7.1.4 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 with compression 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

Approach	Unit of allocation	Guarantee	Deployment to date	Complexity	Mechanisms
Making the best of best-effort service	none	none, or soft	everywhere	minimal	application-layer support, CDN, over-provisioning
Differential QoS	classes of flows	none, or soft	some	medium	policing, scheduling
Guaranteed QoS	individual flows	soft or hard, once a flow is admitted	little	high	policing, scheduling, call admission and signaling

Table 7.1 ♦ Three approaches to supporting multimedia applications

single image consisting of 1024 pixels, with each pixel encoded into 24 bits (8 bits each for the colors red, green, and blue), requires 3 Mbytes 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 Kbytes and the transmission time also drops by a factor of 10.

The topics 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. Many universities offer entire courses on audio compression and on video compression. We therefore provide here only 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:

- 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.
- 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 two, for example, 256 quantization values.
- 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 the original analog signal. Thus, there is a clear trade-off 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), **G.723.3** (both 6.4 and 5.3 kbps), and a large number of proprietary techniques. A popular compression technique for near CD-quality stereo music is **MPEG 1 layer 3**, more commonly known as **MP3**. MP3 encoders typically compress to rates of 96 kbps, 128 kbps, and 160 kbps, and produce 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, 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 simply to 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 on the JPEG standard for image compression by exploiting temporal redundancy across images in addition to the spatial redundancy exploited by JPEG. The **H.261** video compression standards are also very popular in the Internet. In addition there are numerous proprietary schemes, including Apple's QuickTime and Real Networks' encoders.

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].

CASE HISTORY

STREAMING STORED AUDIO AND VIDEO: FROM REALNETWORKS TO YOUTUBE

RealNetworks, a pioneer in audio and video streaming, was the first company to bring Internet audio to the mainstream. Its initial product—the RealAudio system released in 1995—included an audio encoder, an audio server, and an audio player. Allowing users to browse, select, and stream audio content from the Internet on demand, it quickly became a popular distribution system for providers of entertainment, educational, and news content.

Today audio and video streaming are among the most popular services in the Internet. Not only is there is a plethora of companies offering streamed content, but there is also a myriad of different server, player, and protocol technologies being employed. A few interesting examples (as of 2007) include:

- **Rhapsody from RealNetworks:** Provides streaming and downloading subscription services to users. Rhapsody uses its own proprietary client, which retrieves songs from its proprietary server over HTTP. As a song arrives over HTTP, it is played out through the Rhapsody client. Access to downloaded content is restricted through a Digital Rights Management (DRM) system.
- **MSN Video:** Users stream a variety of content, including international news and music video clips. Video is played through the popular Windows Media Player (WMP), which is available in almost all Windows hosts. Communication between WMP and the Microsoft servers is done with the proprietary MMS (Microsoft Media Server) protocol, which typically attempts to stream content over RTSP/RTP; if that fails because of firewalls, it attempts to retrieve content over HTTP.
- **Muze:** Provides an audio sample service to retailers, such as BestBuy and Yahoo. Music samples selected at these retailer sites actually come from Muze, and are streamed through WMP. Muze, Rhapsody, YouTube, and many other streaming content providers use content distribution networks (CDNs) to distribute their content, as discussed in Section 7.3.
- **YouTube:** The immensely popular video-sharing service uses a Flash-based client (embedded in the Web page). Communication between the client and the YouTube servers is done over HTTP.

What is in store for the future? Today most of the streaming video content is low-quality, encoded at rates of 500 kbps or less. Video quality will certainly improve as broadband and fiber-to-the-home Internet access become more pervasive. And very possibly our handheld music players will no longer store music—instead we'll get it all, on-demand, from wireless channels!

7.4 Protocols for Real-Time Interactive Applications

Real-time interactive applications, including Internet phone and video conferencing, promise to drive much of the future Internet growth. It is therefore not surprising that standards bodies, such as the IETF and ITU, have been busy for many years (and continue to be busy!) at hammering out standards for this class of applications. With the appropriate standards in place for real-time interactive applications, independent companies will be able to create new and compelling products that interoperate with each other. In this section we examine RTP, SIP, and H.323 for real-time interactive applications. All three sets of standards are enjoying widespread implementation in industry products.

7.4.1 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 3550, is such a standard. RTP can be used for transporting common formats such as PCM, GSM, and MP3 for sound and MPEG and H.263 for video. It can also be used for transporting proprietary sound and video formats. Today, RTP enjoys widespread implementation in hundreds of products and research prototypes. It is also complementary to other important real-time interactive protocols, including SIP and H.323.

In this section we provide an introduction to RTP and to its companion protocol, RTCP. We also encourage you to visit Henning Schulzrinne's RTP site [Schulzrinne-RTP 2007], which provides a wealth of information on the subject.

Also, you may want to visit the RAT site [RAT 2007], which documents an Internet phone application that uses RTP.

RTP Basics

RTP typically runs on top of UDP. The sending side encapsulates a media chunk within an RTP packet, then encapsulates the packet in a UDP segment, and then hands the segment to IP. The receiving side extracts the RTP packet from the UDP segment, then extracts the media chunk from the RTP packet, and then passes the chunk to the media player for decoding and rendering.

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 sending side 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 RTP header is normally 12 bytes. 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 play back 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. In Section 7.4.3 we'll see that RTP is often used in conjunction with the Internet telephony standards.

It should be emphasized that RTP does not provide any mechanism to ensure timely delivery of data or provide other quality-of-service (QoS) guarantees; it does not even guarantee delivery of packets or prevent out-of-order delivery of packets. Indeed, RTP encapsulation is seen only 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 video conference between two participants, four RTP streams could be opened—two streams for transmitting the audio (one in each direction) and two streams for transmitting the video (again, one in each direction). However, many popular encoding techniques—including MPEG 1 and MPEG 2—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 video conference application, belong to an **RTP session**.

RTP Packet Header Fields

As shown in Figure 7.10, the four main RTP packet header fields are the payload type, sequence number, timestamp, and source identifier fields.

The payload type field in the RTP packet is 7 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 7.2 lists some of the audio payload types currently supported by RTP.

For a video stream, the payload type is used to indicate the type of video encoding (for example, motion JPEG, MPEG 1, MPEG 2, H.261). Again, the sender can change video encoding on the fly during a session. Table 7.3 lists some of the video payload types currently supported by RTP. The other important fields are the following:

- *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 preceding section, the receiver can use timestamps 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.

Figure 7.10 ♦ RTP header fields

Payload-Type Number	Audio Format	Sampling Rate	Rate
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	16 kHz	48–64 kbps
14	MPEG Audio	90 kHz	—
15	G.728	8 kHz	16 kbps

Table 7.2 ♦ Audio payload types supported by RTP

- *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 is 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.

Developing Software Applications with RTP

There are two approaches to developing an RTP-based networked application. The first approach is for the application developer to incorporate RTP by hand—that is, actually to write the code that performs RTP encapsulation at the sender side and RTP unraveling at the receiver side. The second approach is for the application developer to use existing RTP libraries (for C programmers) and Java classes (for

Payload-Type Number	Video Format
26	Motion JPEG
31	H.261
32	MPEG 1 video
33	MPEG 2 video

Table 7.3 ♦ Some video payload types supported by RTP

Java programmers), which perform the encapsulation and unraveling for the application. Since you may be itching to write your first multimedia networking application using RTP, let us now elaborate a little on these two approaches. (The programming assignment at the end of this chapter will guide you through the creation of an RTP application.) We'll do this in the context of unicast communication (rather than for multicast).

Recall from Chapter 2 that the UDP API requires the sending process to set, for each UDP segment it sends, the destination IP address and the destination port number before popping the packet into the UDP socket. The UDP segment will then wander through the Internet and (if the segment is not lost due to, for example, router buffer overflow) eventually arrive at the door of the receiving process for the application. This door is fully addressed by the destination IP address and the destination port number. In fact, any IP datagram containing this destination IP address and destination port number will be directed to the receiving process's UDP door. (The UDP API also lets the application developer set the UDP source port number; however, this value has no effect on which process the segment is sent to.) It is important to note that RTP does not mandate a specific port number. When the application developer creates an RTP application, the developer specifies the port numbers for the two sides of the application.

As part of the programming assignment for this chapter, you will write an RTP server that encapsulates stored video frames within RTP packets. You will do this by hand; that is, your application will grab a video frame, add the RTP headers to the frame to create an RTP packet, and then pass the RTP frame to the UDP socket. To do this, you will need to create placeholder fields for the various RTP headers, including a sequence number field and a timestamp field. And for each RTP packet that is created, you will have to set the sequence number and the timestamp appropriately. You will explicitly code all of these RTP operations into the sender side of your application. As shown in Figure 7.11, your API to the network will be the standard UDP socket API.

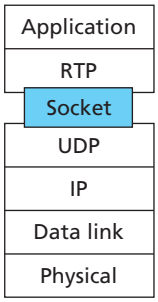

Figure 7.11 ♦ RTP is part of the application and lies above the UDP socket.

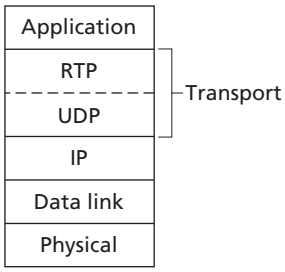

Figure 7.12 ♦ RTP can be viewed as a sublayer of the transport layer.

An alternative approach (not done in the programming assignment) is to use a Java RTP class (or a C RTP library for C programmers) to implement the RTP operations. With this approach, as shown in Figure 7.12, the application developer is given the impression that RTP is part of the transport layer, with an RTP/UDP API between the application layer and the transport layer. Without getting into the nitty-gritty details (as they are class/library-dependent), when sending a chunk of media into the API, the sending side of the application needs to provide the interface with the media chunk itself, a payload-type number, an SSRC, and a timestamp, along with a destination port number and an IP destination address. We mention here that the Java Media Framework (JMF) includes a complete RTP implementation.

7.4.2 RTP Control Protocol (RTCP)

RFC 3550 also specifies RTCP, a protocol that a networked multimedia application can use in conjunction with RTP. As shown in the multicast scenario in Figure 7.13, 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. (The RTCP port number is set to be equal to the RTP port number plus one.)

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 3550] 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

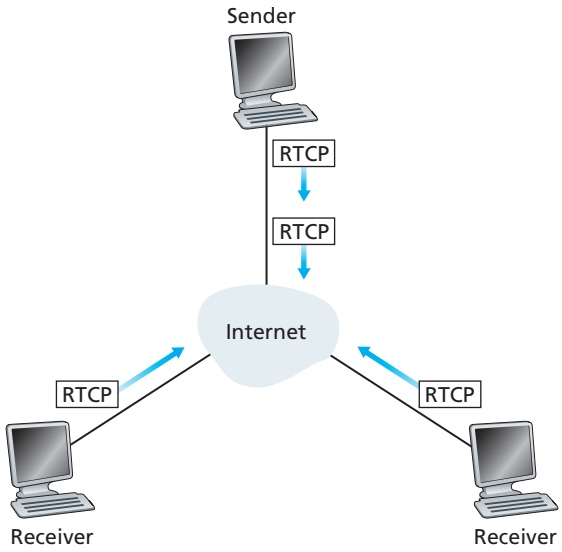

Figure 7.13 ♦ Both senders and receivers send RTCP messages.

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 all the session'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 a smoothed estimate of the variation in the 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 video conferencing 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* (i.e., 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 when the packet was created. Thus the RTCP sender report packets associate the sampling clock with 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 the 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

You may 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 percent 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 percent of 2 Mbps, or 100 kbps, as follows. The protocol gives 75 percent of this rate, or 75 kbps, to the receivers; it gives the remaining 25 percent 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

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

And the period for transmitting RTCP packets for a receiver is

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

7.4.3 SIP

Imagine a world in which, when you are working on your PC, your phone calls arrive over the Internet to your PC. When you get up and start walking around, your new phone calls are automatically routed to your PDA. And when you are driving in your car, your new phone calls are automatically routed to some Internet appliance in your car. In this same world, while participating in a conference call, you can access an address book to call and invite other participants into the conference. The other participants may be at their PCs, or walking with their PDAs, or driving their cars—no matter where they are, your invitation is transparently routed to them. In this same world, when you browse an individual's homepage, there will be a link "Call Me"; clicking on this link establishes an Internet phone session between your PC and the owner of the homepage (wherever that person might be).

In this world, there is no longer a circuit-switched telephone network. Instead, all calls pass over the Internet—from end to end. In this same world, companies no longer use private branch exchanges (PBXs), that is, local circuit switches for handling intracompany telephone calls. Instead, the intracompany phone traffic flows over the company's high-speed LAN.

All of this may sound like science fiction. And, of course, today's circuit-switched networks and PBXs are not going to disappear completely in the near future [Jiang 2001]. Nevertheless, protocols and products exist to turn this vision into a reality. Among the most promising protocols in this direction is the Session

Initiation Protocol (SIP), defined in [RFC 3261]. SIP is a lightweight protocol that does the following:

- It provides mechanisms for establishing calls between a caller and a callee over an IP network. It allows the caller to notify the callee that it wants to start a call. It allows the participants to agree on media encodings. It also allows participants to end calls.
- It provides mechanisms for the caller to determine the current IP address of the callee. Users do not have a single, fixed IP address because they may be assigned addresses dynamically (using DHCP) and because they may have multiple IP devices, each with a different IP address.
- It provides mechanisms for call management, such as adding new media streams during the call, changing the encoding during the call, inviting new participants during the call, call transfer, and call holding.

Setting Up a Call to a Known IP Address

To understand the essence of SIP, it is best to take a look at a concrete example. In this example, Alice is at her PC and she wants to call Bob, who is also working at his PC. Alice's and Bob's PCs are both equipped with SIP-based software for making and receiving phone calls. In this initial example, we'll assume that Alice knows the IP address of Bob's PC. Figure 7.14 illustrates the SIP call-establishment process.

In Figure 7.14, we see that an SIP session begins when Alice sends Bob an INVITE message, which resembles an HTTP request message. This INVITE message is sent over UDP to the well-known port 5060 for SIP. (SIP messages can also be sent over TCP.) The INVITE message includes an identifier for Bob (bob@193.64.210.89), an indication of Alice's current IP address, an indication that Alice desires to receive audio, which is to be encoded in format AVP 0 (PCM encoded μ-law) and encapsulated in RTP, and an indication that she wants to receive the RTP packets on port 38060. After receiving Alice's INVITE message, Bob sends an SIP response message, which resembles an HTTP response message. This response SIP message is also sent to the SIP port 5060. Bob's response includes a 200 OK as well as an indication of his IP address, his desired encoding and packetization for reception, and his port number to which the audio packets should be sent. Note that in this example Alice and Bob are going to use different audio-encoding mechanisms: Alice is asked to encode her audio with GSM whereas Bob is asked to encode his audio with PCM μ-law. After receiving Bob's response, Alice sends Bob an SIP acknowledgment message. After this SIP transaction, Bob and Alice can talk. (For visual convenience, Figure 7.14 shows Alice talking after Bob, but in truth they would normally talk at the same time.) Bob will encode and packetize the audio as requested and send the audio packets to port number 38060 at IP address

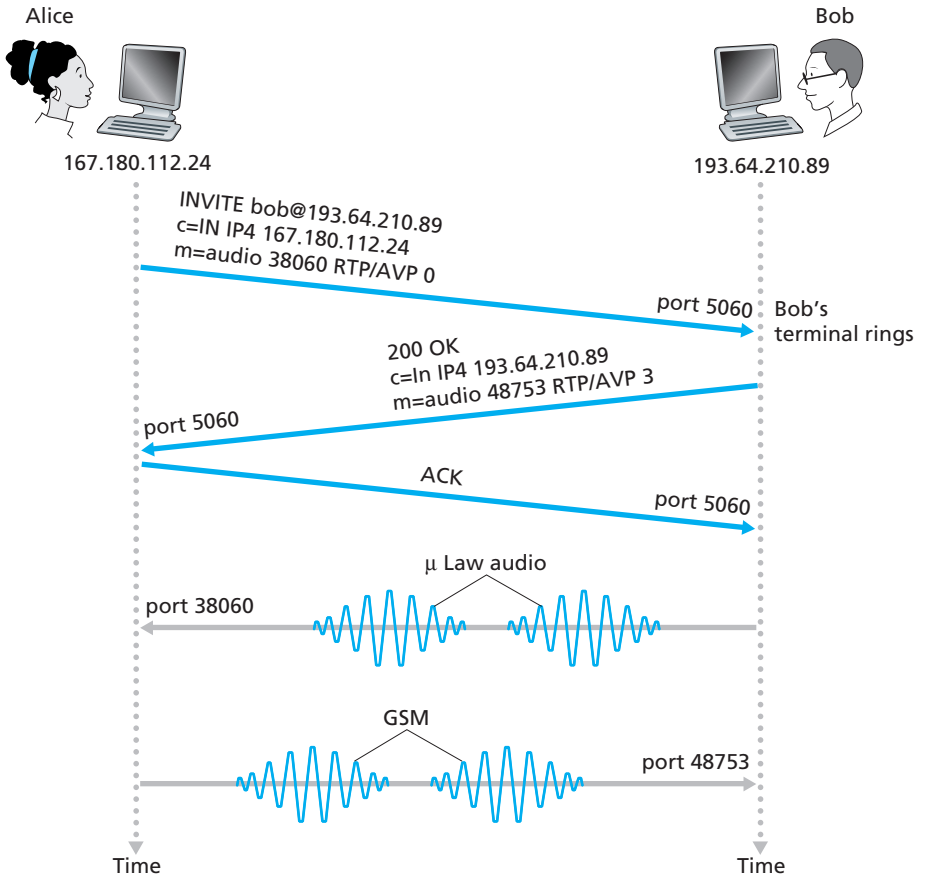

Figure 7.14 ♦ SIP call establishment when Alice knows Bob's IP address

167.180.112.24. Alice will also encode and packetize the audio as requested and send the audio packets to port number 48753 at IP address 193.64.210.89.

From this simple example, we have learned a number of key characteristics of SIP. First, SIP is an out-of-band protocol: the SIP messages are sent and received in sockets that are different from those used for sending and receiving the media data. Second, the SIP messages themselves are ASCII-readable and resemble HTTP messages. Third, SIP requires all messages to be acknowledged, so it can run over UDP or TCP.

In this example, let's consider what would happen if Bob does not have a PCM μ-law codec for encoding audio. In this case, instead of responding with 200 OK, Bob would likely respond with a 600 Not Acceptable and list in the message all the codecs he can use. Alice would then choose one of the listed codecs and send another INVITE message, this time advertising the chosen codec. Bob could also simply

reject the call by sending one of many possible rejection reply codes. (There are many such codes, including "busy," "gone," "payment required," and "forbidden.")

SIP Addresses

In the previous example, Bob's SIP address is sip:bob@193.64.210.89. However, we expect many—if not most—SIP addresses to resemble e-mail addresses. For example, Bob's address might be sip:bob@domain.com. When Alice's SIP device sends an INVITE message, the message would include this e-mail-like address; the SIP infrastructure would then route the message to the IP device that Bob is currently using (as we'll discuss below). Other possible forms for the SIP address could be Bob's legacy phone number or simply Bob's first/middle/last name (assuming it is unique).

An interesting feature of SIP addresses is that they can be included in Web pages, just as people's e-mail addresses are included in Web pages with the mailto URL. For example, suppose Bob has a personal homepage, and he wants to provide a means for visitors to the homepage to call him. He could then simply include the URL sip:bob@domain.com. When the visitor clicks on the URL, the SIP application in the visitor's device is launched and an INVITE message is sent to Bob.

SIP Messages

In this short introduction to SIP, we'll not cover all SIP message types and headers. Instead, we'll take a brief look at the SIP INVITE message, along with a few common header lines. Let us again suppose that Alice wants to initiate an IP phone call to Bob, and this time Alice knows only Bob's SIP address, bob@domain.com, and does not know the IP address of the device that Bob is currently using. Then her message might look something like this:

```
INVITE sip:bob@domain.com SIP/2.0
Via: SIP/2.0/UDP 167.180.112.24
From: sip:alice@hereway.com
To: sip:bob@domain.com
Call-ID: a2e3a@pigeon.hereway.com
Content-Type: application/sdp
Content-Length: 885

c=IN IP4 167.180.112.24
m=audio 38060 RTP/AVP 0
```

The INVITE line includes the SIP version, as does an HTTP request message. Whenever an SIP message passes through an SIP device (including the device that originates the message), it attaches a Via header, which indicates the IP address of the device. (We'll see soon that the typical INVITE message passes through many SIP

devices before reaching the callee's SIP application.) Similar to an e-mail message, the SIP message includes a From header line and a To header line. The message includes a Call-ID, which uniquely identifies the call (similar to the message-ID in e-mail). It includes a Content-Type header line, which defines the format used to describe the content contained in the SIP message. It also includes a Content-Length header line, which provides the length in bytes of the content in the message. Finally, after a carriage return and line feed, the message contains the content. In this case, the content provides information about Alice's IP address and how Alice wants to receive the audio.

Name Translation and User Location

In the example in Figure 7.14, we assumed that Alice's SIP device knew the IP address where Bob could be contacted. But this assumption is quite unrealistic, not only because IP addresses are often dynamically assigned with DHCP, but also because Bob may have multiple IP devices (for example, different devices for his home, work, and car). So now let us suppose that Alice knows only Bob's e-mail address, bob@domain.com, and that this same address is used for SIP-based calls. In this case, Alice needs to obtain the IP address of the device that the user bob@domain.com is currently using. To find this out, Alice creates an INVITE message that begins with INVITE bob@domain.com SIP/2.0 and sends this message to an **SIP proxy**. The proxy will respond with an SIP reply that might include the IP address of the device that bob@domain.com is currently using. Alternatively, the reply might include the IP address of Bob's voicemail box, or it might include a URL of a Web page (that says "Bob is sleeping. Leave me alone!"). Also, the result returned by the proxy might depend on the caller: if the call is from Bob's wife, he might accept the call and supply his IP address; if the call is from Bob's mother-in-law, he might respond with the URL that points to the I-am-sleeping Web page!

Now, you are probably wondering, how can the proxy server determine the current IP address for bob@domain.com? To answer this question, we need to say a few words about another SIP device, the **SIP registrar**. Every SIP user has an associated registrar. Whenever a user launches an SIP application on a device, the application sends an SIP register message to the registrar, informing the registrar of its current IP address. For example, when Bob launches his SIP application on his PDA, the application would send a message along the lines of:

```
REGISTER sip:domain.com SIP/2.0
Via: SIP/2.0/UDP 193.64.210.89
From: sip:bob@domain.com
To: sip:bob@domain.com
Expires: 3600
```

Bob's registrar keeps track of Bob's current IP address. Whenever Bob switches to a new SIP device, the new device sends a new register message, indicating the

new IP address. Also, if Bob remains at the same device for an extended period of time, the device will send refresh register messages, indicating that the most recently sent IP address is still valid. (In the example above, refresh messages need to be sent every 3600 seconds to maintain the address at the registrar server.) It is worth noting that the registrar is analogous to a DNS authoritative name server: the DNS server translates fixed host names to fixed IP addresses; the SIP registrar translates fixed human identifiers (for example, bob@domain.com) to dynamic IP addresses. Often SIP registrars and SIP proxies are run on the same host.

Now let's examine how Alice's SIP proxy server obtains Bob's current IP address. From the preceding discussion we see that the proxy server simply needs to forward Alice's INVITE message to Bob's registrar/proxy. The registrar/proxy could then forward the message to Bob's current SIP device. Finally, Bob, having now received Alice's INVITE message, could send an SIP response to Alice.

As an example, consider Figure 7.15, in which jim@umass.edu, currently working on 217.123.56.89, wants to initiate a Voice over IP (VoIP) session with keith@upenn.edu, currently working on 197.87.54.21. The following steps are taken: (1) Jim sends an INVITE message to the umass SIP proxy. (2) The proxy does a DNS lookup on the SIP registrar upenn.edu (not shown in diagram) and then

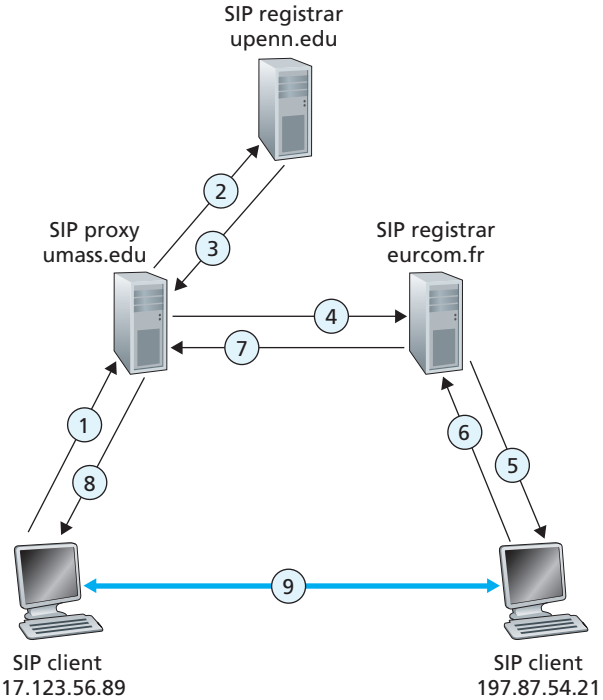

Figure 7.15 ♦ Session initiation, involving SIP proxies and registrars

forwards the message to the registrar server. (3) Because keith@upenn.edu is no longer registered at the upenn registrar, the upenn registrar sends a redirect response, indicating that it should try keith@eurecom.fr. (4) The umass proxy sends an INVITE message to the eurecom SIP registrar. (5) The eurecom registrar knows the IP address of keith@eurecom.fr and forwards the INVITE message to the host 197.87.54.21, which is running Keith's SIP client. (6–8) An SIP response is sent back through registrars/proxies to the SIP client on 217.123.56.89. (9) Media is sent directly between the two clients. (There is also an SIP acknowledgment message, which is not shown.)

Our discussion of SIP has focused on call initiation for voice calls. SIP, being a signaling protocol for initiating and ending calls in general, can be used for video conference calls as well as for text-based sessions. In fact, SIP has become a fundamental component in many instant messaging applications. Readers desiring to learn more about SIP are encouraged to visit Henning Schulzrinne's SIP Web site [Schulzrinne-SIP 2007]. In particular, on this site you will find open source software for SIP clients and servers [SIP Software 2007].

7.4.4 H.323

As an alternative to SIP, H.323 is a popular standard for real-time audio and video conferencing among end systems on the Internet. As shown in Figure 7.16, the standard also covers how end systems attached to the Internet communicate with

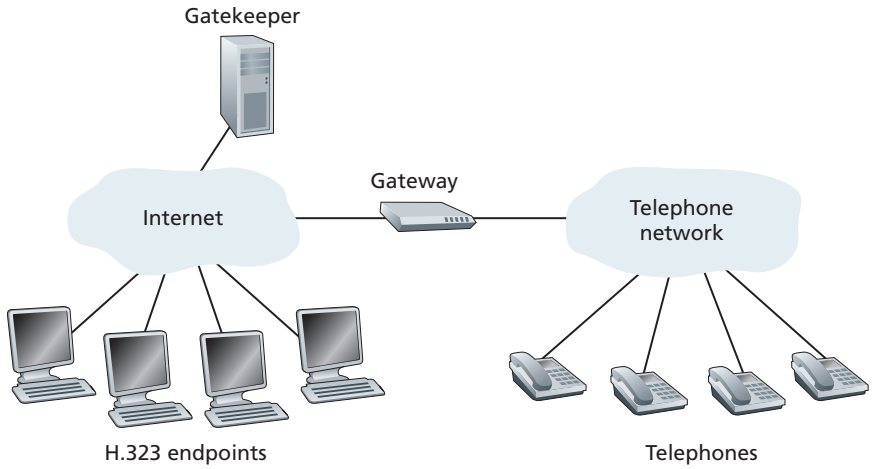

Figure 7.16 ♦ H.323 end systems attached to the Internet can communicate with telephones attached to a circuit-switched telephone network.

telephones attached to ordinary circuit-switched telephone networks. (SIP does this as well, although we did not discuss it.) The H.323 gatekeeper is a device similar to an SIP registrar.

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

- A specification for how end points 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 end points to agree on a common encoding.
- A specification for how audio and video chunks are encapsulated and sent over the network. In particular, H.323 mandates RTP for this purpose.
- A specification for how end points communicate with their respective gatekeepers.
- A specification for how Internet phones communicate through a gateway with ordinary phones in the PSTN.

Minimally, each H.323 end point *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 end point 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. Video capabilities for an H.323 end point are optional. However, if an end point does support video, then it must (at the very least) support the QCIF H.261 (176 x 144 pixels) video standard.

H.323 is a comprehensive umbrella standard, which, in addition to the standards and protocols described above, mandates an H.245 control protocol, a Q.931 signaling channel, and an RAS protocol for registration with the gatekeeper.

We conclude this section by highlighting some of the most important differences between H.323 and SIP.

- H.323 is a complete, vertically integrated suite of protocols for multimedia conferencing: signaling, registration, admission control, transport, and codecs.
- SIP, on the other hand, addresses only session initiation and management and is a single component. SIP works with RTP but does not mandate it. It works with G.711 speech codecs and QCIF H.261 video codecs but does not mandate them. It can be combined with other protocols and services.
- H.323 comes from the ITU (telephony), whereas SIP comes from the IETF and borrows many concepts from the Web, DNS, and Internet e-mail.
- H.323, being an umbrella standard, is large and complex. SIP uses the KISS principle: keep it simple, stupid.

For an excellent discussion of H.323, SIP, and VoIP in general, see [Hersent 2000].

7.5 Providing Multiple Classes of Service

In previous sections we learned how sequence numbers, timestamps, FEC, RTP, and H.323 can be used by multimedia applications in today's Internet. CDNs represent a system-wide solution for distributing multimedia content. But are these techniques alone enough to support reliable and robust multimedia applications, such as an IP telephony service that is equivalent to that in today's telephone network? Before answering this question, let's recall again that today's Internet provides a best-effort service to all of its applications; that is, it does not make any promises about the 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. Because every packet, including delay-sensitive audio and video packets, is treated equally at the routers, 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.

But if the goal is to provide a service model that provides something more than the one-size-fits-all best-effort service in today's Internet, exactly what type of service is to be provided? One simple enhanced service model is to divide traffic into classes, and provide different levels of service to these different classes of traffic. For example, an ISP might well want to provide a higher class of service to delay-sensitive Voice over IP or teleconferencing traffic (and charge more for this service!) than to elastic traffic such as FTP or HTTP. We're all familiar with different classes of service from our everyday lives—first-class airline passengers get better service than business class passengers, who in turn get better service than those of us who fly economy class; VIPs are provided immediate entry to events while everyone else waits in line; elders are revered in some countries and provided seats of honor and the finest food at a table.

It's important to note that such differential service is provided among *aggregates* of traffic, i.e., among classes of traffic, not among individual connections. For example, all first-class passengers are handled the same (with no first-class passenger receiving any better treatment than any other first-class passenger), just as all VoIP packets would receive the same treatment within the network, independent of the particular end-end connection to which they belong. As we will see, by dealing with a small number of traffic aggregates, rather than a large number of individual connections, the new network mechanisms required to provide better-than-best service can be kept relatively simple.

The early Internet designers clearly had this notion of multiple classes of service in mind. Recall the type-of-service (ToS) field in the IPv4 header in Figure 4.13. IEN123 [ISI 1979] describes the ToS field also present in an ancestor of the IPv4 datagram as follows: "The Type of Service [field] provides an indication of the abstract

parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important that other traffic." Even three decades ago, the vision of providing different levels of service to different levels of traffic was clear! However, it's taken us an equally long period of time to realize this vision.

We'll begin our study in Section 7.5.1 by considering several scenarios that will motivate the need for specific mechanisms for supporting multiple classes of service. We'll then cover two important topics—link-level scheduling and packet classification/policing in Section 7.5.2. In Section 7.5.3, we'll cover Diffserv—the Internet's current standard for providing differentiated service.

7.5.1 Motivating Scenarios

Figure 7.17 shows a simple network scenario. 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 H1 and H2 exceeds 1.5 Mbps. Let's now consider several scenarios, each of which will provide us with important insight into the need for specific mechanisms for supporting multiple classes of service.

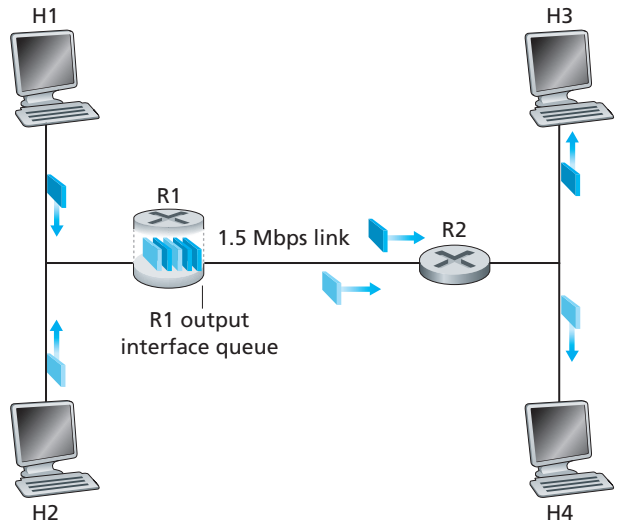

Figure 7.17 ♦ A simple network with two applications

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

Scenario 1 is illustrated in Figure 7.18. 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.

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. 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 insight into mechanisms needed to provide multiple classes of traffic:

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

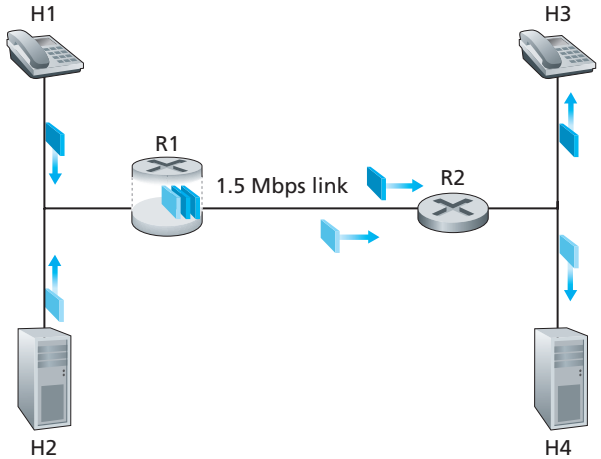

Figure 7.18 ♦ Competing audio and FTP applications

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" (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 insight 1:

> **Insight 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.

Scenario 3: A Misbehaving Audio Application and an FTP Transfer

Suppose now that somehow (by use of mechanisms that we'll 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, they 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 classes of traffic and also possibly among flows within the same traffic class, in order to protect one flow from another, misbehaving flow. The notion of protecting individual flows within a given traffic class from each other contradicts our earlier observation that packets from all flows within a class should be treated the same. In practice, packets within a class are indeed treated the same at routers within the network core. However, at the edge of the network, packets within a given flow may be monitored to ensure that the aggregate rate of an individual flow does not exceed a given value.

These considerations give rise to our second insight:

Insight 2: It is desirable to provide a degree of isolation among traffic classes and among flows, so that one class or flow is not adversely affected by another that misbehaves.

In the following section, we will examine several specific mechanisms for providing this isolation among traffic classes or flows. We note here that two broad approaches can be taken. First, it is possible to police traffic, as shown in Figure 7.19. If a traffic class or 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 these criteria are 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.

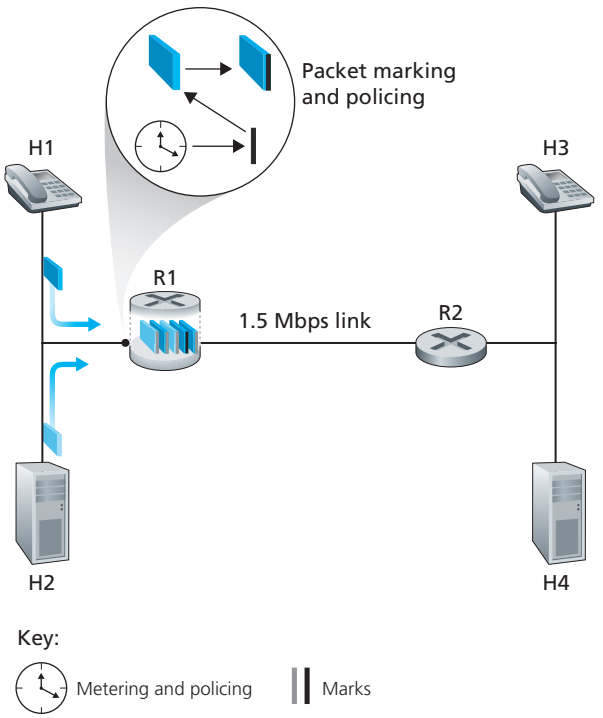

Figure 7.19 ♦ Policing (and marking) the audio and FTP traffic flows

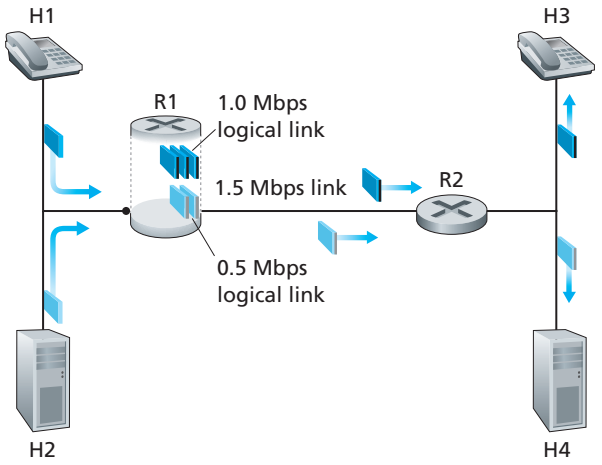

Figure 7.20 ♦ Logical isolation of audio and FTP application flows

In Figure 7.19, the packet classification and marking mechanism (Observation 1) and the policing mechanism (Observation 2) are co-located at the edge of the network, either in the end system or at an edge router.

An alternative approach for providing isolation among traffic classes or flows is for the link-level packet-scheduling mechanism to explicitly allocate a fixed amount of link bandwidth to each class or flow. For example, the audio flow could be allocated 1 Mbps 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 7.20.

With strict enforcement of the link-level allocation of bandwidth, a class or 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 others. 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 class or flow to use another's unused bandwidth at any given point in time. This consideration gives rise to our third insight:

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

7.5.2 Scheduling and Policing Mechanisms

Now that we have gained insight into the mechanisms needed to provide different classes of service, let's now consider two of the most important mechanisms—scheduling and policing—in detail.

Scheduling Mechanisms

Recall from our discussion in Section 1.3 and Section 4.3 that packets belonging to various network flows are multiplexed 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**. Let us now consider several of the most important link-scheduling disciplines in more detail.

First-In-First-Out (FIFO)

Figure 7.21 shows the queuing model abstractions for the FIFO link-scheduling discipline. Packets arriving at the link output queue wait 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.

The FIFO (also known as first-come-first-served, or FCFS) scheduling discipline 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 7.22 shows 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

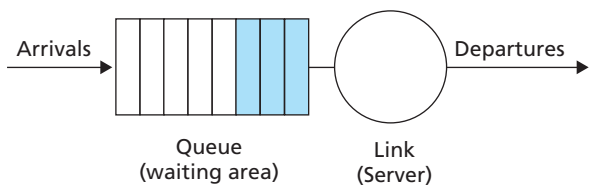

Figure 7.21 ♦ FIFO queuing abstraction

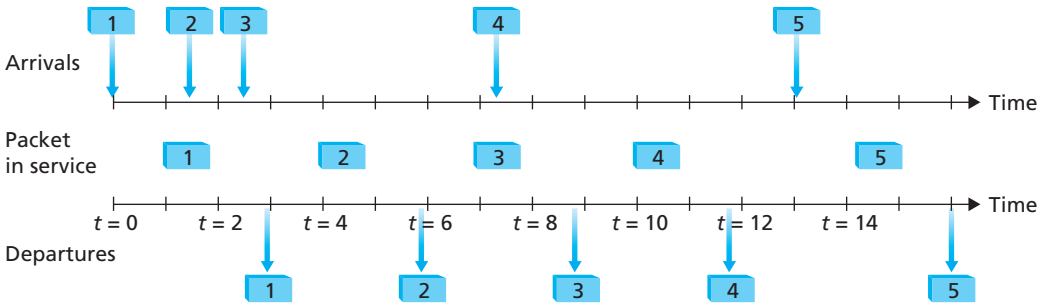

Figure 7.22 ♦ The FIFO queue in operation

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.

Priority Queuing

Under **priority queuing**, packets arriving at the output link are classified into priority classes at the output queue, as shown in Figure 7.23. 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 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

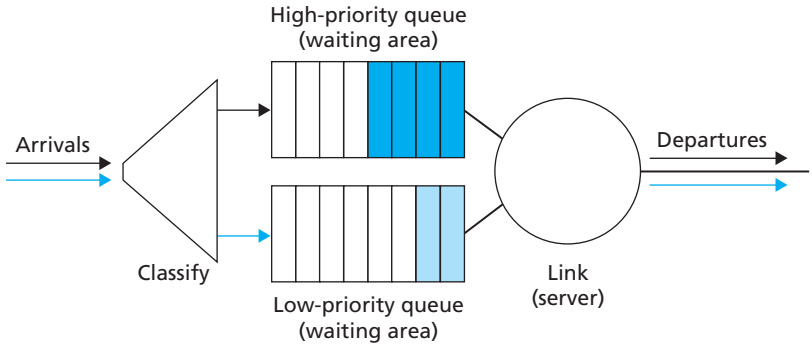

Figure 7.23 ♦ Priority queuing model

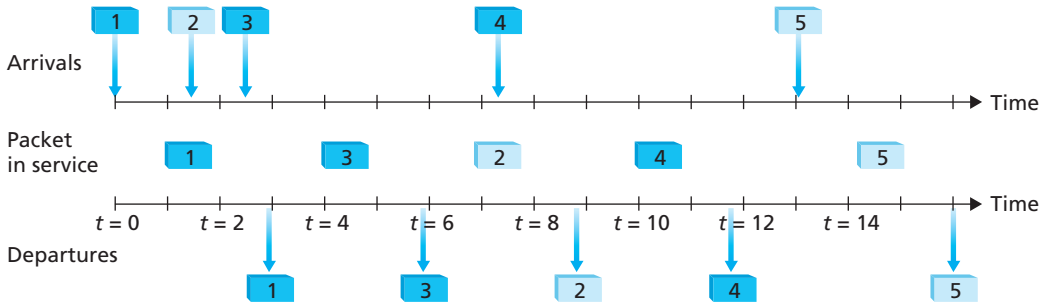

Figure 7.24 ♦ Operation of the priority queue

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 7.24 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 2 (a low-priority packet). Under a nonpreemptive 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.

Round Robin and Weighted Fair Queuing (WFQ)

Under the **round robin queuing discipline**, packets are 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 7.25 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

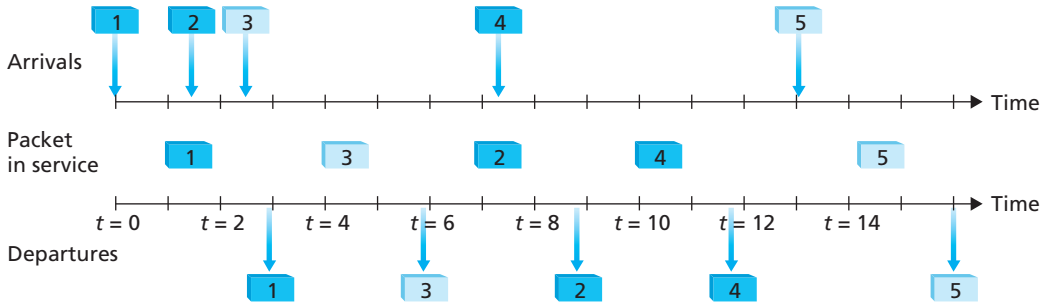

Figure 7.25 ♦ Operation of the two-class round robin queue

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.

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 7.26. Arriving packets are classified and queued in the appropriate per-class waiting area. As in round robin scheduling, a WFQ scheduler will serve classes in a circular manner—first serving class 1, then serving class 2, then serving class 3, and then (assuming there are three

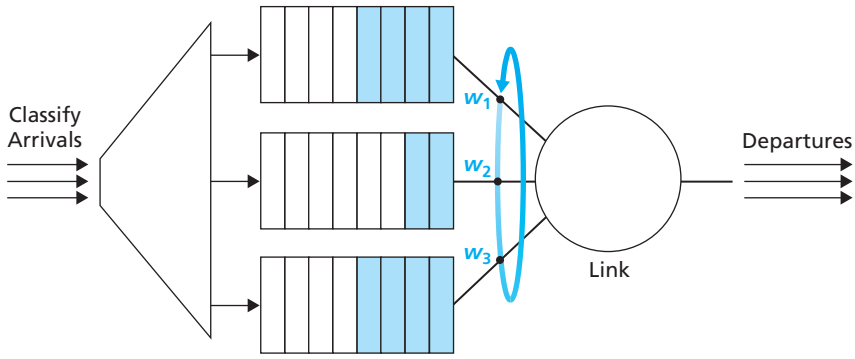

Figure 7.26 ♦ Weighted fair queuing (WFQ)

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 when it finds an empty class queue.

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 $w_i/(\sum w_j)$, 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 $w_i/(\sum w_j)$ of the bandwidth. Thus, for a link with transmission rate *R,* class *i* will always achieve a throughput of at least $R \cdot w_i/(\sum w_j)$. 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 2007].

Policing: The Leaky Bucket

One of our insights from Section 7.5.1 was that policing, the regulation of the rate at which a class or flow (we will assume the unit of policing is a flow in our discussion below) is allowed to inject packets into the network, is an important QoS mechanism. 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, 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 7.27, 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.

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*

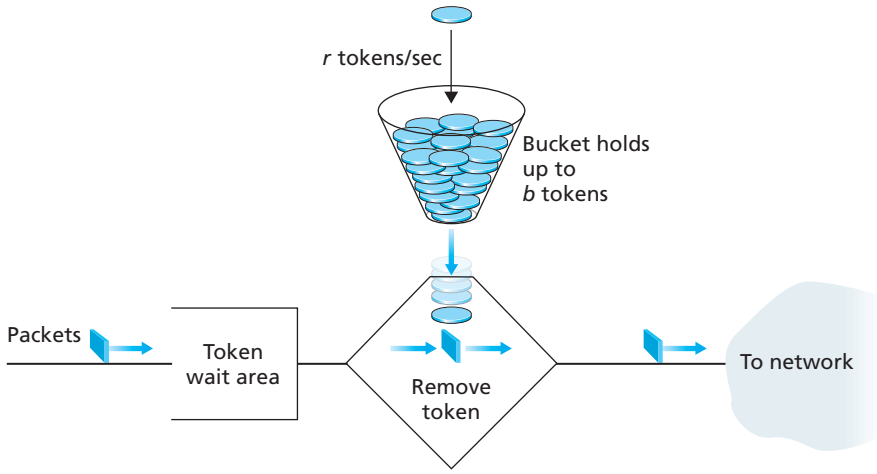

Figure 7.27 ♦ The leaky bucket policer

is $rt + b$. Thus, the token-generation rate, r, serves to limit the long-term average rate at which packets 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 = Provable Maximum Delay in a Queue

We'll soon examine the so-called Intserv and Diffserv approaches for providing quality of service in the Internet. We'll 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 link 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 or a collection of many such connections, see Figure 7.28.

Recall from our discussion of WFQ that each flow, i, is guaranteed to receive a share of the link bandwidth equal to at least $R \cdot w_i/(\sum w_j)$, 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

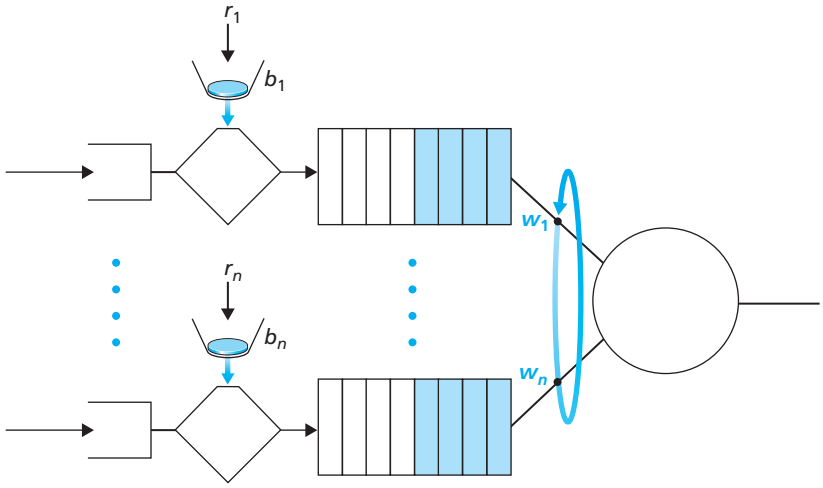

Figure 7.28 ♦ n multiplexed leaky bucket flows with WFQ scheduling

of at least $R \cdot w_i/(\sum w_j)$ packet/sec, the last of these packets will then have a maximum delay, $d_{\max,}$ until its transmission is completed, where

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

The rationale behind 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 $R \cdot w_1/(\sum w_j)$ packets per second, then the amount of time until the last bit of the last packet is transmitted cannot be more than $b_1/(R \cdot w_1/(\sum w_j))$. A homework problem asks you to prove that as long as $r_1 < R \cdot w_1/(\sum w_j)$, then $d_{\max}$ is indeed the maximum delay that any packet in flow 1 will ever experience in the WFQ queue.

7.5.3 Diffserv

The Internet Diffserv architecture [RFC 2475; Kilkki 1999] aims to provide service differentiation—that is, the ability to handle different "classes" of traffic in different ways within the Internet—and to do so in a scalable and flexible manner. 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 Diffserv architecture is flexible in the sense that it does not define specific services or service classes. Instead, Diffserv 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.

Differentiated Services: A Simple Scenario

To set the framework for defining the architectural components of the differentiated service (Diffserv) model, let's begin with the simple network shown in Figure 7.29. In this section, we describe one possible use of the Diffserv components. Many other variations are possible, as described in RFC 2475. Our goal here is to provide an introduction to the key aspects of Diffserv, rather than to describe the architectural model in exhaustive detail. Readers interested in learning more about Diffserv are encouraged to see the comprehensive book [Kilkki 1999].

The Diffserv 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

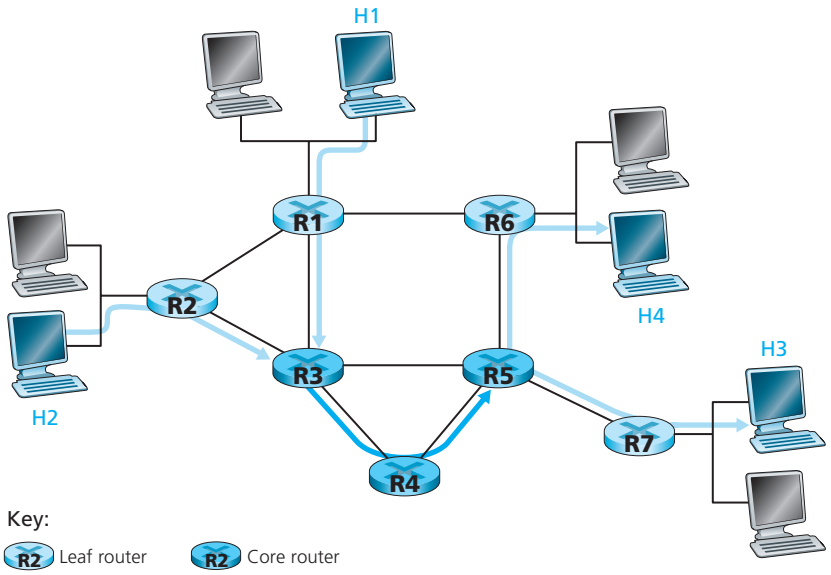

Figure 7.29 ♦ A simple Diffserv network example

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 7.29, 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.

- *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 7.29 receive the same marking as packets being sent 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 services 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 discothèque, a concert, or a football game), people entering the event receive a pass of one type or another: VIP passes for Very Important People; over-21 passes for people who are 21 years old or older (for example, if alcoholic drinks are to be served); backstage passes at concerts; press passes for reporters; even an ordinary pass for the Ordinary Person. These passes are typically distributed upon 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.

Diffserv Traffic Classification and Conditioning

Figure 7.30 provides a logical view of the classification and marking functions 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, and protocol ID) and steers the packet to the appropriate marking function. A packet's

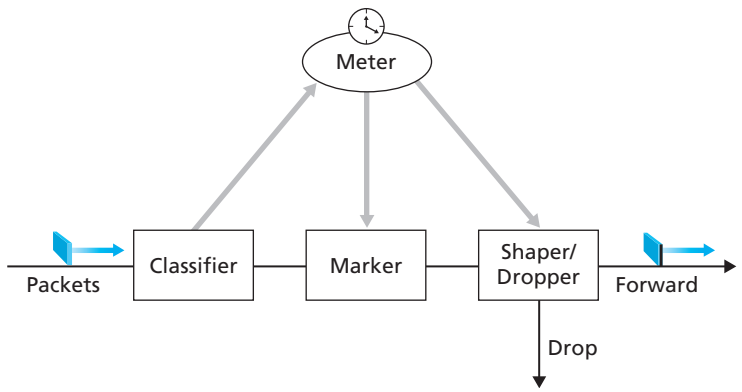

Figure 7.30 ♦ Logical view of packet classification and traffic conditioning at the end router

mark is carried within the DS field [RFC 3260] 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 and the IPv6 traffic class fields that we discussed in Chapter 4.

In some cases, an end user may have agreed to limit its packet-sending rate to conform to a 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 previously 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 and are forwarded along their route to the destination. On the other hand, if the traffic profile is violated, 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 7.30, 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 remark, forward, delay, or drop a packet is a policy issue determined by the network administrator and is *not* specified in the Diffserv architecture.

Per-Hop Behaviors

So far, we have focused on the edge functions in the Diffserv architecture. The second key component of the Diffserv architecture involves the **per-hop behavior** (PHB) performed by Diffserv-capable routers. 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 it:

- A PHB can result in different classes of traffic receiving different performance (that is, different externally observable forwarding behaviors).
- 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 WFQ queue versus a FCFS 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.

Currently, two PHBs have been defined: an expedited forwarding (EF) PHB [RFC 3246] 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.

Diffserv Retrospective

For the past 20 years there have been numerous attempts (for the most part, unsuccessful) to introduce QoS into packet-switched networks. The various attempts have failed so far more for economic and legacy reasons that because of technical reasons. These attempts include end-to-end ATM networks and TCP/IP networks. Let's take a look at a few of the issues involved in the context of Diffserv (which we will study briefly in the following section).

So far we have implicitly assumed that Diffserv is deployed within a single administrative domain. The more typical case is where an end-to-end service must be fashioned from multiple ISPs sitting between communicating end systems. In order to provide end-to-end Diffserv service, all the ISPs between the end systems not only must provide this service, but most also cooperate and make settlements in order to offer end customers true end-end service. Without this kind of cooperation, ISPs directly selling Diffserv service to customers will find themselves repeatedly saying: "Yes, we know you paid extra, but we don't have a service agreement with one of our higher-tier ISPs. I'm sorry that there were many gaps in your VoIP call!"

Even within a single administrative domain, Diffserv alone is not enough to provide quality of service guarantees to a particular class of service. Diffserv only allows different classes of traffic to receive different levels of performance. If a network is severely under-dimensioned, even the high-priority class of traffic may receive unacceptably bad performance. Thus, to be effective, Diffserv must be coupled with proper network dimensioning (see Section 7.3.5). Diffserv *can*, however,

make an ISP's investment in network capacity go farther. By making resources available to high-priority (and high-paying) classes of traffic whenever needed (at the expense of the lower-priority classes of traffic), the ISP can deliver a high level of performance to these high-priority classes. When these resources are not needed by the high-priority classes, they can be used by the lower-priority traffic classes (who have presumably paid less for this lower class of service).

Another concern with these advanced services is the need to police and possibly shape traffic, which may turn out to be complex and costly. One also needs to bill the services differently, most likely by volume rather than with a fixed monthly fee as currently done by most ISPs—another costly requirement for the ISP. Finally, if Diffserv were actually in place and the network ran at only moderate load, most of the time there would be no perceived difference between a best-effort service and a Diffserv service. Indeed, today, end-to-end delay is usually dominated by access rates and router hops rather than by queuing delays in the routers. Imagine the unhappy Diffserv customer who has paid for premium service but finds that the best-effort service being provided to others almost always has the same performance as premium service!

7.6 Providing Quality of Service Guarantees

In the previous section we have seen that packet marking and policing, traffic isolation, and link-level scheduling can provide one class of service with better performance than another. Under certain scheduling disciplines, such as priority scheduling, the lower classes of traffic are essentially "invisible" to the highest-priority class of traffic. With proper network dimensioning, the highest class of service can indeed achieve extremely low packet loss and delay—essentially circuit-like performance. But can the network *guarantee* that an on-going flow in a high-priority traffic class will continue to receive such service throughout the flow's duration using only the mechanisms that we have described so far? It can not. In this section, we'll see why yet additional network mechanisms and protocols are needed to provide quality of service *guarantees*.

7.6.1 A Motivating Example

Let's return to our scenario from section 7.5.1 and consider two 1 Mbps audio applications transmitting their packets over the 1.5 Mbps link, as shown in Figure 7.31. The combined data rate of the two flows (2 Mbps) exceeds the link capacity. Even with classification and marking, isolation of flows, and sharing of unused bandwidth, (of which there is none), this is clearly a losing proposition. There is simply not enough bandwidth to accommodate the needs of both applications at the same time. If the two applications equally share the bandwidth, each would receive only

0.75 Mbps. Looked at another way, each application would lose 25 percent of its transmitted packets. This is such an unacceptably low QoS that both audio applications are completely unusable; there's no need even to transmit any audio packets in the first place.

Given that the two applications in Figure 7.31 cannot both be satisfied simultaneously, what should the network do? Allowing both to proceed with an unusable QoS wastes network resources on application flows that ultimately provide no utility to the end user. The answer is hopefully clear—one of the application flows should be blocked (i.e., denied access to the network), while the other should be allowed to proceed on, using the full 1 Mbps needed by the application. 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, 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.

By explicitly admitting or blocking flows based on their resource requirements, and the source-requirements of already-admitted flows, the network can guarantee that admitted flows will be able to receive their requested QoS. 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. This then is our fourth insight (in addition to the three earlier insights from Section 7.5.1) into the mechanisms needed to provide QoS.

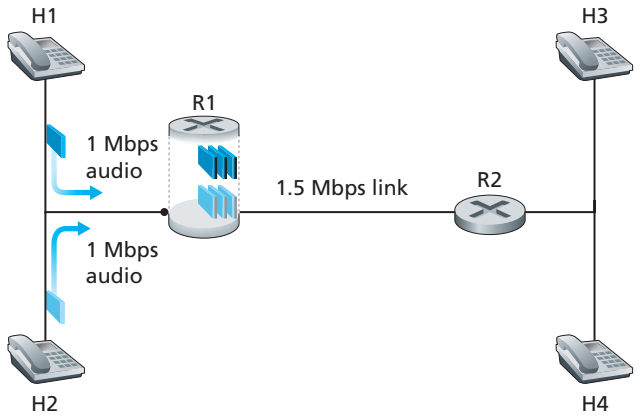

Figure 7.31 ♦ Two competing audio applications overloading the R1-to-R2 link

Insight 4: If sufficient resources will not always be available, and QoS is to be *guaranteed*, 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).

7.6.2 Resource Reservation, Call Admission, Call Setup

Our motivating example highlights the need for several new network mechanisms and protocols if a call (an end-end flow) is to be guaranteed a given quality of service once it begins:

- *Resource reservation.* The only way to *guarantee* that a call will have the resources (link bandwidth, buffers) needed to meet its desired QoS is to explicitly allocate those resources to the call—a process known in networking parlance as **resource reservation**. Once resources are reserved, the call has on-demand access to these resources throughout its duration, regardless of the demands of all other calls. If a call reserves and receives a guarantee of x Mbps of link bandwidth, and never transmits at a rate greater than x, the call will see loss- and delay-free performance.
- *Call admission.* If resources are to be reserved, then the network must have a mechanism for calls to request and reserve resources—a process known as call admission. Since resources are not infinite, a call making a call admission request will be denied admission, i.e., be blocked, if the requested resources are not available. Such a call admission is performed by the telephone network—we request resources when we dial a number. If the circuits (TDMA slots) needed to complete the call are available, the circuits are allocated and the call is completed. If the circuits are not available, then the call is blocked, and we receive a busy signal. A blocked call can try again to gain admission to the network, but it is not allowed to send traffic into the network until it has successfully completed the call admission process.

  Of course, just as the restaurant manager from Section 1.3.1 should not accept reservations for more tables than the restaurant has, a router that allocates link bandwidth should not allocate more than is available at that link. Typically, a call may reserve only a fraction of the link's bandwidth, and so a router may allocate link bandwidth to more than one call. However, the sum of the allocated bandwidth to all calls should be less than the link capacity.

1. *Call setup signaling.* The call admission process described above requires that a call be able to reserve sufficient resources at each and every network router on its source-to-destination path to ensure that its end-to-end QoS requirement is met. 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. A signaling protocol is needed to coordinate these various activities—the per-hop allocation of local resources, as well as the overall end-end decision of whether or not the call has been able to reserve sufficient resources at each and every router on the end-end path. This is the job of the **call setup protocol**.

Figure 7.32 depicts the call setup process. Let's 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 a call's QoS requirement, that call 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 Internet's Intserv architecture, the so-called Rspec (R for reservation) [RFC 2215] defines the specific QoS being requested by a call; the so-called Tspec (T for traffic) [RFC 2210] characterizes the traffic the sender will be sending into the network or that the receiver will be receiving from the network, respectively. The specific form of the Rspec and Tspec will vary, depending on the service requested, as discussed below. In ATM networks, the user traffic description and the QoS parameter information

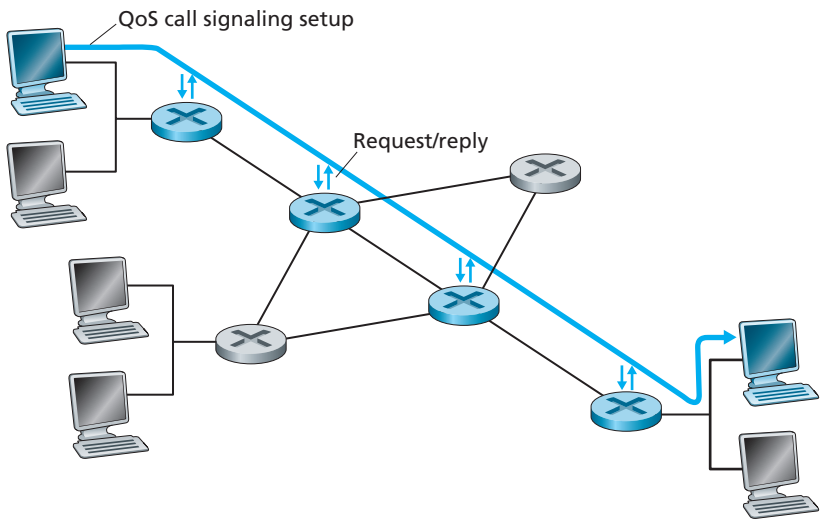

Figure 7.32 ♦ The call setup process

elements carry information for similar purposes as the Tspec and Rspec receptively; see [Black 1997] for details.

2. *Signaling for call setup.* A call's traffic descriptor and QoS request must be carried to the routers at which resources will be reserved for the call. In the Internet, the RSVP protocol [RFC 2210] is used for this purpose within the Intserv architecture. In ATM networks, the Q2931b [Black 1997] protocol carries this information among the ATM network's switches and end point.
3. *Per-element call admission.* Once a router receives the traffic specification and QoS, it must 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 calls. Recall that in Section 7.5.3, for example, we saw how the combination of a leaky-bucket-controlled source and WFQ can be used to determine the maximum queuing delay for that source. Per-element call admission is shown in Figure 7.33.

For additional discussion of call setup and admission, see [Breslau 2000; Roberts 2004].

7.6.3 Guaranteed QoS in the Internet: Intserv and RSVP

The integrated services (**Intserv**) architecture is a framework developed within the IETF to provide individualized QoS guarantees to individual application sessions in the Internet. Intserv's 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 7.5.2) with

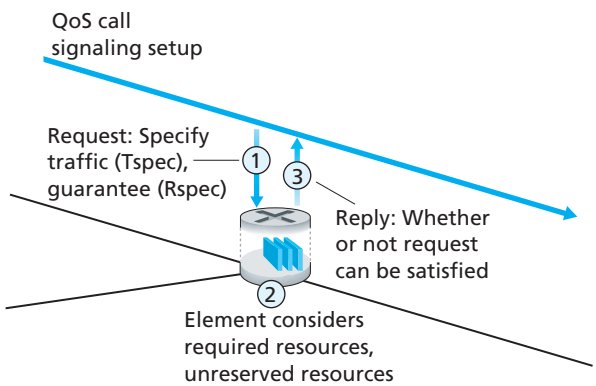

Figure 7.33 ♦ Per-element call behavior

PRINCIPLES IN PRACTICE

THE PRINCIPLE OF SOFT STATE

RSVP is used to install state (bandwidth reservations) in routers, and is known as a *soft-state* protocol. Broadly speaking, we associate the term *soft state* with signaling approaches in which installed state times out (and is removed) unless periodically refreshed by the receipt of a signaling message (typically from the entity that initially installed the state) indicating that the state should continue to remain installed. Since unrefreshed state will eventually time out, soft-state signaling requires neither explicit state removal nor a procedure to remove orphaned state should the state installer crash. Similarly, since state installation and refresh messages will be followed by subsequent periodic refresh messages, reliable signaling is not required. The term *soft state* was coined by Clark [Clark 1988], who described the notion of periodic state refresh messages being sent by an end system, and suggested that with such refresh messages, state could be lost in a crash and then automatically restored by subsequent refresh messages—all transparently to the end system and without invoking any explicit crash-recovery procedures:

> ". . . the state information would not be critical in maintaining the desired type of service associated with the flow. Instead, that type of service would be enforced by the end points, which would periodically send messages to ensure that the proper type of service was being associated with the flow. In this way, the state information associated with the flow could be lost in a crash without permanent disruption of the service features being used. I call this concept "soft state," and it may very well permit us to achieve our primary goals of survivability and flexibility. . ."

Roughly speaking, then, the essence of a soft-state approach is the use of best-effort periodic state installation/refresh by the state installer and state-removal-by-timeout at the state holder. Soft-state approaches have been taken in numerous protocols, including RSVP , PIM (Section 4.7) , SIP (Section 7.4.3), and IGMP (Section 4.7), and in forwarding tables in transparent bridges (Section 5.6).

Hard-state signaling takes the converse approach to soft state—installed state remains installed unless explicitly removed by the receipt of a state-teardown message from the state installer. Since the state remains installed unless explicitly removed, hard-state signaling requires a mechanism to remove an orphaned state that remains after the state installer has crashed or departed without removing the state. Similarly, since state installation and removal are performed only once (and without state refresh or state timeout), it is important for the state installer to know when the state has been installed or removed. Reliable (rather than best-effort) signaling protocols are thus typically associated with hard-state protocols. Roughly speaking, then, the essence of a hard-state approach is the reliable and explicit installation and removal of state information. Hard-state approaches have been taken in protocols such as ST-II [Partridge 1992, RFC 1190] and Q.2931 [ITU-T Q.2931 1994].

RSVP has provided for explicit (although optional) removal of reservations since its conception.

ACK-based reliable signaling was introduced as an extension to RSVP in [RFC 2961] and was also suggested in [Pan 1997]. RSVP has thus optionally adopted some elements of a hard-state signaling approach. For a discussion and comparison of soft-state versus hard-state protocols, see [Ji 2003].

parameters (r,b) and the requested service is characterized by the transmission rate, R, at which packets will be transmitted. In essence, a call 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 7.5.2 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. A second form of Intserv service guarantee has also been defined, known as controlled load service, which specifies that a call will receive "a quality of service closely approximating the QoS that same flow would receive from an unloaded network element" [RFC 2211].

The Resource ReSerVation Protocol (RSVP) [RFC 2205; Zhang 1993] is an Internet signaling protocol that could be used to perform the call setup signaling needed by Intserv. RSVP has also been used in conjunction with Diffserv to coordinate Diffserv functions across multiple networks, and has also been extended and used as a signaling protocol in other circumstances, perhaps most notably in the form of RSVP-TE [RFC 3209] for MPLS signaling, as discussed in Section 5.8.2.

In an Intserv context, 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 along the end-end path shown in Figure 7.32. The two principal characteristics of RSVP are:

- It provides **reservations for bandwidth in multicast trees**, with unicast being handled as a degenerate case of multicast. This is particularly important for multimedia applications such as streaming-broadcast-TV-over-IP, where many receivers may want to receive the same multimedia traffic being sent from a single source.
- It is **receiver-oriented**; that is, the receiver of a data flow initiates and maintains the resource reservation used for that flow. The innovative, receiver-centric view taken by RSVP puts receivers firmly in control of the traffic they receive, for example allowing different receivers to receive and view a multimedia multicast

at different resolutions. This contrasts with the sender-centric view of signaling adopted in ATM's Q2931b.

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 using the policing and scheduling mechanisms (leaky bucket, priority scheduling, weighted fair queuing) discussed in Section 7.5. For more information about RSVP, see [RFC 2205; Zhang 1993] and the additional online electronic material associated with this book.