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,

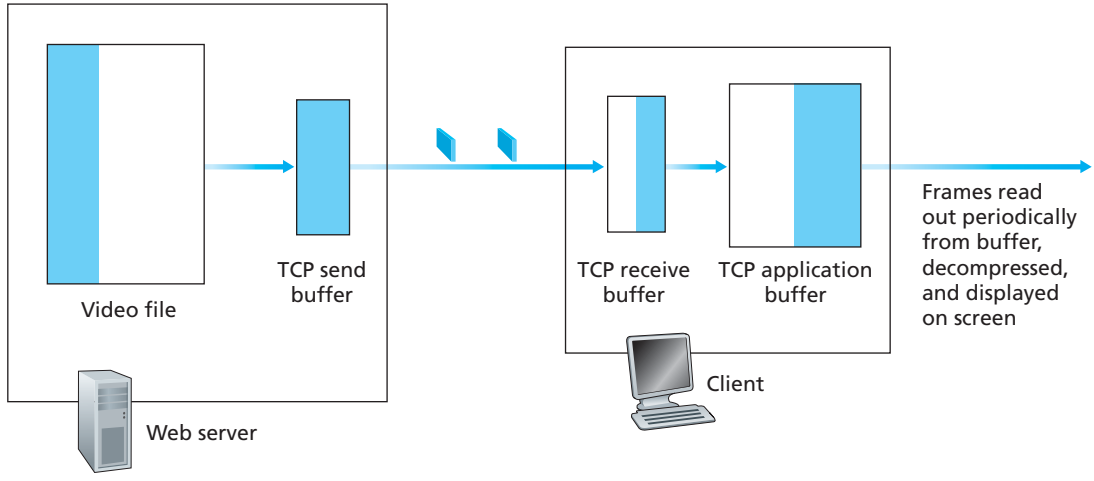

Figure 7.2 ♦ Streaming stored video over HTTP/TCP

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.

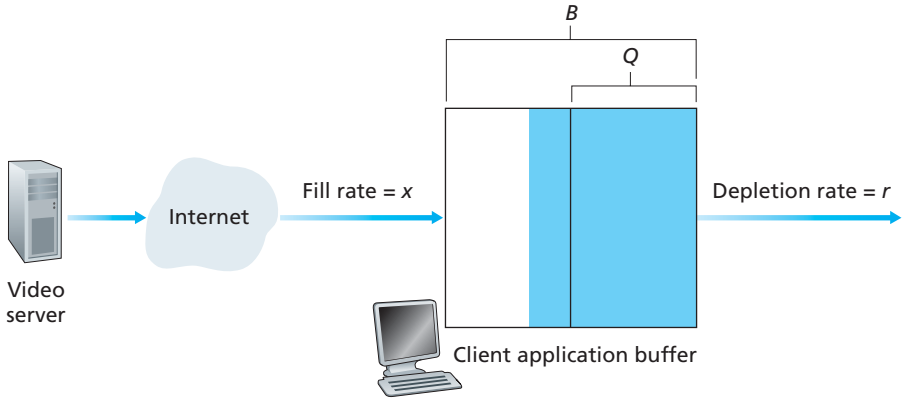

Figure 7.3 ♦ Analysis of client-side buffering for video streaming

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.*