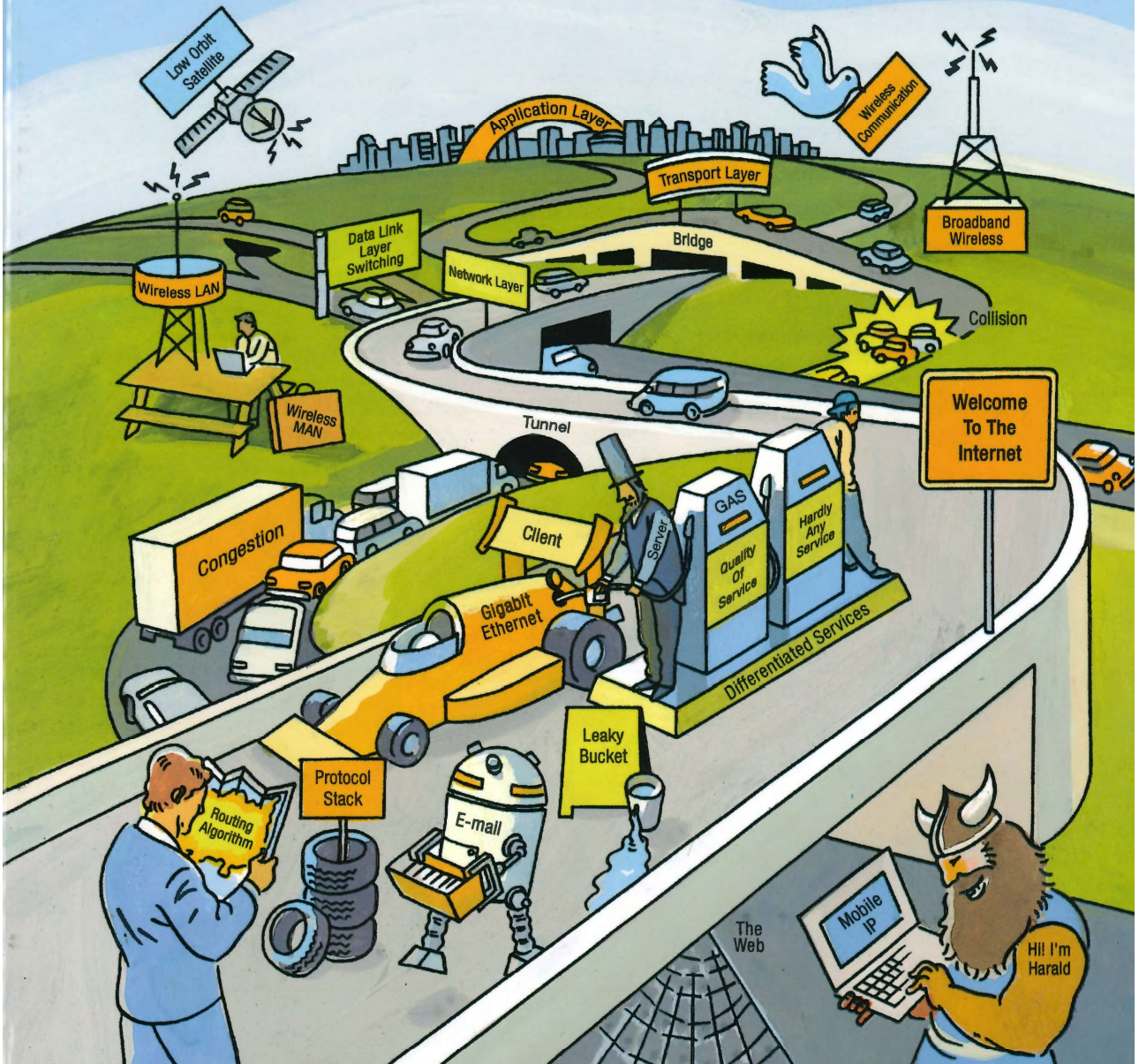


FOURTH EDITION

Computer Networks

ANDREW S. TANENBAUM



Computer Networks

Fourth Edition

Andrew S. Tanenbaum

*Vrije Universiteit
Amsterdam, The Netherlands*



Prentice Hall PTR
Upper Saddle River, NJ 07458
www.phptr.com

Library of Congress Cataloging-in-Publication Data

Tanenbaum, Andrew S.
Computer networks / Andrew S. Tanenbaum.--4th ed.
p. cm.
Includes bibliographical references.
ISBN 0-13-066102-3
1. Computer networks. I. Title.
TK5105.5 .T36 2002
004.6--dc21
2002029263

Editorial/production supervision: *Patti Guerrieri*
Cover design director: *Jerry Votta*
Cover designer: *Anthony Gemmellaro*
Cover design: *Andrew S. Tanenbaum*
Art director: *Gail Cocker-Bogusz*
Interior Design: *Andrew S. Tanenbaum*
Interior graphics: *Hadel Studio*
Typesetting: *Andrew S. Tanenbaum*
Manufacturing buyer: *Maura Zaldivar*
Executive editor: *Mary Franz*
Editorial assistant: *Noreen Regina*
Marketing manager: *Dan DePasquale*



© 2003, 1996 Pearson Education, Inc.
Publishing as Prentice Hall PTR
Upper Saddle River, New Jersey 07458

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale.

For information regarding corporate and government bulk discounts please contact:
Corporate and Government Sales (800) 382-3419 or corpsales@pearsontechgroup.com

All products or services mentioned in this book are the trademarks or service marks of their respective companies or organizations.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-066102-3

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd.
Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education — Japan
Pearson Education Malaysia, Pte. Ltd.

communicating via parameters. If the called procedure is now moved to a remote machine, the code will fail because the global variables are no longer shared.

These problems are not meant to suggest that RPC is hopeless. In fact, it is widely used, but some restrictions are needed to make it work well in practice.

Of course, RPC need not use UDP packets, but RPC and UDP are a good fit and UDP is commonly used for RPC. However, when the parameters or results may be larger than the maximum UDP packet or when the operation requested is not idempotent (i.e., cannot be repeated safely, such as when incrementing a counter), it may be necessary to set up a TCP connection and send the request over it rather than use UDP.

6.4.3 The Real-Time Transport Protocol

Client-server RPC is one area in which UDP is widely used. Another one is real-time multimedia applications. In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people discovered that each application was reinventing more or less the same real-time transport protocol. It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea. Thus was **RTP (Real-time Transport Protocol)** born. It is described in RFC 1889 and is now in widespread use.

The position of RTP in the protocol stack is somewhat strange. It was decided to put RTP in user space and have it (normally) run over UDP. It operates as follows. The multimedia application consists of multiple audio, video, text, and possibly other streams. These are fed into the RTP library, which is in user space along with the application. This library then multiplexes the streams and encodes them in RTP packets, which it then stuffs into a socket. At the other end of the socket (in the operating system kernel), UDP packets are generated and embedded in IP packets. If the computer is on an Ethernet, the IP packets are then put in Ethernet frames for transmission. The protocol stack for this situation is shown in Fig. 6-25(a). The packet nesting is shown in Fig. 6-25(b).

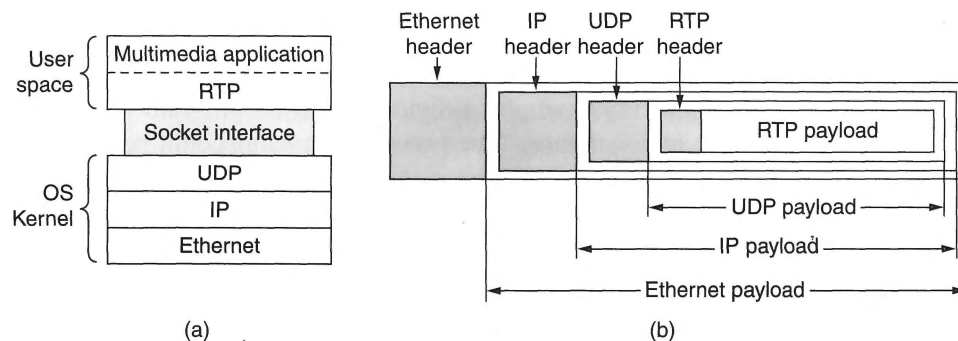


Figure 6-25. (a) The position of RTP in the protocol stack. (b) Packet nesting.

As a consequence of this design, it is a little hard to say which layer RTP is in. Since it runs in user space and is linked to the application program, it certainly looks like an application protocol. On the other hand, it is a generic, application-independent protocol that just provides transport facilities, so it also looks like a transport protocol. Probably the best description is that it is a transport protocol that is implemented in the application layer.

The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting). Because RTP just uses normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, jitter, etc.

Each packet sent in an RTP stream is given a number one higher than its predecessor. This numbering allows the destination to determine if any packets are missing. If a packet is missing, the best action for the destination to take is to approximate the missing value by interpolation. Retransmission is not a practical option since the retransmitted packet would probably arrive too late to be useful. As a consequence, RTP has no flow control, no error control, no acknowledgements, and no mechanism to request retransmissions.

Each RTP payload may contain multiple samples, and they may be coded any way that the application wants. To allow for interworking, RTP defines several profiles (e.g., a single audio stream), and for each profile, multiple encoding formats may be allowed. For example, a single audio stream may be encoded as 8-bit PCM samples at 8 kHz, delta encoding, predictive encoding, GSM encoding, MP3, and so on. RTP provides a header field in which the source can specify the encoding but is otherwise not involved in how encoding is done.

Another facility many real-time applications need is timestamping. The idea here is to allow the source to associate a timestamp with the first sample in each packet. The timestamps are relative to the start of the stream, so only the differences between timestamps are significant. The absolute values have no meaning. This mechanism allows the destination to do a small amount of buffering and play each sample the right number of milliseconds after the start of the stream, independently of when the packet containing the sample arrived. Not only does timestamping reduce the effects of jitter, but it also allows multiple streams to be synchronized with each other. For example, a digital television program might have a video stream and two audio streams. The two audio streams could be for stereo broadcasts or for handling films with an original language soundtrack and a soundtrack dubbed into the local language, giving the viewer a choice. Each stream comes from a different physical device, but if they are timestamped from a single counter, they can be played back synchronously, even if the streams are transmitted somewhat erratically.

The RTP header is illustrated in Fig. 6-26. It consists of three 32-bit words and potentially some extensions. The first word contains the *Version* field, which

is already at 2. Let us hope this version is very close to the ultimate version since there is only one code point left (although 3 could be defined as meaning that the real version was in an extension word).

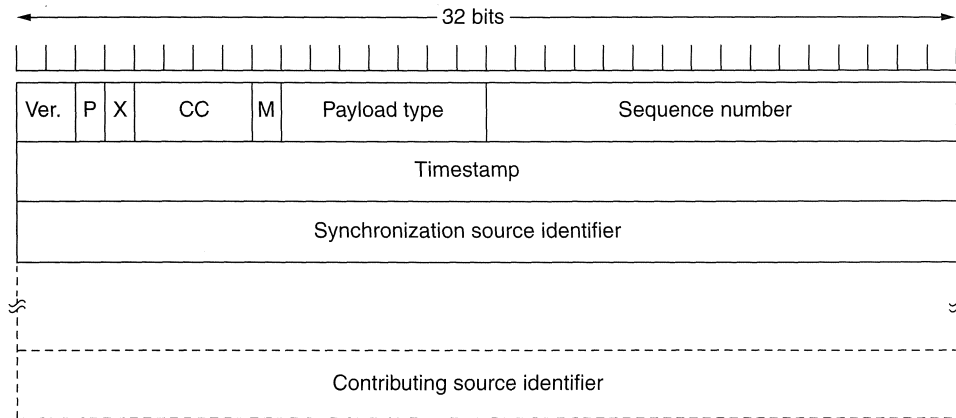


Figure 6-26. The RTP header.

The *P* bit indicates that the packet has been padded to a multiple of 4 bytes. The last padding byte tells how many bytes were added. The *X* bit indicates that an extension header is present. The format and meaning of the extension header are not defined. The only thing that is defined is that the first word of the extension gives the length. This is an escape hatch for any unforeseen requirements.

The *CC* field tells how many contributing sources are present, from 0 to 15 (see below). The *M* bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands. The *Payload type* field tells which encoding algorithm has been used (e.g., uncompressed 8-bit audio, MP3, etc.). Since every packet carries this field, the encoding can change during transmission. The *Sequence number* is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.

The timestamp is produced by the stream's source to note when the first sample in the packet was made. This value can help reduce jitter at the receiver by decoupling the playback from the packet arrival time. The *Synchronization source identifier* tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto a single stream of UDP packets. Finally, the *Contributing source identifiers*, if any, are used when mixers are present in the studio. In that case, the mixer is the synchronizing source, and the streams being mixed are listed here.

RTP has a little sister protocol (little sibling protocol?) called **RTCP (Real-time Transport Control Protocol)**. It handles feedback, synchronization, and the user interface but does not transport any data. The first function can be used

to provide feedback on delay, jitter, bandwidth, congestion, and other network properties to the sources. This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data rate when there is trouble in the network. By providing continuous feedback, the encoding algorithms can be continuously adapted to provide the best quality possible under the current circumstances. For example, if the bandwidth increases or decreases during the transmission, the encoding may switch from MP3 to 8-bit PCM to delta encoding as required. The *Payload type* field is used to tell the destination what encoding algorithm is used for the current packet, making it possible to vary it on demand.

RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync.

Finally, RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

More information about RTP can be found in (Perkins, 2002).

6.5 THE INTERNET TRANSPORT PROTOCOLS: TCP

UDP is a simple protocol and it has some niche uses, such as client-server interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet. Let us now study it in detail.

6.5.1 Introduction to TCP

TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

TCP was formally defined in RFC 793. As time went on, various errors and inconsistencies were detected, and the requirements were changed in some areas. These clarifications and some bug fixes are detailed in RFC 1122. Extensions are given in RFC 1323.

Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or part of the kernel. In all cases, it manages TCP streams and interfaces to the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64 KB (in practice, often

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. 7-59.

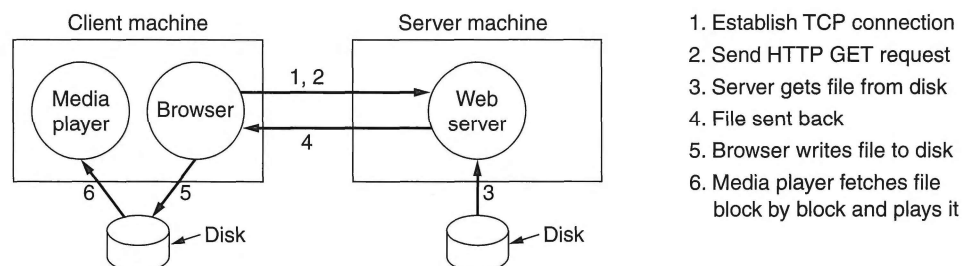


Figure 7-59. A straightforward way to implement clickable music on a Web page.

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. 7-60. 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.

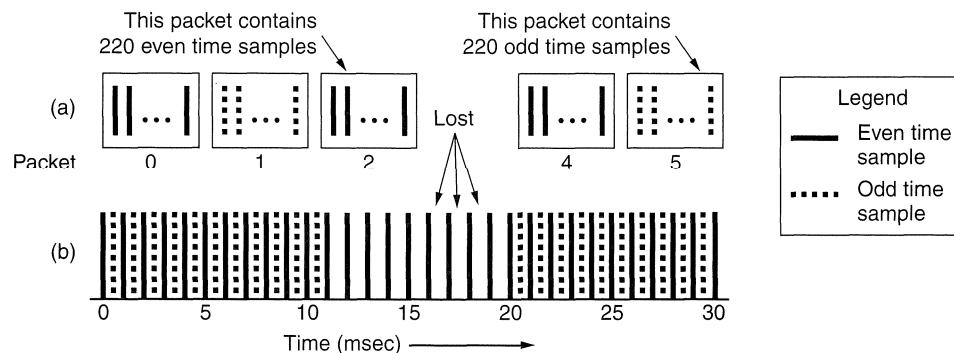


Figure 7-60. When packets carry alternate samples, the loss of a packet reduces the temporal resolution rather than creating a gap in time.

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. 7-61. 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

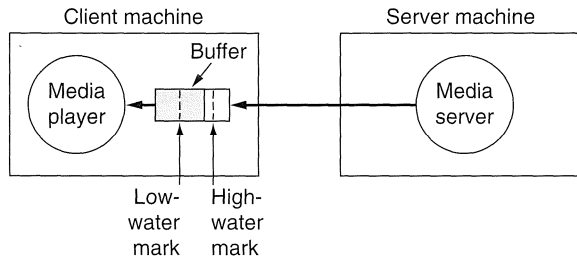


Figure 7-61. The media player buffers input from the media server and plays from the buffer rather than directly from the network.

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. 7-62.

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 7-62. 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. 7-63. 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.

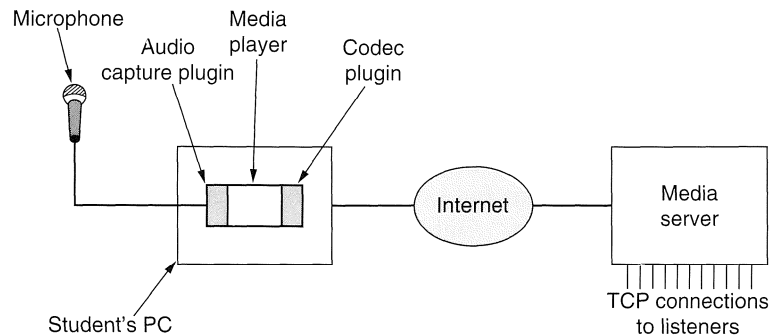


Figure 7-63. A student radio station.

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. 7-64. 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**.

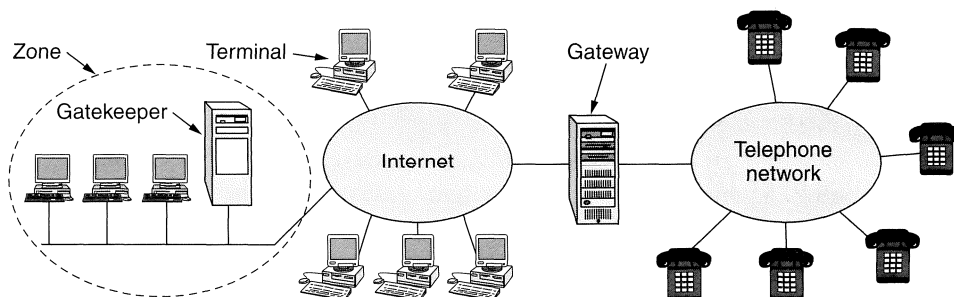


Figure 7-64. The H.323 architectural model for Internet telephony.

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. 7-65.

Speech	Control			
G.7xx	RTCP	H.225 (RAS)	Q.931 (Call signaling)	H.245 (Call control)
RTP				
UDP			TCP	
IP				
Data link protocol				
Physical layer protocol				

Figure 7-65. 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. 7-66. 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.

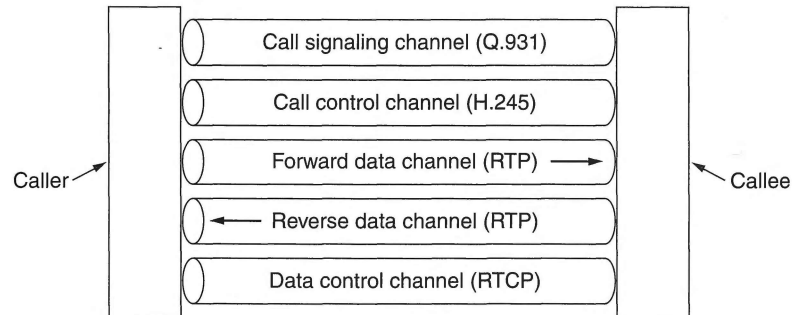


Figure 7-66. Logical channels between the caller and callee during a call.

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. 7-67.

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 7-67. 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. 7-42, 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. 7-68. 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.

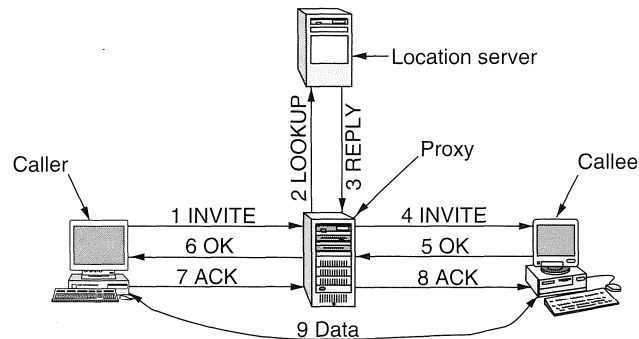


Figure 7-68. Use a proxy and redirection servers with SIP.

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. 7-69.

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 7-69. 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-