7.4 MULTIMEDIA

The wireless Web is an exciting new development, but it is not the only one. For many people, multimedia is the holy grail of networking. When the word is mentioned, both the propeller heads and the suits begin salivating as if on cue. The former see immense technical challenges in providing (interactive) video on demand to every home. The latter see equally immense profits in it. Since multimedia requires high bandwidth, getting it to work over fixed connections is hard enough. Even VHS-quality video over wireless is a few years away, so our treatment will focus on wired systems.

Literally, multimedia is just two or more media. If the publisher of this book wanted to join the current hype about multimedia, it could advertise the book as using multimedia technology. After all, it contains two media: text and graphics (the figures). Nevertheless, when most people refer to multimedia, they generally mean the combination of two or more **continuous media**, that is, media that have to be played during some well-defined time interval, usually with some user interaction. In practice, the two media are normally audio and video, that is, sound plus moving pictures.

However, many people often refer to pure audio, such as Internet telephony or Internet radio as multimedia as well, which it is clearly not. Actually, a better term is **streaming media**, but we will follow the herd and consider real-time audio to be multimedia as well. In the following sections we will examine how computers process audio and video, how they are compressed, and some network applications of these technologies. For a comprehensive (three volume) treatment on networked multimedia, see (Steinmetz and Nahrstedt, 2002; Steinmetz and Nahrstedt, 2003a; and Steinmetz and Nahrstedt, 2003b).

7.4.1 Introduction to Digital Audio

An audio (sound) wave is a one-dimensional acoustic (pressure) wave. When an acoustic wave enters the ear, the eardrum vibrates, causing the tiny bones of the inner ear to vibrate along with it, sending nerve pulses to the brain. These pulses are perceived as sound by the listener. In a similar way, when an acoustic wave strikes a microphone, the microphone generates an electrical signal, representing the sound amplitude as a function of time. The representation, processing, storage, and transmission of such audio signals are a major part of the study of multimedia systems.

The frequency range of the human ear runs from 20 Hz to 20,000 Hz. Some animals, notably dogs, can hear higher frequencies. The ear hears logarithmically, so the ratio of two sounds with power A and B is conventionally expressed in **dB** (**decibels**) according to the formula

$$\text{dB} = 10 \log_{10}(A/B)$$

If we define the lower limit of audibility (a pressure of about 0.0003 dyne/cm^2) for a 1-kHz sine wave as 0 dB, an ordinary conversation is about 50 dB and the pain threshold is about 120 dB, a dynamic range of a factor of 1 million.

The ear is surprisingly sensitive to sound variations lasting only a few milliseconds. The eye, in contrast, does not notice changes in light level that last only a few milliseconds. The result of this observation is that jitter of only a few milliseconds during a multimedia transmission affects the perceived sound quality more than it affects the perceived image quality.

Audio waves can be converted to digital form by an **ADC** (**Analog Digital Converter**). An ADC takes an electrical voltage as input and generates a binary number as output. In Fig. 7-1(a) we see an example of a sine wave. To represent this signal digitally, we can sample it every ΔT seconds, as shown by the bar heights in Fig. 7-1(b). If a sound wave is not a pure sine wave but a linear superposition of sine waves where the highest frequency component present is f, then the Nyquist theorem (see Chap. 2) states that it is sufficient to make samples at a frequency $2f$. Sampling more often is of no value since the higher frequencies that such sampling could detect are not present.

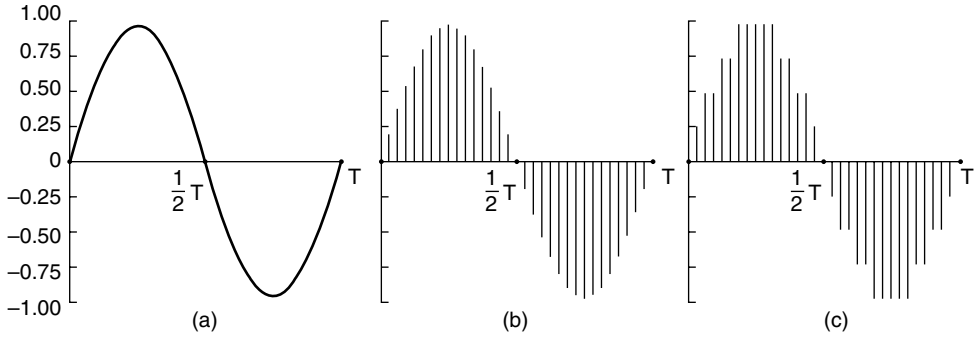

Figure 7-1. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

Digital samples are never exact. The samples of Fig. 7-1(c) allow only nine values, from −1.00 to +1.00 in steps of 0.25. An 8-bit sample would allow 256 distinct values. A 16-bit sample would allow 65,536 distinct values. The error introduced by the finite number of bits per sample is called the **quantization noise**. If it is too large, the ear detects it.

Two well-known examples where sampled sound is used are the telephone and audio compact discs. Pulse code modulation, as used within the telephone system, uses 8-bit samples made 8000 times per second. In North America and Japan, 7 bits are for data and 1 is for control; in Europe all 8 bits are for data. This system gives a data rate of 56,000 bps or 64,000 bps. With only 8000 samples/sec, frequencies above 4 kHz are lost.

Audio CDs are digital with a sampling rate of 44,100 samples/sec, enough to capture frequencies up to 22,050 Hz, which is good enough for people, but bad for canine music lovers. The samples are 16 bits each and are linear over the range of amplitudes. Note that 16-bit samples allow only 65,536 distinct values, even though the dynamic range of the ear is about 1 million when measured in steps of the smallest audible sound. Thus, using only 16 bits per sample introduces some quantization noise (although the full dynamic range is not covered—CDs are not supposed to hurt). With 44,100 samples/sec of 16 bits each, an audio CD needs a bandwidth of 705.6 kbps for monaural and 1.411 Mbps for stereo. While this is lower than what video needs (see below), it still takes almost a full T1 channel to transmit uncompressed CD quality stereo sound in real time.

Digitized sound can be easily processed by computers in software. Dozens of programs exist for personal computers to allow users to record, display, edit, mix, and store sound waves from multiple sources. Virtually all professional sound recording and editing are digital nowadays.

Music, of course, is just a special case of general audio, but an important one. Another important special case is speech. Human speech tends to be in the 600-Hz to 6000-Hz range. Speech is made up of vowels and consonants, which have different properties. Vowels are produced when the vocal tract is unobstructed, producing resonances whose fundamental frequency depends on the size and shape of the vocal system and the position of the speaker's tongue and jaw. These sounds are almost periodic for intervals of about 30 msec. Consonants are produced when the vocal tract is partially blocked. These sounds are less regular than vowels.

Some speech generation and transmission systems make use of models of the vocal system to reduce speech to a few parameters (e.g., the sizes and shapes of various cavities), rather than just sampling the speech waveform. How these vocoders work is beyond the scope of this book, however.

7.4.2 Audio Compression

CD-quality audio requires a transmission bandwidth of 1.411 Mbps, as we just saw. Clearly, substantial compression is needed to make transmission over the Internet practical. For this reason, various audio compression algorithms have been developed. Probably the most popular one is MPEG audio, which has three layers (variants), of which **MP3** (**MPEG audio layer 3**) is the most powerful and best known. Large amounts of music in MP3 format are available on the Internet, not all of it legal, which has resulted in numerous lawsuits from the artists and copyright owners. MP3 belongs to the audio portion of the MPEG video compression standard. We will discuss video compression later in this chapter; let us look at audio compression now.

Audio compression can be done in one of two ways. In **waveform coding** the signal is transformed mathematically by a Fourier transform into its frequency

components. Figure 2-1(a) shows an example function of time and its Fourier amplitudes. The amplitude of each component is then encoded in a minimal way. The goal is to reproduce the waveform accurately at the other end in as few bits as possible.

The other way, **perceptual coding**, exploits certain flaws in the human auditory system to encode a signal in such a way that it sounds the same to a human listener, even if it looks quite different on an oscilloscope. Perceptual coding is based on the science of **psychoacoustics**—how people perceive sound. MP3 is based on perceptual coding.

The key property of perceptual coding is that some sounds can **mask** other sounds. Imagine you are broadcasting a live flute concert on a warm summer day. Then all of a sudden, a crew of workmen nearby turn on their jackhammers and start tearing up the street. No one can hear the flute any more. Its sounds have been masked by the jackhammers. For transmission purposes, it is now sufficient to encode just the frequency band used by the jackhammers because the listeners cannot hear the flute anyway. This is called **frequency masking**—the ability of a loud sound in one frequency band to hide a softer sound in another frequency band that would have been audible in the absence of the loud sound. In fact, even after the jackhammers stop, the flute will be inaudible for a short period of time because the ear turns down its gain when they start and it takes a finite time to turn it up again. This effect is called **temporal masking**.

To make these effects more quantitative, imagine experiment 1. A person in a quiet room puts on headphones connected to a computer's sound card. The computer generates a pure sine wave at 100 Hz at low, but gradually increasing power. The person is instructed to strike a key when she hears the tone. The computer records the current power level and then repeats the experiment at 200 Hz, 300 Hz, and all the other frequencies up to the limit of human hearing. When averaged over many people, a log-log graph of how much power it takes for a tone to be audible looks like that of Fig. 7-2(a). A direct consequence of this curve is that it is never necessary to encode any frequencies whose power falls below the threshold of audibility. For example, if the power at 100 Hz were 20 dB in Fig. 7-2(a), it could be omitted from the output with no perceptible loss of quality because 20 dB at 100 Hz falls below the level of audibility.

Now consider Experiment 2. The computer runs experiment 1 again, but this time with a constant-amplitude sine wave at, say, 150 Hz, superimposed on the test frequency. What we discover is that the threshold of audibility for frequencies near 150 Hz is raised, as shown in Fig. 7-2(b).

The consequence of this new observation is that by keeping track of which signals are being masked by more powerful signals in nearby frequency bands, we can omit more and more frequencies in the encoded signal, saving bits. In Fig. 7-2, the 125-Hz signal can be completely omitted from the output and no one will be able to hear the difference. Even after a powerful signal stops in some frequency band, knowledge of its temporal masking properties allow us to continue to omit

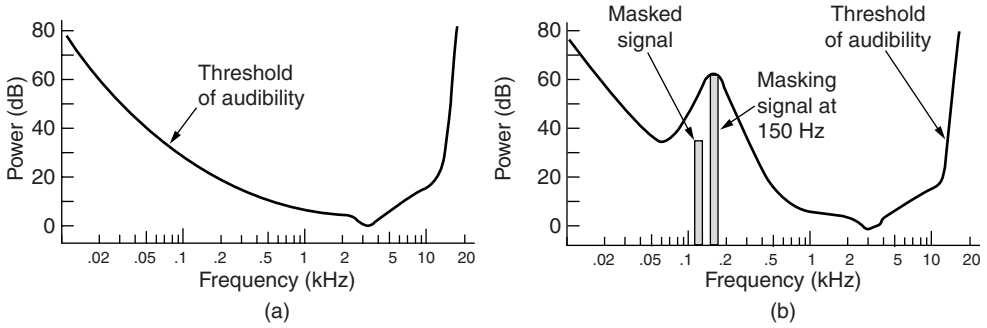

Figure 7-2. (a) The threshold of audibility as a function of frequency. (b) The masking effect.

the masked frequencies for some time interval as the ear recovers. The essence of MP3 is to Fourier-transform the sound to get the power at each frequency and then transmit only the unmasked frequencies, encoding these in as few bits as possible.

With this information as background, we can now see how the encoding is done. The audio compression is done by sampling the waveform at 32 kHz, 44.1 kHz, or 48 kHz. Sampling can be done on one or two channels, in any of four configurations:

1. Monophonic (a single input stream).
2. Dual monophonic (e.g., an English and a Japanese soundtrack).
3. Disjoint stereo (each channel compressed separately).
4. Joint stereo (interchannel redundancy fully exploited).

First, the output bit rate is chosen. MP3 can compress a stereo rock 'n roll CD down to 96 kbps with little perceptible loss in quality, even for rock 'n roll fans with no hearing loss. For a piano concert, at least 128 kbps are needed. These differ because the signal-to-noise ratio for rock 'n roll is much higher than for a piano concert (in an engineering sense, anyway). It is also possible to choose lower output rates and accept some loss in quality.

Then the samples are processed in groups of 1152 (about 26 msec worth). Each group is first passed through 32 digital filters to get 32 frequency bands. At the same time, the input is fed into a psychoacoustic model in order to determine the masked frequencies. Next, each of the 32 frequency bands is further transformed to provide a finer spectral resolution.

In the next phase the available bit budget is divided among the bands, with more bits allocated to the bands with the most unmasked spectral power, fewer bits allocated to unmasked bands with less spectral power, and no bits allocated to masked bands. Finally, the bits are encoded using Huffman encoding, which assigns short codes to numbers that appear frequently and long codes to those that occur infrequently.

There is actually more to the story. Various techniques are also used for noise reduction, antialiasing, and exploiting the interchannel redundancy, if possible, but these are beyond the scope of this book. A more formal mathematical introduction to the process is given in (Pan, 1995).

7.4.3 Streaming Audio

Let us now move from the technology of digital audio to three of its network applications. Our first one is streaming audio, that is, listening to sound over the Internet. This is also called music on demand. In the next two, we will look at Internet radio and voice over IP, respectively.

The Internet is full of music Web sites, many of which list song titles that users can click on to play the songs. Some of these are free sites (e.g., new bands looking for publicity); others require payment in return for music, although these often offer some free samples as well (e.g., the first 15 seconds of a song). The most straightforward way to make the music play is illustrated in Fig. 7-3.

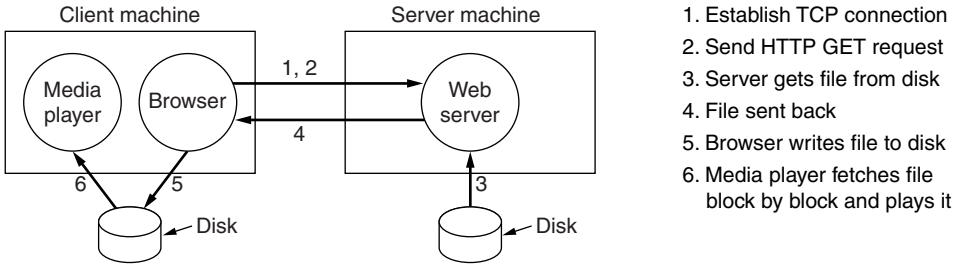

Figure 7-3. A straightforward way to implement clickable music on a Web page.

The process starts when the user clicks on a song. Then the browser goes into action. Step 1 is for it to establish a TCP connection to the Web server to which the song is hyperlinked. Step 2 is to send over a *GET* request in HTTP to request the song. Next (steps 3 and 4), the server fetches the song (which is just a file in MP3 or some other format) from the disk and sends it back to the browser. If the file is larger than the server's memory, it may fetch and send the music a block at a time.

Using the MIME type, for example, *audio/mp3*, (or the file extension), the browser looks up how it is supposed to display the file. Normally, there will be a helper application such as RealOne Player, Windows Media Player, or Winamp,

associated with this type of file. Since the usual way for the browser to communicate with a helper is to write the content to a scratch file, it will save the entire music file as a scratch file on the disk (step 5) first. Then it will start the media player and pass it the name of the scratch file. In step 6, the media player starts fetching and playing the music, block by block.

In principle, this approach is completely correct and will play the music. The only trouble is that the entire song must be transmitted over the network before the music starts. If the song is 4 MB (a typical size for an MP3 song) and the modem is 56 kbps, the user will be greeted by almost 10 minutes of silence while the song is being downloaded. Not all music lovers like this idea. Especially since the next song will also start with 10 minutes of download time, and the one after that as well.

To get around this problem without changing how the browser works, music sites have come up with the following scheme. The file linked to the song title is not the actual music file. Instead, it is what is called a **metafile**, a very short file just naming the music. A typical metafile might be only one line of ASCII text and look like this:

```
rtsp://joes-audio-server/song-0025.mp3
```

When the browser gets the 1-line file, it writes it to disk on a scratch file, starts the media player as a helper, and hands it the name of the scratch file, as usual. The media player then reads the file and sees that it contains a URL. Then it contacts *joes-audio-server* and asks for the song. Note that the browser is not in the loop any more.

In most cases, the server named in the metafile is not the same as the Web server. In fact, it is generally not even an HTTP server, but a specialized media server. In this example, the media server uses **RTSP** (**Real Time Streaming Protocol**), as indicated by the scheme name *rtsp*. It is described in RFC 2326.

The media player has four major jobs to do:

1. Manage the user interface.
2. Handle transmission errors.
3. Decompress the music.
4. Eliminate jitter.

Most media players nowadays have a glitzy user interface, sometimes simulating a stereo unit, with buttons, knobs, sliders, and visual displays. Often there are interchangeable front panels, called **skins**, that the user can drop onto the player. The media player has to manage all this and interact with the user.

Its second job is dealing with errors. Real-time music transmission rarely uses TCP because an error and retransmission might introduce an unacceptably long gap in the music. Instead, the actual transmission is usually done with a

protocol like RTP, which we studied in Chap. 6. Like most real-time protocols, RTP is layered on top of UDP, so packets may be lost. It is up to the player to deal with this.

In some cases, the music is interleaved to make error handling easier to do. For example, a packet might contain 220 stereo samples, each containing a pair of 16-bit numbers, normally good for 5 msec of music. But the protocol might send all the odd samples for a 10-msec interval in one packet and all the even samples in the next one. A lost packet then does not represent a 5 msec gap in the music, but loss of every other sample for 10 msec. This loss can be handled easily by having the media player interpolate using the previous and succeeding samples. estimate the missing value.

The use of interleaving to achieve error recovery is illustrated in Fig. 7-4. Here each packet holds the alternate time samples for an interval of 10 msec. Consequently, losing packet 3, as shown, does not create a gap in the music, but only lowers the temporal resolution for some interval. The missing values can be interpolated to provide continuous music. This particular scheme only works with uncompressed sampling, but shows how clever coding can convert a lost packet into lower quality rather than a time gap. However, RFC 3119 gives a scheme that works with compressed audio.

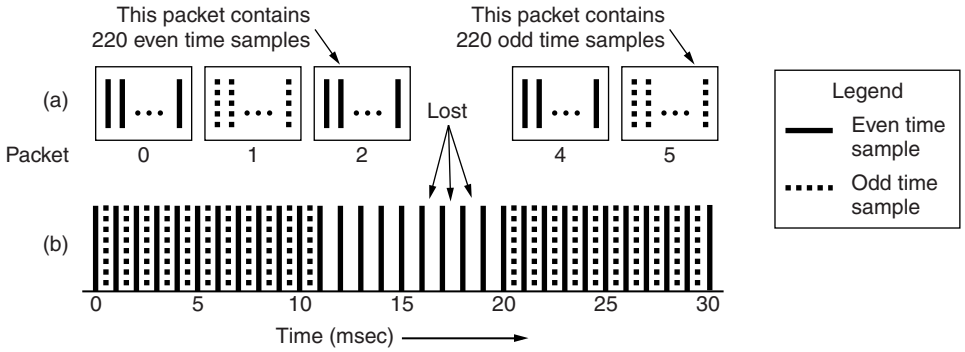

Figure 7-4. When packets carry alternate samples, the loss of a packet reduces the temporal resolution rather than creating a gap in time.

The media player's third job is decompressing the music. Although this task is computationally intensive, it is fairly straightforward.

The fourth job is to eliminate jitter, the bane of all real-time systems. All streaming audio systems start by buffering about 10–15 sec worth of music before starting to play, as shown in Fig. 7-5. Ideally, the server will continue to fill the buffer at the exact rate it is being drained by the media player, but in reality this may not happen, so feedback in the loop may be helpful.

Two approaches can be used to keep the buffer filled. With a **pull server**, as long as there is room in the buffer for another block, the media player just keeps

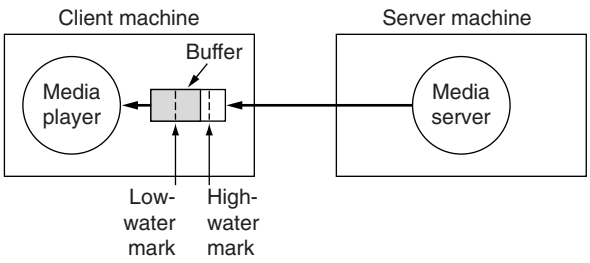

Figure 7-5. The media player buffers input from the media server and plays from the buffer rather than directly from the network.

sending requests for an additional block to the server. Its goal is to keep the buffer as full as possible.

The disadvantage of a pull server is all the unnecessary data requests. The server knows it has sent the whole file, so why have the player keep asking? For this reason, it is rarely used.

With a **push server**, the media player sends a *PLAY* request and the server just keeps pushing data at it. There are two possibilities here: the media server runs at normal playback speed or it runs faster. In both cases, some data is buffered before playback begins. If the server runs at normal playback speed, data arriving from it are appended to the end of the buffer and the player removes data from the front of the buffer for playing. As long as everything works perfectly, the amount of data in the buffer remains constant in time. This scheme is simple because no control messages are required in either direction.

The other push scheme is to have the server pump out data faster than it is needed. The advantage here is that if the server cannot be guaranteed to run at a regular rate, it has the opportunity to catch up if it ever gets behind. A problem here, however, is potential buffer overruns if the server can pump out data faster than it is consumed (and it has to be able to do this to avoid gaps).

The solution is for the media player to define a **low-water mark** and a **high-water mark** in the buffer. Basically, the server just pumps out data until the buffer is filled to the high-water mark. Then the media player tells it to pause. Since data will continue to pour in until the server has gotten the pause request, the distance between the high-water mark and the end of the buffer has to be greater than the bandwidth-delay product of the network. After the server has stopped, the buffer will begin to empty. When it hits the low-water mark, the media player tells the media server to start again. The low-water mark has to be positioned so that buffer underrun does not occur.

To operate a push server, the media player needs a remote control for it. This is what RTSP provides. It is defined in RFC 2326 and provides the mechanism for the player to control the server. It does not provide for the data stream, which is usually RTP. The main commands provided for by RTSP are listed in Fig. 7-6.

Command	Server action
DESCRIBE	List media parameters
SETUP	Establish a logical channel between the player and the server
PLAY	Start sending data to the client
RECORD	Start accepting data from the client
PAUSE	Temporarily stop sending data
TEARDOWN	Release the logical channel

Figure 7-6. RTSP commands from the player to the server.

7.4.4 Internet Radio

Once it became possible to stream audio over the Internet, commercial radio stations got the idea of broadcasting their content over the Internet as well as over the air. Not so long after that, college radio stations started putting their signal out over the Internet. Then college *students* started their own radio stations. With current technology, virtually anyone can start a radio station. The whole area of Internet radio is very new and in a state of flux, but it is worth saying a little bit about.

There are two general approaches to Internet radio. In the first one, the programs are prerecorded and stored on disk. Listeners can connect to the radio station's archives and pull up any program and download it for listening. In fact, this is exactly the same as the streaming audio we just discussed. It is also possible to store each program just after it is broadcast live, so the archive is only running, say, half an hour, or less behind the live feed. The advantages of this approach are that it is easy to do, all the techniques we have discussed work here too, and listeners can pick and choose among all the programs in the archive.

The other approach is to broadcast live over the Internet. Some stations broadcast over the air and over the Internet simultaneously, but there are increasingly many radio stations that are Internet only. Some of the techniques that are applicable to streaming audio are also applicable to live Internet radio, but there are also some key differences.

One point that is the same is the need for buffering on the user side to smooth out jitter. By collecting 10 or 15 seconds worth of radio before starting the playback, the audio can be kept going smoothly even in the face of substantial jitter over the network. As long as all the packets arrive before they are needed, it does not matter when they arrived.

One key difference is that streaming audio can be pushed out at a rate greater than the playback rate since the receiver can stop it when the high-water mark is hit. Potentially, this gives it the time to retransmit lost packets, although this strategy is not commonly used. In contrast, live radio is always broadcast at exactly the rate it is generated and played back.

Another difference is that a live radio station usually has hundreds or thousands of simultaneous listeners whereas streaming audio is point to point. Under these circumstances, Internet radio should use multicasting with the RTP/RTSP protocols. This is clearly the most efficient way to operate.

In current practice, Internet radio does not work like this. What actually happens is that the user establishes a TCP connection to the station and the feed is sent over the TCP connection. Of course, this creates various problems, such as the flow stopping when the window is full, lost packets timing out and being retransmitted, and so on.

The reason TCP unicasting is used instead of RTP multicasting is threefold. First, few ISPs support multicasting, so that is not a practical option. Second, RTP is less well known than TCP and radio stations are often small and have little computer expertise, so it is just easier to use a protocol that is widely understood and supported by all software packages. Third, many people listen to Internet radio at work, which in practice, often means behind a firewall. Most system administrators configure their firewall to protect their LAN from unwelcome visitors. They usually allow TCP connections from remote port 25 (SMTP for e-mail), UDP packets from remote port 53 (DNS), and TCP connections from remote port 80 (HTTP for the Web). Almost everything else may be blocked, including RTP. Thus, the only way to get the radio signal through the firewall is for the Web site to pretend it is an HTTP server, at least to the firewall, and use HTTP servers, which speak TCP. These severe measures, while providing only minimal security. often force multimedia applications into drastically less efficient modes of operation.

Since Internet radio is a new medium, format wars are in full bloom. RealAudio, Windows Media Audio, and MP3 are aggressively competing in this market to become the dominant format for Internet radio. A newcomer is Vorbis, which is technically similar to MP3 but open source and different enough that it does not use the patents MP3 is based on.

A typical Internet radio station has a Web page listing its schedule, information about its DJs and announcers, and many ads. There are also one or more icons listing the audio formats it supports (or just LISTEN NOW if only one format is supported). These icons or LISTEN NOW are linked metafiles of the type we discussed above.

When a user clicks on one of the icons, the short metafile is sent over. The browser uses its MIME type or file extension to determine the appropriate helper (i.e., media player) for the metafile. Then it writes the metafile to a scratch file on disk, starts the media player, and hands it the name of the scratch file. The media player reads the scratch file, sees the URL contained in it (usually with scheme *http* rather than *rtsp* to get around the firewall problem and because some popular multimedia applications work that way), contacts the server, and starts acting like a radio. As an aside, audio has only one stream, so *http* works, but for video, which has at least two streams, *http* fails and something like *rtsp* is really needed.

Another interesting development in the area of Internet radio is an arrangement in which anybody, even a student, can set up and operate a radio station. The main components are illustrated in Fig. 7-7. The basis of the station is an ordinary PC with a sound card and a microphone. The software consists of a media player, such as Winamp or Freeamp, with a plug-in for audio capture and a codec for the selected output format, for example, MP3 or Vorbis.

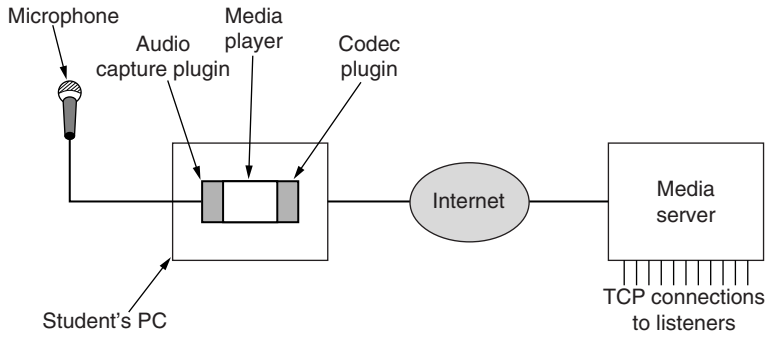

Figure 7-7. A student radio station.

The audio stream generated by the station is then fed over the Internet to a large server, which handles distributing it to large numbers of TCP connections. The server typically supports many small stations. It also maintains a directory of what stations it has and what is currently on the air on each one. Potential listeners go to the server, select a station, and get a TCP feed. There are commercial software packages for managing all the pieces, as well as open source packages such as icecast. There are also servers that are willing to handle the distribution for a fee.

7.4.5 Voice over IP

Once upon a time, the public switched telephone system was primarily used for voice traffic with a little bit of data traffic here and there. But the data traffic grew and grew, and by 1999, the number of data bits moved equaled the number of voice bits (since voice is in PCM on the trunks, it can be measured in bits/sec). By 2002, the volume of data traffic was an order of magnitude more than the volume of voice traffic and still growing exponentially, with voice traffic being almost flat (5% growth per year).

As a consequence of these numbers, many packet-switching network operators suddenly became interested in carrying voice over their data networks. The amount of additional bandwidth required for voice is minuscule since the packet networks are dimensioned for the data traffic. However, the average person's

phone bill is probably larger than his Internet bill, so the data network operators saw Internet telephony as a way to earn a large amount of additional money without having to put any new fiber in the ground. Thus **Internet telephony** (also known as **voice over IP**), was born.

H.323

One thing that was clear to everyone from the start was that if each vendor designed its own protocol stack, the system would never work. To avoid this problem, a number of interested parties got together under ITU auspices to work out standards. In 1996 ITU issued recommendation **H.323** entitled "Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-Guaranteed Quality of Service.'' Only the telephone industry would think of such a name. The recommendation was revised in 1998, and this revised H.323 was the basis for the first widespread Internet telephony systems.

H.323 is more of an architectural overview of Internet telephony than a specific protocol. It references a large number of specific protocols for speech coding, call setup, signaling, data transport, and other areas rather than specifying these things itself. The general model is depicted in Fig. 7-8. At the center is a **gateway** that connects the Internet to the telephone network. It speaks the H.323 protocols on the Internet side and the PSTN protocols on the telephone side. The communicating devices are called **terminals**. A LAN may have a **gatekeeper**, which controls the end points under its jurisdiction, called a **zone**.

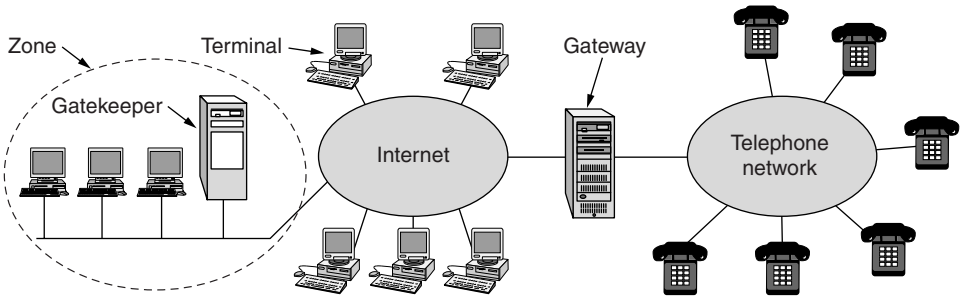

Figure 7-8. The H.323 architectural model for Internet telephony.

A telephone network needs a number of protocols. To start with, there is a protocol for encoding and decoding speech. The PCM system we studied in Chap. 2 is defined in ITU recommendation **G.711**. It encodes a single voice channel by sampling 8000 times per second with an 8-bit sample to give uncompressed speech at 64 kbps. All H.323 systems must support G.711. However, other speech compression protocols are also permitted (but not required). They use different compression algorithms and make different trade-offs between quality and

bandwidth. For example, **G.723.1** takes a block of 240 samples (30 msec of speech) and uses predictive coding to reduce it to either 24 bytes or 20 bytes. This algorithm gives an output rate of either 6.4 kbps or 5.3 kbps (compression factors of 10 and 12), respectively, with little loss in perceived quality. Other codecs are also allowed.

Since multiple compression algorithms are permitted, a protocol is needed to allow the terminals to negotiate which one they are going to use. This protocol is called **H.245**. It also negotiates other aspects of the connection such as the bit rate. RTCP is need for the control of the RTP channels. Also required is a protocol for establishing and releasing connections, providing dial tones, making ringing sounds, and the rest of the standard telephony. ITU **Q.931** is used here. The terminals need a protocol for talking to the gatekeeper (if present). For this purpose, **H.225** is used. The PC-to-gatekeeper channel it manages is called the **RAS** (**Registration/Admission/Status**) channel. This channel allows terminals to join and leave the zone, request and return bandwidth, and provide status updates, among other things. Finally, a protocol is needed for the actual data transmission. RTP is used for this purpose. It is managed by RTCP, as usual. The positioning of all these protocols is shown in Fig. 7-9.

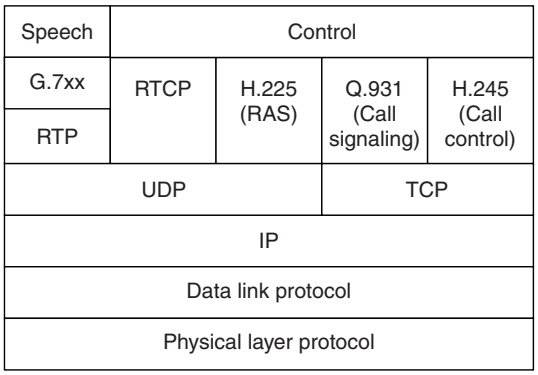

Figure 7-9. The H.323 protocol stack.

To see how these protocols fit together, consider the case of a PC terminal on a LAN (with a gatekeeper) calling a remote telephone. The PC first has to discover the gatekeeper, so it broadcasts a UDP gatekeeper discovery packet to port 1718. When the gatekeeper responds, the PC learns the gatekeeper's IP address. Now the PC registers with the gatekeeper by sending it a RAS message in a UDP packet. After it has been accepted, the PC sends the gatekeeper a RAS admission message requesting bandwidth. Only after bandwidth has been granted may call setup begin. The idea of requesting bandwidth in advance is to allow the gatekeeper to limit the number of calls to avoid oversubscribing the outgoing line in order to help provide the necessary quality of service.

The PC now establishes a TCP connection to the gatekeeper to begin call setup. Call setup uses existing telephone network protocols, which are connection oriented, so TCP is needed. In contrast, the telephone system has nothing like RAS to allow telephones to announce their presence, so the H.323 designers were free to use either UDP or TCP for RAS, and they chose the lower-overhead UDP.

Now that it has bandwidth allocated, the PC can send a Q.931 *SETUP* message over the TCP connection. This message specifies the number of the telephone being called (or the IP address and port, if a computer is being called). The gatekeeper responds with a Q.931 *CALL PROCEEDING* message to acknowledge correct receipt of the request. The gatekeeper then forwards the *SETUP* message to the gateway.

The gateway, which is half computer, half telephone switch, then makes an ordinary telephone call to the desired (ordinary) telephone. The end office to which the telephone is attached rings the called telephone and also sends back a Q.931 *ALERT* message to tell the calling PC that ringing has begun. When the person at the other end picks up the telephone, the end office sends back a Q.931 *CONNECT* message to signal the PC that it has a connection.

Once the connection has been established, the gatekeeper is no longer in the loop, although the gateway is, of course. Subsequent packets bypass the gatekeeper and go directly to the gateway's IP address. At this point, we just have a bare tube running between the two parties. This is just a physical layer connection for moving bits, no more. Neither side knows anything about the other one.

The H.245 protocol is now used to negotiate the parameters of the call. It uses the H.245 control channel, which is always open. Each side starts out by announcing its capabilities, for example, whether it can handle video (H.323 can handle video) or conference calls, which codecs it supports, etc. Once each side knows what the other one can handle, two unidirectional data channels are set up and a codec and other parameters assigned to each one. Since each side may have different equipment, it is entirely possible that the codecs on the forward and reverse channels are different. After all negotiations are complete, data flow can begin using RTP. It is managed using RTCP, which plays a role in congestion control. If video is present, RTCP handles the audio/video synchronization. The various channels are shown in Fig. 7-10. When either party hangs up, the Q.931 call signaling channel is used to tear down the connection.

When the call is terminated, the calling PC contacts the gatekeeper again with a RAS message to release the bandwidth it has been assigned. Alternatively, it can make another call.

We have not said anything about quality of service, even though this is essential to making voice over IP a success. The reason is that QoS falls outside the scope of H.323. If the underlying network is capable of producing a stable, jitter-free connection from the calling PC (e.g., using the techniques we discussed in Chap. 5) to the gateway, then the QoS on the call will be good; otherwise it will not be. The telephone part uses PCM and is always jitter free.

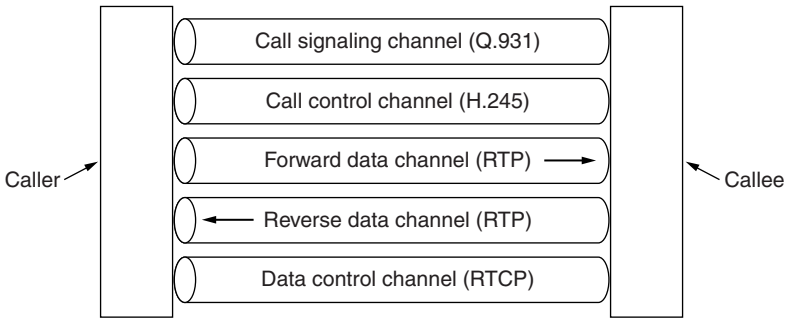

Figure 7-10. Logical channels between the caller and callee during a call.

SIP—The Session Initiation Protocol

H.323 was designed by ITU. Many people in the Internet community saw it as a typical telco product: large, complex, and inflexible. Consequently, IETF set up a committee to design a simpler and more modular way to do voice over IP. The major result to date is the **SIP** (**Session Initiation Protocol**), which is described in RFC 3261. This protocol describes how to set up Internet telephone calls, video conferences, and other multimedia connections. Unlike H.323, which is a complete protocol suite, SIP is a single module, but it has been designed to interwork well with existing Internet applications. For example, it defines telephone numbers as URLs, so that Web pages can contain them, allowing a click on a link to initiate a telephone call (the same way the *mailto* scheme allows a click on a link to bring up a program to send an e-mail message).

SIP can establish two-party sessions (ordinary telephone calls), multiparty sessions (where everyone can hear and speak), and multicast sessions (one sender, many receivers). The sessions may contain audio, video, or data, the latter being useful for multiplayer real-time games, for example. SIP just handles setup, management, and termination of sessions. Other protocols, such as RTP/RTCP, are used for data transport. SIP is an application-layer protocol and can run over UDP or TCP.

SIP supports a variety of services, including locating the callee (who may not be at his home machine) and determining the callee's capabilities, as well as handling the mechanics of call setup and termination. In the simplest case, SIP sets up a session from the caller's computer to the callee's computer, so we will examine that case first.

Telephone numbers in SIP are represented as URLs using the *sip* scheme, for example, *sip:ilse@cs.university.edu* for a user named Ilse at the host specified by the DNS name *cs.university.edu*. SIP URLs may also contain IPv4 addresses, IPv6 address, or actual telephone numbers.

The SIP protocol is a text-based protocol modeled on HTTP. One party sends a message in ASCII text consisting of a method name on the first line, followed by additional lines containing headers for passing parameters. Many of the headers are taken from MIME to allow SIP to interwork with existing Internet applications. The six methods defined by the core specification are listed in Fig. 7-11.

Method	Description
INVITE	Request initiation of a session
ACK	Confirm that a session has been initiated
BYE	Request termination of a session
OPTIONS	Query a host about its capabilities
CANCEL	Cancel a pending request
REGISTER	Inform a redirection server about the user's current location

Figure 7-11. The SIP methods defined in the core specification.

To establish a session, the caller either creates a TCP connection with the callee and sends an *INVITE* message over it or sends the *INVITE* message in a UDP packet. In both cases, the headers on the second and subsequent lines describe the structure of the message body, which contains the caller's capabilities, media types, and formats. If the callee accepts the call, it responds with an HTTP-type reply code (a three-digit number using the groups of Fig. 7-0, 200 for acceptance). Following the reply-code line, the callee also may supply information about its capabilities, media types, and formats.

Connection is done using a three-way handshake, so the caller responds with an *ACK* message to finish the protocol and confirm receipt of the 200 message.

Either party may request termination of a session by sending a message containing the *BYE* method. When the other side acknowledges it, the session is terminated.

The *OPTIONS* method is used to query a machine about its own capabilities. It is typically used before a session is initiated to find out if that machine is even capable of voice over IP or whatever type of session is being contemplated.

The *REGISTER* method relates to SIP's ability to track down and connect to a user who is away from home. This message is sent to a SIP location server that keeps track of who is where. That server can later be queried to find the user's current location. The operation of redirection is illustrated in Fig. 7-12. Here the caller sends the *INVITE* message to a proxy server to hide the possible redirection. The proxy then looks up where the user is and sends the *INVITE* message there. It then acts as a relay for the subsequent messages in the three-way handshake. The *LOOKUP* and *REPLY* messages are not part of SIP; any convenient protocol can be used, depending on what kind of location server is used.

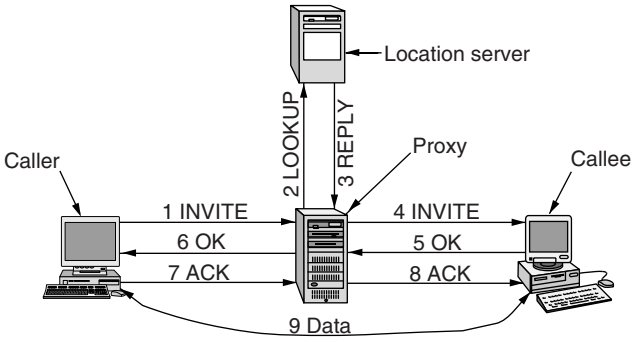

Figure 7-12. Use a proxy and redirection servers with SIP.

SIP has a variety of other features that we will not describe here, including call waiting, call screening, encryption, and authentication. It also has the ability to place calls from a computer to an ordinary telephone, if a suitable gateway between the Internet and telephone system is available.

Comparison of H.323 and SIP

H.323 and SIP have many similarities but also some differences. Both allow two-party and multiparty calls using both computers and telephones as end points. Both support parameter negotiation, encryption, and the RTP/RTCP protocols. A summary of the similarities and differences is given in Fig. 7-13.

Although the feature sets are similar, the two protocols differ widely in philosophy. H.323 is a typical, heavyweight, telephone-industry standard, specifying the complete protocol stack and defining precisely what is allowed and what is forbidden. This approach leads to very well defined protocols in each layer, easing the task of interoperability. The price paid is a large, complex, and rigid standard that is difficult to adapt to future applications.

In contrast, SIP is a typical Internet protocol that works by exchanging short lines of ASCII text. It is a lightweight module that interworks well with other Internet protocols but less well with existing telephone system signaling protocols. Because the IETF model of voice over IP is highly modular, it is flexible and can be adapted to new applications easily. The downside is potential interoperability problems, although these are addressed by frequent meetings where different implementers get together to test their systems.

Voice over IP is an up-and-coming topic. Consequently, there are several books on the subject already. A few examples are (Collins, 2001; Davidson and Peters, 2000; Kumar et al., 2001; and Wright, 2001). The May/June 2002 issue of *Internet Computing* has several articles on this topic.

Item	H.323	SIP
Designed by	ITU	IETF
Compatibility with PSTN	Yes	Largely
Compatibility with Internet	No	Yes
Architecture	Monolithic	Modular
Completeness	Full protocol stack	SIP just handles setup
Parameter negotiation	Yes	Yes
Call signaling	Q.931 over TCP	SIP over TCP or UDP
Message format	Binary	ASCII
Media transport	RTP/RTCP	RTP/RTCP
Multiparty calls	Yes	Yes
Multimedia conferences	Yes	No
Addressing	Host or telephone number	URL
Call termination	Explicit or TCP release	Explicit or timeout
Instant messaging	No	Yes
Encryption	Yes	Yes
Size of standards	1400 pages	250 pages
Implementation	Large and complex	Moderate
Status	Widely deployed	Up and coming

Figure 7-13. Comparison of H.323 and SIP

7.4.6 Introduction to Video

We have discussed the ear at length now; time to move on to the eye (no, this section is not followed by one on the nose). The human eye has the property that when an image appears on the retina, the image is retained for some number of milliseconds before decaying. If a sequence of images is drawn line by line at 50 images/sec, the eye does not notice that it is looking at discrete images. All video (i.e., television) systems exploit this principle to produce moving pictures.

Analog Systems

To understand video, it is best to start with simple, old-fashioned black-and-white television. To represent the two-dimensional image in front of it as a one-dimensional voltage as a function of time, the camera scans an electron beam rapidly across the image and slowly down it, recording the light intensity as it goes. At the end of the scan, called a **frame**, the beam retraces. This intensity as a function of time is broadcast, and receivers repeat the scanning process to re-

construct the image. The scanning pattern used by both the camera and the receiver is shown in Fig. 7-14. (As an aside, CCD cameras integrate rather than scan, but some cameras and all monitors do scan.)

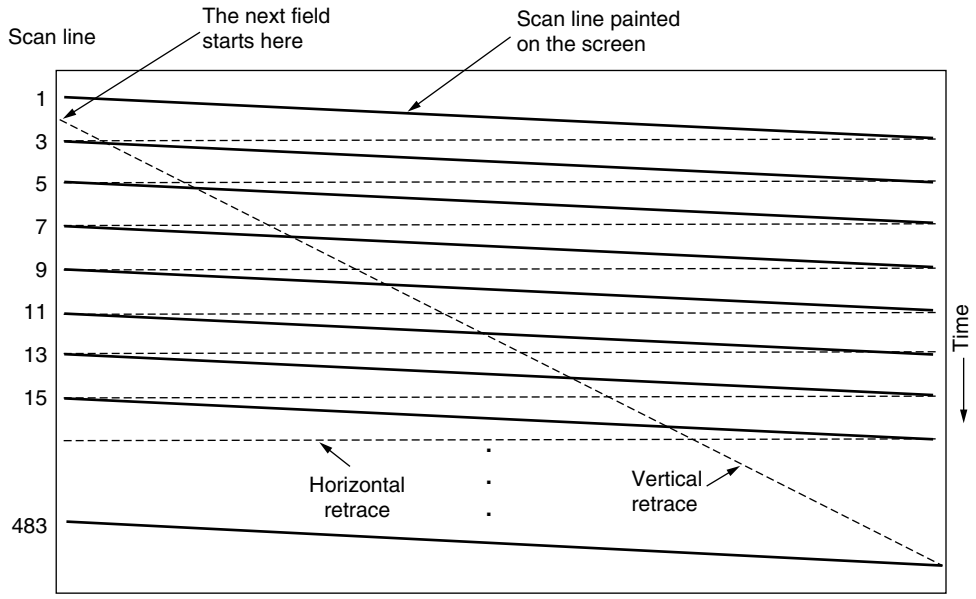

Figure 7-14. The scanning pattern used for NTSC video and television.

The exact scanning parameters vary from country to country. The system used in North and South America and Japan has 525 scan lines, a horizontal-to-vertical aspect ratio of 4:3, and 30 frames/sec. The European system has 625 scan lines, the same aspect ratio of 4:3, and 25 frames/sec. In both systems, the top few and bottom few lines are not displayed (to approximate a rectangular image on the original round CRTs). Only 483 of the 525 NTSC scan lines (and 576 of the 625 PAL/SECAM scan lines) are displayed. The beam is turned off during the vertical retrace, so many stations (especially in Europe) use this time to broadcast TeleText (text pages containing news, weather, sports, stock prices, etc.).

While 25 frames/sec is enough to capture smooth motion, at that frame rate many people, especially older ones, will perceive the image to flicker (because the old image has faded off the retina before the new one appears). Rather than increase the frame rate, which would require using more scarce bandwidth, a different approach is taken. Instead of the scan lines being displayed in order, first all the odd scan lines are displayed, then the even ones are displayed. Each of these half frames is called a **field**. Experiments have shown that although people notice flicker at 25 frames/sec, they do not notice it at 50 fields/sec. This technique is called **interlacing**. Noninterlaced television or video is called **progressive**. Note that movies run at 24 fps, but each frame is fully visible for 1/24 sec.

Color video uses the same scanning pattern as monochrome (black and white), except that instead of displaying the image with one moving beam, it uses three beams moving in unison. One beam is used for each of the three additive primary colors: red, green, and blue (RGB). This technique works because any color can be constructed from a linear superposition of red, green, and blue with the appropriate intensities. However, for transmission on a single channel, the three color signals must be combined into a single **composite** signal.

When color television was invented, various methods for displaying color were technically possible, and different countries made different choices, leading to systems that are still incompatible. (Note that these choices have nothing to do with VHS versus Betamax versus P2000, which are recording methods.) In all countries, a political requirement was that programs transmitted in color had to be receivable on existing black-and-white television sets. Consequently, the simplest scheme, just encoding the RGB signals separately, was not acceptable. RGB is also not the most efficient scheme.

The first color system was standardized in the United States by the **National Television Standards Committee**, which lent its acronym to the standard: **NTSC**. Color television was introduced in Europe several years later, by which time the technology had improved substantially, leading to systems with greater noise immunity and better colors. These systems are called **SECAM** (**SEquentiel Couleur Avec Memoire**), which is used in France and Eastern Europe, and **PAL** (**Phase Alternating Line**) used in the rest of Europe. The difference in color quality between the NTSC and PAL/SECAM has led to an industry joke that NTSC really stands for Never Twice the Same Color.

To allow color transmissions to be viewed on black-and-white receivers, all three systems linearly combine the RGB signals into a **luminance** (brightness) signal and two **chrominance** (color) signals, although they all use different coefficients for constructing these signals from the RGB signals. Oddly enough, the eye is much more sensitive to the luminance signal than to the chrominance signals, so the latter need not be transmitted as accurately. Consequently, the luminance signal can be broadcast at the same frequency as the old black-and-white signal, so it can be received on black-and-white television sets. The two chrominance signals are broadcast in narrow bands at higher frequencies. Some television sets have controls labeled brightness, hue, and saturation (or brightness, tint, and color) for controlling these three signals separately. Understanding luminance and chrominance is necessary for understanding how video compression works.

In the past few years, there has been considerable interest in **HDTV** (**High Definition TeleVision**), which produces sharper images by roughly doubling the number of scan lines. The United States, Europe, and Japan have all developed HDTV systems, all different and all mutually incompatible. Did you expect otherwise? The basic principles of HDTV in terms of scanning, luminance, chrominance, and so on, are similar to the existing systems. However, all three formats

have a common aspect ratio of 16:9 instead of 4:3 to match them better to the format used for movies (which are recorded on 35 mm film, which has an aspect ratio of 3:2).

Digital Systems

The simplest representation of digital video is a sequence of frames, each consisting of a rectangular grid of picture elements, or **pixels**. Each pixel can be a single bit, to represent either black or white. The quality of such a system is similar to what you get by sending a color photograph by fax—awful. (Try it if you can; otherwise photocopy a color photograph on a copying machine that does not rasterize.)

The next step up is to use 8 bits per pixel to represent 256 gray levels. This scheme gives high-quality black-and-white video. For color video, good systems use 8 bits for each of the RGB colors, although nearly all systems mix these into composite video for transmission. While using 24 bits per pixel limits the number of colors to about 16 million, the human eye cannot even distinguish this many colors, let alone more. Digital color images are produced using three scanning beams, one per color. The geometry is the same as for the analog system of Fig. 7-14 except that the continuous scan lines are now replaced by neat rows of discrete pixels.

To produce smooth motion, digital video, like analog video, must display at least 25 frames/sec. However, since good-quality computer monitors often rescan the screen from images stored in memory at 75 times per second or more, interlacing is not needed and consequently is not normally used. Just repainting (i.e., redrawing) the same frame three times in a row is enough to eliminate flicker.

In other words, smoothness of motion is determined by the number of *different* images per second, whereas flicker is determined by the number of times the screen is painted per second. These two parameters are different. A still image painted at 20 frames/sec will not show jerky motion, but it will flicker because one frame will decay from the retina before the next one appears. A movie with 20 different frames per second, each of which is painted four times in a row, will not flicker, but the motion will appear jerky.

The significance of these two parameters becomes clear when we consider the bandwidth required for transmitting digital video over a network. Current computer monitors most use the 4:3 aspect ratio so they can use inexpensive, mass-produced picture tubes designed for the consumer television market. Common configurations are 1024×768, 1280×960, and 1600×1200. Even the smallest of these with 24 bits per pixel and 25 frames/sec needs to be fed at 472 Mbps. It would take a SONET OC-12 carrier to manage this, and running an OC-12 SONET carrier into everyone's house is not exactly on the agenda. Doubling this rate to avoid flicker is even less attractive. A better solution is to transmit 25

frames/sec and have the computer store each one and paint it twice. Broadcast television does not use this strategy because television sets do not have memory. And even if they did have memory, analog signals cannot be stored in RAM without conversion to digital form first, which requires extra hardware. As a consequence, interlacing is needed for broadcast television but not for digital video.

7.4.7 Video Compression

It should be obvious by now that transmitting uncompressed video is completely out of the question. The only hope is that massive compression is possible. Fortunately, a large body of research over the past few decades has led to many compression techniques and algorithms that make video transmission feasible. In this section we will study how video compression is accomplished.

All compression systems require two algorithms: one for compressing the data at the source, and another for decompressing it at the destination. In the literature, these algorithms are referred to as the **encoding** and **decoding** algorithms, respectively. We will use this terminology here, too.

These algorithms exhibit certain asymmetries that are important to understand. First, for many applications, a multimedia document, say, a movie will only be encoded once (when it is stored on the multimedia server) but will be decoded thousands of times (when it is viewed by customers). This asymmetry means that it is acceptable for the encoding algorithm to be slow and require expensive hardware provided that the decoding algorithm is fast and does not require expensive hardware. After all, the operator of a multimedia server might be quite willing to rent a parallel supercomputer for a few weeks to encode its entire video library, but requiring consumers to rent a supercomputer for 2 hours to view a video is not likely to be a big success. Many practical compression systems go to great lengths to make decoding fast and simple, even at the price of making encoding slow and complicated.

On the other hand, for real-time multimedia, such as video conferencing, slow encoding is unacceptable. Encoding must happen on-the-fly, in real time. Consequently, real-time multimedia uses different algorithms or parameters than storing videos on disk, often with appreciably less compression.

A second asymmetry is that the encode/decode process need not be invertible. That is, when compressing a file, transmitting it, and then decompressing it, the user expects to get the original back, accurate down to the last bit. With multimedia, this requirement does not exist. It is usually acceptable to have the video signal after encoding and then decoding be slightly different from the original. When the decoded output is not exactly equal to the original input, the system is said to be **lossy**. If the input and output are identical, the system is **lossless**. Lossy systems are important because accepting a small amount of information loss can give a huge payoff in terms of the compression ratio possible.

The JPEG Standard

A video is just a sequence of images (plus sound). If we could find a good algorithm for encoding a single image, this algorithm could be applied to each image in succession to achieve video compression. Good still image compression algorithms exist, so let us start our study of video compression there. The **JPEG** (**Joint Photographic Experts Group**) standard for compressing continuous-tone still pictures (e.g., photographs) was developed by photographic experts working under the joint auspices of ITU, ISO, and IEC, another standards body. It is important for multimedia because, to a first approximation, the multimedia standard for moving pictures, MPEG, is just the JPEG encoding of each frame separately, plus some extra features for interframe compression and motion detection. JPEG is defined in International Standard 10918.

JPEG has four modes and many options. It is more like a shopping list than a single algorithm. For our purposes, though, only the lossy sequential mode is relevant, and that one is illustrated in Fig. 7-15. Furthermore, we will concentrate on the way JPEG is normally used to encode 24-bit RGB video images and will leave out some of the minor details for the sake of simplicity.

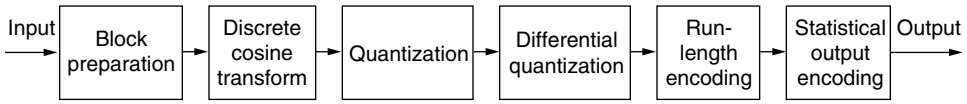

Figure 7-15. The operation of JPEG in lossy sequential mode.

Step 1 of encoding an image with JPEG is block preparation. For the sake of specificity, let us assume that the JPEG input is a 640×480 RGB image with 24 bits/pixel, as shown in Fig. 7-16(a). Since using luminance and chrominance gives better compression, we first compute the luminance, Y, and the two chrominances, I and Q (for NTSC), according to the following formulas:

$$\begin{aligned} Y &= 0.30R + 0.59G + 0.11B \\ I &= 0.60R - 0.28G - 0.32B \\ Q &= 0.21R - 0.52G + 0.31B \end{aligned}$$

For PAL, the chrominances are called U and V and the coefficients are different, but the idea is the same. SECAM is different from both NTSC and PAL.

Separate matrices are constructed for Y, I, and Q, each with elements in the range 0 to 255. Next, square blocks of four pixels are averaged in the I and Q matrices to reduce them to 320×240. This reduction is lossy, but the eye barely notices it since the eye responds to luminance more than to chrominance. Nevertheless, it compresses the total amount of data by a factor of two. Now 128 is subtracted from each element of all three matrices to put 0 in the middle of the

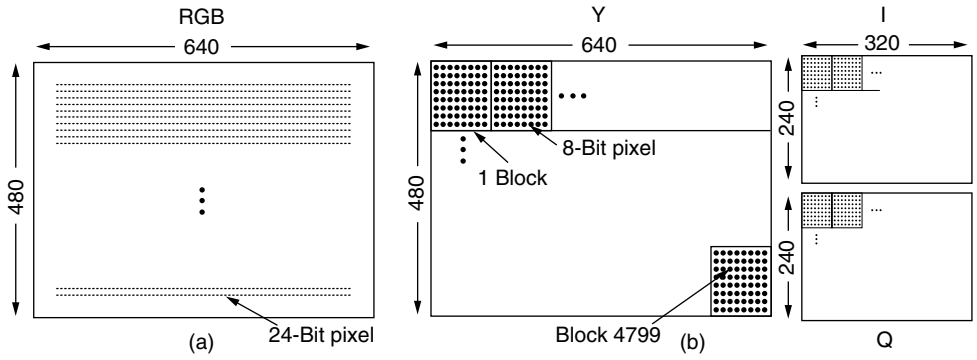

Figure 7-16. (a) RGB input data. (b) After block preparation.

range. Finally, each matrix is divided up into 8×8 blocks. The Y matrix has 4800 blocks; the other two have 1200 blocks each, as shown in Fig. 7-16(b).

Step 2 of JPEG is to apply a **DCT** (**Discrete Cosine Transformation**) to each of the 7200 blocks separately. The output of each DCT is an 8×8 matrix of DCT coefficients. DCT element (0, 0) is the average value of the block. The other elements tell how much spectral power is present at each spatial frequency. In theory, a DCT is lossless, but in practice, using floating-point numbers and transcendental functions always introduces some roundoff error that results in a little information loss. Normally, these elements decay rapidly with distance from the origin, (0, 0), as suggested by Fig. 7-17.

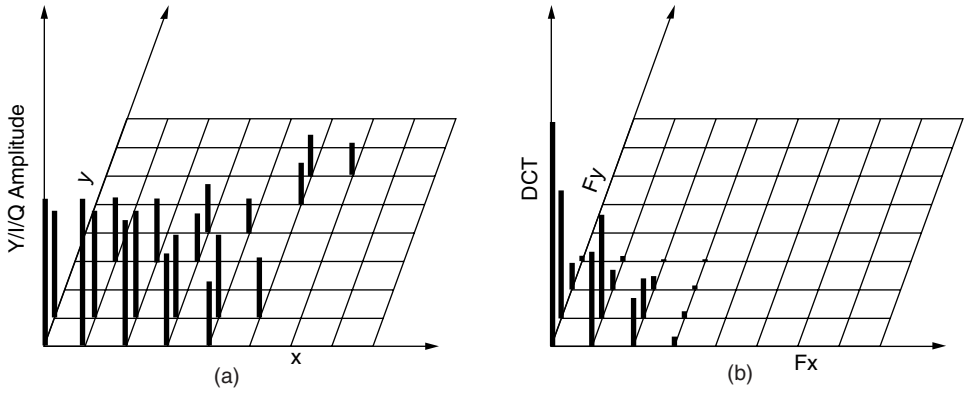

Figure 7-17. (a) One block of the Y matrix. (b) The DCT coefficients.

Once the DCT is complete, JPEG moves on to step 3, called **quantization**, in which the less important DCT coefficients are wiped out. This (lossy) transforma-

tion is done by dividing each of the coefficients in the 8×8 DCT matrix by a weight taken from a table. If all the weights are 1, the transformation does nothing. However, if the weights increase sharply from the origin, higher spatial frequencies are dropped quickly.

An example of this step is given in Fig. 7-18. Here we see the initial DCT matrix, the quantization table, and the result obtained by dividing each DCT element by the corresponding quantization table element. The values in the quantization table are not part of the JPEG standard. Each application must supply its own, allowing it to control the loss-compression trade-off.

DCT Coefficients

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantization table

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Quantized coefficients

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 7-18. Computation of the quantized DCT coefficients.

Step 4 reduces the (0, 0) value of each block (the one in the upper-left corner) by replacing it with the amount it differs from the corresponding element in the previous block. Since these elements are the averages of their respective blocks, they should change slowly, so taking the differential values should reduce most of them to small values. No differentials are computed from the other values. The (0, 0) values are referred to as the DC components; the other values are the AC components.

Step 5 linearizes the 64 elements and applies run-length encoding to the list. Scanning the block from left to right and then top to bottom will not concentrate the zeros together, so a zigzag scanning pattern is used, as shown in Fig. 7-19. In this example, the zig zag pattern produces 38 consecutive 0s at the end of the matrix. This string can be reduced to a single count saying there are 38 zeros, a technique known as **run-length encoding**.

Now we have a list of numbers that represent the image (in transform space). Step 6 Huffman-encodes the numbers for storage or transmission, assigning common numbers shorter codes that uncommon ones.

JPEG may seem complicated, but that is because it *is* complicated. Still, since it often produces a 20:1 compression or better, it is widely used. Decoding a JPEG image requires running the algorithm backward. JPEG is roughly symmetric: decoding takes as long as encoding. This property is not true of all compression algorithms, as we shall now see.

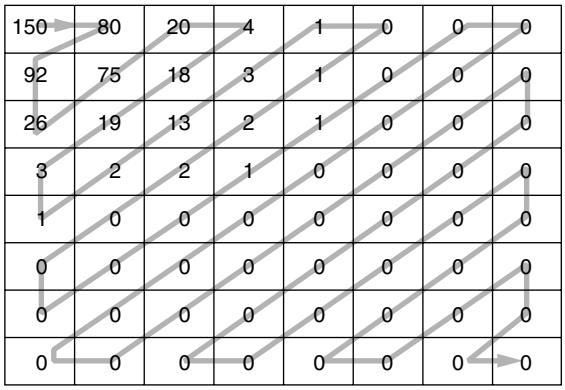

Figure 7-19. The order in which the quantized values are transmitted.

The MPEG Standard

Finally, we come to the heart of the matter: the **MPEG** (**Motion Picture Experts Group**) standards. These are the main algorithms used to compress videos and have been international standards since 1993. Because movies contain both images and sound, MPEG can compress both audio and video. We have already examined audio compression and still image compression, so let us now examine video compression.

The first standard to be finalized was MPEG-1 (International Standard 11172). Its goal was to produce video-recorder-quality output (352×240 for NTSC) using a bit rate of 1.2 Mbps. A 352×240 image with 24 bits/pixel and 25 frames/sec requires 50.7 Mbps, so getting it down to 1.2 Mbps is not entirely trivial. A factor of 40 compression is needed. MPEG-1 can be transmitted over twisted pair transmission lines for modest distances. MPEG-1 is also used for storing movies on CD-ROM.

The next standard in the MPEG family was MPEG-2 (International Standard 13818), which was originally designed for compressing broadcast-quality video into 4 to 6 Mbps, so it could fit in a NTSC or PAL broadcast channel. Later, MPEG-2 was expanded to support higher resolutions, including HDTV. It is very common now, as it forms the basis for DVD and digital satellite television.

The basic principles of MPEG-1 and MPEG-2 are similar, but the details are different. To a first approximation, MPEG-2 is a superset of MPEG-1, with additional features, frame formats, and encoding options. We will first discuss MPEG-1, then MPEG-2.

MPEG-1 has three parts: audio, video, and system, which integrates the other two, as shown in Fig. 7-20. The audio and video encoders work independently, which raises the issue of how the two streams get synchronized at the receiver.

This problem is solved by having a 90-kHz system clock that outputs the current time value to both encoders. These values are 33 bits, to allow films to run for 24 hours without wrapping around. These timestamps are included in the encoded output and propagated all the way to the receiver, which can use them to synchronize the audio and video streams.

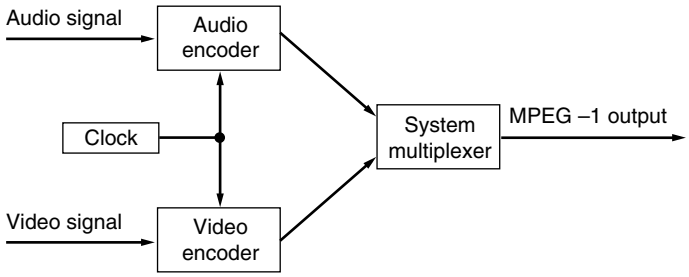

Figure 7-20. Synchronization of the audio and video streams in MPEG-1.

Now let us consider MPEG-1 video compression. Two kinds of redundancies exist in movies: spatial and temporal. MPEG-1 uses both. Spatial redundancy can be utilized by simply coding each frame separately with JPEG. This approach is occasionally used, especially when random access to each frame is needed, as in editing video productions. In this mode, a compressed bandwidth in the 8- to 10-Mbps range is achievable.

Additional compression can be achieved by taking advantage of the fact that consecutive frames are often almost identical. This effect is smaller than it might first appear since many moviemakers cut between scenes every 3 or 4 seconds (time a movie and count the scenes). Nevertheless, even a run of 75 highly similar frames offers the potential of a major reduction over simply encoding each frame separately with JPEG.

For scenes in which the camera and background are stationary and one or two actors are moving around slowly, nearly all the pixels will be identical from frame to frame. Here, just subtracting each frame from the previous one and running JPEG on the difference would do fine. However, for scenes where the camera is panning or zooming, this technique fails badly. What is needed is some way to compensate for this motion. This is precisely what MPEG does; it is the main difference between MPEG and JPEG.

MPEG-1 output consists of four kinds of frames:

1. I (Intracoded) frames: Self-contained JPEG-encoded still pictures.
2. P (Predictive) frames: Block-by-block difference with the last frame.
3. B (Bidirectional) frames: Differences between the last and next frame.
4. D (DC-coded) frames: Block averages used for fast forward.

I-frames are just still pictures coded using a variant of JPEG, also using full-resolution luminance and half-resolution chrominance along each axis. It is necessary to have I-frames appear in the output stream periodically for three reasons. First, MPEG-1 can be used for a multicast transmission, with viewers tuning it at will. If all frames depended on their predecessors going back to the first frame, anybody who missed the first frame could never decode any subsequent frames. Second, if any frame were received in error, no further decoding would be possible. Third, without I-frames, while doing a fast forward or rewind, the decoder would have to calculate every frame passed over so it would know the full value of the one it stopped on. For these reasons, I-frames are inserted into the output once or twice per second.

P-frames, in contrast, code interframe differences. They are based on the idea of **macroblocks**, which cover 16×16 pixels in luminance space and 8×8 pixels in chrominance space. A macroblock is encoded by searching the previous frame for it or something only slightly different from it.

An example of where P-frames would be useful is given in Fig. 7-21. Here we see three consecutive frames that have the same background, but differ in the position of one person. The macroblocks containing the background scene will match exactly, but the macroblocks containing the person will be offset in position by some unknown amount and will have to be tracked down.

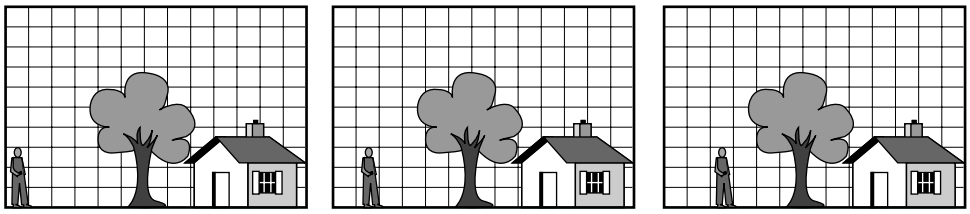

Figure 7-21. Three consecutive frames.

The MPEG-1 standard does not specify how to search, how far to search, or how good a match has to be to count. This is up to each implementation. For example, an implementation might search for a macroblock at the current position in the previous frame, and all other positions offset $\pm\Delta x$ in the x direction and $\pm\Delta y$ in the y direction. For each position, the number of matches in the luminance matrix could be computed. The position with the highest score would be declared the winner, provided it was above some predefined threshold. Otherwise, the macroblock would be said to be missing. Much more sophisticated algorithms are also possible, of course.

If a macroblock is found, it is encoded by taking the difference with its value in the previous frame (for luminance and both chrominances). These difference matrices are then subject to the discrete cosine transformation, quantization, run-length encoding, and Huffman encoding, just as with JPEG. The value for the

macroblock in the output stream is then the motion vector (how far the macroblock moved from its previous position in each direction), followed by the Huffman-encoded list of numbers. If the macroblock is not located in the previous frame, the current value is encoded with JPEG, just as in an I-frame.

Clearly, this algorithm is highly asymmetric. An implementation is free to try every plausible position in the previous frame if it wants to, in a desperate attempt to locate every last macroblock, no matter where it moved to. This approach will minimize the encoded MPEG-1 stream at the expense of very slow encoding. This approach might be fine for a one-time encoding of a film library but would be terrible for real-time videoconferencing.

Similarly, each implementation is free to decide what constitutes a "found" macroblock. This freedom allows implementers to compete on the quality and speed of their algorithms, but always produce compliant MPEG-1. No matter what search algorithm is used, the final output is either the JPEG encoding of the current macroblock or the JPEG encoding of the difference between the current macroblock and one in the previous frame at a specified offset from the current one.

So far, decoding MPEG-1 is straightforward. Decoding I-frames is the same as decoding JPEG images. Decoding P-frames requires the decoder to buffer the previous frame and then build up the new one in a second buffer based on fully encoded macroblocks and macroblocks containing differences from the previous frame. The new frame is assembled macroblock by macroblock.

B-frames are similar to P-frames, except that they allow the reference macroblock to be in either a previous frame or in a succeeding frame. This additional freedom allows improved motion compensation and is also useful when objects pass in front of, or behind, other objects. To do B-frame encoding, the encoder needs to hold three decoded frames in memory at once: the past one, the current one, and the future one. Although B-frames give the best compression, not all implementations support them.

D-frames are only used to make it possible to display a low-resolution image when doing a rewind or fast forward. Doing the normal MPEG-1 decoding in real time is difficult enough. Expecting the decoder to do it when slewing through the video at ten times normal speed is asking a bit much. Instead, the D-frames are used to produce low-resolution images. Each D-frame entry is just the average value of one block, with no further encoding, making it easy to display in real time. This facility is important to allow people to scan through a video at high speed in search of a particular scene. The D-frames are generally placed just before the corresponding I-frames so if fast forwarding is stopped, it will be possible to start viewing at normal speed.

Having finished our treatment of MPEG-1, let us now move on to MPEG-2. MPEG-2 encoding is fundamentally similar to MPEG-1 encoding, with I-frames, P-frames, and B-frames. D-frames are not supported, however. Also, the discrete cosine transformation uses a 10×10 block instead of a 8×8 block, to give 50

percent more coefficients, hence better quality. Since MPEG-2 is targeted at broadcast television as well as DVD, it supports both progressive and interlaced images, in contrast to MPEG-1, which supports only progressive images. Other minor details also differ between the two standards.

Instead of supporting only one resolution level, MPEG-2 supports four: low (352×240), main (720×480), high-1440 (1440×1152), and high (1920×1080). Low resolution is for VCRs and backward compatibility with MPEG-1. Main is the normal one for NTSC broadcasting. The other two are for HDTV. For high-quality output, MPEG-2 usually runs at 4–8 Mbps.

7.4.8 Video on Demand

Video on demand is sometimes compared to an electronic video rental store. The user (customer) selects any one of a large number of available videos and takes it home to view. Only with video on demand, the selection is made at home using the television set's remote control, and the video starts immediately. No trip to the store is needed. Needless to say, implementing video on demand is a wee bit more complicated than describing it. In this section, we will give an overview of the basic ideas and their implementation.

Is video on demand really like renting a video, or is it more like picking a movie to watch from a 500-channel cable system? The answer has important technical implications. In particular, video rental users are used to the idea of being able to stop a video, make a quick trip to the kitchen or bathroom, and then resume from where the video stopped. Television viewers do not expect to put programs on pause.

If video on demand is going to compete successfully with video rental stores, it may be necessary to allow users to stop, start, and rewind videos at will. Giving users this ability virtually forces the video provider to transmit a separate copy to each one.

On the other hand, if video on demand is seen more as advanced television, then it may be sufficient to have the video provider start each popular video, say, every 10 minutes, and run these nonstop. A user wanting to see a popular video may have to wait up to 10 minutes for it to start. Although pause/resume is not possible here, a viewer returning to the living room after a short break can switch to another channel showing the same video but 10 minutes behind. Some material will be repeated, but nothing will be missed. This scheme is called **near video on demand**. It offers the potential for much lower cost, because the same feed from the video server can go to many users at once. The difference between video on demand and near video on demand is similar to the difference between driving your own car and taking the bus.

Watching movies on (near) demand is but one of a vast array of potential new services possible once wideband networking is available. The general model that

many people use is illustrated in Fig. 7-22. Here we see a high-bandwidth (national or international) wide area backbone network at the center of the system. Connected to it are thousands of local distribution networks, such as cable TV or telephone company distribution systems. The local distribution systems reach into people's houses, where they terminate in **set-top boxes**, which are, in fact, powerful, specialized personal computers.

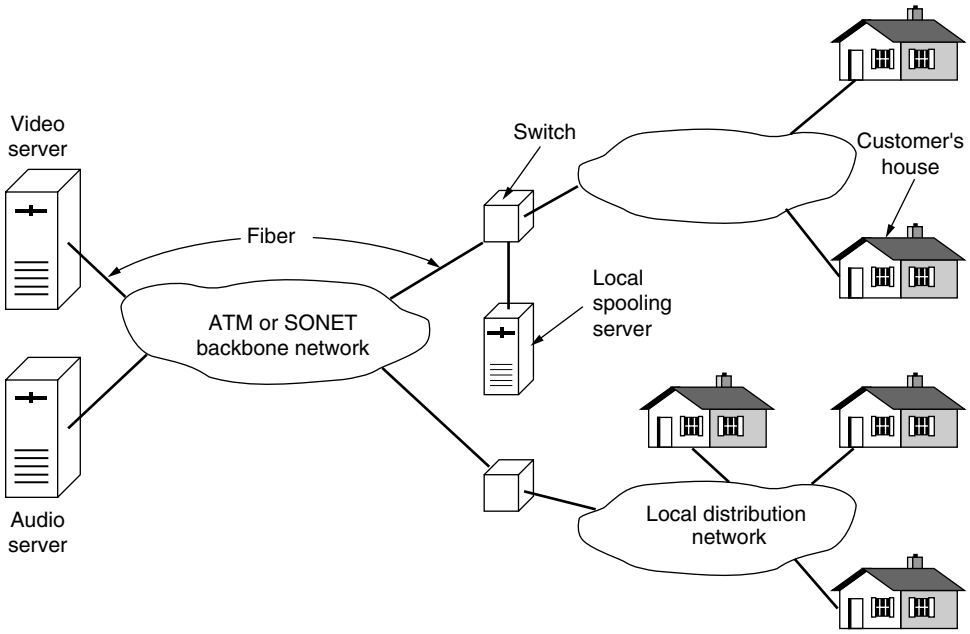

Figure 7-22. Overview of a video-on-demand system.

Attached to the backbone by high-bandwidth optical fibers are numerous information providers. Some of these will offer pay-per-view video or pay-per-hear audio CDs. Others will offer specialized services, such as home shopping (letting viewers rotate a can of soup and zoom in on the list of ingredients or view a video clip on how to drive a gasoline-powered lawn mower). Sports, news, reruns of "I Love Lucy," WWW access, and innumerable other possibilities will no doubt quickly become available.

Also included in the system are local spooling servers that allow videos to be placed closer to the users (in advance), to save bandwidth during peak hours. How these pieces will fit together and who will own what are matters of vigorous debate within the industry. Below we will examine the design of the main pieces of the system: the video servers and the distribution network.

Video Servers

To have (near) video on demand, we need **video servers** capable of storing and outputting a large number of movies simultaneously. The total number of movies ever made is estimated at 65,000 (Minoli, 1995). When compressed in MPEG-2, a normal movie occupies roughly 4 GB of storage, so 65,000 of them would require something like 260 terabytes. Add to this all the old television programs ever made, sports films, newsreels, talking shopping catalogs, etc., and it is clear that we have an industrial-strength storage problem on our hands.

The cheapest way to store large volumes of information is on magnetic tape. This has always been the case and probably always will be. An Ultrium tape can store 200 GB (50 movies) at a cost of about $1–$2 per movie. Large mechanical tape servers that hold thousands of tapes and have a robot arm for fetching any tape and inserting it into a tape drive are commercially available now. The problem with these systems is the access time (especially for the 50th movie on a tape), the transfer rate, and the limited number of tape drives (to serve n movies at once, the unit would need n drives).

Fortunately, experience with video rental stores, public libraries, and other such organizations shows that not all items are equally popular. Experimentally, when N movies are available, the fraction of all requests being for the kth most popular one is approximately C/k. Here C is computed to normalize the sum to 1, namely,

$$C = 1/(1 + 1/2 + 1/3 + 1/4 + 1/5 + \cdots + 1/N)$$

Thus, the most popular movie is seven times as popular as the number seven movie. This result is known as **Zipf's law** (Zipf, 1949).

The fact that some movies are much more popular than others suggests a possible solution in the form of a storage hierarchy, as shown in Fig. 7-23. Here, the performance increases as one moves up the hierarchy.

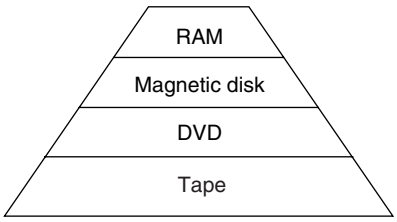

Figure 7-23. A video server storage hierarchy.

An alternative to tape is optical storage. Current DVDs hold 4.7 GB, good for one movie, but the next generation will hold two movies. Although seek times are slow compared to magnetic disks (50 msec versus 5 msec), their low cost and

high reliability make optical juke boxes containing thousands of DVDs a good alternative to tape for the more heavily used movies.

Next come magnetic disks. These have short access times (5 msec), high transfer rates (320 MB/sec for SCSI 320), and substantial capacities (> 100 GB), which makes them well suited to holding movies that are actually being transmitted (as opposed to just being stored in case somebody ever wants them). Their main drawback is the high cost for storing movies that are rarely accessed.

At the top of the pyramid of Fig. 7-23 is RAM. RAM is the fastest storage medium, but also the most expensive. When RAM prices drop to $50/GB, a 4-GB movie will occupy $200 dollars worth of RAM, so having 100 movies in RAM will cost $20,000 for the 200 GB of memory. Still, for a video server feeding out 100 movies, just keeping all the movies in RAM is beginning to look feasible. And if the video server has 100 customers but they are collectively watching only 20 different movies, it begins to look not only feasible, but a good design.

Since a video server is really just a massive real-time I/O device, it needs a different hardware and software architecture than a PC or a UNIX workstation. The hardware architecture of a typical video server is illustrated in Fig. 7-24. The server has one or more high-performance CPUs, each with some local memory, a shared main memory, a massive RAM cache for popular movies, a variety of storage devices for holding the movies, and some networking hardware, normally an optical interface to a SONET or ATM backbone at OC-12 or higher. These subsystems are connected by an extremely high speed bus (at least 1 GB/sec).

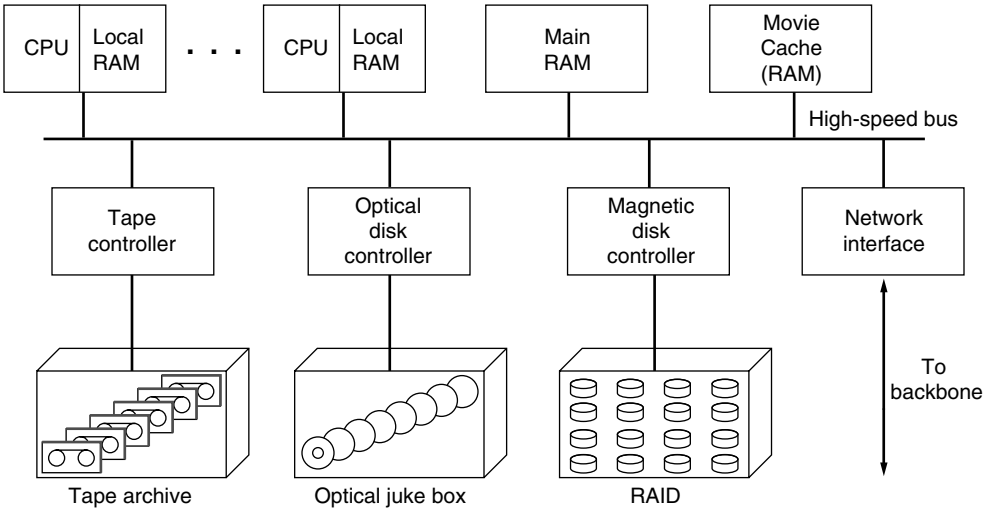

Figure 7-24. The hardware architecture of a typical video server.

Now let us take a brief look at video server software. The CPUs are used for accepting user requests, locating movies, moving data between devices, customer

billing, and many other functions. Some of these are not time critical, but many others are, so some, if not all, the CPUs will have to run a real-time operating system, such as a real-time microkernel. These systems normally break work up into small tasks, each with a known deadline. The scheduler can then run an algorithm such as nearest deadline next or the rate monotonic algorithm (Liu and Layland, 1973).

The CPU software also defines the nature of the interface that the server presents to the clients (spooling servers and set-top boxes). Two designs are popular. The first one is a traditional file system, in which the clients can open, read, write, and close files. Other than the complications introduced by the storage hierarchy and real-time considerations, such a server can have a file system modeled after that of UNIX.

The second kind of interface is based on the video recorder model. The commands to the server request it to open, play, pause, fast forward, and rewind files. The difference with the UNIX model is that once a *PLAY* command is given, the server just keeps pumping out data at a constant rate, with no new commands required.

The heart of the video server software is the disk management software. It has two main jobs: placing movies on the magnetic disk when they have to be pulled up from optical or tape storage, and handling disk requests for the many output streams. Movie placement is important because it can greatly affect performance.

Two possible ways of organizing disk storage are the disk farm and the disk array. With the **disk farm**, each drive holds some number of entire movies. For performance and reliability reasons, each movie should be present on at least two drives, maybe more. The other storage organization is the **disk array** or **RAID** (**Redundant Array of Inexpensive Disks**), in which each movie is spread out over multiple drives, for example, block 0 on drive 0, block 1 on drive 1, and so on, with block $n - 1$ on drive $n - 1$. After that, the cycle repeats, with block n on drive 0, and so forth. This organizing is called **striping**.

A striped disk array has several advantages over a disk farm. First, all n drives can be running in parallel, increasing the performance by a factor of n. Second, it can be made redundant by adding an extra drive to each group of n, where the redundant drive contains the block-by-block exclusive OR of the other drives, to allow full data recovery in the event one drive fails. Finally, the problem of load balancing is solved (manual placement is not needed to avoid having all the popular movies on the same drive). On the other hand, the disk array organization is more complicated than the disk farm and highly sensitive to multiple failures. It is also ill-suited to video recorder operations such as rewinding or fast forwarding a movie.

The other job of the disk software is to service all the real-time output streams and meet their timing constraints. Only a few years ago, this required complex disk scheduling algorithms, but with memory prices so low now, a much simpler

approach is beginning to be possible. For each stream being served, a buffer of, say, 10 sec worth of video (5 MB) is kept in RAM. It is filled by a disk process and emptied by a network process. With 500 MB of RAM, 100 streams can be fed directly from RAM. Of course, the disk subsystem must have a sustained data rate of 50 MB/sec to keep the buffers full, but a RAID built from high-end SCSI disks can handle this requirement easily.

The Distribution Network

The distribution network is the set of switches and lines between the source and destination. As we saw in Fig. 7-22, it consists of a backbone, connected to a local distribution network. Usually, the backbone is switched and the local distribution network is not.

The main requirement imposed on the backbone is high bandwidth. It used to be that low jitter was also a requirement, but with even the smallest PC now able to buffer 10 sec of high-quality MPEG-2 video, low jitter is not a requirement anymore.

Local distribution is highly chaotic, with different companies trying out different networks in different regions. Telephone companies, cable TV companies, and new entrants, such as power companies, are all convinced that whoever gets there first will be the big winner. Consequently, we are now seeing a proliferation of technologies being installed. In Japan, some sewer companies are in the Internet business, arguing that they have the biggest pipe of all into everyone's house (they run an optical fiber through it, but have to be very careful about precisely where it emerges). The four main local distribution schemes for video on demand go by the acronyms ADSL, FTTC, FTTH, and HFC. We will now explain each of these in turn.

ADSL is the first telephone industry's entrant in the local distribution sweepstakes. We studied ADSL in Chap. 2 and will not repeat that material here. The idea is that virtually every house in the United States, Europe, and Japan already has a copper twisted pair going into it (for analog telephone service). If these wires could be used for video on demand, the telephone companies could clean up.

The problem, of course, is that these wires cannot support even MPEG-1 over their typical 10-km length, let alone MPEG-2. High-resolution, full-color, full motion video needs 4–8 Mbps, depending on the quality desired. ADSL is not really fast enough except for very short local loops.

The second telephone company design is **FTTC** (**Fiber To The Curb**). In FTTC, the telephone company runs optical fiber from the end office into each residential neighborhood, terminating in a device called an **ONU** (**Optical Network Unit**). On the order of 16 copper local loops can terminate in an ONU. These loops are now so short that it is possible to run full-duplex T1 or T2 over

them, allowing MPEG-1 and MPEG-2 movies, respectively. In addition, videoconferencing for home workers and small businesses is now possible because FTTC is symmetric.

The third telephone company solution is to run fiber into everyone's house. It is called **FTTH** (**Fiber To The Home**). In this scheme, everyone can have an OC-1, OC-3, or even higher carrier if that is required. FTTH is very expensive and will not happen for years but clearly will open a vast range of new possibilities when it finally happens. In Fig. 7-7 we saw how everybody could operate his or her own radio station. What do you think about each member of the family operating his or her own personal television station? ADSL, FTTC, and FTTH are all point-to-point local distribution networks, which is not surprising given how the current telephone system is organized.

A completely different approach is **HFC** (**Hybrid Fiber Coax**), which is the preferred solution currently being installed by cable TV providers. It is illustrated in Fig. 2-47(a). The story goes something like this. The current 300- to 450-MHz coax cables are being replaced by 750-MHz coax cables, upgrading the capacity from 50 to 75 6-MHz channels to 125 6-MHz channels. Seventy-five of the 125 channels will be used for transmitting analog television.

The 50 new channels will each be modulated using QAM-256, which provides about 40 Mbps per channel, giving a total of 2 Gbps of new bandwidth. The headends will be moved deeper into the neighborhoods so that each cable runs past only 500 houses. Simple division shows that each house can then be allocated a dedicated 4-Mbps channel, which can handle an MPEG-2 movie.

While this sounds wonderful, it does require the cable providers to replace all the existing cables with 750-MHz coax, install new headends, and remove all the one-way amplifiers—in short, replace the entire cable TV system. Consequently, the amount of new infrastructure here is comparable to what the telephone companies need for FTTC. In both cases the local network provider has to run fiber into residential neighborhoods. Again, in both cases, the fiber terminates at an optoelectrical converter. In FTTC, the final segment is a point-to-point local loop using twisted pairs. In HFC, the final segment is a shared coaxial cable. Technically, these two systems are not really as different as their respective proponents often make out.

Nevertheless, there is one real difference that is worth pointing out. HFC uses a shared medium without switching and routing. Any information put onto the cable can be removed by any subscriber without further ado. FTTC, which is fully switched, does not have this property. As a result, HFC operators want video servers to send out encrypted streams so customers who have not paid for a movie cannot see it. FTTC operators do not especially want encryption because it adds complexity, lowers performance, and provides no additional security in their system. From the point of view of the company running a video server, is it a good idea to encrypt or not? A server operated by a telephone company or one of its subsidiaries or partners might intentionally decide not to encrypt its videos,

claiming efficiency as the reason but really to cause economic losses to its HFC competitors.

For all these local distribution networks, it is possible that each neighborhood will be outfitted with one or more spooling servers. These are, in fact, just smaller versions of the video servers we discussed above. The big advantage of these local servers is that they move some load off the backbone.

They can be preloaded with movies by reservation. If people tell the provider which movies they want well in advance, they can be downloaded to the local server during off-peak hours. This observation is likely to lead the network operators to lure away airline executives to do their pricing. One can envision tariffs in which movies ordered 24 to 72 hours in advance for viewing on a Tuesday or Thursday evening before 6 P.M, or after 11 P.M. get a 27 percent discount. Movies ordered on the first Sunday of the month before 8 A.M. for viewing on a Wednesday afternoon on a day whose date is a prime number get a 43 percent discount, and so on.

7.4.9 The MBone—The Multicast Backbone

While all these industries are making great—and highly publicized—plans for future (inter)national digital video on demand, the Internet community has been quietly implementing its own digital multimedia system, **MBone** (**Multicast Backbone**). In this section we will give a brief overview of what it is and how it works.

MBone can be thought of as Internet television. Unlike video on demand, where the emphasis is on calling up and viewing precompressed movies stored on a server, MBone is used for broadcasting live video in digital form all over the world via the Internet. It has been operational since early 1992. Many scientific conferences, especially IETF meetings, have been broadcast, as well as newsworthy scientific events, such as space shuttle launches. A Rolling Stones concert was once broadcast over MBone as were portions of the Cannes Film Festival. Whether this qualifies as a newsworthy scientific event is arguable.

Technically, MBone is a virtual overlay network on top of the Internet. It consists of multicast-capable islands connected by tunnels, as shown in Fig. 7-25. In this figure, MBone consists of six islands, *A* through *F*, connected by seven tunnels. Each island (typically a LAN or group of interconnected LANs) supports hardware multicast to its hosts. The tunnels propagate MBone packets between the islands. Some day in the future, when all the routers are capable of handling multicast traffic directly, this superstructure will no longer be needed, but for the moment, it does the job.

Each island contains one or more special routers called **mrouters** (**multicast routers**). Some of these are actually normal routers, but most are just UNIX workstations running special user-level software (but as the root). The mrouters

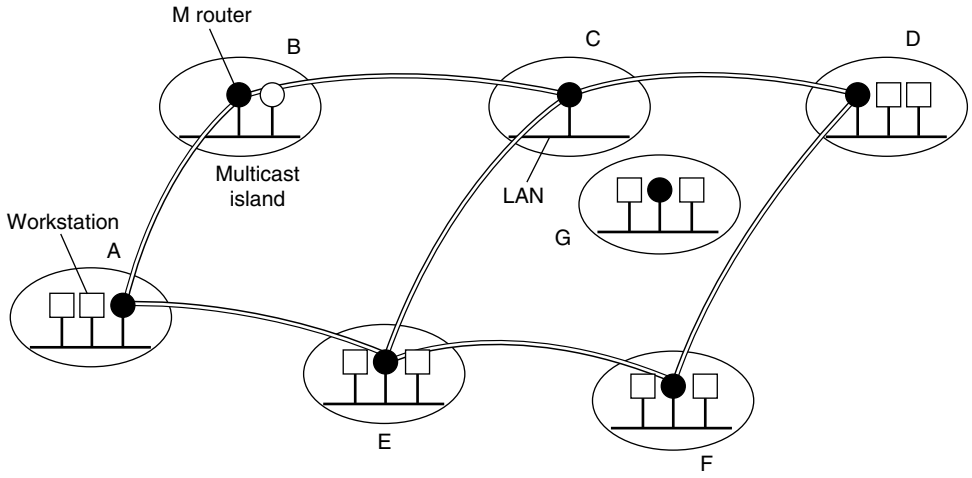

Figure 7-25. MBone consists of multicast islands connected by tunnels.

are logically connected by tunnels. MBone packets are encapsulated within IP packets and sent as regular unicast packets to the destination mrouter's IP address.

Tunnels are configured manually. Usually, a tunnel runs above a path for which a physical connection exists, but this is not a requirement. If, by accident, the physical path underlying a tunnel goes down, the mrouters using the tunnel will not even notice it, since the Internet will automatically reroute all the IP traffic between them via other lines.

When a new island appears and wishes to join MBone, such as *G* in Fig. 7-25, its administrator sends a message announcing its existence to the MBone mailing list. The administrators of nearby sites then contact him to arrange to set up tunnels. Sometimes existing tunnels are reshuffled to take advantage of the new island to optimize the topology. After all, tunnels have no physical existence. They are defined by tables in the mrouters and can be added, deleted, or moved simply by changing these tables. Typically, each country on MBone has a backbone, with regional islands attached to it. Normally, MBone is configured with one or two tunnels crossing the Atlantic and Pacific oceans, making MBone global in scale.

Thus, at any instant, MBone consists of a specific topology consisting of islands and tunnels, independent of the number of multicast addresses currently in use and who is listening to them or watching them. This situation is very similar to a normal (physical) subnet, so the normal routing algorithms apply to it. Consequently, MBone initially used a routing algorithm, **DVMRP** (**Distance Vector Multicast Routing Protocol**) based on the Bellman-Ford distance vector algorithm. For example, in Fig. 7-25, island *C* can route to *A* either via *B* or via *E* (or conceivably via *D*). It makes its choice by taking the values those nodes give it

about their respective distances to *A* and then adding its distance to them. In this way, every island determines the best route to every other island. The routes are not actually used in this way, however, as we will see shortly.

Now let us consider how multicasting actually happens. To multicast an audio or video program, a source must first acquire a class D multicast address, which acts like a station frequency or channel number. Class D addresses are reserved by a program that looks in a database for free multicast addresses. Many multicasts may be going on at once, and a host can "tune" to the one it is interested in by listening to the appropriate multicast address.

Periodically, each mrouter sends out an IGMP broadcast packet limited to its island asking who is interested in which channel. Hosts wishing to (continue to) receive one or more channels send another IGMP packet back in response. These responses are staggered in time, to avoid overloading the local LAN. Each mrouter keeps a table of which channels it must put out onto its LAN, to avoid wasting bandwidth by multicasting channels that nobody wants.

Multicasts propagate through MBone as follows. When an audio or video source generates a new packet, it multicasts it to its local island, using the hardware multicast facility. This packet is picked up by the local mrouter, which then copies it into all the tunnels to which it is connected.

Each mrouter getting such a packet via a tunnel then checks to see if the packet came along the best route, that is, the route that its table says to use to reach the source (as if it were a destination). If the packet came along the best route, the mrouter copies the packet to all its other tunnels. If the packet arrived via a suboptimal route, it is discarded. Thus, for example, in Fig. 7-25, if *C*'s tables tell it to use *B* to get to *A*, then when a multicast packet from *A* reaches *C* via *B*, the packet is copied to *D* and *E*. However, when a multicast packet from *A* reaches *C* via *E* (not the best path), it is simply discarded. This algorithm is just the reverse path forwarding algorithm that we saw in Chap. 5. While not perfect, it is fairly good and very simple to implement.

In addition to using reverse path forwarding to prevent flooding the Internet, the IP *Time to live* field is also used to limit the scope of multicasting. Each packet starts out with some value (determined by the source). Each tunnel is assigned a weight. A packet is passed through a tunnel only if it has enough weight. Otherwise it is discarded. For example, transoceanic tunnels are normally configured with a weight of 128, so packets can be limited to the continent of origin by being given an initial *Time to live* of 127 or less. After passing through a tunnel, the *Time to live* field is decremented by the tunnel's weight.

While the MBone routing algorithm works, much research has been devoted to improving it. One proposal keeps the idea of distance vector routing, but makes the algorithm hierarchical by grouping MBone sites into regions and first routing to them (Thyagarajan and Deering, 1995).

Another proposal is to use a modified form of link state routing instead of distance vector routing. In particular, an IETF working group modified OSPF to

make it suitable for multicasting within a single autonomous system. The resulting multicast OSPF is called **MOSPF** (Moy, 1994). What the modifications do is have the full map built by MOSPF keep track of multicast islands and tunnels, in addition to the usual routing information. Armed with the complete topology, it is straightforward to compute the best path from every island to every other island using the tunnels. Dijkstra's algorithm can be used, for example.

A second area of research is inter-AS routing. Here an algorithm called **PIM** (**Protocol Independent Multicast**) was developed by another IETF working group. PIM comes in two versions, depending on whether the islands are dense (almost everyone wants to watch) or sparse (almost nobody wants to watch). Both versions use the standard unicast routing tables, instead of creating an overlay topology as DVMRP and MOSPF do.

In PIM-DM (dense mode), the idea is to prune useless paths. Pruning works as follows. When a multicast packet arrives via the "wrong" tunnel, a prune packet is sent back through the tunnel telling the sender to stop sending it packets from the source in question. When a packet arrives via the "right" tunnel, it is copied to all the other tunnels that have not previously pruned themselves. If all the other tunnels have pruned themselves and there is no interest in the channel within the local island, the mrouter sends a prune message back through the "right" channel. In this way, the multicast adapts automatically and only goes where it is wanted.

PIM-SM (spare mode), described in RFC 2362, works differently. The idea here is to prevent saturating the Internet because three people in Berkeley want to hold a conference call over a class D address. PIM-SM works by setting up rendezvous points. Each of the sources in a PIM-SM multicast group send their packets to the rendezvous points. Any site interested in joining up asks one of the rendezvous points to set up a tunnel to it. In this way, all PIM-SM traffic is transported by unicast instead of by multicast. PIM-SM is becoming more popular, and the MBone is migrating toward its use. As PIM-SM becomes more widely used, MOSPF is gradually disappearing. On the other hand, the MBone itself seems to be somewhat stagnant and will probably never catch on in a big way.

Nevertheless, networked multimedia is still an exciting and rapidly moving field, even if the MBone does not become a huge success. New technologies and applications are announced daily. Increasingly, multicasting and quality of service are coming together, as discussed in (Striegel and Manimaran, 2002). Another hot topic is wireless multicast (Gossain et al., 2002). The whole area of multicasting and everything related to it are likely to remain important for years to come.