7.2 Streaming Stored Audio and Video

In recent years, audio/video streaming has become a popular application and a significant consumer of network bandwidth. In audio/video streaming, clients request compressed audio/video files that reside on servers. As we'll soon discuss, these servers can be ordinary Web servers or can be special streaming servers tailored for the audio/video streaming application. Upon client request, the server sends an audio/video file to the client by sending the file into a socket. Although both TCP and UDP can be used, today the majority of streaming audio/video traffic is transported by TCP. (Firewalls are often configured to block UDP traffic. Moreover, by using TCP, with its reliable delivery, the entire file gets transferred to the client without packet loss, allowing the file to re-played from a local cache in the future.) [Sripanidkulchai 2004]. Once the requested audio/video file starts to arrive, the client begins to render the file (typically) within a few seconds. Some systems also provide for user interactivity, for example, pause/resume and temporal jumps within the audio/video file. The **real-time streaming protocol (RTSP)**, discussed at the end of this section, is a public-domain protocol for providing user interactivity.

Users often request audio/video streaming through a Web client (that is, browser), but then display and control audio/video playout using a **media player**, such as Windows Media Player or a Flash player. The media player performs several functions, including the following:

- *Decompression.* Audio/video is almost always compressed to save disk storage and network bandwidth. A media player must decompress the audio/video on the fly during playout.
- *Jitter removal.* Packet jitter is the variability of source-to-destination delays of packets within the same packet stream. Since audio and video must be played out with the same timing with which it was recorded, a receiver will buffer received packets for a short period of time to remove this jitter. We'll examine this topic in detail in Section 7.3.

7.2.1 Accessing Audio and Video Through a Web Server

Stored audio/video can reside either on a Web server that delivers the audio/video to the client over HTTP, or on a dedicated audio/video streaming server using HTTP or some other protocol. In this subsection, we examine delivery of audio/video from a Web server; in the next subsection, we examine delivery from a streaming server. The delivery of streaming multimedia via HTTP has become popular because firewalls (see Chapter 8) will often allow HTTP traffic to pass through while proprietary protocols are blocked by the firewall.

Consider first the case of audio streaming. When an audio file resides on a Web server, the audio file is an ordinary object in the server's file system, just as HTML and

JPEG files are. When a user wants to hear the audio file, the user's host establishes a TCP connection with the Web server and sends an HTTP request for the object. Upon receiving a request, the Web server encapsulates the audio file in an HTTP response message and sends the response message back into the TCP connection. The case of video can be a little trickier, if the audio and video parts of the video are stored in two files. It is also possible that the audio and video are interleaved in the same file, so that only one object need be sent to the client. To keep our discussion simple, for the case of video we assume that the audio and video are contained in one file.

In many implementations of streaming audio/video over HTTP, client-side functionality is split into two parts. The browser's job is to request a **metafile** that provides information (for example, a URL and type of encoding, so that the appropriate media player can be identified) about the multimedia file that is to be streamed over HTTP. This metafile is then passed from the browser to the media player, whose job is to contact the HTTP server, which then sends the multimedia file to the media player over HTTP. These steps are illustrated in Figure 7.1:

1. The user clicks on a hyperlink for an audio/video file. The hyperlink does not point directly to the audio/video file, but instead to a metafile. The metafile contains the URL of the actual audio/video file. The HTTP response message that encapsulates the metafile includes a content-type header line that indicates the specific audio/video application.
2. The client browser examines the content-type header line of the response message, launches the associated media player, and passes the entire body of the response message (that is, the metafile) to the media player.

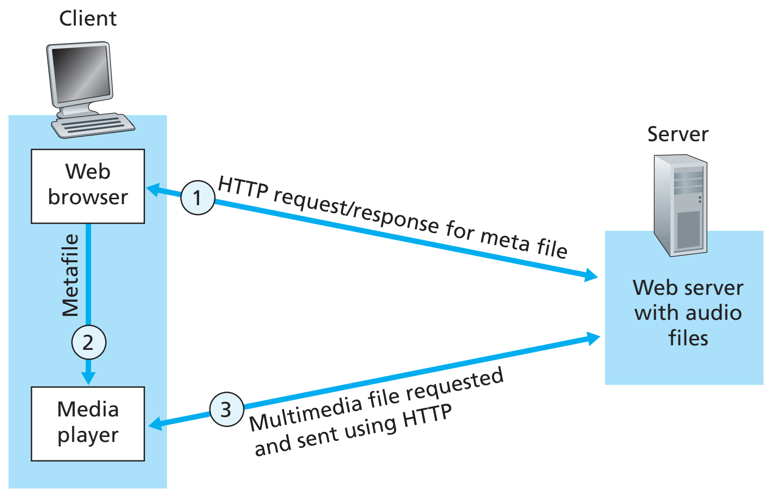
Client
Server
Web browser
1
HTTP request/response for meta file
Metafile
2
Web server with audio files
Media player
3
Multimedia file requested and sent using HTTP

Figure 7.1 ♦ Web server sends audio/video directly to the media player

3. The media player sets up a TCP connection directly with the HTTP server. The media player sends an HTTP request message for the audio/video file into the TCP connection. The audio/video file is sent within an HTTP response message to the media player. The media player streams out the audio/video file.

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

We have just learned how a metafile can allow a media player to communicate directly with a Web server that stores an audio/video file. Yet many companies that sell products for audio/video streaming do not recommend the architecture we just described. They instead recommend streaming stored audio/video from dedicated streaming servers, which have been optimized for streaming.

7.2.2 Sending Multimedia from a Streaming Server to a Helper Application

A streaming server could be a proprietary streaming server, such as those marketed by RealNetworks and Microsoft, or could be a public-domain streaming server. With a streaming server, audio/video can be sent over HTTP/TCP; or it could be sent over UDP using application-layer protocols that may be better tailored than HTTP to audio/video streaming.

This architecture requires two servers, as shown in Figure 7.2. One server, the Web server, serves Web pages (including metafiles). The second server, the **streaming server**, serves the audio/video files. The two servers can run on the same end system or on two distinct end systems. The steps for this architecture are similar to those described in the preceding subsection. However, now the media player requests the file from a streaming server rather than from a Web server, and now the media player and streaming server can interact using their own protocols. These protocols can allow for rich user interaction with the audio/video stream.

In the architecture of Figure 7.2, there are many options for delivering the audio/video from the streaming server to the media player. A partial list of the options is given below.

1. The audio/video is sent over UDP at a constant rate equal to the drain rate at the receiver (which is the encoded rate of the audio/video). For example, if the audio is compressed using GSM at a rate of 13 kbps, then the server clocks out the compressed audio file at 13 kbps. As soon as the client receives compressed audio/video from the network, it decompresses the audio/video and plays it back.
2. This is the same as the first option, but the media player delays playout for two to five seconds in order to eliminate network-induced jitter. The client accomplishes this task by placing the compressed media that it receives from the network into a **client buffer**, as shown in Figure 7.3. Once the client has

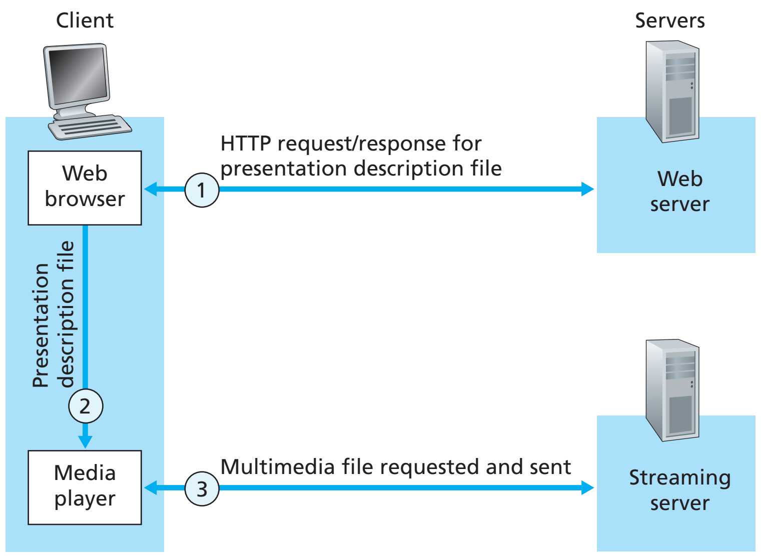
Client
Servers
Web browser
HTTP request/response for presentation description file
1
Web server
Presentation description file
2
Media player
Multimedia file requested and sent
3
Streaming server

Figure 7.2 ♦ Streaming from a streaming server to a media player

prefetched a few seconds of the media, it begins to drain the buffer. For this, and the previous option, the fill rate $x(t)$ is equal to the drain rate d, except when there is packet loss, in which case $x(t)$ is momentarily less than d.

3. The media is sent over TCP. The server pushes the media file into the TCP socket as quickly as it can; the client (that is, media player) reads from the TCP socket as quickly as it can and places the compressed video into the media player buffer. After an initial two- to five-second delay, the media player reads from its buffer at

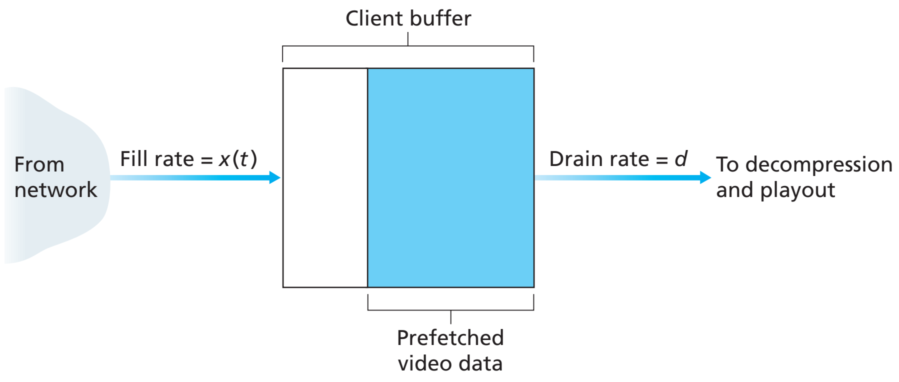

Figure 7.3 ♦ Client buffer being filled at rate $x(t)$ and drained at rate d

a rate d and forwards the compressed media to decompression and playback. Because TCP retransmits lost packets, it has the potential to provide better sound quality than UDP. On the other hand, the fill rate $x(t)$ now fluctuates with time due to TCP congestion control and window flow control. In fact, after packet loss, TCP congestion control may reduce the instantaneous rate to less than d for long periods of time. This can empty the client buffer (a process known as **starvation**) and introduce undesirable pauses into the output of the audio/video stream at the client. [Wang 2004] shows that when the average TCP throughput is roughly twice the media bit rate, TCP streaming results in minimal starvation and low startup delays.

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

7.2.3 Real-Time Streaming Protocol (RTSP)

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

Before getting into the details of RTSP, let us first indicate what RTSP does not do.

- RTSP does not define compression schemes for audio and video.
- RTSP does not define how audio and video are encapsulated in packets for transmission over a network; encapsulation for streaming media can be provided by RTP or by a proprietary protocol. (RTP is discussed in Section 7.4.) For example, RealNetworks' audio/video servers and players use RTSP to send control information to each other, but the media stream itself can be encapsulated in RTP packets or in some proprietary data format.
- RTSP does not restrict how streamed media is transported; it can be transported over UDP or TCP.
- RTSP does not restrict how the media player buffers the audio/video. The audio/video can be played out as soon as it begins to arrive at the client, it can be

played out after a delay of a few seconds, or it can be downloaded in its entirety before playout.

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

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

Let's now walk through a simple RTSP example, which is illustrated in Figure 7.4. The Web browser first requests a presentation description file from a Web server. The presentation description file can have references to several continuous-media files as well as directives for synchronization of the continuous-media files. Each reference to a continuous-media file begins with the URL method, `rtsp://`. Below we provide a sample presentation file that has been adapted from [Schulzrinne 1997]. In this presentation, an audio and video stream are played in parallel and in lip sync (as part of the same group). For the audio stream, the media player can choose (switch) between two audio recordings, a low-fidelity recording and a high-fidelity recording. (The format of the file is similar to SMIL [SMIL 2009], which is used by many streaming products to define synchronized multimedia presentations.)

The Web server encapsulates the presentation description file in an HTTP response message and sends the message to the browser. When the browser receives the HTTP response message, the browser invokes a media player (that is, the helper application) based on the content-type field of the message. The presentation description file includes references to media streams, using the URL method `rtsp://`, as in the sample above. As shown in Figure 7.4, the player and the server then send each other a series of RTSP messages. The player sends an RTSP SETUP request, and the server responds with an RTSP OK message. The player sends an RTSP PLAY request, say, for low-fidelity audio, and the server responds with an RTSP OK message. At this point, the streaming server pumps the low-fidelity audio into its own in-band channel. Later, the media player sends an RTSP PAUSE request, and the server responds with an RTSP OK message. When the user is finished, the media player sends an RTSP TEARDOWN request, and the server confirms with an RTSP OK response.

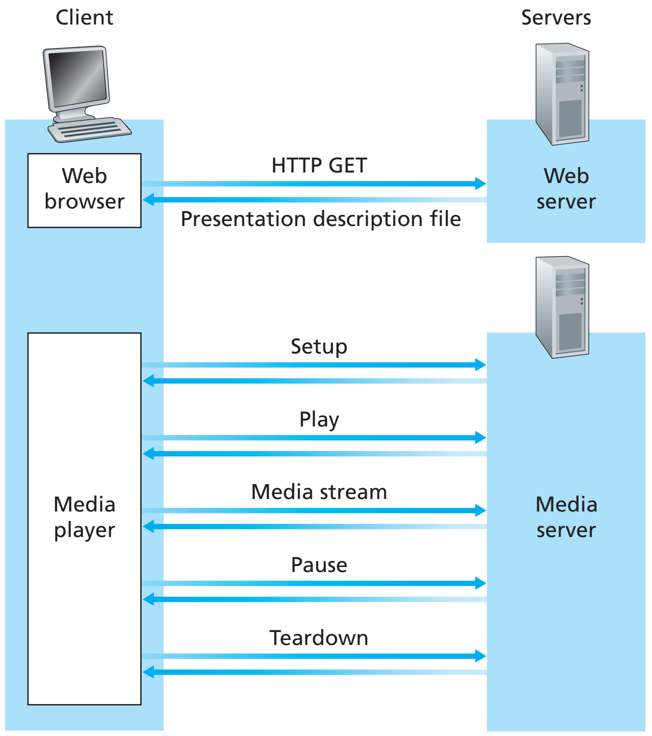

Figure 7.4 ♦ Interaction between client and server using RTSP.

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

Now let's take a brief look at the actual RTSP messages. The following is a simplified example of an RTSP session between a client (C:) and a sender (S:).

```
C: SETUP rtsp://audio.example.com/twister/audio RTSP/1.0
   Cseq: 1
   Transport: rtp/udp; compression; port=3056; mode=PLAY
S: RTSP/1.0 200 OK
   Cseq: 1
   Session: 4231
C: PLAY rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
   Range: npt=0-
   Cseq: 2
   Session: 4231
S: RTSP/1.0 200 OK
   Cseq: 2
   Session: 4231
C: PAUSE rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
   Range: npt=37
   Cseq: 3
   Session: 4231
S: RTSP/1.0 200 OK
   Cseq: 3
   Session: 4231
C: TEARDOWN rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
   Cseq: 4
   Session: 4231
S: RTSP/1.0 200 OK
   Cseq: 4
   Session: 4231
```

It is interesting to note the similarities between HTTP and RTSP. All request and response messages are in ASCII text, the client employs standardized methods (SETUP, PLAY, PAUSE, and so on), and the server responds with standardized reply codes. One important difference, however, is that the RTSP server keeps track of the state of the client for each ongoing RTSP session. For example, the server keeps track of whether the client is in an initialization state, a play state, or a pause state (see the programming assignment for this chapter). The session and sequence numbers, which are part of each RTSP request and response, help the server keep track of the session state. The session number is fixed throughout the entire session; the client increments the sequence number each time it sends a new message; the server echoes back the session number and the current sequence number.

As shown in the example, the client initiates the session with the SETUP request, providing the URL of the file to be streamed and the RTSP version. The setup message includes the client port number to which the media should be sent. The setup message also indicates that the media should be sent over UDP using the RTP packetization protocol (to be discussed in Section 7.4). Notice that in this example, the player chose not to play back the complete presentation, but instead only the low-fidelity portion of the presentation.

RTSP is actually capable of doing much more than described in this brief introduction. In particular, RTSP has facilities that allow clients to stream toward the server (for example, for recording). RTSP has been adopted by RealNetworks, one of the industry leaders in audio/video streaming. Henning Schulzrinne makes available a Web page on RTSP [Schulzrinne-RTSP 2009].

At the end of this chapter, you will find a programming assignment for creating a video-streaming system (both server and client) that leverages RTSP. This assignment involves writing code that actually constructs and sends RTSP messages at the client. The assignment provides the RTSP server code, which parses the RTSP messages and constructs appropriate responses. Readers interested in obtaining a deeper understanding of RTSP are highly encouraged to work through this interesting assignment.

7.3 Making the Best of the Best-Effort Service

The Internet's network-layer protocol, IP, provides a best-effort service. That is to say that the service makes its best effort to move each datagram from source to destination as quickly as possible. However, it does not make any promises whatsoever about the extent of the end-to-end delay for an individual packet, or about the extent of packet jitter and packet loss within the packet stream. The lack of guarantees about delay and packet jitter poses significant challenges to the design of real-time multimedia applications such as Internet phone and real-time video conferencing, which are acutely sensitive to packet delay, jitter, and loss.

In this section, we'll cover several ways in which the performance of multimedia applications over a best-effort network can be enhanced. Our focus will be on application-layer techniques, i.e., approaches that do not require any changes in the network core or even in the transport layer at the end hosts. We first describe the effects of packet loss, delay, and delay jitter on multimedia applications. We then cover techniques for recovering from such impairments. We then describe how content distribution networks and resource over-provisioning can be used to avoid such impairments in the first place.

7.3.1 The Limitations of a Best-Effort Service

We mentioned above that best-effort service can lead to packet loss, excessive end-to-end delay, and packet jitter. Let's examine these issues in more detail. To keep the discussion concrete, we discuss these mechanisms in the context of an **Internet**