第12章　常规服务质量

当今，随着多媒体技术的普遍应用，同时由于 Internet的广泛流行，许多网络都越来越不堪重负。基本的原因便是各种应用（特别是多媒体应用）对带宽的要求越来越大，以至于出现了“ 供不应求 ”的局面。在所谓的共享媒体网络上（如以太网），这个问题尤其突出，因为所有通信数据都有着相同的地位。即使一个非常简单的应用，也可能造成数据在网络上泛滥成灾，造成整个网络的瘫痪。为此，人们提出了“ 服务质量 ”（Quality of Service，简称QoS）的概念。QoS实际是一系列组件，允许对网上的数据进行不同处理，并可为其分配不同的优先级。若一个网络具备 QoS功能，便可根据实际需要，对其进行配置，以便为程序员提供下述能力：

- 禁止非适应性协议（如UDP）滥用网络资源。
- 针对“ 最大努力 ”通信，以及高优先级或低优先级的通信，对资源进行明确划分。
- 为冠名用户保留资源。
- 为用户分派资源访问的优先级。

常规服务质量（Generic Quality of Service，GQOS）是微软对QoS的一种实施方案。目前，微软已提供了具有QoS能力的TCP/IP及UDP/IP提供者，可在Windows 98及Windows 2000上使用。要注意的是，ATM本身便已提供了对QoS的支持。

本章将向大家介绍QoS的原理，以及它在Win32平台上的实现方式。首先要讨论的是，为了对不同的网络传输（网络通信）进行区分对待，哪些组件是必要的。随后，我们将探讨如何利用Winsock接口来写程序，使其能够利用这些组件，为一些对时间及网络带宽要求颇为严格的应用提供服务。本章的大部分内容都围绕 IP网络上的QoS展开。在本章末，我们将讨论QoS在ATM网络上的情况，它与IP网络上的QoS稍有区别。

注意　贯穿全章，我们都会把“ 服务质量 ”简称为QoS。此外，读者完全可以假定我们讨论的都是微软实施的这一套QoS。

12.1　背景知识

QoS需要三个组件才能正常发挥作用：

- 网络上的设备：比如路由器和网关等等，它们可注意到这种服务上的区别。
- 本地工作站：可为自己引入的网络传输分派相应的优先级。
- 策略组件：谁能使用可用的带宽，以及允许多少人使用。

然而，在我们深入讨论这些组件之前，首先还是来看看“ 资源预约协议 ”（Resource Reservation Protocol，RSVP）的问题。这是在QoS发送者及QoS接收者之间使用的一种传输协议。RSVP在QoS中扮演了一个非常重要的角色，而且是 QoS之三个主要组件的大集成者。

12.1.1　资源预约协议

可将RSVP想象成一种“ 粘合剂 ”，它负责将网络、应用以及策略组件粘合成一个整体。

通过一个网络，RSVP携带着资源预约请求信息，而这个网络可能由不同的媒体构成。沿着数据路径，RSVP可将一名用户的QoS请求传播到配置了RSVP功能的所有网络设备上，允许将资源从所有RSVP设备中“预约”或“保留”出来。这样一来，网络节点便可根据网络的现状，判断出它是否能达到要求的服务级别（或服务品质）。

RSVP协议在预约网络资源时，需要贯穿全网，建立端到点的“数据流”。这里的“数据流”实际就是一个网络路径，除了同一个或多个发送者联系在一起之外，还要同一个特定的QoS级别联系在一起。作为发送方主机，假如希望自己发出的数据分派到一个特定级别的QoS，便需向目标接收者（可能不止一个）发送一条PATH（路径）消息。在这条PATH消息中，主要包含了对带宽的要求。沿着这条路径，相关的参数便会传播至目标接收者。

作为一个接收方主机，假如对这个数据有兴趣，便可为数据流保留相应的资源（以及自发送者那里来的完整路径）。因此，它需要发回一条RESV（就是“预约”的意思）消息，对发送者作出回应。此后，位于中途的RSVP设备便必须判断自己是否能提供要求的带宽，并核查发出资源请求的用户是否真的有权来做这样的事情。假如拿出请求的带宽毫无问题，而且根据用户的安全策略设置，表明用户有权发出这样的请求，那么位于中途的每个RSVP设备都会腾出需要的资源，并将RESV消息传回当初的发送者。

发送者收到RESV消息之后，QoS数据便可开始“流动”了。在这个“流”内的每个端点都会定期发送PATH和RESV消息，重申资源预约，以及在可用带宽级别发生更改之后，提供相关的网络信息。此外，通过PATH和RESV消息的定时刷新，RSVP协议便能“永葆青春”；或者说，一直保持“动态”。假如出现了一条更好（比如更快）的传输路径，刷新过的消息便能发现那条新路由。本章后面讨论Winsock提供的QoS机制时，还会将重点放回RSVP协议，并讲解如何用Winsock API调用提供对它的支持。

对于会话设置和RSVP来说，有个问题切不可掉以轻心——资源的预约永远都是“单向”的！即便应用程序同时为数据发送及接收请求带宽，预约仍然是单向进行的。此时会为发送请求初始化一个会话，并为接收请求初始化另一个会话。在本章后面，还会详细论述RSVP会话的初始化要求。

12.1.2 网络组件

为使“端到端”的QoS能正常运作，两个端点之间的网络设备必须能对数据通信的优先级加以区分。只有这样，它们才能在满足QoS要求的前提下，对应用程序要接收的数据进行正确的路由（选择）。除此以外，在应用程序发出带宽请求之后，这些网络设备必须能判断出网络中是否存在足够的带宽。为满足这一系列要求，我们创建了下面这些必不可少的组件：

- 802.1p：在子网中区分数据包优先次序的一种标准，它在包的访问媒体控制（MAC）头中另行设置了3个位（二进制位）。
- IP优先级：用于为IP包建立优先级的一种方法。
- 第2层传信：将RSVP对象映射成正宗WAN QoS组件（网络的位于OSI第2层）的一种机制。
- 子网带宽管理器（SBM）：用于对共享媒体网络带宽进行管理的组件。
- 资源预约协议（RSVP）：用于携带（负载）QoS请求以及相关信息的一种协议。沿着发送者与接收者之间的数据路径，这些信息会传递给配备了QoS能力的网络设备。注

意发送者只能有一个，但接收者可以有多个。

1. 802.1p

这个标准用于确保QoS要求的质量等级，同时避免对存在于网络 HUB和交换机的所有数据包进行同等对待。HUB和交换机均位于OSI参考模型的第2层，所以只有在每个包开头的媒体访问控制头内进行设置，才有意义。

802.1p是为网络数据包分配优先等级的一种标准，它需要在 MAC头内设置一个总长3位的值（优先位）。在非802.1p网络中，假如一个子网变得堵塞，连接不通，那么交换机和路由器便不能跟上数据的传输，同时会造成通信的延迟。而另一方面，在 802.1p网络中，根据优先位的设定，交换机和路由器可对进入的数据进行优先级的分派，优先对待那些优先等级较高的包。

假如要为QoS实现802.1p，那么硬件必须具备特殊的能力，可识别这个 3位字段。网卡（NIC）、网络驱动程序以及网络交换机都必须具备 802.1p功能。

2. IP优先级

IP优先级（IP Precedence）是从一个比802.1p高的层次，对优先值进行指定的方法。利用这个方法，通过OSI模型第3层设备（比如路由器）的数据包可以分别拥有不同的优先等级设置。为实现IP优先级，我们需要在IP头内设置“服务类型”（TOS）字段，以建立不同的优先等级。换言之，等级较高的通信会从路由器那里获得质量更好的服务。

和802.1p一样，为使IP优先级正常工作，网络上的所有第 3层设备都必须能够理解 IP优先位的设定，并以它为准，对传输的数据进行分别对待。

3. 第2层传信

若数据通过一个广域网（WAN）传输，那么第2层传信是颇为必要的。通常，一个 WAN同时链接了数个网络，那些网络使用的硬件设备则五花八门，不一而足。总之，网络通信的环境非常复杂。因此，在数据传输的时候， WAN应该能够同时处理第1、第2以及第3层信息。为保障端到端的QoS（服务质量），WAN链接必须理解QoS通信的优先等级。为达到这个目的，QoS提供了一种方法，可将 RSVP和其他QoS参数映射成WAN本身能够理解的第 2层传信方法——WAN技术利用这些方法为实现自己的 QoS。

4. 子网带宽管理器

子网带宽管理器（Subnet Bandwidth Manager，SBM）负责对一个指定共享媒体网络（如以太网）上的资源进行管理。 SBM也要面向QoS应用，施加以策略为基础的访问权限控制。在共享媒体网络上， SBM是至关重要的一环，因为假若端点为某个应用请求 QoS，那么每个网络设备都要根据自己专用资源的分配情况，要么许可、要么拒绝这一请求。那么， SBM是如何来解决这个问题的呢？它的办法是与“许可控制服务”（Admission Control Service，ACS）紧密联系在一起，该服务属于策略组件的一部分。 SBM必须通过检查，确保请求带宽的那个应用（或对应的用户）拥有足够的权限，有权作出这样的要求。要注意的是，一个网络的SBM完全可能是一个特殊的主机，正在运行 Windows 2000 Server网络操作系统。

12.1.3 应用组件

现在，大家已经知道了为提供对 QoS的支持，对网络提出了哪些要求。接下来，我们必须考虑本地系统如何根据应用程序请求的 QoS等级，对数据的优先级进行分派。要想使本地系

统支持QoS，下述组件是必需的：

- GQOS服务提供者：负责调用其他QoS组件的一个服务提供者。
- 通信控制模块：该模块负责控制离开计算机的通信数据。这个模块包括常规包分类器、包调度器以及包封装器。
- 资源预约协议（RSVP）。
- GQOS API函数：提供GQOS的编程接口，如Winsock。
- 通信控制（TC）API函数：为通信控制组件提供编程接口，对本地主机上的数据传输进行管理。

1. GQOS服务提供者

GQOS服务提供者是一种GQOS组件，可调用几乎所有最终的QoS机制。GQOS服务提供者可初始化通信控制功能（如果可以的话），同时为所有GQOS功能实施、维护以及控制RSVP传信。

要想在自己的主机上找到具有QoS能力服务提供者，可用WSAEnumProtocols函数查询提供者目录。用来核实某个提供者是否支持QoS的标志位于WSAPROTOCOL_INFO结构之内，该结构正是自WSAEnumProtocols调用返回的。我们在此感兴趣的字段是dwServiceFlags1，看看它的值是否为XP1_QOS_SUPPORTED就行了。如果是，便表明该提供者支持QoS。要想了解WSAEnumProtocols的详情，可参考第5章。

2. 通信控制模块

通信控制（Traffic Control，TC）在QoS中扮演了一个至关重要的角色。在TC内，在启用TC的那个网络节点之内和之外的所有数据包都需要定义优先级。正是由于在这里对不同的数据包分配了不同的优先级，所以随着它们流经系统和网络，最终传遍整个网络，所以会直接对QoS的特征产生影响。TC模块通过三个模块来实现：常规包分类器、包调度器以及包封装器。

(1) 常规包分类器

常规包分类器（Generic Packet Classifier，GPC）的职责是对网络组件内的数据包进行分类，并分配它的优先等级。GPC定义优先级的依据是CPU时间或在网上进行传输这样的活动。

为完成优先级的定义，GPC需要创建索引表，对网络堆栈内的服务进行分类。这是为网络通信划分优先等级的第一个步骤。

(2) 包调度器

数据包调度器（Packet Scheduler）控制着数据的传输方式，这是QoS的一项关键功能。数据包调试器实际是一个通信控制模块，规定一个应用程序或者一个数据流允许多少数据。换言之，它需要为一个特定的数据流规定相应的QoS参数集合。

包调度器采用由GPC提供的优先级分配方案，并为不同的优先级提供不同等级的服务。例如，由GPC分类为“高优先级”的数据会在包调度器内得到优先对待。

(3) 包封装器

包封装器（Packet Shaper）的作用是规划数据从数据流到网络的传输。大多数应用程序都是以“爆发”形式来读写数据的；然而，许多QoS应用都需要以固定的速度，来进行数据的发送。因此，包封装器需要在一个时间范围内，对数据的传输事先作好安排，使网络使用尽可能地平稳，营造出负担比较平衡的一个网络。

3. 通信控制API

通信控制API是本地主机上用来规划网络通信的那些组件的接口。其中包括用于处理常规包分类器、包调度器以及包封装器的方法。有些通信控制函数是在对 Winsock GQOS的调用过程中，“ 顺便 ”调用的。此时需要由 GQOS服务提供者为那些函数提供服务。然而，假如一个应用程序需要直接操作通信控制（ TC）组件，便可考虑直接使用这些通信控件 API函数。然而，对这些函数的深入探讨已超出了本书的范围（请参考 Platform SDK，获得进一步的信息；本书配套光盘提供)。至于Winsock GQOS API的情况，本章稍后便会详加论述。

12.1.4 策略组件

“ 策略 ”（Policy）是GQOS的最后一个组件，它负责将资源分配给具有 QoS能力的应用程序。策略组件是系统管理员最感兴趣的东西，他们需要根据用户，或根据应用程序请求的带宽类别，对资源的分配加以控制。策略组件包括：

- 许可控制服务（ACS）：一种Windows 2000服务器服务，用于拦截RSVP PATH和RESV消息，对QoS客户机的访问加以控制，将访问划分为通过 QoS提供的各个保证等级。
- 本地策略模块（LPM）：面向SBM（子网带宽管理器），以通过ACS配置的策略为准，提供资源访问决策服务。
- 策略元素（PE）：驻留于客户机，提供身份证明信息，以完成预约请求。

1. 许可控制服务

许可控制服务（Admission Control Service，ACS）面向具有QoS功能的应用程序，对网络带宽的使用进行规划。这是通过 RSVP协议来完成的。 ACS会同时拦截PATH和RESV消息，校验发出请求的应用程序是否拥有足够的权限。拦截到一条 RSVP消息后，便会传递给本地策略模块（LPM），由后者进行实际的身份验证。

ACS驻留于安装了Windows 2000操作系统的机器上，可由系统管理员加以控制，后者可分别针对用户、应用程序或者用户组，对他们享受的资源加以限制。

2. 本地策略模块

本地策略模块（Local Policy Module，LPM）与前述的ACS存在着紧密的联系。因为先要由ACS拦截RSVP消息，插入用户信息，再传递给 LPM。之后， LPM会在“ 活动目录 ”（Active Directory）里找到用户，以验证其策略信息。假如申请的网络资源可用（由SBM决定），而且身份资料正确无误（拥有足够的权限），那么由ACS拦截到的RSVP消息便会传给“ 下一跳 ”。当然，假如用户不够资格申请一个特定的 QoS等级，便会生成一个错误，并在 RSVP消息里返回。

注意 为保证策略的成功检查，用户必须是Windows 2000域的一份子。

3. 策略元素

在这个组件中，实际包含了 LPM请求的策略信息。本书不会讨论这些数据结构的详情，因为它们主要与网络资源的管理有关，而那并非本书的主题。

12.2 QoS和Winsock

在前一节内，我们已探讨了成功构建一个端到端 QoS网络所需的各种组件。接下来，让我们把重点转向Winsock 2。利用这套API，我们可从一个应用程序中实现对 QoS的访问。首先

讨论的是大多数 Winsock调用都要用到的顶级 QoS结构。接下来，讨论那些能在套接字调用QoS的Winsock函数，同时讲解在套接字上启用了 QoS之后，如何再将QoS中止。我们要讨论的最后一个问题是“ 提供者 ”特有的一些对象，可用来影响 QoS服务提供者的一些行为，或影响自它们那里返回的信息。

表面看，先讨论主要的QoS结构，再讨论QoS函数，最后再回过头来讨论提供者特有的结构，似乎有些“ 杂乱无章 ”。但是，我们之所以这样做，完全是为了让大家首先全面理解那些主要结构如何与Winsock API调用打交道，再来接触由特定提供者决定的一些选项的细节。

12.2.1 QoS结构

对QoS编程来说，其中心结构便是“ QoS ”结构，该结构包括：

- 一个FLOWSPEC结构，用于描述应用程序用以发送数据的 QoS等级。
- 一个FLOWSPEC结构，用于描述应用程序用以接收数据的 QoS等级。
- 一个提供者特有的缓冲区，用来指定各个提供者具有的某些 QoS特征（后文详述）。

1. QoS

QoS结构同时为数据的发送及接收指定相应的 QoS参数。它的定义如下：

```
typedef struct _QualityOfService
{
    FLOWSPEC    SendingFlowspec;
    FLOWSPEC    ReceivingFlowspec;
    WSABUF      ProviderSpecific;
} QOS, FAR * LPQOS;
```

其中，FLOWSPEC结构定义了通信的特征，以及每个传输方向的具体要求。而 Provider Specific字段用于返回信息，并可更改 QoS的行为。这些取决于不同提供者的选项将在本章后面部分讲述。

2. FLOWSPEC

FLOWSPEC是一种基本结构，用于对单个流进行描述。要记住的是，一个“ 数据流 ”描述的是朝一个方向传输的数据。该结构定义如下：

```
typedef struct _flowspec
{
    ULONG          TokenRate;
    ULONG          TokenBucketSize;
    ULONG          PeakBandwidth;
    ULONG          Latency;
    ULONG          DelayVariation;
    SERVICETYPE    ServiceType;
    ULONG          MaxSduSize;
    ULONG          MinimumPolicedSize;
} FLOWSPEC, *PFLOWSPEC, FAR *LPFLOWSPEC;
```

接下来，让我们看看每个FLOWSPEC结构字段的含义。

(1) TokenRate

TokenRate字段定义了数据的传输速度，单位是“ 字节 /秒 ”。Token是“ 令牌 ”的意思。作为一个应用程序，它可以按此速度传输数据，但假如由于某种原因，它需要用一个较低的速度传输，便可积累下额外的令牌，以便更多的数据能在以后传输出去。然而，一个应用程序

可累积的令牌数量要受 PeakBandwidth的制约。同时，令牌本身的尺寸也要受到TokenBucketSize字段的限制。通过对令牌总量加以限制，便可防止不活动的数据流积下大量令牌。因为这样极易造成可用带宽的极大浪费。尽管数据流可在一定时间内，积累传输所需的“信用点”（令牌），但最多只能积累到由TokenBucketSize的大小，而且由于它们的“峰值传输”受到PeakBandwidth的限制，所以传输控制以及网络设备资源的完整性得到了有效保证。之所以说传输控制得到了保证，是由于数据流不能一次发送任意多的数据。而网络设备资源完整性得到了保证，是由于这些设备不会受到突发性的“峰值”传输量的冲击。

由于采取了这些限制措施，应用程序只有在累积了足够的“信用点”之后，才能开始传输数据。假如没有达到对信用点数量的要求，那么对应用程序而言，要么等积累到足够多的信用点再发送数据，要么完全放弃数据的传输。通信控制（TC）模块负责决定对于等待许久都没有发送出去的数据，该如何进行处理。因此，在设计应用程序的时候，就应注意将TokenRate请求设置成一个合理的数量。

假如应用程序不要求对传输速度进行安排，可将这一字段设为QOS_NOT_SPECIFIED（-1）。

(2) TokenBucketSize

如前所述，TokenBucketSize字段面向一个指定的流，限制它能累积的“信用点”数量。例如，对一个视频应用程序而言，可考虑将该字段设为准备传输的视频帧的大小，因为我们要求每个视频帧最好都能一次传送出去。而对于那些要求数据传输速度固定的应用，应考虑将该字段设置成允许一些变化。类似于TokenRate字段，TokenBucketSize采用的单位也是“字节／秒”。

(3) PeakBandwidth

PeakBandwidth规定了在指定时间范围内，最多能传送多少数据。事实上，这个值对应着“突发”传输允许的最大数据量。这是一个至关重要的值，因为可用它防止应用程序积累数量众多的传输令牌，然后一古脑地传送出去，造成网络上的数据“泛滥成灾”。PeakBandwidth的原意便是“高峰带宽”，采用的单位仍然是“字节／秒”。

(4) Latency

Latency字段规定了从一个位传送出去之后，到接收者（可能有多个）收到它之前，最多允许存在多长时间的延迟。至于具体怎样对该值进行解释，要由在 ServiceType字段中请求的服务等级决定。Latency指定的是一个时间长度，采用的单位是“毫秒”。

(5) DelayVariation

DelayVariation指定一个数据包允许的最短及最长延迟时间的差距。通常，应用程序依据这个值来决定安排多大的缓冲区空间，来接收数据，同时仍然维持最初的数据传输样式。DelayVariation的单位也为毫秒。

(6) ServiceType

ServiceType（服务类型）字段定义了数据流要求的服务等级。可指定下述服务类型：

- SERVICETYPE_NOTRAFFIC：表明沿这个方向，不会有数据传输出去。
- SERVICETYPE_BESTEFFORT：指出服务类型取决于FLOWSPEC结构中规定的参数。系统会进行恰当的努力，来维持那个服务等级。然而，却不对数据包是否能正常投递出去，提供任何保障。这正是所谓的“最大努力”通信。
- SERVICETYPE_CONTROLLEDLOAD：表明数据传输的质量非常近似于在“无负担

通信”的条件下，由“最大努力”服务所提供的质量。所谓“无负担通信”，通常可分为两种情况。第一种情况，数据包的丢失接近传输媒介正常的出错率；第二种情况，通信延迟不会大幅超过投递数据所经历的最小延迟时间。

- SERVICETYPE_GUARANTEED：在连接建立期间，严格保证数据传输以 TokenRate字段规定的速度进行。然而，假如实际的数据传输速度超过 TokenSize，那么数据可能延误，也可能丢弃（具体由通信控制或 TC配置决定）。除此以外，假如没有超过TokenRate的设定，Latency（延迟）时间也必须保证。

除了这四种服务类型之外，另外还有几个标志，可用来提供返回给应用程序的信息。这些信息性标志可通过与任何有效 ServiceType标志的OR（或）运算，结合到一起。表 12-1全面总结了这些信息性的标志。

表12-1 服务类型修改符标志

值	含 义
SERVICETYPE_NETWORK_UNAVAILABLE	指定在数据的发送或接收方向，服务不可用
SERVICETYPE_GENERAL_INFORMATION	指出一个数据流支持的所有服务类型
SERVICETYPE_NOCHANGE	指出请求的 QoS服务等级没有发生变化。这个标志可自一次Winsock调用返回。另外，应用程序可通过与 QoS的重新协商，来指定这个标志，表明在指定的方向上， QoS等级没有发生变化
SERVICE_IMMEDIATE_TRAFFIC_CONTROL	应用程序可用这个标志要求系统立即调用通信控制（ TC），而不是进行“最大努力”通信，除非一条 RESV消息抵达
SERVICE_NO_TRAFFIC_CONTROL	该标志可与其他 ServiceType标志通过OR（或）运算合并到一起，完全禁止通信控制
SERVICE_NO_QOS_SIGNALING	这个标志随前述的立即通信控制标志（ SERVICETYPE_IMMEDIATE_TRAFFIC_CONTROL）一道使用，禁止发出任何RSVP传信消息。尽管可调用本地通信控制，但却不能发出 RSVP PATH消息。这个标志亦可与一个负责接收的 FLOWSPEC结构联合使用，从而禁止 RESV消息的自动生成。应用程序会收到通知，知道一条 PATH 消息已经抵达，然后要执行 WSAIoctl（SIO_SET_QOS），取消对这个标志的设置，允许 RESV消息送出

(7) MaxSduSize

MaxSduSize字段指出在某个流内传输时，数据包的最大长度是多少。 MaxSduSize以字节数为单位。

(8) MinimumPolicedSize

MinimumPolicedSize字段定义了某个流内允许的最小数据包长度。 MinimumPolicedSize的值也用字节数为单位。

12.2.2 QoS调用函数

现在，假定我们想让自己的应用程序在网络上发出一个请求，提出特定的带宽要求。有四个函数可以初始化这个过程。一旦启动了一个 RSVP会话之后，应用程序便可注册 FD_QOS事件。QoS状态信息和错误代码会以 FD_QOS事件的形式，传输给应用程序。应用程序可注册以普通方式来接收这些事件：在 WSAAsyncSelect或WSAEventSelect函数的事件字段内，设置FD_QOS标志。

假如用FLOWSPEC结构来建立一个连接，同时指定的是默认值（QOS_NOT_SPECIFIED），亦即“没有指定QoS”，那么FD_QOS通知就显得特别重要。应用程序发出了对QoS的请求之后，位于基层的“提供者”便会定时更新FLOWSPEC结构，指出当前的网络状态，并会利用一个FD_QOS事件，向应用程序发出通知。通过这种信息，应用程序便可请求或修改QoS级别，以反映出可用的带宽大小。在此要注意的是，更新信息只指出了本地可用的带宽有多大，并不一定同时反映着“端到端”通信的带宽。

建好一个数据流之后，可用的网络带宽可能发生变化，或者参与通信的某一方临时改变了主意，想更改要求的QoS服务等级。此时，通过对已分配资源的一次重新协商，会造成FD_QOS事件的生成，向应用程序指出发生了什么改变。此时，应用程序应调用SIO_GET_QOS，以取得新的资源等级。本章后面讲到QoS编程时，QoS事件传信以及状态信息的详情还会继续深入下去。

1. WSAConnect

客户机可使用一个WSAConnect函数，初始化与一个服务器的单播（也称为单点传送）QoS连接。要求的QoS值通过lpSQOS参数传递过去。目前，成组QoS（Group QoS）尚未得以支持或实现；所以对lpGQOS这个参数而言，应为其传递一个空值（NULL）。

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

其中，WSAConnect调用既可用于“面向连接”的套接字，也可用于“无连接”的套接字。对面向连接的套接字来说，该函数除建立连接之外，还会生成适当的PATH以及（／或者）RESV消息。而对无连接的套接字来说，必须将一个端点的地址同套接字关联到一起，让服务提供者知道将PATH和RESV消息发送哪里。若在一个无连接的套接字上使用WSAConnect，要注意的一个问题是，只有传给那个目标地址的数据才会由系统根据套接字的既定QoS等级进行封装。换句话说，WSAConnect用于联系一个无连接套接字上的一个端点，在套接字的有效期内，数据只能在那两个端点之间传递。如果需要在保证QoS的前提下将数据发给多个端点，请用WSAIoctl和SIO_SET_QOS来指定每一个新端点。

2. WSAAccept

WSAAccept函数负责接受一个具备QoS功能的客户机的连接。该函数的原型如下：

```
SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR *addr,
    LPINT  addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD  dwCallbackData
);
```

如果想支持条件函数，必须建立如下原型：

```
int CALLBACK ConditionalFunc(
```

```
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR *g,
    DWORD dwCallbackData
);
```

但令人非常遗憾的是，QoS服务提供者并不保证能够返回客户机通过 lpSQOS参数实际请求的QoS值。也就是说，要想在客户机套接字上启用 QoS，在WSAAccept调用之前以及之后，必须调用WSAIoctl，同时设置SIO_SET_QOS。假如在监听套接字上设置 QoS，那些值在默认情况下，会复制给客户机套接字。

实际上，条件函数根本没什么用处。使用 TCP，我们实际根本不能拒绝一个客户机连接，因为到条件函数调用的时候，连接已在 TCP这一级建立起来了。除此以外，即使已经有一个PATH消息抵达，QoS服务提供者也不会将有效的 QoS参数传递给条件函数。总之，请不要使用WSAAccept条件函数。

注意 若在Windows 98上使用WSAAccept，那么必须特别注意这个问题。但假如确实随WSAAccept使用了条件函数，同时lpSQOS参数不为空（NULL），便必须设置QOS（使用SIO_SET_QOS），否则WSAAccept调用会失败。

3. WSAJoinLeaf

WSAJoinLeaf用于多点通信。在第11章中，我们在一个更深入的层次，探讨了多播（多点传送）的问题。该函数的定义如下：

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

要想使一个应用程序加入多播会话，必须创建一个套接字，同时设置相应的标志（WSA_FLAG_MULTIPOINT_C_ROOT、WSA_FLAG_MULTIPOINT_C_LEAF、WSA_FLAG_MULTIPOINT_D_ROOT和WSA_FLAG_MULTIPOINT_D_LEAF）。应用程序设置多点通信的时候，需要在lpSQOS参数中设定QoS参数。

用WSAJoinLeaf加入IP多播组时，多播组的加入操作以及 QoS RSVP会话的设置是分开来进行的。事实上，一个多播组的加入也只是“有可能”成功。函数返回的时候，预约请求尚未完成。在以后的某个时间，我们会接收到一个 FD_QOS事件，通知我们早先请求的资源分配操作到底是成功，还是失败。

在此请留意针对多播数据设置的“存在时间”（TTL）。假如计划使用 SIO_MULTICAST_SCOPE或IP_MULTICAST_TTL来设置TTL，那么TTL必须在调用 WSAJoinLeaf或者调用

SIO_SET_QOS Ioctl命令对套接字的QoS进行设置之前进行设置。假如在QoS已经设好之后再来设置TTL，那么除非QoS通过SIO_SET_QOS进行了重新协商，否则TTL不会产生任何效果。TTL设置的值也会通过RSVP请求传递出去。

在套接字上设置QoS之前，便应完成TTL的设置，这是至关重要的一个注意事项，因为套接上的TTL设置也会对RSVP消息的TTL产生影响，从而直接影响你的资源预约请求最终能传播到几个网络中去。举个例子来说，如果我们希望在一个 IP多播组内设置几个端点，而那个多播组同时跨越了3个网络。此时，最理想的设定是让 TTL=3，使我们以后产生的任何网络通信都不会“蔓延”至其他不相干的网络。假如在调用 WSAJointLeaf之前，没有设置TTL，那么在RSVP消息送出的时候，会自动采用的默认的 TTL设置：63！显然，这会造成主机从跨越许多个网络的地方，预约资源，从而造成通信效率的严重下降。

4. WSAIoctl

在设置了I/O控制选项SIO_SET_QOS的前提下，如果调用WSAIoctl函数，那么既可用来在一个连接或未连接的套接字上首次请求 QoS，亦可用来在初次QoS请求完成以后，对QoS的要求进行重新协商。使用 WSAIoctl的一个好处在于，假如 QoS请求失败，那么经由提供者所特有的一些信息，更详细的错误信息可在失败后返回，供我们参考，找出错误出在哪里。在第9章中，我们已探讨了WSAIoctl函数的详情，并解释了具体如何调用它（即设置 SIO_SET_QOS或SIO_GET_QOS）。

SIO_SET_QOS选项用于在一个套接字上设置或修改 QoS参数。随SIO_SET_QOS调用WSAIoctl的一个好处在于，我们可指定由具体提供者决定（提供者所特有）的一些对象，以便进一步推敲、落实QoS的行为。在下一节内，我们打算专门讲述提供者所特有的全部对象。特别要指出的是，假如一个应用程序采用的是“无连接”套接字，而且不愿意使用 WSAConnect，那么可随SIO_SET_QOS一道，来调用WSAIoctl，同时在提供者特有缓冲区内，指定目标地址对象，从而与一个“端点”联系到一起，最终促使一个 RSVP会话的正常建立。设定QoS参数时，请以lpvInBuffer的形式，传递QoS结构，同时用cbInBuffer来指定传递的字节量大小。

SIO_GET_QOS选项则在FD_QOS事件接收到之后使用。若某个应用程序收到了这种事件通知，就应随SIO_GET_QOS一道，发出对WSAIoctl的一个调用，对产生该事件的原因进行调查。就像我们早先指出的那样，之所以会产生 FD_QOS事件，原因不外乎网络中可用带宽发生了变化，或者通信对方进行了重新协商。要想获得一个套接字的 QoS值，以lpvOutBuffer的形式，传递一个足够大的缓冲区，同时用 cbOutBuffer指定具体大小。输入参数可为 NULL（空值），或为0（零值）。调用SIO_GET_QOS时，要注意的一点是必须传递一个足够大的缓冲区，以便装下整个QoS结构，包括由具体提供者决定的对象。 ProviderSpecific字段（亦即一个WASBUF结构）位于QoS结构之内。如果将 len字段设为QUERY_PS_SIZE，同时buf字段为NULL（空），那么在自WSAIoctl返回后，len字段便会发生更新，换成需要的大小。此外，假如由于缓冲区过小，造成调用失败，那么 len字段的值也会更新为正确的大小。只有在Windows 2000操作系统中，才支持对缓冲区大小的查询。而对 Windows 98来说，请务必提前提供一个足够大的缓冲区——选好一个大缓冲区后，便不要另行变化。

随WSAIoctl一道，还可使用另一个I/O控制命令：SIO_CHK_QOS。该命令可用于查询如表12-2所示的6个值。调用这个命令时， lpvInBuffer参数会指向一个DWORD（双字），它会设

为三个标志之一。lpvOutBuffer参数也应指向一个DWORD；而且在返回之后，请求的值也会返回。最常用的标志是ALLOWED_TO_SEND_DATA。假如发送者初始化了一条PATH消息，但却没有收到任何RESV消息，指出QoS等级的成功分配，便可使用该标志。若发送者在设置了ALLOWED_TO_SEND_DATA的前提下，使用SIO_CHK_QOS这个I/O控制命令，便会对网络进行查询，了解当前可用的“尽最大努力”通信是否足以发送在 QoS结构中描述的那种数据（该结构已传递给一个发出QoS调用的函数）。欲知详情，请参阅第9章对该I/O控制命令的详细解释。

表12-2 SIO_CHK_QOS标志

SIO_CHK_QOS标志	说 明	返 回 值
ALLOW_TO_SEND_DATA	指出数据的发送是立即开始，还是应该在应用程序接收到一条RESV消息后开始	BOOL（布尔值，下同）
ABLE_TO_RECV_RSVP	指出发送者的接口是否支持RSVP功能	BOOL
LINE_RATE	返回接口的带宽容量	DWORD（双字）
LOCAL_TRAFFIC_CONTROL	返回传输控制（TC）是否已经安装，并可开始使用	BOOL
LOCAL_QOSABILITY	返回QoS是否可用	BOOL
END_TO_END_QOSABILITY	判断端到端的QoS是否可在网络上使用	BOOL

表12-2列出的返回值实际返回的是1或0值，分别对应“是”或“否”这两种值。假如系统当前不能得到确切的值，表中最后四个选项可返回常数 INFO_NOT_AVAILABLE（信息不可用）。

12.3 QoS中止

在前一节内，我们探讨了如何在一个套接字上调用 QoS。接下来，我们打算对QoS服务的中止进行讨论。下列每个事件都会造成RSVP的中止，同时停止为一个套接字处理“通信控制”（TC）。

- 用closesocket函数关闭一个套接字。
- 用shutdown函数关闭一个套接字。
- 用一个空的远程地址（通信对方的地址）调用 WSAConnect。
- 使用SERVICETYPE_NOTRAFFIC或者SERVICE_TYPE_BESTEFFORT服务类型，调用WSAIoctl和SIO_SET_QOS。

除上述列表的第二项事件之外，其他各个事件都是很容易明白的。对于 shutdown函数来说，请务必注意它既可能意味着数据发送的停止，也可能意味着接收的停止。发生两种情况的任意一个，都会造成在与之对应的那个方向上，数据流的停止。换言之，假如随 SD_SEND（发送）调用shutdown，那么对于正在接收的数据，QoS仍然是有效的。

由提供者决定的对象

本小节打算讲述那些由具体提供者所决定的对象。它们会作为 QoS结构的ProviderSpecific字段的一部分进行传递。这些对象要么通过 FD_QOS事件，将QoS信息返回至我们的应用程序；要么可随其他QoS参数一道，传递给WSAIoctl，同时设定SIO_SET_QOS选项，以便对QoS的行为作进一步的落实。

在由提供者决定的对象中，每个都包含了一个 QOS_OBJECT_HDR（对象头）结构。该

结构是这种对象的第一个成员。这个结构指定了提供者特有对象的类型。该结构是至关重要的，因为这些提供者对象通常会在经过了对 SIO_GET_QOS的一次调用之后，在QoS结构中返回。使用QOS_OBJECT_HDR，我们的应用程序可以标识出每个对象，同时对其签名进行解码。对象头的定义如下：

```
typedef struct
{
    ULONG   ObjectType;
    ULONG   ObjectLength;
} QOS_OBJECT_HDR, *LPQOS_OBJECT_HDR;
```

其中，ObjectType指出预设的提供者特有对象的类型是什么。而 ObjectLength指出整个对象的长度是多少。在这个长度中，同时包括了对象头，以及提供者特有对象本身。表 12-3对可选的对象类型标志进行了总结。

表12-3 对象类型

提供者对象	对象结构
QOS_OBJECT_PRIORITY	QOS_PRIORITY
QOS_OBJECT_SD_MODE	QOS_SD_MODE
QOS_OBJECT_TRAFFIC_CLASS	QOS_TRAFFIC_CLASS
QOS_OBJECT_DESTADDR	QOS_DESTADDR
QOS_OBJECT_SHAPER_QUEUE_DROP_MODE	QOS_SHAPER_QUEUE_LIMIT_DROP_MODE
QOS_OBJECT_SHAPER_QUEUE_LIMIT	QOS_SHAPER_QUEUE_LIMIT
RSVP_OBJECT_STATUS_INFO	RSVP_STATUS_INFO
RSVP_OBJECT_RESERVE_INFO	RSVP_RESERVE_INFO
RSVP_OBJECT_ADSPEC	RSVP_ADSPEC
RSVP_OBJECT_POLICY_INFO	RSVP_POLICY_INFO
QOS_OBJECT_END_OF_LIST	无。没有更多的对象

1. QoS优先级

QoS优先级定义了数据流的绝对优先级大小。优先级本身的取值范围在 0到7之间，从最低级到最高级，优先程度越来越高！ QOS_PRIORITY结构定义如下：

```
typedef struct _QOS_PRIORITY
{
    QOS_OBJECT_HDR  ObjectHdr;
    UCHAR           SendPriority
    UCHAR           SendFlags;
    UCHAR           ReceivePriority;
    UCHAR           Unused;
} QOS_PRIORITY, *LPQOS_PRIORITY;
```

这些优先级设定决定了本地对应数据流传输的优先等级大小（在负责发送数据的主机计算机内部），它相对于来自其他流的数据传输。对一个数据流来说，默认的优先等级是 3。这一优先级需要与FLOWSPEC的ServiceType（服务类型）参数联合使用，最终决定应将何种优先级应用于“ 包调度器 ”（Packet Scheduler）内部的数据流。 SendFlags和ReceivePriority目前尚未使用，但将来却不一定。

2. QoS封装丢弃模式

QoS对象定义了通信控制（TC）的“ 包封装器 ”（Packet Shaper）如何处理一个指定流的数据。假如一个数据流与 FLOWSPEC中指定的参数不符，那么在处理这个流时，通常就需要

用到该属性。换言之，假如应用程序采用比发送 FLOWSPEC的TokenRate字段设置的更快的速度来发送数据，我们便认为两者“ 不相符 ”。这个对象定义了作为本地系统，该如何应付这种情况。QOS_SD_MODE结构定义如下：

```
typedef struct _QOS_SD_MODE
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            ShapeDiscardMode;
} QOS_SD_MODE, *LPQOS_SD_MODE;
```

其中，ShapeDiscardMode字段的值可从表12-4中选择。

大家或许会觉得奇怪，为何需要使用 TC_NONCONF_DISCARD模式，在数据还没有通过线缆发送出去之前，便将其丢弃呢？事实上，在传送声音或影像数据时，有时便要采取这种看似“ 反常 ”的行动。大多数情况下， FLOWSPEC结构在设置的时候，都会让一个数据包的大小正好等于一个影像帧的大小，或令其相当于一个小的声音片断。假如出于某个方面的原因，数据包发生了不相符的情况，那么对应用程序来说，采取哪种对策更好一些呢？是稍等一下，直至两者相符为止（就像 TC_NONCONF_SHAPE的情况）吗？还是完全丢弃当前数据包，然后转移到下一个包的传递呢？通常，对那些对“ 实时性 ”要求较为严格的应用，比如影音文件的传输，那么一种更好的做法是丢弃当前帧，立即转移到下一个帧。

表12-4 QoS封装丢弃模式标志

标 志	说 明
TC_NONCONF_BORROW	为高优先等级的数据流提供了服务之后，用数据流接收剩余的资源。这种类型的流既不是“ 封装器 ”（Shaper），也不是“ 排序器 ”（Sequencer）。假如为TokenRate指定了一个值，那么数据包可能出现不一致的情况，可能会降级低于“ 尽最大努力 ”优先级
TC_NONCONF_SHAPE	必须为TokenRate指定一个值。不相符的数据包会保持在“ 包封装器 ”（Packet Shaper）内，直至相符为止
TC_NONCONF_DISCARD	必须为TokenRate指定一个值。不一致的数据包会被无情地丢弃

3. QoS通信类别

QOS_TRAFFIC_CLASS结构可携带由一个第 2层网络设备提供给主机的 802.1p通信类参数。该结构的定义如下：

```
typedef struct _QOS_TRAFFIC_CLASS
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            TrafficClass;
} QOS_TRAFFIC_CLASS, *LPQOS_TRAFFIC_CLASS;
```

主机会用结构中指定的 TrafficClass值，填写对应的、已传送的数据包的 MAC头。尽管该结构已包容在Qos.h文件中，但却不允许应用程序对自己的优先级进行设置。

4. QoS目标地址

QOS_DESTADDR结构用于为一个“ 无连接 ”的发送套接字指定目标地址，同时不必使用一个WSAConnect调用。此时，除非确切知道了无连接套接字的目标地址，否则不会有RSVP PATH或者RESV消息发送出去。我们可用 SIO_SET_QOS这个I/O控制命令对目标地址进行设置。结构定义如下：

```
typedef struct _QOS_DESTADDR
```

```
{
    QOS_OBJECT_HDR          ObjectHdr;
    const struct sockaddr *SocketAddress;
    ULONG                   SocketAddressLength;
} QOS_DESTADDR, *LPQOS_DESTADDR;
```

其中，SocketAddress字段指定的是一个SOCKADDR结构，它为指定的协议定义了端点地址。SocketAddressLength则指定了SOCKADDR这个结构的长度。

5. QoS封装器队列限制丢弃模式

这个结构定义了在抵达一个流的封装器队列长度限制后，用于代替数据包默认丢弃方案的任何一种自定义丢弃方案。请大家记住，对于“传输控制”的传输封装器（Traffic Shaper）模块来说，我们对其进行配置，在当前队列中的数据与FLOWSPEC结构的定义不相符时，丢弃任何多余的数据。结构定义如下：

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT_DROP_MODE
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            DropMode;
}  QOS_SHAPER_QUEUE_LIMIT_DROP_MODE,
  *LPQOS_SHAPER_QUEUE_LIMIT_DROP_MODE;
```

对DropMode来说，可选的两个值如表12-5所示。

表12-5 封装器队列限制丢弃模式

标 志	含 义
QOS_SHAPER_DROP_FROM_HEAD	从队列头中丢弃数据包（此乃默认行为）
QOS_SHAPER_DROP_INCOMING	一旦抵达队列限制，便丢弃任何进入的数据包

6. QoS封装器队列限制

我们可用自定义的“封装器队列限制结构”来代替封装器队列的默认流限制。该结构的定义如下：

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            QueueSizeLimit;
} QOS_SHAPER_QUEUE_LIMIT, *LPQOS_SHAPER_QUEUE_LIMIT;
```

其中，QueueSizeLimit指定封装器队列的长度，以字节为单位。或设定一个较大的封装器队列长度，可防止数据的“无辜”丢弃，允许那些数据在不会用光缓冲区空间的前提下，正常地“排队”。

7. RSVP状态信息

RSVP状态信息对象用于返回RSVP特有的一些错误，以及一些状态信息。结构定义如下：

```
typedef struct _RSVP_STATUS_INFO {
    QOS_OBJECT_HDR       ObjectHdr;
    ULONG            StatusCode;
    ULONG            ExtendedStatus1;
    ULONG            ExtendedStatus2;
} RSVP_STATUS_INFO, *LPRSVP_STATUS_INFO;
```

其中，StatusCode字段对应于返回的RSVP消息。在表12-6中，我们总结了可能用到的代

码。另两个字段——ExtendedStatus1和ExtendedStatus2则为与具体提供者相关的信息保留。

表12-6 RSVP状态信息代码

标 志	含 义
WSA_QOS_RECEIVERS	至少有一条RESV消息抵达
WSA_QOS_SENDERS	至少有一条PATH消息抵达
WSA_NO_QOS_RECEIVERS	没有接收者
WSA_NO_QOS_SENDERS	没有发送者
WSA_QOS_REQUEST_CONFIRMED	预约已被确认
WSA_QOS_ADMISSION_FAILURE	由于缺乏资源，导致请求失败
WSA_QOS_POLICY_FAILURE	由于管理方面的原因，或者由于身份资料的错误，造成请求被拒
WSA_QOS_BAD_STYLE	未知或冲突的样式
WSA_QOS_BAD_OBJECT	RSVP_FILTERSPEC结构的某个部分或与具体提供者相关的缓冲区存在问题
WSA_QOS_TRAFFIC_CTRL_ERROR	FLOWSPEC结构的某个部分存在问题
WSA_QOS_GENERIC_ERROR	常规错误
ERROR_IO_PENDING	重复操作被取消

通常情况下，在收到一条 RSVP消息后，应用程序需要接收一个 FD_QOS事件，并调用SIO_GET_QOS，以取得一个QoS结构，其中包含了一个 RSVP_STATUS_INFO对象。例如，对于具有QoS能力的，以UDP为基础的接收者来说，会生成包含了一条 WSA_QOS_SENDERS消息的一个FD_QOS事件，指出有“人”请求QoS服务，希望将数据传给接收者。

8. RSVP预约信息

RSVP预约信息对象用于保存与RSVP有关的信息，以便通过Winsock 2 QoS API函数以及与特定提供者对应的缓冲区，对双方之间的交互进行调整。 RSVP_RESERVE_INFO对象会覆盖默认的预约样式，并由一个QoS接收者使用。该对象定义如下：

```
typedef struct _RSVP_RESERVE_INFO
{
    QOS_OBJECT_HDR    ObjectHdr;
    ULONG             Style;
    ULONG             ConfirmRequest;
    ULONG             NumPolicyElements;
    LPRSVP_POLICY     PolicyElementList;
    ULONG             NumFlowDesc;
    LPFLOWDESCRIPTOR  FlowDescList;
} RSVP_RESERVE_INFO, *LPRSVP_RESERVE_INFO;
```

其中，Style字段指定了需要应用于该接收者的过滤器类型。在表 12-7中，我们总结了目前可选的一些过滤器类型，以及不同类的接收者使用的默认过滤器类型。注意每类过滤器稍后都会详加论述。假如ConfirmRequest字段不为零值，那么一旦接收应用程序收到 RESV请求之后，就会发出通知。 NumPolicyElements与PolicyElementList字段是联系起来的。其中包含了RSVP_POLICY对象的数量，那些对象保存于 PolicyElementList字段中。本章稍后，我们还会对RSVP_POLICY进行定义。接下来，且让我们看看各种不同的过滤器样式，以及各自的特征。

9. RSVP_DEFAULT_STYLE

这个标志指示QoS服务提供者使用默认样式。表12-7总结了各种不同接收者采用的默认样式。单播接收者使用固定过滤器，而通配符用于多播接收者。调用 WSAConnect的UDP接收者

亦可使用固定过滤器。

表12-7 默认过滤器样式

过滤器样式	默认用户
固定过滤器	单播接收者，已连接的UDP接收者
通配符	多播接收者，未连接的UDP接收者
共享显式	无

10. RSVP_FIXED_FILTER_STYLE

通常，这种样式会在接收者以及单独一个数据源之间，建立起一个提供了 QoS保障的数据流。对于单播接收者以及建立连接的 UDP接收者来说，它们采用的便是这种样式：NumFlowDesc设为1，而FlowDescList包含了发送者的地址。但是，我们也完全可以设置采用了多个固定过滤器的样式，允许一个接收者从多个明确指定的数据源处，预约各自独立的数据流。举个例子来说，假如我们的接收者想从三个发送者那里接收数据，并需要为每一个都保证（预约）20Kbps的带宽。此时，我们就需要考虑采用多固定过滤器样式。在这个例子中，NumFlowDesc应设为3，而FlowDescList应包含三个地址，每个FLOWSPEC一个。另外，我们亦可为每个发送者分配不同的 QoS等级；它们并一定要完全相同。要注意的是，单播接收者和建立连接的 UDP接收者不可使用多个固定的过滤器。在图 12-1中，我们展示了FLOWDESCRIIPTOR（流描述符）和RSVP_FILTERSPEC（RSVP过滤器规格）这两个结构间的关系。

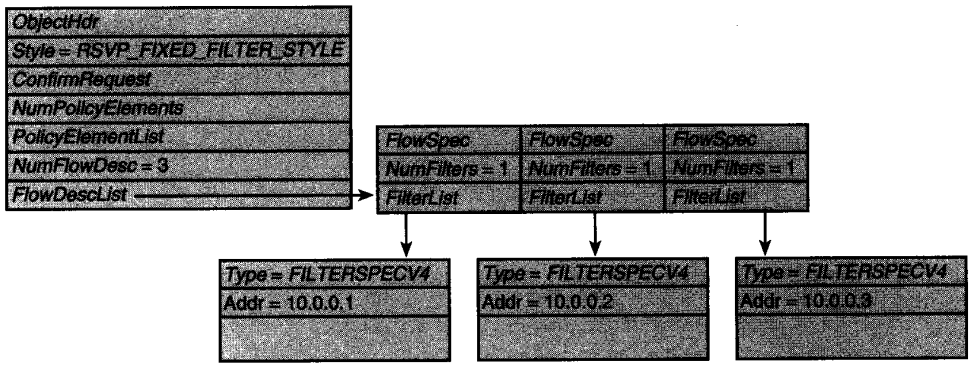

图12-1 多固定过滤器样式

11. RSVP_WILDCARD_STYLE

多播接收者和未连接的 UDP接收者采用的是通配符样式（ WILDCARD）。要想为TCP连接或者已经连接的 UDP接收者使用这一样式，请务必将 NumFlowDesc设为0，并将FlowDescList设为NULL。对于未建好连接的UDP接收者以及多播应用来说，这是它们的默认过滤器样式 —— 因为发送者的地址当时是未知的。

12. RSVP_SHARED_EXPLICIT_STYLE

在某种程度上，该样式类似于多固定过滤器样式，只是两者的网络资源有所区别。对前者来说，不再为每个发送者都分配网络资源。相反，网络资源需要在所有发送者之中共享！在这种情况下，NumFlowDesc为1，而FlowDescList包含了由发送者地址构成的一个列表。图12-2对这一样式进行了阐释。

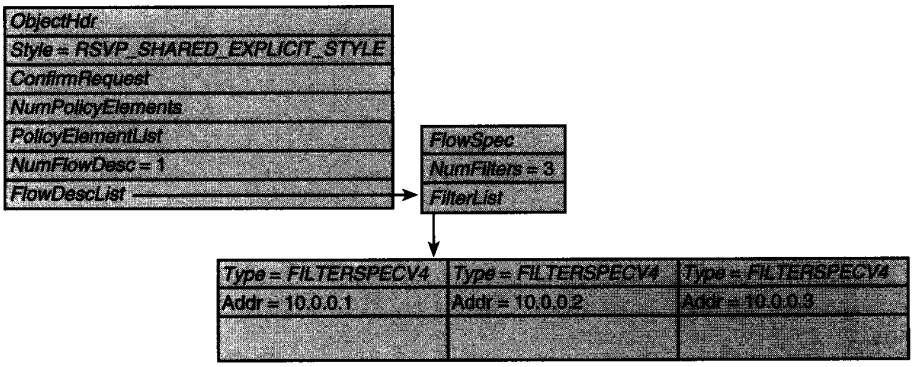

图12-2 共享显式样式

在我们讨论RSVP样式时，已向大家讲解了最后两个字段：NumFlowDesc（数据流的目标数量）和FlowDescList（数据流目标地址列表）。至于具体如何使用这两个字段，要取决于样式本身。其中，NumFlowDesc定义了FLOWDESCRIPTOR（流描述符）结构在FlowDescList字段中的数量。该结构定义如下：

```
typedef struct _FLOWDESCRIPTOR
{
    FLOWSPEC              FlowSpec;
    ULONG                 NumFilters;
    LPRSVP_FILTERSPEC     FilterList;
} FLOWDESCRIPTOR, *LPFLOWDESCRIPTOR;
```

该对象用于定义由 FlowSpec给定的每个 FLOWSPEC的过滤器类型。同样地，在NumFilters字段中，包含了 FilterList数组中存在的 RSVP_FILTERSPEC对象的数量。RSVP_FILTERSPEC对象的定义如下：

```
typedef struct _RSVP_FILTERSPEC {
    FilterType   Type;
    union {
        RSVP_FILTERSPEC_V4       FilterSpecV4;
        RSVP_FILTERSPEC_V6       FilterSpecV6;
        RSVP_FILTERSPEC_V6_FLOW  FilterSpecV6Flow;
        RSVP_FILTERSPEC_V4_GPI   FilterSpecV4Gpi;
        RSVP_FILTERSPEC_V6_GPI   FilterSpecV6Gpi;
    };
} RSVP_FILTERSPEC, *LPRSVP_FILTERSPEC;
```

其中，第一个字段Type是对下述值的一个简单列举：

```
typedef enum {
        FILTERSPECV4 = 1,
        FILTERSPECV6,
        FILTERSPECV6_FLOW,
        FILTERSPECV4_GPI,
        FILTERSPECV6_GPI,
        FILTERSPEC_END
} FilterType;
```

列举指定了联盟中存在的对象。每个过滤器的规格（FILTERSPEC）定义如下：

```
typedef struct _RSVP_FILTERSPEC_V4 {
    IN_ADDR_IPV4    Address;
    USHORT          Unused;
    USHORT          Port;
} RSVP_FILTERSPEC_V4, *LPRSVP_FILTERSPEC_V4;

typedef struct _RSVP_FILTERSPEC_V6 {
    IN_ADDR_IPV6    Address;
    USHORT          UnUsed;
    USHORT          Port;
} RSVP_FILTERSPEC_V6, *LPRSVP_FILTERSPEC_V6;

typedef struct _RSVP_FILTERSPEC_V6_FLOW {
    IN_ADDR_IPV6    Address;
    UCHAR           UnUsed;
    UCHAR           FlowLabel[3];
} RSVP_FILTERSPEC_V6_FLOW, *LPRSVP_FILTERSPEC_V6_FLOW;

typedef struct _RSVP_FILTERSPEC_V4_GPI {
    IN_ADDR_IPV4    Address;
    ULONG           GeneralPortId;
} RSVP_FILTERSPEC_V4_GPI, *LPRSVP_FILTERSPEC_V4_GPI;

typedef struct _RSVP_FILTERSPEC_V6_GPI {
    IN_ADDR_IPV6    Address;
    ULONG           GeneralPortId;
} RSVP_FILTERSPEC_V6_GPI, *LPRSVP_FILTERSPEC_V6_GPI;
```

13. RSVP广告规范

RSVP_ADSPEC对象定义了RSVP“广告规范”（Adspec）中包含的信息。在该RSVP对象中，通常需要指出有哪些类型的服务可用（受控制负载，或者质量担保），PATH消息是否会碰到一个非RSVP跳跃，以及路径上最小的MTU是多大。下面是该结构的定义：

```
typedef struct _RSVP_ADSPEC
{
    QOS_OBJECT_HDR      ObjectHdr;
    AD_GENERAL_PARAMS   GeneralParams;
    ULONG               NumberOfServices;
    CONTROL_SERVICE     Services[1];
} RSVP_ADSPEC, *LPRSVP_ADSPEC;
```

我们感兴趣的第一个字段是GeneralParams，它是类型为AD_GENERAL_PARAMS（常规广告参数）的一个结构。该结构用来定义一些常规的特征参数，正如其名。该对象的定义如下：

```
typedef struct _AD_GENERAL_PARAMS
{
    ULONG       IntServAwareHopCount;
    ULONG       PathBandwidthEstimate;
    ULONG       MinimumLatency;
    ULONG       PathMTU;
    ULONG       Flags;
} AD_GENERAL_PARAMS, *LPAD_GENERAL_PARAMS;
```

其中，IntServAwareHopCount指定的是符合“集成服务”（IntServ）要求的跳数(hop)。PathBandwidthEstimate指定的是从发送者到接收者，至少能使用多大的带宽。

MinimumLatency指定的是数据包通过路由器转发时，最小延迟时间的一个总计，以毫秒为单位。PathMTU指定的是“最大传输单元”（MTU），亦即一个数据包最多能在多大。如超过这一设定，数据包在传输过程中便可能发生分解。Flags字段目前尚未使用。

14. RSVP策略信息

我们要讨论的最后一个提供者对象是RSVP策略信息。该对象的含义非常暧昧——其中包含了来自RSVP的、尚未定义的任意数量的策略元素。该结构定义如下：

```
typedef struct _RSVP_POLICY_INFO  {
    QOS_OBJECT_HDR     ObjectHdr;
    ULONG              NumPolicyElement;
    RSVP_POLICY        PolicyElement[1];
} RSVP_POLICY_INFO, *LPRSVP_POLICY_INFO;
```

其中，NumPolicyElement字段指定了在PolicyElement数组中，总共存在多少个RSVP_POLICY结构。下面是该结构的定义：

```
typedef struct _RSVP_POLICY {
    USHORT  Len;
    USHORT  Type;
    UCHAR   Info[4];
} RSVP_POLICY, *LPRSVP_POLICY;
```

RSVP_POLICY结构是由RSVP传送的数据，对应于策略元素，与我们目前的需要关系不大。

12.4 QoS编程

QoS的核心在于RSVP会话的建立。除非发送出RSVP PATH以及RESV消息，而且得到了处理，同时正确预约了带宽，否则这个会话是不会草率地建立起来的。对应用程序来说，知道RSVP消息在什么时候建立是至关重要的。对发送者来说，在生成PATH消息之前，首先必须知道三个参数：

- 发送FLOWSPEC成员。
- 源IP地址及端口。
- 目标IP地址、端口及协议。

只要调用了一个具有QoS功能的函数，那么FLOWSPEC成员便是已知的了。那些函数包括WSAConnect、WSAJoinLeaf、WSAIoctl（设置SIO_SET_QOS选项）等等。源IP地址和端口号却暂时无法知道，直至完成了套接字的本地绑定。这种绑定既可是暗示性进行的（隐式），比如通过连接来绑定；亦可是明确进行的。最后，应用程序需要知道数据的目的地。在完成了一个连接调用之后，或在建立了“无连接”的UDP连接之后，或者用SIO_SET_QOS这个I/O控制命令在与具体提供者有关的数据中设置了QOS_DESTADDR之后，才能知道数据的目标地址是什么。

类似地，要想生成一条RSVP RESV消息，事先必须知道三件事情：

- 接收FLOWSPEC成员。
- 每个发送者的地址和端口。
- 接收套接字的本地地址和端口。

接收FLOWSPEC成员可通过任何具有QoS能力的Winsock函数获得。每个发送者的地址取

决于过滤器的“样式”，后者可用由具体提供者决定的结构 RSVP_RESERVE_INFO来人工设定（前面已经具体讲过了）。否则的话，亦可从一条 PATH消息中取得这种信息。当然，取决于具体的套接字类型，有时为了生成 RESV消息，并不一定非要事先收到一条 PATH消息，从而取得发送者的地址。这样的一个典型例子便是多播通信中采用的通配符过滤器。发出去的RESV消息会应用于一个会话中的所有发送者。对于单播和 UDP接收者来说，本地地址与端口号不用多费什么脑筋便能取得。然而，多播接收者却不然。在多播接收者的情况下，本地地址与端口分别是多播地址以及它对应的端口号。

在本节内，我们首先要来看看各种不同的套接字类型，以及它们如何与 QoS服务提供者及RSVP消息打交道。然后，我们会解释 QoS服务提供者如何向应用程序发出通知，告知它们发生了某种特定的事件。正确理解了这些概念之后，再来写一个提供 QoS功能的应用程序，便能起到事半功倍效果。要想写出这样的应用程序，其实只需知道如何取得 QoS的质量保证，知道这些保证何时能够兑现，并且知道它们何时和怎样发生变化。

12.4.1 RSVP和套接字类型

到目前为止，大家已对 PATH及RESV RSVP消息如何生成有了一定的概念。在后续的小节中，我们打算讲解各类不同的套接字 —— UDP、TCP以及多播UDP。在这个过程中，大家会学习到它们如何与QoS服务提供者沟通，以生成PATH和RESV消息。

1. 单播UDP

由于我们可选择使用“已连接”和“未连接”的两种 UDP套接字，所以在单播 UDP套接字上设置QoS时，有大量选项可供选择。对 UDP发送者而言，发送 FLOWSPEC是从QoS的某个调用函数中获得的。本地地址和端口号既可通过一个“显式”绑定调用中取得，亦可通过由WSAConnect完成的一个“隐式”绑定中取得。最后要知道的是接收应用程序的地址及端口，这要么是在WSAConnect中明确指定的，要么是通过 QOS_DESTADDR这个由具体提供者决定的结构来传递的（指定 SIO_SET_QOS选项）。要注意的是，假如用 SIO_SET_QOS来设置QoS，那么套接字必须在事前绑定好。

对UDP接收者来说，可调用 WSAConnect，将接收应用程序限制成单独一个发送者。除此以外，使用SIO_SET_QOS这个I/O控制命令，应用程序可指定一个 QOS_DESTADDR（目标地址）结构。否则，也可以在调用 SIO_SET_QOS的时候，不提供任何类型的目标地址。在这种情况下，会生成一条 RESV消息，同时采用通配过滤器样式。事实上，无论通过 WSAConnect指定目标地址，还是通过 QOS_DESTADDR结构来指定，都只有在自己希望应用程序只从一个采用固定过滤器样式的发送者那里接收数据的时候，才能采取这样的做法。

UDP接收者实际可同时调用 WSAConnect及SIO_SET_QOS这两个I/O控制命令，并可按任意顺序调用。假定在 WSAConnect之前调用 SIO_SET_QOS，那么首先会创建一条采用通配过滤器的RESV消息。一旦连接调用完毕，前一次 RESV会话便会关闭；并采用固定过滤器样式，建立一次新的会话。除此以外，若在 WSAConnect之后调用SIO_SET_QOS，那么采用固定过滤器的RESV消息不会中断 RSVP会话，并生成一个通配过滤器样式。相反，它只是简单地更新与现有RSVP会话关联在一起的QoS参数。

2. 单播TCP

TCP会话存在着两种可能性。首先，发送者可能是客户机，它建立与服务器的连接，并

发送数据。第二种可能性则是客户机与之连接的那个服务器才是发送者。在发送者是客户机的情况下，QoS参数可直接在WSAConnect调用中指定，这会造成PATH消息的发出。调用连接之前，也可以调用I/O控制命令SIO_SET_QOS，但除非其中的一个连接调用知道了目标地址，否则根本不会生成任何PATH消息。

假如发送者不巧是服务器，那么服务器需要调用 WSAAccept函数，来接受客户机的连接请求。然而，这个函数并未提供一种方式，可供我们在接受的套接字上设置 QoS。假如通过SIO_SET_QOS，在发出对WSAAccept的调用之间便已设好了QoS，那么以后负责“接受”的套接字会继承监听套接字上设定的 QoS等级。要注意的是，假如发送者在 WSAAccept中使用条件函数，那么函数应同时传递在建立连接的那个客户机上设置的 QoS值。然而，我们目前遇到的并不是这种情况。 QoS服务提供者传递的只是一些垃圾 —— 无论Windows 98，还是Windows 2000，都会产生这样的行为。例外在于，假如在 Windows 98中，lpSQOS参数不为“空”（NULL），那么必须在条件函数内部，通过 I/O控制命令SIO_SET_QOS，设置某种QoS值。否则，即使返回了CF_ACCEPT，WSAAccept调用也会失败。在接受之后， QoS亦可在客户机套接字上进行设置。

现在，让我们来看看负责接收的TCP应用。第一种情况是调用WSAConnect，同时设置一个接收FLOWSPEC。在这种情况下， QoS服务提供者会创建一个RESV（预约）请求。假如未将QoS参数提供给WSAConnect，那么SIO_SET_QOS这个I/O控制命令可在以后的某个时间设置（结果便造成了一条RESV消息）。最后一种组合是服务器作为接收者使用，这和发送的情况差不多。发出一个 WSAAccept调用之前， QoS可在监听套接字上设置。此时，客户机套接字会继承相同的QoS等级。否则，亦可在条件函数中设置QoS，或在套接字已被接受之后设置。无论在哪种情况下，一旦有一条 PATH消息抵达，那么QoS服务提供者都会生成一条 RESV消息。

3. 多播

多播发送者的行为和 UDP发送者的行为差不多，唯一的例外在于，现在需要调用 WSAJoinLeaf，从而成为多播组的一名成员，而不是调用 WSAConnect，并指定一个目标地址！可用WSAJoinLeaf对QoS进行设置，或通过一个SIO_SET_QOS调用，对其进行单独设置。多播会话地址用于构成RSVP会话对象，并将该对象包括在RSVP PATH消息之中。

在多播接收者的情况下，除非已通过 WSAJoinLeaf函数指定了多播地址，否则是不会生成RESV消息的。由于多播接收者没有指定一个对方地址（通信对方的地址），所以QoS提供者在生成RESV消息的时候，会采用通配过滤器样式。 QoS服务提供者并不禁止一个套接字同时加入多个多播组。在这种情况下，服务提供者会将 RESV消息发给所有组，只有它们有一条相符的PATH消息。为每个WSAJoinLeaf提供的QoS参数会在每条RESV消息中使用，但假如加入多个组之后，在套接字上调用 SIO_SET_QOS，那么新的QoS参数会应用于已经加入的所有多播组。

若发送者将数据发给一个多播组，那么只有在发送者已经加入它的前提下，才会造成相应的QoS参数应用于那些数据。换言之，假如加入了一个多播组，但在使用 sendto或WSASendTo的时候，却将另外某个多播组设为目的地， QoS便不会应用于那些数据。除此以外，假如一个套接字加入了一个多播组，并指定了一个特定的方向（例如，在 WSAJoinLeaf的dwFlags参数中使用JL_SENDER_ONLY或JL_RECEIVER_ONLY），那么QoS也会相应地产

生作用。若某个套接字仅被设定为接收者，那么在其发送数据的时候，不会享受到 QoS提供的任何服务。

12.4.2 QoS通知

迄今为止，大家已学习了如何为TCP、UDP以及多播UDP套接字调用QoS。大家已经知道，根据自己到底是接收还是发送数据，会有对应的 RSVP事件产生。然而，这些RSVP消息的完成并非与调用它们的API调用严格地关联在一起。也就是说，假如为一个 TCP接收套接字发出一个WSAConnect调用，那么会生成一条RESV消息，但RESV消息与实际的API调用却是无关的。这是由于在调用返回时，并不能保证预约已获得确认，也无法担保网络资源已正确分配。正是考虑到这方面的原因，我们增加了一个新的异步事件： FD_QOS，它需要投递给套接字。典型情况下，在发生下述事件时，便会投递一个 FD_QOS事件通知：

- 通知应用程序的QoS请求被接受，还是被拒绝。
- 网络提供的QoS发生明显改变（与以前协商的值相反）。
- 指出一个QoS通信对方是否已作好准备，可开始收发一个特定流的数据。

1. 注册FD_QOS通知

为利用这些通知带来的好处，应用程序首先必须“ 注册 ”，指明在发生FD_QOS事件的时候，自己想收到通知。有两个办法可完成应用程序的注册。首先，我们可使用 WSAEventSelect或者WSAAsyncSelect，同时在事件标志的按位“ 或 ”运算中，将 FD_QOS这个标志包括进来。然而，只有已发出了对某个 QoS调用函数的一个调用，应用程序才有资格接收 FD_QOS事件。要注意的是，在某些情况下，应用程序可能想接收 FD_QOS事件，同时不想在套接字上设置QoS等级。要达到这个目的，我们可设置一个 QoS结构，在它的发送及接收 FLOWSPEC成员中包含QOS_NOT_SPECIFIED或者SERVICETYPE_NOTRAFFIC标志。这样做唯一的缺点在于，针对自己希望接收事件通知的那个 QoS方向， SERVICE_NO_QOS_ SIGNALING标志必须同SERVICETYPE_NOTRAFFIC标志通过“ 或 ”运算合并到一起。

如果需要准确地了解如何调用两个异步选择函数，请参考第 8章。在该章中，我们对其进行了更为深入、全面地讲解。事件触发后，假如立即使用 WSAEventSelect，那么请调用WSAEnumNetworkEvents函数，获取附加的状态码（如果有的话）。这个函数也在第8章进行了详述。然而，我们在此仍然要稍微讨论一下，因为对 QoS应用来说，它非常简单、短小和重要。我们需要在调用中传递一个套接字句柄、事件句柄以及一个 WSANETWORKEVENTS对象，以便在提供的结构中返回并设置事件信息。该结构定义如下：

```
typedef struct _WSANETWORKEVENTS
{
    long    lNetworkEvents;
    int     iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

其中，lNetworkEvents字段可设为已经触发的所有事件标志的一个按位“ 或 ”运算的结果。要想知道是否发生了一个特定的事件，只需将这个字段的值与那个事件的标志进行一次简单的“ 与 ”运算 —— 使用&运算符。如结果不为零，表明那个事件已经触发。 iErrorCode数组指出遇到了什么错误，或者在 QoS的情况下，指出状态信息。假如触发了一个事件，与那个事件对应的一个标志便会用于索引这个数组。假如数组索引为 0，表明没有发生错误；否则，这

个值就是包含了错误代码的那个数组元素的索引位置。举个例子来说，假定触发了 FD_QOS事件，那么可用FD_QOS_BIT标志对 iErrorCode数组进行索引，检查是否存在错误，或取得相应的状态信息。其他所有Winsock异步事件（FD_READ_BIT和FD_WRITE_BIT等等）都定义了类似的索引标志。

2. RSVP通知

我们早先已经提到，有两个办法均可接收到QoS通知消息。这种信息实际与本节主题——获取QoS事件的结果——密切相关。如果已进行了注册，希望随 WSAAsyncSelect或WSAEventSelect接收FD_QOS通知，而且实际接收到了一个 FD_QOS事件通知，那么我们必须执行对WSAIoctl的一个调用，同时设置 SIO_GET_QOS这个I/O控制选项，以便知道具体是谁触发了事件。实际上并不一定需要为 FD_QOS事件注册，可以通过重叠式 I/O，简单地调用WSAIoctl，同时设置SIO_GET_QOS选项。这同时也要求我们指定一个完成例程。一旦 QoS服务提供者检测到QoS发生了一个改变，便会调用这个例程。发生回调的时候，在输出缓冲内，应该能找到一个QoS结构。

无论在哪种情况下，一旦QoS内发生了变化，我们的应用程序便能相应地得到通知，前提是已为FD_QOS进行了注册，或者使用了重叠 I/O以及SIO_GET_QOS。如果我们选择的是为FD_QOS进行注册，那么在收到事件通知之后，通常还需要调用 WSAIoctl，同时使用SIO_GET_QOS这个I/O控制命令。对这两种方法来说，在返回的 QoS结构中，都只包含了针对单独一个方向的 QoS信息。换言之，对于无效方向上的 FLOWSPEC结构来说，在其ServiceType字段中，会设置SERVICETYPE_NOCHANGE（服务类型未改变）。除此以外，同时可能发生多个QoS事件。在这种情况下，我们应循环调用 WSAIoctl和SIO_GET_QOS，直到返回SOCKET_ERROR，同时WSAGetLastError返回一个WSAEWOULDBLOCK值。调用SIO_GET_QOS时，要注意的最后一个问题是缓冲区的大小。触发了一个 FD_QOS事件之后，可能返回与具体提供者有关的对象。事实上，经常都会返回一个 RSVP_STATUS_INFO结构，只要缓冲区足够大！请参考以前讲述 WSAIoctl的内容，了解具体如何判断一个缓冲区的正确大小。

假如应用程序采用了某个异步事件函数，那么特别要注意的一个问题是，一旦发生FD_QOS事件，那么无论如何都必须执行一个 SIO_GET_QOS操作，以便重新允许FD_QOS通知行为。

现在，大家已经知道了如何接收 QoS事件通知，以及如何获取新的QoS参数（那些事件的结果），但到底会发生何种类型的通知呢？对一个 QoS事件来说，触发它的第一个、也是最明显的一个原因是某个指定数据流的 FLOWSPEC参数发生了改变。举个例子来说，假定我们将一个套接字设置成提供“尽最大努力”的服务。那么，作为 QoS的服务提供者，它会向我们的应用程序定时发出通知，指出网络当前的状态是什么。除此以外，假如我们设置一个“受控制的负载”（Controlled Load），同时设置其他一系列参数，那么一旦发生预约，与令牌大小及令牌速度对应的 QoS参数可能会与我们请求的值稍有区别。在我们的应用程序中，一旦收到一个QoS通知，便应立即将返回的 FLOWSPEC同我们最初请求的进行对比，确定应用程序能够继续运行下去。还要记住的是，在提供 QoS能力的一个套接字发挥作用期间，随时都可执行一个SIO_SET_QOS命令，更改任何参数。这也会促使一个 QoS事件通知的生成，可知会与当前RSVP会话联系在一起的通信方（也许不止一个）。作为一个“健壮”的应用程序，

必须能够应付所有这些情况。

除更新QoS参数以外，QoS事件通知还可以告诉我们发生了另外一些事情，比如由发送者或接收者发出的通知等等。在前文的表 12-6中，我们已列出了所有可能的事件。有两个办法可获得这些状态代码。第一个办法是作为 RSVP_STATUS_INFO对象的一部分。发生了一个QoS事件，且已进行了对 SIO_GET_QOS的一个调用之后，一个 RSVP_STATUS_INFO对象便可能作为提供者特有缓冲区的一部分而返回。第二个办法，如果用 WSAEventSelect来注册事件，这些代码便可在自 WSAEnumNetworkEvents返回的WSANETWORKEVENTS结构中返回。表12-6定义的代码可在iErrorCode数组中找到，该数组由 FD_QOS_BIT进行索引。表中总结的头五个代码并非错误代码。通过这些代码，可知道与 QoS连接状态有关的宝贵信息。表中列出的其他状态代码则全部属于 QoS错误。虽然发生了这些错误，但它们并不能阻止数据的继续收发，它们的作用仅仅是“ 指出 ”在 QoS会话中发生了一个错误。当然，在这种情况下发送的数据根本无法承诺提供我们要求的 QoS保证。

3. WSA_QOS_RECEIVERS和WSA_QOS_NO_RECEIVERS

进行“ 单播 ”通信的时候，一旦发送者开始运行，并接收到了第一条 RESV消息，一个WSA_QOS_RECEIVERS便会上传给应用程序。假如接收者采取了任何操作来禁止 QoS，其结果便是一条RESV取消消息。发送者收到这条消息后， WSA_QOS_NO_RECEIVERS（无接收者）便会上传给应用程序。当然，在单播通信的情况下，许多接收者都只是简单地将套接字关闭了事，从而同时生成了 FD_CLOSE以及WSA_QOS_NO_RECEIVERS事件。大多数情况下，应用程序对此作出的响应也仅仅是将发送套接字关闭完事。

而在“ 多播 ”通信的情况下，只要接收者的数量发生了改变，而且不为零，那么作为发送应用程序，无论如何都会收到 WSA_QOS_RECEIVERS。换言之，每次有一个QoS接收者加入多播组时，以及每次有一个接收者脱离这个组时，只要组内至少还剩有一个接收者，多播发送者都会收到WSA_QOS_RECEIVERS。

4. WSA_QOS_SENDERS和WSA_QOS_NO_SENDERS

发送者通知和接收者事件差不多，唯一的例外是它处理的是 PATH消息的收到事件。对单播接收者来说，启动之后，假若收到第一条PATH消息，便会生成一个WSA_QOS_SENDERS；而PATH撤消消息会造成一条WSA_QOS_NO_SENDERS消息的生成。

类似地，一旦发送者的数量增加或减少，而且不为零，那么多播接收者就会收到一个WSA_QOS_SENDERS通知。一旦发送者的数量变为 0，WSA_QOS_NO_SENDERS消息便会传递给应用程序。

5. WSA_QOS_REQUEST_CONFIRMED

这是最后一条状态消息。如果应用程序要求在确认了一个预约请求之后，便收到相应的通知，那么负责接收的 QoS应用程序会收到这个消息。在 RSVP_STATUS_INFO结构内，有一个名为ConfirmRequest（确认请求）的字段。假如该字段的值不为零，便意味着在预约请求通过确认后， QoS服务提供者需要向应用程序发出一个通知。该对象是一个需要由提供者决定的选项，可随QoS结构传递给SIO_SET_QOS这个I/O控制命令。

12.4.3 QoS模板

Winsock提供了几个预先定义好的 QoS结构，我们称其为“ 模板 ”（Template）。作为应用

程序，可按名字对其进行查询。这些模板为某些常见的音频及视频编码／解码规范（如 G711和H263QCIF）定义了相应的QoS参数。WSAGetQOSByName定义如下：

```
BOOL WSAGetQOSByName(
    SOCKET s,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);
```

假如不知道已安装的那些模板的名字，首先可用该函数“列举”出所有模板名。但要想真正成功，必须用 lpQOSName提供一个足够大的缓冲区，并将它的第一个字符设为空字符，并为lpQOS传递一个空指针。如下述代码所示：

```
WSABUF  wbuf;
char    cbuf[1024];

cbuf[0] = '\0';
wbuf.buf = cbuf;
wbuf.len = 1024;
WSAGetQOSByName(s, &wbuf, NULL);
```

返回之后，在字符缓冲内，会填入一个字串数组，各字串之间用一个空字符（ NULL）加以分隔，而且整个列表会用另一个空字符加以中止。换言之，最后一个字串条目会有两个连续的空字符。根据这个缓冲区的内容，便能知道已安装的所有模板的名字，并可从中查询一个特定的。例如，下述代码可对 G711模板进行查询：

```
QOS     qos;
WSABUF  wbuf;

wbuf.buf = "G711";
wbuf.len = 4;
WSAGetQOSByName(s, &wbuf, &qos);
```

若请求的QoS模板不存在，那么查询会返回FALSE，错误代码是WSAEINVAL。假若成功，函数会返回TRUE。本书配套光盘上的示例 Qostemplate.c向大家阐述了如何列举已安装的 QoS模板。

除此以外，我们还可以安装自己的 QoS模板，使其他应用程序能根据名字对其进行查询。此时要用到两个函数：WSCInstallQOSTemplate以及WSCRemoveQOSTemplate。其中，第一个函数用于QoS模板的安装，而第二个用于删除。函数原型如下：

```
BOOL WSCInstallQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);

BOOL WSCRemoveQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName
);
```

这两个函数的用法其实是一目了然的。要想安装一个模板，需要调用 WSCInstallQOS Template，同时指定GUID、模板名以及QoS参数。GUID是该模板一个独一无二的标识符，可由Uuidgen.exe这样的工具程序来生成。要想删除模板，只需调用 WSCRemoveQOSTemplate并指定那个模板的名字，同时指定与安装过程相同的一个 GUID就可以了。若成功，这两个函数

都会返回TRUE。

12.5 示例

在本节内，我们打算通过三个实际的例子，来展示QoS的使用。第一个例子使用的是TCP，它在三个例子中显得最“直接了当”，因为它是“面向连接”的。第二个例子使用UDP，而且没有使用任何连接调用。最后一个例子使用的则是多播UDP。在所有这三个例子中，我们都会使用WSAEventSelect，因为它比WSAAsyncSelect稍微简单一些。对于TCP单播的例子，我们在此给出了完整的源码。但对UDP以及多播示例来说，却只给出了一些重要的代码片断，这是由于它们的大多数概念都是相同的，无论使用的套接字是何种类型。要想观看完整源码，请参考本书的配套光盘。这三个例子都要依赖于两个支持例程：PrintQos以及FindProtocolInfo，它们分别定义于Printqos.c以及Provider.c中。前一个例程用于打印QoS结构的内容，而后一个例程用于在提供者目录中查找具有指定属性（如QoS）的一种协议。

12.5.1 单播TCP

TCP QoS示例的完整源码是一份很长的清单，请见后文。本例的源码亦可在光盘上的第12章目录下找到，文件名为Qostcp.c。尽管该清单很长，但假如仔细观察，就会发现它并不是特别复杂。大多数代码都只不过是一些普通的WSASelect代码，我们已在第8章详细讲述过了。唯一的例外是我们在产生FD_QOS事件时做的事情。主函数只是进行解析出参数，建立起套接字，再调用Server或Client函数（具体取决于应用程序作为服务器还是客户机调用）。下面，让我们首先来看看客户机连接的情况。

在这里的所有例子中，都可用一个命令行参数告诉示例何时设置QoS：连接之前、连接过程中、建立连接之后，还是在对方请求QoS进行QoS的本地设置之后（在表12-8中，我们列出了Qostcp.c可以选用的命令行参数）。假如决定在连接建立之前设置QoS（对客户机来说），可将套接字绑定到一个随机端口，再调用SIO_SET_QOS，同时设定一个QOS FLOWSPEC。要注意的是，在调用SIO_SET_QOS之前，实际并非真正需要进行这种绑定。这是由于除非发出一个连接调用，否则对方的地址是不知道的。而只有确切知道了对方的地址，才能建立起一个RSVP会话。

假如用户选择在连接过程中设置QoS，那么示范代码会将QoS结构传递进入WSAConnect调用。这个调用会建立一个RSVP会话，并将客户机同指定的服务器连接起来。否则的话，用户需要指出这个例子应该等候对方设置QoS，而且没有QoS结构传递给WSAConnect。相反，代码会分析发送QoS结构，拿SERVICE_NO_QOS_SIGNALING标志同FLOWSPEC结构中的ServiceType字段进行或（OR）运算，并随SIO_SET_QOS命令一道调用WSAIoctl。这样便可告诉QoS服务提供者不要调用通信控制，而是依然等候RSVP消息。

设好QoS后，会注册客户机希望收到通知的事件，包括FD_QOS。注意为使应用程序正常接收到FD_QOS，QoS事先必须在套接字上设置好，要求接收FD_QOS。一旦收到FD_QOS，客户机便会在WSAWaitForMultipleEvents的一个循环中等待。假如设定的某个事件被触发，循环便会中断。发生了一个事件之后，事件便会在WSAEnumNetworkEvents中被列举出来，同时附带已有的任何错误。

Qostcp.c的大多数部分都在与其他事件打交道，比如FD_READ，FD_WRITE以及。

FD_CLOSE等等，其过程和第8章讲述过的WSAEventSelect示范代码差不多。唯一要注意的是FD_WRITE事件中的一个问题。我们提供的一个命令行选项是等候接收到一条 RSVP PATH消息，再将数据发送出去。假如要传输的数据可能超过网络上的可用带宽，那么这样做便显得颇为必要。AbleToSend函数会调用SIO_CHK_QOS，以判断请求的QoS参数是否在可用带宽的限制范围之内。如答案是肯定的，便可放心大胆地发出数据；否则，应等候确认发出数据的信号。

就客户机的情况来说，我们在此打算接收WSA_QOS_RECEIVERS消息，以初始化对一条RESV消息的接收。可在收到一个 FD_QOS事件后，再来做这些事情。此时，我们可调用SIO_CHK_QOS命令，取得相关的状态信息。该WSA_QOS_RECEIVERS标志可以两种方式返回。第一种方式，可在WSANETWORKEVENTS（网络事件）结构的iErrorCode字段中返回这个标志，它对应于由 FD_QOS_BIT索引的某个数组元素。第二种方式，可在传递给WSAIoctl的缓冲区内返回一个RSVP_STATUS_INFO结构（使用SIO_GET_QOS这个I/O控制命令）。在这个结构中，也可能在其 StatusCode（状态代码）字段中包含了 WSA_QOS_RECEIVERS标志。如决定等待发送标志，那么我们需要检查来自 WSANETWORKEVENTS的错误字段，了解是否返回了一个RSVP_STATUS_INFO结构。假如该标志位已设置，便可放心地发出数据。到此为止，所有工作结束！要想为客户机提供对 QoS的支持，相应的代码是非常直观的。

而换到服务器那一边，我们的这个例子便显得稍微有一些复杂，但也仅仅牵涉到它需要管理零个或多个客户机连接。对负责监听的套接字来说，它和客户机套接字一样，都在一个名为sc的数组中进行控制。其中，数组元素 0对应的是监听套接字，而剩下的元素均分配给可能的客户机连接。全局变量 nConns包含了当前连接的客户机数量。只要完成了一个客户机连接，当时剩下的所有活动套接字都会自动向套接字数组的起始处靠拢。另外，还有一个对应的事件句柄数组。

如用户决定在接受客户机连接之前，先设好 QoS，那么服务器首先会绑定监听套接字，再设置接收QoS。监听套接字上设置的任何QoS参数都会复制给客户机连接（除非服务器正在使用AcceptEx）。监听套接字会对自己进行注册，要求只接收 FD_ACCEPT事件通知。在服务器例程中，剩下的部分是一个大循环，等候在套接字句柄数组上的事件。最开始的时候，数组中唯一的套接字便是监听套接字。而随着越来越多的客户机连接建立起来，还会出现更多的套接字，以及与它们对应的事件。假如由某个事件造成 WSAWaitForMultipleEvents这个等待循环中止，而且在数组元素 0中指出了事件句柄是什么，就表明那个事件是在监听套接字上发生的。如发生这种情况，我们的程序会调用 WSAEnumNetworkEvents，以调查具体发生的是什么事件。假如事件是在某个客户机套接字上发生的，那么代码会调用控制器例程：HandleClientEvents。

在监听套接字上，我们最感兴趣的事件是 FD_ACCEPT。若发生这种事件，便会随一个条件函数，调用WSAAccept。但是请记住，此时传递给条件函数的 QoS参数是不可信任的。假如在Windows 98操作系统中， QoS参数不为空值（NULL），那么必须设置某种形式的 QoS。Windows 2000则不存在这方面的限制； QoS可在任何时候设置。假如用户指出 QoS要在接受调用的过程中设置，那么它会在条件函数中进行。一旦“接受”了客户机套接字，便会创建一个对应的事件句柄，而且为那个套接字注册恰当的事件。

函数HandleClientEvents可用来控制客户机套接字上发生的任何事件。读和写事件用不着多说，唯一要注意的是在发送之前，是否等待预约确认。假如用户要求在发送数据之前，等候预约确认的到达，那么客户机需要等待在一个 FD_QOS事件中返回的 WSA_QOS_RECEIVERS消息。该消息返回之后，除非接收到 FD_QOS，否则数据仍然不能发送出去。这个例子最重要的部分便是如何在套接字以及 FD_QOS控制器中进行QoS的设置。

程序清单12-1 单播TCP示例（Qostcp.c）

```
// Module: Qostcp.c
//
#include <winsock2.h>
#include <windows.h>
#include <qos.h>
#include <qossp.h>

#include "provider.h"
#include "printqos.h"

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

#define QOS_BUFFER_SZ       16000 // Default buffer size for
                                  // SIO_GET_QOS
#define DATA_BUFFER_SZ       2048 // Send/Recv buffer size

#define SET_QOS_NONE            0   // No QOS
#define SET_QOS_BEFORE          1   // Set QOS on listening socket
#define SET_QOS_DURING          2   // Set QOS in conditional accept
#define SET_QOS_AFTER           3   // Set QOS after accept
#define SET_QOS_EVENT           4   // Wait for FD_QOS and then set

#define MAX_CONN                10

int  iSetQos,           // When to set QOS?
     nConns;
BOOL bServer,           // Client or server?
     bWaitToSend,       // Wait to send data until RESV
     bConfirmResv;
char szServerAddr[64]; // Server's address
QOS  clientQos,         // QOS client structure
     serverQos;         // QOS server structure
RSVP_RESERVE_INFO  qosreserve;

//
// Set up some common FLOWSPEC structures
//
const FLOWSPEC flowspec_notraffic = {QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     SERVICETYPE_NOTRAFFIC,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED};
```

```
const FLOWSPEC flowspec_g711 = {8500,
                                680,
                                17000,
                                QOS_NOT_SPECIFIED,
                                QOS_NOT_SPECIFIED,
                                SERVICETYPE_CONTROLLEDLOAD,
                                340,
                                340};

const FLOWSPEC flowspec_guaranteed = {17000,
                                      1260,
                                      34000,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      SERVICETYPE_GUARANTEED,
                                      340,
                                      340};

//
// Function: SetReserveInfo
//
// Description:
//    For receivers, if a confirmation is requested this must be
//    done with an RSVP_RESERVE_INFO structure
//
void SetQosReserveInfo(QOS *lpqos)
{
    qosreserve.ObjectHdr.ObjectType = RSVP_OBJECT_RESERVE_INFO;
    qosreserve.ObjectHdr.ObjectLength = sizeof(RSVP_RESERVE_INFO);
    qosreserve.Style = RSVP_DEFAULT_STYLE;
    qosreserve.ConfirmRequest = bConfirmResv;
    qosreserve.NumPolicyElement  = 0;
    qosreserve.PolicyElementList = NULL;
    qosreserve.FlowDescList = NULL;

    lpqos->ProviderSpecific.buf = (char *)&qosreserve;
    lpqos->ProviderSpecific.len = sizeof(qosreserve);

    return;
}
//
// Function: InitQos
//
// Description:
//    Set up the client and server QOS structures. This is
//    broken out into a separate function so that you can change
//    the requested QOS parameters to see how that affects
//    the application.
//
void InitQos()
{
    clientQos.SendingFlowspec = flowspec_g711;
    clientQos.ReceivingFlowspec =  flowspec_notraffic;
    clientQos.ProviderSpecific.buf = NULL;
    clientQos.ProviderSpecific.len = 0;
```

```
    serverQos.SendingFlowspec = flowspec_notraffic;
    serverQos.ReceivingFlowspec = flowspec_g711;
    serverQos.ProviderSpecific.buf = NULL;
    serverQos.ProviderSpecific.len = 0;
    if (bConfirmResv)
        SetQosReserveInfo(&serverQos);
}

//
// Function: usage
//
// Description:
//    Print out usage information
//
void usage(char *progname)
{
    printf("usage: %s -q:x -s -c:IP\n", progname);
    printf("      -q:[b,d,a,e] When to request QOS\n");
    printf("          b        Set QOS before bind or connect\n");
    printf("          d        Set QOS during accept cond func\n");
    printf("          a        Set QOS after session setup\n");
    printf("          e        Set QOS only upon receipt of FD_QOS\n");
    printf("      -s           Act as server\n");
    printf("      -c:Server-IP Act as client\n");
    printf("      -w           Wait to send until RESV has arrived\n");
    printf("      -r           Confirm reservation request\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse command line arguments and set global variables to
//    indicate how the application should act
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    // Initialize globals to a default value
    //
    iSetQos = SET_QOS_NONE;
    bServer = TRUE;
    bWaitToSend = FALSE;
    bConfirmResv = FALSE;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'q':        // When to set QOS
                    if (tolower(argv[i][3]) == 'b')
                        iSetQos = SET_QOS_BEFORE;
```

```
                else if (tolower(argv[i][3]) == 'd')
                    iSetQos = SET_QOS_DURING;
                else if (tolower(argv[i][3]) == 'a')
                    iSetQos = SET_QOS_AFTER;
                else if (tolower(argv[i][3]) == 'e')
                    iSetQos = SET_QOS_EVENT;
                else
                    usage(argv[0]);
                break;
            case 's':        // Server
                printf("Server flag set!\n");
                bServer = TRUE;
                break;
            case 'c':        // Client
                printf("Client flag set!\n");
                bServer = FALSE;
                if (strlen(argv[i]) > 3)
                    strcpy(szServerAddr, &argv[i][3]);
                else
                    usage(argv[0]);
                break;
            case 'w':       // Wait to send data until
                            // RESV has arrived
                bWaitToSend = TRUE;
                break;
            case 'r':
                bConfirmResv = TRUE;
                break;
            default:
                usage(argv[0]);
                break;
        }
    }
  }
  return;
}

//
// Function: AbleToSend
//
// Description:
//    Checks to see whether data can be sent on the socket before
//    any RESV messages have arrived. This function checks to see whether
//    the best-effort level currently available on the network is
//    sufficient for the QOS levels that were set on the socket.
//
BOOL AbleToSend(SOCKET s)
{
    int     ret;
    DWORD   dwCode = ALLOWED_TO_SEND_DATA,
            dwValue,
            dwBytes;

    ret = WSAIoctl(s, SIO_CHK_QOS, &dwCode, sizeof(dwCode),
                &dwValue, sizeof(dwValue), &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
```

```
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return FALSE;
    }
    return (BOOL)dwValue;
}

//
// Function: ChkForQosStatus
//
// Description:
//    Check for the presence of an RSVP_STATUS_INFO object and
//    determine whether the supplied flags are present in that
//    object
//
DWORD ChkForQosStatus(QOS *lpqos, DWORD dwFlags)
{
    QOS_OBJECT_HDR   *objhdr = NULL;
    RSVP_STATUS_INFO *status = NULL;
    char             *bufptr = NULL;
    BOOL              bDone = FALSE;
    DWORD             objcount = 0;

    if (lpqos->ProviderSpecific.len == 0)
        return 0;

    bufptr = lpqos->ProviderSpecific.buf;
    objhdr = (QOS_OBJECT_HDR *)bufptr;


    while (!bDone)
    {
        if (objhdr->ObjectType == RSVP_OBJECT_STATUS_INFO)
        {
            status = (RSVP_STATUS_INFO *)objhdr;
            if (status->StatusCode & dwFlags)
                return 1;
        }
        else if (objhdr->ObjectType == QOS_OBJECT_END_OF_LIST)
            bDone = TRUE;

        bufptr += objhdr->ObjectLength;
        objcount += objhdr->ObjectLength;
        objhdr = (QOS_OBJECT_HDR *)bufptr;

        if (objcount >= lpqos->ProviderSpecific.len)
            bDone = TRUE;
    }
    return 0;
}

//
// Function: HandleClientEvents
//
// Description:
```

```
//    This function is called by the Server function to handle
//    events that occurred on client SOCKET handles. The socket
//    array is passed in along with the event array and the index
//    of the client who received the signal. Within the function,
//    the event is decoded and the appropriate action occurs.
//
void HandleClientEvents(SOCKET socks[], HANDLE events[], int index)
{
    WSANETWORKEVENTS  ne;
    char              databuf[4096];
    WSABUF            wbuf;
    DWORD             dwBytesRecv,
                      dwFlags;
    int               ret,
                      i;

    // Enumerate the network events that occurred
    //
    ret = WSAEnumNetworkEvents(socks[index], events[index], &ne);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAEnumNetworkEvents() failed: %d\n",
            WSAGetLastError());
        return;
    }
    // Data to be read
    //
    if ((ne.lNetworkEvents & FD_READ) == FD_READ)
    {
        wbuf.buf = databuf;
        wbuf.len = 4096;

        if (ne.iErrorCode[FD_READ_BIT])
            printf("FD_READ error: %d\n",
                ne.iErrorCode[FD_READ_BIT]);
        else
            printf("FD_READ\n");

        dwFlags = 0;
        ret = WSARecv(socks[index], &wbuf, 1, &dwBytesRecv,
            &dwFlags, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSARecv() failed: %d\n", WSAGetLastError());
            return;
        }
        wbuf.len = dwBytesRecv;

        printf("Read: %d bytes\n", dwBytesRecv);
    }
    // Able to write data; nothing to do here
    //
    if ((ne.lNetworkEvents & FD_WRITE) == FD_WRITE)
    {
        if (ne.iErrorCode[FD_WRITE_BIT])
```

```
            printf("FD_WRITE error: %d\n",
                ne.iErrorCode[FD_WRITE_BIT]);
        else
            printf("FD_WRITE\n");
    }
    // The client closed the connection. Close the socket on our
    // end and clean up the data structures.
    //
    if ((ne.lNetworkEvents & FD_CLOSE) == FD_CLOSE)
    {
        if (ne.iErrorCode[FD_CLOSE_BIT])
            printf("FD_CLOSE error: %d\n",
                ne.iErrorCode[FD_CLOSE_BIT]);
        else
            printf("FD_CLOSE ...\n");
        closesocket(socks[index]);
        WSACloseEvent(events[index]);

        socks[index] = INVALID_SOCKET;
        //
        // Remote the client socket entry from the array and
        // compact the remaining clients to the beginning of the
        // array
        //
        for(i = index; i < MAX_CONN - 1; i++)
            socks[i] = socks[i + 1];
        nConns--;
    }
    // Received an FD_QOS event. This could mean several things.
    //
    if ((ne.lNetworkEvents & FD_QOS) == FD_QOS)
    {
        char         buf[QOS_BUFFER_SZ];
        QOS         *lpqos = NULL;
        DWORD        dwBytes;

        if (ne.iErrorCode[FD_QOS_BIT])
            printf("FD_QOS error: %d\n",
                ne.iErrorCode[FD_QOS_BIT]);
        else
            printf("FD_QOS\n");

        lpqos = (QOS *)buf;
        lpqos->ProviderSpecific.buf = &buf[sizeof(QOS)];
        lpqos->ProviderSpecific.len = sizeof(buf) - sizeof(QOS);

        ret = WSAIoctl(socks[index], SIO_GET_QOS, NULL, 0,
                buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
                WSAGetLastError());
            return;
        }
        PrintQos(lpqos);
        //
        // See whether we're set for receiving FD_QOS events only.
```

```
        // If so, we need to invoke QOS on the connection
        // now; otherwise, client will never receive a RESV message.
        //
        if (iSetQos == SET_QOS_EVENT)
        {
            lpqos->ReceivingFlowspec.ServiceType =
                serverQos.ReceivingFlowspec.ServiceType;

            ret = WSAIoctl(socks[index], SIO_SET_QOS, lpqos,
                dwBytes, NULL, 0, &dwBytes, NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
                    WSAGetLastError());
                return;
            }
            //
            // Change iSetQos so we don't set QOS again if we
            // receive another FD_QOS event
            //
            iSetQos = SET_QOS_BEFORE;
        }
    }
    return;
}

//
// Function: SrvCondAccept
//
// Description:
//    This is the conditional function for WSAAccept. The QOS service
//    provider is limited in that the QOS values passed into here are
//    unreliable, so the option SET_QOS_DURING is useless unless we call
//    SIO_SET_QOS with our own values (as opposed to the values the client
//    is requesting since those values are supposed to be returned in
//    lpSQOS). Note that on Windows 98, if lpSQOS is not NULL you have to
//    set some QOS values (with SIO_SET_QOS) in the conditional
//    function; otherwise, WSAAccept will fail.
//
int CALLBACK SrvCondAccept(LPWSABUF lpCallerId,
    LPWSABUF lpCallerdata, LPQOS lpSQOS, LPQOS lpGQOS,
    LPWSABUF lpCalleeId, LPWSABUF lpCalleeData, GROUP *g,
    DWORD dwCallbackData)
{
    DWORD       dwBytes = 0;
    SOCKET      s = (SOCKET)dwCallbackData;
    SOCKADDR_IN client;
    int         ret;

    if (nConns == MAX_CONN)
        return CF_REJECT;

    memcpy(&client, lpCallerId->buf, lpCallerId->len);
    printf("Client request: %s\n", inet_ntoa(client.sin_addr));

    if (iSetQos == SET_QOS_EVENT)
```

```
    {
        printf("Setting for event!\n");
        serverQos.SendingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;
        serverQos.ReceivingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;

        ret = WSAIoctl(s, SIO_SET_QOS, &serverQos,
            sizeof(serverQos), NULL, 0, &dwBytes, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAIoctl() failed: %d\n",
                WSAGetLastError());
            return CF_REJECT;
        }
    }
    return CF_ACCEPT;
}

//
// Function: Server
//
// Description:
//    This server routine handles incoming client connections.
//    First it sets up the listening socket, sets QOS when
//    appropriate, and waits for incoming clients and events.
//
void Server(SOCKET s)
{
SOCKET          sc[MAX_CONN + 1];
WSAEVENT        hAllEvents[MAX_CONN+1];
SOCKADDR_IN     local,
                client;
int             clientsz,
                ret,
                i;
DWORD           dwBytesRet;
WSANETWORKEVENTS ne;

// Initialize the arrays to invalid values
//
for(i = 0; i < MAX_CONN+1; i++)
{
    hAllEvents[i] = WSA_INVALID_EVENT;
    sc[i] = INVALID_SOCKET;
}
// Array index 0 will be our listening socket
//
hAllEvents[0] = WSACreateEvent();
sc[0]         = s;
nConns        = 0;

local.sin_family = AF_INET;
local.sin_port = htons(5150);
local.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;
}
listen(s, 7);

if (iSetQos == SET_QOS_BEFORE)
{
    ret = WSAIoctl(sc[0], SIO_SET_QOS, &serverQos,
        sizeof(serverQos), NULL, 0, &dwBytesRet, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Set QOS on listening socket:\n");
    PrintQos(&serverQos);
}
if (WSAEventSelect(sc[0], hAllEvents[0], FD_ACCEPT) ==
    SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;
}

while (1)
{
    ret = WSAWaitForMultipleEvents(nConns+1, hAllEvents, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
        printf("WSAWaitForMultipleObject() failed: %d\n",
            WSAGetLastError());
        return;
    }
    if ((i = ret - WSA_WAIT_EVENT_0) > 0)  // Client network event
        HandleClientEvents(sc, hAllEvents, i);
    else
    {
        ret = WSAEnumNetworkEvents(sc[0], hAllEvents[0],
            &ne);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAEnumNetworkevents() failed: %d\n",
                WSAGetLastError());
            return;
        }
        if ((ne.lNetworkEvents & FD_ACCEPT) == FD_ACCEPT)
        {
            if (ne.iErrorCode[FD_ACCEPT_BIT])
                printf("FD_ACCEPT error: %d\n",
                    ne.iErrorCode[FD_ACCEPT_BIT]);
            else
```

```
                printf("FD_ACCEPT\n");

            clientsz = sizeof(client);
            sc[++nConns] = WSAAccept(s, (SOCKADDR *)&client,
                &clientsz, SrvCondAccept, sc[nConns]);
            if (sc[nConns] == SOCKET_ERROR)
            {
                printf("WSAAccept() failed: %d\n",
                    WSAGetLastError());
                nConns--;
                return;
            }
                hAllEvents[nConns] = WSACreateEvent();

                Sleep(10000);
                if (iSetQos == SET_QOS_AFTER)
                {
                    ret = WSAIoctl(sc[nConns], SIO_SET_QOS,
                        &serverQos, sizeof(serverQos), NULL, 0,
                        &dwBytesRet, NULL, NULL);
                    if (ret == SOCKET_ERROR)
                    {
                        printf("WSAIoctl() failed: %d\n",
                            WSAGetLastError());
                        return;
                    }
                }
                ret = WSAEventSelect(sc[nConns],
                    hAllEvents[nConns], FD_READ | FD_WRITE |
                    FD_CLOSE | FD_QOS);
                if (ret == SOCKET_ERROR)
                {
                    printf("WSAEventSelect() failed: %d\n",
                        WSAGetLastError());
                    return;
                }
            }
            if (ne.lNetworkEvents & FD_CLOSE)
                printf("FD_CLOSEn");
            if (ne.lNetworkEvents & FD_READ)
                printf("FD_READn");
            if (ne.lNetworkEvents & FD_WRITE)
                printf("FD_WRITEn");
            if (ne.lNetworkEvents & FD_QOS)
                printf("FD_QOS\n");
        }
    }
    return;
}

//
// Function: Client
//
// Description:
//    The client routine initiates the connection, sets QOS when
//    appropriate, and handles incoming events.
```

```
//
void Client(SOCKET s)
{
    SOCKADDR_IN  server,
                 local;
WSABUF           wbuf;
DWORD            dwBytes,
                 dwBytesSent,
                 dwBytesRecv,
                 dwFlags;
HANDLE           hEvent;
int              ret, i;
char             databuf[DATA_BUFFER_SZ];
QOS             *lpqos;
WSANETWORKEVENTS ne;

hEvent = WSACreateEvent();
if (hEvent == NULL)
{
    printf("WSACreateEvent() failed: %d\n", WSAGetLastError());
    return;
}

lpqos = NULL;
if (iSetQos == SET_QOS_BEFORE)
{
    local.sin_family = AF_INET;
    local.sin_port = htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(s, (SOCKADDR *)&local, sizeof(local)) ==
        SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return;
    }
    ret = WSAIoctl(s, SIO_SET_QOS, &clientQos,
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
}
else if (iSetQos == SET_QOS_DURING)
    lpqos = &clientQos;
else if (iSetQos == SET_QOS_EVENT)
{
    clientQos.SendingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;
    clientQos.ReceivingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;

    ret = WSAIoctl(s, SIO_SET_QOS, &clientQos,
```

```
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return;
    }
}

server.sin_family = AF_INET;
server.sin_port = htons(5150);
server.sin_addr.s_addr = inet_addr(szServerAddr);

printf("Connecting to: %s\n", inet_ntoa(server.sin_addr));

ret = WSAConnect(s, (SOCKADDR *)&server, sizeof(server),
    NULL, NULL, lpqos, NULL);
if (ret == SOCKET_ERROR)
{
    printf("WSAConnect() failed: %d\n", WSAGetLastError());
    return;
}


ret = WSAEventSelect(s, hEvent, FD_READ | FD_WRITE |
    FD_CLOSE | FD_QOS);
if (ret == SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;
}

wbuf.buf = databuf;
wbuf.len = DATA_BUFFER_SZ;

memset(databuf, '#', DATA_BUFFER_SZ);
databuf[DATA_BUFFER_SZ-1] = 0;

while (1)
{
    ret = WSAWaitForMultipleEvents(1, &hEvent, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
        printf("WSAWaitForMultipleEvents() failed: %d\n",
        WSAGetLastError());
    return;
}

ret = WSAEnumNetworkEvents(s, hEvent, &ne);
if (ret == SOCKET_ERROR)
{
    printf("WSAEnumNetworkEvents() failed: %d\n",
        WSAGetLastError());
    return;
}
if (ne.lNetworkEvents & FD_READ)
```

```
{
    if (ne.iErrorCode[FD_READ_BIT])
        printf("FD_READ error: %d\n",
            ne.iErrorCode[FD_READ_BIT]);
    else
        printf("FD_READ\n");

    wbuf.len = 4096;
    dwFlags = 0;
    ret = WSARecv(s, &wbuf, 1, &dwBytesRecv, &dwFlags,
        NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSARecv() failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Read: %d bytes\n", dwBytesRecv);

    wbuf.len = dwBytesRecv;
    ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0, NULL,
        NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSASend() failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Sent: %d bytes\n", dwBytesSent);
}
if (ne.lNetworkEvents & FD_WRITE)
{
    if (ne.iErrorCode[FD_WRITE_BIT])
        printf("FD_WRITE error: %d\n",
            ne.iErrorCode[FD_WRITE_BIT]);
    else
        printf("FD_WRITE\n");

    if (!bWaitToSend)
    {
        wbuf.buf = databuf;
        wbuf.len = DATA_BUFFER_SZ;
        //
        // If the network can't support the bandwidth,
        // don't send
        //
        if (!AbleToSend(s))
        {
            printf("Network is unable to provide "
                "sufficient best-effort bandwidth\n");
            printf("before the reservation "
                "request is approved\n");
        }

        for(i = 0; i < 1; i++)
        {
```

```
            ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: %d\n",
                    WSAGetLastError());
                return;
            }
            printf("Sent: %d bytes\n", dwBytesSent);
        }
    }
}
if (ne.lNetworkEvents & FD_CLOSE)
{
    if (ne.iErrorCode[FD_CLOSE_BIT])
        printf("FD_CLOSE error: %d\n",
            ne.iErrorCode[FD_CLOSE_BIT]);
    else
        printf("FD_CLOSE ...\n");
    closesocket(s);
    WSACloseEvent(hEvent);
    return;
}
if (ne.lNetworkEvents & FD_QOS)
{
    char        buf[QOS_BUFFER_SZ];
    QOS        *lpqos = NULL;
DWORD       dwBytes;
BOOL        bRecvRESV = FALSE;

if (ne.iErrorCode[FD_QOS_BIT])
{
    printf("FD_QOS error: %d\n",
        ne.iErrorCode[FD_QOS_BIT]);
    if (ne.iErrorCode[FD_QOS_BIT] == WSA_QOS_RECEIVERS)
        bRecvRESV = TRUE;
}
else
    printf("FD_QOS\n");

lpqos = (QOS *)buf;
ret = WSAIoctl(s, SIO_GET_QOS, NULL, 0,
        buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
if (ret == SOCKET_ERROR)
{
    printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
        WSAGetLastError());
    return;
}
PrintQos(lpqos);
//
// Check to see whether a status object is returned
// in the QOS structure that might also contain the
// WSA_QOS_RECEIVERS flag
//
if (ChkForQosStatus(lpqos, WSA_QOS_RECEIVERS))
```

```
    bRecvRESV = TRUE;

if (iSetQos == SET_QOS_EVENT)
{
    lpqos->SendingFlowspec.ServiceType =
        clientQos.SendingFlowspec.ServiceType;
    ret = WSAIoctl(s, SIO_SET_QOS, lpqos, dwBytes,
            NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
    //
    // Change iSetQos so that we don't set QOS again if we
    // receive another FD_QOS event
    //
    iSetQos = SET_QOS_BEFORE;
}
            if (bWaitToSend && bRecvRESV)
            {
                wbuf.buf = databuf;
                wbuf.len = DATA_BUFFER_SZ;

                for(i = 0; i < 1; i++)
                {
                    ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                        NULL, NULL);
                    if (ret == SOCKET_ERROR)
                    {
                        printf("WSASend() failed: %d\n",
                            WSAGetLastError());
                        return;
                    }
                    printf("Sent: %d bytes\n", dwBytesSent);
                }
            }
        }
    }
    return;
}

//
// Function: main
//
// Description:
//    Initialize Winsock, parse command line arguments, create
//    a QOS TCP socket, and call the appropriate handler
//    routine depending on the arguments supplied
//
int main(int argc, char **argv)
{
    WSADATA           wsd;
    WSAPROTOCOL_INFO *pinfo = NULL;
```

```
    SOCKET            s;

    // Parse the command line
    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Unable to load Winsock: %d\n", GetLastError());
        return -1;
    }
    pinfo = FindProtocolInfo(AF_INET, SOCK_STREAM, IPPROTO_TCP,
        XP1_QOS_SUPPORTED);
    if (!pinfo)
    {
        printf("unable to find suitable provider!\n");
        return -1;
    }
    printf("Provider returned: %s\n", pinfo->szProtocol);

    s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
        FROM_PROTOCOL_INFO, pinfo, 0, WSA_FLAG_OVERLAPPED);
    if (s == INVALID_SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    InitQos();

    if (bServer)
        Server(s);
    else
        Client(s);

    closesocket(s);
    WSACleanup();
    return 0;
}
```

表12-8 Qostcp.c命令行参数

参 数	含 义
-q:[b,d,a,e]	指明在之前（b）、之间（d）、之后（a）或在设置前等待一个FD_QOS事件（e）设置QoS
-s	以服务器的身份运行
-c: 服务器IP地址	以客户机的身份运行，同时与指定IP地址的服务器建立连接
-w	收到一条RESV消息后，再开始发送数据
-r	设置一旦预约被确认便接收通知的选项

12.5.2 单播UDP

采用单播UDP通信，可比TCP多出几种可能性。配套光盘上的UDP示例名字叫作Qosudp.c。它兼具发送者及接收者的双重身份。如作为发送者使用，那么有两种方法可指示 QoS服务提供者将数据发到哪里。请记住，要想建立一个RSVP会话，那么对方的地址是必须提前知道的。要想做到这一点，可在指定QOS_DESTADDR（目标地址）对象的前提下，调用 WSAConnect

或SIO_SET_QOS命令。在我们提供的这个单播UDP示例中，还专门提供了一个参数，用于指示何时对QoS进行设置。假如用户要求QoS在绑定或建立连接之前设置，那么需要在WSAConnect调用中指定QoS参数。假如用户决定在完成了会话的建立之后再来设置QoS，那么在调用WSAConnect的时候，便不必同时设置QoS参数，而且以后调用SIO_SET_QOS的时候，不必提供QOS_DESTADDR对象。最后，假若用户要求只有在接收到一个FD_QOS事件通知的前提下，才对QoS进行设置，那么尽管需要用一个QOS_DESTADDR对象来调用SIO_SET_QOS，但SERVICE_QOS_NO_SIGNALING这个标志需要与ServiceType这个FLOWSPEC字段通过“或”运算合并到一起。

而作为接收端使用时，该程序的选项要少得多。用于指示何时设置QoS的标志在这里全都派不上用场。可在准备接收数据之前设置QoS，亦可要求接收者等待一个FD_QOS事件的发生。这是由于UDP并不接收任何连接请求，即在接受函数的调用过程中，或在会话建立起来之后，并不存在什么QoS设置。作为一个接收者，它亦可选择指定一个不同的过滤器样式，比如固定过滤器，或者“共享显式”过滤器等等。假如指定了一个不同的过滤器，那么必须使用-r:IP选项，明确设定一个IP地址。在RSVP_RESERVE_INFO（预约信息）结构中，SetQosReceivers函数需要用一个RSVP_FILTERSPEC（过滤器规格）结构的内容向其中填充。后者定义了发送者的IP地址。设置过滤器时，要特别留意的一个问题是必须指定发送者的端口号。也就是说，作为接收者，它在同发送者那一方建立绑定关系之前，必须先知道每个发送者的IP地址。

请注意，接收者有时亦可使用WSAConnect将发送者的IP地址同套接字联系到一起。然而，由于UDP接收者可能同时指定多个不同的过滤器样式，而且可能存在多个发送者，所以在此不能使用WSAConnect。要记住的是，假如用WSAConnect关联了一个端点的IP地址，那么发送与接收操作只能针对那个端点进行，同时QoS也会同它联系到一起。

将这个单播UDP的例子同前面的TCP示例比较，会发现两者其实很相似。唯一的例外便是它们采用了不同的方式在套接字上设置QoS。UDP程序要求发送者指定接收者的IP地址，以便调用RSVP，而QoS针对这一目的提供了两种方法。而究其本质，TCP程序默认是在连接调用过程中做到这一点的。对两个程序来说，它们的事件循环几乎完全相同。但请不要忘记UDP应用程序的一个重大“陷阱”：对一个套接字来说，假如使用的不是WSAConnect，那么用SIO_SET_QOS在它上面设置任何QoS选项之前，套接字本身必须先建好“本地绑定”关系！同INADDR_ANY和端口0建立绑定关系是完全合法的；当然，亦可指定一个具体的IP和端口。而假若使用的是WSAConnect调用，那么这种绑定的建立是“隐式”或自动进行的；换言之，毋需我们去干涉。因此，假如在那个时候设置QoS，那么事前并不一定要进行“显式”或明确绑定。

12.5.3 多播UDP

我们提供的最后一个例子是多播QoS，在本书配套光盘上对应的文件名是Qosmcast.c。该程序的核心函数便是大家或许早已熟悉的WSAJoinLeaf。如前一章所述，应用程序必须先通过对该函数的调用，来加入一个多播组。程序加入多播组后，接下来的事情便是传递QoS参数。这儿的多播例子采用的参数与单播UDP的例子完全相同。我们可自行决定何时在套接字上对QoS进行设置。如选择在接受条件函数的执行期间设置QoS，那么QoS会传递进入

WSAJoinLeaf调用之中。否则的话，可随SIO_SET_QOS一道，通过对WSAIoctl的调用，来完成QoS的设置。

对于接收者来说，它允许用户自行设定过滤器的样式：固定过滤器或“共享显式”过滤器。请大家记住，多播UDP在默认情况下采用的是一个通配样式。若指定了一个不同样式的过滤器类型，那么这种改变只会应用于接收者。而且假如对这两种过滤器类型之一感兴趣，可用一个“-r:发送者IP”命令行选项，指定每个发送者的地址。用户可用-f选项来决定采用什么过滤器，若设为se，表明使用“共享显式”过滤器；而若设为ff，表明使用固定过滤器。

对发送者和接收者来说，都可用-m选项来指定要加入的多播组。要注意的是，可重复使用这一选项，从而根据自己的需要，加入任意数量的多播组。-s选项指出当前的程序应作为“发送者”使用。-w选项用于指示发送者，令其在等到一个WSA_QOS_RECEIVERS通知之后，再开始数据的发送。最后，使用-q选项，可指出何时对QoS进行设置。无论什么时候进行QoS的设置，套接字都会与端口5150建立本地绑定关系。在实际应用中，你可选择任意端口；或干脆设为0，让程序为自己自动选择一个端口。然而，假如接收者打算设置固定或共享显式过滤器，那么它必须提供发送者的IP及端口。为使问题简化，我们在此使用了一个固定端口。和单播UDP的例子不同，这儿的接收者不要求事先建立与端口的本地绑定关系，再来设置QoS。原因在于，我们调用WSAJoinLeaf的时候，其实已自动绑定了套接字——如果它尚未绑定的话。大家或许会提出的另一个问题是：能不能用套接字选项命令IP_ADD_MEMBERSHIP（加入组）和IP_DROP_MEMBERSHIP（脱离组）来代替WSAJoinLeaf函数？答案是否定的，你不可这样做！假如使用那些命令，那么请求的QoS参数不会应用于套接字。

对于不同时候设置QoS会发生什么事情，我们在此不打算作深一步的探讨，因为这与单播UDP的情况非常相似。经过这一系列的学习，大家现在应已非常熟悉如何建立RSVP会话，以及生成PATH和RESV消息需要哪些必要条件！

12.6 ATM和QoS

如早先所述，QoS是ATM网络一种“与生俱来”能力。无论Windows 2000还是Windows 98（安装SP1的服务）都支持来自Winsock的、固有的ATM编程技术，这在第6章已进行了非常全面的论述。QoS是ATM网络默认提供的一种功能；换言之，在ATM网络上、原先通过IP实现QoS必需的一系列网络、应用程序及策略组件均成了多余的东西。这些组件包括许可控制服务（Admission Control Service）以及RSVP协议。相反，ATM交换机会自动进行带宽的分配，并可自动预防带宽的浪费。

除我们已经指出的这些差异之外，Winsock API函数与ATM QoS配合使用时，其行为也与通过IP使用QoS时存在一些区别。第一个重要的区别是对于QoS带宽请求来说，对它的控制是作为连接请求的一部分来进行的。这和通过IP实现QoS时是不同的，后者要求RSVP会话的建立与正式的连接分离开。此外，假如在ATM环境中拒绝了一个带宽请求，那么连接就会失败。

这样便导致了第二项区别：两个ATM提供者均是“面向连接”的。因此，我们不必操心如何为一个无连接的套接字设置QoS等级，再来指定要与之通信的端点。另一个主要区别在于，对一个连接而言，只需由一方来设置QoS参数就可以了。也就是说，假如客户机希望对一个连接的QoS进行设定，那么在传给WSAConnect的QoS结构中，无论发送还是接收FLOWSPEC结构都会被设定。随后，这些值可立即应用于此次连接。而通过IP实现对QoS的

支持时，发送者必须先请求一个特定的 QoS等级，然后由接收者负责实际的预约。除此以外，还有可能需要用WSAIoctl及SIO_SET_QOS来设置监听套接字上的QoS。这些值可应用于任何进入的连接。这同时意味着，在连接设置期间， QoS必须设好！注意不能在一个已经建好的连接上设置QoS。

这样便引入了我们的最后一个论点：一旦为某个连接设好了 QoS，便不能通过调用WSAIoctl以及SIO_SET_QOS，进行QoS的重新协商。若在一个连接上设好了 QoS，它的作用会一直持续下去，直到连接关闭。

注意RSVP在ATM的环境中已经不复存在，而且也不会发生任何“传信”事件。换言之，表12.6总结的任何状态标志都不会生成。 QoS是在建立连接的过程中设置好的，除非这个连接关闭，否则永远都不会产生任何额外的通知或事件。

12.7 小结

QoS使应用程序如虎添翼，可放心大胆地享用一个稳定的网络服务等级，从而保障视频及音频信号等流畅地播放。建立 QoS连接显得多少有些麻烦，但千万不要畏难。最需要掌握的便是怎样以及何时生成RSVP消息；再以此为依据，相应地编写程序代码。