CHAPTER 7

Multimedia Networking

People in all corners of the world are currently using the Internet to watch movies and television shows on demand. Internet movie and television distribution companies such as Netflix and Hulu in North America and Youku and Kankan in China have practically become household names. But people are not only watching Internet videos, they are using sites like YouTube to upload and distribute their own user-generated content, becoming Internet video producers as well as consumers. Moreover, network applications such as Skype, Google Talk, and QQ (enormously popular in China) allow people to not only make "telephone calls" over the Internet, but to also enhance those calls with video and multi-person conferencing. In fact, we can safely predict that by the end of the current decade almost all video distribution and voice conversations will take place end-to-end over the Internet, often to wireless devices connected to the Internet via 4G and WiFi access networks.

We begin this chapter with a taxonomy of multimedia applications in Section 7.1. We'll see that a multimedia application can be classified as either *streaming stored audio/video*, *conversational voice/video-over-IP*, or *streaming live audio/video*. We'll see that each of these classes of applications has its own unique service requirements that differ significantly from those of traditional elastic applications such as e-mail, Web browsing, and remote login. In Section 7.2, we'll examine video streaming in some detail. We'll explore many of the underlying principles behind video streaming, including client buffering, prefetching, and adapting video

quality to available bandwidth. We will also investigate Content Distribution Networks (CDNs), which are used extensively today by the leading video streaming systems. We then examine the YouTube, Netflix, and Kankan systems as case studies for streaming video. In Section 7.3, we investigate conversational voice and video, which, unlike elastic applications, are highly sensitive to end-to-end delay but can tolerate occasional loss of data. Here we'll examine how techniques such as adaptive playout, forward error correction, and error concealment can mitigate against network-induced packet loss and delay. We'll also examine Skype as a case study. In Section 7.4, we'll study RTP and SIP, two popular protocols for real-time conversational voice and video applications. In Section 7.5, we'll investigate mechanisms within the network that can be used to distinguish one class of traffic (e.g., delay-sensitive applications such as conversational voice) from another (e.g., elastic applications such as browsing Web pages), and provide differentiated service among multiple classes of traffic.

7.1 Multimedia Networking Applications

We define a multimedia network application as any network application that employs audio or video. In this section, we provide a taxonomy of multimedia applications. We'll see that each class of applications in the taxonomy has its own unique set of service requirements and design issues. But before diving into an in-depth discussion of Internet multimedia applications, it is useful to consider the intrinsic characteristics of the audio and video media themselves.

7.1.1 Properties of Video

Perhaps the most salient characteristic of video is its **high bit rate**. Video distributed over the Internet typically ranges from 100 kbps for low-quality video conferencing to over 3 Mbps for streaming high-definition movies. To get a sense of how video bandwidth demands compare with those of other Internet applications, let's briefly consider three different users, each using a different Internet application. Our first user, Frank, is going quickly through photos posted on his friends' Facebook pages. Let's assume that Frank is looking at a new photo every 10 seconds, and that photos are on average 200 Kbytes in size. (As usual, throughout this discussion we make the simplifying assumption that 1 Kbyte = 8,000 bits.) Our second user, Martha, is streaming music from the Internet ("the cloud") to her smartphone. Let's assume Martha is listening to many MP3 songs, one after the other, each encoded at a rate of 128 kbps. Our third user, Victor, is watching a video that has been encoded at 2 Mbps. Finally, let's suppose that the session length for all three users is 4,000 seconds (approximately 67 minutes). Table 7.1 compares the bit rates and the total bytes transferred for these three users. We see that video streaming consumes by far

	Bit rate	Bytes transferred in 67 min
Facebook Frank	160 kbps	80 Mbytes
Martha Music	128 kbps	64 Mbytes
Victor Video	2 Mbps	1 Gbyte

Table 7.1 ♦ Comparison of bit-rate requirements of three Internet applications

the most bandwidth, having a bit rate of more than ten times greater than that of the Facebook and music-streaming applications. Therefore, when designing networked video applications, the first thing we must keep in mind is the high bit-rate requirements of video. Given the popularity of video and its high bit rate, it is perhaps not surprising that Cisco predicts [Cisco 2011] that streaming and stored video will be approximately 90 percent of global consumer Internet traffic by 2015.

Another important characteristic of video is that it can be compressed, thereby trading off video quality with bit rate. 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 by **video compression**. *Spatial redundancy* is the redundancy within a given image. Intuitively, an image that consists of mostly white space has a high degree of redundancy and can be efficiently compressed without significantly sacrificing image quality. *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 instead more efficient simply to indicate during encoding that the subsequent image is exactly the same. Today's off-the-shelf compression algorithms can compress a video to essentially any bit rate desired. Of course, the higher the bit rate, the better the image quality and the better the overall user viewing experience.

We can also use compression to create **multiple versions** of the same video, each at a different quality level. For example, we can use compression to create, say, three versions of the same video, at rates of 300 kbps, 1 Mbps, and 3 Mbps. Users can then decide which version they want to watch as a function of their current available bandwidth. Users with high-speed Internet connections might choose the 3 Mbps version; users watching the video over 3G with a smartphone might choose the 300 kbps version. Similarly, the video in a video conference application can be compressed "on-the-fly" to provide the best video quality given the available end-to-end bandwidth between conversing users.

7.1.2 Properties of Audio

Digital audio (including digitized speech and music) has significantly lower bandwidth requirements than video. Digital audio, however, has its own unique properties that must be considered when designing multimedia network applications. To understand these properties, let's first consider how analog audio (which humans and musical instruments generate) is converted to a digital signal:

- The analog audio signal is 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 such 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 audio sample—is represented by one byte. The bit representations of all the samples are then 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. For playback through audio speakers, the digital signal can then be converted back—that is, decoded—to an analog signal. However, the decoded analog signal is only an approximation of the original signal, and the sound quality may be noticeably degraded (for example, high-frequency sounds may be missing in the decoded signal). By increasing the sampling rate and the number of quantization values, the decoded signal can better approximate the original analog signal. Thus (as with video), there is a trade-off between the quality of the decoded signal and the bit-rate and storage 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, resulting in 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.

PCM-encoded speech and music, however, are rarely used in the Internet. Instead, as with video, compression techniques are used to reduce the bit rates of the stream. Human speech can be compressed to less than 10 kbps and still be intelligible. A popular compression technique for near CD-quality stereo music is **MPEG 1 layer 3**, more commonly known as **MP3**. MP3 encoders can compress to many different rates; 128 kbps is the most common encoding rate and produces very little sound degradation. A related standard is **Advanced Audio Coding (AAC)**, which has been popularized by Apple. As with video, multiple versions of a prerecorded audio stream can be created, each at a different bit rate.

Although audio bit rates are generally much less than those of video, users are generally much more sensitive to audio glitches than video glitches. Consider, for example, a video conference taking place over the Internet. If, from time to time, the video signal is lost for a few seconds, the video conference can likely proceed without too much user frustration. If, however, the audio signal is frequently lost, the users may have to terminate the session.

7.1.3 Types of Multimedia Network Applications

The Internet supports a large variety of useful and entertaining multimedia applications. In this subsection, we classify multimedia applications into three broad categories: *(i) streaming stored audio/video*, *(ii) conversational voice/video-over-IP*, and *(iii) streaming live audio/video*. As we will soon see, each of these application categories has its own set of service requirements and design issues.

Streaming Stored Audio and Video

To keep the discussion concrete, we focus here on streaming stored video, which typically combines video and audio components. Streaming stored audio (such as streaming music) is very similar to streaming stored video, although the bit rates are typically much lower.

In this class of applications, the underlying medium is prerecorded video, such as a movie, a television show, a prerecorded sporting event, or a prerecorded user-generated video (such as those commonly seen on YouTube). These prerecorded videos are placed on servers, and users send requests to the servers to view the videos *on demand*. Many Internet companies today provide streaming video, including YouTube (Google), Netflix, and Hulu. By some estimates, streaming stored video makes up over 50 percent of the downstream traffic in the Internet access networks today [Cisco 2011]. Streaming stored video has three key distinguishing features.

- *Streaming.* In a streaming stored video application, the client typically begins video playout within a few seconds after it begins receiving the video from the server. This means that the client will be playing out from one location in the video while at the same time receiving later parts of the video from the server. This technique, known as **streaming**, avoids having to download the entire video file (and incurring a potentially long delay) before playout begins.
- *Interactivity.* Because the media is prerecorded, the user may pause, reposition forward, reposition backward, fast-forward, and so on through the video content. The time from when the user makes such a request until the action manifests itself at the client should be less than a few seconds for acceptable responsiveness.
- *Continuous playout.* Once playout of the video begins, it should proceed according to the original timing of the recording. Therefore, data must be received from the server in time for its playout at the client; otherwise, users

experience video frame freezing (when the client waits for the delayed frames) or frame skipping (when the client skips over delayed frames).

By far, the most important performance measure for streaming video is average throughput. In order to provide continuous playout, the network must provide an average throughput to the streaming application that is at least as large the bit rate of the video itself. As we will see in Section 7.2, by using buffering and prefetching, it is possible to provide continuous playout even when the throughput fluctuates, as long as the average throughput (averaged over 5–10 seconds) remains above the video rate [Wang 2008].

For many streaming video applications, prerecorded video is stored on, and streamed from, a CDN rather than from a single data center. There are also many P2P video streaming applications for which the video is stored on users' hosts (peers), with different chunks of video arriving from different peers that may spread around the globe. Given the prominence of Internet video streaming, we will explore video streaming in some depth in Section 7.2, paying particular attention to client buffering, prefetching, adapting quality to bandwidth availability, and CDN distribution.

Conversational Voice- and Video-over-IP

Real-time conversational voice over the Internet is often referred to as **Internet telephony**, since, from the user's perspective, it is similar to the traditional circuit-switched telephone service. It is also commonly called **Voice-over-IP (VoIP)**. Conversational video is similar, except that it includes the video of the participants as well as their voices. Most of today's voice and video conversational systems allow users to create conferences with three or more participants. Conversational voice and video are widely used in the Internet today, with the Internet companies Skype, QQ, and Google Talk boasting hundreds of millions of daily users.

In our discussion of application service requirements in Chapter 2 (Figure 2.4), we identified a number of axes along which application requirements can be classified. Two of these axes—timing considerations and tolerance of data loss—are particularly important for conversational voice and video applications. Timing considerations are important because audio and video conversational applications are highly **delay-sensitive**. For a conversation with two or more interacting speakers, the delay from when a user speaks or moves until the action is manifested at the other end should be less than a few hundred milliseconds. For voice, delays smaller than 150 milliseconds are not perceived by a human listener, delays between 150 and 400 milliseconds can be acceptable, and delays exceeding 400 milliseconds can result in frustrating, if not completely unintelligible, voice conversations.

On the other hand, conversational multimedia applications are **loss-tolerant**—occasional loss only causes occasional glitches in audio/video playback, and these losses can often be partially or fully concealed. These delay-sensitive but loss-tolerant

characteristics are clearly different from those of elastic data applications such as Web browsing, e-mail, social networks, and remote login. For elastic applications, long delays are annoying but not particularly harmful; the completeness and integrity of the transferred data, however, are of paramount importance. We will explore conversational voice and video in more depth in Section 7.3, paying particular attention to how adaptive playout, forward error correction, and error concealment can mitigate against network-induced packet loss and delay.

Streaming Live Audio and Video

This third class of applications is similar to traditional broadcast radio and television, except that transmission takes place over the Internet. These applications allow a user to receive a *live* radio or television transmission—such as a live sporting event or an ongoing news event—transmitted from any corner of the world. Today, thousands of radio and television stations around the world are broadcasting content over the Internet.

Live, broadcast-like applications often have many users who receive the same audio/video program at the same time. Although the distribution of live audio/video to many receivers can be efficiently accomplished using the IP multicasting techniques described in Section 4.7, multicast distribution is more often accomplished today via application-layer multicast (using P2P networks or CDNs) or through multiple separate unicast streams. As with streaming stored multimedia, the network must provide each live multimedia flow with an average throughput that is larger than the video consumption rate. Because the event is live, delay can also be an issue, although the timing constraints are much less stringent than those for conversational voice. Delays of up to ten seconds or so from when the user chooses to view a live transmission to when playout begins can be tolerated. We will not cover streaming live media in this book because many of the techniques used for streaming live media—initial buffering delay, adaptive bandwidth use, and CDN distribution—are similar to those for streaming stored media.

7.2 Streaming Stored Video

For streaming video applications, prerecorded videos are placed on servers, and users send requests to these servers to view the videos on demand. The user may watch the video from beginning to end without interruption, may stop watching the video well before it ends, or interact with the video by pausing or repositioning to a future or past scene. Streaming video systems can be classified into three categories: **UDP streaming**, **HTTP streaming**, and **adaptive HTTP streaming**. Although all three types of systems are used in practice, the majority of today's systems employ HTTP streaming and adaptive HTTP streaming.

A common characteristic of all three forms of video streaming is the extensive use of client-side application buffering to mitigate the effects of varying end-to-end delays and varying amounts of available bandwidth between server and client. For streaming video (both stored and live), users generally can tolerate a small several-second initial delay between when the client requests a video and when video playout begins at the client. Consequently, when the video starts to arrive at the client, the client need not immediately begin playout, but can instead build up a reserve of video in an application buffer. Once the client has built up a reserve of several seconds of buffered-but-not-yet-played video, the client can then begin video playout. There are two important advantages provided by such **client buffering**. First, client-side buffering can absorb variations in server-to-client delay. If a particular piece of video data is delayed, as long as it arrives before the reserve of received-but-not-yet-played video is exhausted, this long delay will not be noticed. Second, if the server-to-client bandwidth briefly drops below the video consumption rate, a user can continue to enjoy continuous playback, again as long as the client application buffer does not become completely drained.

Figure 7.1 illustrates client-side buffering. In this simple example, suppose that video is encoded at a fixed bit rate, and thus each video block contains video frames that are to be played out over the same fixed amount of time, $\triangle$. The server transmits the first video block at t_0, the second block at $t_0 + \triangle$, the third block at $t_0 + 2\triangle$, and so on. Once the client begins playout, each block should be played out $\triangle$ time units after the previous block in order to reproduce the timing of the original recorded video. Because of the variable end-to-end network delays, different video blocks experience different delays. The first video block arrives at the client at t_1 and the second block arrives at t_2. The network delay for the ith block is the horizontal distance between the time the block was transmitted by the server and the

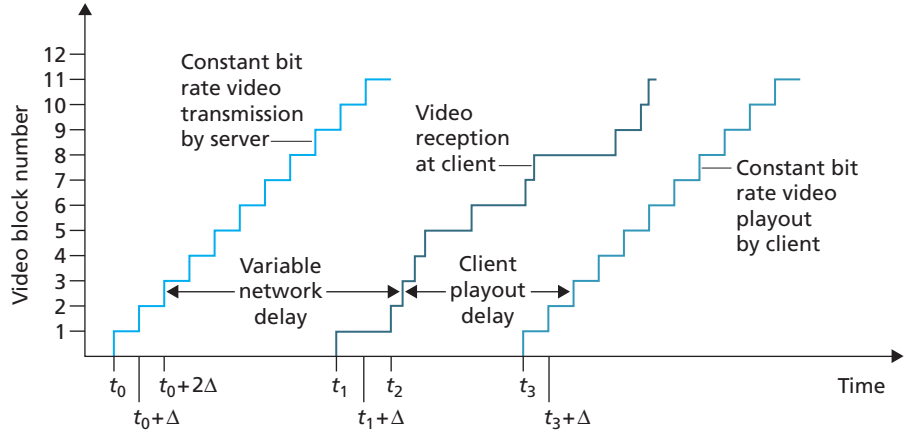


Figure 7.1 ♦ Client playout delay in video streaming

time it is received at the client; note that the network delay varies from one video block to another. In this example, if the client were to begin playout as soon as the first block arrived at t_1, then the second block would not have arrived in time to be played out at out at $t_1 + \triangle$. In this case, video playout would either have to stall (waiting for block 1 to arrive) or block 1 could be skipped—both resulting in undesirable playout impairments. Instead, if the client were to delay the start of playout until t_3, when blocks 1 through 6 have all arrived, periodic playout can proceed with *all* blocks having been received before their playout time.

7.2.1 UDP Streaming

We only briefly discuss UDP streaming here, referring the reader to more in-depth discussions of the protocols behind these systems where appropriate. With UDP streaming, the server transmits video at a rate that matches the client's video consumption rate by clocking out the video chunks over UDP at a steady rate. For example, if the video consumption rate is 2 Mbps and each UDP packet carries 8,000 bits of video, then the server would transmit one UDP packet into its socket every (8000 bits)/(2 Mbps) = 4 msec. As we learned in Chapter 3, because UDP does not employ a congestion-control mechanism, the server can push packets into the network at the consumption rate of the video without the rate-control restrictions of TCP. UDP streaming typically uses a small client-side buffer, big enough to hold less than a second of video.

Before passing the video chunks to UDP, the server will encapsulate the video chunks within transport packets specially designed for transporting audio and video, using the Real-Time Transport Protocol (RTP) [RFC 3550] or a similar (possibly proprietary) scheme. We delay our coverage of RTP until Section 7.3, where we discuss RTP in the context of conversational voice and video systems.

Another distinguishing property of UDP streaming is that in addition to the server-to-client video stream, the client and server also maintain, in parallel, a separate control connection over which the client sends commands regarding session state changes (such as pause, resume, reposition, and so on). This control connection is in many ways analogous to the FTP control connection we studied in Chapter 2. The Real-Time Streaming Protocol (RTSP) [RFC 2326], explained in some detail in the companion Web site for this textbook, is a popular open protocol for such a control connection.

Although UDP streaming has been employed in many open-source systems and proprietary products, it suffers from three significant drawbacks. First, due to the unpredictable and varying amount of available bandwidth between server and client, constant-rate UDP streaming can fail to provide continuous playout. For example, consider the scenario where the video consumption rate is 1 Mbps and the server-to-client available bandwidth is usually more than 1 Mbps, but every few minutes the available bandwidth drops below 1 Mbps for several seconds. In such a scenario, a UDP streaming system that transmits video at a constant rate of 1 Mbps over RTP/UDP would likely provide a poor user experience, with freezing or skipped frames soon after the available bandwidth falls below 1 Mbps. The second drawback

of UDP streaming is that it requires a media control server, such as an RTSP server, to process client-to-server interactivity requests and to track client state (e.g., the client's playout point in the video, whether the video is being paused or played, and so on) for *each* ongoing client session. This increases the overall cost and complexity of deploying a large-scale video-on-demand system. The third drawback is that many firewalls are configured to block UDP traffic, preventing the users behind these firewalls from receiving UDP video.

7.2.2 HTTP Streaming

In HTTP streaming, the video is simply stored in an HTTP server as an ordinary file with a specific URL. When a user wants to see the video, the client establishes a TCP connection with the server and issues an HTTP GET request for that URL. The server then sends the video file, within an HTTP response message, as quickly as possible, that is, as quickly as TCP congestion control and flow control will allow. On the client side, the bytes are collected in a client application buffer. Once the number of bytes in this buffer exceeds a predetermined threshold, the client application begins playback—specifically, it periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen.

We learned in Chapter 3 that when transferring a file over TCP, the server-to-client transmission rate can vary significantly due to TCP's congestion control mechanism. In particular, it is not uncommon for the transmission rate to vary in a "saw-tooth" manner (for example, Figure 3.53) associated with TCP congestion control. Furthermore, packets can also be significantly delayed due to TCP's retransmission mechanism. Because of these characteristics of TCP, the conventional wisdom in the 1990s was that video streaming would never work well over TCP. Over time, however, designers of streaming video systems learned that TCP's congestion control and reliable-data transfer mechanisms do not necessarily preclude continuous playout when client buffering and prefetching (discussed in the next section) are used.

The use of HTTP over TCP also allows the video to traverse firewalls and NATs more easily (which are often configured to block most UDP traffic but to allow most HTTP traffic). Streaming over HTTP also obviates the need for a media control server, such as an RTSP server, reducing the cost of a large-scale deployment over the Internet. Due to all of these advantages, most video streaming applications today—including YouTube and Netflix—use HTTP streaming (over TCP) as its underlying streaming protocol.

Prefetching Video

We just learned, client-side buffering can be used to mitigate the effects of varying end-to-end delays and varying available bandwidth. In our earlier example in Figure 7.1, the server transmits video at the rate at which the video is to be played

out. However, for streaming *stored* video, the client can attempt to download the video at a rate *higher* than the consumption rate, thereby **prefetching** video frames that are to be consumed in the future. This prefetched video is naturally stored in the client application buffer. Such prefetching occurs naturally with TCP streaming, since TCP's congestion avoidance mechanism will attempt to use all of the available bandwidth between server and client.

To gain some insight into prefetching, let's take a look at a simple example. Suppose the video consumption rate is 1 Mbps but the network is capable of delivering the video from server to client at a constant rate of 1.5 Mbps. Then the client will not only be able to play out the video with a very small playout delay, but will also be able to increase the amount of buffered video data by 500 Kbits every second. In this manner, if in the future the client receives data at a rate of less than 1 Mbps for a brief period of time, the client will be able to continue to provide continuous playback due to the reserve in its buffer. [Wang 2008] shows that when the average TCP throughput is roughly twice the media bit rate, streaming over TCP results in minimal starvation and low buffering delays.

Client Application Buffer and TCP Buffers

Figure 7.2 illustrates the interaction between client and server for HTTP streaming. At the server side, the portion of the video file in white has already been sent into the server's socket, while the darkened portion is what remains to be sent. After "passing through the socket door," the bytes are placed in the TCP send buffer before being transmitted into the Internet, as described in Chapter 3. In Figure 7.2,

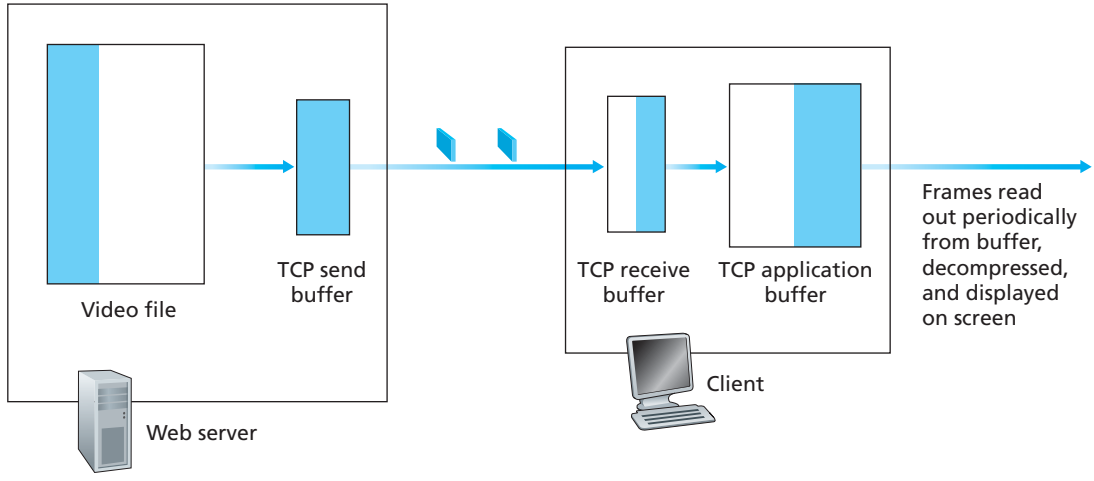


Figure 7.2 ♦ Streaming stored video over HTTP/TCP

because the TCP send buffer is shown to be full, the server is momentarily prevented from sending more bytes from the video file into the socket. On the client side, the client application (media player) reads bytes from the TCP receive buffer (through its client socket) and places the bytes into the client application buffer. At the same time, the client application periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen. Note that if the client application buffer is larger than the video file, then the whole process of moving bytes from the server's storage to the client's application buffer is equivalent to an ordinary file download over HTTP—the client simply pulls the video off the server as fast as TCP will allow!

Consider now what happens when the user pauses the video during the streaming process. During the pause period, bits are not removed from the client application buffer, even though bits continue to enter the buffer from the server. If the client application buffer is finite, it may eventually become full, which will cause "back pressure" all the way back to the server. Specifically, once the client application buffer becomes full, bytes can no longer be removed from the client TCP receive buffer, so it too becomes full. Once the client receive TCP buffer becomes full, bytes can no longer be removed from the client TCP send buffer, so it also becomes full. Once the TCP send buffer becomes full, the server cannot send any more bytes into the socket. Thus, if the user pauses the video, the server may be forced to stop transmitting, in which case the server will be blocked until the user resumes the video.

In fact, even during regular playback (that is, without pausing), if the client application buffer becomes full, back pressure will cause the TCP buffers to become full, which will force the server to reduce its rate. To determine the resulting rate, note that when the client application removes f bits, it creates room for f bits in the client application buffer, which in turn allows the server to send f additional bits. Thus, the server send rate can be no higher than the video consumption rate at the client. Therefore, *a full client application buffer indirectly imposes a limit on the rate that video can be sent from server to client when streaming over HTTP.*

Analysis of Video Streaming

Some simple modeling will provide more insight into initial playout delay and freezing due to application buffer depletion. As shown in Figure 7.3, let B denote the size (in bits) of the client's application buffer, and let Q denote the number of bits that must be buffered before the client application begins playout. (Of course, $Q < B$.) Let r denote the video consumption rate—the rate at which the client draws bits out of the client application buffer during playback. So, for example, if the video's frame rate is 30 frames/sec, and each (compressed) frame is 100,000 bits, then $r = 3$ Mbps. To see the forest through the trees, we'll ignore TCP's send and receive buffers.

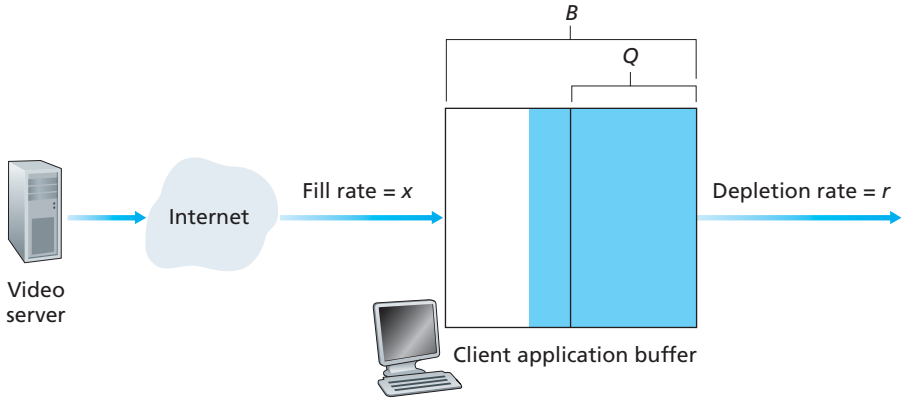


Figure 7.3 ♦ Analysis of client-side buffering for video streaming

Let's assume that the server sends bits at a constant rate x whenever the client buffer is not full. (This is a gross simplification, since TCP's send rate varies due to congestion control; we'll examine more realistic time-dependent rates $x(t)$ in the problems at the end of this chapter.) Suppose at time $t = 0$, the application buffer is empty and video begins arriving to the client application buffer. We now ask at what time $t = t_p$ does playout begin? And while we are at it, at what time $t = t_f$ does the client application buffer become full?

First, let's determine t_p, the time when Q bits have entered the application buffer and playout begins. Recall that bits arrive to the client application buffer at rate x and *no* bits are removed from this buffer before playout begins. Thus, the amount of time required to build up Q bits (the initial buffering delay) is $t_p = Q/x$.

Now let's determine t_f, the point in time when the client application buffer becomes full. We first observe that if $x < r$ (that is, if the server send rate is less than the video consumption rate), then the client buffer will never become full! Indeed, starting at time t_p, the buffer will be depleted at rate r and will only be filled at rate $x < r$. Eventually the client buffer will empty out entirely, at which time the video will freeze on the screen while the client buffer waits another t_p seconds to build up Q bits of video. *Thus, when the available rate in the network is less than the video rate, playout will alternate between periods of continuous playout and periods of freezing.* In a homework problem, you will be asked to determine the length of each continuous playout and freezing period as a function of Q, r, and x. Now let's determine t_f for when $x > r$. In this case, starting at time t_p, the buffer increases from Q to B at rate $x - r$ since bits are being depleted at rate r but are arriving at rate x, as shown in Figure 7.3. Given these hints, you will be asked in a homework problem to determine t_f, the time the client buffer becomes full. Note that *when the available rate in the network is more than the video rate, after the initial buffering delay, the user will enjoy continuous playout until the video ends.*

Early Termination and Repositioning the Video

HTTP streaming systems often make use of the **HTTP byte-range header** in the HTTP GET request message, which specifies the specific range of bytes the client currently wants to retrieve from the desired video. This is particularly useful when the user wants to reposition (that is, jump) to a future point in time in the video. When the user repositions to a new position, the client sends a new HTTP request, indicating with the byte-range header from which byte in the file should the server send data. When the server receives the new HTTP request, it can forget about any earlier request and instead send bytes beginning with the byte indicated in the byte-range request.

While we are on the subject of repositioning, we briefly mention that when a user repositions to a future point in the video or terminates the video early, some prefetched-but-not-yet-viewed data transmitted by the server will go unwatched—a waste of network bandwidth and server resources. For example, suppose that the client buffer is full with B bits at some time t_0 into the video, and at this time the user repositions to some instant $t > t_0 + B/r$ into the video, and then watches the video to completion from that point on. In this case, all B bits in the buffer will be unwatched and the bandwidth and server resources that were used to transmit those B bits have been completely wasted. There is significant wasted bandwidth in the Internet due to early termination, which can be quite costly, particularly for wireless links [Ihm 2011]. For this reason, many streaming systems use only a moderate-size client application buffer, or will limit the amount of prefetched video using the byte-range header in HTTP requests [Rao 2011].

Repositioning and early termination are analogous to cooking a large meal, eating only a portion of it, and throwing the rest away, thereby wasting food. So the next time your parents criticize you for wasting food by not eating all your dinner, you can quickly retort by saying they are wasting bandwidth and server resources when they reposition while watching movies over the Internet! But, of course, two wrongs do not make a right—both food and bandwidth are not to be wasted!

7.2.3 Adaptive Streaming and DASH

Although HTTP streaming, as described in the previous subsection, has been extensively deployed in practice (for example, by YouTube since its inception), it has a major shortcoming: All clients receive the same encoding of the video, despite the large variations in the amount of bandwidth available to a client, both across different clients and also over time for the same client. This has led to the development of a new type of HTTP-based streaming, often referred to as **Dynamic Adaptive Streaming over HTTP (DASH)**. In DASH, the video is encoded into several different versions, with each version having a different bit rate and, correspondingly, a different quality level. The client dynamically requests chunks of video segments of a few seconds in length from the different versions. When the amount of available

bandwidth is high, the client naturally selects chunks from a high-rate version; and when the available bandwidth is low, it naturally selects from a low-rate version. The client selects different chunks one at a time with HTTP GET request messages [Akhshabi 2011].

On one hand, DASH allows clients with different Internet access rates to stream in video at different encoding rates. Clients with low-speed 3G connections can receive a low bit-rate (and low-quality) version, and clients with fiber connections can receive a high-quality version. On the other hand, DASH allows a client to adapt to the available bandwidth if the end-to-end bandwidth changes during the session. This feature is particularly important for mobile users, who typically see their bandwidth availability fluctuate as they move with respect to the base stations. Comcast, for example, has deployed an adaptive streaming system in which each video source file is encoded into 8 to 10 different MPEG-4 formats, allowing the highest quality video format to be streamed to the client, with adaptation being performed in response to changing network and device conditions.

With DASH, each video version is stored in the HTTP server, each with a different URL. The HTTP server also has a **manifest file**, which provides a URL for each version along with its bit rate. The client first requests the manifest file and learns about the various versions. The client then selects one chunk at a time by specifying a URL and a byte range in an HTTP GET request message for each chunk. While downloading chunks, the client also measures the received bandwidth and runs a *rate determination algorithm* to select the chunk to request next. Naturally, if the client has a lot of video buffered and if the measured receive bandwidth is high, it will choose a chunk from a high-rate version. And naturally if the client has little video buffered and the measured received bandwidth is low, it will choose a chunk from a low-rate version. DASH therefore allows the client to freely switch among different quality levels. Since a sudden drop in bit rate by changing versions may result in noticeable visual quality degradation, the bit-rate reduction may be achieved using multiple intermediate versions to smoothly transition to a rate where the client's consumption rate drops below its available receive bandwidth. When the network conditions improve, the client can then later choose chunks from higher bit-rate versions.

By dynamically monitoring the available bandwidth and client buffer level, and adjusting the transmission rate with version switching, DASH can often achieve continuous playout at the best possible quality level without frame freezing or skipping. Furthermore, since the client (rather than the server) maintains the intelligence to determine which chunk to send next, the scheme also improves server-side scalability. Another benefit of this approach is that the client can use the HTTP byte-range request to precisely control the amount of prefetched video that it buffers locally.

We conclude our brief discussion of DASH by mentioning that for many implementations, the server not only stores many versions of the video but also separately stores many versions of the audio. Each audio version has its own quality level and bit rate and has its own URL. In these implementations, the client dynamically selects both video and audio chunks, and locally synchronizes audio and video playout.

7.2.4 Content Distribution Networks

Today, many Internet video companies are distributing on-demand multi-Mbps streams to millions of users on a daily basis. YouTube, for example, with a library of hundreds of millions of videos, distributes hundreds of millions of video streams to users around the world every day [Ding 2011]. Streaming all this traffic to locations all over the world while providing continuous playout and high interactivity is clearly a challenging task.

For an Internet video company, perhaps the most straightforward approach to providing streaming video service is to build a single massive data center, store all of its videos in the data center, and stream the videos directly from the data center to clients worldwide. But there are three major problems with this approach. First, if the client is far from the data center, server-to-client packets will cross many communication links and likely pass through many ISPs, with some of the ISPs possibly located on different continents. If one of these links provides a throughput that is less than the video consumption rate, the end-to-end throughput will also be below the consumption rate, resulting in annoying freezing delays for the user. (Recall from Chapter 1 that the end-to-end throughput of a stream is governed by the throughput in the bottleneck link.) The likelihood of this happening increases as the number of links in the end-to-end path increases. A second drawback is that a popular video will likely be sent many times over the same communication links. Not only does this waste network bandwidth, but the Internet video company itself will be paying its provider ISP (connected to the data center) for sending the *same* bytes into the Internet over and over again. A third problem with this solution is that a single data center represents a single point of failure—if the data center or its links to the Internet goes down, it would not be able to distribute *any* video streams.

In order to meet the challenge of distributing massive amounts of video data to users distributed around the world, almost all major video-streaming companies make use of **Content Distribution Networks (CDNs)**. A CDN manages servers in multiple geographically distributed locations, stores copies of the videos (and other types of Web content, including documents, images, and audio) in its servers, and attempts to direct each user request to a CDN location that will provide the best user experience. The CDN may be a **private CDN**, that is, owned by the content provider itself; for example, Google's CDN distributes YouTube videos and other types of content. The CDN may alternatively be a **third-party CDN** that distributes content on behalf of multiple content providers; Akamai's CDN, for example, is a third-party CDN that distributes Netflix and Hulu content, among others. A very readable overview of modern CDNs is [Leighton 2009].

CDNs typically adopt one of two different server placement philosophies [Huang 2008]:

- **Enter Deep.** One philosophy, pioneered by Akamai, is to *enter deep* into the access networks of Internet Service Providers, by deploying server clusters in access ISPs all over the world. (Access networks are described in Section 1.3.)

CASE STUDY

GOOGLE'S NETWORK INFRASTRUCTURE

To support its vast array of cloud services—including search, gmail, calendar, YouTube video, maps, documents, and social networks—Google has deployed an extensive private network and CDN infrastructure. Google's CDN infrastructure has three tiers of server clusters:

- Eight "mega data centers," with six located in the United States and two located in Europe [Google Locations 2012], with each data center having on the order of 100,000 servers. These mega data centers are responsible for serving dynamic (and often personalized) content, including search results and gmail messages.
- About 30 "bring-home" clusters (see discussion in 7.2.4), with each cluster consisting on the order of 100–500 servers [Adhikari 2011a]. The cluster locations are distributed around the world, with each location typically near multiple tier-1 ISP PoPs. These clusters are responsible for serving static content, including YouTube videos [Adhikari 2011a].
- Many hundreds of "enter-deep" clusters (see discussion in 7.2.4), with each cluster located within an access ISP. Here a cluster typically consists of tens of servers within a single rack. These enter-deep servers perform TCP splitting (see Section 3.7) and serve static content [Chen 2011], including the static portions of Web pages that embody search results.

All of these data centers and cluster locations are networked together with Google's own private network, as part of one enormous AS (AS 15169). When a user makes a search query, often the query is first sent over the local ISP to a nearby enter-deep cache, from where the static content is retrieved; while providing the static content to the client, the nearby cache also forwards the query over Google's private network to one of the mega data centers, from where the personalized search results are retrieved. For a YouTube video, the video itself may come from one of the bring-home caches, whereas portions of the Web page surrounding the video may come from the nearby enter-deep cache, and the advertisements surrounding the video come from the data centers. In summary, except for the local ISPs, the Google cloud services are largely provided by a network infrastructure that is independent of the public Internet.

Akamai takes this approach with clusters in approximately 1,700 locations. The goal is to get close to end users, thereby improving user-perceived delay and throughput by decreasing the number of links and routers between the end user and the CDN cluster from which it receives content. Because of this highly distributed design, the task of maintaining and managing the clusters becomes challenging.

- **Bring Home.** A second design philosophy, taken by Limelight and many other CDN companies, is to *bring the ISPs home* by building large clusters at a smaller number (for example, tens) of key locations and connecting these clusters using a private high-speed network. Instead of getting inside the access ISPs, these CDNs typically place each cluster at a location that is simultaneously near the PoPs (see Section 1.3) of many tier-1 ISPs, for example, within a few miles of both AT&T and Verizon PoPs in a major city. Compared with the enter-deep design philosophy, the bring-home design typically results in lower maintenance and management overhead, possibly at the expense of higher delay and lower throughput to end users.

Once its clusters are in place, the CDN replicates content across its clusters. The CDN may not want to place a copy of every video in each cluster, since some videos are rarely viewed or are only popular in some countries. In fact, many CDNs do not push videos to their clusters but instead use a simple pull strategy: If a client requests a video from a cluster that is not storing the video, then the cluster retrieves the video (from a central repository or from another cluster) and stores a copy locally while streaming the video to the client at the same time. Similar to Internet caches (see Chapter 2), when a cluster's storage becomes full, it removes videos that are not frequently requested.

CDN Operation

Having identified the two major approaches toward deploying a CDN, let's now dive down into the nuts and bolts of how a CDN operates. When a browser in a user's host is instructed to retrieve a specific video (identified by a URL), the CDN must intercept the request so that it can (1) determine a suitable CDN server cluster for that client at that time, and (2) redirect the client's request to a server in that cluster. We'll shortly discuss how a CDN can determine a suitable cluster. But first let's examine the mechanics behind intercepting and redirecting a request.

Most CDNs take advantage of DNS to intercept and redirect requests; an interesting discussion of such a use of the DNS is [Vixie 2009]. Let's consider a simple example to illustrate how DNS is typically involved. Suppose a content provider, NetCinema, employs the third-party CDN company, KingCDN, to distribute its videos to its customers. On the NetCinema Web pages, each of its videos is assigned a URL that includes the string "video" and a unique identifier for the video itself; for example, Transformers 7 might be assigned http://video.netcinema.com/6Y7B23V. Six steps then occur, as shown in Figure 7.4:

1. The user visits the Web page at NetCinema.
2. When the user clicks on the link http://video.netcinema.com/6Y7B23V, the user's host sends a DNS query for video.netcinema.com.

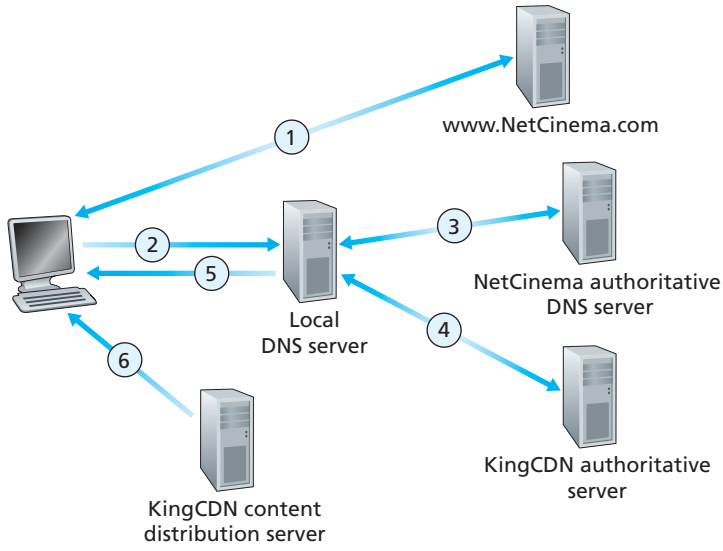


Figure 7.4 ♦ DNS redirects a user's request to a CDN server

3. The user's Local DNS Server (LDNS) relays the DNS query to an authoritative DNS server for NetCinema, which observes the string "video" in the hostname video.netcinema.com. To "hand over" the DNS query to KingCDN, instead of returning an IP address, the NetCinema authoritative DNS server returns to the LDNS a hostname in the KingCDN's domain, for example, a1105.kingcdn.com.
4. From this point on, the DNS query enters into KingCDN's private DNS infrastructure. The user's LDNS then sends a second query, now for a1105.kingcdn.com, and KingCDN's DNS system eventually returns the IP addresses of a KingCDN content server to the LDNS. It is thus here, within the KingCDN's DNS system, that the CDN server from which the client will receive its content is specified.
5. The LDNS forwards the IP address of the content-serving CDN node to the user's host.
6. Once the client receives the IP address for a KingCDN content server, it establishes a direct TCP connection with the server at that IP address and issues an HTTP GET request for the video. If DASH is used, the server will first send to the client a manifest file with a list of URLs, one for each version of the video, and the client will dynamically select chunks from the different versions.

Cluster Selection Strategies

At the core of any CDN deployment is a **cluster selection strategy**, that is, a mechanism for dynamically directing clients to a server cluster or a data center within the CDN. As we just saw, the CDN learns the IP address of the client's LDNS server via the client's DNS lookup. After learning this IP address, the CDN needs to select an appropriate cluster based on this IP address. CDNs generally employ proprietary cluster selection strategies. We now briefly survey a number of natural approaches, each of which has its own advantages and disadvantages.

One simple strategy is to assign the client to the cluster that is **geographically closest**. Using commercial geo-location databases (such as Quova [Quova 2012] and Max-Mind [MaxMind 2012]), each LDNS IP address is mapped to a geographic location. When a DNS request is received from a particular LDNS, the CDN chooses the geographically closest cluster, that is, the cluster that is the fewest kilometers from the LDNS "as the bird flies." Such a solution can work reasonably well for a large fraction of the clients [Agarwal 2009]. However, for some clients, the solution may perform poorly, since the geographically closest cluster may not be the closest cluster along the network path. Furthermore, a problem inherent with all DNS-based approaches is that some end-users are configured to use remotely located LDNSs [Shaikh 2001; Mao 2002], in which case the LDNS location may be far from the client's location. Moreover, this simple strategy ignores the variation in delay and available bandwidth over time of Internet paths, always assigning the same cluster to a particular client.

In order to determine the best cluster for a client based on the *current* traffic conditions, CDNs can instead perform periodic **real-time measurements** of delay and loss performance between their clusters and clients. For instance, a CDN can have each of its clusters periodically send probes (for example, ping messages or DNS queries) to all of the LDNSs around the world. One drawback of this approach is that many LDNSs are configured to not respond to such probes.

An alternative to sending extraneous traffic for measuring path properties is to use the characteristics of recent and ongoing traffic between the clients and CDN servers. For instance, the delay between a client and a cluster can be estimated by examining the gap between server-to-client SYNACK and client-to-server ACK during the TCP three-way handshake. Such solutions, however, require redirecting clients to (possibly) suboptimal clusters from time to time in order to measure the properties of paths to these clusters. Although only a small number of requests need to serve as probes, the selected clients can suffer significant performance degradation when receiving content (video or otherwise) [Andrews 2002; Krishnan 2009]. Another alternative for cluster-to-client path probing is to use DNS query traffic to measure the delay between clients and clusters. Specifically, during the DNS phase (within Step 4 in Figure 7.4), the client's LDNS can be occasionally directed to different DNS authoritative servers installed at the various cluster locations, yielding DNS traffic that can then be measured between the LDNS and these cluster locations.

In this scheme, the DNS servers continue to return the optimal cluster for the client, so that delivery of videos and other Web objects does not suffer [Huang 2010].

A very different approach to matching clients with CDN servers is to use **IP anycast** [RFC 1546]. The idea behind IP anycast is to have the routers in the Internet route the client's packets to the "closest" cluster, as determined by BGP. Specifically, as shown in Figure 7.5, during the IP-anycast configuration stage, the CDN company assigns the *same* IP address to each of its clusters, and *uses standard BGP* to advertise this IP address from each of the different cluster locations. When a BGP router receives multiple route advertisements for this same IP address, it treats these advertisements as providing different paths to the same physical location (when, in fact, the advertisements are for different paths to *different* physical locations). Following standard operating procedures, the BGP router will then pick the "best" (for example, closest, as determined by AS-hop counts) route to the IP address according to its local route selection mechanism. For example, if one BGP route

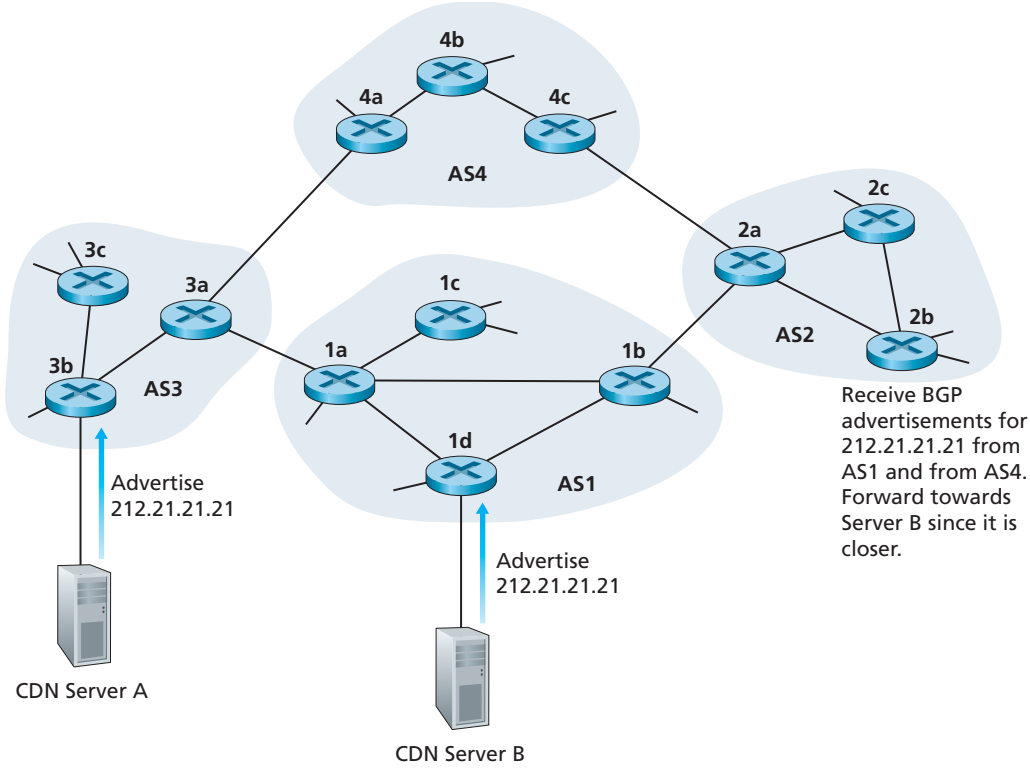


Figure 7.5 ♦ Using IP anycast to route clients to closest CDN cluster

(corresponding to one location) is only one AS hop away from the router, and all other BGP routes (corresponding to other locations) are two or more AS hops away, then the BGP router would typically choose to route packets to the location that needs to traverse only one AS (see Section 4.6). After this initial configuration phase, the CDN can do its main job of distributing content. When any client wants to see any video, the CDN's DNS returns the anycast address, no matter where the client is located. When the client sends a packet to that IP address, the packet is routed to the "closest" cluster as determined by the preconfigured forwarding tables, which were configured with BGP as just described. This approach has the advantage of finding the cluster that is closest to the client rather than the cluster that is closest to the client's LDNS. However, the IP anycast strategy again does not take into account the dynamic nature of the Internet over short time scales [Ballani 2006].

Besides network-related considerations such as delay, loss, and bandwidth performance, there are many additional important factors that go into designing a cluster selection strategy. Load on the clusters is one such factor—clients should not be directed to overloaded clusters. ISP delivery cost is another factor—the clusters may be chosen so that specific ISPs are used to carry CDN-to-client traffic, taking into account the different cost structures in the contractual relationships between ISPs and cluster operators.

7.2.5 Case Studies: Netflix, YouTube, and Kankan

We conclude our discussion of streaming stored video by taking a look at three highly successful large-scale deployments: Netflix, YouTube, and Kankan. We'll see that all these systems take very different approaches, yet employ many of the underlying principles discussed in this section.

Netflix

Generating almost 30 percent of the downstream U.S. Internet traffic in 2011, Netflix has become the leading service provider for online movies and TV shows in the United States [Sandvine 2011]. In order to rapidly deploy its large-scale service, Netflix has made extensive use of third-party cloud services and CDNs. Indeed, Netflix is an interesting example of a company deploying a large-scale online service by renting servers, bandwidth, storage, and database services from third parties while using hardly any infrastructure of its own. The following discussion is adapted from a very readable measurement study of the Netflix architecture [Adhikari 2012]. As we'll see, Netflix employs many of the techniques covered earlier in this section, including video distribution using a CDN (actually multiple CDNs) and adaptive streaming over HTTP.

Figure 7.6 shows the basic architecture of the Netflix video-streaming platform. It has four major components: the registration and payment servers, the Amazon cloud, multiple CDN providers, and clients. In its own hardware infrastructure, Netflix maintains registration and payment servers, which handle registration of new

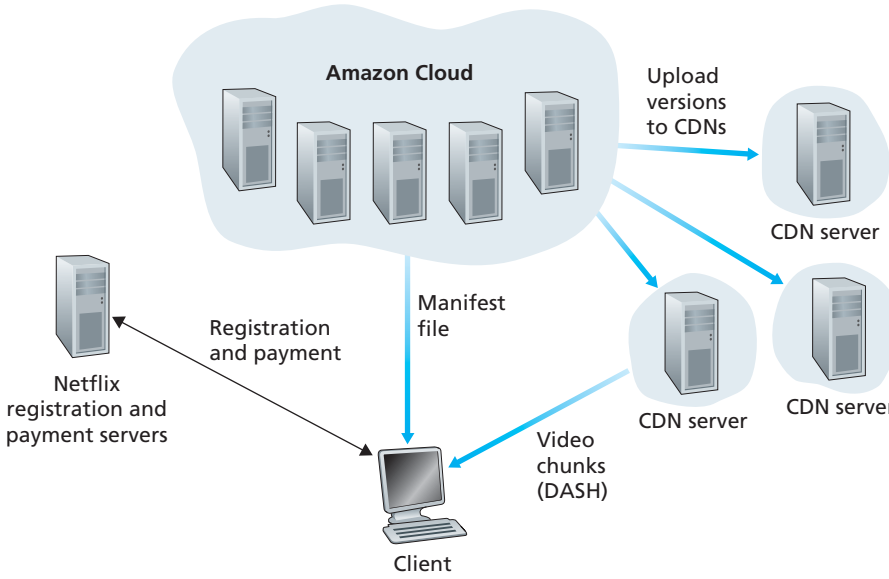


Figure 7.6 ♦ Netflix video streaming platform

accounts and capture credit-card payment information. Except for these basic functions, Netflix runs its online service by employing machines (or virtual machines) in the Amazon cloud. Some of the functions taking place in the Amazon cloud include:

- *Content ingestion.* Before Netflix can distribute a movie to its customers, it must first ingest and process the movie. Netflix receives studio master versions of movies and uploads them to hosts in the Amazon cloud.
- *Content processing.* The machines in the Amazon cloud create many different formats for each movie, suitable for a diverse array of client video players running on desktop computers, smartphones, and game consoles connected to televisions. A different version is created for each of these formats and at multiple bit rates, allowing for adaptive streaming over HTTP using DASH.
- *Uploading versions to the CDNs.* Once all of the versions of a movie have been created, the hosts in the Amazon cloud upload the versions to the CDNs.

To deliver the movies to its customers on demand, Netflix makes extensive use of CDN technology. In fact, as of this writing in 2012, Netflix employs not one but *three* third-party CDN companies at the same time—Akamai, Limelight, and Level-3.

Having described the components of the Netflix architecture, let's take a closer look at the interaction between the client and the various servers that are involved in

movie delivery. The Web pages for browsing the Netflix video library are served from servers in the Amazon cloud. When the user selects a movie to "Play Now," the user's client obtains a manifest file, also from servers in the Amazon cloud. The manifest file includes a variety of information, including a ranked list of CDNs and the URLs for the different versions of the movie, which are used for DASH playback. The ranking of the CDNs is determined by Netflix, and may change from one streaming session to the next. Typically the client will select the CDN that is ranked highest in the manifest file. After the client selects a CDN, the CDN leverages DNS to redirect the client to a specific CDN server, as described in Section 7.2.4. The client and that CDN server then interact using DASH. Specifically, as described in Section 7.2.3, the client uses the byte-range header in HTTP GET request messages, to request chunks from the different versions of the movie. Netflix uses chunks that are approximately four-seconds long [Adhikari 2012]. While the chunks are being downloaded, the client measures the received throughput and runs a rate-determination algorithm to determine the quality of the next chunk to request.

Netflix embodies many of the key principles discussed earlier in this section, including adaptive streaming and CDN distribution. Netflix also nicely illustrates how a major Internet service, generating almost 30 percent of Internet traffic, can run almost entirely on a third-party cloud and third-party CDN infrastructures, using very little infrastructure of its own!

YouTube

With approximately half a billion videos in its library and half a billion video views per day [Ding 2011], YouTube is indisputably the world's largest video-sharing site. YouTube began its service in April 2005 and was acquired by Google in November 2006. Although the Google/YouTube design and protocols are proprietary, through several independent measurement efforts we can gain a basic understanding about how YouTube operates [Zink 2009; Torres 2011; Adhikari 2011a].

As with Netflix, YouTube makes extensive use of CDN technology to distribute its videos [Torres 2011]. Unlike Netflix, however, Google does not employ third-party CDNs but instead uses its own private CDN to distribute YouTube videos. Google has installed server clusters in many hundreds of different locations. From a subset of about 50 of these locations, Google distributes YouTube videos [Adhikari 2011a]. Google uses DNS to redirect a customer request to a specific cluster, as described in Section 7.2.4. Most of the time, Google's cluster selection strategy directs the client to the cluster for which the RTT between client and cluster is the lowest; however, in order to balance the load across clusters, sometimes the client is directed (via DNS) to a more distant cluster [Torres 2011]. Furthermore, if a cluster does not have the requested video, instead of fetching it from somewhere else and relaying it to the client, the cluster may return an HTTP redirect message, thereby redirecting the client to another cluster [Torres 2011].

YouTube employs HTTP streaming, as discussed in Section 7.2.2. YouTube often makes a small number of different versions available for a video, each with a different bit rate and corresponding quality level. As of 2011, YouTube does not employ adaptive streaming (such as DASH), but instead requires the user to manually select a version. In order to save bandwidth and server resources that would be wasted by repositioning or early termination, YouTube uses the HTTP byte range request to limit the flow of transmitted data after a target amount of video is prefetched.

A few million videos are uploaded to YouTube every day. Not only are YouTube videos streamed from server to client over HTTP, but YouTube uploaders also upload their videos from client to server over HTTP. YouTube processes each video it receives, converting it to a YouTube video format and creating multiple versions at different bit rates. This processing takes place entirely within Google data centers. Thus, in stark contrast to Netflix, which runs its service almost entirely on third-party infrastructures, Google runs the entire YouTube service within its own vast infrastructure of data centers, private CDN, and private global network interconnecting its data centers and CDN clusters. (See the case study on Google's network infrastructure in Section 7.2.4.)

Kankan

We just saw that for both the Netflix and YouTube services, servers operated by CDNs (either third-party or private CDNs) stream videos to clients. Netflix and YouTube not only have to pay for the server hardware (either directly through ownership or indirectly through rent), but also for the bandwidth the servers use to distribute the videos. Given the scale of these services and the amount of bandwidth they are consuming, such a "client-server" deployment is extremely costly.

We conclude this section by describing an entirely different approach for providing video on demand over the Internet at a large scale—one that allows the service provider to significantly reduce its infrastructure and bandwidth costs. As you might suspect, this approach uses P2P delivery instead of client-server (via CDNs) delivery. P2P video delivery is used with great success by several companies in China, including Kankan (owned and operated by Xunlei), PPTV (formerly PPLive), and PPs (formerly PPstream). Kankan, currently the leading P2P-based video-on-demand provider in China, has over 20 million unique users viewing its videos every month.

At a high level, P2P video streaming is very similar to BitTorrent file downloading (discussed in Chapter 2). When a peer wants to see a video, it contacts a tracker (which may be centralized or peer-based using a DHT) to discover other peers in the system that have a copy of that video. This peer then requests chunks of the video file in parallel from these other peers that have the file. Different from downloading with BitTorrent, however, requests are preferentially made for chunks that are to be played back in the near future in order to ensure continuous playback.

The Kankan design employs a tracker and its own DHT for tracking content. Swarm sizes for the most popular content involve tens of thousands of peers, typically larger than the largest swarms in BitTorrent [Dhungel 2012]. The Kankan protocols—for communication between peer and tracker, between peer and DHT, and among peers—are all proprietary. Interestingly, for distributing video chunks among peers, Kankan uses UDP whenever possible, leading to massive amounts of UDP traffic within China's Internet [Zhang M 2010].

7.3 Voice-over-IP

Real-time conversational voice over the Internet is often referred to as **Internet telephony**, since, from the user's perspective, it is similar to the traditional circuit-switched telephone service. It is also commonly called **Voice-over-IP (VoIP)**. In this section we describe the principles and protocols underlying VoIP. Conversational video is similar in many respects to VoIP, except that it includes the video of the participants as well as their voices. To keep the discussion focused and concrete, we focus here only on voice in this section rather than combined voice and video.

7.3.1 Limitations of the Best-Effort IP Service

The Internet's network-layer protocol, IP, provides best-effort service. That is to say the service makes its best effort to move each datagram from source to destination as quickly as possible but makes no promises whatsoever about getting the packet to the destination within some delay bound or about a limit on the percentage of packets lost. The lack of such guarantees poses significant challenges to the design of real-time conversational applications, which are acutely sensitive to packet delay, jitter, and loss.

In this section, we'll cover several ways in which the performance of VoIP over a best-effort network can be enhanced. Our focus will be on application-layer techniques, that is, approaches that do not require any changes in the network core or even in the transport layer at the end hosts. To keep the discussion concrete, we'll discuss the limitations of best-effort IP service in the context of a specific VoIP example. The sender generates bytes at a rate of 8,000 bytes per second; every 20 msecs the sender gathers these bytes into a chunk. A chunk and a special header (discussed below) are encapsulated in a UDP segment, via a call to the socket interface. Thus, the number of bytes in a chunk is (20 msecs)·(8,000 bytes/sec) = 160 bytes, and a UDP segment is sent every 20 msecs.

If each packet makes it to the receiver with a constant end-to-end delay, then packets arrive at the receiver periodically every 20 msecs. In these ideal conditions,

the receiver can simply play back each chunk as soon as it arrives. But unfortunately, some packets can be lost and most packets will not have the same end-to-end delay, even in a lightly congested Internet. For this reason, the receiver must take more care in determining (1) when to play back a chunk, and (2) what to do with a missing chunk.

Packet Loss

Consider one of the UDP segments generated by our VoIP application. The UDP segment is encapsulated in an IP datagram. As the datagram wanders through the network, it passes through router buffers (that is, queues) while waiting for transmission on outbound links. It is possible that one or more of the buffers in the path from sender to receiver is full, in which case the arriving IP datagram may be discarded, never to arrive at the receiving application.

Loss could be eliminated by sending the packets over TCP (which provides for reliable data transfer) rather than over UDP. However, retransmission mechanisms are often considered unacceptable for conversational real-time audio applications such as VoIP, because they increase end-to-end delay [Bolot 1996]. Furthermore, due to TCP congestion control, packet loss may result in a reduction of the TCP sender's transmission rate to a rate that is lower than the receiver's drain rate, possibly leading to buffer starvation. This can have a severe impact on voice intelligibility at the receiver. For these reasons, most existing VoIP applications run over UDP by default. [Baset 2006] reports that UDP is used by Skype unless a user is behind a NAT or firewall that blocks UDP segments (in which case TCP is used).

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

End-to-End Delay

End-to-end delay is the accumulation of transmission, processing, and queuing delays in routers; propagation delays in links; and end-system processing delays. For real-time conversational applications, such as VoIP, end-to-end delays smaller than 150 msecs are not perceived by a human listener; delays between 150 and 400

msecs can be acceptable but are not ideal; and delays exceeding 400 msecs can seriously hinder the interactivity in voice conversations. The receiving side of a VoIP application will typically disregard any packets that are delayed more than a certain threshold, for example, more than 400 msecs. Thus, packets that are delayed by more than the threshold are effectively lost.

Packet Jitter

A crucial component of end-to-end delay is the varying queuing delays that a packet experiences in the network's routers. Because of these varying delays, the time from when a packet is generated at the source until it is received at the receiver can fluctuate from packet to packet, as shown in Figure 7.1. This phenomenon is called **jitter**. As an example, consider two consecutive packets in our VoIP application. The sender sends the second packet 20 msecs after sending the first packet. But at the receiver, the spacing between these packets can become greater than 20 msecs. To see this, suppose the first packet arrives at a nearly empty queue at a router, but just before the second packet arrives at the queue a large number of packets from other sources arrive at the same queue. Because the first packet experiences a small queuing delay and the second packet suffers a large queuing delay at this router, the first and second packets become spaced by more than 20 msecs. The spacing between consecutive packets can also become less than 20 msecs. To see this, again consider two consecutive packets. Suppose the first packet joins the end of a queue with a large number of packets, and the second packet arrives at the queue before this first packet is transmitted and before any packets from other sources arrive at the queue. In this case, our two packets find themselves one right after the other in the queue. If the time it takes to transmit a packet on the router's outbound link is less than 20 msecs, then the spacing between first and second packets becomes less than 20 msecs.

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

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

7.3.2 Removing Jitter at the Receiver for Audio

For our VoIP application, where packets are being generated periodically, the receiver should attempt to provide periodic playout of voice chunks in the presence

of random network jitter. This is typically done by combining the following two mechanisms:

- *Prepending each chunk with a* ***timestamp.*** The sender stamps each chunk with the time at which the chunk was generated.
- ***Delaying playout*** *of chunks at the receiver*. As we saw in our earlier discussion of Figure 7.1, the playout delay of the received audio chunks must be long enough so that most of the packets are received before their scheduled playout times. This playout delay can either be fixed throughout the duration of the audio session or vary adaptively during the audio session lifetime.

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

Fixed Playout Delay

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

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

The trade-off between the playback delay and packet loss is illustrated in Figure 7.7. The figure shows the times at which packets are generated and played out for a single talk spurt. Two distinct initial playout delays are considered. As shown by the leftmost staircase, the sender generates packets at regular intervals—say, every 20 msecs. The first packet in this talk spurt is received at time r. As shown in the figure, the arrivals of subsequent packets are not evenly spaced due to the network jitter.

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

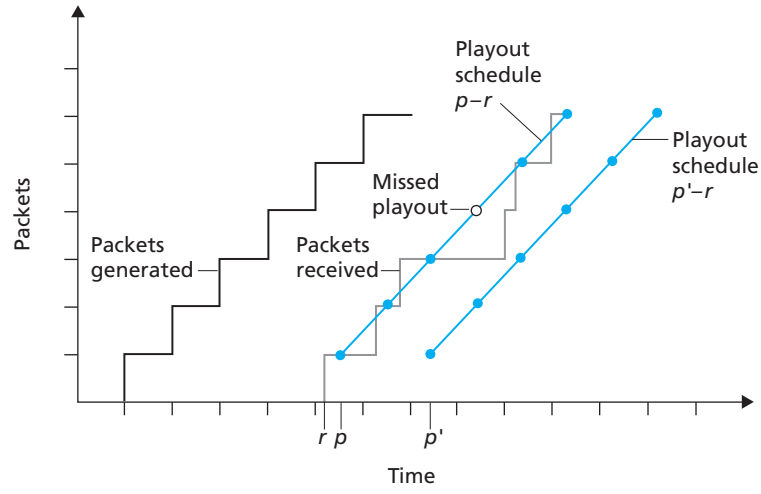


Figure 7.7 ♦ Packet loss for different fixed playout delays

Adaptive Playout Delay

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

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

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

t_i = the timestamp of the ith packet = the time the packet was generated by the sender

r_i = the time packet i is received by receiver

p_i = the time packet i is played at receiver

The end-to-end network delay of the ith packet is $r_i - t_i$. Due to network jitter, this delay will vary from packet to packet. Let d_i denote an estimate of the *average*

network delay upon reception of the ith packet. This estimate is constructed from the timestamps as follows:

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

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

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

The estimates d_i and v_i are calculated for every packet received, although they are used only to determine the playout point for the first packet in any talk spurt.

Once having calculated these estimates, the receiver employs the following algorithm for the playout of packets. If packet i is the first packet of a talk spurt, its playout time, p_i, is computed as:

$$p_i = t_i + d_i + Kv_i$$

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

$$q_i = p_i - t_i$$

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

$$p_j = t_j + q_i$$

The algorithm just described makes perfect sense assuming that the receiver can tell whether a packet is the first packet in the talk spurt. This can be done by examining the signal energy in each received packet.

7.3.3 Recovering from Packet Loss

We have discussed in some detail how a VoIP application can deal with packet jitter. We now briefly describe several schemes that attempt to preserve acceptable audio

quality in the presence of packet loss. Such schemes are called **loss recovery schemes**. Here we define packet loss in a broad sense: A packet is lost either if it never arrives at the receiver or if it arrives after its scheduled playout time. Our VoIP example will again serve as a context for describing loss recovery schemes.

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

Forward Error Correction (FEC)

The basic idea of FEC is to add redundant information to the original packet stream. For the cost of marginally increasing the transmission rate, the redundant information can be used to reconstruct approximations or exact versions of some of the lost packets. Following [Bolot 1996] and [Perkins 1998], we now outline two simple FEC mechanisms. The first mechanism sends a redundant encoded chunk after every n chunks. The redundant chunk is obtained by exclusive OR-ing the n original chunks [Shacham 1990]. In this manner if any one packet of the group of $n + 1$ packets is lost, the receiver can fully reconstruct the lost packet. But if two or more packets in a group are lost, the receiver cannot reconstruct the lost packets. By keeping $n + 1$, the group size, small, a large fraction of the lost packets can be recovered when loss is not excessive. However, the smaller the group size, the greater the relative increase of the transmission rate. In particular, the transmission rate increases by a factor of $1/n$, so that, if $n = 3$, then the transmission rate increases by 33 percent. Furthermore, this simple scheme increases the playout delay, as the receiver must wait to receive the entire group of packets before it can begin playout. For more practical details about how FEC works for multimedia transport see [RFC 5109].

The second FEC mechanism is to send a lower-resolution audio stream as the redundant information. For example, the sender might create a nominal audio stream and a corresponding low-resolution, low-bit rate audio stream. (The nominal stream could be a PCM encoding at 64 kbps, and the lower-quality stream could be a GSM encoding at 13 kbps.) The low-bit rate stream is referred to as the redundant stream. As shown in Figure 7.8, the sender constructs the nth packet by taking the nth chunk from the nominal stream and appending to it the $(n - 1)$st chunk from the redundant stream. In this manner, whenever there is nonconsecutive packet loss, the receiver can conceal the loss by playing out the low-bit rate encoded chunk that arrives with the subsequent packet. Of course, low-bit rate chunks give lower quality than the nominal chunks. However, a stream of mostly high-quality chunks, occasional low-quality chunks, and no missing chunks gives good overall audio quality. Note that in this scheme, the receiver only has to receive two packets before playback, so that the increased playout delay is small. Furthermore, if the low-bit

rate encoding is much less than the nominal encoding, then the marginal increase in the transmission rate will be small.

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

Interleaving

As an alternative to redundant transmission, a VoIP application can send interleaved audio. As shown in Figure 7.9, the sender resequences units of audio data before transmission, so that originally adjacent units are separated by a certain distance in the transmitted stream. Interleaving can mitigate the effect of packet losses. If, for example, units are 5 msecs in length and chunks are 20 msecs (that is, four units per chunk), then the first chunk could contain units 1, 5, 9, and 13; the second chunk could contain units 2, 6, 10, and 14; and so on. Figure 7.9 shows that the loss of a single packet from an interleaved stream results in multiple small gaps in the reconstructed stream, as opposed to the single large gap that would occur in a noninterleaved stream.

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

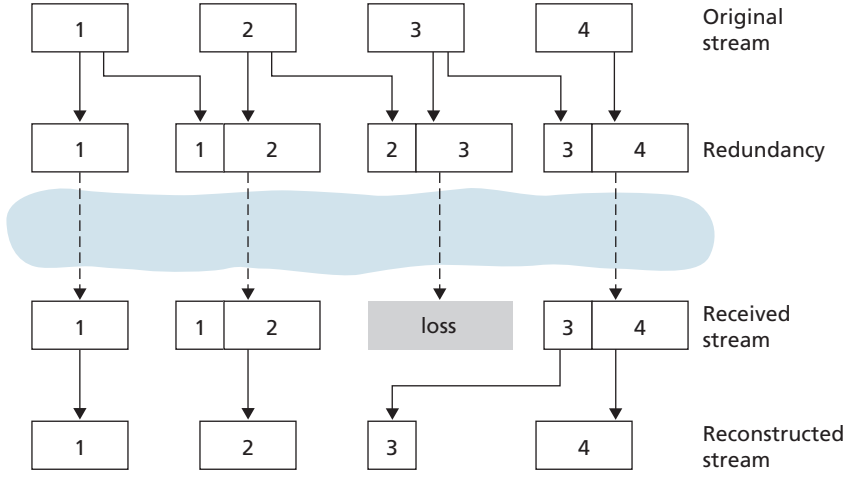


Figure 7.8 ♦ Piggybacking lower-quality redundant information

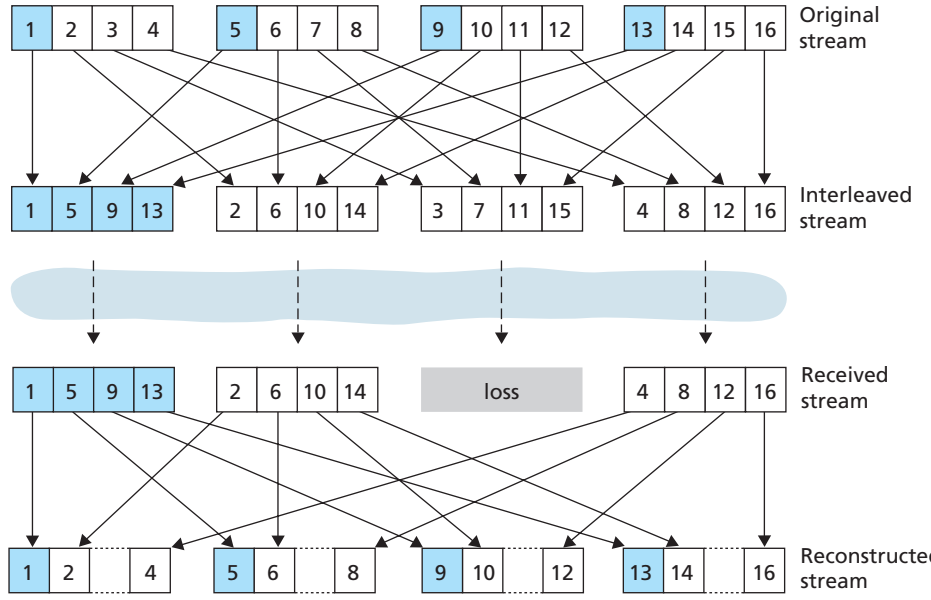


Figure 7.9 ♦ Sending interleaved audio

Error Concealment

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

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

7.3.4 Case Study: VoIP with Skype

Skype is an immensely popular VoIP application with over 50 million accounts active on a daily basis. In addition to providing host-to-host VoIP service, Skype offers host-to-phone services, phone-to-host services, and multi-party host-to-host

video conferencing services. (Here, a host is again any Internet connected IP device, including PCs, tablets, and smartphones.) Skype was acquired by Microsoft in 2011 for over $8 billion.

Because the Skype protocol is proprietary, and because all Skype's control and media packets are encrypted, it is difficult to precisely determine how Skype operates. Nevertheless, from the Skype Web site and several measurement studies, researchers have learned how Skype generally works [Baset 2006; Guha 2006; Chen 2006; Suh 2006; Ren 2006; Zhang X 2012]. For both voice and video, the Skype clients have at their disposal many different codecs, which are capable of encoding the media at a wide range of rates and qualities. For example, video rates for Skype have been measured to be as low as 30 kbps for a low-quality session up to almost 1 Mbps for a high quality session [Zhang X 2012]. Typically, Skype's audio quality is better than the "POTS" (Plain Old Telephone Service) quality provided by the wire-line phone system. (Skype codecs typically sample voice at 16,000 samples/sec or higher, which provides richer tones than POTS, which samples at 8,000/sec.) By default, Skype sends audio and video packets over UDP. However, control packets are sent over TCP, and media packets are also sent over TCP when firewalls block UDP streams. Skype uses FEC for loss recovery for both voice and video streams sent over UDP. The Skype client also adapts the audio and video streams it sends to current network conditions, by changing video quality and FEC overhead [Zhang X 2012].

Skype uses P2P techniques in a number of innovative ways, nicely illustrating how P2P can be used in applications that go beyond content distribution and file sharing. As with instant messaging, host-to-host Internet telephony is inherently P2P since, at the heart of the application, pairs of users (that is, peers) communicate with each other in real time. But Skype also employs P2P techniques for two other important functions, namely, for user location and for NAT traversal.

As shown in Figure 7.10, the peers (hosts) in Skype are organized into a hierarchical overlay network, with each peer classified as a super peer or an ordinary peer. Skype maintains an index that maps Skype usernames to current IP addresses (and port numbers). This index is distributed over the super peers. When Alice wants to call Bob, her Skype client searches the distributed index to determine Bob's current IP address. Because the Skype protocol is proprietary, it is currently not known how the index mappings are organized across the super peers, although some form of DHT organization is very possible.

P2P techniques are also used in Skype **relays**, which are useful for establishing calls between hosts in home networks. Many home network configurations provide access to the Internet through NATs, as discussed in Chapter 4. Recall that a NAT prevents a host from outside the home network from initiating a connection to a host within the home network. If *both* Skype callers have NATs, then there is a problem—neither can accept a call initiated by the other, making a call seemingly impossible. The clever use of super peers and relays nicely solves this problem. Suppose that when Alice signs in, she is assigned to a non-NATed super peer and initiates a session to that super peer. (Since Alice is initiating the session, her NAT permits this session.) This session allows Alice and her super peer to

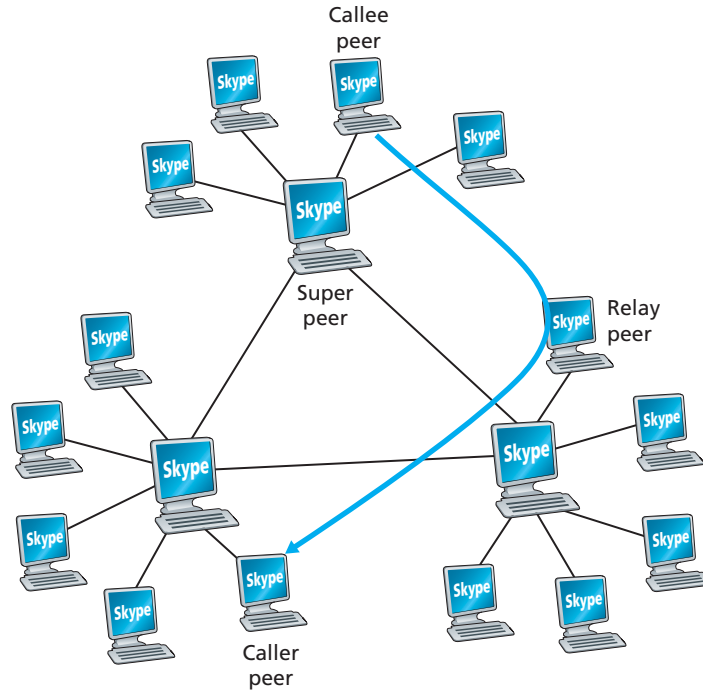


Figure 7.10 ♦ Skype peers

exchange control messages. The same happens for Bob when he signs in. Now, when Alice wants to call Bob, she informs her super peer, who in turn informs Bob's super peer, who in turn informs Bob of Alice's incoming call. If Bob accepts the call, the two super peers select a third non-NATed super peer—the relay peer—whose job will be to relay data between Alice and Bob. Alice's and Bob's super peers then instruct Alice and Bob respectively to initiate a session with the relay. As shown in Figure 7.10, Alice then sends voice packets to the relay over the Alice-to-relay connection (which was initiated by Alice), and the relay then forwards these packets over the relay-to-Bob connection (which was initiated by Bob); packets from Bob to Alice flow over these same two relay connections in reverse. And *voila!*—Bob and Alice have an end-to-end connection even though neither can accept a session originating from outside.

Up to now, our discussion on Skype has focused on calls involving two persons. Now let's examine multi-party audio conference calls. With $N > 2$ participants, if each user were to send a copy of its audio stream to each of the $N - 1$ other users, then a total of $N(N - 1)$ audio streams would need to be sent into the network to support the audio conference. To reduce this bandwidth usage, Skype employs a clever distribution

technique. Specifically, each user sends its audio stream to the conference initiator. The conference initiator combines the audio streams into one stream (basically by adding all the audio signals together) and then sends a copy of each combined stream to each of the other $N - 1$ participants. In this manner, the number of streams is reduced to $2(N - 1)$. For ordinary two-person video conversations, Skype routes the call peer-to-peer, unless NAT traversal is required, in which case the call is relayed through a non-NATed peer, as described earlier. For a video conference call involving $N > 2$ participants, due to the nature of the video medium, Skype does not combine the call into one stream at one location and then redistribute the stream to all the participants, as it does for voice calls. Instead, each participant's video stream is routed to a server cluster (located in Estonia as of 2011), which in turn relays to each participant the $N - 1$ streams of the $N - 1$ other participants [Zhang X 2012]. You may be wondering why each participant sends a copy to a server rather than directly sending a copy of its video stream to each of the other $N - 1$ participants? Indeed, for both approaches, $N(N - 1)$ video streams are being collectively received by the N participants in the conference. The reason is, because upstream link bandwidths are significantly lower than downstream link bandwidths in most access links, the upstream links may not be able to support the $N - 1$ streams with the P2P approach.

VoIP systems such as Skype, QQ, and Google Talk introduce new privacy concerns. Specifically, when Alice and Bob communicate over VoIP, Alice can sniff Bob's IP address and then use geo-location services [MaxMind 2012; Quova 2012] to determine Bob's current location and ISP (for example, his work or home ISP). In fact, with Skype it is possible for Alice to block the transmission of certain packets during call establishment so that she obtains Bob's current IP address, say every hour, without Bob knowing that he is being tracked and without being on Bob's contact list. Furthermore, the IP address discovered from Skype can be correlated with IP addresses found in BitTorrent, so that Alice can determine the files that Bob is downloading [LeBlond 2011]. Moreover, it is possible to partially decrypt a Skype call by doing a traffic analysis of the packet sizes in a stream [White 2011].

7.4 Protocols for Real-Time Conversational Applications

Real-time conversational applications, including VoIP and video conferencing, are compelling and very popular. 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 conversational applications, independent companies are creating new products that interoperate with each other. In this section we examine RTP and SIP for real-time conversational applications. Both standards are enjoying widespread implementation in industry products.

7.4.1 RTP

In the previous section, we learned that the sender side of a VoIP application appends header fields to the audio 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, ACC, 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 many products and research prototypes. It is also complementary to other important real-time interactive protocols, such as SIP.

In this section, we provide an introduction to RTP. We also encourage you to visit Henning Schulzrinne's RTP site [Schulzrinne-RTP 2012], which provides a wealth of information on the subject. Also, you may want to visit the RAT site [RAT 2012], which documents VoIP 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 VoIP software and they both incorporate RTP into their product, there may be some hope that a user using one of the VoIP products will be able to communicate with a user using the other VoIP product. In Section 7.4.2, we'll see that RTP is often used in conjunction with SIP, an important standard for Internet telephony.

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 7.11, 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

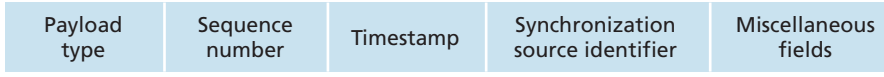


Figure 7.11 ♦ RTP header fields

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

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

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.

- *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.

7.4.2 SIP

The Session Initiation Protocol (SIP), defined in [RFC 3261; RFC 5411], is an open and 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 7.12 illustrates the SIP call-establishment process.

In Figure 7.12, 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 7.12 shows Alice talking after Bob, but in truth they

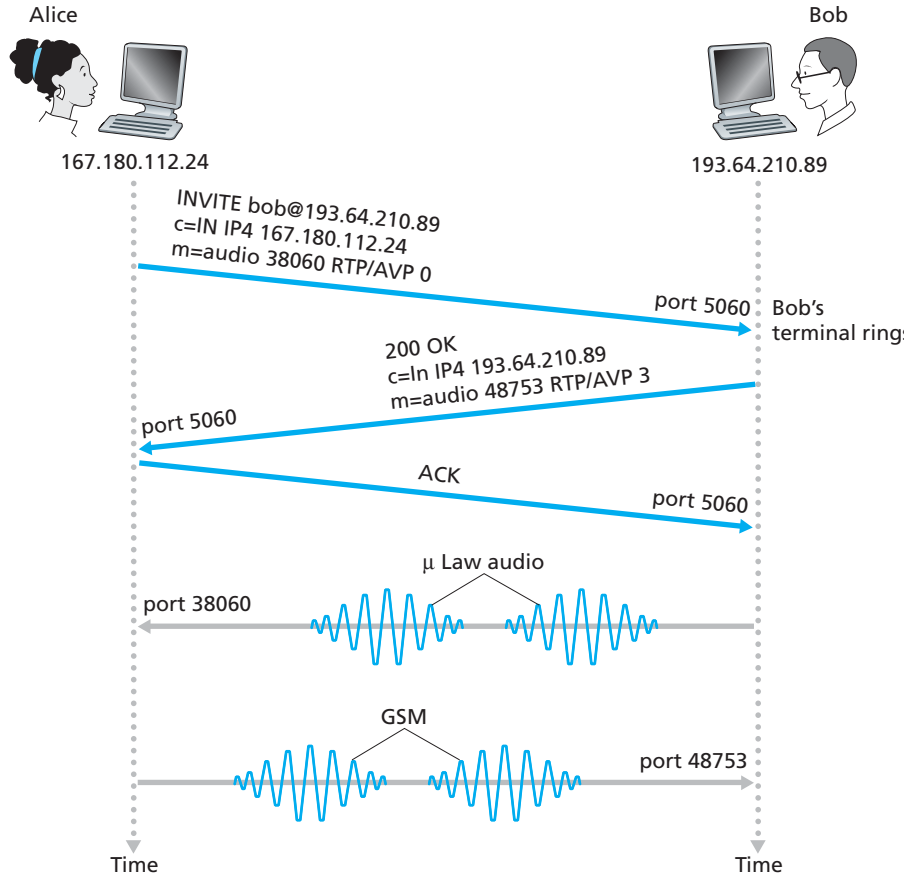


Figure 7.12 ♦ SIP call establishment when Alice knows Bob's IP address

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 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 a VoIP 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 7.12, 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 7.13, 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 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

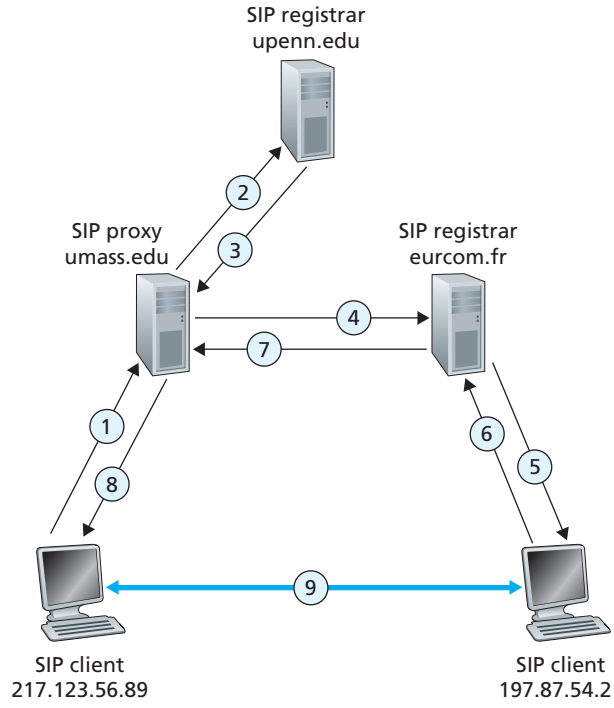


Figure 7.13 ♦ Session initiation, involving SIP proxies and registrars

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 2012]. In particular, on this site you will find open source software for SIP clients and servers [SIP Software 2012].

7.5 Network Support for Multimedia

In Sections 7.2 through 7.4, we learned how application-level mechanisms such as client buffering, prefetching, adapting media quality to available bandwidth, adaptive playout, and loss mitigation techniques can be used by multimedia applications

to improve a multimedia application's performance. We also learned how content distribution networks and P2P overlay networks can be used to provide a *system-level* approach for delivering multimedia content. These techniques and approaches are all designed to be used in today's best-effort Internet. Indeed, they are in use today precisely because the Internet provides only a single, best-effort class of service. But as designers of computer networks, we can't help but ask whether the *network* (rather than the applications or application-level infrastructure alone) might provide mechanisms to support multimedia content delivery. As we'll see shortly, the answer is, of course, "yes"! But we'll also see that a number of these new network-level mechanisms have yet to be widely deployed. This may be due to their complexity and to the fact that application-level techniques together with best-effort service and properly dimensioned network resources (for example, bandwidth) can indeed provide a "good-enough" (even if not-always-perfect) end-to-end multimedia delivery service.

Table 7.4 summarizes three broad approaches towards providing network-level support for multimedia applications.

- *Making the best of best-effort service.* The application-level mechanisms and infrastructure that we studied in Sections 7.2 through 7.4 can be successfully used in a well-dimensioned network where packet loss and excessive end-to-end

Approach	Granularity	Guarantee	Mechanisms	Complexity	Deployment to date
Making the best of best-effort service.	all traffic treated equally	none, or soft	application-layer support, CDNs, overlays, network-level resource provisioning	minimal	everywhere
Differentiated service	different classes of traffic treated differently	none, or soft	packet marking, policing, scheduling	medium	some
Per-connection Quality-of-Service (QoS) Guarantees	each source-destination flows treated differently	soft or hard, once flow is admitted	packet marking, policing, scheduling; call admission and signaling	light	little

Table 7.4 ♦ Three network-level approaches to supporting multimedia applications

delay rarely occur. When demand increases are forecasted, the ISPs deploy additional bandwidth and switching capacity to continue to ensure satisfactory delay and packet-loss performance [Huang 2005]. We'll discuss such **network dimensioning** further in Section 7.5.1.

- *Differentiated service.* Since the early days of the Internet, it's been envisioned that different types of traffic (for example, as indicated in the Type-of-Service field in the IP4v packet header) could be provided with different classes of service, rather than a single one-size-fits-all best-effort service. With **differentiated service**, one type of traffic might be given strict priority over another class of traffic when both types of traffic are queued at a router. For example, packets belonging to a real-time conversational application might be given priority over other packets due to their stringent delay constraints. Introducing differentiated service into the network will require new mechanisms for packet marking (indicating a packet's class of service), packet scheduling, and more. We'll cover differentiated service, and new network mechanisms needed to implement this service, in Section 7.5.2.
- *Per-connection Quality-of-Service (QoS) Guarantees.* With per-connection QoS guarantees, each instance of an application explicitly reserves end-to-end bandwidth and thus has a guaranteed end-to-end performance. A **hard guarantee** means the application will receive its requested quality of service (QoS) with certainty. A **soft guarantee** means the application will receive its requested quality of service with high probability. For example, if a user wants to make a VoIP call from Host A to Host B, the user's VoIP application reserves bandwidth explicitly in each link along a route between the two hosts. But permitting applications to make reservations and requiring the network to honor the reservations requires some big changes. First, we need a protocol that, on behalf of the applications, reserves link bandwidth on the paths from the senders to their receivers. Second, we'll need new scheduling policies in the router queues so that per-connection bandwidth reservations can be honored. Finally, in order to make a reservation, the applications must give the network a description of the traffic that they intend to send into the network and the network will need to police each application's traffic to make sure that it abides by that description. These mechanisms, when combined, require new and complex software in hosts and routers. Because per-connection QoS guaranteed service has not seen significant deployment, we'll cover these mechanisms only briefly in Section 7.5.3.

7.5.1 Dimensioning Best-Effort Networks

Fundamentally, the difficulty in supporting multimedia applications arises from their stringent performance requirements––low end-to-end packet delay, delay

jitter, and loss—and the fact that packet delay, delay jitter, and loss occur whenever the network becomes congested. A first approach to improving the quality of multimedia applications—an approach that can often be used to solve just about any problem where resources are constrained—is simply to "throw money at the problem" and thus simply avoid resource contention. In the case of networked multimedia, this means providing enough link capacity throughout the network so that network congestion, and its consequent packet delay and loss, never (or only very rarely) occurs. With enough link capacity, packets could zip through today's Internet without queuing delay or loss. From many perspectives this is an ideal situation—multimedia applications would perform perfectly, users would be happy, and this could all be achieved with no changes to Internet's best-effort architecture.

The question, of course, is how much capacity is "enough" to achieve this nirvana, and whether the costs of providing "enough" bandwidth are practical from a business standpoint to the ISPs. The question of how much capacity to provide at network links in a given topology to achieve a given level of performance is often known as **bandwidth provisioning**. The even more complicated problem of how to design a network topology (where to place routers, how to interconnect routers with links, and what capacity to assign to links) to achieve a given level of end-to-end performance is a network design problem often referred to as **network dimensioning**. Both bandwidth provisioning and network dimensioning are complex topics, well beyond the scope of this textbook. We note here, however, that the following issues must be addressed in order to predict application-level performance between two network end points, and thus provision enough capacity to meet an application's performance requirements.

- *Models of traffic demand between network end points.* Models may need to be specified at both the call level (for example, users "arriving" to the network and starting up end-to-end applications) and at the packet level (for example, packets being generated by ongoing applications). Note that workload may change over time.
- *Well-defined performance requirements.* For example, a performance requirement for supporting delay-sensitive traffic, such as a conversational multimedia application, might be that the probability that the end-to-end delay of the packet is greater than a maximum tolerable delay be less than some small value [Fraleigh 2003].
- *Models to predict end-to-end performance for a given workload model, and techniques to find a minimal cost bandwidth allocation that will result in all user requirements being met.* Here, researchers are busy developing performance models that can quantify performance for a given workload, and optimization techniques to find minimal-cost bandwidth allocations meeting performance requirements.

Given that today's best-effort Internet could (from a technology standpoint) support multimedia traffic at an appropriate performance level if it were dimensioned to do so, the natural question is why today's Internet doesn't do so. The answers are primarily economic and organizational. From an economic standpoint, would users be willing to pay their ISPs enough for the ISPs to install sufficient bandwidth to support multimedia applications over a best-effort Internet? The organizational issues are perhaps even more daunting. Note that an end-to-end path between two multimedia end points will pass through the networks of multiple ISPs. From an organizational standpoint, would these ISPs be willing to cooperate (perhaps with revenue sharing) to ensure that the end-to-end path is properly dimensioned to support multimedia applications? For a perspective on these economic and organizational issues, see [Davies 2005]. For a perspective on provisioning tier-1 backbone networks to support delay-sensitive traffic, see [Fraleigh 2003].

7.5.2 Providing Multiple Classes of Service

Perhaps the simplest enhancement to the one-size-fits-all best-effort service in today's Internet 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 email or HTTP. Alternatively, an ISP may simply want to provide a higher quality of service to customers willing to pay more for this improved service. A number of residential wired-access ISPs and cellular wireless-access ISPs have adopted such tiered levels of service—with platinum-service subscribers receiving better performance than gold- or silver-service subscribers.

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, that is, 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-to-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." More than four decades ago, the vision of providing different levels of service to different classes of traffic was clear! However, it's taken us an equally long period of time to realize this vision.

Motivating Scenarios

Let's begin our discussion of network mechanisms for providing multiple classes of service with a few motivating scenarios.

Figure 7.14 shows a simple network scenario in which 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 further suppose that a 1 Mbps audio application (for example, a CD-quality audio call) shares the 1.5 Mbps link between R1 and R2 with an HTTP Web-browsing application that is downloading a Web page from H2 to H4.

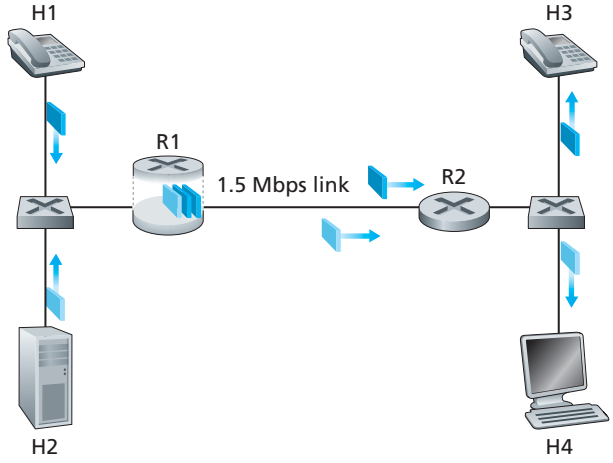


Figure 7.14 ♦ Competing audio and HTTP applications

In the best-effort Internet, the audio and HTTP 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 Web server 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 HTTP Web-browsing 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 HTTP 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 HTTP traffic using the R1-to-R2 link only when no audio traffic is queued. In order for R1 to distinguish between the audio and HTTP 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.

Note that although our example considers a competing multimedia and elastic flow, the same insight applies to the case that platinum, gold, and silver classes of service are implemented—a packet-marking mechanism is still needed to indicate that class of service to which a packet belongs.

Now suppose that the router is configured to give priority to packets marked as belonging to the 1 Mbps audio application. Since the outgoing link speed is 1.5 Mbps, even though the HTTP packets receive lower priority, they can 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 HTTP 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 class of service as the audio application, were sharing the link's bandwidth; they too could collectively starve the FTP session. Ideally, one wants a degree of isolation among classes of traffic so that one class of traffic can be protected from the other. This protection could be implemented at different places in the network—at each and every router, at first entry to the network, or at inter-domain network boundaries. This then is our second insight:

> **Insight 2:** It is desirable to provide a degree of **traffic isolation** among classes so that one class is not adversely affected by another class of traffic that misbehaves.

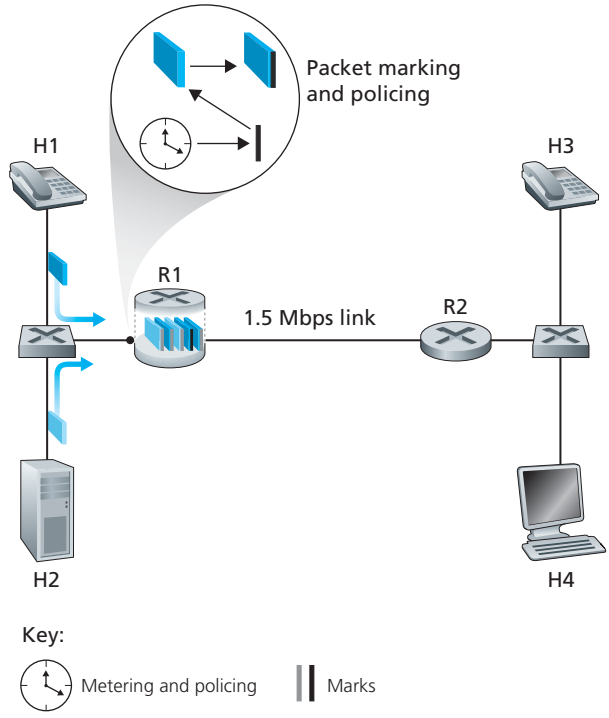


Figure 7.15 ♦ Policing (and marking) the audio and HTTP traffic classes

We'll examine several specific mechanisms for providing such isolation among traffic classes. We note here that two broad approaches can be taken. First, it is possible to perform **traffic policing**, as shown in Figure 7.15. 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'll examine shortly is perhaps the most widely used policing mechanism. In Figure 7.15, the packet classification and marking mechanism (Insight 1) and the policing mechanism (Insight 2) are both implemented together at the network's edge, either in the end system or at an edge router.

A complementary approach for providing isolation among traffic classes is for the link-level packet-scheduling mechanism to explicitly allocate a fixed

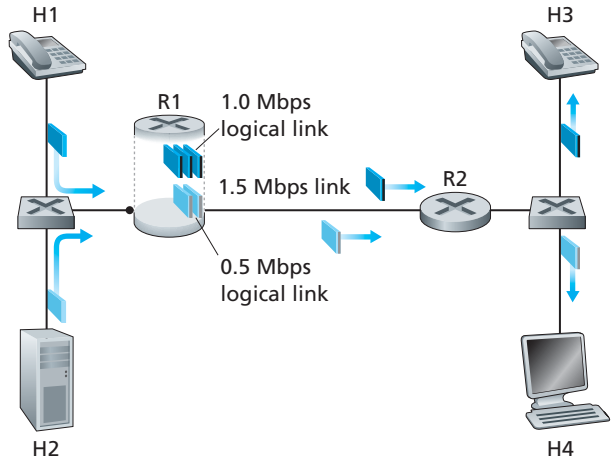


Figure 7.16 ♦ Logical isolation of audio and HTTP traffic classes

amount of link bandwidth to each class. For example, the audio class could be allocated 1 Mbps at R1, and the HTTP class could be allocated 0.5 Mbps. In this case, the audio and HTTP flows see a logical link with capacity 1.0 and 0.5 Mbps, respectively, as shown in Figure 7.16. With strict enforcement of the link-level allocation of bandwidth, a class 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 HTTP 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. Since bandwidth is a "use-it-or-lose-it" resource, there is no reason to prevent HTTP traffic from using bandwidth not used by the audio traffic. We'd like to use bandwidth as efficiently as possible, never wasting it when it could be otherwise used. This 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.

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 7.17 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 7.18 shows the FIFO queue in operation. Packet arrivals are indicated by numbered arrows above the upper timeline, with the number indicating the order

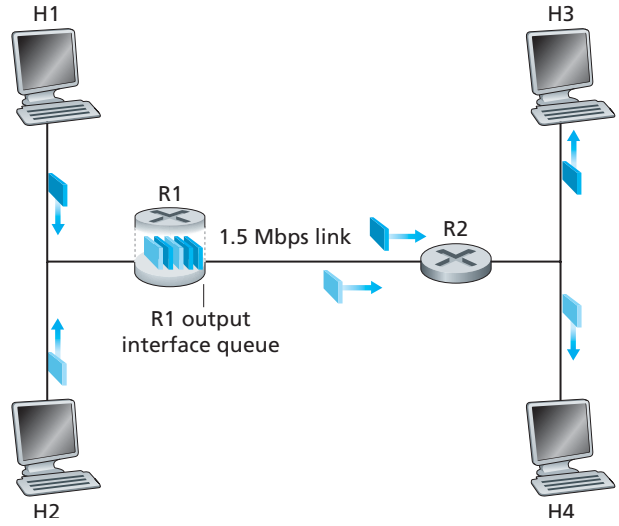


Figure 7.17 ♦ FIFO queuing abstraction

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

Priority Queuing

Under **priority queuing**, packets arriving at the output link are classified into priority classes at the output queue, as shown in Figure 7.19. 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 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 7.20 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

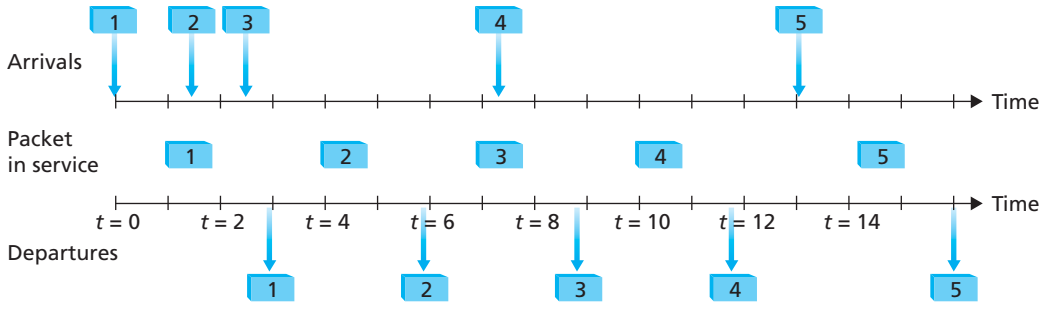


Figure 7.18 ♦ The FIFO queue in operation

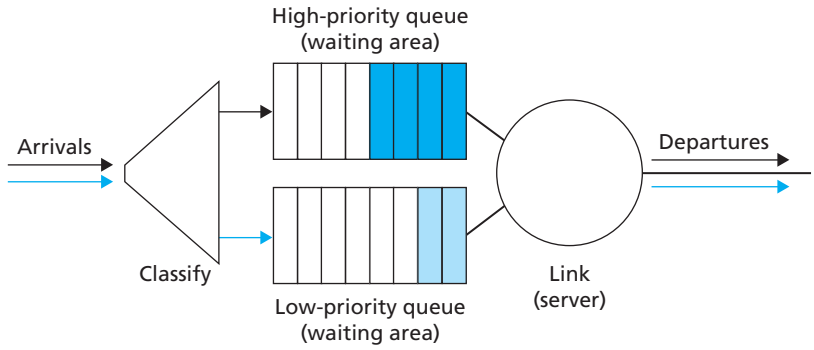


Figure 7.19 ♦ Priority queuing model

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

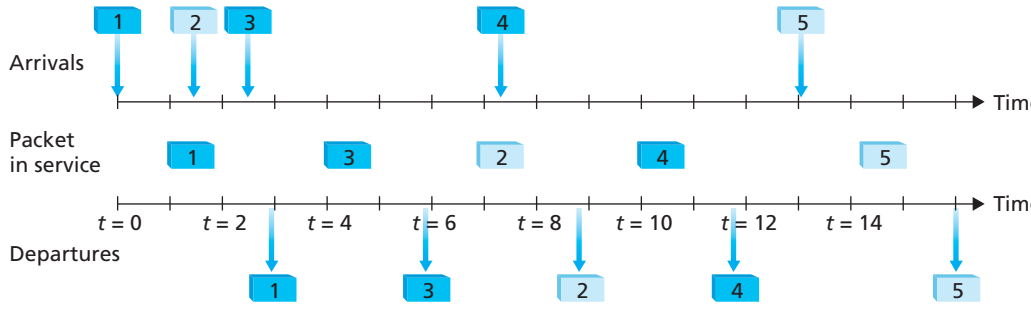


Figure 7.20 ♦ Operation of the priority queue

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

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 7.22. 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 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

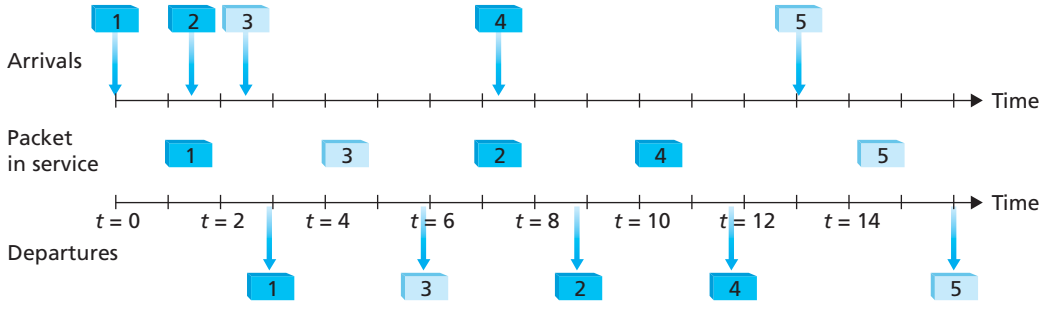


Figure 7.21 ♦ Operation of the two-class round robin queue

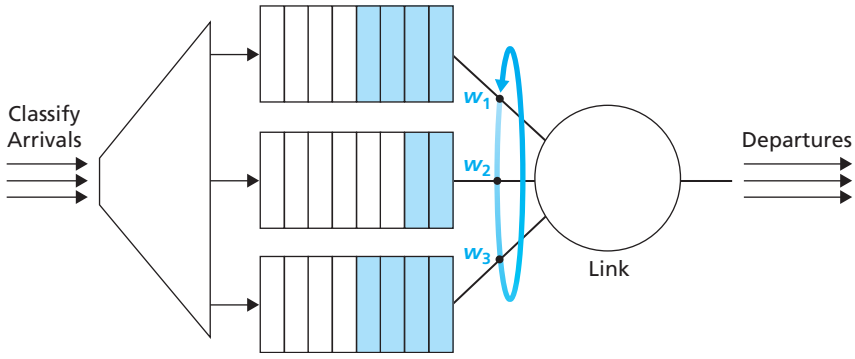


Figure 7.22 ♦ Weighted fair queuing (WFQ)

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 2012].

Policing: The Leaky Bucket

One of our earlier insights 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 7.23, 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

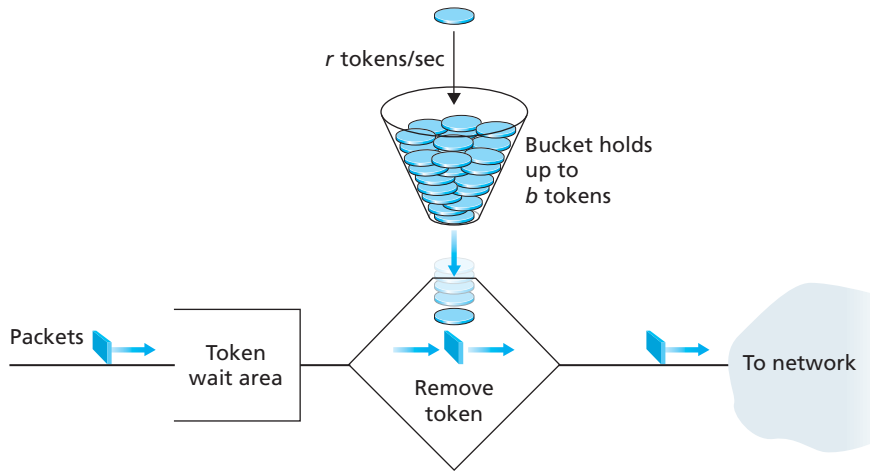


Figure 7.23 ♦ The leaky bucket policer

token from the token bucket. If the token bucket is empty, the packet must wait for a token. (An alternative is for the packet to be dropped, although we will not consider that option here.) Let us now consider how this behavior polices a traffic flow. Because there can be at most b tokens in the bucket, the maximum burst size for a leaky-bucket-policed flow is b packets. Furthermore, because the token generation rate is r, the maximum number of packets that can enter the network of *any* interval of time of length t is $rt + b$. Thus, the token-generation rate, r, serves to limit the long-term average rate at which 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

Let's close our discussion of scheduling and policing by showing how the two can be combined to provide a bound on the delay through a router's queue. Let's consider 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 7.24.

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

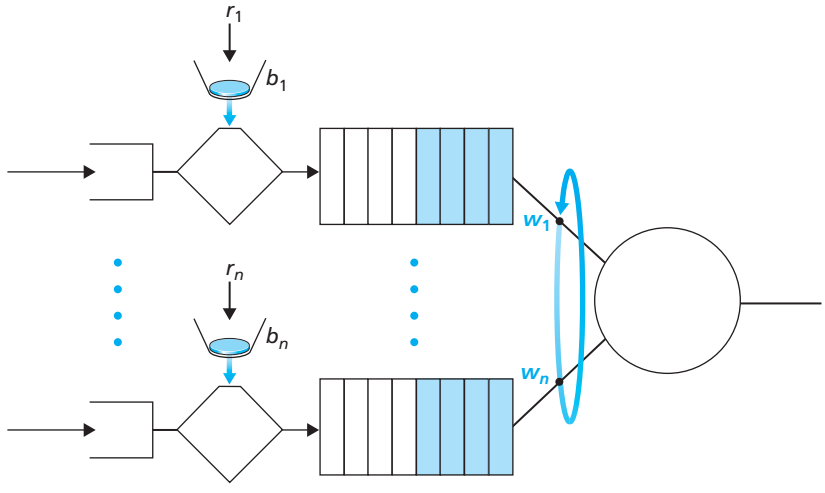


Figure 7.24 ♦ n multiplexed leaky bucket flows with WFQ scheduling

rate of the link in packets/sec. What then is the maximum delay that a packet will experience while waiting for service in the WFQ (that is, after passing through the leaky bucket)? Let us focus on flow 1. Suppose that flow 1's token bucket is initially full. A burst of b_1 packets then arrives to the leaky bucket policer for flow 1. These packets remove all of the tokens (without wait) from the leaky bucket and then join the WFQ waiting area for flow 1. Since these b_1 packets are served at a rate of at least $R \cdot w_i/(\sum w_j)$ packet/sec, the last of these packets will then have a maximum delay, $d_{\text{max,}}$ 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

Having seen the motivation, insights, and specific mechanisms for providing multiple classes of service, let's wrap up our study of approaches toward proving multiple classes of service with an example—the Internet Diffserv architecture [RFC 2475; RFC Kilkki 1999]. Diffserv provides service differentiation—that is, the ability to handle different classes of traffic in different ways within the Internet in a scalable manner. The need for scalability arises from the fact that millions of simultaneous source-destination traffic flows may be present at a backbone router. We'll 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 network's edge.

Let's begin with the simple network shown in Figure 7.25. We'll describe one possible use of Diffserv here; other variations are possible, as described in RFC 2475. 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 traffic or at the first Diffserv-capable router that the traffic passes through), arriving packets are marked. More specifically, the differentiated service (DS) field in the IPv4 or IPv6 packet header is set to some value [RFC 3260]. 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. For example, in Figure 7.25, packets being sent from H1 to H3 might be marked

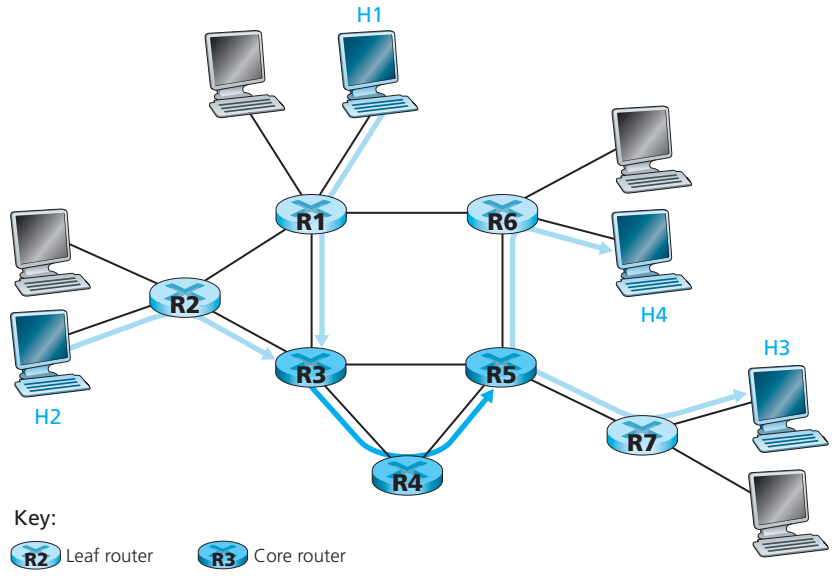


Figure 7.25 ♦ A simple Diffserv network example

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 (PHB) 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 7.25 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 Diffserv architecture obviates the need to keep router state for individual source-destination pairs—a critical consideration in making Diffserv scalable.

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.

Figure 7.26 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. As noted above, a packet's marking is carried in the DS field in the packet header.

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 7.26, 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.

So far, we have focused on the marking and policing 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:

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

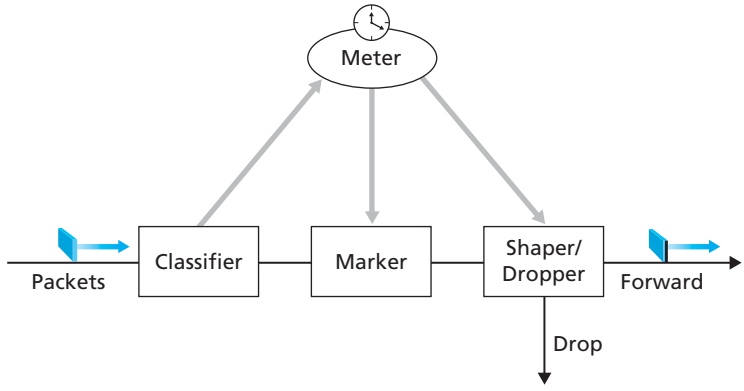


Figure 7.26 ♦ A simple Diffserv network example

- 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.

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. The **assured forwarding** PHB divides traffic into four classes, where each AF class is guaranteed to be provided with some minimum amount of bandwidth and buffering.

Let's close our discussion of Diffserv with a few observations regarding its service model. First, we have implicitly assumed that Diffserv is deployed within a single administrative domain, but typically 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 must not only provide this service, but most also cooperate and make settlements in order to offer end customers true end-to-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 the ISP that dropped and delayed your traffic. I'm sorry that there were so many gaps in your

VoIP call!" Second, 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, 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 more for premium service but finds that the best-effort service being provided to others almost always has the same performance as premium service!

7.5.4 Per-Connection Quality-of-Service (QoS) Guarantees: Resource Reservation and Call Admission

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 ongoing 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 cannot. In this section, we'll see why yet additional network mechanisms and protocols are required when a hard service guarantee is provided to individual connections.

Let's return to our scenario from Section 7.5.2 and consider two 1 Mbps audio applications transmitting their packets over the 1.5 Mbps link, as shown in Figure 7.27. The combined data rate of the two flows (2 Mbps) exceeds the link

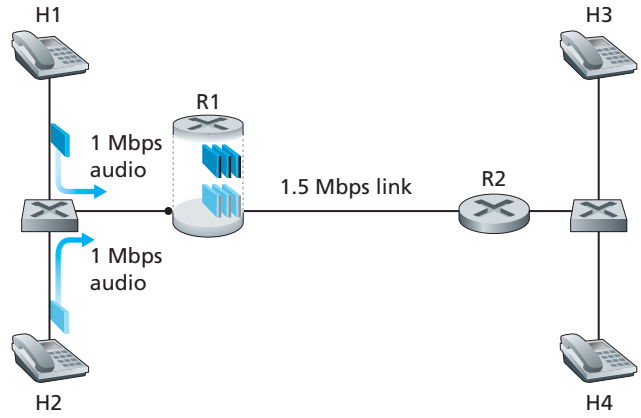


Figure 7.27 ♦ Two competing audio applications overloading the R1-to-R2 link

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 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 7.27 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 (that is, 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 in 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 7.5.2) into the mechanisms needed to provide QoS.

> **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).

Our motivating example in Figure 7.27 highlights the need for several new network mechanisms and protocols if a call (an end-to-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. Since resources are not infinite, a call making a call admission request will be denied admission, that is, 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, 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 if hard quality of service guarantees are to be provided.
- *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-to-end decision of whether or not the call has been able to reserve sufficient resources at each and every router on the end-to-end path. This is the job of the **call setup protocol**, as shown in Figure 7.28. The **RSVP protocol** [Zhang 1993, RFC 2210] was proposed for this purpose within an Internet architecture for providing quality-of-service guarantees. In ATM networks, the Q2931b protocol [Black 1995] carries this information among the ATM network's switches and end point.

Despite a tremendous amount of research and development, and even products that provide for per-connection quality of service guarantees, there has been almost no extended deployment of such services. There are many possible reasons. First and foremost, it may well be the case that the simple application-level mechanisms that we studied in Sections 7.2 through 7.4, combined with proper

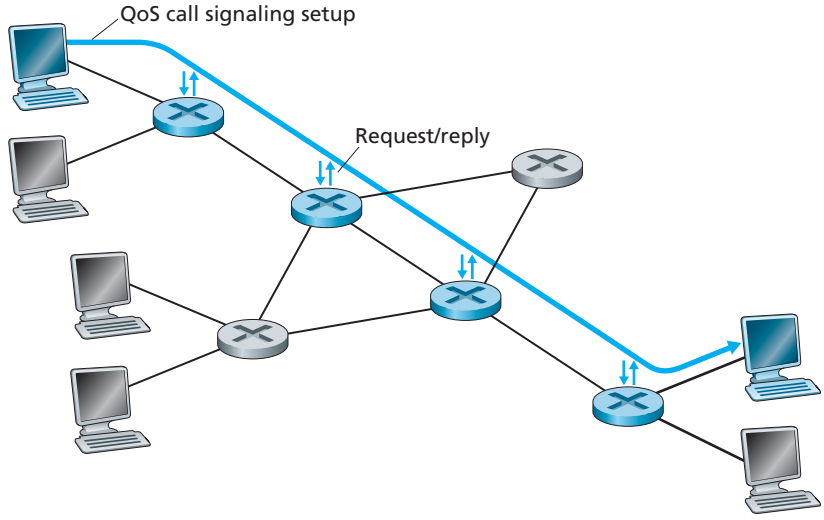


Figure 7.28 ♦ The call setup process

network dimensioning (Section 7.5.1) provide "good enough" best-effort network service for multimedia applications. In addition, the added complexity and cost of deploying and managing a network that provides per-connection quality of service guarantees may be judged by ISPs to be simply too high given predicted customer revenues for that service.

7.6 Summary

Multimedia networking is one of the most exciting developments in the Internet today. People throughout the world are spending less time in front of their radios and televisions, and are instead turning to the Internet to receive audio and video transmissions, both live and prerecorded. This trend will certainly continue as high-speed wireless Internet access becomes more and more prevalent. Moreover, with sites like YouTube, users have become producers as well as consumers of multimedia Internet content. In addition to video distribution, the Internet is also being used to transport phone calls. In fact, over the next 10 years, the Internet, along with wireless Internet access, may make the traditional circuit-switched telephone system a thing of the past. VoIP not only provides phone service inexpensively, but also provides numerous value-added services, such as video conferencing, online directory services, voice messaging, and integration into social networks such as Facebook and Google+.

In Section 7.1, we described the intrinsic characteristics of video and voice, and then classified multimedia applications into three categories: (i) streaming stored audio/video, (ii) conversational voice/video-over-IP, and (iii) streaming live audio/video.

In Section 7.2, we studied streaming stored video in some depth. For streaming video applications, prerecorded videos are placed on servers, and users send requests to these servers to view the videos on demand. We saw that streaming video systems can be classified into three categories: UDP streaming, HTTP streaming, and adaptive HTTP streaming. Although all three types of systems are used in practice, the majority of today's systems employ HTTP streaming and adaptive HTTP streaming. We observed that the most important performance measure for streaming video is average throughput. In Section 7.2 we also investigated CDNs, which help distribute massive amounts of video data to users around the world. We also surveyed the technology behind three major Internet video-streaming companies: Netflix, YouTube, and Kankan.

In Section 7.3, we examined how conversational multimedia applications, such as VoIP, can be designed to run over a best-effort network. For conversational multimedia, timing considerations are important because conversational applications are highly delay-sensitive. On the other hand, conversational multimedia applications are loss-tolerant—occasional loss only causes occasional glitches in audio/video playback, and these losses can often be partially or fully concealed. We saw how a combination of client buffers, packet sequence numbers, and timestamps can greatly alleviate the effects of network-induced jitter. We also surveyed the technology behind Skype, one of the leading voice- and video-over-IP companies. In Section 7.4, we examined two of the most important standardized protocols for VoIP, namely, RTP and SIP.

In Section 7.5, we introduced how several network mechanisms (link-level scheduling disciplines and traffic policing) can be used to provide differentiated service among several classes of traffic.

Homework Problems and Questions

Chapter 7 Review Questions

SECTION 7.1

R1. Reconstruct Table 7.1 for when Victor Video is watching a 4 Mbps video, Facebook Frank is looking at a new 100 Kbyte image every 20 seconds, and Martha Music is listening to 200 kbps audio stream.

R2. There are two types of redundancy in video. Describe them, and discuss how they can be exploited for efficient compression.

R3. Suppose an analog audio signal is sampled 16,000 times per second, and each sample is quantized into one of 1024 levels. What would be the resulting bit rate of the PCM digital audio signal?

R4. Multimedia applications can be classified into three categories. Name and describe each category.

SECTION 7.2

R5. Streaming video systems can be classified into three categories. Name and briefly describe each of these categories.

R6. List three disadvantages of UDP streaming.

R7. With HTTP streaming, are the TCP receive buffer and the client's application buffer the same thing? If not, how do they interact?

R8. Consider the simple model for HTTP streaming. Suppose the server sends bits at a constant rate of 2 Mbps and playback begins when 8 million bits have been received. What is the initial buffering delay t_p?

R9. CDNs typically adopt one of two different server placement philosophies. Name and briefly describe these two philosophies.

R10. Several cluster selection strategies were described in Section 7.2.4. Which of these strategies finds a good cluster with respect to the client's LDNS? Which of these strategies finds a good cluster with respect to the client itself?

R11. Besides network-related considerations such as delay, loss, and bandwidth performance, there are many additional important factors that go into designing a cluster selection strategy. What are they?

SECTION 7.3

R12. What is the difference between end-to-end delay and packet jitter? What are the causes of packet jitter?

R13. Why is a packet that is received after its scheduled playout time considered lost?

R14. Section 7.3 describes two FEC schemes. Briefly summarize them. Both schemes increase the transmission rate of the stream by adding overhead. Does interleaving also increase the transmission rate?

SECTION 7.4

R15. How are different RTP streams in different sessions identified by a receiver? How are different streams from within the same session identified?

R16. What is the role of a SIP registrar? How is the role of an SIP registrar different from that of a home agent in Mobile IP?

SECTION 7.5

R17. In Section 7.5, we discussed non-preemptive priority queuing. What would be preemptive priority queuing? Does preemptive priority queuing make sense for computer networks?

R18. Give an example of a scheduling discipline that is *not* work conserving.

R19. Give an example from queues you experience in your everyday life of FIFO, priority, RR, and WFQ.

Problems

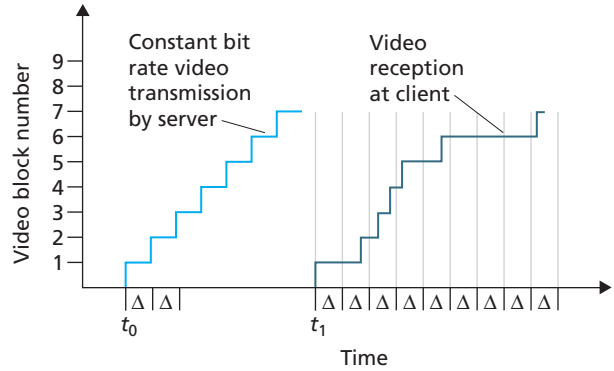
P1. Consider the figure below. Similar to our discussion of Figure 7.1, suppose that video is encoded at a fixed bit rate, and thus each video block contains video frames that are to be played out over the same fixed amount of time, $\triangle$. The server transmits the first video block at t_0, the second block at $t_0 + \triangle$, the third block at $t_0 + 2\triangle$, and so on. Once the client begins playout, each block should be played out $\triangle$ time units after the previous block.

a. Suppose that the client begins playout as soon as the first block arrives at t_1. In the figure below, how many blocks of video (including the first block) will have arrived at the client in time for their playout? Explain how you arrived at your answer.

b. Suppose that the client begins playout now at $t_1 + \triangle$. How many blocks of video (including the first block) will have arrived at the client in time for their playout? Explain how you arrived at your answer.

c. In the same scenario at (b) above, what is the largest number of blocks that is ever stored in the client buffer, awaiting playout? Explain how you arrived at your answer.

d. What is the smallest playout delay at the client, such that every video block has arrived in time for its playout? Explain how you arrived at your answer.



P2. Recall the simple model for HTTP streaming shown in Figure 7.3. Recall that B denotes the size of the client's application buffer, and Q denotes the number of bits that must be buffered before the client application begins playout. Also r denotes the video consumption rate. Assume that the server sends bits at a constant rate x whenever the client buffer is not full.

a. Suppose that $x < r$. As discussed in the text, in this case playout will alternate between periods of continuous playout and periods of freezing. Determine the length of each continuous playout and freezing period as a function of Q, r, and x.

b. Now suppose that $x > r$. At what time $t = t_f$ does the client application buffer become full?

P3. Recall the simple model for HTTP streaming shown in Figure 7.3. Suppose the buffer size is infinite but the server sends bits at variable rate $x(t)$. Specifically, suppose $x(t)$ has the following saw-tooth shape. The rate is initially zero at time $t = 0$ and linearly climbs to H at time $t = T$. It then repeats this pattern again and again, as shown in the figure below.

a. What is the server's average send rate?

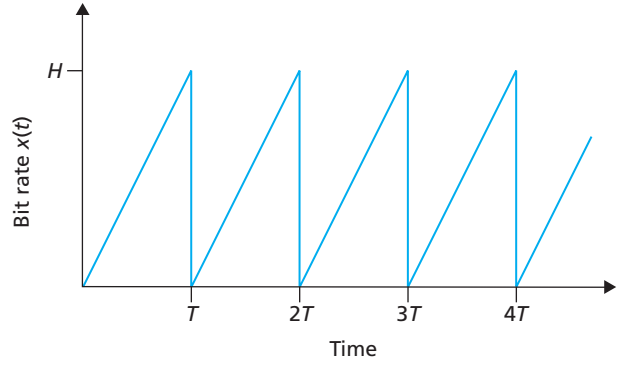
b. Suppose that $Q = 0$, so that the client starts playback as soon as it receives a video frame. What will happen?

c. Now suppose $Q > 0$. Determine as a function of Q, H, and T the time at which playback first begins.

d. Suppose $H > 2r$ and $Q = HT/2$. Prove there will be no freezing after the initial playout delay.

e. Suppose $H > 2r$. Find the smallest value of Q such that there will be no freezing after the initial playback delay.

f. Now suppose that the buffer size B is finite. Suppose $H > 2r$. As a function of Q, B, T, and H, determine the time $t = t_f$ when the client application buffer first becomes full.



P4. Recall the simple model for HTTP streaming shown in Figure 7.3. Suppose the client application buffer is infinite, the server sends at the constant rate x, and the video consumption rate is r with $r < x$. Also suppose playback begins immediately. Suppose that the user terminates the video early at time $t = E$. At the time of termination, the server stops sending bits (if it hasn't already sent all the bits in the video).

   a. Suppose the video is infinitely long. How many bits are wasted (that is, sent but not viewed)?

   b. Suppose the video is T seconds long with $T > E$. How many bits are wasted (that is, sent but not viewed)?

P5. Consider a DASH system for which there are N video versions (at N different rates and qualities) and N audio versions (at N different rates and versions). Suppose we want to allow the player to choose at any time any of the N video versions and any of the N audio versions.

   a. If we create files so that the audio is mixed in with the video, so server sends only one media stream at given time, how many files will the server need to store (each a different URL)?

   b. If the server instead sends the audio and video streams separately and has the client synchronize the streams, how many files will the server need to store?

P6. In the VoIP example in Section 7.3, let h be the total number of header bytes added to each chunk, including UDP and IP header.

   a. Assuming an IP datagram is emitted every 20 msecs, find the transmission rate in bits per second for the datagrams generated by one side of this application.

   b. What is a typical value of h when RTP is used?

P7. Consider the procedure described in Section 7.3 for estimating average delay d_i. Suppose that $u = 0.1$. Let $r_1 - t_1$ be the most recent sample delay, let $r_2 - t_2$ be the next most recent sample delay, and so on.

   a. For a given audio application suppose four packets have arrived at the receiver with sample delays $r_4 - t_4$, $r_3 - t_3$, $r_2 - t_2$, and $r_1 - t_1$. Express the estimate of delay d in terms of the four samples.

   b. Generalize your formula for n sample delays.

   c. For the formula in Part b, let n approach infinity and give the resulting formula. Comment on why this averaging procedure is called an exponential moving average.

P8. Repeat Parts a and b in Question P7 for the estimate of average delay deviation.

P9. For the VoIP example in Section 7.3, we introduced an online procedure (exponential moving average) for estimating delay. In this problem we will

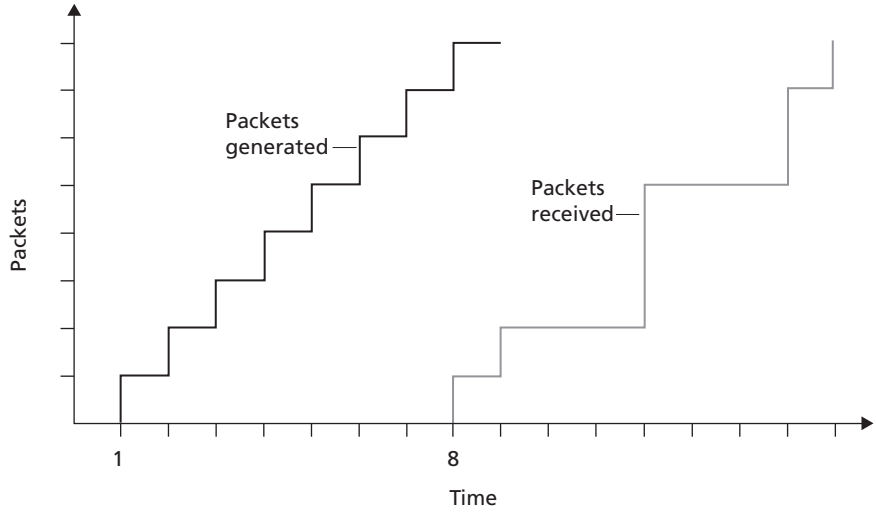
examine an alternative procedure. Let t_i be the timestamp of the ith packet received; let r_i be the time at which the ith packet is received. Let d_n be our estimate of average delay after receiving the nth packet. After the first packet is received, we set the delay estimate equal to $d_1 = r_1 - t_1$.

a. Suppose that we would like $d_n = (r_1 - t_1 + r_2 - t_2 + \ldots + r_n - t_n)/n$ for all n. Give a recursive formula for d_n in terms of d_{n-1}, r_n, and t_n.

b. Describe why for Internet telephony, the delay estimate described in Section 7.3 is more appropriate than the delay estimate outlined in Part a.

P10. Compare the procedure described in Section 7.3 for estimating average delay with the procedure in Section 3.5 for estimating round-trip time. What do the procedures have in common? How are they different?

P11. Consider the figure below (which is similar to Figure 7.7). A sender begins sending packetized audio periodically at $t = 1$. The first packet arrives at the receiver at $t = 8$.



a. What are the delays (from sender to receiver, ignoring any playout delays) of packets 2 through 8? Note that each vertical and horizontal line segment in the figure has a length of 1, 2, or 3 time units.

b. If audio playout begins as soon as the first packet arrives at the receiver at $t = 8$, which of the first eight packets sent will *not* arrive in time for playout?

c. If audio playout begins at $t = 9$, which of the first eight packets sent will not arrive in time for playout?

d. What is the minimum playout delay at the receiver that results in all of the first eight packets arriving in time for their playout?

P12. Consider again the figure in P11, showing packet audio transmission and reception times.

a. Compute the estimated delay for packets 2 through 8, using the formula for d_i from Section 7.3.2. Use a value of $u = 0.1$.

b. Compute the estimated deviation of the delay from the estimated average for packets 2 through 8, using the formula for v_i from Section 7.3.2. Use a value of $u = 0.1$.

P13. Recall the two FEC schemes for VoIP described in Section 7.3. Suppose the first scheme generates a redundant chunk for every four original chunks. Suppose the second scheme uses a low-bit rate encoding whose transmission rate is 25 percent of the transmission rate of the nominal stream.

a. How much additional bandwidth does each scheme require? How much playback delay does each scheme add?

b. How do the two schemes perform if the first packet is lost in every group of five packets? Which scheme will have better audio quality?

c. How do the two schemes perform if the first packet is lost in every group of two packets? Which scheme will have better audio quality?

P14. a. Consider an audio conference call in Skype with $N > 2$ participants. Suppose each participant generates a constant stream of rate r bps. How many bits per second will the call initiator need to send? How many bits per second will each of the other $N - 1$ participants need to send? What is the total send rate, aggregated over all participants?

b. Repeat part (a) for a Skype video conference call using a central server.

c. Repeat part (b), but now for when each peer sends a copy of its video stream to each of the $N - 1$ other peers.

P15. a. Suppose we send into the Internet two IP datagrams, each carrying a different UDP segment. The first datagram has source IP address A1, destination IP address B, source port P1, and destination port T. The second datagram has source IP address A2, destination IP address B, source port P2, and destination port T. Suppose that A1 is different from A2 and that P1 is different from P2. Assuming that both datagrams reach their final destination, will the two UDP datagrams be received by the same socket? Why or why not?

b. Suppose Alice, Bob, and Claire want to have an audio conference call using SIP and RTP. For Alice to send and receive RTP packets to and from Bob and Claire, is only one UDP socket sufficient (in addition to the socket needed for the SIP messages)? If yes, then how does Alice's SIP client distinguish between the RTP packets received from Bob and Claire?

P16. True or false:

a. If stored video is streamed directly from a Web server to a media player, then the application is using TCP as the underlying transport protocol.

b. When using RTP, it is possible for a sender to change encoding in the middle of a session.

c. All applications that use RTP must use port 87.

d. If an RTP session has a separate audio and video stream for each sender, then the audio and video streams use the same SSRC.

e. In differentiated services, while per-hop behavior defines differences in performance among classes, it does not mandate any particular mechanism for achieving these performances.

f. Suppose Alice wants to establish an SIP session with Bob. In her INVITE message she includes the line: m=audio 48753 RTP/AVP 3 (AVP 3 denotes GSM audio). Alice has therefore indicated in this message that she wishes to send GSM audio.

g. Referring to the preceding statement, Alice has indicated in her INVITE message that she will send audio to port 48753.

h. SIP messages are typically sent between SIP entities using a default SIP port number.

i. In order to maintain registration, SIP clients must periodically send REGISTER messages.

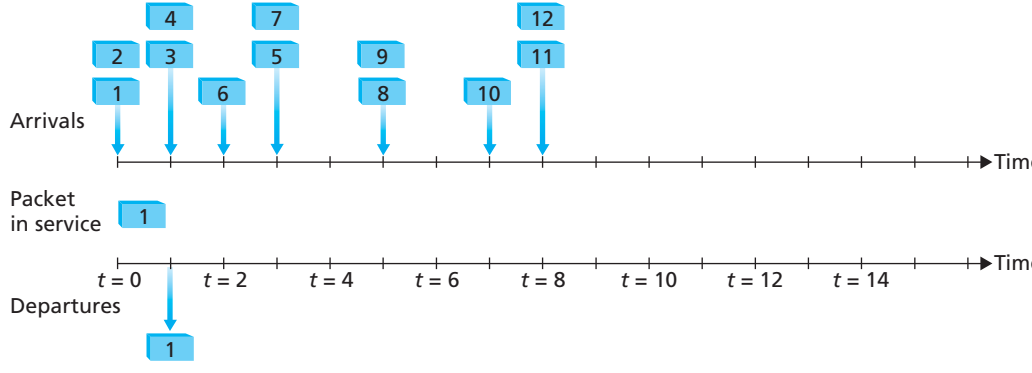
j. SIP mandates that all SIP clients support G.711 audio encoding.

P17. Suppose that the WFQ scheduling policy is applied to a buffer that supports three classes, and suppose the weights are 0.5, 0.25, and 0.25 for the three classes.

a. Suppose that each class has a large number of packets in the buffer. In what sequence might the three classes be served in order to achieve the WFQ weights? (For round robin scheduling, a natural sequence is 123123123 . . .).

b. Suppose that classes 1 and 2 have a large number of packets in the buffer, and there are no class 3 packets in the buffer. In what sequence might the three classes be served in to achieve the WFQ weights?

P18. Consider the figure below. Answer the following questions:



a. Assuming FIFO service, indicate the time at which packets 2 through 12 each leave the queue. For each packet, what is the delay between its arrival and the beginning of the slot in which it is transmitted? What is the average of this delay over all 12 packets?

b. Now assume a priority service, and assume that odd-numbered packets are high priority, and even-numbered packets are low priority. Indicate the time at which packets 2 through 12 each leave the queue. For each packet, what is the delay between its arrival and the beginning of the slot in which it is transmitted? What is the average of this delay over all 12 packets?

c. Now assume round robin service. Assume that packets 1, 2, 3, 6, 11, and 12 are from class 1, and packets 4, 5, 7, 8, 9, and 10 are from class 2. Indicate the time at which packets 2 through 12 each leave the queue. For each packet, what is the delay between its arrival and its departure? What is the average delay over all 12 packets?

d. Now assume weighted fair queueing (WFQ) service. Assume that odd-numbered packets are from class 1, and even-numbered packets are from class 2. Class 1 has a WFQ weight of 2, while class 2 has a WFQ weight of 1. Note that it may not be possible to achieve an idealized WFQ schedule as described in the text, so indicate why you have chosen the particular packet to go into service at each time slot. For each packet what is the delay between its arrival and its departure? What is the average delay over all 12 packets?

e. What do you notice about the average delay in all four cases (FIFO, RR, priority, and WFQ)?

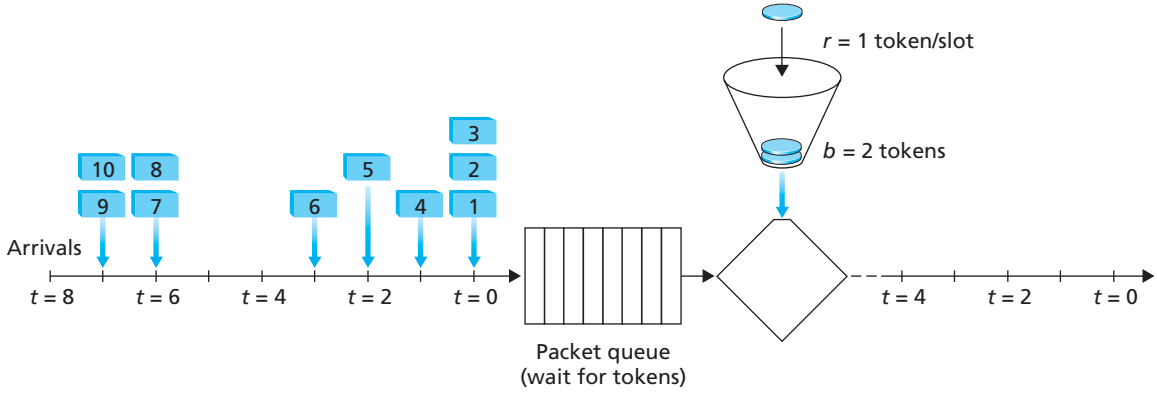
P19. Consider again the figure for P18.

a. Assume a priority service, with packets 1, 4, 5, 6, and 11 being high-priority packets. The remaining packets are low priority. Indicate the slots in which packets 2 through 12 each leave the queue.

b. Now suppose that round robin service is used, with packets 1, 4, 5, 6, and 11 belonging to one class of traffic, and the remaining packets belonging to the second class of traffic. Indicate the slots in which packets 2 through 12 each leave the queue.

c. Now suppose that WFQ service is used, with packets 1, 4, 5, 6, and 11 belonging to one class of traffic, and the remaining packets belonging to the second class of traffic. Class 1 has a WFQ weight of 1, while class 2 has a WFQ weight of 2 (note that these weights are different than in the previous question). Indicate the slots in which packets 2 through 12 each leave the queue. See also the caveat in the question above regarding WFQ service.

P20. Consider the figure below, which shows a leaky bucket policer being fed by a stream of packets. The token buffer can hold at most two tokens, and is initially full at $t = 0$. New tokens arrive at a rate of one token per slot. The output link speed is such that if two packets obtain tokens at the beginning of a time slot, they can both go to the output link in the same slot. The timing details of the system are as follows:



1. Packets (if any) arrive at the beginning of the slot. Thus in the figure, packets 1, 2, and 3 arrive in slot 0. If there are already packets in the queue, then the arriving packets join the end of the queue. Packets proceed towards the front of the queue in a FIFO manner.
2. After the arrivals have been added to the queue, if there are any queued packets, one or two of those packets (depending on the number of available tokens) will each remove a token from the token buffer and go to the output link during that slot. Thus, packets 1 and 2 each remove a token from the buffer (since there are initially two tokens) and go to the output link during slot 0.
3. A new token is added to the token buffer if it is not full, since the token generation rate is $r = 1$ token/slot.
4. Time then advances to the next time slot, and these steps repeat.

Answer the following questions:

a. For each time slot, identify the packets that are in the queue and the number of tokens in the bucket, immediately after the arrivals have been processed (step 1 above) but before any of the packets have passed through the queue and removed a token. Thus, for the $t = 0$ time slot in the example above, packets 1, 2 and 3 are in the queue, and there are two tokens in the buffer.

b. For each time slot indicate which packets appear on the output after the token(s) have been removed from the queue. Thus, for the $t = 0$ time slot in the example above, packets 1 and 2 appear on the output link from the leaky buffer during slot 0.

P21. Repeat P20 but assume that $r = 2$. Assume again that the bucket is initially full.

P22. Consider P21 and suppose now that $r = 3$, and that $b = 2$ as before. Will your answer to the question above change?

P23. Consider the leaky-bucket policer that polices the average rate and burst size of a packet flow. We now want to police the peak rate, p, as well. Show how the output of this leaky-bucket policer can be fed into a second leaky bucket policer so that the two leaky buckets in series police the average rate, peak rate, and burst size. Be sure to give the bucket size and token generation rate for the second policer.

P24. A packet flow is said to conform to a leaky-bucket specification (r,b) with burst size b and average rate r if the number of packets that arrive to the leaky bucket is less than $rt + b$ packets in every interval of time of length t for all t. Will a packet flow that conforms to a leaky-bucket specification (r,b) ever have to wait at a leaky bucket policer with parameters r and b? Justify your answer.

P25. Show that as long as $r_1 < R\, w_1/(\sum w_j)$, then $d_{\max}$ is indeed the maximum delay that any packet in flow 1 will ever experience in the WFQ queue.

Programming Assignment

In this lab, you will implement a streaming video server and client. The client will use the real-time streaming protocol (RTSP) to control the actions of the server. The server will use the real-time protocol (RTP) to packetize the video for transport over UDP. You will be given Python code that partially implements RTSP and RTP at the client and server. Your job will be to complete both the client and server code. When you are finished, you will have created a client-server application that does the following:

- The client sends SETUP, PLAY, PAUSE, and TEARDOWN RTSP commands, and the server responds to the commands.
- When the server is in the playing state, it periodically grabs a stored JPEG frame, packetizes the frame with RTP, and sends the RTP packet into a UDP socket.
- The client receives the RTP packets, removes the JPEG frames, decompresses the frames, and renders the frames on the client's monitor.

The code you will be given implements the RTSP protocol in the server and the RTP depacketization in the client. The code also takes care of displaying the transmitted video. You will need to implement RTSP in the client and RTP server. This programming assignment will significantly enhance the student's understanding of RTP, RTSP, and streaming video. It is highly recommended. The assignment also suggests a number of optional exercises, including implementing the RTSP DESCRIBE command at both client and server. You can find full details of the assignment, as well as an overview of the RTSP protocol, at the Web site http://www.awl.com/kurose-ross.