通过套接字s发出的任何多播数据而言，都会通过分配了那个IP地址的网络接口送出。

再次提醒大家， ATM并不需要单独用一个套接字选项来设置接口。调用 WSAJoinLeaf之前，c_root可同一个特定的接口明确绑定到一起。类似地，客户机必须同一个特定的 ATM接口绑定到一起，以便等候随accept或WSAAccept发出的加入邀请。

3. IP_MULTICAST_LOOP

最后一个套接字选项决定了应用程序是否接收自己的多播数据。如应用程序加入了一个多播组，并向那个组发送数据，应用程序便会接收到那些数据。数据送出的时候，假如有一个recvfrom调用正处于“ 待决 ”状态，调用便会返回那个数据的一个副本。要注意的是，若只是想让应用程序将数据发给一个多播组，那么它并不一定非要加入多播组。只有在希望接收发给那个组的数据时，才需要加入那个组。设计 IP_MUTLICAST_LOOP套接字选项的目的便是禁止将数据返还给本地接口。使用这个选项，我们可将一个整数值作为 optval参数传递，它其实仅仅是一个“ 布尔 ”值，用于指出到底允许还是禁止多播返还。请来看看下述示范代码：

```
int    optval;

optval = 0;     // Disable loopback
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&optval,
    sizeof(int)) == SOCKET_ERROR)
{
    // Error
}
```

另外，与 WSAIoctl或ioctlsocket配套使用的一个 ioctl命令（SIO_MULTIPOINT_LOOPBACK）亦可做到同样的事情。但不幸的是，这个套接字选项并未在 Windows 95、Windows 98或Windows NT 4中实现。而且在默认情况下，多播数据的“ 返还 ”是允许的。假如用这个套接字选项禁止了数据返还，便会收到一个 WSAENOPROTOOPT错误。

按照定义， ATM根节点（亦即允许向多播组发送数据的唯一成员）不会接收到自己发出的数据，因为根套接字并非一个叶套接字。在 ATM多播环境中，只有叶套接字才能接收数据。当初用来创建c_root的过程亦可用来创建一个独立的 c_leaf节点，并由c_root邀请加入。然而，这却并非一个真正意义上的数据“ 返还 ”（Loopback）。

11.4.4 拨号网络多播的一处限制

若试图在一个“ 远程访问服务 ”（RAS）接口或拨号接口上进行多播数据的收发，一处限制务必小心在意。这一限制实际是由拨号连接的那个服务器造成的。目前，大多数以Windows为基础的拨号服务器运行的都是 Windows NT 4操作系统，它们通常没有提供一个IGMP代理。这便意味着，我们发出的任何组成员加入请求都无法离开 Windows NT 4服务器。这样一来，我们的应用程序便不能加入任何多播组，所以也无法进行多播数据的收发。与Windows 2000配套提供的RAS服务器则包含了一个IGMP代理，只是在默认情况下并未开启。

但是，只要启用了这个代理，拨号客户机除了能加入多播组之外，还能进行多播数据的收发。

11.5 小结

若应用程序要求同时与多个“端点”通信，同时又不想招致广播通信为网络带来的重大开销，那么多播便是很好的一种选择。本章对多播技术进行了定义，并向大家展示了各种不同的多播模型。随后，我们探讨了 IP多播和ATM点到多点通信如何应用于这些模型。最后，我们解释了Winsock API具体如何提供对多播通信的支持。同时介绍了 Winsock 1和Winsock 2的方法。前者使用的是套接字选项，而后者使用的是更为合理的 WSAJoinLeaf函数。