6.2 Streaming Stored Audio and Video

In recent years, audio/video streaming has become a popular class of applications and a major consumer of network bandwidth. We expect this trend to continue for several reasons. First, the cost of disk storage is decreasing at phenomenal rates, even faster than processing and bandwidth costs; the cheap storage will lead to an exponential increase in the amount of stored/audio video in the Internet. Second, improvements in Internet infrastructure, such as high-speed residential access (i.e., cable modems and ADSL as discussed in Chapter 1) network caching of video (see Section 2.2), and a new QoS-oriented Internet protocols (see Sections 6.5-6.9) will greatly facilitate the distribution of stored audio and video. And third, there is an enormous pent-up demand for high quality video streaming, an application which combines two existing killer communication technologies -- television and the on-demand Web.

In audio/video streaming, clients request compressed audio/video files, which are resident on servers. As we shall discuss in this section, the servers can be "ordinary" Web servers, or can be special streaming servers tailored for the audio/video streaming application. The files on the servers can contain any type of audio/video content, including a professor's lectures, rock songs, movies, television shows, recorded sporting events, etc. Upon client request, the server directs an audio/video file to the client by sending the file into a socket. (Sockets are discussed in Sections 2.6-2.7.) Both TCP and UDP socket connections are used in practice. Before sending the audio/video file into the network, the file may is segmented, and the segments are typically encapsulated with special headers appropriate for audio/video traffic. **Real-Time Protocol (RTP)**, discussed in Section 6.4, is a public-domain standard for encapsulating the segments. Once the client begins to receive the requested audio/video file, the client begins to render the file (typically) within a few seconds. Most of the existing products also provide for user interactivity, e.g., pause/resume and temporal jumps to the future and past of the audio/video file. User interactivity also requires a protocol for client/server interaction. **Real Time Streaming Protocol (RTSP)**, discussed at the end of this section, is a public-domain protocol for providing user interactivity.

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

- **Decompresssion**: Audio/video is almost always compressed to save disk storage and network bandwidth. A media player has to decompress the audio/video on the fly during play out.
- **Jitter-removal**: Packet jitter is the variability of packet delays within the same packet stream. Packet jitter, if not suppressed, can easily lead to unintelligible audio and video. As we shall examine in some detail in Section 6.3, packet jitter can often be limited by buffering audio/video for a few seconds at the client before playback.
- **Error correction:** Due to unpredictable congestion in the Internet, a fraction of packets in the packet stream can be lost. If this fraction becomes too large, user perceived audio/video quality becomes unacceptable. To this end, many streaming systems attempt to recover from losses by either (i) reconstructing loss packets through the transmission of redundant packets, (ii) by having the client explicitly request retransmissions of lost packets, (iii) or both.
- **Graphical user interface with control knobs:** This is the actual interface that the user interacts with. It typically includes volume controls, pause/resume buttons, sliders for making temporal

jumps in the audio/video stream, etc.

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

6.2.1 Acessing Audio and Video from a Web Server

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

Consider first the case of audio streaming. When an audio file resides on a Web server, the audio file is an ordinary object in the server's file system, just as are HTML and JPEG files. When a user wants to hear the audio file, its host establishes a TCP connection with the Web server and sends an HTTP request for the object (see Section 2.2); upon receiving such a request, the Web server bundles the audio file in an HTTP response message and sends the response message back into the TCP connection. The case of video can be a little more tricky, only because the audio and video parts of the "video" may be stored in two different files, that is, they may be two different objects in the Web server's file system. In this case, two separate HTTP requests are sent to the server (over two separate TCP connections for HTTP/1.0), and the audio and video files arrive at the client in parallel. It is up to the client to manage the synchronization of the two streams. It is also possible that the audio and video are interleaved in the same file, so that only one object has to be sent to the client. To keep the discussion simple, for the case of "video" we assume that the audio and video is contained in one file for the remainder of this section.

A naive architecture for audio/video streaming is shown in Figure 6.2.1. In this architecture:

1. The browser process establishes a TCP connection with the Web server and requests the audio/video file with an HTTP request message.
2. The Web server sends to the browser the audio/video file in an HTTP response message.
3. The content-type: header line in the HTTP response message indicates a specific audio/video encoding. The client browser examines the content-type of the response message, launches the associated media player, and passes the file to the media player.
4. The media player then renders the audio/video file.

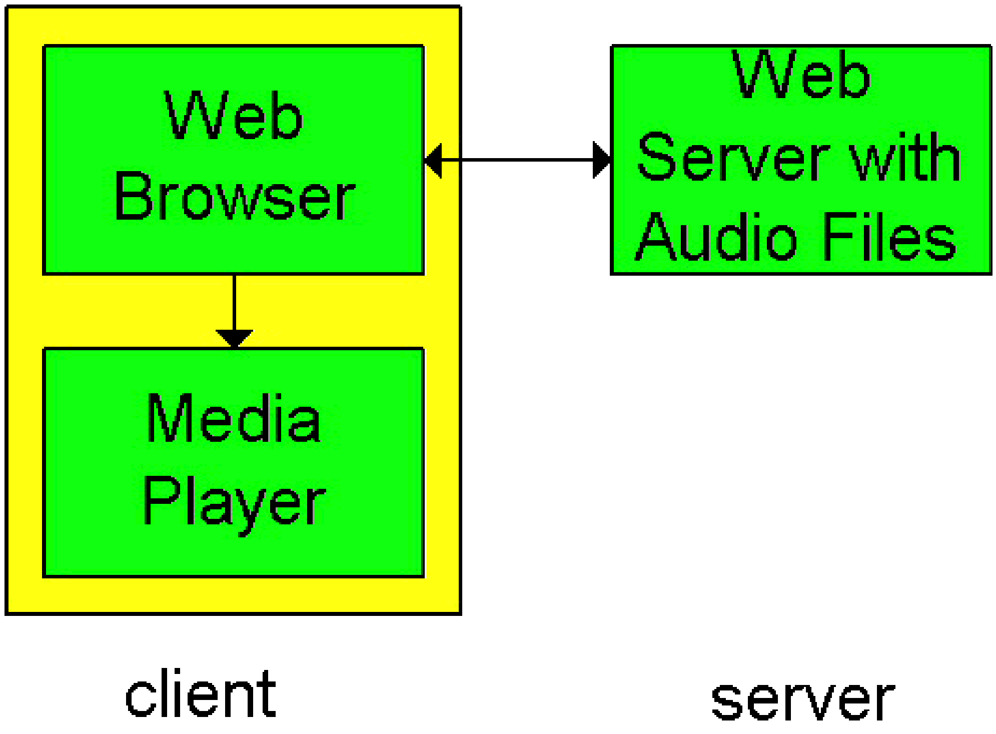

Figure 6.2-1 A naive implementation for audio streaming.

Although this approach is very simple, it has a major drawback: the media player (i.e., the helper application) must interact with the server through the intermediary of a Web browser. This can lead to many problems. In particular, when the browser is an intermediary, the entire object must be downloaded before the browser passes the object to a helper application; the resulting initial delay is typically unacceptable for audio/video clips of moderate length. For this reason, audio/video streaming implementations typically have the server send the audio/video file directly to the media player process. In other words, a direct socket connection is made between the server process and the media player process. As shown in Figure 6.2-2, this is typically done by making use of a **meta file**, which is a file that provides information (e.g., URL, type of encoding, etc.) about the audio/video file that is to be streamed.

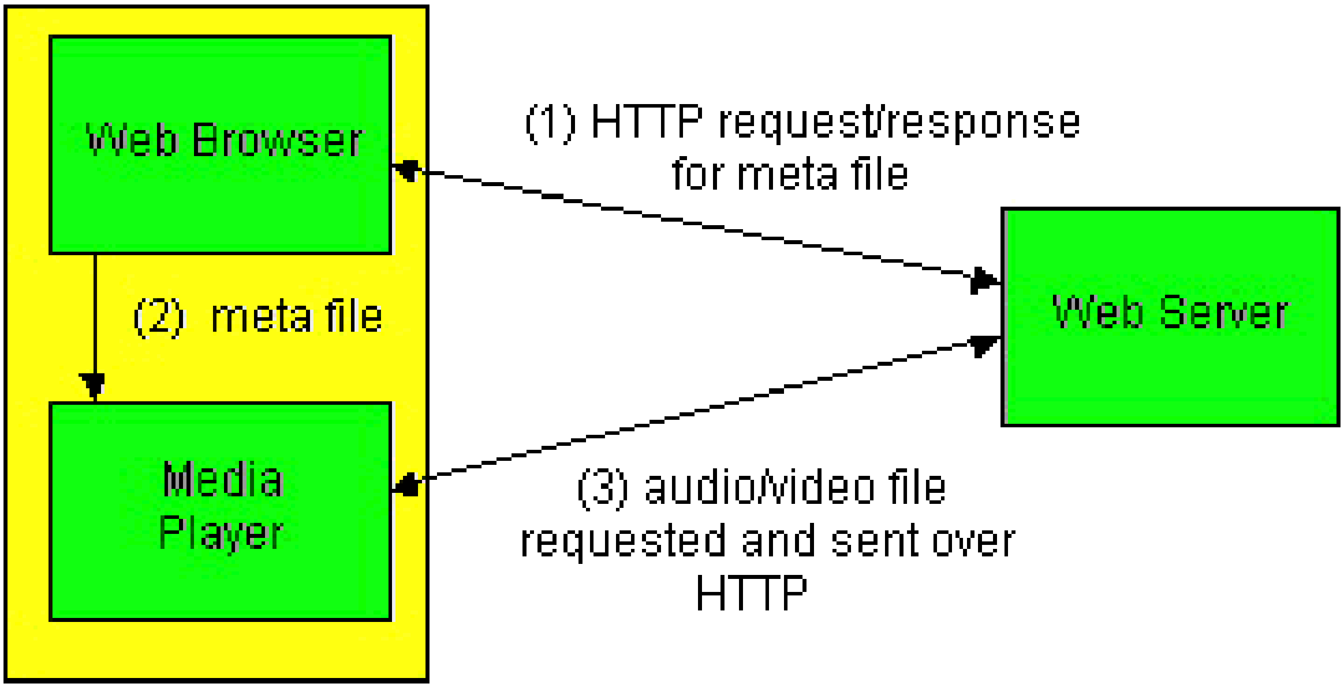

Figure 6.2-2 Web server sends audio/video directly to the media player.

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

1. The user clicks on a hyperlink for an audio/video file.
2. The hyperlink does not point directly to the audio/video file, but instead to a meta file. The meta file contains the the URL of the actual audio/video file. The HTTP response message that encapsulates the meta file includes a content-type: header line that indicates the specific audio/video application.
3. The client browser examines the content-type header line of the response message, launches the associated media player, and passes the entity body of the response message (i.e., the meta file) to the media player.
4. The media player sets up a TCP connection directly with the HTTP server. The media player sends an HTTP request message for the audio/video file into the TCP connection.
5. The audio/video file is sent within an HTTP response message to the media player. The media player streams out the audio/video file.

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

We have have just learned how a meta file can allow a media player to dialogue directly with a Web server housing an audio/video. Yet many companies that sell products for audio/video streaming do not recommend the architecture we just described. This is because the architecture has the media player communicate with the server over HTTP and hence also over TCP. HTTP is often considered insufficiently rich to allow for satisfactory user interaction with the server; in particular, HTTP does not easily allow a user (through the media server) to send pause/resume, fast-forward and temporal jump commands to the server. TCP is often considered inappropriate for audio/video streaming, particularly when users are behind slow modem links. This is because, upon packet loss, the TCP sender rate almost comes to a halt, which can result in extended periods of time during which the media player is starved. Nevertheless, audio and video is often streamed from Web servers over TCP with satisfactory results.

6.2.2 Sending Multimedia from a Streaming Server to Helper Application

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

This architecture requires two servers, as shown in Figure 6.2-3. One server, the HTTP server, serves Web pages (including meta files). The second server, the **streaming server,** serves the audio/video files. The two servers can run on the same end system or on two distinct end systems. (If the Web server is very busy serving Web pages, it may be advantageous to put the streaming server on its own machine.) The steps for this architecture are similar to those described in the previous architecture. However, now the media player requests the file from a streaming server rather than from a Web server, and now the media player and streaming server can interact using their own protocols. These protocols can allow for rich user interaction with the audio/video stream. Furthermore, the audio/video file can be sent to the media player over UDP instead of TCP.

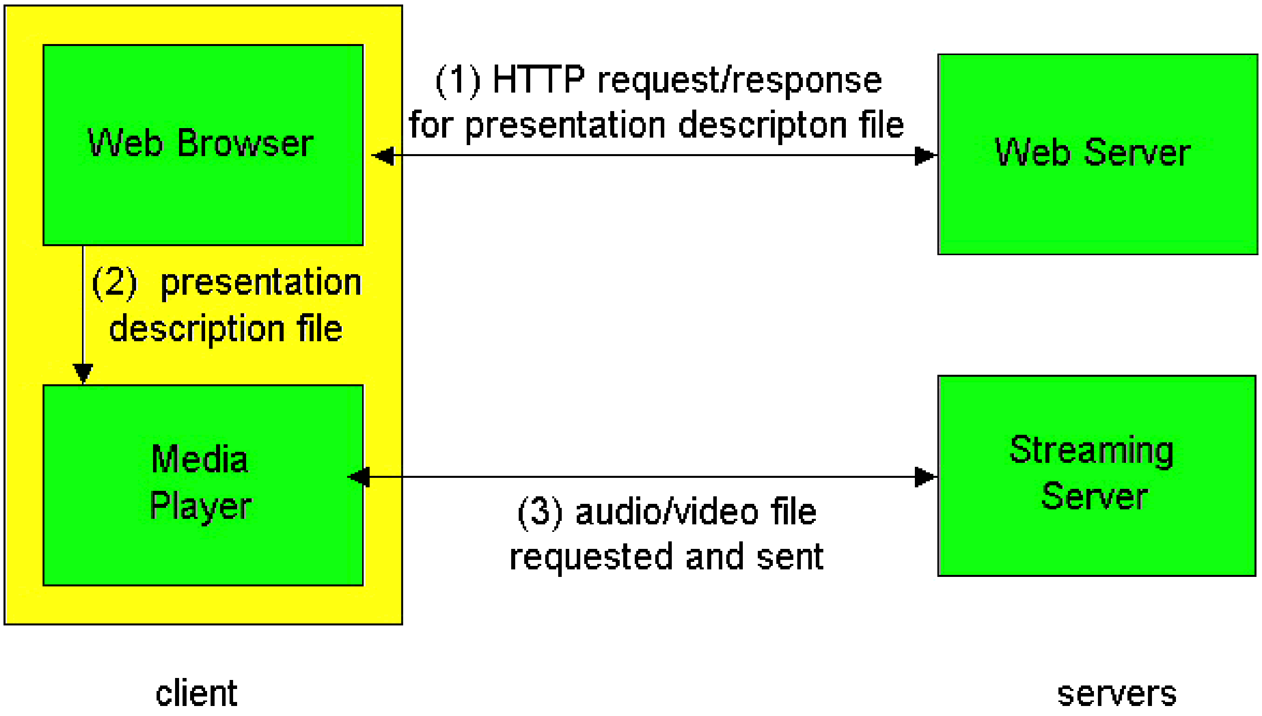

Figure 6.2-3 Streaming from a streaming server to a media player.

In the architecture of Figure 6.2-3, there are many options for delivering the audio/video from the streaming server to the media player. A partial list of the options is given below:

1. The audio/video is sent over UDP at a constant rate equal to the drain rate at the reciever (which is the encoded rate of the audio/video). For example, if the audio is compressed using GSM at a rate of 13 Kbps, then the server clocks out the compressed audio file at 13 Kbps. As soon as the client receives compressed audio/video from the network, it decompresses the audio/video and plays it back.
2. This is the same as option 1, but the media player delays play out for 2-5 seconds in order to eliminate network induced jitter. The client accomplishes this task by placing the compressed media that it receives from the network into a **client buffer**, as shown in Figure 6.2-4. Once the client has "prefetched" a few seconds of the media, it begins to drain the buffer. For this and the previous option, the drain rate d is equal to the fill rate $x(t)$, except when there is packet loss, in which case $x(t)$ is less momentarily less than d.
3. The audio is sent over TCP and the media player delays play out for 2-5 seconds. The server passes data to the TCP socket at a constant rate equal to the receiver drain rate d. TCP retransmits lost packets, and thereby possibly improves sound quality. But the fill rate $x(t)$ now fluctuates with time due to TCP slow start and window flow control, even when there is no packet loss. If there is no packet loss, the average fill rate should be approximately equal to the drain rate d. Furthermore, after packet loss TCP congestion control may reduce the instantaneous rate to less than d for long periods of time. This can can empty the client buffer and introduce undesirable pauses into the output of the audio/video stream at the client.
4. This is the same as option 3, but now the media player uses a large client buffer - large enough to hold the much if not all of the audio/video file (possibly within disk storage). The server pushes the audio/video file into its TCP socket as quickly as it can; the client reads from its TCP socket as quickly as it can, and places the decompressed audio video into the large client buffer. In this case, TCP makes use of all the instantaneous bandwidth available to the connection, so that at times $x(t)$ can be much larger than d. When the instantaneous bandwidth drops below the drain rate, the receiver does not experience loss as long as the client buffer is nonempty.

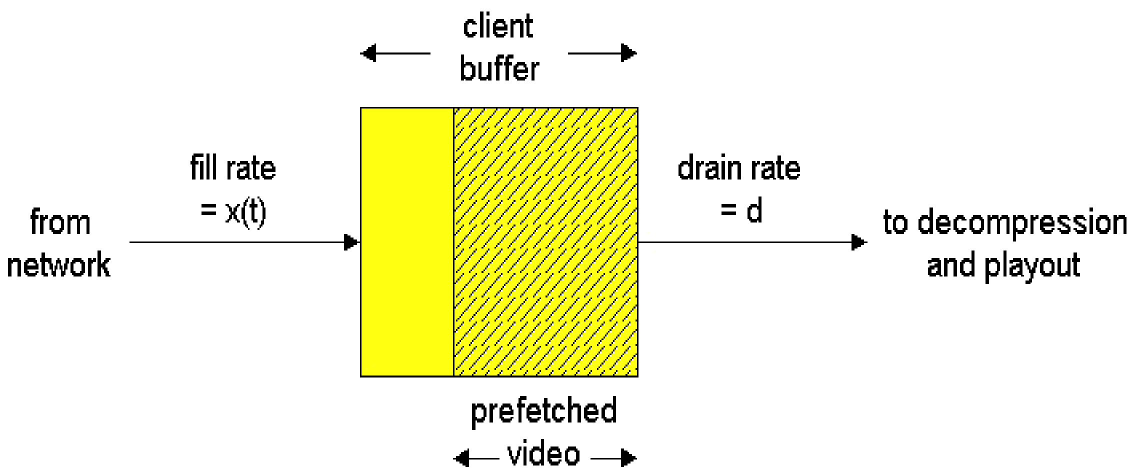

Figure 6.2-4 Client buffer being filled at rate $x(t)$ and drained at rate d.

6.2.3 Real Time Streaming Protocol (RTSP)

Audio, video and SMIL presentations, etc., are often referred to as continuous media. (SMIL stands for Synchronized Multimedia Integration Language; it is a document language standard, as is HTML. As its name suggests, SMIL defines how continuous media objects, as well as static objects, are synchronized in a presentation that unravels over time. An indepth discussion of SMIL is beyond the scope of this book.) Users typically want to *control* the playback of continous media by pausing playback, repositioning playback to a future or past point of time, visual fast-forwarding playback, visual rewinding playback, etc. This functionality is similar to what to a user has with a VCR when watching a video cassette or with a CD player when listening to CD music. To allow a user to control playback, the media player and server need a protocol for exchanging playback control information. RTSP, defined in [RFC 2326], is such a protocol.

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

- RTSP does not define compression schemes for audio and video.
- RTSP does not define the how audio and video is encapusalated in packets for transmission over a network; encapsulation for streaming media can be provided by RTP or by a proprietary protocol. (RTP is discussed in Section 6.4) For example, RealMedia?s G2 server and player use RTSP to send control information to each other. But the media stream itself can be encapsulated RTP packets or with some proprietary RealNetworks scheme.
- RTSP does not restrict how the the 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 play out.

So if RTSP doesn't do any of the above, what does RTSP do? RTSP is a protocol that allows a media player to control the transmission of a media stream. As mentioned above, control actions inlcude pause/resume, repositioning of playback, fast forward and rewind. RTSP is a so-called **out-of-band protocol**. In particular, the RTSP messages are sent out-of-band, whereas the media stream, whose packet structure is not defined by RTSP, is considered ?in-band?. The RTSP messages use different port numbers than the media stream. RTSP uses port number 554. (If the RTSP messages were to use the same port numbers as the media stream, then RTSP messages would be said to be ?interleaved? with the media stream.) The RTSP specification [RFC 2326] permits RTSP messages to be sent either over TCP or UDP.

Recall from Section 2.3 that File Transfer Protocol (FTP) also uses the out-of-band notion. In particular, FTP uses two client/server pairs of sockets, each pair with its own port number: one client/server socket pair supports a TCP connection that transports control information; the other client/server socket pair supports a TCP connection that actually transports the file. The control TCP connection is the so-called out-of-band channel whereas the TCP connection that transports the file is the so-called data channel. The out-of-band channel is used for sending remote directory changes, remote file deletion, remote file renaming, file download requests, etc. The inband channel transports the file itself. The RTSP channel is in many ways similar to FTP's control channel.

Let us now walk through a simple RTSP example, which is illustrated in Figure 6.2-5. The Web browser first requests a presentation description file from a Web server. The presentation description file can have references to several continous-media files as well as directives for syncrhonization of the continuous-media files. Each reference to a continuous-media file begins with the the URL method, rtsp:// . Below we provide a sample presentation file, which has been adapted from the paper [Schulzrinne]. In this presentation, an audio and video stream are played in parallel and in lipsync (as part of the same "group"). For the audio stream, the media player can choose ("switch") among two audio recordings, a low fidelity recording and a hi fidelity recording.

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

The Web server encapsulates the presentation description file in an HTTP response message and sends the message to the browser. When the browser receives the HTTP response message, the browser invokes a media player (i.e., the helper application) based on the content-type: field of the message. The presentation description file includes references to media streams, using the URL method rtsp:// , as shown in the above sample. As shown in Figure 6.2-4, the player and the server then send each other a series of RTSP messages. The player sends an RTSP SETUP request, and the server sends an RTSP SETUP response. The player sends an RTSP PLAY request, say, for lofi auido, and server sends RTSP PLAY response. At this point, the streaming server pumps the lofi audio into its own in-band channel. Later, the media player sends an RTSP PAUSE request, and the server responds with an RTSP PAUSE response. When the user is finished, the media player sends an RTSP TEARDOWN request, and the server responds with an RTSP TEARDOWN response.

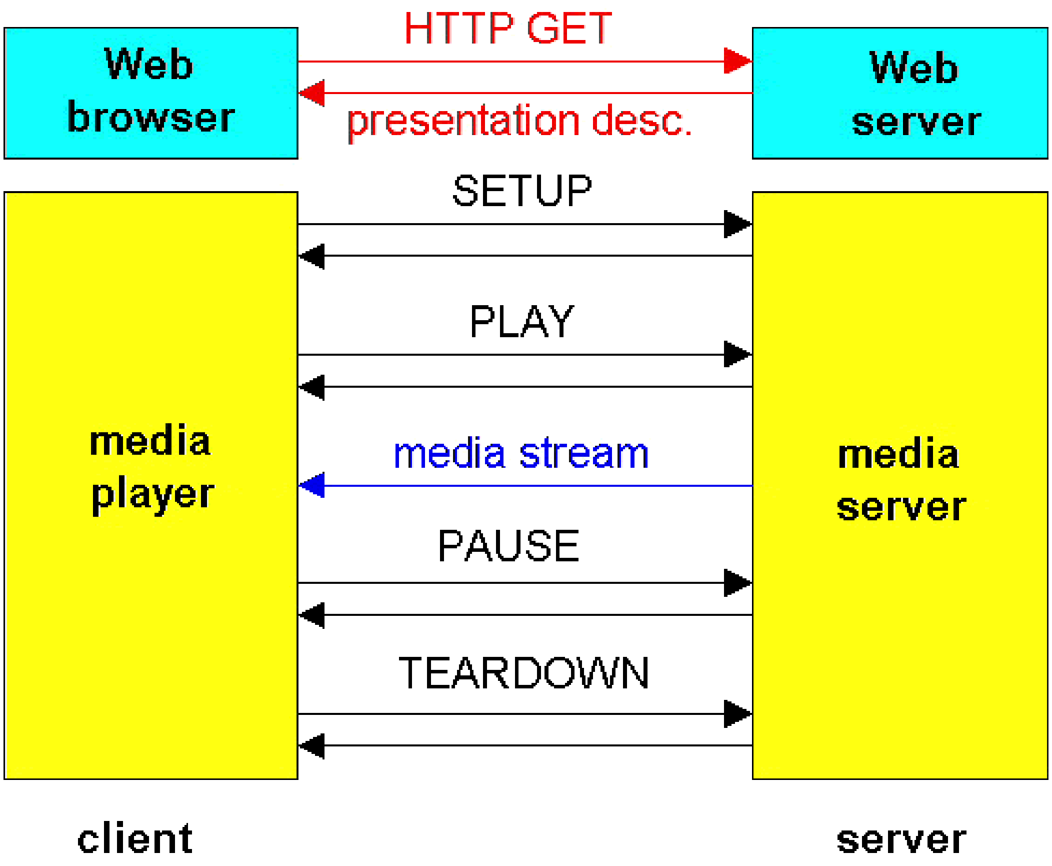

Figure 6.2-4 Interaction between client and server using RTSP

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

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

S: RTSP/1.0 200 1 OK
Session 4231

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

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

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

S: 200 3 OK

Notice that in this example, the player choose not to playback the complete present, but instead only hte lofi portion of the presentation. The RTSP protocol is actually capable of doing much more than described in this brief introduction. In particular, RTSP has facilities that allows clients to stream towards the server (e.g., for recording). RTSP has been adapted by RealNetworks, currently the industry leader in audio/video streaming. RealNetworks makes available a nice page on RTSP [RealNetworks].

References

[Schulzrinne] H. Schulzrinne, "A Comprehensive Multimedia Control Architectuure for the Internet," NOSSDAV'97 (Network and Operating System Support for Digital Audio and Video), St. Louis, Missouri; May 19, 1997. Online version available.

[RealNetworks] RTSP Resource Center, http://www.real.com/devzone/library/fireprot/rtsp/

[RFC 2326] H. Schulzrinne, A. Rao, R. Lanphier, "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998.

Return to Table of Contents
