Chapter 6: Multimedia Networking

In this chapter we consider networking applications whose data contains audio and video content. We refer to these applications as multimedia networking applications. Multimedia networking applications are typically highly sensitive to delay but are loss tolerant. After surveying and classifying different types of multimedia applications, we examine their deployment in a best-effort network, such as today's Internet. We explore how a combination of client buffers, packet sequence numbers and timestamps can greatly alleviate the effects of network induced delay and jitter. We also study how forward error correction and packet interleaving can improve user perceived performance when a fraction of packets are lost or significantly delayed. We examine the RTP and H.323 protocols for real-time telephony and video conferencing in the Internet. We then look at how the Internet can evolve to provide improved QoS (Quality of Service) to its applications. We identify several principles for providing QoS, including packet marking and classification, isolation of packet flows, efficient use of resources, and call admission. We survey several scheduling and policing mechanisms that provide the foundation of a QoS network architecture. We then discuss new Internet standards for QoS, including the Integrated Services and the Differentiated Services standards.

Online Book

6.1: Multimedia Networking Applications

Having completed our journey down the protocol stack in Chapter 5, we now have a strong grounding in the principles and practice of computer networking. This foundation will serve us well as we turn in this chapter to a topic that cuts across many layers of the protocol stack: multimedia networking.

The last few years have witnessed an explosive growth in the development and deployment of networked applications that transmit and receive audio and video content over the Internet. New **multimedia networking applications** (also referred to as **continuous media applications**)--entertainment video, IP telephony, Internet radio, multimedia WWW sites, teleconferencing, interactive games, virtual worlds, distance learning, and much more--seem to be announced daily. The service requirements of these applications differ significantly from those of traditional data-oriented applications such as the Web text/image, e-mail, FTP, and DNS applications that we examined in Chapter 2. In particular, multimedia applications are highly sensitive to end-to-end delay and delay variation, but can tolerate occasional loss of data. These fundamentally different service requirements suggest that a network architecture that has been designed primarily for data communication may not be well suited for supporting multimedia applications. Indeed, we'll see in this chapter that a

number of efforts are currently underway to extend the Internet architecture to provide explicit support for the service requirements of these new multimedia applications.

We'll begin our study of multimedia networking in a top-down manner (of course!) by describing several multimedia applications and their service requirements in Section 6.1. In Section 6.2, we look at how today's Web servers stream audio and video over the Internet to clients. In Section 6.3 we examine a specific multimedia application, Internet telephony, in detail, with the goal of illustrating some of the difficulties encountered (and solutions developed) when applications must necessarily use today's best-effort Internet transport service. In Section 6.4 we describe the RTP protocol, an emerging application-layer standard for framing and controlling the transmission of multimedia data.

In the second half of this chapter we turn our attention toward the future and towards the lower layers of the protocol stack, where we examine recent advances aimed at developing a next-generation network architecture that provides explicit support for the service requirements of multimedia applications. We'll see that rather than providing only a single best-effort service class, these future architectures will also include service classes that provide quality-of-service (QoS) performance guarantees to multimedia applications. In Section 6.5 we identify key principles that will lie at the foundation of this next generation architecture. In Section 6.6 we examine specific packet-level scheduling and policing mechanisms that will be important pieces of this future architecture. Sections 6.7 and 6.9 introduce the so-called Intserv and Diffserv architectures, emerging Internet standards for the next generation QoS-sensitive Internet. In Section 6.8, we examine RSVP, a signaling protocol that plays a key role in both Intserv and Diffserv.

In our discussion in Chapter 2 of application service requirements, we identified a number of axes along which these requirements can be classified. Two of these characteristics--timing considerations and tolerance to data loss--are particularly important for networked multimedia applications. Multimedia applications are highly **delay sensitive.** We will see shortly that packets that incur a sender-to-receiver delay of more than a few hundred milliseconds (for Internet telephony) to a few seconds (for streaming of stored multimedia) are essentially useless. On the other hand, multimedia networking applications are also typically **loss tolerant**--occasional loss only causes occasional glitches in the audio/video playback, and these losses can be often partially or fully concealed. These service requirements are clearly different from those of elastic applications such as Web text/image, e-mail, FTP, and Telnet. For these applications, long delays are annoying but not particularly harmful, and the integrity of transferred data is of paramount importance.

6.1.1: Examples of Multimedia Applications

The Internet carries a large variety of exciting multimedia applications. In the following sections, we consider three broad classes of multimedia applications.

Streaming, Stored Audio and Video

In this class of applications, clients request on-demand compressed audio or video files that are stored on servers. Stored audio files might contain audio from a professor's lecture (you are urged to visit the Web site for this book to try this out!), rock songs, symphonies, archives of famous radio broadcasts, or archived historical recordings. Stored video files might contain video of a professor's lecture, full-length movies, prerecorded television shows, documentaries, video archives of historical events, cartoons, or music video clips. There are three key distinguishing features of this class of applications.

- *Stored media.* The multimedia content has been prerecorded and is stored at the server. As a result, a user may pause, rewind, fast-forward or index through the multimedia content. The time from when a client makes such a request until the action manifests itself at the client should be on the order of 1 to 10 seconds for acceptable responsiveness.

- *Streaming.* In most stored audio/video applications, a client begins playout of the audio/video a few seconds after it begins receiving the file from the server. This means that the client will be playing out audio/video from one location in the file while it is receiving later parts of the file from the server. This technique, known as **streaming**, avoids having to download the entire file (and incurring a potentially long delay) before beginning playout. There are many streaming multimedia products, including RealPlayer from RealNetworks [RealNetworks 2000] and Microsoft's Windows Media [Microsoft Windows Media 2000]. There are also applications such as Napster [Napster 2000], however, that require an entire audio file to be downloaded before playout begins.

- *Continuous playout.* Once playout of the multimedia begins, it should proceed according to the original timing of the recording. This places critical delay constraints on data delivery. Data must be received from the server in time for its playout at the client; otherwise, it is considered useless. In Section 6.3, we'll consider the consequences of this requirement in detail. The end-to-end delay constraints for streaming, stored media are typically less stringent than those for live, interactive applications such as Internet telephony and video conferencing (see below).

Case History

Real Networks: Bringing Audio to the Internet Foreground

RealNetworks, pioneers in streaming audio and video products, was the first company to bring audio to the Internet mainstream. The company began under the name Progressive Networks in 1995. Its initial product--the RealAudio system-- included an audio encoder, an audio server, and an audio player. The RealAudio system enabled users to browse, select and play back audio content on demand, as easily as using a standard video cassette player/recorder. It quickly became popular for providers of entertainment, information, and news content to deliver audio on demand services that can be accessed and played back immediately. In early 1997, RealNetworks expanded its product line to include video as well as audio. RealNetwork products currently incorporate RTP and RTSP protocols.

Over the past few years, RealNetworks has seen tough competition from Microsoft (which also has minority ownership of RealNetworks). In 1997 Microsoft began to market its own streaming media products, essentially setting the stage for a "media-player war," similar to the browser war between Netscape and Microsoft. But RealNetworks and Microsoft have diverged on some of the underlying technology choices in their players. Waging the tug of war in the marketplace and in Internet standards groups, both companies are seeking to have their own formats and protocols become the standard for the Internet.

Streaming of Live Audio and Video

This class of application is similar to traditional broadcast radio and television, except that transmission takes place over the Internet. These applications allow a user to receive a *live* radio or television transmission emitted from any corner of the world. (For example, one of the authors of this book often listens to his favorite Philadelphia radio stations from his home in France. The other author regularly listened to live broadcasts of his university's beloved basketball team while he was living in France for a year.) See [Yahoo!Broadcast 2000] and [NetRadio 2000] for Internet radio station guides.

Since streaming live audio/video is not stored, a client cannot fast forward through the media. However, with local storage of received data, other interactive operations such as pausing and rewinding though live multimedia transmissions are possible in some applications. Live, broadcast-like applications often have many clients who are receiving the same audio/video program. Distribution of live audio/ video to many receivers can be efficiently accomplished using the multicasting techniques we studied in Section 4.8. At the time of the writing of this book, however, this type of distribution is more often accomplished through multiple separate unicast streams. As with streaming stored multimedia, continuous playout is required, although the timing constraints are less stringent than for live interactive applications. Delays of up to tens of seconds from when the user requests the delivery/playout of a live transmission to when playout begins can be tolerated.

Case History

Voice over the Internet

Given the worldwide popularity of the telephone system, since the late 1980s many Internet visionaries have repeatedly predicted that the next Internet killer application would be some sort of voice application. These predictions were accompanied with Internet

telephony research and product development. For example, researchers created Internet phone prototypes in the 1980s, years before the Web was popularized. And numerous startups produced PC-to-PC Internet phone products throughout the 1990s. But none of these prototypes or products really caught on with mainstream Internet users (even though some were bundled with popular browsers). Not until 1999 did voice communication begin to get popularized in the Internet.

Three classes of voice communication applications began to see significant usage in the late 1990s. The first class is the PC-to-phone applications, which allow an Internet user with an Internet connection and a microphone to call any ordinary telephone. Two companies active in the PC-to-phone space are Net2Phone [Net2Phone 2000] and Dialpad [Dialpad 2000]. These PC-to-phone services tend to be free and hence enormously popular with people who love to talk but are on a budget. (Dialpad, which launched in October 1999, claims to have attracted over 3 million users in less than three months). The second class of applications consists of the voice chat applications, for which many companies currently provide products, including Hearme [Hearme 2000], Firetalk [Firetalk 2000], and Lipstream [Lipstream 2000]. These products allow the members in a chat room to converse with their voices, although only one person can talk at a time. The third class of applications is that of asynchronous voice applications, including voice e-mail and voice message boards. These applications allow voice messages to be archived and browsed. Some of the companies in this last space include Wimba [Wimba 2000], Onebox [Onebox 2000], and RocketTalk [RocketTalk 2000].

Real-time Interactive Audio and Video

This class of applications allows people to use audio/video to communicate with each other in real time. Real-time interactive audio is often referred to as **Internet phone,** since, from the user's perspective, it is similar to traditional circuit-switched telephone service. Internet phone can potentially provide PBX, local, and long-distance telephone service at very low cost. It can also facilitate computer-telephone integration (CTI), group real-time communication, directory services, caller identification, caller filtering, and more. There are many Internet telephone products currently available. With real-time interactive video, also called video con ferencing, individuals communicate visually as well as orally. There are also many real-time interactive video products currently available for the Internet, including Microsoft's NetMeeting. Note that in a real-time interactive audio/video application, a user can speak or move at anytime. For a conversation with interaction among multiple speakers, the delay from when a user speaks or moves until the action is manifested at the receiving hosts should be less than a few hundred milliseconds. For voice, delays smaller than 150 milliseconds are not perceived by a human listener, delays between 150 and 400 milliseconds can be acceptable, and delays exceeding 400 milliseconds can result in frustrating, if not completely unintelligible, voice conversations.

6.1.2: Hurdles for Multimedia in Today's Internet

Recall from Chapter 4 that today's Internet's network-layer protocol provides a **best-effort service** to all the datagrams it carries. In other words, the Internet makes its best effort to move each datagram from sender to receiver as quickly as possible. However, best-effort service does not make any promises whatsoever about the end-to-end delay for an individual packet. Nor does the service make any promises about the variation of packet delay within a packet stream. As we learned in Chapter 3, because TCP and UDP run over IP, neither of these protocols can make any delay guarantees to invoking applications. Due to the lack of any special effort to deliver packets in a timely manner, it is an extremely

challenging problem to develop successful multimedia networking applications for the Internet. To date, multimedia over the Internet has achieved significant but limited success. For example, streaming stored audio/video with user-interactivity delays of five-to-ten seconds is now commonplace in the Internet. But during peak traffic periods, performance may be unsatisfactory, particularly when intervening links are congested links (such as congested transoceanic links).

Internet phone and real-time interactive video has, to date, been less successful than streaming stored audio/video. Indeed, real-time interactive voice and video impose rigid constraints on packet delay and packet jitter. **Packet jitter** is the variability of packet delays within the same packet stream. Real-time voice and video can work well in regions where bandwidth is plentiful, and hence delay and jitter are minimal. But quality can deteriorate to unacceptable levels as soon as the real-time voice or video packet stream hits a moderately congested link.

The design of multimedia applications would certainly be more straightforward if there were some sort of first-class and second-class Internet services, whereby first-class packets are limited in number and receive priority service in router queues. Such a first-class service could be satisfactory for delay-sensitive applications. But to date, the Internet has mostly taken an egalitarian approach to packet scheduling in router queues. All packets receive equal service; no packets, including delay-sensitive audio and video packets, receive special priority in the router queues. No matter how much money you have or how important you are, you must join the end of the line and wait your turn! In the latter half of this chapter, we'll examine proposed architectures that aim to remove this restriction.

So for the time being we have to live with best-effort service. But given this constraint, we can make several design decisions and employ a few tricks to improve the user-perceived quality of a multimedia networking application. For example, we can send the audio and video over UDP, and thereby circumvent TCP's low throughput when TCP enters its slow-start phase. We can delay playback at the receiver by 100 msecs or more in order to diminish the effects of network-induced jitter. We can timestamp packets at the sender so that the receiver knows when the packets should be played back. For stored audio/video we can prefetch data during playback when client storage and extra bandwidth is available. We can even send redundant information in order to mitigate the effects of network-induced packet loss. We'll investigate many of these techniques in the rest of the first half of this chapter.

6.1.3: How Should the Internet Evolve to Better Support Multimedia?

Today there is a tremendous--and sometimes ferocious--debate about how the Internet should evolve in order to better accommodate multimedia traffic with its rigid timing constraints. At one extreme, some researchers argue that it isn't necessary to make any fundamental changes to best-effort

service and the underlying Internet protocols. Instead, they argue that it is only necessary to add more bandwidth to the links (along with network caching for stored information and multicast support for one-to-many real-time streaming). Opponents of this viewpoint argue that additional bandwidth can be costly, and that as soon as it is put in place it will be eaten up by new bandwidth-hungry applications (for example, high-definition video on demand).

At the other extreme, some researchers argue that fundamental changes should be made to the Internet so that applications can explicitly reserve end-to-end bandwidth. These researchers feel, for example, that if a user wants to make an Internet phone call from host A to host B, then the user's Internet phone application should be able to explicitly reserve bandwidth in each link along a route from host A to host B. But allowing applications to make reservations and requiring the network to honor the reservations requires some big changes. First we need a protocol that, on the behalf of applications, reserves bandwidth from the senders to their receivers. Second, we must modify scheduling policies in the router queues so that bandwidth reservations can be honored. With these new scheduling policies, not all packets get equal treatment; instead, those that reserve (and pay) more get more. Third, in order to honor reservations, the applications must give the network a description of the traffic that they intend to send into the network. The network must then police each application's traffic to make sure that it abides by the description. Finally, the network must have a means of determining whether it has sufficient available bandwidth to support any new reservation request. These mechanisms, when combined, require new and complex software in the hosts and routers as well as new types of services. We'll look into these mechanisms in more detail, when we examine the so-called Intserv model in Section 6.7.

There is a camp between the two extremes--the so-called differentiated ser vices camp. This camp wants to make relatively small changes at the network and transport layers, and introduce simple pricing and policing schemes at the edge of the network (that is, at the interface between the user and the user's ISP). The idea is to introduce a small number of classes (possibly just two classes), assign each datagram to one of the classes, give datagrams different levels of service according to their class in the router queues, and charge users according to the class of packets that they are sending into the network. We will cover differentiated services in Section 6.9.

6.1.4: Audio and Video Compression

Before audio and video can be transmitted over a computer network, it must be digitized and compressed. The need for digitization is obvious: Computer networks transmit bits, so all transmitted information must be represented as a sequence of bits. Compression is important because uncompressed audio and video consume a tremendous amount of storage and bandwidth; removing the inherent redundancies in digitized audio and

video signals can reduce the amount of data that needs to be stored and transmitted by orders of magnitude. As an example, a single image consisting of 1024 pixels x 1024 pixels with each pixel encoded into 24 bits requires 3 MB of storage without compression. It would take seven minutes to send this image over a 64 Kbps link. If the image is compressed at a modest 10:1 compression ratio, the storage requirement is reduced to 300 KB and the transmission time also drops by a factor of 10.

The fields of audio and video compression are vast. They have been active areas of research for more than 50 years, and there are now literally hundreds of popular techniques and standards for both audio and video compression. Most universities offer entire courses on audio and video compression, and often offer a separate course on audio compression and a separate course on video compression. We therefore provide here a brief and high-level introduction to the subject.

Audio Compression in the Internet

A continuously varying analog audio signal (which could emanate from speech or music) is normally converted to a digital signal as follows:

1. The analog audio signal is first sampled at some fixed rate, for example, at 8,000 samples per second. The value of each sample is an arbitrary real number.

2. Each of the samples is then "rounded" to one of a finite number of values. This operation is referred to as "quantization." The number of finite values--called quantization values--is typically a power of 2, for example, 256 quantization values.

3. Each of the quantization values is represented by a fixed number of bits. For example, if there are 256 quantization values, then each value--and hence each sample--is represented by 1 byte. Each of the samples is converted to its bit representation. The bit representations of all the samples are concatenated together to form the digital representation of the signal.

As an example, if an analog audio signal is sampled at 8,000 samples per second and each sample is quantized and represented by 8 bits, then the resulting digital signal will have a rate of 64,000 bits per second. This digital signal can then be converted back--that is, decoded--to an analog signal for playback. However, the decoded analog signal is typically different from the original audio signal. By increasing the sampling rate and the number of quantization values, the decoded signal can approximate (and even be exactly equal to) the original analog signal. Thus, there is a clear tradeoff between the quality of the decoded signal and the storage and bandwidth requirements of the digital signal.

The basic encoding technique that we just described is called **pulse code modulation (PCM).** Speech encoding often uses PCM, with a sampling rate of 8,000 samples per second and 8 bits per sample, giving a rate of 64 Kbps. The audio compact disk (CD) also uses PCM, with a sampling rate of

44,100 samples per second with 16 bits per sample; this gives a rate of 705.6 Kbps for mono and 1.411 Mbps for stereo.
A bit rate of 1.411 Mbps for stereo music exceeds most access rates, and even 64 Kbps for speech exceeds the access rate for a dial-up modem user. For these reasons, PCM-encoded speech and music are rarely used in the Internet. Instead compression techniques are used to reduce the bit rates of the stream. Popular compression techniques for speech include **GSM** (13 Kbps), **G.729** (8 Kbps), and **G.723.3** (both 6.4 and 5.3 Kbps), and also a large number of proprietary techniques, including those used by RealNetworks. A popular compression technique for near CD-quality stereo music is **MPEG layer 3,** more commonly known as **MP3.** MP3 compresses the bit rate for music to 128 or 112 Kbps, and produces very little sound degradation. When an MP3 file is broken up into pieces, each piece is still playable. This headerless file format allows MP3 music files to be streamed across the Internet (assuming the playback bit rate and speed of the Internet connection are compatible). The MP3 compression standard is complex, using psychoacoustic masking, redundancy reduction, and bit reservoir buffering.

Video Compression in the Internet

A video is a sequence of images, with images typically being displayed at a constant rate, for example at 24 or 30 images per second. An uncompressed, digitally encoded image consists of an array of pixels, with each pixel encoded into a number of bits to represent luminance and color. There are two types of redundancy in video, both of which can be exploited for compression. Spatial redundancy is the redundancy within a given image. For example, an image that consists of mostly white space can be efficiently compressed. Temporal redundancy reflects repetition from image to subsequent image. If, for example, an image and the subsequent image are exactly the same, there is no reason to re-encode the subsequent image; it is more efficient to simply indicate during encoding that the subsequent image is exactly the same.
The MPEG compression standards are among the most popular compression techniques. These include **MPEG 1** for CD-ROM quality video (1.5 Mbps), **MPEG 2** for high-quality **DVD** video (3-6 Mbps), and **MPEG 4** for object-oriented video compression. The MPEG standard draws heavily from the JPEG standard for image compression. The **H.261** video compression standards are also very popular in the Internet. There are also numerous proprietary standards.
Readers interested in learning more about audio and video encoding are encouraged to see [Rao 1996] and [Solari 1997]. A good book on multimedia networking in general is [Crowcroft 1999].

6.2: Streaming Stored Audio and Video

In recent years, audio/video streaming has become a popular application and a major consumer of network bandwidth. This trend is likely to continue for several reasons. First, the cost of disk storage is decreasing rapidly, even faster than processing and bandwidth costs. Cheap storage will lead to a significant increase in the amount of stored audio/video in the Internet. For example, shared MP3 audio files of rock music via [Napster 2000] has become incredibly popular among college and high school students. Second, improvements in Internet infrastructure, such as high-speed residential access (that is, cable modems and ADSL, as discussed in Chapter 1), network caching of video (see Section 2.2), and new QoS-oriented Internet protocols (see Sections 6.5-6.9) will greatly facilitate the distribution of stored audio and video. And third, there is an enormous pent-up demand for high-quality video streaming, an application that combines two existing killer communication technologies--television and the on-demand Web.

In audio/video streaming, clients request compressed audio/video files that are resident on servers. As we'll discuss in this section, these servers can be "ordinary" Web servers, or can be special streaming servers tailored for the audio/video streaming application. Upon client request, the server directs an audio/video file to the client by sending the file into a socket. Both TCP and UDP socket connections are used in practice. Before sending the audio/video file into the network, the file is segmented, and the segments are typically encapsulated with special headers appropriate for audio/video traffic. The **Real-time protocol (RTP),** discussed in Section 6.4, is a public-domain standard for encapsulating such segments. Once the client begins to receive the requested audio/video file, the client begins to render the file (typically) within a few seconds. Most existing products also provide for user interactivity, for example, pause/resume and temporal jumps within the audio/video file. This user interactivity also requires a protocol for client/server interaction. **Real-time streaming protocol (RTSP),** discussed at the end of this section, is a public-domain protocol for providing user interactivity.

Audio/video streaming is often requested by users through a Web client (that is, browser). But because audio/video playout is not integrated directly in today's Web clients, a separate **helper application** is required for playing out the audio/video. The helper application is often called a **media player,** the most popular of which are currently RealNetworks' Real Player and the Microsoft Windows Media Player. The media player performs several functions, including:

- *Decompression.* Audio/video is almost always compressed to save disk storage and network bandwidth. A media player must

decompress the audio/video on the fly during playout.

- *Jitter removal.* Packet jitter is the variability of source-to-destination delays of packets within the same packet stream. Since audio and video must be played out with the same timing with which it was recorded, a receiver will buffer received packets for a short period of time to remove this jitter. We'll examine this topic in detail in Section 6.3.
- *Error correction.* Due to unpredictable congestion in the Internet, a fraction of packets in the packet stream can be lost. If this fraction becomes too large, user-perceived audio/video quality becomes unacceptable. To this end, many streaming systems attempt to recover from losses by either (1) reconstructing lost packets through the transmission of redundant packets, (2) by having the client explicitly request retransmissions of lost packets, (3) masking loss by interpolating the missing data from the received data.
- *Graphical user interface with control knobs.* This is the actual interface that the user interacts with. It typically includes volume controls, pause/resume buttons, sliders for making temporal jumps in the audio/video stream, and so on.

Plug-ins may be used to embed the user interface of the media player within the window of the Web browser. For such embeddings, the browser reserves screen space on the current Web page, and it is up to the media player to manage the screen space. But either appearing in a separate window or within the browser window (as a plug-in), the media player is a program that is being executed separately from the browser.

6.2.1: Accessing Audio and Video from a Web Server

Stored audio/video can reside either on a Web server that delivers the audio/video to the client over HTTP, or on an audio/video streaming server that delivers the audio/video over non-HTTP protocols (protocols that can be either proprietary or open standards). In this subsection, we examine delivery of audio/video from a Web server; in the next subsection, we examine delivery from a streaming server.

Consider first the case of audio streaming. When an audio file resides on a Web server, the audio file is an ordinary object in the server's file system, just as are HTML and JPEG files. When a user wants to hear the audio file, the user's host establishes a TCP connection with the Web server and sends an HTTP request for the object (see Section 2.2). Upon receiving a request, the Web server bundles the audio file in an HTTP response message and sends the response message back into the TCP connection. The case of video can be a little more tricky, because the audio and video parts of the "video" may be stored in two different files, that is, they may be two different objects in the Web server's file system. In this case, two separate HTTP requests are sent to the server (over two separate TCP

connections for HTTP/1.0), and the audio and video files arrive at the client in parallel. It is up to the client to manage the synchronization of the two streams. It is also possible that the audio and video are interleaved in the same file, so that only one object need be sent to the client. To keep our discussion simple, for the case of "video" we assume that the audio and video are contained in one file.

A naive architecture for audio/video streaming is shown in Figure 6.1. In this architecture:

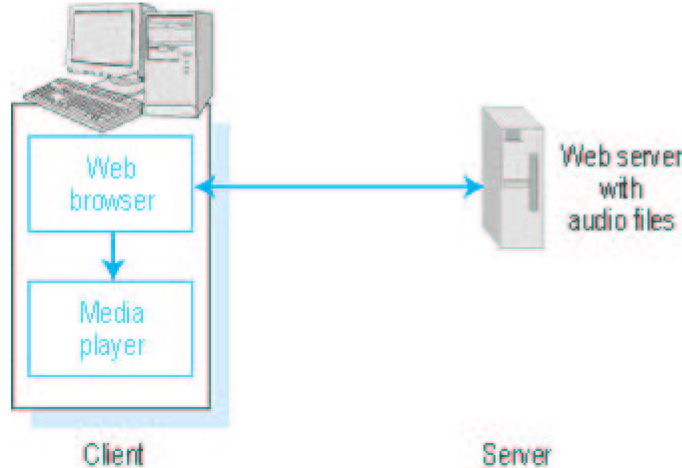

Figure 6.1: A naive implementation for audio streaming

1. The browser process establishes a TCP connection with the Web server and requests the audio/video file with an HTTP request message.

2. The Web server sends to the browser the audio/video file in an HTTP response message.

3. The content-type header line in the HTTP response message indicates a specific audio/video encoding. The client browser examines the content-type of the response message, launches the associated media player, and passes the file to the media player.

4. The media player then renders the audio/video file.

Although this approach is very simple, it has a major drawback: The media player (that is, the helper application) must interact with the server through the intermediary of a Web browser. This can lead to many problems. In particular, when the browser is an intermediary, the entire object must be downloaded before the browser passes the object to a helper application. The resulting delay before playout can begin is typically unacceptable for audio/video clips of moderate length. For this reason, audio/video streaming implementations typically have the server send the audio/video file directly to the media player process. In other words, a direct socket connection is made between the server process and the media player process. As shown in Figure 6.2, this is typically done by making use of a **meta file,** a file that provides information (for example, URL, type of

encoding) about the audio/video file that is to be streamed.

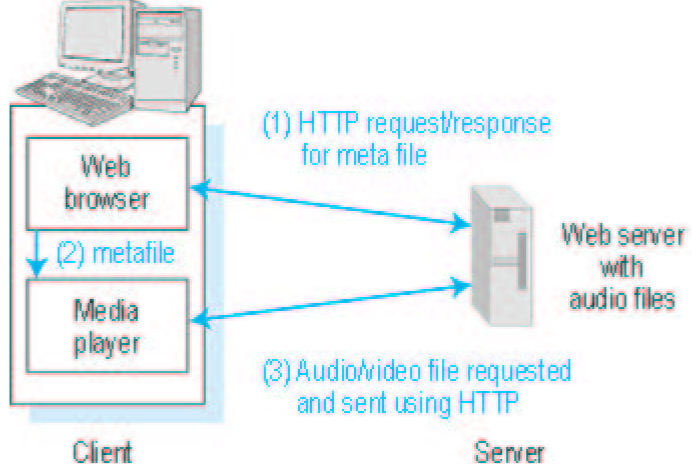

Figure 6.2: Web server sends audio/video directly to the media player

A direct TCP connection between the server and the media player is obtained as follows:

1. The user clicks on a hyperlink for an audio/video file.

2. The hyperlink does not point directly to the audio/video file, but instead to a meta file. The meta file contains the URL of the actual audio/video file. The HTTP response message that encapsulates the meta file includes a content-type header line that indicates the specific audio/video application.

3. The client browser examines the content-type header line of the response message, launches the associated media player, and passes the entire body of the response message (that is, the meta file) to the media player.

4. The media player sets up a TCP connection directly with the HTTP server. The media player sends an HTTP request message for the audio/video file into the TCP connection.

5. The audio/video file is sent within an HTTP response message to the media player. The media player streams out the audio/video file.

The importance of the intermediate step of acquiring the meta file is clear. When the browser sees the content-type for the file, it can launch the appropriate media player, and thereby have the media player directly contact the server.

We have just learned how a meta file can allow a media player to dialogue directly with a Web server housing an audio/video. Yet many companies that sell products for audio/video streaming do not recommend the architecture we just described. This is because the architecture has the media player communicate with the server over HTTP and hence also over TCP. HTTP is often considered insufficiently rich to allow for satisfactory user interaction with the server; in particular, HTTP does not easily allow a

user (through the media server) to send pause/resume, fast-forward, and temporal jump commands to the server.

6.2.2: Sending Multimedia from a Streaming Server to a Helper Application

In order to get around HTTP and/or TCP, audio/video can be stored on and sent from a streaming server to the media player. This streaming server could be a proprietary streaming server, such as those marketed by RealNetworks and Microsoft, or could be a public-domain streaming server. With a streaming server, audio/video can be sent over UDP (rather than TCP) using application-layer protocols that may be better tailored than HTTP to audio/video streaming.

This architecture requires two servers, as shown in Figure 6.3. One server, the HTTP server, serves Web pages (including meta files). The second server, the **streaming server,** serves the audio/video files. The two servers can run on the same end system or on two distinct end systems. The steps for this architecture are similar to those described in the previous architecture. However, now the media player requests the file from a streaming server rather than from a Web server, and now the media player and streaming server can interact using their own protocols. These protocols can allow for rich user interaction with the audio/video stream.

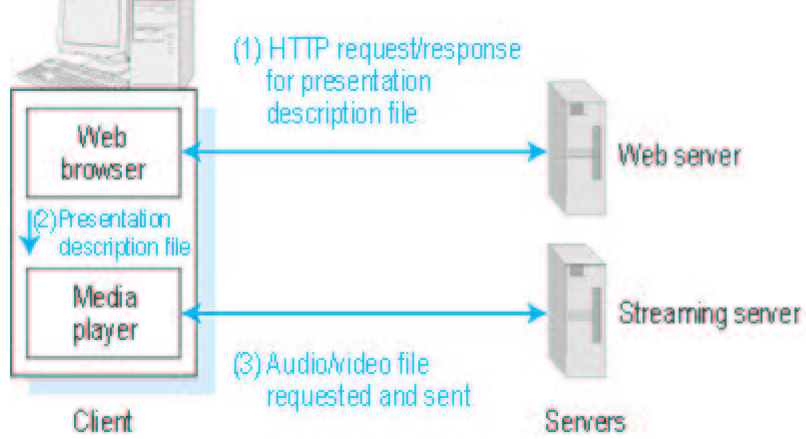

Figure 6.3: Streaming from a streaming server to a media player

In the architecture of Figure 6.3, there are many options for delivering the audio/ video from the streaming server to the media player. A partial list of the options is given below:

1. The audio/video is sent over UDP at a constant rate equal to the drain rate at the receiver (which is the encoded rate of the audio/video). For example, if the audio is compressed using GSM at a rate of 13 Kbps, then the server clocks out the compressed audio file at 13 Kbps. As soon as the client receives compressed audio/video from the network, it decompresses the audio/video and plays it back.

2. This is the same as option 1, but the media player delays playout for

2-5 seconds in order to eliminate network-induced jitter. The client accomplishes this task by placing the compressed media that it receives from the network into a **client buffer,** as shown in Figure 6.4. Once the client has "prefetched" a few seconds of the media, it begins to drain the buffer. For this, and the previous option, the fill rate $x(t)$ is equal to the drain rate d, except when there is packet loss, in which case $x(t)$ is momentarily less than d.

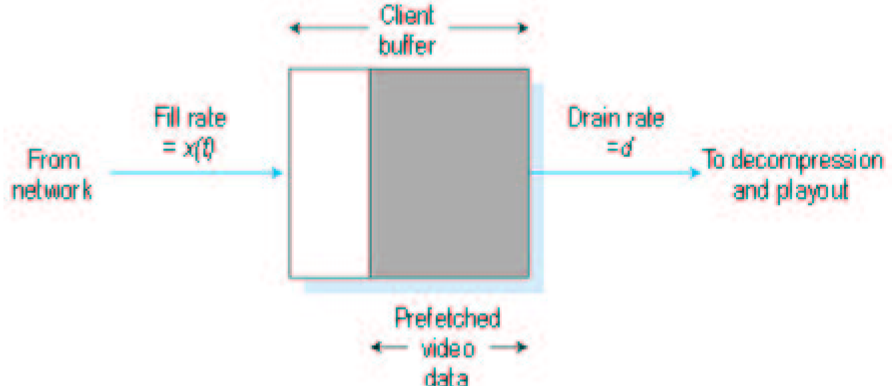

Figure 6.4: Client buffer being filled at rate *x(t)* and drained at rate *d*

3. The media is sent over TCP. The server pushes the media file into the TCP socket as quickly as it can; the client (i.e., media player) reads from the TCP socket as quickly as it can, and places the compressed video into the media player buffer. After an initial 2-5 second delay, the media player reads from its buffer at a rate d and forwards the compressed media to decompression and playback. Because TCP retransmits lost packets, it has the potential to provide better sound quality than UDP. On the other hand, the fill rate $x(t)$ now fluctuates with time due to TCP congestion control and window flow control. In fact, after packet loss, TCP congestion control may reduce the instantaneous rate to less than d for long periods of time. This can empty the client buffer and introduce undesirable pauses into the output of the audio/video stream at the client.

   For the third option, the behavior of $x(t)$ will very much depend on the size of the client buffer (which is not to be confused with the TCP receive buffer). If this buffer is large enough to hold all of the media file (possibly within disk storage), then TCP will make use of all the instantaneous bandwidth available to the connection, so that $x(t)$ can become much larger than d. If $x(t)$ becomes much larger than d for long periods of time, then a large portion of media is prefetched into the client, and subsequent client starvation is unlikely. If, on the other hand, the client buffer is small, then $x(t)$ will fluctuate around the drain rate d. Risk of client starvation is much larger in this case.

6.2.3: Real-Time Streaming Protocol (RTSP)

Many Internet multimedia users (particularly those who grew up with a remote TV control in hand) will want to *control* the playback of continuous media by pausing playback, repositioning playback to a future or past point of time, visual fast-forwarding playback, visual rewinding playback, and so on. This functionality is similar to what a user has with a VCR when watching a video cassette or with a CD player when listening to a music CD. To allow a user to control playback, the media player and server need a protocol for exchanging playback control information. RTSP, defined in RFC 2326, is such a protocol.

But before getting into the details of RTSP, let us first indicate what RTSP does not do:

- RTSP does not define compression schemes for audio and video.
- RTSP does not define how audio and video is encapsulated in packets for transmission over a network; encapsulation for streaming media can be provided by RTP or by a proprietary protocol. (RTP is discussed in Section 6.4.) For example, RealMedia's G2 server and player use RTSP to send control information to each other. But the media stream itself can be encapsulated in RTP packets or in some proprietary data format.
- RTSP does not restrict how streamed media is transported; it can be transported over UDP or TCP.
- RTSP does not restrict how the media player buffers the audio/video. The audio/video can be played out as soon as it begins to arrive at the client, it can be played out after a delay of a few seconds, or it can be downloaded in its entirety before playout.

So if RTSP doesn't do any of the above, what does RTSP do? RTSP is a protocol that allows a media player to control the transmission of a media stream. As mentioned above, control actions include pause/resume, repositioning of playback, fast forward and rewind. RTSP is a so-called **out-of-band protocol.** In particular, the RTSP messages are sent out-of-band, whereas the media stream, whose packet structure is not defined by RTSP, is considered "in-band." RTSP messages use a different port number, 544, than the media stream. The RTSP specification [RFC 2326] permits RTSP messages to be sent over either TCP or UDP.

Recall from Section 2.3, that file transfer protocol (FTP) also uses the out-of-band notion. In particular, FTP uses two client/server pairs of sockets, each pair with its own port number: one client/server socket pair supports a TCP connection that transports control information; the other client/server socket pair supports a TCP connection that actually transports the file. The RTSP channel is in many ways similar to FTP's control channel.

Let us now walk through a simple RTSP example, which is illustrated in Figure 6.5. The Web browser first requests a presentation description file from a Web server. The presentation description file can have references to several continuous-media files as well as directives for synchronization of

the continuous-media files. Each reference to a continuous-media file begins with the URL method, rtsp://.

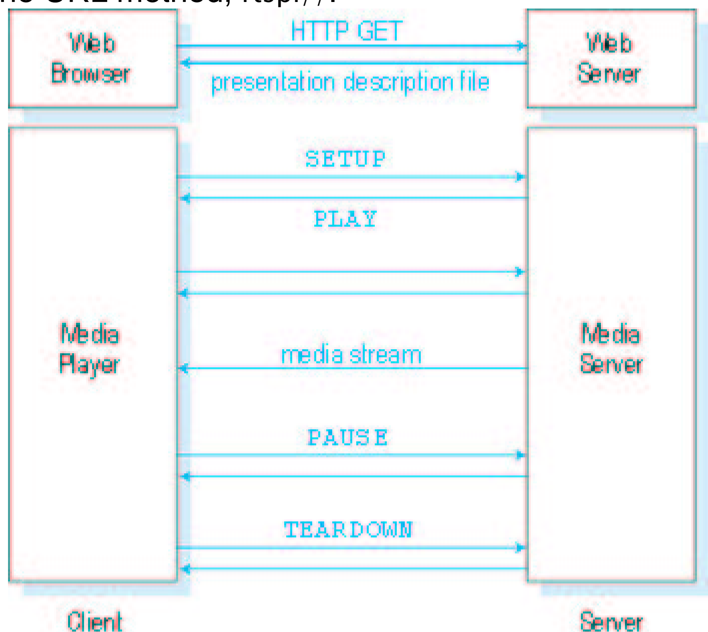

Figure 6.5: Interaction between client and server using RTSP

Below we provide a sample presentation file that has been adapted from [Schulzrinne 1997]. In this presentation, an audio and video stream are played in parallel and in lip sync (as part of the same "group"). For the audio stream, the media player can choose ("switch") between two audio recordings, a low-fidelity recording and a high-fidelity recording.

```
<title>Twister</title>
<session>
  <group language=en lipsync>
     <switch>
        <track type=audio
           e="PCMU/8000/1"
           src="rtsp://audio.example.com/twister/audio.en/lofi">
        <track type=audio
           e="DVI4/16000/2" pt="90 DVI4/8000/1"
           src="rtsp://audio.example.com/twister/audio.en/hifi">
     </switch>
    <track type="video/jpeg"
           src="rtsp://video.example.com/twister/video">
  </group>
</session>
```

The Web server encapsulates the presentation description file in an HTTP response message and sends the message to the browser. When the browser receives the HTTP response message, the browser invokes a media player (that is, the helper application) based on the content-type field of the message. The presentation description file includes references to

media streams, using the URL method rtsp://, as shown in the above sample. As shown in Figure 6.5, the player and the server then send each other a series of RTSP messages. The player sends an RTSP SETUP request, and the server sends an RTSP SETUP response. The player sends an RTSP PLAY request, say, for low-fidelity audio, and the server sends an RTSP PLAY response. At this point, the streaming server pumps the low-fidelity audio into its own in-band channel. Later, the media player sends an RTSP PAUSE request, and the server responds with an RTSP PAUSE response. When the user is finished, the media player sends an RTSP TEARDOWN request, and the server responds with an RTSP TEARDOWN response.

Each RTSP session has a session identifier, which is chosen by the server. The client initiates the session with the SETUP request, and the server responds to the request with an identifier. The client repeats the session identifier for each request, until the client closes the session with the TEARDOWN request. The following is a simplified example of an RTSP session between a client (**C:**) and a sender (**S:**).

C: SETUP rtsp://audio.example.com/twister/audio RTSP/1.0
  Transport: rtp/udp; compression; port=3056; mode=PLAY

S: RTSP/1.0 200 1 OK
  Session 4231

C: PLAY rtsp://audio.example.com/twister/audio.en/lofi
    RTSP/1.0
  Session: 4231
  Range: npt=0-

C: PAUSE rtsp://audio.example.com/twister/audio.en/
    lofi RTSP/1.0
  Session: 4231
  Range: npt=37

C: TEARDOWN rtsp://audio.example.com/twister/audio.en/
  lofi RTSP/1.0 Session: 4231

S: 200 3 OK

Notice that in this example, the player chose not to play back the complete presentation, but instead only the low-fidelity portion of the presentation. The RTSP protocol is actually capable of doing much more than described in this brief introduction. In particular, RTSP has facilities that allow clients to stream toward the server (for example, for recording). RTSP has been adopted by RealNetworks, currently the industry leader in audio/video streaming. Henning Schulzrinne makes available a Web page on RTSP [Schulzrinne 1999].

© 2000-2001 by Addison Wesley Longman

6.3: Internet Phone Example

The Internet's network-layer protocol, IP, provides a best-effort service. That is to say that the Internet makes its best effort to move each datagram from source to destination as quickly as possible. However, best-effort service does not make any promises whatsoever on the extent of the end-to-end delay for an individual packet, or on the extent of packet jitter and packet loss within the packet stream.

Real-time interactive multimedia applications, such as Internet phone and real-time video conferencing, are acutely sensitive to packet delay, jitter, and loss. Fortunately, designers of these applications can introduce several useful mechanisms that can preserve good audio and video quality as long as delay, jitter, and loss are not excessive. In this section, we examine some of these mechanisms. To keep the discussion concrete, we discuss these mechanisms in the context of an **Internet phone application,** described below. The situation is similar for real-time video conferencing applications [Bolot 1994].

The speaker in our Internet phone application generates an audio signal consisting of alternating talk spurts and silent periods. In order to conserve bandwidth, our Internet phone application only generates packets during talk spurts. During a talk spurt the sender generates bytes at a rate of 8 Kbytes per second, and every 20 milliseconds the sender gathers bytes into chunks. Thus, the number of bytes in a chunk is (20 msecs) · (8 Kbytes/sec) = 160 bytes. A special header is attached to each chunk, the contents of which is discussed below. The chunk and its header are encapsulated in a UDP segment, and then the UDP datagram is sent into the socket interface. Thus, during a talk spurt, a UDP segment is sent every 20 msec.

If each packet makes it to the receiver and has a small constant end-to-end delay, then packets arrive at the receiver periodically every 20 msec during a talk spurt. In these ideal conditions, the receiver can simply play back each chunk as soon as it arrives. But, unfortunately, some packets can be lost and most packets will not have the same end-to-end delay, even in a lightly congested Internet. For this reason, the receiver must take more care in (1) determining when to play back a chunk, and (2) determining what to do with a missing chunk.

6.3.1: The Limitations of a Best-Effort Service

We mentioned that the best-effort service can lead to packet loss, excessive end-to-end delay, and delay jitter. Let's examine these issues in more detail.

Packet Loss

Consider one of the UDP segments generated by our Internet phone application. The UDP segment is encapsulated in an IP datagram. As the datagram wanders through the network, it passes through buffers (that is, queues) in the routers in order to access outbound links. It is possible that one or more of the buffers in the route from sender to receiver is full and cannot admit the IP datagram. In this case, the IP datagram is discarded, never to arrive at the receiving application.

Loss could be eliminated by sending the packets over TCP rather than over UDP. Recall that TCP retransmits packets that do not arrive at the destination. However, retransmission mechanisms are often considered unacceptable for interactive real-time audio applications such as Internet phone, because they increase end-to-end delay [Bolot 1996]. Furthermore, due to TCP congestion control, after packet loss the transmission rate at the sender can be reduced to a rate that is lower than the drain rate at the receiver. This can have a severe impact on voice intelligibility at the receiver. For these reasons, almost all existing Internet phone applications run over UDP and do not bother to retransmit lost packets.

But losing packets is not necessarily as grave as one might think. Indeed, packet loss rates between 1% and 20% can be tolerated, depending on how the voice is encoded and transmitted, and on how the loss is concealed at the receiver. For example, forward error correction (FEC) can help conceal packet loss. We'll see below that with FEC, redundant information is transmitted along with the original information so that some of the lost original data can be recovered from the redundant information. Nevertheless, if one or more of the links between sender and receiver is severely congested, and packet loss exceeds 10-20%, then there is really nothing that can be done to achieve acceptable sound quality. Clearly, best-effort service has its limitations.

End-to-End Delay

End-to-end delay is the accumulation of transmission processing and queuing delays in routers, propagation delays, and end-system processing delays along a path from source to destination. For highly interactive audio applications, such as Internet phone, end-to-end delays smaller than 150 milliseconds are not perceived by a human listener; delays between 150 and 400 milliseconds can be acceptable but not ideal; and delays exceeding 400 milliseconds can seriously hinder the interactivity in voice

conversations. The receiver in an Internet phone application will typically disregard any packets that are delayed more than a certain threshold, for example, more than 400 milliseconds. Thus, packets that are delayed by more than the threshold are effectively lost.

Delay Jitter

A crucial component of end-to-end delay is the random queuing delays in the routers. Because of these varying delays within the network, the time from when a packet is generated at the source until it is received at the receiver can fluctuate from packet to packet. This phenomenon is called **jitter.**

As an example, consider two consecutive packets within a talk spurt in our Internet phone application. The sender sends the second packet 20 msec after sending the first packet. But at the receiver, the spacing between these packets can become greater than 20 msec. To see this, suppose the first packet arrives at a nearly empty queue at a router, but just before the second packet arrives at the queue a large number of packets from other sources arrive to the same queue. Because the second packet suffers a large queuing delay, the first and second packets become spaced apart by more than 20 msecs. The spacing between consecutive packets can also become less than 20 msecs. To see this, again consider two consecutive packets within a talk spurt. Suppose the first packet joins the end of a queue with a large number of packets, and the second packet arrives at the queue before packets from other sources arrive at the queue. In this case, our two packets find themselves right behind each other in the queue. If the time it takes to transmit a packet on the router's inbound link is less than 20 msecs, then the first and second packets become spaced apart by less than 20 msecs.

The situation is analogous to driving cars on roads. Suppose you and your friend are each driving in your own cars from San Diego to Phoenix. Suppose you and your friend have similar driving styles, and that you both drive at 100 km/ hour, traffic permitting. Finally, suppose your friend starts out one hour before you. Then, depending on intervening traffic, you may arrive at Phoenix more or less than one hour after your friend.

If the receiver ignores the presence of jitter, and plays out chunks as soon as they arrive, then the resulting audio quality can easily become unintelligible at the receiver. Fortunately, jitter can often be removed by using **sequence numbers, timestamps,** and a **playout delay,** as discussed below.

6.3.2: Removing Jitter at the Receiver for Audio

For a voice application such as Internet phone or audio-on-demand, the receiver should attempt to provide synchronous playout of voice chunks in the presence of random network jitter.

This is typically done by combining the following three mechanisms:

- **Prefacing each chunk with** a **sequence number**. The sender increments the sequence number by one for each of the packet it generates.
- **Prefacing each chunk with** a **timestamp**. The sender stamps each chunk with the time at which the chunk was generated.
- **Delaying playout** of chunks at the receiver. The playout delay of the received audio chunks must be long enough so that most of the packets are received before their scheduled playout times. This playout delay can either be fixed throughout the duration of the conference, or it may vary adaptively during the conference's lifetime. Packets that do not arrive before their scheduled playout times are considered lost and forgotten; as noted above, the receiver may use some form of speech interpolation to attempt to conceal the loss.

We now discuss how these three mechanisms, when combined, can alleviate or even eliminate the effects of jitter. We examine two playback strategies: fixed playout delay and adaptive playout delay.

Fixed Playout Delay

With the fixed delay strategy, the receiver attempts to playout each chunk exactly *q* msecs after the chunk is generated. So if a chunk is timestamped at time *t*, the receiver plays out the chunk at time *t* + *q*, assuming the chunk has arrived by that time. Packets that arrive after their scheduled playout times are discarded and considered lost.

What is a good choice for *q*? Internet telephone can support delays up to about 400 msecs, although a more satisfying interactive experience is achieved with smaller values of *q*. On the other hand, if *q* is made much smaller than 400 msecs, then many packets may miss their scheduled playback times due to the network-induced delay jitter. Roughly speaking, if large variations in end-to-end delay are typical, it is preferable to use a large *q*; on the other hand, if delay is small and variations in delay are also small, it is preferable to use a small *q*, perhaps less than 150 msecs.

The tradeoff between the playback delay and packet loss is illustrated in Figure 6.6. The figure shows the times at which packets are generated and played out for a single talkspurt. Two distinct initial playout delays are considered. As shown by the leftmost staircase, the sender generates packets at regular

intervals--say, every 20 msec. The first packet in this talkspurt is received at time *r*. As shown in the figure, the arrivals of subsequent packets are not evenly spaced due to the network jitter.

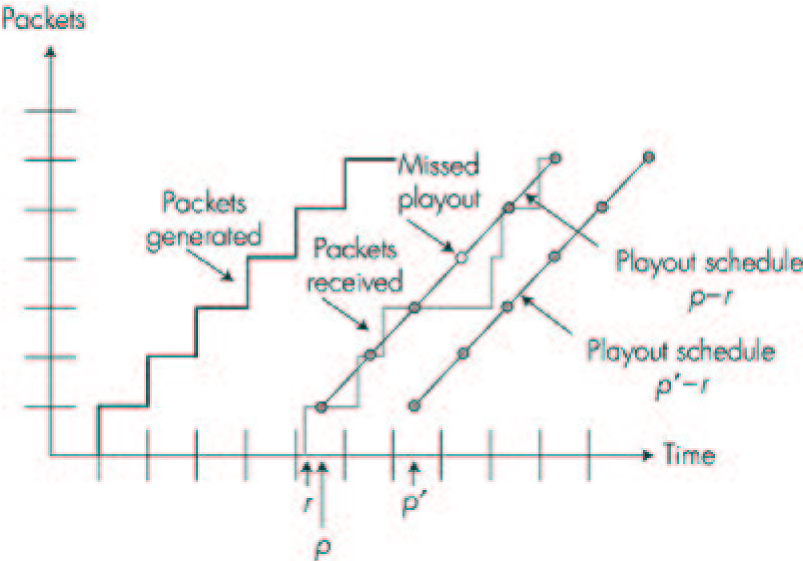

Figure 6.6: Packet loss for different fixed playout delays

For the first playout schedule, the fixed initial playout delay is set to *p* - *r*. With this schedule, the fourth packet does not arrive by its scheduled playout time, and the receiver considers it lost. For the second playout schedule, the fixed initial playout delay is set to *p*' - *r*. For this schedule, all packets arrive before their scheduled playout times, and there is therefore no loss.

Adaptive Playout Delay

The above example demonstrates an important delay-loss tradeoff that arises when designing a playout strategy with fixed playout delays. By making the initial playout delay large, most packets will make their deadlines and there will therefore be negligible loss; however, for interactive services such as Internet phone, long delays can become bothersome if not intolerable. Ideally, we would like the play-out delay to be minimized subject to the constraint that the loss be below a few percent.

The natural way to deal with this tradeoff is to estimate the network delay and the variance of the network delay, and to adjust the playout delay accordingly at the beginning of each talkspurt. This adaptive adjustment of playout delays at the beginning of the talkspurts will cause the sender's silent periods to be compressed and elongated; however, compression and elongation of silence by a small amount is not noticeable in speech.

Following [Ramjee 1994], we now describe a generic algorithm that the receiver can use to adaptively adjust its playout delays. To this end, let

t_i = timestamp of the *i*th packet = the time packet was generated by

sender
r_i = the time packet i is received by receiver
p_i = the time packet i is played at receiver
The end-to-end network delay of the ith packet is $r_i - t_i$. Due to network jitter, this delay will vary from packet to packet. Let d_i denote an estimate of the *average* network delay upon reception of the ith packet. This estimate is constructed from the timestamps as follows:

$$d_i = (1 - u)\ d_{i-1} + u\ (r_i - t_i)$$

where u is a fixed constant (for example, $u = 0.01$). Thus d_i is a smoothed average of the observed network delays $r_1 - t_1, \ldots, r_i - t_i$. The estimate places more weight on the recently observed network delays than on the observed network delays of the distant past. This form of estimate should not be completely unfamiliar; a similar idea is used to estimate round-trip times in TCP, as discussed in Chapter 3. Let v_i denote an estimate of the average deviation of the delay from the estimated average delay. This estimate is also constructed from the timestamps:

$$v_i = (1 - u)\ v_{i-1} + u\ |\ r_i - t_i - d_i\ |$$

The estimates d_i and v_i are calculated for every packet received, although they are only used to determine the playout point for the first packet in any talkspurt.
Once having calculated these estimates, the receiver employs the following algorithm for the playout of packets. If packet i is the first packet of a talkspurt, its playout time, p_i, is computed as:

$$p_i = t_i + d_i + Kv_i$$

where K is a positive constant (for example, $K = 4$). The purpose of the Kv_i term is to set the playout time far enough into the future so that only a small fraction of the arriving packets in the talkspurt will be lost due to late arrivals. The playout point for any subsequent packet in a talkspurt is computed as an offset from the point in time when the first packet in the talkspurt was played out. In particular, let

$$q_i = p_i - t_i$$

be the length of time from when the first packet in the talkspurt is generated until it is played out. If packet j also belongs to this talkspurt, it is played out at time

$$p_j = t_j + q_i$$

The algorithm just described makes perfect sense assuming that the receiver can tell whether a packet is the first packet in the talkspurt. If there is no packet loss, then the receiver can determine whether packet i is the first packet of the talkspurt by comparing the timestamp of the ith packet with the timestamp of the $(i - 1)$st packet. Indeed, if $t_i - t_{i-1} > 20$ msec, then the receiver knows that ith packet starts a new talkspurt. But now suppose there is occasional packet loss. In this case, two successive

packets received at the destination may have timestamps that differ by more than 20 msec when the two packets belong to the same talkspurt. So here is where the sequence numbers are particularly useful. The receiver can use the sequence numbers to determine whether a difference of more than 20 msec in timestamps is due to a new talkspurt or to lost packets.

6.3.3: Recovering from Packet Loss

We have discussed in some detail how an Internet phone application can deal with packet jitter. We now briefly describe several schemes that attempt to preserve acceptable audio quality in the presence of packet loss. Such schemes are called **loss recovery schemes.** Here we define packet loss in a broad sense: a packet is lost if either it never arrives at the receiver or if it arrives after its scheduled playout time. Our Internet phone example will again serve as a context for describing loss recovery schemes.

As mentioned at the beginning of this section, retransmitting lost packets is not appropriate in an interactive real-time application such as Internet phone. Indeed, retransmitting a packet that has missed its playout deadline serves absolutely no purpose. And retransmitting a packet that overflowed a router queue cannot normally be accomplished quickly enough. Because of these considerations, Internet phone applications often use some type of loss anticipation scheme. Two types of loss-anticipation schemes are **forward error correction (FEC)** and **interleaving.**

Forward Error Correction (FEC)

The basic idea of FEC is to add redundant information to the original packet stream. For the cost of marginally increasing the transmission rate of the audio of the stream, the redundant information can be used to reconstruct "approximations" or exact versions of some of the lost packets. Following [Bolot 1996] and [Perkins 1998], we now outline two FEC mechanisms. The first mechanism sends a redundant encoded chunk after every n chunks. The redundant chunk is obtained by exclusive OR-ing the n original chunks [Shacham 1990]. In this manner if any one packet of the group of $n + 1$ packets is lost, the receiver can fully reconstruct the lost packet. But if two or more packets in a group are lost, the receiver cannot reconstruct the lost packets. By keeping $n + 1$, the group size, small, a large fraction of the lost packets can be recovered when loss is not excessive. However, the smaller the group size, the greater the relative increase of the transmission rate of the audio stream. In particular, the transmission rate increases by a factor of $1/n$; for example, if $n = 3$, then the transmission rate increases by 33%. Furthermore, this simple scheme increases the playout delay, as the receiver must wait to receive the entire group of packets before it can begin playout.

The second FEC mechanism is to send a lower-resolution audio stream as the redundant information. For example, the sender might create a nominal audio stream and a corresponding low-resolution low-bit rate audio stream. (The nominal stream could be a PCM encoding at 64 Kbps and the lower-quality stream could be a GSM encoding at 13 Kbps.) The low-bit-rate stream is referred to as the redundant stream. As shown in Figure 6.7, the sender constructs the *n*th packet by taking the *n*th chunk from the nominal stream and appending to it the (*n* - 1)st chunk from the redundant stream. In this manner, whenever there is nonconsecutive packet loss, the receiver can conceal the loss by playing out the low-bit-rate encoded chunk that arrives with the subsequent packet. Of course, low-bit-rate chunks give lower quality than the nominal chunks. However, a stream of mostly high-quality chunks, occasional low-quality chunks, and no missing chunks gives good overall audio quality. Note that in this scheme, the receiver only has to receive two packets before playback, so that the increased playout delay is small. Furthermore, if the low-bit-rate encoding is much less than the nominal encoding, then the marginal increase in the transmission rate will be small.

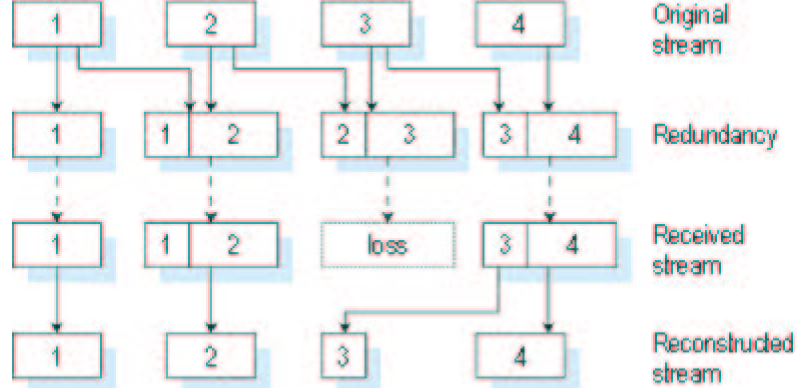

Figure 6.7: Piggybacking lower-quality redundant information

In order to cope with consecutive loss, a simple variation can be employed. Instead of appending just the (*n* - 1)st low-bit-rate chunk to the *n*th nominal chunk, the sender can append the (*n* - 1)st and (*n* - 2)nd low-bit-rate chunk, or append the (*n* - 1)st and (*n* - 3)rd low-bit-rate chunk, etc. By appending more low-bit-rate chunks to each nominal chunk, the audio quality at the receiver becomes acceptable for a wider variety of harsh best-effort environments. On the other hand, the additional chunks increase the transmission bandwidth and the playout delay.

Free Phone [Freephone 1999] and RAT [RAT 1999] are well-documented Internet phone applications that use FEC. They can transmit lower-quality audio streams along with the nominal audio stream, as described above.

Interleaving

As an alternative to redundant transmission, an Internet phone

application can send interleaved audio. As shown in Figure 6.8, the sender resequences units of audio data before transmission, so that originally adjacent units are separated by a certain distance in the transmitted stream. Interleaving can mitigate the effect of packet losses. If, for example, units are 5 msec in length and chunks are 20 msec (that is, 4 units per chunk), then the first chunk could contain units 1, 5, 9, 13; the second chunk could contain units 2, 6, 10, 14; and so on. Figure 6.8 shows that the loss of a single packet from an interleaved stream results in multiple small gaps in the reconstructed stream, as opposed to the single large gap that would occur in a noninterleaved stream.

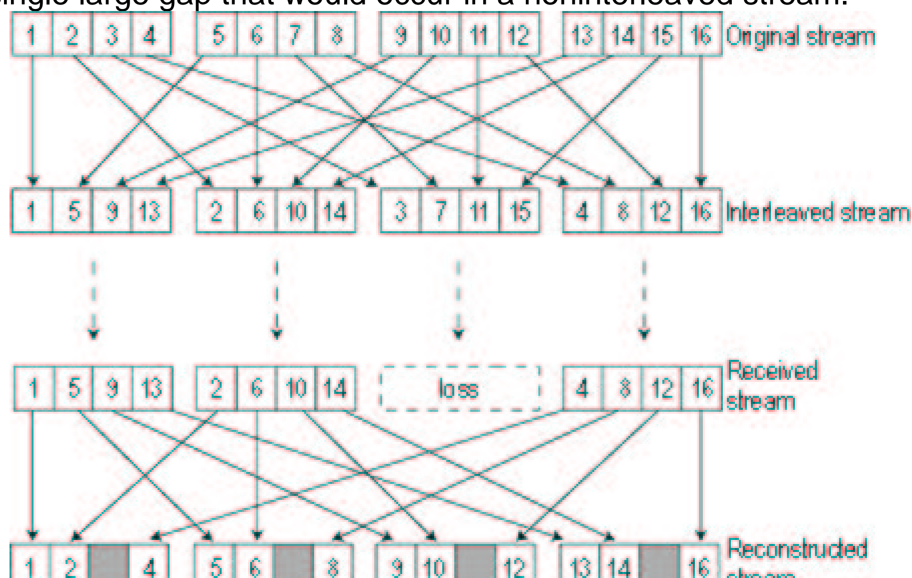

Figure 6.8: Sending interleaved audio

Interleaving can significantly improve the perceived quality of an audio stream [Perkins 1998]. It also has low overhead. The obvious disadvantage of interleaving is that it increases latency. This limits its use for interactive applications such as Internet phone, although it can perform well for streaming stored audio. A major advantage of interleaving is that it does not increase the bandwidth requirements of a stream.

Receiver-Based Repair of Damaged Audio Streams

Receiver-based recovery schemes attempt to produce a replacement for a lost packet that is similar to the original. As discussed in [Perkins 1998], this is possible since audio signals, and in particular speech, exhibit large amounts of short-term self similarity. As such, these techniques work for relatively small loss rates (less than 15%), and for small packets (4-40 msec). When the loss length approaches the length of a phoneme (5-100 msec) these techniques breakdown, since whole phonemes may be missed by the listener.

Perhaps the simplest form of receiver-based recovery is packet repetition. Packet repetition replaces lost packets with copies of

the packets that arrived immediately before the loss. It has low computational complexity and performs reasonably well. Another form of receiver-based recovery is interpolation, which uses audio before and after the loss to interpolate a suitable packet to cover the loss. It performs somewhat better than packet repetition, but is significantly more computationally intensive [Perkins 1998].

6.3.4: Streaming Stored Audio and Video

Let us conclude this section with a few words about streaming stored audio and video. Streaming stored audio/video applications also typically use sequence numbers, timestamps, and playout delay to alleviate or even eliminate the effects of network jitter. However, there is an important difference between real-time interactive audio/video and streaming stored audio/video. Specifically, streaming of stored audio/video can tolerate significantly larger delays. Indeed, when a user requests an audio/video clip, the user may find it acceptable to wait five seconds or more before playback begins. And most users can tolerate similar delays after interactive actions such as a temporal jump within the media stream. This greater tolerance for delay gives the application developer greater flexibility when designing stored media applications.

6.4: RTP

In the previous section we learned that the sender side of a multimedia application appends header fields to the audio/video chunks before passing them to the transport layer. These header fields include sequence numbers and timestamps. Since most multimedia networking applications can make use of sequence numbers and timestamps, it is convenient to have a standardized packet structure that includes fields for audio/video data, sequence number, and timestamp, as well as other potentially useful fields. RTP, defined in RFC 1889, is such a standard. RTP can be used for transporting common formats such as PCM or GSM for sound and MPEG1 and MPEG2 for video. It can also be used for transporting proprietary sound and video formats.

In this section we provide a short introduction to RTP and to its companion protocol, RTCP. We also discuss the role of RTP in the H.323 standard for real-time interactive audio and video conferencing. The reader is encouraged to visit Henning

Schulzrinne's RTP site [Schulzrinne 1999], which provides a wealth of information on the subject. Also, readers may want to visit the Free Phone site [Freephone 1999], which describes an Internet phone application that uses RTP.

6.4.1: RTP Basics

RTP typically runs on top of UDP. Specifically, chunks of audio or video data that are generated by the sending side of a multimedia application are encapsulated in RTP packets. Each RTP packet is in turn encapsulated in a UDP segment. Because RTP provides services (such as timestamps and sequence numbers) to the multimedia application, RTP can be viewed as a **sublayer of the transport layer,** as shown in Figure 6.9.

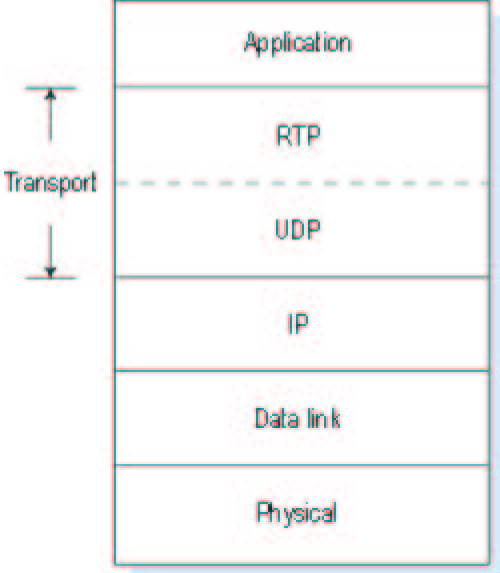

Figure 6.9: RTP can be viewed as a sublayer of the transport layer

From the application developer's perspective, however, RTP is not part of the transport layer but instead part of the application layer. This is because the developer must integrate RTP into the application. Specifically, for the sender side of the application, the developer must write application code that creates the RTP encapsulating packets. The application then sends the RTP packets into a UDP socket interface. Similarly, at the receiver side of the application, RTP packets enter the application through a UDP socket interface. The developer therefore must write code into the application that extracts the media chunks from the RTP packets. This is illustrated in Figure 6.10.

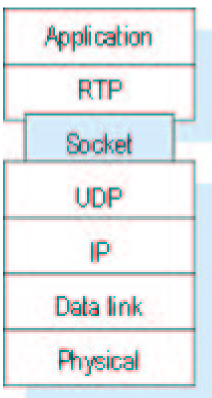

Figure 6.10: From a developer's perspective, RTP is part of the application layer

As an example, consider the use of RTP to transport voice. Suppose the voice source is PCM encoded (that is, sampled, quantized, and digitized) at 64 Kbps. Further suppose that the application collects the encoded data in 20 msec chunks, that is, 160 bytes in a chunk. The application precedes each chunk of the audio data with an **RTP header** that includes the type of audio encoding, a sequence number, and a timestamp. The audio chunk along with the RTP header form the **RTP packet.** The RTP packet is then sent into the UDP socket interface. At the receiver side, the application receives the RTP packet from its socket interface. The application extracts the audio chunk from the RTP packet, and uses the header fields of the RTP packet to properly decode and playback the audio chunk.

If an application incorporates RTP--instead of a proprietary scheme to provide payload type, sequence numbers, or timestamps--then the application will more easily interoperate with other networked multimedia applications. For example, if two different companies develop Internet phone software and they both incorporate RTP into their product, there may be some hope that a user using one of the Internet phone products will be able to communicate with a user using the other Internet phone product. At the end of this section we'll see that RTP has been incorporated into an important part of an Internet telephony standard.

It should be emphasized that RTP in itself does not provide any mechanism to ensure timely delivery of data or provide other quality of service guarantees; it does not even guarantee delivery of packets or prevent out-of-order delivery of packets. Indeed, RTP encapsulation is only seen at the end systems. Routers do not distinguish between IP datagrams that carry RTP packets and IP datagrams that don't.

RTP allows each source (for example, a camera or a microphone) to be assigned its own independent RTP stream of packets. For example, for a videoconference between two participants, four RTP streams could be opened--two streams for transmitting the

audio (one in each direction) and two streams for the video (again, one in each direction). However, many popular encoding techniques--including MPEG1 and MPEG2--bundle the audio and video into a single stream during the encoding process. When the audio and video are bundled by the encoder, then only one RTP stream is generated in each direction.
RTP packets are not limited to unicast applications. They can also be sent over one-to-many and many-to-many multicast trees. For a many-to-many multicast session, all of the session's senders and sources typically use the same multicast group for sending their RTP streams. RTP multicast streams belonging together, such as audio and video streams emanating from multiple senders in a videoconference application, belong to an **RTP session.**

6.4.2: RTP Packet Header Fields

As shown in Figure 6.11, the four main RTP packet header fields are the payload type, sequence number, timestamp, and the source identifier fields.

Payload type	Sequence number	Timestamp	Synchronization source identifier	Miscellaneous fields

Figure 6.11: RTP header fields

The payload type field in the RTP packet is seven bits long. For an audio stream, the payload type field is used to indicate the type of audio encoding (for example, PCM, adaptive delta modulation, linear predictive encoding) that is being used. If a sender decides to change the encoding in the middle of a session, the sender can inform the receiver of the change through this payload type field. The sender may want to change the encoding in order to increase the audio quality or to decrease the RTP stream bit rate. Table 6.1 lists some of the audio payload types currently supported by RTP.

Table 6.1: Some audio payload types supported by RTP

Payload Type Number	**Audio Format**	**Sampling Rate**	**Throughput**
0	PCM μ-law	8 KHz	64 Kbps
1	1016	8 KHz	4.8 Kbps
3	GSM	8 KHz	

13 Kbps

7
LPC
8 KHz
2.4 Kbps

9
G.722
8 KHz
48-64 Kbps

14
MPEG Audio
90 KHz
--

15
G.728
8 KHz
16 Kbps

For a video stream, the payload type is used to indicate the type of video encoding (for example, motion JPEG, MPEG1, MPEG2, H.261). Again, the sender can change video encoding on-the-fly during a session. Table 6.2 lists some of the video payload types currently supported by RTP.

Table 6.2: Some video payload types supported by RTP

Payload Type Number	**Video Format**
26	Motion JPEG
31	H.261
32	MPEG1 video
33	MPEG2 video

The other important fields are:

- *Sequence number field.* The sequence number field is 16 bits long. The sequence number increments by one for each RTP packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. For example, if the receiver side of the application receives a stream of RTP packets with a gap between sequence numbers 86 and 89, then the receiver knows that packets 87 and 88 are missing. The receiver can then attempt to conceal the lost

data.

- *Timestamp field.* The timestamp field is 32 bits long. It reflects the sampling instant of the first byte in the RTP data packet. As we saw in the previous section, the receiver can use timestamps in order to remove packet jitter introduced in the network and to provide synchronous playout at the receiver. The timestamp is derived from a sampling clock at the sender. As an example, for audio, the timestamp clock increments by one for each sampling period (for example, each 125 μsec for an 8 kHz sampling clock); if the audio application generates chunks consisting of 160 encoded samples, then the timestamp increases by 160 for each RTP packet when the source is active. The timestamp clock continues to increase at a constant rate even if the source is inactive.
- *Synchronization source identifier (SSRC).* The SSRC field is 32 bits long. It identifies the source of the RTP stream. Typically, each stream in an RTP session has a distinct SSRC. The SSRC is not the IP address of the sender, but instead a number that the source assigns randomly when the new stream is started. The probability that two streams get assigned the same SSRC is very small. Should this happen, the two sources pick a new SSRC value.

6.4.3: RTP Control Protocol (RTCP)

RFC 1889 also specifies RTCP, a protocol that a multimedia networking application can use in conjunction with RTP. As shown in the multicast scenario in Figure 6.12, RTCP packets are transmitted by each participant in an RTP session to all other participants in the session using IP multicast. For an RTP session typically there is a single multicast address and all RTP and RTCP packets belonging to the session use the multicast address. RTP and RTCP packets are distinguished from each other through the use of distinct port numbers.

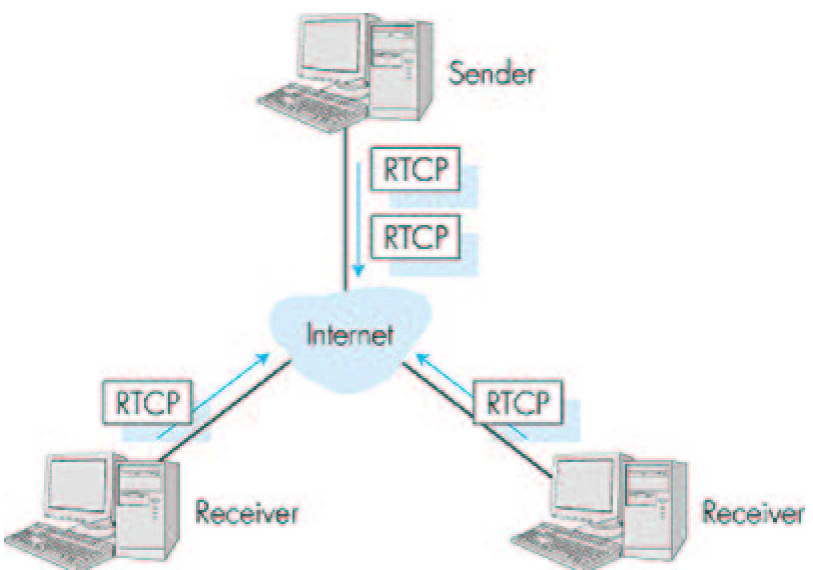

Figure 6.12: Both senders and receivers send RTCP messages

RTCP packets do not encapsulate chunks of audio or video. Instead, RTCP packets are sent periodically and contain sender and/or receiver reports that announce statistics that can be useful to the application. These statistics include number of packets sent, number of packets lost, and interarrival jitter. The RTP specification [RFC 1889] does not dictate what the application should do with this feedback information; this is up to the application developer. Senders can use the feedback information, for example, to modify their transmission rates. The feedback information can also be used for diagnostic purposes; for example, receivers can determine whether problems are local, regional, or global.

RTCP Packet Types

For each RTP stream that a receiver receives as part of a session, the receiver generates a reception report. The receiver aggregates its reception reports into a single RTCP packet. The packet is then sent into the multicast tree that connects together all the sessions's participants. The reception report includes several fields, the most important of which are listed below.

- The SSRC of the RTP stream for which the reception report is being generated.
- The fraction of packets lost within the RTP stream. Each receiver calculates the number of RTP packets lost divided by the number of RTP packets sent as part of the stream. If a sender receives reception reports indicating that the receivers are receiving only a small fraction of the sender's transmitted packets, it can switch to a lower encoding rate, with the aim of decreasing network congestion and improving the reception rate.

- The last sequence number received in the stream of RTP packets.
- The interarrival jitter, which is calculated as the average interarrival time between successive packets in the RTP stream.

For each RTP stream that a sender is transmitting, the sender creates and transmits RTCP sender report packets. These packets include information about the RTP stream, including:

- The SSRC of the RTP stream.
- The timestamp and wall clock time of the most recently generated RTP packet in the stream.
- The number of packets sent in the stream.
- The number of bytes sent in the stream.

Sender reports can be used to synchronize different media streams within an RTP session. For example, consider a videoconferencing application for which each sender generates two independent RTP streams, one for video and one for audio. The timestamps in these RTP packets are tied to the video and audio sampling clocks, and are not tied to the *wall clock time* (that is, to real time). Each RTCP sender report contains, for the most recently generated packet in the associated RTP stream, the timestamp of the RTP packet and the wall clock time for when the packet was created. Thus the RTCP sender report packets associate the sampling clock to the real-time clock. Receivers can use this association in RTCP sender reports to synchronize the playout of audio and video.

For each RTP stream that a sender is transmitting, the sender also creates and transmits source description packets. These packets contain information about the source, such as e-mail address of the sender, the sender's name, and the application that generates the RTP stream. It also includes the SSRC of the associated RTP stream. These packets provide a mapping between the source identifier (that is, the SSRC) and the user/host name.

RTCP packets are stackable, that is, receiver reception reports, sender reports, and source descriptors can be concatenated into a single packet. The resulting packet is then encapsulated into a UDP segment and forwarded into the multicast tree.

RTCP Bandwidth Scaling

The astute reader will have observed that RTCP has a potential scaling problem. Consider, for example, an RTP session that consists of one sender and a large number of receivers. If each of the receivers periodically generates RTCP packets, then the

aggregate transmission rate of RTCP packets can greatly exceed the rate of RTP packets sent by the sender. Observe that the amount of RTP traffic sent into the multicast tree does not change as the number of receivers increases, whereas the amount of RTCP traffic grows linearly with the number of receivers. To solve this scaling problem, RTCP modifies the rate at which a participant sends RTCP packets into the multicast tree as a function of the number of participants in the session. Also, since each participant sends control packets to everyone else, each participant can estimate the total number of participants in the session [Friedman 1999].

RTCP attempts to limit its traffic to 5% of the session bandwidth. For example, suppose there is one sender, which is sending video at a rate of 2 Mbps. Then RTCP attempts to limit its traffic to 5% of 2 Mbps, or 100 Kbps, as follows. The protocol gives 75% of this rate, or 75 Kbps, to the receivers; it gives the remaining 25% of the rate, or 25 Kbps, to the sender. The 75 Kbps devoted to the receivers is equally shared among the receivers. Thus, if there are *R* receivers, then each receiver gets to send RTCP traffic at a rate of 75/*R* Kbps and the sender gets to send RTCP traffic at a rate of 25 Kbps. A participant (a sender or receiver) determines the RTCP packet transmission period by dynamically calculating the average RTCP packet size (across the entire session) and dividing the average RTCP packet size by its allocated rate. In summary, the period for transmitting RTCP packets for a sender is

$$T = \frac{\text{number of senders}}{.25 \cdot .05 \text{ session bandwidth}} (\text{avg. RTCP packet size})$$

And the period for transmitting RTCP packets for a receiver is

$$T = \frac{\text{number of receivers}}{.75 \cdot .05 \text{ session bandwidth}} (\text{avg. RTCP packet size})$$

6.4.4: H.323

H.323 is a standard for real-time audio and video conferencing among end systems on the Internet. As shown in Figure 6.13, the standard also covers how end systems attached to the Internet communicate with telephones attached to ordinary circuit-switched telephone networks. In principle, if manufacturers of Internet telephony and video conferencing all conform to H.323, then all their products should be able to interoperate, and should be able to communicate with ordinary telephones. We discuss H.323 in this section, as it provides an application context for RTP. Indeed, we’ll see below that RTP is an integral part of the H.323 standard.

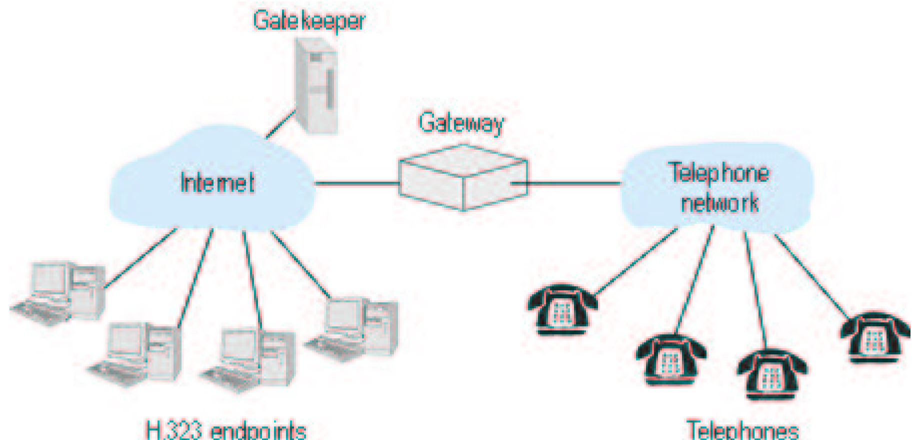

Figure 6.13: H.323 end systems attached to the Internet can communicate with telephones attached to a circuit-switched telephone network

H.323 **end points** (terminals) can be standalone devices (for example, Web phones and Web TVs) or applications in a PC (for example, Internet phone or video conferencing software). H.323 equipment also includes **gateways** and **gatekeepers.** Gateways permit communication among H.323 end points and ordinary telephones in a circuit-switched telephone network. Gatekeepers, which are optional, provide address translation, authorization, bandwidth management, accounting, and billing. We will discuss gatekeepers in more detail at the end of this section.

The H.323 standard is an umbrella specification that includes:

- A specification for how endpoints negotiate common audio/video encodings. Because H.323 supports a variety of audio and video encoding standards, a protocol is needed to allow the communicating endpoints to agree on a common encoding.
- A specification for how audio and video chunks are encapsulated and sent over network. As you may have guessed, this is where RTP comes into the picture.
- A specification for how endpoints communicate with their respective gatekeepers.
- A specification for how Internet phones communicate through a gateway with ordinary phones in the public circuit-switched telephone network.

Figure 6.14 shows the H.323 protocol architecture.

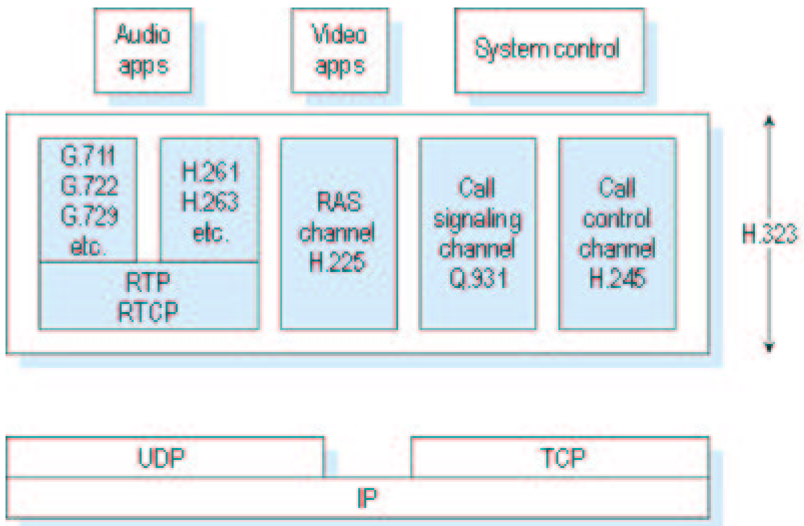

Figure 6.14: H.323 protocol architecture

Minimally, each H.323 endpoint *must* support the G.711 speech compression standard. G.711 uses PCM to generate digitized speech at either 56 Kbps or 64 Kbps. Although H.323 requires every endpoint to be voice capable (through G.711), video capabilities are optional. Because video support is optional, manufacturers of terminals can sell simpler speech terminals as well as more complex terminals that support both audio and video. As shown in Figure 6.14, H.323 also requires that all H.323 end points use the following protocols:

- *RTP.* The sending side of an endpoint encapsulates all media chunks within RTP packets. The sending side then passes the RTP packets to UDP.
- *H.245.* An "out-of-band" control protocol for controlling media between H.323 endpoints. This protocol is used to negotiate a common audio or video compression standard that will be employed by all the participating endpoints in a session.
- *Q.931.* A signaling protocol for establishing and terminating calls. This protocol provides traditional telephone functionality (for example, dial tones and ringing) to H.323 endpoints and equipment.
- *RAS (Registration/Admission/Status) channel protocol.* A protocol that allows end points to communicate with a gatekeeper (if a gatekeeper is present).

Audio and Video Compression

The H.323 standard supports a specific set of audio and video compression techniques. Let's first consider audio. As we just mentioned, all H.323 end points must support the G.711 speech

encoding standard. Because of this requirement, two H.323 end points will always be able to default to G.711 and communicate. But H.323 allows terminals to support a variety of other speech compression standards, including G.723.1, G.722, G.728, and G.729. Many of these standards compress speech to rates that are suitable for 28.8 Kbps dial-up modems. For example, G.723.1 compresses speech to either 5.3 Kbps or 6.3 Kbps, with sound quality that is comparable to G.711.

As we mentioned earlier, video capabilities for an H.323 endpoint are optional. However, if an endpoint does support video, then it must (at the very least) support the QCIF H.261 (176 x144 pixels) video standard. A video-capable endpoint may optionally support other H.261 schemes, including CIF, 4CIF, 16CIF, and the H.263 standard. As the H.323 standard evolves, it will likely support a longer list of audio and video compression schemes.

H.323 Channels

When an end point participates in an H.323 session, it maintains several channels, as shown in Figure 6.15. Examining Figure 6.15, we see that an end point can support many simultaneous RTP media channels. For each media type, there will typically be one send media channel and one receive media channel; thus, if audio and video are sent in separate RTP streams, there will typically be four media channels. Accompanying the RTP media channels, there is one RTCP media control channel, as discussed in Section 6.4.3. All of the RTP and RTCP channels run over UDP. In addition to the RTP/RTCP channels, two other channels are required: the call control channel and the call signaling channel. The H.245 call control channel is a TCP connection that carries H.245 control messages. Its principal tasks are (1) opening and closing media channels, and (2) capability exchange, that is, before sending media, endpoints agree on an encoding algorithm. H.245, being a control protocol for real-time interactive applications, is analogous to RTSP, the control protocol for streaming of stored multimedia that we studied in Section 6.2.3. Finally, the Q.931 call signaling channel provides classical telephone functionality, such as dial tone and ringing.

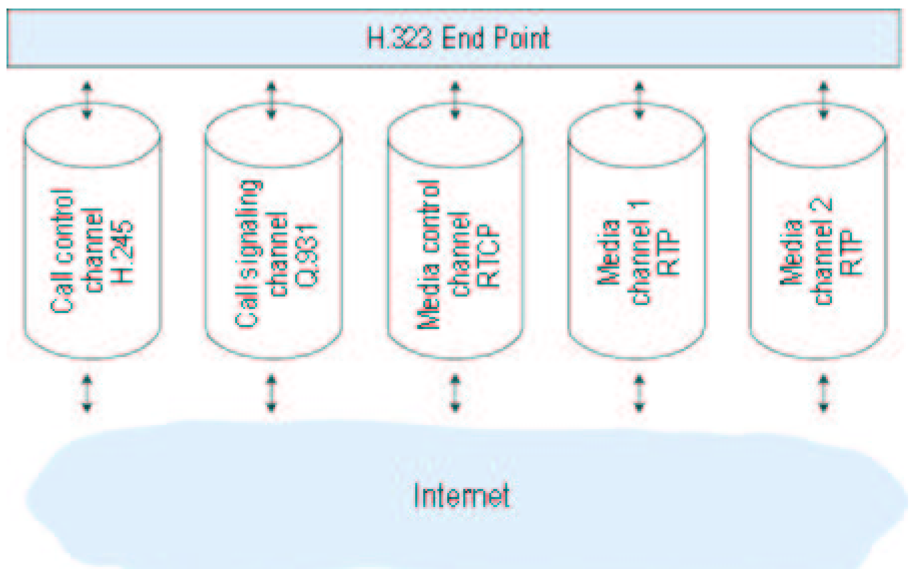

Figure 6.15: H.323 channels

Gatekeepers

The gatekeeper is an optional H.323 device. Each gatekeeper is responsible for an H.323 zone. A typical deployment scenario is shown in Figure 6.16. In this scenario, the H.323 terminals and the gatekeeper are all attached to the same LAN, and the H.323 zone is the LAN itself. If a zone has a gatekeeper, then all H.323 terminals in the zone are required to communicate with it using the RAS protocol, which runs over TCP. Address translation is one of the more important gatekeeper services. Each terminal can have an alias address, such as the name of the person at the terminal, the e-mail address of the person at the terminal, and so on. The gateway translates these alias addresses to IP addresses. This address translation service is similar to the DNS service, covered in Section 2.5. Another gatekeeper service is bandwidth management: The gatekeeper can limit the number of simultaneous real-time conferences in order to save some bandwidth for other applications running over the LAN. Optionally, H.323 calls can be routed through gatekeeper, which is useful for billing.

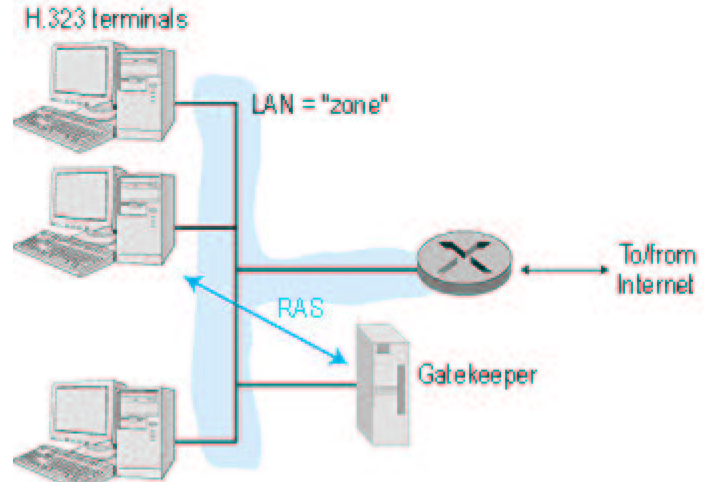

Figure 6.16: H.323 terminals and gatekeeper on the same LAN

The H.323 terminal must register itself with the gatekeeper in its zone. When the H.323 application is invoked at the terminal, the terminal uses RAS to send its IP address and alias (provided by user) to the gatekeeper. If the gatekeeper is present in a zone, each terminal in the zone must contact the gatekeeper to ask permission to make a call. Once it has permission, the terminal can send the gatekeeper an e-mail address, alias string, or phone extension for the terminal it wants to call, which may be in another zone. If necessary, a gatekeeper will poll other gatekeepers in other zones to resolve an IP address.

An excellent tutorial on H.323 is provided by [WebProForum 1999]. The reader is also encouraged to see [Rosenberg 1999] for an alternative architecture to H.323 for providing telephone service in the Internet.

Online Book

6.5: Beyond Best-Effort

In previous sections we learned how sequence numbers, timestamps, FEC, RTP, and H.323 can be used by multimedia applications in today's Internet. But are these techniques alone enough to support reliable and robust multimedia applications, for example, an IP telephony service that is equivalent to a service in today's telephone network? Before answering this question, let us recall again that today's Internet provides a best-effort service to all of its applications, that is, does not make any promises about the quality of service (QoS) an application will receive. An application will receive whatever level of performance (for example, end-to-end packet

delay and loss) that the network is able to provide at that moment. Recall also that today's public Internet does not allow delay-sensitive multimedia applications to request any special treatment. All packets are treated equally at the routers, including delay-sensitive audio and video packets. Given that all packets are treated equally, all that's required to ruin the quality of an ongoing IP telephone call is enough interfering traffic (that is, network congestion) to noticeably increase the delay and loss seen by an IP telephone call.

In this section, we will identify *new* architectural components that can be added to the Internet architecture to shield an application from such congestion and thus make high-quality networked multimedia applications a reality. Many of the issues that we will discuss in this, and the remaining sections of this chapter, are currently under active discussion in the IETF Diffserv, Intserv, and RSVP working groups.

Figure 6.17 shows a simple network scenario we'll use to illustrate the most important architectural components that have been proposed for the Internet in order to provide explicit support for the QoS needs of multimedia applications. Suppose that two application packet flows originate on hosts H1 and H2 on one LAN and are destined for hosts H3 and H4 on another LAN. The routers on the two LANs are connected by a 1.5 Mbps link. Let's assume the LAN speeds are significantly higher than 1.5 Mbps, and focus on the output queue of router R1; it is here that packet delay and packet loss will occur if the aggregate sending rate of the H1 and H2 exceeds 1.5 Mbps. Let's now consider several scenarios, each of which will provide us with important insight into the underlying principles for providing QoS guarantees to multimedia applications.

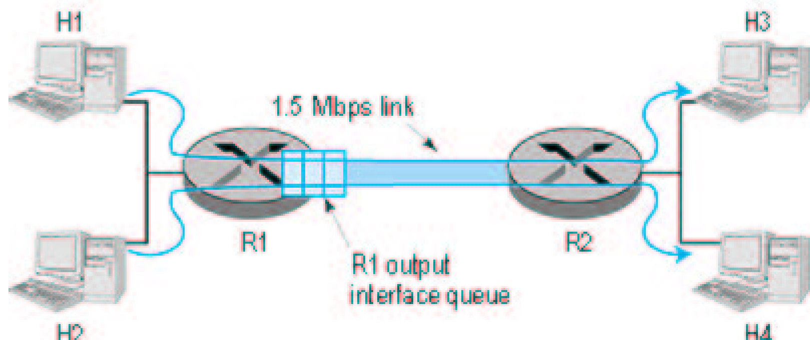

Figure 6.17: A simple network with two applications

6.5.1: Scenario 1: A 1 Mbps Audio Application and an FTP Transfer

Scenario 1 is illustrated in Figure 6.18. Here, a 1 Mbps audio application (for example, a CD-quality audio call) shares the 1.5 Mbps link between R1 and R2 with an FTP application that is transferring a file from H2 to H4. In the best-effort Internet, the audio and FTP packets are mixed in the output queue at R1 and (typically) transmitted in a first-in-first-out (FIFO) order. In

this scenario, a burst of packets from the FTP source could potentially fill up the queue, causing IP audio packets to be excessively delayed or lost due to buffer overflow at R1. How should we solve this potential problem? Given that the FTP application does not have time constraints, our intuition might be to give strict priority to audio packets at R1. Under a strict priority scheduling discipline, an audio packet in the R1 output buffer would always be transmitted before any FTP packet in the R1 output buffer. The link from R1 to R2 would look like a dedicated link of 1.5 Mbps to the audio traffic, with FTP traffic using the R1-to-R2 link only when no audio traffic is queued.

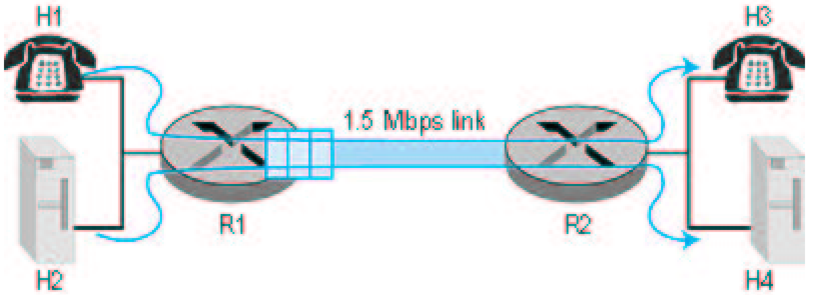

Figure 6.18: Competing audio and FTP applications

In order for R1 to distinguish between the audio and FTP packets in its queue, each packet must be marked as belonging to one of these two "classes" of traffic. Recall from Section 4.7, that this was the original goal of the Type-of-Service (ToS) field in IPv4. As obvious as this might seem, this then is our first principle underlying the provision of quality-of-service guarantees:

> **Principle 1:** Packet marking allows a router to distinguish among packets belonging to different classes of traffic.

6.5.2: Scenario 2: A 1 Mbps Audio Application and a High Priority FTP Transfer

Our second scenario is only slightly different from scenario 1. Suppose now that the FTP user has purchased "platinum service" (that is, high-priced) Internet access from its ISP, while the audio user has purchased cheap, low-budget Internet service that costs only a minuscule fraction of platinum service. Should the cheap user's audio packets be given priority over FTP packets in this case? Arguably not. In this case, it would seem more reasonable to distinguish packets on the basis of the sender's IP address. More generally, we see that it is necessary for a router to *classify* packets according to some criteria. This then calls for a slight modification to principle 1:

> **Principle 1 (modified):** Packet classification allows a router to distinguish among packets belonging to different classes of traffic.

Explicit packet marking is one way in which packets may be distinguished. However, the marking carried by a packet does not, by itself, mandate that

the packet will receive a given quality of service. Marking is but one *mechanism* for distinguishing packets. The manner in which a router distinguishes among packets by treating them differently is a *policy* decision.

6.5.3: Scenario 3: A Misbehaving Audio Application and an FTP Transfer

Suppose now that somehow (by use of mechanisms that we will study in subsequent sections), the router knows it should give priority to packets from the 1 Mbps audio application. Since the outgoing link speed is 1.5 Mbps, even though the FTP packets receive lower priority, they will still, on average, receive 0.5 Mbps of transmission service. But what happens if the audio application starts sending packets at a rate of 1.5 Mbps or higher (either maliciously or due to an error in the application)? In this case, the FTP packets will starve, that is, will not receive any service on the R1-to-R2 link. Similar problems would occur if multiple applications (for example, multiple audio calls), all with the same priority, were sharing a link's bandwidth; one noncompliant flow could degrade and ruin the performance of the other flows. Ideally, one wants a degree of *isolation* among flows, in order to protect one flow from another misbehaving flow. This, then, is a second underlying principle the provision of QoS guarantees.

> **Principle 2:** It is desirable to provide a degree of isolation among traffic flows, so that one flow is not adversely affected by another misbehaving flow.

In the following section, we will examine several specific mechanisms for providing this isolation among flows. We note here that two broad approaches can be taken. First, it is possible to "police" traffic flows, as shown in Figure 6.19. If a traffic flow must meet certain criteria (for example, that the audio flow not exceed a peak rate of 1 Mbps), then a policing mechanism can be put into place to ensure that this criteria is indeed observed. If the policed application misbehaves, the policing mechanism will take some action (for example, drop or delay packets that are in violation of the criteria) so that the traffic actually entering the network conforms to the criteria. The leaky bucket mechanism that we examine in the following section is perhaps the most widely used policing mechanism. In Figure 6.19, the packet classification and marking mechanism (Principle 1) and the policing mechanism (Principle 2) are co-located at the "edge" of the network, either in the end system, or at an edge router.

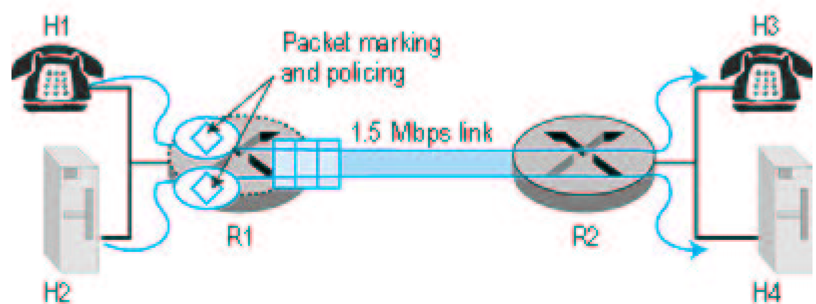

Figure 6.19: Policing (and marking) the audio and FTP traffic flows

An alternate approach for providing isolation among traffic flows is for the link-level packet scheduling mechanism to explicitly allocate a fixed amount of link bandwidth to each application flow. For example, the audio flow could be allocated 1Mbps at R1, and the FTP flow could be allocated 0.5 Mbps. In this case, the audio and FTP flows see a logical link with capacity 1.0 and 0.5 Mbps, respectively, as shown in Figure 6.20.

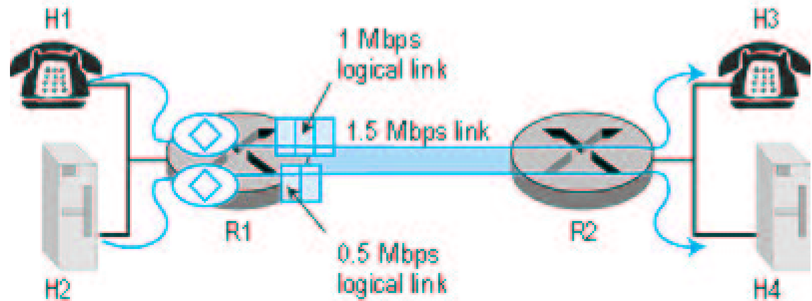

Figure 6.20: Logical isolation of audio and FTP application flows

With strict enforcement of the link-level allocation of bandwidth, a flow can use only the amount of bandwidth that has been allocated; in particular, it cannot utilize bandwidth that is not currently being used by the other applications. For example, if the audio flow goes silent (for example, if the speaker pauses and generates no audio packets), the FTP flow would still not be able to transmit more than 0.5 Mbps over the R1-to-R2 link, even though the audio flow's 1 Mbps bandwidth allocation is not being used at that moment. It is therefore desirable to use bandwidth as efficiently as possible, allowing one flow to use another flow's unused bandwidth at any given point in time. This is the third principle underlying the provision of quality of service:

> **Principle 3:** While providing isolation among flows, it is desirable to use resources (for example, link bandwidth and buffers) as efficiently as possible.

6.5.4: Scenario 4: Two 1 Mbps Audio Applications over an Overloaded 1.5 Mbps Link

In our final scenario, two 1-Mbps audio connections transmit their packets over the 1.5 Mbps link, as shown in Figure 6.21. The combined data rate of the two flows (2 Mbps) exceeds the link capacity. Even with classification and marking (Principle 1), isolation of flows (Principle 2), and sharing of

unused bandwidth (Principle 3), of which there is none, this is clearly a losing proposition. There is simply not enough bandwidth to accommodate the applications' needs. If the two applications equally share the bandwidth, each would receive only 0.75 Mbps. Looked at another way, each application would lose 25% of its transmitted packets. This is such an unacceptably low quality of service that the application is completely unusable; there's no need even to transmit any audio packets in the first place.

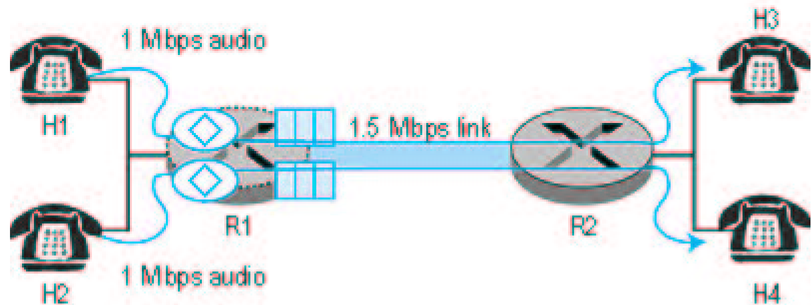

Figure 6.21: Two competing audio applications overloading the R1-to-R2 link

For a flow that needs a minimum quality of service in order to be considered "usable," the network should either allow the flow to use the network or else *block* the flow from using the network. The telephone network is an example of a network that performs such call blocking--if the required resources (an end-to-end circuit, in the case of the telephone network) cannot be allocated to the call, the call is blocked (prevented from entering the network) and a busy signal is returned to the user. In our example above, there is no gain in allowing a flow into the network if it will not receive a sufficient QoS to be considered "usable." Indeed, there is a *cost* to admitting a flow that does not receive its needed QoS, as network resources are being used to support a flow that provides no utility to the end user.

Implicit with the need to provide a guaranteed QoS to a flow is the need for the flow to declare its QoS requirements. This process of having a flow declare its QoS requirement, and then having the network either accept the flow (at the required QoS) or block the flow is referred to as the **call admission** process. The need for call admission is the fourth underlying principle in the provision of QoS guarantees:

> **Principle 4:** A call admission process is needed in which flows declare their QoS requirements and are then either admitted to the network (at the required QoS) or blocked from the network (if the required QoS cannot be provided by the network).

In our discussion above, we have identified four basic principles in providing QoS guarantees for multimedia applications. These principles are illustrated in Figure 6.22. In the following section, we consider various *mechanisms* for implementing these principles. In the sections following that, we examine proposed Internet service models for providing QoS

guarantees.

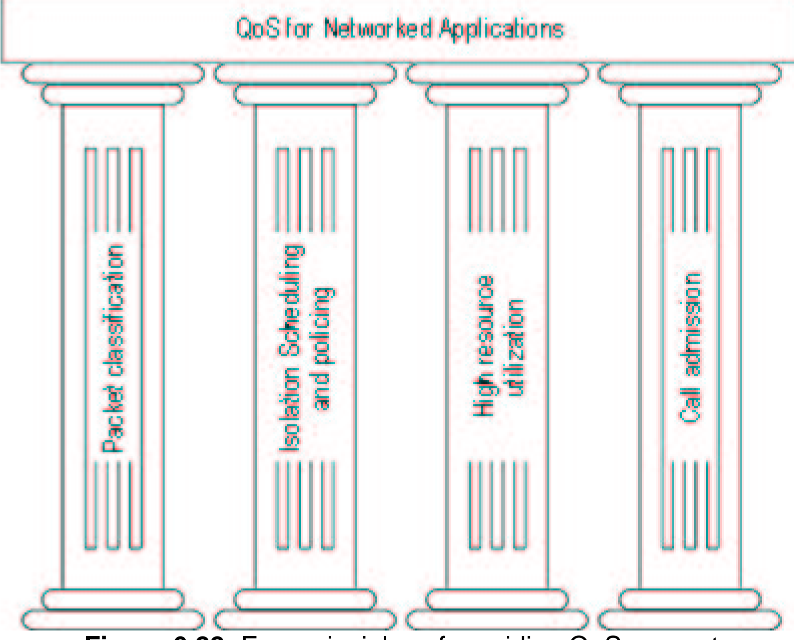

Figure 6.22: Four principles of providing QoS support

© 2000-2001 by Addison Wesley Longman
A division of Pearson Education

Online Book

6.6: Scheduling and Policing Mechanisms

In the previous section, we identified the important underlying principles in providing quality-of-service (QoS) guarantees to networked multimedia applications. In this section, we will examine various mechanisms that are used to provide these QoS guarantees.

6.6.1: Scheduling Mechanisms

Recall from our discussion in Section 1.6 and Section 4.6, that packets belonging to various network flows are multiplexed together and queued for transmission at the output buffers associated with a link. The manner in which queued packets are selected for transmission on the link is known as the **link scheduling discipline.** We saw in the previous section that the link scheduling discipline plays an important role in providing QoS

guarantees. Let us now consider several of the most important link scheduling disciplines in more detail.

First-In-First-Out (FIFO)

Figure 6.23 shows the queuing model abstractions for the First-in-First-Out (FIFO) link scheduling discipline. Packets arriving at the link output queue are queued for transmission if the link is currently busy transmitting another packet. If there is not sufficient buffering space to hold the arriving packet, the queue's **packet discarding policy** then determines whether the packet will be dropped ("lost") or whether other packets will be removed from the queue to make space for the arriving packet. In our discussion below we will ignore packet discard. When a packet is completely transmitted over the outgoing link (that is, receives service) it is removed from the queue.

Figure 6.23: FIFO queuing abstraction

The FIFO scheduling discipline (also known as First-Come-First-Served--FCFS) selects packets for link transmission in the same order in which they arrived at the output link queue. We're all familiar with FIFO queuing from bus stops (particularly in England, where queuing seems to have been perfected) or other service centers, where arriving customers join the back of the single waiting line, remain in order, and are then served when they reach the front of the line.

Figure 6.24 shows an example of the FIFO queue in operation. Packet arrivals are indicated by numbered arrows above the upper timeline, with the number indicating the order in which the packet arrived. Individual packet departures are shown below the lower timeline. The time that a packet spends in service (being transmitted) is indicated by the shaded rectangle between the two timelines. Because of the FIFO discipline, packets leave in the same order in which they arrived. Note that after the departure of packet 4, the link remains idle (since packets 1 through 4 have been transmitted and removed from the queue) until the arrival of packet 5.

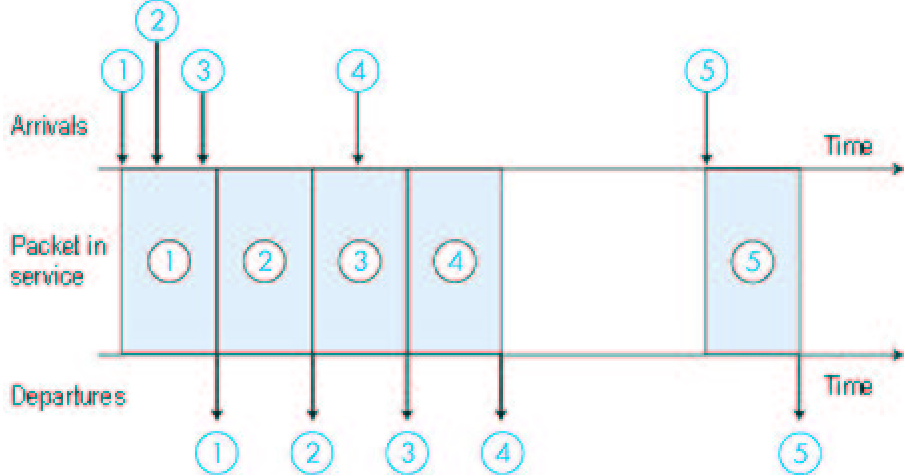

Figure 6.24: The FIFO queue in operation

Priority Queuing

Under **priority queuing,** packets arriving at the output link are classified into one of two or more priority classes at the output queue, as shown in Figure 6.25. As discussed in the previous section, a packet's priority class may depend on an explicit marking that it carries in its packet header (for example, the value of the Type of Service (ToS) bits in an IPv4 packet), its source or destination IP address, its destination port number, or other criteria. Each priority class typically has its own queue. When choosing a packet to transmit, the priority queuing discipline will transmit a packet from the highest priority class that has a nonempty queue (that is, has packets waiting for transmission). The choice among packets *in the same priority class* is typically done in a FIFO manner.

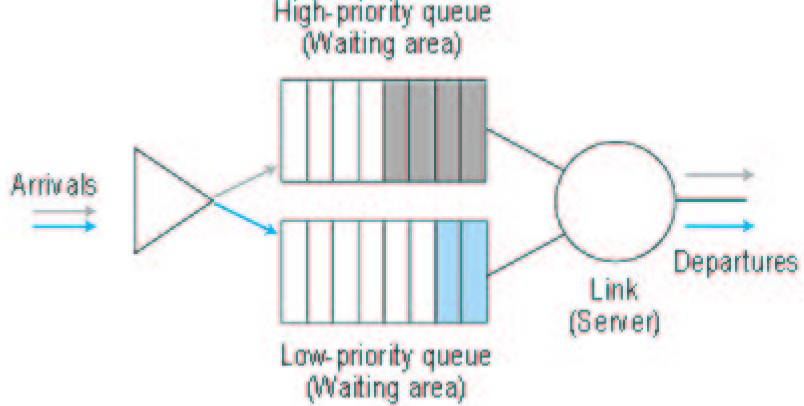

Figure 6.25: Priority queuing model

Figure 6.26 illustrates the operation of a priority queue with two priority classes. Packets 1, 3, and 4 belong to the high-priority class and packets 2 and 5 belong to the low-priority class. Packet 1 arrives and, finding the link idle, begins transmission. During the transmission of packet 1, packets 2 and 3 arrive and are queued in the low- and high-priority queues, respectively. After the

transmission of packet 1, packet 3 (a high-priority packet) is selected for transmission over packet 2 (which, even though it arrived earlier, is a low-priority packet). At the end of the transmission of packet 3, packet 2 then begins transmission. Packet 4 (a high-priority packet) arrives during the transmission of packet 3 (a low-priority packet). Under a so-called non-preemptive priority queuing discipline, the transmission of a packet is not interrupted once it has begun. In this case, packet 4 queues for transmission and begins being transmitted after the transmission of packet 2 is completed.

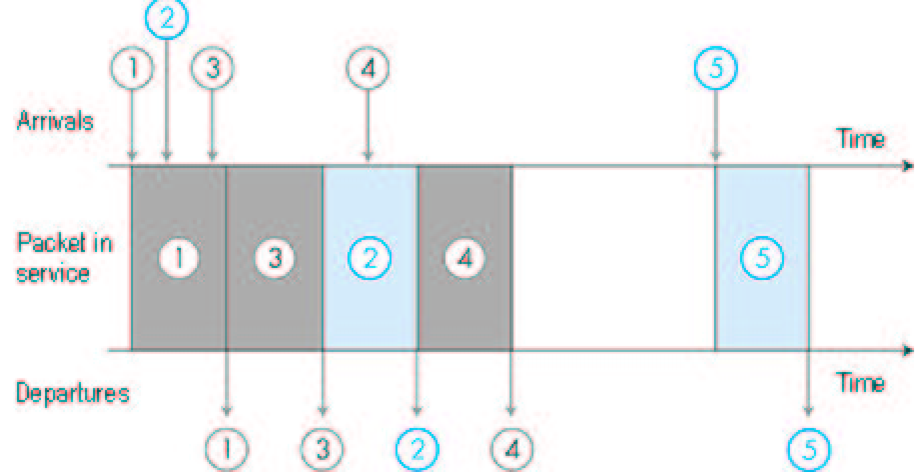

Figure 6.26: Operation of teh priority queue

Round Robin and Weighted Fair Queuing (WFQ)

Under the **round robin queuing discipline,** packets are again sorted into classes, as with priority queuing. However, rather than there being a strict priority of service among classes, a round robin scheduler alternates service among the classes. In the simplest form of round robin scheduling, a class 1 packet is transmitted, followed by a class 2 packet, followed by a class 1 packet, followed by a class 2 packet, and so on. A so-called work-conserving queuing discipline will never allow the link to remain idle whenever there are packets (of any class) queued for transmission. A **work-conserving round robin discipline** that looks for a packet of a given class but finds none will immediately check the next class in the round robin sequence.

Figure 6.27 illustrates the operation of a two-class round robin queue. In this example, packets 1, 2, and 4 belong to class 1, and packets 3 and 5 belong to the second class. Packet 1 begins transmission immediately upon arrival at the output queue. Packets 2 and 3 arrive during the transmission of packet 1 and thus queue for transmission. After the transmission of packet 1, the link scheduler looks for a class-2 packet and thus transmits packet 3. After the transmission of packet 3, the scheduler looks for a class-1 packet and thus transmits packet 2. After the transmission of packet 2, packet 4 is the only queued packet; it is thus

transmitted immediately after packet 2.

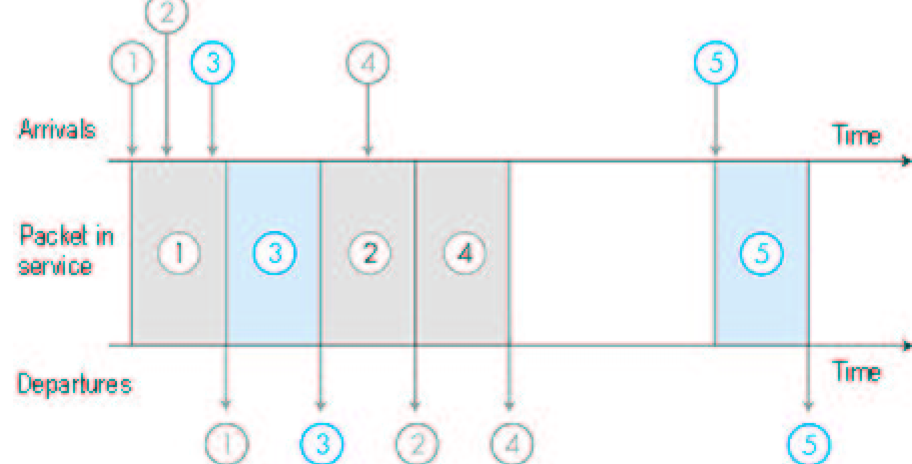

Figure 6.27: Operation of the two-class round robin queue

A generalized abstraction of round robin queuing that has found considerable use in QoS architectures is the so-called **weighted fair queuing (WFQ) discipline** [Demers 1990; Parekh 1993]. WFQ is illustrated in Figure 6.28. Arriving packets are again classified and queued in the appropriate per-class waiting area. As in round robin scheduling, a WFQ scheduler will again serve classes in a circular manner--first serving class 1, then serving class 2, then serving class 3, and then (assuming there are three classes) repeating the service pattern. WFQ is also a work-conserving queuing discipline and thus will immediately move on to the next class in the service sequence upon finding an empty class queue.

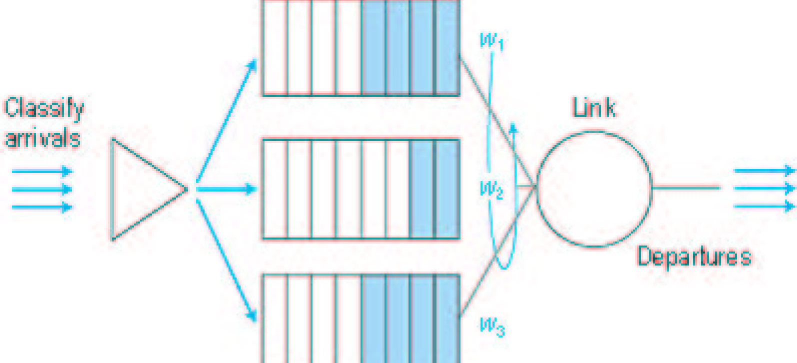

Figure 6.28: Weighted Fair Queuing (WFQ)

WFQ differs from round robin in that each class may receive a *differential* amount of service in any interval of time. Specifically, each class, *i*, is assigned a weight, w_i. Under WFQ, during any interval of time during which there are class *i* packets to send, class *i* will then be guaranteed to receive a fraction of service equal to $w_i/(\Sigma w_j)$, where the sum in the denominator is taken over all classes that also have packets queued for transmission. In the worst case, even if all classes have queued packets, class *i* will

still be guaranteed to receive a fraction $w_i/(\Sigma w_j)$ of the bandwidth. Thus, for a link with transmission rate R, class i will always achieve a throughput of at least $R \cdot w_i/(\Sigma w_j)$. Our description of WFQ has been an idealized one, as we have not considered the fact that packets are discrete units of data and a packet's transmission will not be interrupted to begin transmission of another packet; [Demers 1990] and [Parekh 1993] discuss this packetization issue. As we will see in the following sections, WFQ plays a central role in QoS architectures. It is also available in today's router products [Cisco QoS 1997]. (Intranets that use WFQ-capable routers can therefore provide QoS to their internal flows.)

6.6.2: Policing: The Leaky Bucket

In Section 6.5, we also identified **policing,** the regulation of the rate at which a flow is allowed to inject packets into the network, as one of the cornerstones of any QoS architecture. But what aspects of a flow's packet rate should be policed? We can identify three important policing criteria, each differing from the other according to the time scale over which the packet flow is policed:

- *Average rate.* The network may wish to limit the long-term average rate (packets per time interval) at which a flow's packets can be sent into the network. A crucial issue here is the interval of time over which the average rate will be policed. A flow whose average rate is limited to 100 packets per second is more constrained than a source that is limited to 6,000 packets per minute, even though both have the same average rate over a long enough interval of time. For example, the latter constraint would allow a flow to send 1,000 packets in a given second-long interval of time (subject to the constraint that the rate be less than 6,000 packets over a minute-long interval containing these 1,000 packets), while the former constraint would disallow this sending behavior.

- *Peak rate.* While the average rate-constraint limits the amount of traffic that can be sent into the network over a relatively long period of time, a peak-rate constraint limits the maximum number of packets that can be sent over a shorter period of time. Using our example above, the network may police a flow at an average rate of 6,000 packets per minute, while limiting the flow's peak rate to 1,500 packets per second.

- *Burst size.* The network may also wish to limit the maximum number of packets (the "burst" of packets) that can be sent into the network over an extremely short interval of time. In the limit, as the interval length approaches zero, the burst

size limits the number of packets that can be instantaneously sent into the network. Even though it is physically impossible to instantaneously send multiple packets into the network (after all, every link has a physical transmission rate that cannot be exceeded!), the abstraction of a maximum burst size is a useful one.

The leaky bucket mechanism is an abstraction that can be used to characterize these policing limits. As shown in Figure 6.29, a leaky bucket consists of a bucket that can hold up to *b* tokens. Tokens are added to this bucket as follows. New tokens, which may potentially be added to the bucket, are always being generated at a rate of *r* tokens per second. (We assume here for simplicity that the unit of time is a second.) If the bucket is filled with less than *b* tokens when a token is generated, the newly generated token is added to the bucket; otherwise the newly generated token is ignored, and the token bucket remains full with *b* tokens.

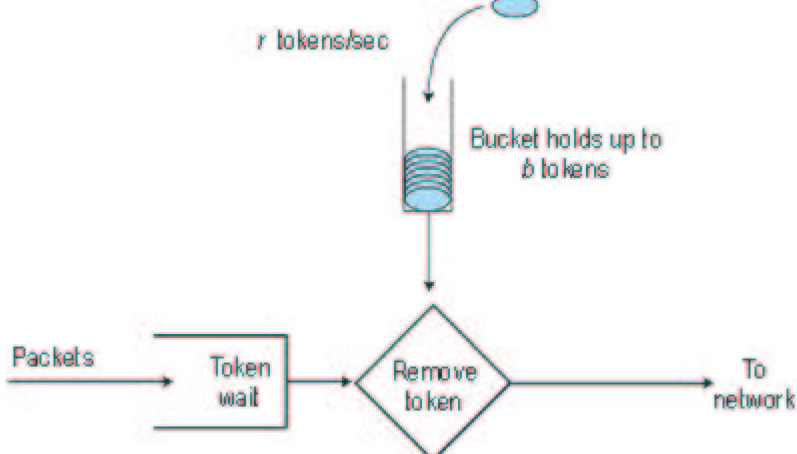

Figure 6.29: The Leaky Bucket Policer

Let us now consider how the leaky bucket can be used to police a packet flow. Suppose that before a packet is transmitted into the network, it must first remove a token from the token bucket. If the token bucket is empty, the packet must wait for a token. (An alternative is for the packet to be dropped, although we will not consider that option here.) Let us now consider how this behavior polices a traffic flow. Because there can be at most *b* tokens in the bucket, the maximum burst size for a leaky-bucket-policed flow is *b* packets. Furthermore, because the token generation rate is *r*, the maximum number of packets that can enter the network of *any* interval of time of length *t* is $rt + b$. Thus, the token generation rate, *r*, serves to limit the long-term average rate at which the packet can enter the network. It is also possible to use leaky buckets (specifically, two leaky buckets in series) to police a flow's peak rate in addition to the long-term average rate; see the homework problems at the end of this chapter.

Leaky Bucket + Weighted Fair Queuing Provides Provable

Maximum Delay in a Queue

In Sections 6.7 and 6.9 we will examine the so-called Intserv and Diffserv approaches for providing quality of service in the Internet. We will see that both leaky bucket policing and WFQ scheduling can play an important role. Let us thus close this section by considering a router's output that multiplexes *n* flows, each policed by a leaky bucket with parameters b_i and r_i, $i = 1, \ldots, n$, using WFQ scheduling. We use the term "flow" here loosely to refer to the set of packets that are not distinguished from each other by the scheduler. In practice, a flow might be comprised of traffic from a single end-to-end connection (as in Intserv) or a collection of many such connections (as in Diffserv), see Figure 6.30.

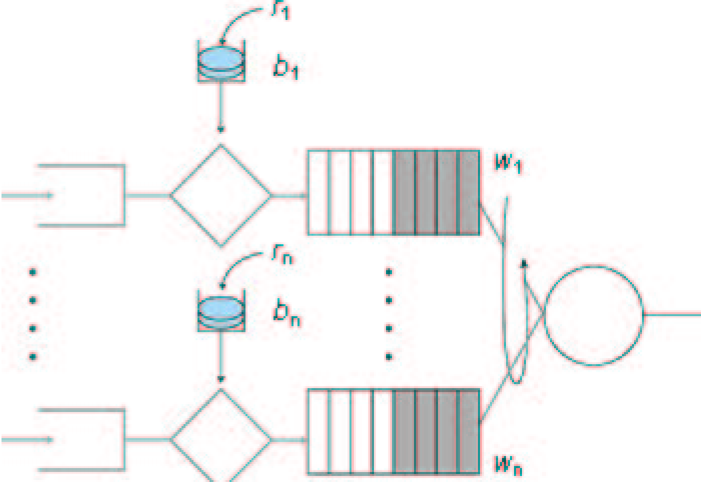

Figure 6.30: *n* multiplexed leaky bucket flows with WFQ scheduling

Recall from our discussion of WFQ that each flow, *i*, is guaranteed to receive a share of the link bandwidth equal to at least $R \cdot w_i/(\Sigma w_j)$, where *R* is the transmission rate of the link in packets/sec. What then is the maximum delay that a packet will experience while waiting for service in the WFQ (that is, after passing through the leaky bucket)? Let us focus on flow 1. Suppose that flow 1's token bucket is initially full. A burst of b_1 packets then arrives to the leaky bucket policer for flow 1. These packets remove all of the tokens (without wait) from the leaky bucket and then join the WFQ waiting area for flow 1. Since these b_1 packets are served at a rate of at least $R \cdot w_1/(\Sigma w_j)$ packet/sec., the last of these packets will then have a maximum delay, d_{max}, until its transmission is completed, where

$$d_{max} = \frac{b_1}{R \cdot w_1 / \Sigma w_j}$$

The justification of this formula is that if there are b_1 packets in the queue and packets are being serviced (removed) from the queue at a rate of at least $R \cdot w_1/ (\Sigma w_j)$ packets per second, then the amount of time until the last bit of the last packet is transmitted

cannot be more than $b_1/(R \cdot w_1/(\Sigma w_j))$. A homework problem asks you to prove that as long as $r_1 < R \cdot w_1/(\Sigma w_j)$, then d_{max} is indeed the maximum delay that any packet in flow 1 will ever experience in the WFQ queue.

© 2000-2001 by Addison Wesley Longman
A division of Pearson Education

Online Book

6.7: Integrated Services

In the previous sections, we identified both the principles and the mechanisms used to provide quality of service in the Internet. In this section, we consider how these ideas are exploited in a particular architecture for providing quality of service in the Internet--the so-called Intserv (Integrated Services) Internet architecture. Intserv is a framework developed within the IETF to provide individualized quality-of-service guarantees to individual application sessions. Two key features lie at the heart of Intserv architecture:

- *Reserved resources.* A router is required to know what amounts of its resources (buffers, link bandwidth) are already reserved for ongoing sessions.
- *Call setup.* A session requiring QoS guarantees must first be able to reserve sufficient resources at each network router on its source-to-destination path to ensure that its end-to-end QoS requirement is met. This call setup (also known as call admission) process requires the participation of each router on the path. Each router must determine the local resources required by the session, consider the amounts of its resources that are already committed to other ongoing sessions, and determine whether it has sufficient resources to satisfy the per-hop QoS requirement of the session at this router without violating local QoS guarantees made to an already-admitted session.

Figure 6.31 depicts the call setup process.

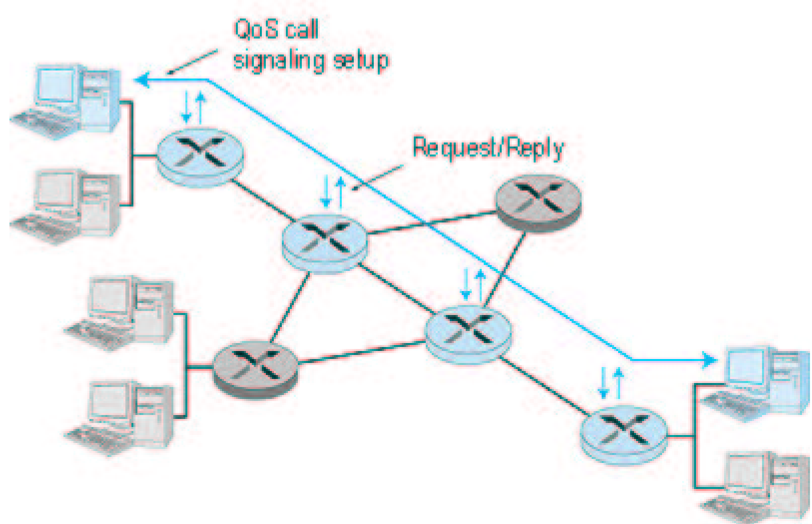

Figure 6.31: The call setup process

Let us now consider the steps involved in call admission in more detail:

1. *Traffic characterization and specification of the desired QoS.* In order for a router to determine whether or not its resources are sufficient to meet the QoS requirements of a session, that session must first declare its QoS requirement, as well as characterize the traffic that it will be sending into the network, and for which it requires a QoS guarantee. In the Intserv architecture, the so-called Rspec (R for reserved) defines the specific QoS being requested by a connection; the so-called Tspec (T for traffic) characterizes the traffic the sender will be sending into the network, or the receiver will be receiving from the network. The specific form of the Rspec and Tspec will vary, depending on the service requested, as discussed below. The Tspec and Rspec are defined in part in RFC 2210 and RFC 2215.

2. *Signaling for call setup.* A session's Tspec and Rspec must be carried to the routers at which resources will be reserved for the session. In the Internet, the RSVP protocol, which is discussed in detail in the next section, is currently the signaling protocol of choice. RFC 2210 describes the use of the RSVP resource reservation protocol with the Intserv architecture.

3. *Per-element call admission.* Once a router receives the Tspec and Rspec for a session requesting a QoS guarantee, it can determine whether or not it can admit the call. This call admission decision will depend on the traffic specification, the requested type of service, and the existing resource commitments already made by the router to ongoing sessions. Per-element call admission is shown in Figure 6.32.

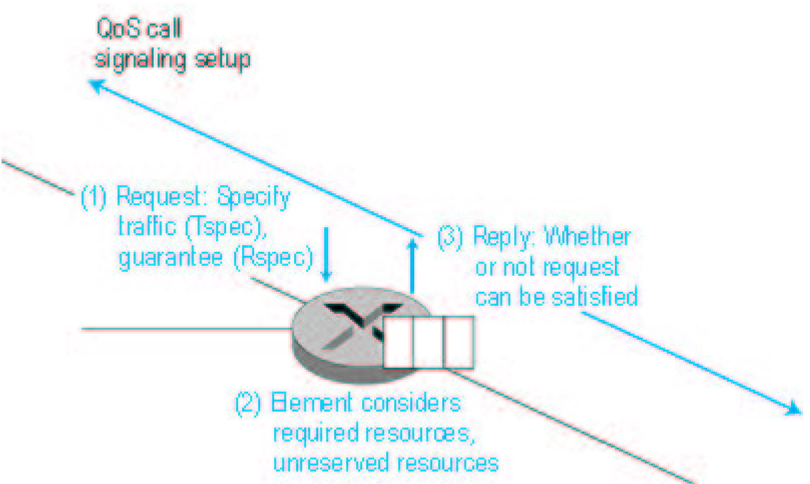

Figure 6.32: Per-element call behavior

The Intserv architecture defines two major classes of service: guaranteed service and controlled-load service. We will see shortly that each provides a very different form of a quality of service guarantee.

6.7.1: Guaranteed Quality of Service

The guaranteed service specification, defined in RFC 2212, provides firm (mathematically provable) bounds on the queuing delays that a packet will experience in a router. While the details behind guaranteed service are rather complicated, the basic idea is really quite simple. To a first approximation, a source's traffic characterization is given by a leaky bucket (see Section 6.6) with parameters (r,b) and the requested service is characterized by a transmission rate, R, at which packets will be transmitted. In essence, a session requesting guaranteed service is requiring that the bits in its packet be guaranteed a forwarding rate of R bits/sec. Given that traffic is specified using a leaky bucket characterization, and a guaranteed rate of R is being requested, it is also possible to bound the maximum queuing delay at the router. Recall that with a leaky bucket traffic characterization, the amount of traffic (in bits) generated over any interval of length t is bounded by $rt + b$. Recall also from Section 6.6, that when a leaky bucket source is fed into a queue that guarantees that queued traffic will be serviced at least at a rate of R bits per second, the maximum queuing delay experienced by any packet will be bounded by b/R, as long as R is greater than r. The actual delay bound guaranteed under the guaranteed service definition is slightly more complicated, due to packetization effects (the simple b/R bound assumes that data is in the form of a fluid-like flow rather than discrete packets), the fact that the traffic arrival process is subject to the peak rate limitation of the input link (the simple b/R bound assumes that a burst of b bits can arrive in zero time), and possible additional variations in a packet's transmission time.

6.7.2: Controlled-Load Network Service

A session receiving controlled-load service will receive "a quality of service closely approximating the QoS that same flow would receive from an unloaded network element" [RFC 2211]. In other words, the session may assume that a "very high percentage" of its packets will successfully pass through the router without being

dropped and will experience a queuing delay in the router that is close to zero. Interestingly, controlled load service makes no quantitative guarantees about performance--it does not specify what constitutes a "very high percentage" of packets nor what quality of service closely approximates that of an unloaded network element.
The controlled-load service targets real-time multimedia applications that have been developed for today's Internet. As we have seen, these applications perform quite well when the network is unloaded, but rapidly degrade in performance as the network becomes more loaded.

Online Book

Online Book

6.8: RSVP

We learned in section 6.7 that in order for a network to provide QoS guarantees, there must be a signaling protocol that allows applications running in hosts to reserve resources in the Internet. RSVP [RFC 2205; Zhang 1993], is such a signaling protocol for the Internet.

When people talk about *resources* in the Internet context, they usually mean link bandwidth and router buffers. To keep the discussion concrete and focused, however, we shall assume that the word *resource* is synonymous with *bandwidth.* For our pedagogic purposes, RSVP stands for bandwidth reservation protocol.

6.8.1: The Essence of RSVP

The RSVP protocol allows applications to reserve bandwidth for their data flows. It is used by a host, on the behalf of an application data flow, to request a specific amount of bandwidth from the network. RSVP is also used by the routers to forward bandwidth reservation requests. To implement RSVP, RSVP software must be present in the receivers, senders, and routers. The two principal characteristics of RSVP are:

1. It provides **reservations for bandwidth in multicast trees** (unicast is handled as a degenerate case of multicast).

2. It is **receiver-oriented,** that is, the receiver of a data flow initiates and maintains the resource reservation used for that flow.

These two characteristics are illustrated in Figure 6.33. The diagram shows a multicast tree with data flowing from the top of the tree to hosts at the bottom of the tree. Although data originates from the sender, the reservation messages originate from the receivers. When a router forwards a reservation message upstream toward the sender, the router may merge the reservation message with

other reservation messages arriving from downstream.

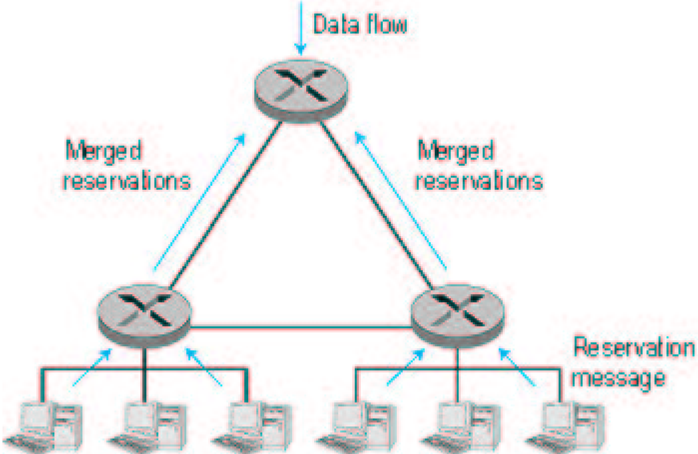

Figure 6.33: RSVP: Multicast- and receiver-oriented

Before discussing RSVP in greater detail, we need to consider the notion of a **session.** As with RTP, a session can consist of multiple multicast data flows. Each sender in a session is the source of one or more data flows; for example, a sender might be the source of a video data flow and an audio data flow. Each data flow in a session has the same multicast address. To keep the discussion concrete, we assume that routers and hosts identify the session to which a packet belongs by the packet's multicast address. This assumption is somewhat restrictive; the actual RSVP specification allows for more general methods to identify a session. Within a session, the data flow to which a packet belongs also needs to be identified. This could be done, for example, with the flow identifier field in IPv6.

What RSVP Is Not

We emphasize that the RSVP standard [RFC 2205] does not specify how the network provides the reserved bandwidth to the data flows. It is merely a protocol that allows the applications to reserve the necessary link bandwidth. Once the reservations are in place, it is up to the routers in the Internet to actually provide the reserved bandwidth to the data flows. This provisioning would likely be done with the scheduling mechanisms (priority scheduling, weighted fair queuing, etc.) discussed in Section 6.6.

It is also important to understand that RSVP is not a routing protocol--it does not determine the links in which the reservations are to be made. Instead it depends on an underlying routing protocol (unicast or multicast) to determine the routes for the flows. Once the routes are in place, RSVP can reserve bandwidth in the links along these routes. (We shall see shortly that when a route changes, RSVP re-reserves resources.) Once the reservations are in place, the routers' packet schedulers must actually provide the reserved bandwidth to the data flows. Thus, RSVP is only one piece--albeit an important piece--in the QoS guarantee puzzle.

RSVP is sometimes referred to as a *signaling protocol.* By this it is meant that RSVP is a protocol that allows hosts to establish and tear down reservations for data flows. The term "signaling protocol" comes from the jargon of the circuit-switched telephony community.

Heterogeneous Receivers

Some receivers can receive a flow at 28.8 Kbps, others at 128 Kbps, and yet

others at 10 Mbps or higher. This heterogeneity of the receivers poses an interesting question. If a sender is multicasting a video to a group of heterogeneous receivers, should the sender encode the video for low quality at 28.8 Kbps, for medium quality at 128 Kbps, or for high quality at 10 Mbps? If the video is encoded at 10 Mbps, then only the users with 10 Mbps access will be able to watch the video. On the other hand, if the video is encoded at 28.8 Kbps, then the 10 Mbps users will have to see a low-quality image when they know they can see something much better.

To resolve this dilemma it is often suggested that video and audio be encoded in layers. For example, a video might be encoded into two layers: a base layer and an enhancement layer. The base layer could have a rate of 20 Kbps whereas the enhancement layer could have a rate of 100 Kbps; in this manner receivers with 28.8 Kbps access could receive the low-quality base-layer image, and receivers with 128 Kbps could receive both layers to construct a high-quality image.

We note that the sender does not need to know the receiving rates of all the receivers. It only needs to know the maximum rate of all its receivers. The sender encodes the video or audio into multiple layers and sends all the layers up to the maximum rate into multicast tree. The receivers pick out the layers that are appropriate for their receiving rates. In order to not excessively waste bandwidth in the network's links, the heterogeneous receivers must communicate to the network the rates they can handle. We shall see that RSVP gives foremost attention to the issue of reserving resources for heterogeneous receivers.

6.8.2: A Few Simple Examples

Let us first describe RSVP in the context of a concrete one-to-many multicast example. Suppose there is a source that is transmitting the video of a major sporting event into the Internet. This session has been assigned a multicast address, and the source stamps all of its outgoing packets with this multicast address. Also suppose that an underlying multicast routing protocol has established a multicast tree from the sender to four receivers as shown below; the numbers next to the receivers are the rates at which the receivers want to receive data. Let us also assume that the video is layered and encoded to accommodate this heterogeneity of receiver rates.

Crudely speaking, RSVP operates as follows for this example. Each receiver sends a **reservation message** upstream into the multicast tree. This reservation message specifies the rate at which the receiver would like to receive the data from the source. When the reservation message reaches a router, the router adjusts its packet scheduler to accommodate the reservation. It then sends a reservation upstream. The amount of bandwidth reserved upstream from the router depends on the bandwidths reserved downstream. In the example in Figure 6.34, receivers R1, R2, R3, and R4 reserve 20 Kbps, 100 Kbps, 3 Mbps, and 3 Mbps, respectively. Thus router D's downstream receivers request a maximum of 3 Mbps. For this one-to-many transmission, Router D sends a reservation message to Router B requesting that Router B reserve 3 Mbps on the link between the two routers. Note that only 3 Mbps are reserved and not 3+3=6 Mbps; this is because receivers R3 and R4 are watching the same sporting event, so their reservations may be merged. Similarly, Router C requests that Router B

reserve 100 Kbps on the link between routers B and C; the layered encoding ensures that receiver R1's 20 Kbps stream is included in the 100 Kbps stream. Once Router B receives the reservation message from its downstream routers and passes the reservations to its schedulers, it sends a new reservation message to its upstream router, Router A. This message reserves 3 Mbps of bandwidth on the link from Router A to Router B, which is again the maximum of the downstream reservations.

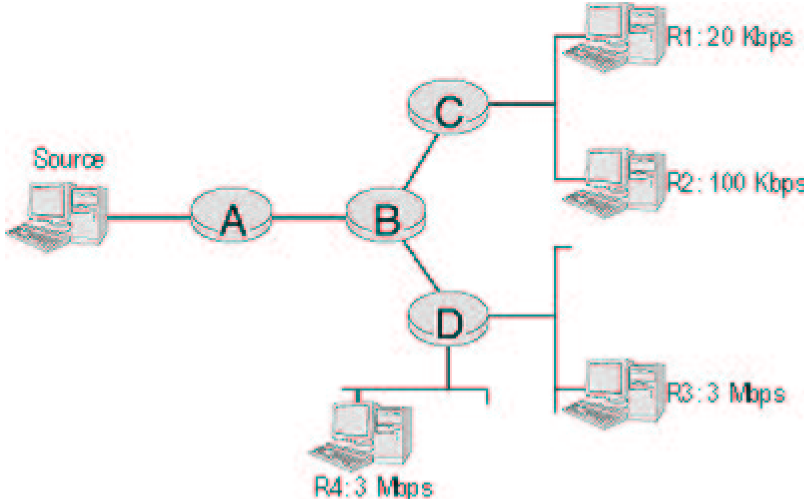

Figure 6.34: An RSVP example

We see from this first example that RSVP is **receiver-oriented,** that is, the receiver of a data flow initiates and maintains the resource reservation used for that flow. Note that each router receives a reservation message from each of its downstream links in the multicast tree and sends only one reservation message into its upstream link.

As another example, suppose that four persons are participating in a video conference, as shown in Figure 6.35. Each person has three windows open on her computer to look at the other three persons. Suppose that the underlying routing protocol has established the multicast tree among the four hosts as shown in the diagram below. Finally, suppose each person wants to see each of the videos at 3 Mbps. Then on each of the links in this multicast tree, RSVP would reserve 9 Mbps in one direction and 3 Mbps in the other direction. Note that RSVP does not merge reservations in this example, as each person wants to receive three distinct streams.

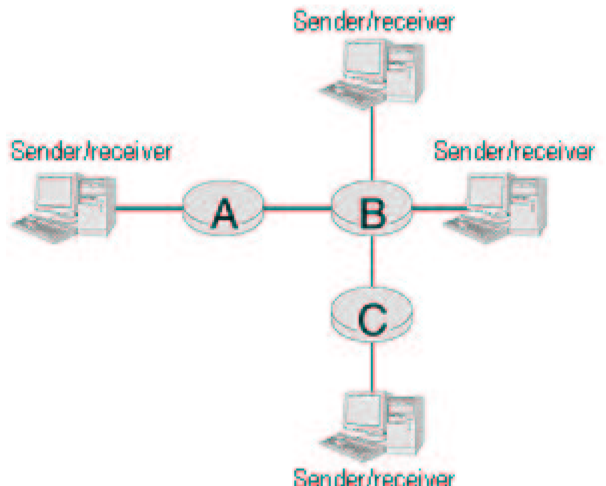

Figure 6.35: An RSVP video conference example

Now consider an audio conference among the same four persons over the same multicast tree. Suppose *b* bps are needed for an isolated audio stream. Because in an audio conference it is rare that more than two persons speak at the same time, it is not necessary to reserve $3 \cdot b$ bps into each receiver; $2 \cdot b$ should suffice. Thus, in this last application we can conserve bandwidth by merging reservations.

Call Admission

Just as the manager of a restaurant should not accept reservations for more tables than the restaurant has, the amount of bandwidth on a link that a router reserves should not exceed the link's capacity. Thus whenever a router receives a new reservation message, it must first determine if its downstream links on the multicast tree can accommodate the reservation. This **admission test** is performed whenever a router receives a reservation message. If the admission test fails, the router rejects the reservation and returns an error message to the appropriate receiver(s).

RSVP does not define the admission test, but it assumes that the routers perform such a test and that RSVP can interact with the test.

6.8.3: Path Messages

So far we have only discussed the RSVP reservation messages. These messages originate at the receivers and flow upstream toward the senders. **Path messages** are another important RSVP message type; they originate at the senders and flow downstream toward the receivers.

The principal purpose of the path messages is to let the routers know the links on which they should forward the reservation messages. Specifically, a path message sent within the multicast tree from a Router A to a Router B contains Router A's unicast IP address. Router B puts this address in a path-state table, and when it receives a reservation message from a downstream node it accesses the table and learns that it should send a reservation message up the multicast tree to Router A. In the future some routing protocols may supply reverse path forwarding information directly, replacing the reverse-routing function of the path state.

Along with some other information, the path messages also contain a *sender*

Tspec, which defines the traffic characteristics of the data stream that the sender will generate (see Section 6.7). This Tspec can be used to prevent over-reservation.

6.8.4: Reservation Styles

Through its **reservation style,** a reservation message specifies whether merging of reservations from the same session is permissible. A reservation style also specifies the session senders from which a receiver desires to receive data. Recall that a router can identify the sender of a datagram from the datagram's source IP address.

There are currently three reservation styles defined: *wildcard-filter style, fixed-filter style,* and *shared-explicit style.*

- *Wildcard-filter style.* When a receiver uses the wildcard-filter style in its reservation message, it is telling the network that it wants to receive all flows from all upstream senders in the session and that its bandwidth reservation is to be shared among the senders.
- *Fixed-filter style.* When a receiver uses the fixed-filter style in its reservation message, it specifies a list of senders from which it wants to receive a data flow along with a bandwidth reservation for each of these senders. These reservations are distinct, that is, they are not to be shared.
- *Shared-explicit style.* When a receiver uses the shared-explicit style in its reservation message, it specifies a list of senders from which it wants to receive a data flow along with a single bandwidth reservation. This reservation is to be shared among all the senders in the list.

Shared reservations, created by the wildcard filter and the shared-explicit styles, are appropriate for a multicast session whose sources are unlikely to transmit simultaneously. Packetized audio is an example of an application suitable for shared reservations; because a limited number of people talk at once, each receiver might issue a wildcard-filter or a shared-explicit reservation request for twice the bandwidth required for one sender (to allow for overspeaking). On the other hand, the fixed-filter reservation, which creates distinct reservations for the flows from different senders, is appropriate for video teleconferencing.

Examples of Reservation Styles

Following the RSVP Internet RFC, let's next consider a few examples of the three reservation styles. In Figure 6.36, a router has two incoming interfaces, labeled A and B, and two outgoing interfaces, labeled C and D. The many-to-many multicast session has three senders--S1, S2, and S3--and three receivers--R1, R2, and R3. Figure 6.36 also shows that interface D is connected to a LAN.

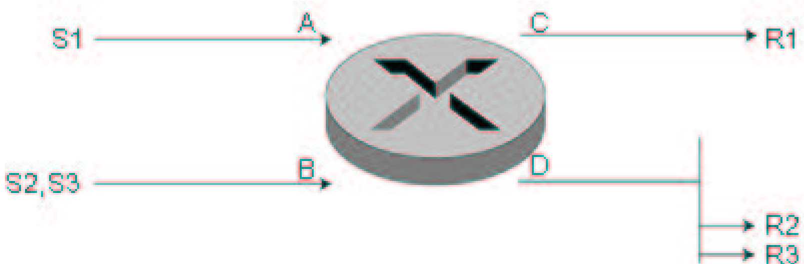

Figure 6.36: Sample scenario for RSVP reservation styles

Suppose first that all of the receivers use the wildcard-filter reservation. As shown in the Figure 6.37, receivers R1, R2, and R3 want to reserve 4*b,* 3*b,* and 2*b,* respectively, where *b* is a given bit rate. In this case, the router reserves 4b on interface C and 3*b* on interface D. Because of the wildcard-filter reservation, the two reservations from R2 and R3 are merged for interface D. The larger of the two reservations is used rather than the sum of reservations. The router then sends a reservation message upstream to interface A and another to interface B; each of these reservation message requests is 4*b,* which is the larger of 3*b* and 4*b.*

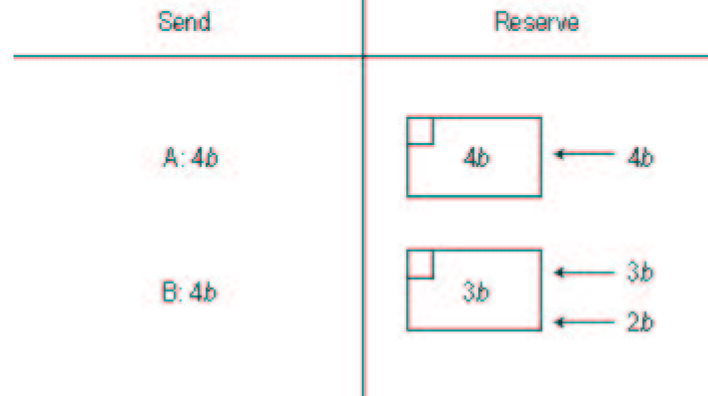

Figure 6.37: Wildcard filter reservations

Now suppose that all of the receivers use the fixed-filter reservation. As shown in Figure 6.38, receiver R1 wants to reserve 4b for source S1 and 5b for source S2; also shown in the figure are the reservation requests from R2 and R3. Because of the fixed-filter style, the router reserves two disjoint chunks of bandwidth on interface C: one chunk of 4*b* for S1 and another chunk of 5*b* for S2. Similarly, the router reserves two disjoint chunks of bandwidth on interface D: one chunk of 3*b* for S1 (the maximum of *b* and 3*b*) and one chunk of *b* for S3. On interface A, the router sends a message with a reservation for S1 of 4*b* (the maximum of 3*b* and 4*b*). On interface B, the router sends a message with a reservation of 5*b* for S2 and *b* for S3.

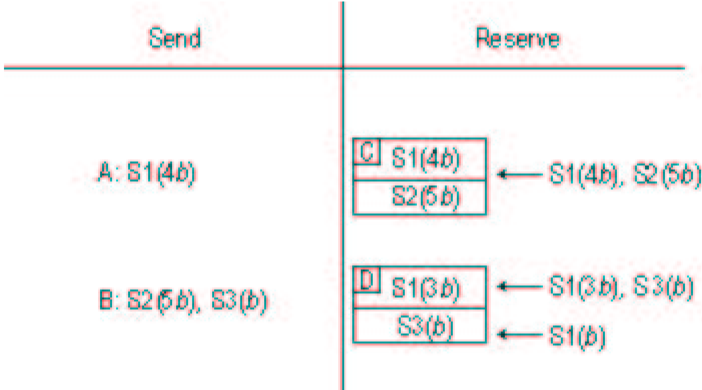

Figure 6.38: Fixed filter reservations

Finally, suppose that each of the receivers use the shared-explicit reservation. As shown in Figure 6.39, receiver R1 desires a pipe of 1*b,* which is to be shared between sources S1 and S2; receiver R2 desires a pipe of 3*b* to be shared between sources S1 and S3; and receiver R3 wants a pipe of 2*b* for source S2. Because of the shared-explicit style, the reservations from R2 and R3 are merged for interface D. Only one pipe is reserved on interface D, although it is reserved at the maximum of the reservation rates. RSVP will reserve on interface B a pipe of 3*b* to be shared by S2 and S3; note that 3*b* is the maximum of the downstream reservations for S2 and S3.

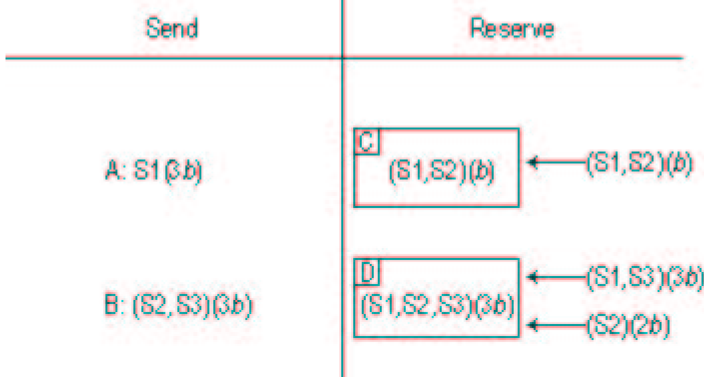

Figure 6.39: Shared-explicit reservations

In each of the above examples the three receivers used the same reservation style. Because receivers make independent decisions, the receivers participating in a session could use different styles. RSVP does not permit, however, reservations of different styles to be merged.

The Principle of Soft State

The reservations in the routers and hosts are maintained with soft states. By this it is meant that each reservation for bandwidth stored in a router has an associated timer. If a reservation's timer expires, then the reservation is removed. If a receiver desires to maintain a reservation, it must periodically refresh the reservation by sending reservation messages. This soft-state principle is also used by other protocols in computer networking. As we learned in Chapter 5, for the routing

tables in transparent bridges, the entries are refreshed by data packets that arrive to the bridge; entries that are not refreshed are timed-out. On the other hand, if a protocol takes explicit actions to modify or release state, then the protocol makes use of hard state. Hard state is employed in virtual circuit networks (VC), in which explicit actions must be taken to adjust VC tables in switching nodes to establish and tear down VCs.

6.8.5: Transport of Reservation Messages

RSVP messages are sent hop-by-hop directly over IP. Thus the RSVP message is placed in the information field of the IP datagram; the protocol number in the IP datagram is set to 46. Because IP is unreliable, RSVP messages can be lost. If an RSVP path or reservation message is lost, a replacement refresh message should arrive soon.

An RSVP reservation message that originates in a host will have the host's IP address in the source address field of the encapsulating IP datagram. It will have the IP address of the first router along the reserve path in the multicast tree in the destination address in the encapsulating IP datagram. When the IP datagram arrives at the first router, the router strips off the IP fields and passes the reservation message to the router's RSVP module. The RSVP module examines the message's multicast address (that is, session identifier) and style type, examines its current state, and then acts appropriately; for example, the RSVP module may merge the reservation with a reservation originating from another interface and then send a new reservation message to the next router upstream in the multicast tree.

Insufficient Resource

Because a reservation request that fails an admission test may embody a number of requests merged together, a reservation error must be reported to all the concerned receivers. These reservation errors are reported within **ResvError messages.** The receivers can then reduce the amount of resource that they request and try reserving again. The RSVP standard provides mechanisms to allow the backtracking of the reservations when insufficient resources are available; unfortunately, these mechanisms add significant complexity to the RSVP protocol. Furthermore, RSVP suffers from the so-called **killer-reservation problem,** whereby a receiver requests a large reservation over and over again, each time getting its reservation rejected due to lack of sufficient resources. Because this large reservation may have been merged with smaller reservations downstream, the large reservation may be excluding smaller reservations from being established. To solve this thorny problem, RSVP uses the ResvError messages to establish additional state in routers, called blockade state. Blockade state in a router modifies the merging procedure to omit the offending reservation from the merge, allowing a smaller request to be forwarded and established. The blockade state adds yet further complexity to the RSVP protocol and its implementation.

6.9: Differentiated Services

In the previous section we discussed how RSVP can be used to reserve *per-flow* resources at routers within the network. The ability to request and reserve per-flow resources, in turn, makes it possible for the Intserv framework to provide quality-of-service guarantees to individual flows. As work on Intserv and RSVP proceeded, however, researchers involved with these efforts (for example, [Zhang 1998]) have begun to uncover some of the difficulties associated with the Intserv model and per-flow reservation of resources:

- *Scalability.* Per-flow resource reservation using RSVP implies the need for a router to process resource reservations and to maintain per-flow state for *each* flow passing though the router. With recent measurements [Thomson 1997] suggesting that even for an OC-3 speed link, approximately 256,000 source-destination pairs might be seen in one minute in a backbone router, per-flow reservation processing represents a considerable overhead in large networks.
- *Flexible service models.* The Intserv framework provides for a small number of prespecified service classes. This particular set of service classes does not allow for more qualitative or relative definitions of service distinctions (for example, "Service class A will receive preferred treatment over service class B."). These more qualitative definitions might better fit our intuitive notion of service distinction (for example, first class versus coach class in air travel; "platinum" versus "gold" versus "standard" credit cards).

These considerations have led to the recent so-called "diffserv" (Differentiated Services) activity [Diffserv 1999] within the Internet Engineering Task Force. The Diffserv working group is developing an architecture for providing *scalable* and *flexible* service differentiation--that is, the ability to handle different "classes" of traffic in different ways within the Internet. The need for *scalability* arises from the fact that hundreds of thousands of simultaneous source-destination traffic flows may be present at a backbone router of the Internet. We will see shortly that this need is met by placing only simple functionality within the network core, with more complex control operations being implemented at the "edge" of the network. The need for *flexibility* arises from the fact that new service classes may arise and old service classes may become obsolete. The differentiated services architecture is flexible in the sense that it does not define specific services or service classes (for example, as is the case with Intserv). Instead, the differentiated services architecture provides the functional components, that is, the pieces of a network architecture, with which such

services can be built. Let us now examine these components in detail.

6.9.1: Differentiated Services: A Simple Scenario

To set the framework for defining the architectural components of the differentiated service model, let us begin with the simple network shown in Figure 6.40. In the following, we describe one possible use of the Diffserv components. Many other possible variations are possible, as described in RFC 2475. Our goal here is to provide an introduction to the key aspects of differentiated services, rather than to describe the architectural model in exhaustive detail.

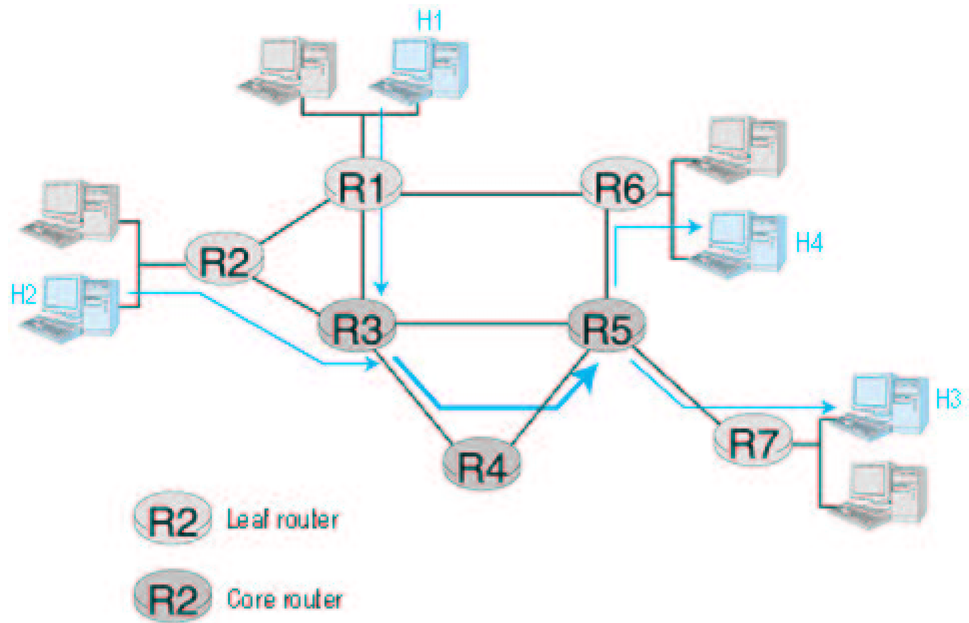

Figure 6.40: A simple diffserv network example

The differentiated services architecture consists of two sets of functional elements:

- *Edge functions: Packet classification and traffic conditioning.* At the incoming "edge" of the network (that is, at either a Diffserv-capable host that generates traffic or at the first Diffserv-capable router that the traffic passes through), arriving packets are marked. More specifically, the Differentiated Service (DS) field of the packet header is set to some value. For example, in Figure 6.40, packets being sent from H1 to H3 might be marked at R1, while packets being sent from H2 to H4 might be marked at R2. The mark that a packet receives identifies the class of traffic to which it belongs. Different classes of traffic will then receive different service within the core network. The RFC defining the differentiated service architecture, RFC 2475, uses the term **behavior aggregate** rather than "class of traffic." After being marked, a packet may then be immediately forwarded into the network, delayed for some time before being forwarded, or it may be discarded. We will see shortly that many factors can influence how a packet is to be marked, and whether it is to be forwarded

immediately, delayed, or dropped.

- *Core function: Forwarding.* When a DS-marked packet arrives at a Diffserv-capable router, the packet is forwarded onto its next hop according to the so-called **per-hop behavior** associated with that packet's class. The per-hop behavior influences how a router's buffers and link bandwidth are shared among the competing classes of traffic. A crucial tenet of the Diffserv architecture is that a router's per-hop behavior will be based *only* on packet markings, that is, the class of traffic to which a packet belongs. Thus, if packets being sent from H1 to H3 in Figure 6.40 receive the same marking as packets from H2 to H4, then the network routers treat these packets as an aggregate, without distinguishing whether the packets originated at H1 or H2. For example, R3 would not distinguish between packets from H1 and H2 when forwarding these packets on to R4. Thus, the differentiated service architecture obviates the need to keep router state for individual source-destination pairs--an important consideration in meeting the scalability requirement discussed at the beginning of this section.

An analogy might prove useful here. At many large-scale social events (for example, a large public reception, a large dance club or discoth□que, a concert, a football game), people entering the event receive a "pass" of one type or another. There are VIP passes for Very Important People; there are over-18 passes for people who are 18 years old or older (for example, if alcoholic drinks are to be served); there are backstage passes at concerts; there are press passes for reporters; there is an ordinary pass for the Ordinary Person. These passes are typically distributed on entry to the event, that is, at the "edge" of the event. It is here at the edge where computationally intensive operations such as paying for entry, checking for the appropriate type of invitation, and matching an invitation against a piece of identification, are performed. Furthermore, there may be a limit on the number of people of a given type that are allowed into an event. If there is such a limit, people may have to wait before entering the event. Once inside the event, one's pass allows one to receive differentiated service at many locations around the event--a VIP is provided with free drinks, a better table, free food, entry to exclusive rooms, and fawning service. Conversely, an Ordinary Person is excluded from certain areas, pays for drinks, and receives only basic service. In both cases, the service received within the event depends solely on the type of one's pass. Moreover, all people within a class are treated alike.

6.9.2: Traffic Classification and Conditioning

In the differentiated services architecture, a packet's mark is carried within the DS field in the IPv4 or IPv6 packet header. The definition of the DS field is intended to supersede the earlier definitions of the IPv4 Type-of-Service field (see Section 4.4) and the IPv6 Traffic Class Field (see Section 4.7). The structure of this eight-bit field is shown below in Figure 6.41.

Figure 6.41: Structure of the DS field in IVv4 and IPv6 header

The six-bit differentiated service code point (DSCP) subfield determines the so-called per-hop behavior (see Section 6.9.3) that the packet will receive within the network. The two-bit CU subfield of the DS field is currently unused. Restrictions are placed on the use of half of the DSCP values in order to preserve backward compatibility with the IPv4 ToS field use; see RFC 2474 for details. For our purposes here, we need only note that a packet's mark, its "code point" in the Diffserv terminology, is carried in the eight-bit Diffserv field.

As noted above, a packet is marked by setting its Diffserv field value at the edge of the network. This can either happen at a Diffserv-capable host or at the first point at which the packet encounters a Diffserv-capable router. For our discussion here, we will assume marking occurs at an edge router that is directly connected to a sender, as shown in Figure 6.40.

Figure 6.42 provides a logical view of the classification and marking function within the edge router. Packets arriving to the edge router are first "classified." The classifier selects packets based on the values of one or more packet header fields (for example, source address, destination address, source port, destination port, protocol ID) and steers the packet to the appropriate marking function. The DS field value is then set accordingly at the marker. Once packets are marked, they are then forwarded along their route to the destination. At each subsequent Diffserv-capable router, marked packets then receive the service associated with their marks. Even this simple marking scheme can be used to support different classes of service within the Internet. For example, all packets coming from a certain set of source IP addresses (for example, those IP addresses that have paid for an expensive priority service within their ISP) could be marked on entry to the ISP, and then receive a specific forwarding service (for example, a higher priority forwarding) at all subsequent Diffserv-capable routers. A question not addressed by the Diffserv working group is how the classifier obtains the "rules" for such classification. This could be done manually, that is, the network administrator could load a table of source addresses that are to be marked in a given way into the edge routers, or this could be done under the control of some yet-to-be-specified signaling protocol.

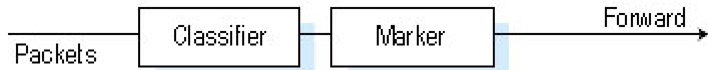

Figure 6.42: Simple packet classification and marking

In Figure 6.42, all packets meeting a given header condition receive the same marking, regardless of the packet arrival rate. In some scenarios, it might also be desirable to limit the rate at which packets bearing a given marking are injected into the network. For example, an end user might negotiate a contract with its ISP to receive high-priority service, but at the

same time agree to limit the maximum rate at which it would send packets into the network. That is, the end user agrees that its packet sending rate would be within some declared **traffic profile.** The traffic profile might contain a limit on the peak rate, as well as the burstiness of the packet flow, as we saw in Section 6.6 with the leaky bucket mechanism. As long as the user sends packets into the network in a way that conforms to the negotiated traffic profile, the packets receive their priority marking. On the other hand, if the traffic profile is violated, the out-of-profile packets might be marked differently, might be shaped (for example, delayed so that a maximum rate constraint would be observed), or might be dropped at the network edge. The role of the **metering function,** shown in Figure 6.43, is to compare the incoming packet flow with the negotiated traffic profile and to determine whether a packet is within the negotiated traffic profile. The actual decision about whether to immediately re-mark, forward, delay, or drop a packet is *not* specified in the Diffserv architecture. The Diffserv architecture only provides the framework for performing packet marking and shaping/dropping; it does *not* mandate any specific policy for what marking and conditioning (shaping or dropping) is actually to be done. The hope, of course, is that the Diffserv architectural components are together flexible enough to accommodate a wide and constant evolving set of services to end users. For a discussion of a policy framework for Diffserv, see [Rajan 1999].

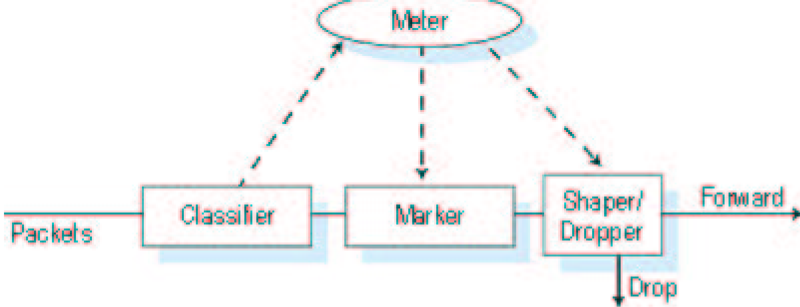

Figure 6.43: Logical view of packet classification and traffic conditioning at the edge router

6.9.3: Per-Hop Behaviors

So far, we have focused on the edge functions in the differentiated services architecture. The second key component of the Diffserv architecture involves the per-hop behavior performed by Diffserv-capable routers. The per-hop behavior (PHB) is rather cryptically, but carefully, defined as "a description of the externally observable forwarding behavior of a Diffserv node applied to a particular Diffserv behavior aggregate" [RFC 2475]. Digging a little deeper into this definition, we can see several important considerations embedded within:

- A PHB can result in different classes of traffic receiving different performance (that is, different externally observable forwarding behavior).
- While a PHB defines differences in performance (behavior) among

classes, it does not mandate any particular mechanism for achieving these behaviors. As long as the externally observable performance criteria are met, any implementation mechanism and any buffer/bandwidth allocation policy can be used. For example, a PHB would not require that a particular packet queuing discipline, for example, a priority queue versus a weighted-fair-queuing queue versus a first-come-first-served queue, be used to achieve a particular behavior. The PHB is the "end," to which resource allocation and implementation mechanisms are the "means."

- Differences in performance must be observable, and hence measurable.

An example of a simple PHB is one that guarantees that a given class of marked packets receive at least *x*% of the outgoing link bandwidth over some interval of time. Another per-hop behavior might specify that one class of traffic will always receive strict priority over another class of traffic--that is, if a high-priority packet and a low-priority packet are present in a router's queue at the same time, the high-priority packet will always leave first. Note that while a priority queuing discipline might be a natural choice for implementing this second PHB, any queuing discipline that implements the required observable behavior is acceptable.

Currently, two PHBs are under active discussion within the Diffserv working group: an expedited forwarding (EF) PHB [RFC 2598] and an assured forwarding (AF) PHB [RFC 2597]:

- The **expedited forwarding** PHB specifies that the departure rate of a class of traffic from a router must equal or exceed a configured rate. That is, during any interval of time, the class of traffic can be guaranteed to receive enough bandwidth so that the output rate of the traffic equals or exceeds this minimum configured rate. Note that the EF per-hop behavior implies some form of isolation among traffic classes, as this guarantee is made *independently* of the traffic intensity of any other classes that are arriving to a router. Thus, even if the other classes of traffic are overwhelming router and link resources, enough of those resources must still be made available to the class to ensure that it receives its minimum rate guarantee. EF thus provides a class with the simple *abstraction* of a link with a minimum guaranteed link bandwidth.

- The **assured forwarding** PHB is more complex. AF divides traffic into four classes, where each AF class is guaranteed to be provided with some minimum amount of bandwidth and buffering. Within each class, packets are further partitioned into one of three "drop preference" categories. When congestion occurs within an AF class, a router can then discard (drop) packets based on their drop preference values. See RFC 2597 for details. By varying the amount of resources allocated to each class, an ISP can provide different

levels of performance to the different AF traffic classes.

The AF PHB could be used as a building block to provide different levels of service to the end systems, for example, Olympic-like gold, silver, and bronze classes of service. But what would be required to do so? If gold service is indeed going to be "better" (and presumably more expensive!) than silver service, then the ISP must ensure that gold packets receive lower delay and/or loss than silver packets. Recall, however, that a minimum amount of bandwidth and buffering are to be allocated to *each* class. What would happen if gold service was allocated *x*% of a link's bandwidth and silver service was allocated *x*/2% of the link's bandwidth, but the traffic intensity of gold packets was 100 times higher than that of silver packets? In this case, it is likely that silver packets would receive *better* performance than the gold packets! (This is an outcome that leaves the silver service buyers happy, but the high-spending gold service buyers extremely unhappy!) Clearly, when creating a service out of a PHB, more than just the PHB itself will come into play. In this example, the dimensioning of resources--determining how much resources will be allocated to each class of service--must be done hand-in-hand with knowledge about the traffic demands of the various classes of traffic.

6.9.4: A Beginning

The differentiated services architecture is still in the early stages of its development and is rapidly evolving. RFCs 2474 and 2475 define the fundamental framework of the Diffserv architecture but themselves are likely to evolve as well. The ways in which PHBs, edge functionality, and traffic profiles can be combined to provide an end-to-end service, such as a virtual leased line service [RFC 2638] or an Olympic-like gold/silver/bronze service [RFC 2597], are still under investigation. In our discussion above, we have assumed that the Diffserv architecture is deployed within a single administrative domain. The (typical) case where an end-to-end service must be fashioned from a connection that crosses several administrative domains, and through non-Diffserv-capable routers, pose additional challenges beyond those described above.

© 2000-2001 by Addison Wesley Longman
A division of Pearson Education

6.10: Summary

Multimedia networking is perhaps the most exciting development in the Internet today. People throughout the world are spending less time in front of their radios and

televisions and are instead turning to the Internet to receive audio and video emissions, both live and prerecorded. As high-speed access penetrates more residences, this trend will continue--couch potatoes throughout the world will access their favorite video programs through the Internet rather than through the traditional broadcast distribution channels. In addition to audio and video distribution, the Internet is also being used to transport phone calls. In fact, over the next 10 years the Internet may render the traditional circuit-switched telephone system nearly obsolete in many countries. The Internet will not only provide phone service for less money, but will also provide numerous value-added services, such as video conferencing, online directory services, and voice messaging services.

In Section 6.1 we classified multimedia applications into three categories: streaming stored audio and video; one-to-many transmission of real-time audio and video; and real-time interactive audio and video. We emphasized that multimedia applications are delay-sensitive and loss-tolerant--characteristics that are very different from static-content applications that are delay tolerant and loss intolerant. We also discussed some of the hurdles that today's best-effort Internet places before multimedia applications. We surveyed several proposals to overcome these hurdles, including simply improving the existing networking infrastructure (by adding more bandwidth, more network caches, and deploying multicast), adding functionality to the Internet so that applications can reserve end-to-end resources (and so that the network can honor these reservations), and finally, introducing service classes to provide service differentiation.

In Sections 6.2-6.4 we examined architectures and mechanisms for multimedia networking in a best-effort network. In Section 6.2 we surveyed several architectures for streaming stored audio and video. We discussed user interaction--such as pause/resume, repositioning, and visual fast forward--and provided an introduction to RTSP, a protocol that provides client-server interaction to streaming applications. In Section 6.3 we examined how interactive real-time applications can be designed to run over a best-effort network. We saw how a combination of client buffers, packet sequence numbers, and timestamps can greatly alleviate the effects of network-induced jitter. We also studied how forward error correction and packet interleaving can improve user-perceived performance when a fraction of the packets are lost or are significantly delayed. In Section 6.4 we explored media chunk encapsulation, and we investigated in some detail one of the more important standards for media encapsulation, namely, RTP. We also looked at how RTP fits into the emerging H.323 architecture for interactive real-time conferencing.

Sections 6.5-6.9 looked at how the Internet can evolve to provide guaranteed QoS to its applications. In Section 6.5 we identified several principles for providing QoS to multimedia applications. These principles include packet marking and classification, isolation of packet flows, efficient use of resources, and call admission. In Section 6.6 we surveyed a variety of scheduling policies and policing mechanisms that can provide the foundation of a QoS networking architecture. The scheduling policies include priority scheduling, round-robin scheduling, and weighted-fair queuing. We then explored the leaky bucket as a policing mechanism, and showed how the leaky

bucket and weighted-fair queuing can be combined to bound the maximum delay a packet experiences at the output queue of a router.

In Sections 6.7-6.9 we showed how these principles and mechanisms have led to the definitions of new standards for providing QoS in the Internet. The first class of these standards is the so-called Intserv standard, which includes two services--the guaranteed QoS service and the controlled load service. Guaranteed QoS service provides hard, mathematical provable guarantees on the delay of each of the individual packets in a flow. Controlled-load service does not provide any hard guarantees, but instead ensures that most of an application’s packets will pass through a seemingly uncongested Internet. The Intserv architecture requires a signaling protocol for reserving bandwidth and buffer resources within the network. In Section 6.8 we examined in some detail an Internet signaling protocol for reservations, namely, RSVP. We indicated that one of the drawbacks of the Intserv architecture is the need for routers to maintain per-flow state, which may not scale. We concluded the chapter in Section 6.9 by outlining a recent and promising proposal for providing QoS in the Internet, namely, the Diffserv architecture. The Diffserv architecture does not require routers to maintain per-flow state; it instead classifies packets into a small number of aggregate classes, to which routers provide per-hop behavior. The Diffserv architecture is still in its infancy, but because the architecture requires relatively minor changes to the existing Internet protocols and infrastructure, it could be deployed relatively quickly.

Now that we have finished our study of multimedia networking, it is time to move on to another exciting topic: network security. Recent advances in multimedia networking may move the distribution of audio and video information to the Internet. As we’ll see in the next chapter, recent advances in network security may well help move the majority of economic transactions to the Internet.

© 2000-2001 by Addison Wesley Longman
A division of Pearson Education