The TCP/IP protocol suite provides separate standards for mail message format and mail transfer. The mail message format, called *822*, uses a blank line to separate a message header and the body. The Simple Mail Transfer Protocol (SMTP) defines how a mail system on one machine transfers mail to a server on another. Version 3 of the Post Office Protocol (POP3) specifies how a user can retrieve the contents of a mailbox; it allows a user to have a permanent mailbox on a computer with continuous Internet connectivity and to access the contents from a computer with intermittent connectivity.

The Multipurpose Internet Mail Extensions (MIME) provides a mechanism that allows arbitrary data to be transferred using SMTP. MIME adds lines to the header of an e-mail message to define the type of the data and the encoding used. MIME's mixed multipart type permits a single message to contain multiple data types.

FOR FURTHER STUDY

The protocols described in this chapter are all specified in Internet RFCs. Postel [RFC 821] describes the Simple Mail Transfer Protocol and gives many examples. The exact format of mail messages is given by Crocker [RFC 822]; many RFCs specify additions and changes. Freed and Borenstein [RFCs 2045, 2046, 2047, 2048 and 2049] specify the standard for MIME, including the syntax of header declarations, the procedure for creating new content types, the interpretation of content types, and the *base64* encoding mentioned in this chapter. Partridge [RFC 974] discusses the relationship between mail routing and the domain name system. Horton [RFC 976] proposes a standard for the UNIX UUCP mail system.

EXERCISES

27.1 Some mail systems force the user to specify a sequence of machines through which the message should travel to reach its destination. The mail protocol in each machine merely passes the message on to the next machine. List three disadvantages of such a scheme.

27.2 Find out if your computing system allows you to invoke SMTP directly.

27.3 Build an SMTP client and use it to deliver a mail message.

27.4 See if you can send mail through a mail gateway and back to yourself.

27.5 Make a list of mail address forms that your site handles and write a set of rules for parsing them.

27.6 Find out how the UNIX *sendmail* program can be used to implement a mail gateway.

27.7 Find out how often your local mail system attempts delivery and how long it will continue before giving up.

27.8 Many mail systems allow users to direct incoming mail to a program instead of storing it in a mailbox. Build a program that accepts your incoming mail, places your mail in a file, and then sends a reply to tell the sender you are on vacation.

27.9 Read the SMTP standard carefully. Then use TELNET to connect to the SMTP port on a remote machine and ask the remote SMTP server to expand a mail alias.

27.10 A user receives mail in which the *To* field specifies the string *important-people*. The mail was sent from a computer on which the alias *important-people* includes no valid mailbox identifiers. Read the SMTP specification carefully to see how such a situation is possible.

27.11 POP3 separates message retrieval and deletion by allowing a user to retrieve and view a message without deleting it from the permanent mailbox. What are the advantages and disadvantages of such separation?

27.12 Read about POP3. How does the *TOP* command operate, and why is it useful?

27.13 Read the MIME standard carefully. What servers can be specified in a MIME external reference?

28

Applications: World Wide Web (HTTP)

28.1 Introduction

This chapter continues the discussion of applications that use TCP/IP technology by focusing on the application that has had the most impact: the *World Wide Web* (*WWW*). After a brief overview of concepts, the chapter examines the primary protocol used to transfer a Web page from a server to a Web browser. The discussion covers caching as well as the basic transfer mechanism.

28.2 Importance Of The Web

During the early history of the Internet, FTP data transfers accounted for approximately one third of Internet traffic, more than any other application. From its inception in the early 1990s, however, the Web had a much higher growth rate. By 1995, Web traffic overtook FTP to become the largest consumer of Internet backbone bandwidth, and has remained the leader ever since. By 2000, Web traffic completely overshadowed other applications.

Although traffic is easy to measure and cite, the impact of the Web cannot be understood from such statistics. More people know about and use the Web than any other Internet application. Most companies have Web sites and on-line catalogs; references to the Web appear in advertising. In fact, for many users, the Internet and the Web are indistinguishable.

28.3 Architectural Components

Conceptually, the Web consists of a large set of documents, called *Web pages*, that are accessible over the Internet. Each Web page is classified as a *hypermedia* document. The suffix *media* is used to indicate that a document can contain items other than text (e.g., graphics images); the prefix *hyper* is used because a document can contain *selectable links* that refer to other, related documents.

Two main building blocks are used to implement the Web on top of the global Internet. A *Web browser* consists of an application program that a user invokes to access and display a Web page. The browser becomes a client that contacts the appropriate *Web server* to obtain a copy of the specified page. Because a given server can manage more than one Web page, a browser must specify the exact page when making a request.

The data representation standard used for a Web page depends on its contents. For example, standard graphics representations such as *Graphics Interchange Format* (*GIF*) or *Joint Picture Encoding Group* (*JPEG*) can be used for a page that contains a single graphics image. Pages that contain a mixture of text and other items are represented using *HyperText Markup Language* (*HTML*). An HTML document consists of a file that contains text along with embedded commands, called *tags*, that give guidelines for display. A tag is enclosed in less-than and greater-than symbols; some tags come in pairs that apply to all items between the pair. For example, the two commands *<CENTER>* and *</CENTER>* cause items between them to be centered in the browser's window.

28.4 Uniform Resource Locators

Each Web page is assigned a unique name that is used to identify it. The name, which is called a *Uniform Resource Locator* (*URL*)†, begins with a specification of the *scheme* used to access the item. In effect, the scheme specifies the transfer protocol; the format of the remainder of the URL depends on the scheme. For example, a URL that follows the *http scheme* has the following form‡:

http: // *hostname [: port] / path [; parameters] [? query]*

where brackets denote an optional item. For now, it is sufficient to understand that the *hostname* string specifies the domain name or IP address of the computer on which the server for the item operates, *:port* is an optional protocol port number needed only in cases where the server does not use the well-known port (*80*), *path* is a string that identifies one particular document on the server, *;parameters* is an optional string that specifies additional parameters supplied by the client, and *?query* is an optional string used when the browser sends a question. A user is unlikely to ever see or use the optional parts directly. Instead, URLs that a user enters contain only a *hostname* and *path*. For example, the URL:

†A URL is a specific type of the more general *Uniform Resource Identifier* (*URI*).

‡Some of the literature refers to the initial string, *http:*, as a *pragma*.

http://www.cs.purdue.edu/people/comer/

specifies the author's Web page. The server operates on computer *www.cs.purdue.edu*, and the document is named */people/comer/*.

The protocol standards distinguish between the *absolute* form of a URL illustrated above, and a *relative* form. A relative URL, which is seldom seen by a user, is only meaningful when the server has already been determined. Relative URLs are useful once communication has been established with a specific server. For example, when communicating with server *www.cs.purdue.edu*, only the string */people/comer/* is needed to specify the document named by the absolute URL above. We can summarize.

> *Each Web page is assigned a unique identifier known as a Uniform Resource Locator (URL). The absolute form of a URL contains a full specification; a relative form that omits the address of the server is only useful when the server is implicitly known.*

28.5 An Example Document

In principle, Web access is straightforward. All access originates with a URL — a user either enters a URL via the keyboard or selects an item which provides the browser with a URL. The browser parses the URL, extracts the information, and uses it to obtain a copy of the requested page. Because the format of the URL depends on the scheme, the browser begins by extracting the scheme specification, and then uses the scheme to determine how to parse the rest of the URL.

An example will illustrate how a URL is produced from a *selectable link* in a document. In fact, a document contains a pair of values for each link: an item to be displayed on the screen and a URL to follow if the user selects the item. In HTML, the pair of tags <A> and </A> are known as an *anchor*. The anchor defines a link; a URL is added to the first tag, and items to be displayed are placed between the two tags. The browser stores the URL internally, and follows it when the user selects the link. For example, the following HTML document contains a selectable link:

```
<HTML>
  The author of this text is
  <A HREF="http://www.cs.purdue.edu/people/comer">
  Douglas Comer.</A>
</HTML>
```

When the document is displayed, a single line of text appears on the screen:

The author of this text is <u>Douglas Comer.</u>

The browser underlines the phrase *Douglas Comer* to indicate that it corresponds to a selectable link. Internally, of course, the browser stores the URL from the <*A*> tag, which it follows when the user selects the link.

28.6 Hypertext Transfer Protocol

The protocol used for communication between a browser and a Web server or between intermediate machines and Web servers is known as the *HyperText Transfer Protocol* (*HTTP*). HTTP has the following set of characteristics:

> *Application Level.* HTTP operates at the application level. It assumes a reliable, connection-oriented transport protocol such as TCP, but does not provide reliability or retransmission itself.
>
> *Request/Response.* Once a transport session has been established, one side (usually a browser) must send an HTTP request to which the other side responds.
>
> *Stateless.* Each HTTP request is self-contained; the server does not keep a history of previous requests or previous sessions.
>
> *Bi-Directional Transfer.* In most cases, a browser requests a Web page, and the server transfers a copy to the browser. HTTP also allows transfer from a browser to a server (e.g., when a user submits a so-called "form").
>
> *Capability Negotiation.* HTTP allows browsers and servers to negotiate details such as the character set to be used during transfers. A sender can specify the capabilities it offers and a receiver can specify the capabilities it accepts.
>
> *Support For Caching.* To improve response time, a browser caches a copy of each Web page it retrieves. If a user requests a page again, HTTP allows the browser to interrogate the server to determine whether the contents of the page has changed since the copy was cached.
>
> *Support For Intermediaries.* HTTP allows a machine along the path between a browser and a server to act as a *proxy server* that caches Web pages and answers a browser's request from its cache.

28.7 HTTP GET Request

In the simplest case, a browser contacts a Web server directly to obtain a page. The browser begins with a URL, extracts the hostname section, uses DNS to map the name into an equivalent IP address, and uses the IP address to form a TCP connection to the server. Once the TCP connection is in place, the browser and Web server use HTTP to communicate; the browser sends a request to retrieve a specific page, and the server responds by sending a copy of the page.

A browser sends an HTTP *GET* command to request a Web page from a server†. The request consists of a single line of text that begins with the keyword *GET* and is followed by a URL and an HTTP version number. For example, to retrieve the Web page in the example above from server *www.cs.purdue.edu*, a browser can send the following request:

```
GET http://www.cs.purdue.edu/people/comer/ HTTP/1.1
```

Once a TCP connection is in place, there is no need to send an absolute URL — the following relative URL will retrieve the same page:

```
GET /people/comer/ HTTP/1.0
```

To summarize:

The Hypertext Transfer Protocol (HTTP) is used between a browser and a Web server. The browser sends a GET request to which a server responds by sending the requested item.

28.8 Error Messages

How should a Web server respond when it receives an illegal request? In most cases, the request has been sent by a browser, and the browser will attempt to display whatever the server returns. Consequently, servers usually generate error messages in valid HTML. For example, one server generates the following error message:

```
<HTML>
  <HEAD> <TITLE>400 Bad Request</TITLE>
  </HEAD>
  <BODY>
    <H1>Bad Request</H1> Your browser sent a request
    that this server could not understand.
  </BODY>
</HTML>
```

The browser uses the "head" of the document (i.e., the items between <HEAD> and </HEAD>) internally, and only shows the "body" to the user. The pair of tags <H1> and </H1> causes the browser to display *Bad Request* as a heading (i.e., large and bold), resulting in two lines of output on the user's screen:

†The standard uses the object-oriented term *method* instead of *command*.

Bad Request

Your browser sent a request that this server could not understand.

28.9 Persistent Connections And Lengths

Early versions of HTTP follow the same paradigm as FTP by using a new TCP connection for each data transfer. That is, a client opens a TCP connection and sends a *GET* request. The server transmits a copy of the requested item, and then closes the TCP connection. Until it encounters an *end of file* condition, the client reads data from the TCP connection. Finally, the client closes its end of the connection.

Version 1.1, which appeared as an RFC in June of 1999, changed the basic HTTP paradigm in a fundamental way. Instead of using a TCP connection for each transfer, version 1.1 adopts a *persistent connection* approach as the default. That is, once a client opens a TCP connection to a particular server, the client leaves the connection in place during multiple requests and responses. When either a client or server is ready to close the connection, it informs the other side, and the connection is closed.

The chief advantage of persistent connections lies in reduced overhead — fewer TCP connections means lower response latency, less overhead on the underlying networks, less memory used for buffers, and less CPU time used. A browser using a persistent connection can further optimize by *pipelining* requests (i.e., send requests back-to-back without waiting for a response). Pipelining is especially attractive in situations where multiple images must be retrieved for a given page, and the underlying internet has both high throughput and long delay.

The chief disadvantage of using a persistent connection lies in the need to identify the beginning and end of each item sent over the connection. There are two possible techniques that handle the situation: either send a length followed by the item, or send a *sentinel value* after the item to mark the end. HTTP cannot reserve a sentinel value because the items transmitted include graphics images that can contain arbitrary sequences of octets. Thus, to avoid ambiguity between sentinel values and data, HTTP uses the approach of sending a length followed by an item of that size.

28.10 Data Length And Program Output

It may not be convenient or even possible for a server to know the length of an item before sending. To understand why, one must know that servers use the *Common Gateway Interface* (*CGI*) mechanism that allows a computer program running on the server machine to create a Web page dynamically. When a request arrives that corresponds to one of the CGI-generated pages, the server runs the appropriate CGI program, and sends the output from the program back to the client as a response. Dynamic Web page generation allows the creation of information that is current (e.g., a list of the current scores in sporting events), but means that the server may not know the exact data size in advance. Furthermore, saving the data to a file before sending it is undesirable for two reasons: it uses resources at the server and delays transmission. Thus, to provide for dynamic Web pages, the HTTP standard specifies that if the server does not know the length of an item *a priori*, the server can inform the browser that it will close the connection after transmitting the item. To summarize:

To allow a TCP connection to persist through multiple requests and responses, HTTP sends a length before each response. If it does not know the length, a server informs the client, sends the response, and then closes the connection.

28.11 Length Encoding And Headers

What representation does a server use to send length information? Interestingly, HTTP borrows the basic format from e-mail, using 822 format and MIME Extensions†. Like a standard 822 message, each HTTP transmission contains a header, a blank line, and the item being sent. Furthermore, each line in the header contains a keyword, a colon, and information. Figure 28.2 lists a few of the possible headers and their meaning.

Header	Meaning
Content-Length	**Size of item in octets**
Content-Type	**Type of the item**
Content-Encoding	**Encoding used for item**
Content-Language	**Language(s) used in item**

Figure 28.1 Examples of items that can appear in the header sent before an item. The *Content-Type* and *Content-Encoding* are taken directly from MIME.

As an example, consider Figure 28.2 which shows a few of the headers that are used when a HTML document is transferred across a persistent TCP connection.

```
Content-Length:   34
Content-Language: en
Content-Encoding: ascii

<HTML> A trivial example. </HTML>
```

Figure 28.2 An illustration of an HTTP transfer with header lines used to specify attributes, a blank line, and the document itself. A *Content-Length* header is required if the connection is persistent.

†See Chapter 27 for a discussion of e-mail, 822 format, and MIME.

In addition to the examples shown in the figure, HTTP includes a wide variety of headers that allow a browser and server to exchange meta information. For example, we said that if a server does not know the length of an item, the server closes the connection after sending the item. However, the server does not act without warning — the server informs the browser to expect a close. To do so, the server includes a *Connection* header before the item in place of a *Content-Length* header:

Connection: close

When it receives a connection header, the browser knows that the server intends to close the connection after the transfer; the browser is forbidden from sending further requests. The next sections describe the purposes of other headers.

28.12 Negotiation

In addition to specifying details about an item being sent, HTTP uses headers to permit a client and server to *negotiate* capabilities. The set of negotiable capabilities includes a wide variety of characteristics about the connection (e.g., whether access is authenticated), representation (e.g., whether graphics images in jpeg format are acceptable or which types of compression can be used), content (e.g., whether text files must be in English), and control (e.g., the length of time a page remains valid).

There are two basic types of negotiation: *server-driven* and *agent-driven* (i.e., browser-driven). Server-driven negotiation begins with a request from a browser. The request specifies a list of preferences along with the URL of the desired item. The server selects, from among the available representations, one that satisfies the browser's preferences. If multiple items satisfy the preferences, the server makes a "best guess." For example, if a document is stored in multiple languages and a request specifies a preference for English, the server will send the English version.

Agent-driven negotiation simply means that a browser uses a two-step process to perform the selection. First, the browser sends a request to the server to ask what is available. The server returns a list of possibilities. The browser selects one of the possibilities, and sends a second request to obtain the item. The disadvantage of agent-driven negotiation is that it requires two server interactions; the advantage is that a browser retains complete control over the choice.

A browser uses an HTTP *Accept* header to specify which media or representations are acceptable. The header lists names of formats with a preference value assigned to each. For example,

```
Accept: text/html, text/plain; q=0.5, text/x-dvi; q=0.8
```

specifies that the browser is willing to accept the *text/html* media type, but if that does not exist, the browser will accept *text/x-dvi*, and, if that does not exist, *text/plain*. The numeric values associated with the second and third entry can be thought of as a *prefer-*

ence level, where no value is equivalent to $q=1$, and a value of $q=0$ means the type is unacceptable. For media types where "quality" is meaningful (e.g., audio), the value of q can be interpreted as a willingness to accept a given media type if it is the best available after other forms are reduced in quality by q percent.

A variety of *Accept* headers exist that correspond to the *Content* headers described earlier. For example, a browser can send any of the following:

```
Accept-Encoding:
Accept-Charset:
Accept-Language:
```

to specify which encodings, character sets, and languages the browser is willing to accept.

To summarize:

HTTP uses MIME-like headers to carry meta information. Both browsers and servers send headers that allow them to negotiate agreement on the document representation and encoding to be used.

28.13 Conditional Requests

HTTP allows a sender to make a request *conditional*. That is, when a browser sends a request, it includes a header that qualifies conditions under which the request should be honored. If the specified condition is not met, the server does not return the requested item. Conditional requests allow a browser to optimize retrieval by avoiding unnecessary transfers. The *If-Modified-Since* request specifies one of the most straightforward conditionals — it allows a browser to avoid transferring an item unless the item has been updated since a specified date. For example, a browser can include the header:

If-Modified-Since: Sat, 01 Jan 2000 05:00:01 GMT

with a *GET* request to avoid a transfer if the item is older than January 1, 2000.

28.14 Support For Proxy Servers

Proxy servers are an important part of the Web architecture because they provide an optimization that decreases latency and reduces the load on servers. However, proxies are not transparent — a browser must be configured to contact a local proxy instead of the original source, and the proxy must be configured to cache copies of Web pages. For example, a corporation in which many employees use the Internet may choose to have a proxy server. The corporation configures all its browsers to send requests to the proxy. The first time a user in the corporation accesses a given Web page, the proxy must obtain a copy from the server that manages the page. The proxy places the copy in its cache, and returns the page as the response to the request. The next time a user accesses the same page, the proxy extracts the data from its cache without sending a request across the Internet. Consequently, traffic from the site to the Internet is significantly reduced.

To guarantee correctness, HTTP includes explicit support for proxy servers. The protocol specifies exactly how a proxy handles each request, how headers should be interpreted by proxies, how a browser negotiates with a proxy, and how a proxy negotiates with a server. Furthermore, several HTTP headers have been designed specifically for use by proxies. For example, one header allows a proxy to authenticate itself to a server, and another allows each proxy that handles an item to record its identity so the ultimate recipient receives a list of all intermediate proxies. Finally, HTTP allows a server to control how proxies handle each Web page. For example, a server can include the *Max-Forwards* header in a response to limit the number of proxies that handle an item before it is delivered to a browser. If the server specifies a count of one, as in:

```
Max-Forwards: 1
```

at most one proxy can handle the item along the path from the server to the browser. A count of zero prohibits any proxy from handling the item.

28.15 Caching

The goal of caching is improved efficiency: a cache reduces both latency and network traffic by eliminating unnecessary transfers. The most obvious aspect of caching is storage: when a Web page is initially accessed, a copy is stored on disk, either by the browser, an intermediate proxy, or both. Subsequent requests for the same page can short-circuit the lookup process and retrieve a copy of the page from the cache instead of the server.

The central question in all caching schemes concerns timing — how long should an item be kept in a cache? On one hand, keeping a cached copy too long results in the copy becoming *stale*, which means that changes to the original are not reflected in the cached copy. On the other hand, if the cached copy is not kept long enough, inefficiency results because the next request must go back to the server.

HTTP allows a server to control caching in two ways. First, when it answers a request for a page, a server can specify caching details, including whether the page can be cached at all, whether a proxy can cache the page, the community with which a cached copy can be shared, the time at which the cached copy must expire, and limits on transformations that can be applied to the copy. Second, HTTP allows a browser to force *revalidation* of a page. To do so, the browser sends a request for the page, and uses a header to specify that the maximum "age" (i.e., the time since a copy of the page was stored) cannot be greater than zero. No copy of the page in a cache can be used to satisfy the request because the copy will have a nonzero age. Thus, only the original server will answer the request. Intermediate proxies along the way will receive a fresh copy for their cache as will the browser that issued the request.

To summarize:

> *Caching is key to the efficient operation of the Web. HTTP allows servers to control whether and how a page can be cached as well as its lifetime; a browser can force a request for a page to bypass caches and obtain a fresh copy from the server that owns the page.*

28.16 Summary

The World Wide Web consists of hypermedia documents stored on a set of Web servers and accessed by browsers. Each document is assigned a URL that uniquely identifies it; the URL specifies the protocol used to retrieve the document, the location of the server, and the path to the document on that server.

The HyperText Markup Language, HTML, allows a document to contain text along with embedded commands that control formatting. HTML also allows a document to contain links to other documents.

A browser and server use the HyperText Transfer Protocol, HTTP, to communicate. HTTP is an application-level protocol with explicit support for negotiation, proxy servers, caching, and persistent connections.

FOR FURTHER STUDY

Berners-Lee, et. al. [RFC 1768] defines URLs. A variety of RFCs contain proposals for extensions. Daniel and Mealling [RFC 2168] considers how to store URLs in the Domain Name System.

Berners-Lee and Connolly [RFC 1866] contains the standard for version 2 of HTML. Nebel and Masinter [RFC 1867] specifies HTML form upload, and Raggett [RFC 1942] gives the standard for tables in HTML.

Fielding et. al. [RFC 2616] specifies version 1.1 of HTTP, which adds many features, including additional support for persistence and caching, to the previous version. Franks et. al. [RFC 2617] considers access authentication in HTTP.

EXERCISES

28.1 Read the standard for URLs. What does a pound sign (#) followed by a string mean at the end of a URL?

28.2 Extend the previous exercise. Is it legal to send the pound sign suffix on a URL to a Web server? Why or why not?

28.3 How does a browser distinguish between a document that contains HTML and a document that contains arbitrary text? To find out, experiment by using a browser to read from a file. Does the browser use the name of the file or the contents to decide how to interpret the file?

28.4 What is the purpose of an HTTP *TRACE* command?

28.5 What is the difference between an HTTP *PUT* command and an HTTP *POST* command? When is each useful?

28.6 When is an HTTP *Keep-Alive* header used?

28.7 Can an arbitrary Web server function as a proxy? To find out, choose an arbitrary Web server and configure your browser to use it as a proxy. Do the results surprise you?

28.8 Read about HTTP's *must-revalidate* cache control directive. Give an example of a Web page that would use such a directive.

28.9 If a browser does not send an HTTP *Content-Length* header before a request, how does a server respond?

29

Applications: Voice And Video Over IP (RTP)

29.1 Introduction

This chapter focuses on the transfer of real-time data such as voice and video over an IP network. In addition to discussing the protocols used to transport such data, the chapter considers two broader issues. First, it examines the question of how IP can be used to provide commercial telephone service. Second, it examines the question of how routers in an IP network can guarantee sufficient service to provide high-quality video and audio reproduction.

Although it was designed and optimized to transport data, IP has successfully carried audio and video since its inception. In fact, researchers began to experiment with audio transmission across the ARPANET before the Internet was in place. By the 1990s, commercial radio stations were sending audio across the Internet, and software was available that allowed an individual to send audio across the Internet or to the standard telephone network. Commercial telephone companies also began using IP technology internally to carry voice.

29.2 Audio Clips And Encoding Standards

The simplest way to transfer audio across an IP network consists of *digitizing* an analog audio signal to produce a data file, using a conventional protocol to transfer the file, and then decoding the digital file to reproduce the original analog signal. Of course, the technique does not work well for interactive exchange because placing encoded audio in a file and transferring the file introduces a long delay. Thus, file transfer is typically used to send short audio recordings, which are known as *audio clips*.

Special hardware is used to form high-quality digitized audio. Known as a *coder/decoder* (*codec*), the device can covert in either direction between an analog audio signal and an equivalent digital representation. The most common type of codec, a *waveform coder*, measures the amplitude of the input signal at regular intervals and converts each sample into a digital value (i.e., an integer)†. To decode, the codec takes a sequence of integers as input and recreates the continuous analog signal that matches the digital values.

Several digital encoding standards exist, with the main tradeoff being between quality of reproduction and the size of digital representation. For example, the conventional telephone system uses the *Pulse Code Modulation* (*PCM*) standard that specifies taking an 8-bit sample every 125 μ seconds (i.e., 8000 times per second). As a result, a digitized telephone call produces data at a rate of 64 Kbps. The PCM encoding produces a surprising amount of output — storing a 128 second audio clip requires one megabyte of memory.

There are three ways to reduce the amount of data generated by digital encoding: take fewer samples per second, use fewer bits to encode each sample, or use a digital compression scheme to reduce the size of the resulting output. Various systems exist that use one or more of the techniques, making it possible to find products that produce encoded audio at a rate of only 2.2 Kbps. However, each technique has disadvantages. The chief disadvantage of taking fewer samples or using fewer bits to encode a sample is lower quality audio — the system cannot reproduce as large a range of sounds. The chief disadvantage of compression is delay — digitized output must be held while it is compressed. Furthermore, because greater reduction in size requires more processing, the best compression either requires a fast CPU or introduces longer delay. Thus, compression is most useful when delay is unimportant (e.g., when the output from a codec is being stored in a file).

29.3 Audio And Video Transmission And Reproduction

Many audio and video applications are classified as *real-time* because they require timely transmission and delivery‡. For example, an interactive telephone call is a real-time exchange because audio must be delivered without significant delay or users find the system unsatisfactory. Timely transfer means more than low delay because the resulting signal is unintelligible unless it is presented in exactly the same order as the original, and with exactly the same timing. Thus, if a sender takes a sample every 125 μ seconds, the receiver must convert digital values to analog at exactly the same rate.

How can a network guarantee that the stream is delivered at exactly the same rate that the sender used? The conventional telephone system introduced one answer: an *isochronous* architecture. Isochronous design means that the entire system, including the digital circuits, must be engineered to deliver output with exactly the same timing as was used to generate input. Thus, an isochronous system that has multiple paths between any two points must be engineered so all paths have exactly the same delay.

†An alternative known as a *voice coder/decoder* (*vocodec*) recognizes and encodes human speech rather than general waveforms.

‡Timeliness is more important than reliability; missing data is merely skipped.

An IP internet is not isochronous. We have already seen that datagrams can be duplicated, delayed, or arrive out of order. Variance in delay is called *jitter*, and is especially pervasive in IP networks. To allow meaningful transmission and reproduction of digitized signals across a network with IP semantics, additional protocol support is required. To handle datagram duplication and out-of-order delivery, each transmission must contain a sequence number. To handle jitter, each transmission must contain a *timestamp* that tells the receiver at which time the data in the packet should be played back. Separating sequence and timing information allows a receiver to reconstruct the signal accurately independent of how the packets arrive. Such timing information is especially critical when a datagram is lost or if the sender stops encoding during periods of silence; it allows the receiver to pause during playback the amount of time specified by the timestamps. To summarize:

Because an IP internet is not isochronous, additional protocol support is required when sending digitized real-time data. In addition to basic sequence information that allows detection of duplicate or reordered packets, each packet must carry a separate timestamp that tells the receiver the exact time at which the data in the packet should be played.

29.4 Jitter And Playback Delay

How can a receiver recreate a signal accurately if the network introduces jitter? The receiver must implement a *playback buffer*† as Figure 29.1 illustrates.

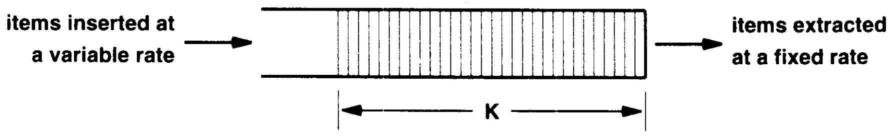
items inserted at a variable rate
items extracted at a fixed rate
K

Figure 29.1 The conceptual organization of a playback buffer that compensates for jitter. The buffer holds K time units of data.

When a session begins, the receiver delays playback and places incoming data in the buffer. When data in the buffer reaches a predetermined threshold, known as the *playback point*, output begins. The playback point, labeled K in the figure, is measured in time units of data to be played. Thus, playback begins when a receiver has accumulated K time unit's worth of data.

As playback proceeds, datagrams continue to arrive. If there is no jitter, new data will arrive at exactly the same rate old data is being extracted and played, meaning the buffer will always contain exactly K time units of unplayed data. If a datagram experi-

†A playback buffer is also called a *jitter buffer*.

ences a small delay, playback is unaffected. The buffer size decreases steadily as data is extracted, and playback continues uninterrupted for *K* time units. When a delayed datagram arrives, the buffer is refilled.

Of course, a playback buffer cannot compensate for datagram loss. In such cases, playback eventually reaches an unfilled position in the buffer, and output pauses for a time period corresponding to the missing data. Furthermore, the choice of *K* is a compromise between loss and delay†. If *K* is too small, a small amount of jitter causes the system to exhaust the playback buffer before the needed data arrives. If *K* is too large, the system remains immune to jitter, but the extra delay, when added to the transmission delay in the underlying network, may be noticeable to users. Despite the disadvantages, most applications that send real-time data across an IP internet depend on playback buffering as the primary solution for jitter.

29.5 Real-Time Transport Protocol (RTP)

The protocol used to transmit digitized audio or video signals over an IP internet is known as the *Real-Time Transport Protocol* (*RTP*). Interestingly, RTP does not contain mechanisms that ensure timely delivery; such guarantees must be made by the underlying system. Instead, RTP provides two key facilities: a sequence number in each packet that allows a receiver to detect out-of-order delivery or loss, and a timestamp that allows a receiver to control playback.

Because RTP is designed to carry a wide variety of real-time data, including both audio and video, RTP does not enforce a uniform interpretation of semantics. Instead, each packet begins with a fixed header; fields in the header specify how to interpret remaining header fields and how to interpret the payload. Figure 29.2 illustrates the format of RTP's fixed header.

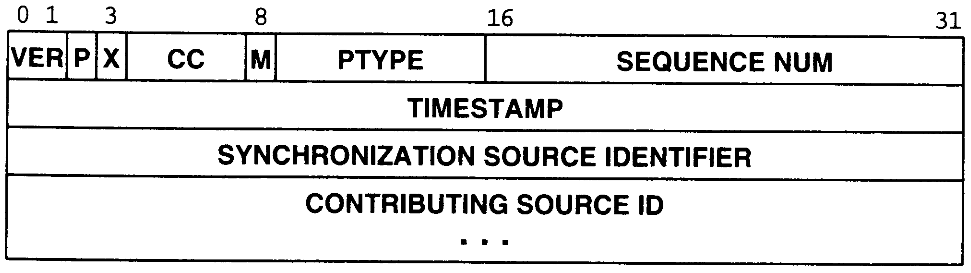

Figure 29.2 Illustration of the fixed header used with RTP. Each message begins with this header; the exact interpretation and additional header fields depend on the payload type, *PTYPE*.

†Although network delay and jitter can be used to determine a value for *K* dynamically, many playback buffering schemes use a constant.

As the figure shows, each packet begins with a two-bit RTP version number in field *VER*; the current version is *2*. The sixteen-bit *SEQUENCE NUM* field contains a sequence number for the packet. The first sequence number in a particular session is chosen at random. Some applications define an optional header extension to be placed between the fixed header and the payload. If the application type allows an extension, the *X* bit is used to specify whether the extension is present in the packet. The interpretation of most of the remaining fields in the header depends on the seven-bit *PTYPE* field that specifies the payload type. The *P* bit specifies whether zero padding follows the payload; it is used with encryption that requires data to be allocated in fixed-size blocks. Interpretation of the *M* (''marker'') bit also depends on the application; it is used by applications that need to mark points in the data stream (e.g., the beginning of each frame when sending video).

The payload type also affects the interpretation of the *TIMESTAMP* field. A *timestamp* is a 32-bit value that gives the time at which the first octet of digitized data was sampled, with the initial timestamp for a session chosen at random. The standard specifies that the timestamp is incremented continuously, even during periods when no signal is detected and no values are sent, but it does not specify the exact granularity. Instead, the granularity is determined by the payload type, which means that each application can choose a clock granularity that allows a receiver to position items in the output with accuracy appropriate to the application. For example, if a stream of audio data is being transmitted over RTP, a logical timestamp granularity of one clock tick per sample is appropriate†. However, if video data is being transmitted, the timestamp granularity needs to be higher than one tick per frame to achieve smooth playback. In any case, the standard allows the timestamps in two packets to be identical, if the data in the two packets was sampled at the same time.

29.6 Streams, Mixing, And Multicasting

A key part of RTP is its support for *translation* (i.e., changing the encoding of a stream at an intermediate station) or *mixing* (i.e., receiving streams of data from multiple sources, combining them into a single stream, and sending the result). To understand the need for mixing, imagine that individuals at multiple sites participate in a conference call using IP. To minimize the number of RTP streams, the group can designate a *mixer*, and arrange for each site to establish an RTP session to the mixer. The mixer combines the audio streams (possibly by converting them back to analog and resampling the resulting signal), and sends the result as a single digital stream.

Fields in the RTP header identify the sender and indicate whether mixing occurred. The field labeled *SYNCHRONIZATION SOURCE IDENTIFIER* specifies the source of a stream. Each source must choose a unique 32-bit identifier; the protocol includes a mechanism for resolving conflicts if they arise. When a mixer combines multiple streams, the mixer becomes the synchronization source for the new stream. Information about the original sources is not lost, however, because the mixer uses the variable-size *CONTRIBUTING SOURCE ID* field to provide the synchronization IDs of streams that

†The *TIMESTAMP* is sometimes referred to as a *MEDIA TIMESTAMP* to emphasize that its granularity depends on the type of signal being measured.

were mixed together. The four-bit *CC* field gives a count of contributing sources; a maximum of 15 sources can be listed.

RTP is designed to work with IP multicasting, and mixing is especially attractive in a multicast environment. To understand why, imagine a teleconference that includes many participants. Unicasting requires a station to send a copy of each outgoing RTP packet to each participant. With multicasting, however, a station only needs to send one copy of the packet, which will be delivered to all participants. Furthermore, if mixing is used, all sources can unicast to a mixer, which combines them into a single stream before multicasting. Thus, the combination of mixing and multicast results in substantially fewer datagrams being delivered to each participating host.

29.7 RTP Encapsulation

Its name implies that RTP is a transport-level protocol. Indeed, if it functioned like a conventional transport protocol, RTP would require each message to be encapsulated directly in an IP datagram. In fact, RTP does not function like a transport protocol; although it is allowed, direct encapsulation in IP does not occur in practice. Instead, RTP runs over UDP, meaning that each RTP message is encapsulated in a UDP datagram. The chief advantage of using UDP is concurrency — a single computer can have multiple applications using RTP without interference.

Unlike many of the application protocols we have seen, RTP does not use a reserved UDP port number. Instead, a port is allocated for use with each session, and the remote application must be informed about the port number. By convention, RTP chooses an even numbered UDP port; the following section explains that a companion protocol, RTCP, uses the next port number.

29.8 RTP Control Protocol (RTCP)

So far, our description of real-time transmission has focused on the protocol mechanisms that allow a receiver to reproduce content. However, another aspect of real-time transmission is equally important: monitoring of the underlying network during the session and providing *out of band* communication between the endpoints. Such a mechanism is especially important in cases where adaptive schemes are used. For example, an application might choose a lower-bandwidth encoding when the underlying network becomes congested, or a receiver might vary the size of its playback buffer when network delay or jitter changes. Finally, an out-of-band mechanism can be used to send information in parallel with the real-time data (e.g., captions to accompany a video stream).

A companion protocol and integral part of RTP, known as the *RTP Control Protocol* (*RTCP*), provides the needed control functionality. RTCP allows senders and receivers to transmit a series of reports to one another that contain additional information about the data being transferred and the performance of the network. RTCP messages are encapsulated in UDP for transmission†, and are sent using a protocol number one greater than the port number of the RTP stream to which they pertain.

29.9 RTCP Operation

RTCP uses five basic message types to allow senders and receivers to exchange information about a session. Figure 29.3 lists the types.

Type	Meaning
200	Sender report
201	Receiver report
202	Source description message
203	Bye message
204	Application specific message

Figure 29.3 The five RTCP message types. Each message begins with a fixed header that identifies the type.

The *bye* and *application specific* messages are the most straightforward. A sender transmits a bye message when shutting down a stream. The application specific message type provides an extension of the basic facility to allow the application to define a message type. For example, an application that sends a closed caption to accompany a video stream might choose to define an RTCP message that supports closed captioning.

Receivers periodically transmit *receiver report* messages that inform the source about conditions of reception. Receiver reports are important for two reasons. First, they allow all receivers participating in a session as well as a sender to learn about reception conditions of other receivers. Second, they allow receivers to adapt their rate of reporting to avoid using excessive bandwidth and overwhelming the sender. The adaptive scheme guarantees that the total control traffic will remain less than 5% of the real-time data traffic, and that receiver reports generate less than 75% of the control traffic. Each receiver report identifies one or more synchronization sources, and contains a separate section for each. A section specifies the highest sequence number packet received from the source, the cumulative and percentage packet loss experienced, time since the last RTCP report arrived from the source, and the interarrival jitter.

Senders periodically transmit a *sender report* message that provides an absolute timestamp. To understand the need for a timestamp, recall that RTP allows each stream to choose a granularity for its timestamp and that the first timestamp is chosen at random. The absolute timestamp in a sender report is essential because it provides the only mechanism a receiver has to *synchronize* multiple streams. In particular, because RTP requires a separate stream for each media type, the transmission of video and accompanying audio requires two streams. The absolute timestamp information allows a receiver to play the two streams simultaneously.

†Because some messages are short, the standard allows multiple RTCP messages to be combined into a single UDP datagram for transmission.

In addition to the periodic sender report messages, senders also transmit *source description* messages which provide general information about the user who owns or controls the source. Each message contains one section for each outgoing RTP stream; the contents are intended for humans to read. For example, the only required field consists of a *canonical name* for the stream owner, a character string in the form:

user @ host

where *host* is either the domain name of the computer or its IP address in dotted decimal form, and *user* is a login name. Optional fields in the source description contain further details such as the user's e-mail address (which may differ from the canonical name), telephone number, the geographic location of the site, the application program or tool used to create the stream, or other textual notes about the source.

29.10 IP Telephony And Signaling

One aspect of real-time transmission stands out as especially important: the use of IP as the foundation for telephone service. Known as *IP telephony* or *voice over IP*, the idea is endorsed by many telephone companies. The question arises, "what additional technologies are needed before IP can be used in place of the existing isochronous telephone system?" Although no simple answer exists, researchers are investigating three components. First, we have seen that a protocol like RTP is needed to transfer a digitized signal across an IP internet correctly. Second, a mechanism is needed to establish and terminate telephone calls. Third, researchers are exploring ways an IP internet can be made to function like an isochronous network.

The telephone industry uses the term *signaling* to refer to the process of establishing a telephone call. Specifically, the signaling mechanism used in the conventional *Public Switched Telephone Network* (*PSTN*) is *Signaling System 7* (*SS7*). SS7 performs call routing before any audio is sent. Given a phone number, it forms a circuit through the network, rings the designated telephone, and connects the circuit when the phone is answered. SS7 also handles details such as call forwarding and error conditions such as the destination phone being busy.

Before IP can be used to make phone calls, signaling functionality must be available. Furthermore, to enable adoption by the phone companies, IP telephony must be compatible with extant telephone standards — it must be possible for the IP telephony system to interoperate with the conventional phone system at all levels. Thus, it must be possible to translate between the signaling used with IP and SS7 just as it must be possible to translate between the voice encoding used with IP and standard PCM encoding. As a consequence, the two signaling mechanisms will have equivalent functionality.

The general approach to interoperability uses a *gateway* between the IP phone system and the conventional phone system. A call can be initiated on either side of the gateway. When a signaling request arrives, the gateway translates and forwards the request; the gateway must also translate and forward the response. Finally, after signaling is complete and a call has been established, the gateway must forward voice in both directions, translating from the encoding used on one side to the encoding used on the other.

Two groups have proposed standards for IP telephony. The ITU has defined a suite of protocols known as *H.323*, and the IETF has proposed a signaling protocol known as the *Session Initiation Protocol* (*SIP*). The next sections summarize the two approaches.

29.10.1 H.323 Standards

The ITU originally created H.323 to allow the transmission of voice over local area network technologies. The standard has been extended to allow transmission of voice over IP internets, and telephone companies are expected to adopt it. H.323 is not a single protocol. Instead, it specifies how multiple protocols can be combined to form a functional IP telephony system. For example, in addition to gateways, H.323 defines devices known as *gatekeepers* that each provide a contact point for telephones using IP. To obtain permission to place outgoing calls and enable the phone system to correctly route incoming calls, each IP telephone must register with a gatekeeper; H.323 includes the necessary protocols.

In addition to specifying a protocol for the transmission of real-time voice and video, the H.323 framework allows participants to transfer data. Thus, a pair of users engaged in an audio-video conference can also share an on-screen whiteboard, send still images, or exchange copies of documents.

H.323 relies on the four major protocols listed in Figure 29.4.

Protocol	Purpose
H.225.0	Signaling used to establish a call
H.245	Control and feedback during the call
RTP	Real-time data transfer (sequence and timing)
T.120	Exchange of data associated with a call

Figure 29.4 The protocols used by H.323 for IP telephony.

Together, the suite of protocols covers all aspects of IP telephony, including phone registration, signaling, real-time data encoding and transfer (both voice and video), and control.

Figure 29.5 illustrates relationships among the protocols that comprise H.323. As the figure shows, the entire suite ultimately depends on UDP and TCP running over IP.

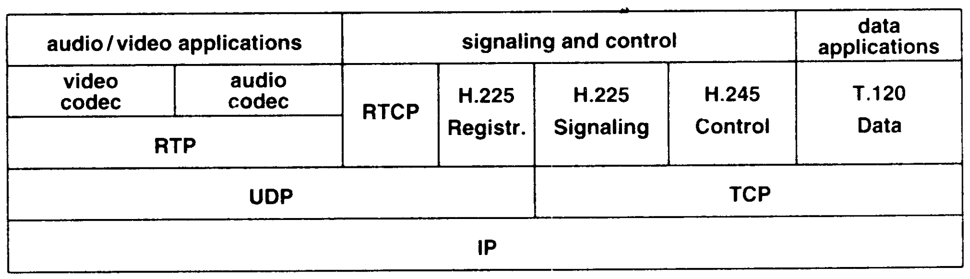

Figure 29.5 Relationship among protocols that comprise the ITU's H.323 IP telephony standard.

29.10.2 Session Initiation Protocol (SIP)

The IETF has proposed an alternative to H.323, called the *Session Initiation Protocol* (*SIP*), that only covers signaling; it does not recommend specific codecs nor does it require the use of RTP for real-time transfer. Thus, SIP does not supply all of H.323 functionality.

SIP uses client-server interaction, with servers being divided into two types. A *user agent server* runs in a SIP telephone. It is assigned an identifier (e.g., *user @ site*), and can receive incoming calls. The second type of server is intermediate (i.e., between two SIP telephones) and handles tasks such as call set up and call forwarding. An intermediate server functions as a *proxy server* that can forward an incoming call request to the next proxy server along the path to the called phone, or as a *redirect server* that tells a caller how to reach the destination.

To provide information about a call, SIP relies on a companion protocol, the *Session Description Protocol* (*SDP*). SDP is especially important in a conference call, because participants join and leave the call dynamically. SDP specifies details such as the media encoding, protocol port numbers, and multicast address.

29.11 Resource Reservation And Quality Of Service

The term *Quality Of Service* (*QoS*) refers to statistical performance guarantees that a network system can make regarding loss, delay, throughput, and jitter. An isochronous network that is engineered to meet strict performance bounds is said to provide QoS guarantees, while a packet switched network that uses best effort delivery is said to provide no QoS guarantee. Is guaranteed QoS needed for real-time transfer of voice and video over IP? If so, how should it be implemented? A major controversy surrounds the two questions. On one hand, engineers who designed the telephone system insist that toll-quality voice reproduction requires the underlying system to provide QoS guarantees about delay and loss for each phone call. On the other hand, engineers who designed IP insist that the Internet works reasonably well without QoS guarantees and that adding per-flow QoS is infeasible because routers will make the system both expensive and slow.

The QoS controversy has produced many proposals, implementations, and experiments. Although it operates without QoS, the Internet is already used to send audio. Technologies like ATM that were derived from the telephone system model provide QoS guarantees for each individual connection. Finally, in Chapter 7 we learned that the IETF adopted a conservative *differentiated services* approach that divides traffic into separate QoS classes. The differentiated services scheme sacrifices fine grain control for less complex forwarding.

29.12 QoS, Utilization, And Capacity

The debate over QoS is reminiscent of earlier debates on resource allocation such as those waged over operating system policies for memory allocation and processor scheduling. The central issue is utilization: when a network has sufficient resources for all traffic, QoS constraints are unnecessary; when traffic exceeds network capacity, no QoS system can satisfy all users' demands. That is, a network with 1% utilization does not need QoS, and a network with 101% utilization will fail under any QoS. In effect, proponents who argue for QoS mechanisms assert that complex QoS mechanisms achieve two goals. First, by dividing the existing resources among more users, they make the system more "fair". Second, by shaping the traffic from each user, they allow the network to run at higher utilization without danger of collapse.

One of the major arguments against complicated QoS mechanisms arises from improvements in the performance of underlying networks. Network capacity has increased dramatically. As long as rapid increases in capacity continue, QoS mechanisms merely represent unnecessary overhead. However, if demand rises more rapidly than capacity, QoS may become an economic issue — by associating higher prices with higher levels of service, ISPs can use cost to ration capacity.

29.13 RSVP

If QoS is needed, how can an IP network provide it? Before announcing the differentiated services solution, the IETF worked on a scheme that can be used to provide QoS in an IP environment. The work produced a pair of protocols: the *Resource ReserVation Protocol* (*RSVP*) and the *Common Open Policy Services* (*COPS*) protocol.

QoS cannot be added to IP at the application layer. Instead, the basic infrastructure must change — routers must agree to reserve resources (e.g., bandwidth) for each flow between a pair of endpoints. There are two aspects. First, before data is sent, the endpoints must send a request that specifies the resources needed, and all routers along the path must agree to supply the resources; the procedure can be viewed as a form of signaling. Second, as datagrams traverse the flow, routers need to monitor and control traffic forwarding. Monitoring, sometimes called *traffic policing*, is needed to ensure that the traffic sent on a flow does not exceed the specified bounds. Control of queueing and forwarding is needed for two reasons. The router must implement a queueing policy that meets the guaranteed bounds on delay, and the router must smooth packet bursts. The latter is sometimes referred to as *traffic shaping*, and is necessary because network traffic is often *bursty*. For example, a flow that specifies an average throughput of 1 Mbps may have 2 Mbps of traffic for a millisecond followed by no traffic for a millisecond. A router can reshape the burst by temporarily queueing incoming datagrams and sending them at a steady rate of 1 Mbps.

RSVP handles reservation requests and replies. It is not a routing protocol, nor does it enforce policies once a flow has been established. Instead, RSVP operates before any data is sent. To initiate an end-to-end flow, an endpoint first sends an RSVP *path* message to determine the path to the destination; the datagram carrying the message uses the *router alert* option to guarantee that routers examine the message. After it receives a reply to its path message, the endpoint sends a request message to reserve resources for the flow. The request specifies QoS bounds desired; each router that forwards the request along to the destination must agree to reserve the resources the request specifies. If any router along the path denies the request, the router uses RSVP to send a negative reply back to the source. If all systems along the path agree to honor the request, RSVP returns a positive reply.

Each RSVP flow is *simplex* (i.e., unidirectional). If an application requires QoS guarantees in two directions, each endpoint must use RSVP to request a flow. Because RSVP uses existing routing, there is no guarantee that the two flows will pass through the same routers, nor does approval of a flow in one direction imply approval in the other. We can summarize:

An endpoint uses RSVP to request a simplex flow through an IP internet with specified QoS bounds. If routers along the path agree to honor the request, they approve it; otherwise, they deny it. If an application needs QoS in two directions, each endpoint must use RSVP to request a separate flow.

29.14 COPS

When an RSVP request arrives, a router must evaluate two aspects: feasibility (i.e., whether the router has the resources necessary to satisfy the request) and policies (i.e., whether the request lies within policy constraints). Feasibility is a local decision — a router can decide how to manage the bandwidth, memory, and processing power that is available locally. However, policy enforcement requires global cooperation — all routers must agree to the same set of policies.

To implement global policies, the IETF architecture uses a two-level model, with client-server interaction between the levels. When a router receives a RSVP request, it becomes a client that consults a server known as a *Policy Decision Point* (*PDP*) to determine whether the request meets policy constraints. The PDP does not handle traffic; it merely evaluates requests to see if they satisfy global policies. If a PDP approves a request, the router must operate as a *Policy Enforcement Point PEP* to ensure traffic does not exceed the approved policy.

The COPS protocol defines the client-server interaction between a router and a PDP (or between a router and a local PDP if the organization has multiple levels of policy servers). Although COPS defines its own message header, the underlying format shares many details with RSVP. In particular, COPS uses the same format as RSVP for individual items in a request message. Thus, when a router receives an RSVP request, it can extract items related to policy, place them in a COPS message, and send the result to a PDP.

29.15 Summary

Analog data such as audio can be encoded in digital form; the hardware to do so is known as a codec. The telephone standard for digital audio encoding, Pulse Code Modulation (PCM), produces digital values at 64 Kbps.

RTP is used to transfer real-time data across an IP network. Each RTP message contains two key pieces of information: a sequence number that a receiver uses to place messages in order and detect lost datagrams and a media timestamp that a receiver uses to determine when to play the encoded values. An associated control protocol, RTCP, is used to supply information about sources and to allow a mixer to combine several streams.

A debate continues over whether Quality of Service (QoS) guarantees are needed to provide real time. Before announcing a differentiated services approach, the IETF designed a pair of protocols that can be used to provide per-flow QoS. Endpoints use RSVP to request a flow with specific QoS; intermediate routers either approve or deny the request. When an RSVP request arrives, a router uses the COPS protocol to contact a Policy Decision Point and verify that the request meets policy constraints.

FOR FURTHER STUDY

Schulzrinne et. al. [RFC 1889] gives the standard for RTP and RTCP. Perkins et. al. [RFC 2198] specifies the transmission of redundant audio data over RTP, and Schulzrinne [RFC 1890] specifies the use of RTP with an audio-video conference. Schulzrinne, Rao, and Lanphier [RFC 2326] describes a related protocol used for streaming of real-time traffic.

Zhang et. al. [RFC 2205] contains the specification for RSVP. Boyle et. al. [draft-rap-cops-06.txt] describes COPS.

EXERCISES

29.1 Read about the Real Time Streaming Protocol, RTSP. What are the major differences between RTSP and RTP?

29.2 Argue that although bandwidth is often cited as an example of the facilities a QoS mechanism can guarantee, delay is a more fundamental resource. (Hint: which constraint can be eased with sufficient money?)

29.3 If an RTP message arrives with a sequence number far greater than the sequence expected, what does the protocol do? Why?

29.4 Are sequence numbers necessary in RTP, or can a timestamp be used instead? Explain.

29.5 Would you prefer an internet where QoS was required for all traffic? Why or why not?

29.6 Measure the utilization on your connection to the Internet. If all traffic required QoS reservation, would service be better or worse? Explain.

30

Applications: Internet Management (SNMP)

30.1 Introduction

In addition to protocols that provide network level services and application programs that use those services, an internet needs software that allows managers to debug problems, control routing, and find computers that violate protocol standards. We refer to such activities as *internet management*. This chapter considers the ideas behind TCP/IP internet management software, and describes an internet management protocol.

30.2 The Level Of Management Protocols

Originally, many wide area networks included management protocols as part of their link level protocols. If a packet switch began misbehaving, the network manager could instruct a neighboring packet switch to send it a special *control packet*. Control packets caused the receiver to suspend normal operation and respond to commands from the manager. The manager could interrogate the packet switch to identify problems, examine or change routes, test one of the communication interfaces, or reboot the switch. Once managers repaired the problem, they could instruct the switch to resume normal operations. Because management tools were part of the lowest level protocol, managers were often able to control switches even if higher level protocols failed.

Unlike a homogeneous wide area network, a TCP/IP internet does not have a single link level protocol. Instead, the internet consists of multiple physical networks interconnected by IP routers. As a result, internet management differs from network management. First, a single manager can control heterogeneous devices, including IP routers, bridges, modems, workstations, and printers. Second, the controlled entities may not share a common link level protocol. Third, the set of machines a manager controls may lie at arbitrary points in an internet. In particular, a manager may need to control one or more machines that do not attach to the same physical network as the manager's computer. Thus, it may not be possible for a manager to communicate with machines being controlled unless the management software uses protocols that provide end-to-end connectivity across an internet. As a consequence, the internet management protocol used with TCP/IP operates above the transport level:

> *In a TCP/IP internet, a manager needs to examine and control routers and other network devices. Because such devices attach to arbitrary networks, protocols for internet management operate at the application level and communicate using TCP/IP transport-level protocols.*

Designing internet management software to operate at the application level has several advantages. Because the protocols can be designed without regard to the underlying network hardware, one set of protocols can be used for all networks. Because the protocols can be designed without regard to the hardware on the managed machine, the same protocols can be used for all managed devices. From a manager's point of view, having a single set of management protocols means uniformity — all routers respond to exactly the same set of commands. Furthermore, because the management software uses IP for communication, a manager can control the routers across an entire TCP/IP internet without having direct attachment to every physical network or router.

Of course, building management software at the application level also has disadvantages. Unless the operating system, IP software, and transport protocol software work correctly, the manager may not be able to contact a router that needs managing. For example, if a router's routing table becomes damaged, it may be impossible to correct the table or reboot the machine from a remote site. If the operating system on a router crashes, it will be impossible to reach the application program that implements the internet management protocols even if the router can still field hardware interrupts and route packets.

30.3 Architectural Model

Despite the potential disadvantages, having TCP/IP management software operate at the application level has worked well in practice. The most significant advantage of placing network management protocols at a high level becomes apparent when one considers a large internet, where a manager's computer does not need to attach directly to all physical networks that contain managed entities. Figure 30.1 shows an example of the architecture.

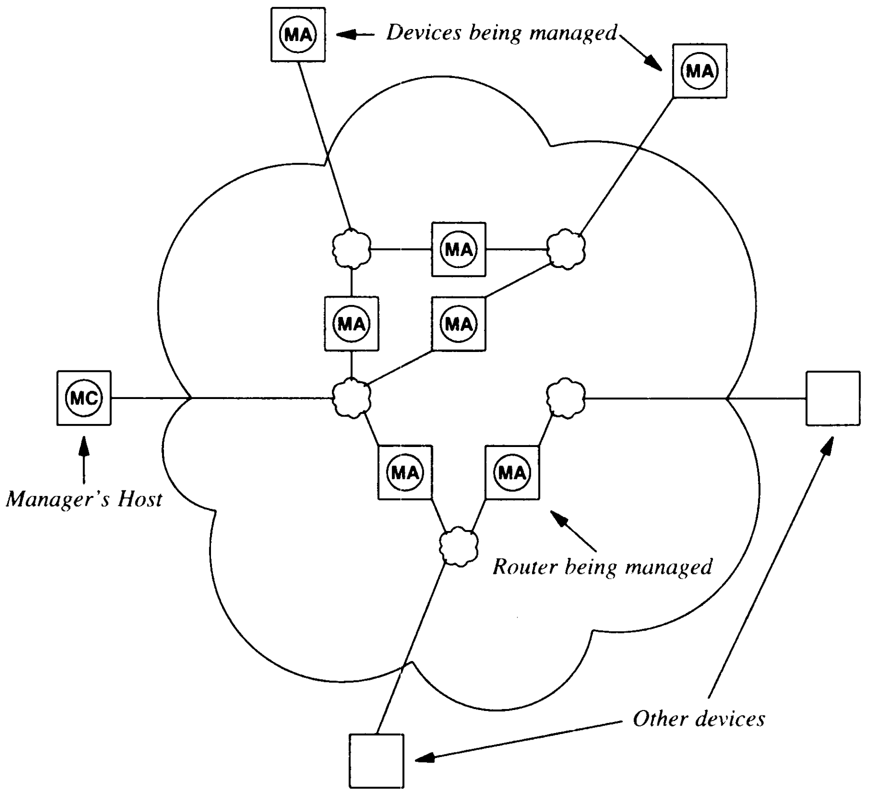

Figure 30.1 Example of network management. A manager invokes management client (MC) software that can contact management agent (MA) software that runs on devices throughout the internet.

As the figure shows, client software usually runs on the manager's workstation. Each participating router or host† runs a server program. Technically, the server software is called a *management agent* or merely an *agent*. A manager invokes client software on the local host computer and specifies an agent with which it communicates. After the client contacts the agent, it sends queries to obtain information or it sends commands to change conditions in the router. Of course, not all devices in a large internet fall under a single manager. Most managers only control devices at their local sites; a large site may have multiple managers.

†Recall that the TCP/IP term *host* can refer to a device (e.g., a printer) or a conventional computer.

Internet management software uses an authentication mechanism to ensure only authorized managers can access or control a particular device. Some management protocols support multiple levels of authorization, allowing a manager specific privileges on each device. For example, a specific router could be configured to allow several managers to obtain information while only allowing a select subset of them to change information or control the router.

30.4 Protocol Framework

TCP/IP network management protocols† divide the management problem into two parts and specify separate standards for each part. The first part concerns communication of information. A protocol specifies how client software running on a manager's host communicates with an agent. The protocol defines the format and meaning of messages clients and servers exchange as well as the form of names and addresses. The second part concerns the data being managed. A protocol specifies which data items a managed device must keep as well as the name of each data item and the syntax used to express the name.

30.4.1 A Standard Network Management Protocol

The TCP/IP standard for network management is the *Simple Network Management Protocol* (*SNMP*). The protocol has evolved through three generations. Consequently, the current version is known as *SNMPv3*, and the predecessors are known as *SNMPv1* and *SNMPv2*. The changes have been minor — all three versions use the same general framework, and many features are backward compatible.

In addition to specifying details such as the message format and the use of transport protocols, the SNMP standard defines the set of operations and the meaning of each. We will see that the approach is minimalistic; a few operations provide all functionality.

30.4.2 A Standard For Managed Information

A device being managed must keep control and status information that the manager can access. For example, a router keeps statistics on the status of its network interfaces, incoming and outgoing packet traffic, dropped datagrams, and error messages generated; a modem keeps statistics about the number of characters sent and received, baud rate, and calls accepted. Although it allows a manager to access statistics, SNMP does not specify exactly which data can be accessed on which devices. Instead, a separate standard specifies the details for each type of device. Known as a *Management Information Base* (*MIB*), the standard specifies the data items a managed device must keep, the operations allowed on each, and the meanings. For example, the MIB for IP specifies that the software must keep a count of all octets that arrive over each network interface and that network management software can only read the count.

†Technically, there is a distinction between internet management protocols and network management protocols. Historically, however, TCP/IP internet management protocols are known as *network management* protocols; we will follow the accepted terminology.

The MIB for TCP/IP divides management information into many categories. The choice of categories is important because identifiers used to specify items include a code for the category. Figure 30.2 lists a few examples.

MIB category	Includes Information About
system	The host or router operating system
interfaces	Individual network interfaces
at	Address translation (e.g., ARP mappings)
ip	Internet Protocol software
icmp	Internet Control Message Protocol software
tcp	Transmission Control Protocol software
udp	User Datagram Protocol software
ospf	Open Shortest Path First software
bgp	Border Gateway Protocol software
rmon	Remote network monitoring
rip-2	Routing Information Protocol software
dns	Domain Name System software

Figure 30.2 Example categories of MIB information. The category is encoded in the identifier used to specify an object.

Keeping the MIB definition independent of the network management protocol has advantages for both vendors and users. A vendor can include SNMP agent software in a product such as a router, with the guarantee that the software will continue to adhere to the standard after new MIB items are defined. A customer can use the same network management client software to manage multiple devices that have slightly different versions of a MIB. Of course, a device that does not have new MIB items cannot provide the information in those items. However, because all managed devices use the same language for communication, they can all parse a query and either provide the requested information or send an error message explaining that they do not have the requested item.

30.5 Examples of MIB Variables

Versions 1 and 2 of SNMP each collected variables together in a single large MIB, with the entire set documented in a single RFC. After publication of the second generation, *MIB-II*, the IETF took a different approach by allowing the publication of many individual MIB documents that each specify the variables for a specific type of device. As a result, more than 100 separate MIBs have been defined as part of the standards process; they specify more than 10,000 individual variables. For example, separate RFCs now exist that specify the MIB variables associated with devices such as: a hardware bridge, an uninterruptible power supply, an ATM switch, and a dialup modem. In addition, many vendors have defined MIB variables for their specific hardware or software products.

Examining a few of the MIB data items associated with TCP/IP protocols will help clarify the contents. Figure 30.3 lists example MIB variables along with their categories.

MIB Variable	Category	Meaning
sysUpTime	system	Time since last reboot
ifNumber	interfaces	Number of network interfaces
ifMtu	interfaces	MTU for a particular interface
ipDefaultTTL	ip	Value IP uses in time-to-live field
ipInReceives	ip	Number of datagrams received
ipForwDatagrams	ip	Number of datagrams forwarded
ipOutNoRoutes	ip	Number of routing failures
ipReasmOKs	ip	Number of datagrams reassembled
ipFragOKs	ip	Number of datagrams fragmented
ipRoutingTable	ip	IP Routing table
icmpInEchos	icmp	Number of ICMP Echo Requests received
tcpRtoMin	tcp	Minimum retransmission time TCP allows
tcpMaxConn	tcp	Maximum TCP connections allowed
tcpInSegs	tcp	Number of segments TCP has received
udpInDatagrams	udp	Number of UDP datagrams received

Figure 30.3 Examples of MIB variables along with their categories.

Most of the items listed in Figure 30.3 are numeric — each value can be stored in a single integer. However, the MIB also defines more complex structures. For example, the MIB variable *ipRoutingTable* refers to an entire routing table. Additional MIB variables define the contents of a routing table entry, and allow the network management protocols to reference an individual entry in the table, including the prefix, address mask, and next hop fields. Of course, MIB variables present only a logical definition of each data item — the internal data structures a router uses may differ from the MIB definition. When a query arrives, software in the agent on the router is responsible for mapping between the MIB variable and the data structure the router uses to store the information.

30.6 The Structure Of Management Information

In addition to the standards that specify MIB variables and their meanings, a separate standard specifies a set of rules used to define and identify MIB variables. The rules are known as the *Structure of Management Information* (*SMI*) specification. To keep network management protocols simple, the SMI places restrictions on the types of variables allowed in the MIB, specifies the rules for naming those variables, and creates rules for defining variable types. For example, the SMI standard includes definitions of terms like *IpAddress* (defining it to be a 4-octet string) and *Counter* (defining it to be an integer in the range of 0 to $2^{32} - 1$), and specifies that they are the terms used to define MIB variables. More important, the rules in the SMI describe how the MIB refers to tables of values (e.g., the IP routing table).

30.7 Formal Definitions Using ASN.1

The SMI standard specifies that all MIB variables must be defined and referenced using ISO's *Abstract Syntax Notation 1* (*ASN.1*†). ASN.1 is a formal language that has two main features: a notation used in documents that humans read and a compact encoded representation of the same information used in communication protocols. In both cases, the precise, formal notation removes any possible ambiguities from both the representation and meaning. For example, instead of saying that a variable contains an integer value, a protocol designer who uses ASN.1 must state the exact form and range of numeric values. Such precision is especially important when implementations include heterogeneous computers that do not all use the same representations for data items.

Besides keeping standards documents unambiguous, ASN.1 also helps simplify the implementation of network management protocols and guarantees interoperability. It defines precisely how to encode both names and data items in a message. Thus, once the documentation of a MIB has been expressed using ASN.1, the human readable form can be translated directly and mechanically into the encoded form used in messages. In summary:

> *The TCP/IP network management protocols use a formal notation called ASN.1 to define names and types for variables in the management information base. The precise notation makes the form and contents of variables unambiguous.*

30.8 Structure And Representation Of MIB Object Names

We said that ASN.1 specifies how to represent both data items and names. However, understanding the names used for MIB variables requires us to know about the underlying namespace. Names used for MIB variables are taken from the *object identifier* namespace administered by ISO and ITU. The key idea behind the object identifier namespace is that it provides a namespace in which all possible objects can be designated. The namespace is not restricted to variables used in network management — it includes names for arbitrary objects (e.g., each international protocol standard document has a name).

The object identifier namespace is *absolute* (*global*), meaning that names are structured to make them globally unique. Like most namespaces that are large and absolute, the object identifier namespace is hierarchical. Authority for parts of the namespace is subdivided at each level, allowing individual groups to obtain authority to assign some of the names without consulting a central authority for each assignment‡.

†ASN.1 is usually pronounced by reading the dot: "A-S-N dot 1".

‡Readers should recall from the Domain Name System discussion in Chapter 24 how authority for a hierarchical namespace is subdivided.

The root of the object identifier hierarchy is unnamed, but has three direct descendants managed by: ISO, ITU, and jointly by ISO and ITU. The descendants are assigned both short text strings and integers that identify them (the text strings are used by humans to understand object names; computer software uses the integers to form compact, encoded representations of the names). ISO has allocated one subtree for use by other national or international standards organizations (including U.S. standards organizations), and the U.S. National Institute for Standards and Technology† has allocated a subtree for the U.S. Department of Defense. Finally, the IAB has petitioned the Department of Defense to allocate it a subtree in the namespace. Figure 30.4 illustrates pertinent parts of the object identifier hierarchy and shows the position of the node used by TCP/IP network management protocols.

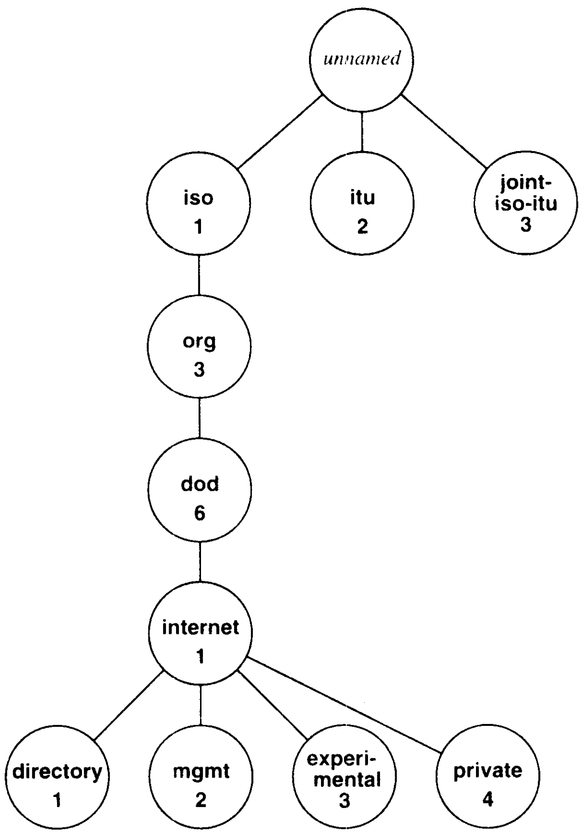

Figure 30.4 Part of the hierarchical object identifier namespace used to name MIB variables. An object's name consists of the numeric labels along a path from the root to the object.

†NIST was formerly the National Bureau of Standards.

The name of an object in the hierarchy is the sequence of numeric labels on the nodes along a path from the root to the object. The sequence is written with periods separating the individual components. For example, the name *1.3.6.1.2* denotes the node labeled *mgmt*, the *Internet management* subtree. The MIB has been assigned a node under the *mgmt* subtree with label *mib* and numeric value *1*. Because all MIB variables fall under that node, they all have names beginning with the prefix *1.3.6.1.2.1*.

Earlier we said that the MIB groups variables into categories. The exact meaning of the categories can now be explained: they are the subtrees of the *mib* node of the object identifier namespace. Figure 30.5 illustrates the idea by showing part of the naming subtree under the *mib* node.

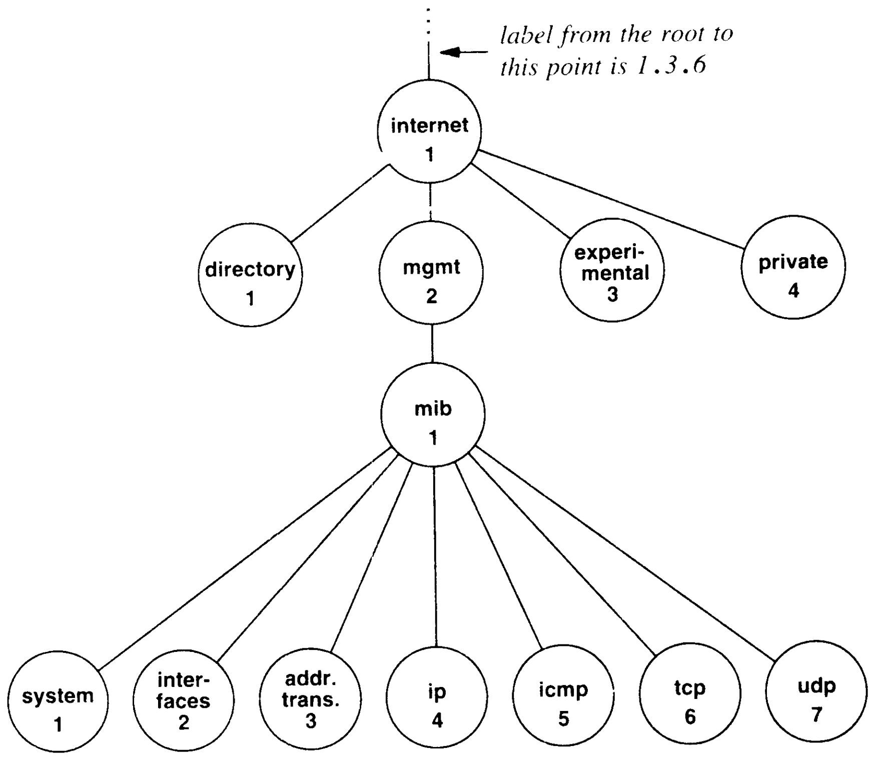

Figure 30.5 Part of the object identifier namespace under the IAB *mib* node. Each subtree corresponds to one of the categories of MIB variables.

Two examples will make the naming syntax clear. Figure 30.5 shows that the category labeled *ip* has been assigned the numeric value *4*. Thus, the names of all MIB variables corresponding to IP have an identifier that begins with the prefix *1.3.6.1.2.1.4*. If one wanted to write out the textual labels instead of the numeric representation, the name would be:

iso.org.dod.internet.mgmt.mib.ip

A MIB variable named *ipInReceives* has been assigned numeric identifier *3* under the *ip* node in the namespace, so its name is:

iso.org.dod.internet.mgmt.mib.ip.ipInReceives

and the corresponding numeric representation is:

1.3.6.1.2.1.4.3

When network management protocols use names of MIB variables in messages, each name has a suffix appended. For simple variables, the suffix *0* refers to the instance of the variable with that name. So, when it appears in a message sent to a router, the numeric representation of *ipInReceives* is:

1.3.6.1.2.1.4.3.0

which refers to the instance of *ipInReceives* on that router. Note that there is no way to guess the numeric value or suffix assigned to a variable. One must consult the published standards to find which numeric values have been assigned to each object type. Thus, programs that provide mappings between the textual form and underlying numeric values do so entirely by consulting tables of equivalences — there is no closed-form computation that performs the transformation.

As a second, more complex example, consider the MIB variable *ipAddrTable*, which contains a list of the IP addresses for each network interface. The variable exists in the namespace as a subtree under *ip*, and has been assigned the numeric value *20*. Therefore, a reference to it has the prefix:

iso.org.dod.internet.mgmt.mib.ip.ipAddrTable

with a numeric equivalent:

1.3.6.1.2.1.4.20

In programming language terms, we think of the IP address table as a one-dimensional array, where each element of the array consists of a structure (record) that contains five items: an IP address, the integer index of an interface corresponding to the entry, an IP subnet mask, an IP broadcast address, and an integer that specifies the maximum datagram size that the router will reassemble. Of course, it is unlikely that a router has such an array in memory. The router may keep this information in many variables or may need to follow pointers to find it. However, the MIB provides a name for the array as if it existed, and allows network management software on individual routers to map table references into appropriate internal variables. The point is:

> *Although they appear to specify details about data structures, MIB standards do not dictate the implementation. Instead, MIB definitions provide a uniform, virtual interface that managers use to access data; an agent must translate between the virtual items in a MIB and the internal implementation.*

Using ASN.1 style notation, we can define *ipAddrTable*:

```
ipAddrTable ::= SEQUENCE OF IpAddrEntry
```

where *SEQUENCE* and *OF* are keywords that define an ipAddrTable to be a one-dimensional array of *IpAddrEntry*s. Each entry in the array is defined to consist of five fields (the definition assumes that *IpAddress* has already been defined).

```
IpAddrEntry ::= SEQUENCE {
    ipAdEntAddr
            IpAddress,
    ipAdEntIfIndex
            INTEGER,
    ipAdEntNetMask
            IpAddress,
    ipAdEntBcastAddr
            IpAddress,
    ipAdEntReasmMaxSize
            INTEGER (0..65535)
}
```

Further definitions must be given to assign numeric values to *ipAddrEntry* and to each item in the *IpAddrEntry* sequence. For example, the definition:

```
ipAddrEntry { ipAddrTable 1 }
```

specifies that *ipAddrEntry* falls under *ipAddrTable* and has numeric value *1*. Similarly, the definition:

```
ipAdEntNetMask { ipAddrEntry 3 }
```

assigns *ipAdEntNetMask* numeric value *3* under *ipAddrEntry.*

We said that *ipAddrTable* was like a one-dimensional array. However, there is a significant difference in the way programmers use arrays and the way network management software uses tables in the MIB. Programmers think of an array as a set of elements that have an index used to select a specific element. For example, the programmer might write *xyz[3]* to select the third element from array *xyz*. ASN.1 syntax does not use integer indices. Instead, MIB tables append a suffix onto the name to select a specific element in the table. For our example of an IP address table, the standard specifies that the suffix used to select an item consists of an IP address. Syntactically, the IP address (in dotted decimal notation) is concatenated onto the end of the object name to form the reference. Thus, to specify the network mask field in the IP address table entry corresponding to address 128.10.2.3, one uses the name:

iso.org.dod.internet.mgmt.mib.ip.ipAddrTable.ipAddrEntry.ipAdEntNetMask.128.10.2.3

which, in numeric form, becomes:

1.3.6.1.2.1.4.20.1.3.128.10.2.3

Although concatenating an index to the end of a name may seem awkward, it provides a powerful tool that allows clients to search tables without knowing the number of items or the type of data used as an index. The next section shows how network management protocols use this feature to step through a table one element at a time.

30.9 Simple Network Management Protocol

Network management protocols specify communication between the network management client program a manager invokes and a network management server program executing on a host or router. In addition to defining the form and meaning of messages exchanged and the representation of names and values in those messages, network management protocols also define administrative relationships among routers being managed. That is, they provide for authentication of managers.

One might expect network management protocols to contain a large number of commands. Some early protocols, for example, supported commands that allowed the manager to: *reboot* the system, *add* or *delete* routes, *disable* or *enable* a particular network interface, or *remove cached address bindings*. The main disadvantage of building management protocols around commands arises from the resulting complexity. The protocol requires a separate command for each operation on a data item. For example, the command to delete a routing table entry differs from the command to disable an interface. As a result, the protocol must change to accommodate new data items.

SNMP takes an interesting alternative approach to network management. Instead of defining a large set of commands, SNMP casts all operations in a *fetch-store paradigm*†. Conceptually, SNMP contains only two commands that allow a manager to fetch a value from a data item or store a value into a data item. All other operations are defined as side-effects of these two operations. For example, although SNMP does not

†The fetch-store paradigm comes from a management protocol system known as HEMS. See Partridge and Trewitt [RFCs 1021, 1022, 1023, and 1024] for details.

have an explicit *reboot* operation, an equivalent operation can be defined by declaring a data item that gives the time until the next reboot and allowing the manager to assign the item a value (including zero).

The chief advantages of using a fetch-store paradigm are stability, simplicity, and flexibility. SNMP is especially stable because its definition remains fixed, even though new data items are added to the MIB and new operations are defined as side-effects of storing into those items. SNMP is simple to implement, understand, and debug because it avoids the complexity of having special cases for each command. Finally, SNMP is especially flexible because it can accommodate arbitrary commands in an elegant framework.

From a manager's point of view, of course, SNMP remains hidden. The user interface to network management software can phrase operations as imperative commands (e.g., *reboot*). Thus, there is little visible difference between the way a manager uses SNMP and other network management protocols. In fact, vendors sell network management software that offers a graphical user interface. Such software displays diagrams of network connectivity, and uses a point-and-click style of interaction.

As Figure 30.6 shows, SNMP offers more than the two operations we have described.

Command	Meaning
get-request	Fetch a value from a specific variable
get-next-request	Fetch a value without knowing its exact name
get-bulk-request	Fetch a large volume of data (e.g., a table)
response	A response to any of the above requests
set-request	Store a value in a specific variable
inform-request	Reference to third-part data (e.g., for a proxy)
snmpv2-trap	Reply triggered by an event
report	Undefined at present

Figure 30.6 The set of possible SNMP operations. *Get-next-request* allows the manager to iterate through a table of items.

Operations *get-request* and *set-request* provide the basic fetch and store operations; *response* provides the reply. SNMP specifies that operations must be *atomic*, meaning that if a single SNMP message specifies operations on multiple variables, the server either performs all operations or none of them. In particular, no assignments will be made if any of them are in error. The *trap* operation allows managers to program servers to send information when an event occurs. For example, an SNMP server can be programmed to send a manager a *trap* message whenever one of the attached networks becomes unusable (i.e., an interface goes down).

30.9.1 Searching Tables Using Names

We said that ASN.1 does not provide mechanisms for declaring arrays or indexing them in the usual sense. However, it is possible to denote individual elements of a table by appending a suffix to the object identifier for the table. Unfortunately, a client program may wish to examine entries in a table for which it does not know all valid suffixes. The *get-next-request* operation allows a client to iterate through a table without knowing how many items the table contains. The rules are quite simple. When sending a *get-next-request*, the client supplies a prefix of a valid object identifier, *P*. The agent examines the set of object identifiers for all variables it controls, and sends a response for the variable that occurs next in lexicographic order. That is, the agent must know the ASN.1 names of all variables and be able to select the first variable with object identifier greater than *P*. Because the MIB uses suffixes to index a table, a client can send the prefix of an object identifier corresponding to a table and receive the first element in the table. The client can send the name of the first element in a table and receive the second, and so on.

Consider an example search. Recall that the *ipAddrTable* uses IP addresses to identify entries in the table. A client that does not know which IP addresses are in the table on a given router cannot form a complete object identifier. However, the client can still use the *get-next-request* operation to search the table by sending the prefix:

iso **.** *org* **.** *dod* **.** *internet* **.** *mgmt* **.** *mib* **.** *ip* **.** *ipAddrTable* **.** *ipAddrEntry* **.** *ipAdEntNetMask*

which, in numeric form, is:

1 **.** *3* **.** *6* **.** *1* **.** *2* **.** *1* **.** *4* **.** *20* **.** *1* **.** *3*

The server returns the network mask field of the first entry in *ipAddrTable*. The client uses the full object identifier returned by the server to request the next item in the table.

30.10 SNMP Message Format

Unlike most TCP/IP protocols, SNMP messages do not have fixed fields. Instead, they use the standard ASN.1 encoding. Thus, a message can be difficult for humans to decode and understand. After examining the SNMP message definition in ASN.1 notation, we will review the ASN.1 encoding scheme briefly, and see an example of an encoded SNMP message.

Figure 30.7 shows how an SNMP message can be described with an ASN.1-style grammar. In general, each item in the grammar consists of a descriptive name followed by a declaration of the item's type. For example, an item such as

msgVersion INTEGER (0..2147483647)

declares the name *msgVersion* to be a nonnegative integer less than or equal to 2147483647.

```
SNMPv3Message ::=
    SEQUENCE {
        msgVersion  INTEGER (0..2147483647),
           -- note: version number 3 is used for SNMPv3
        msgGlobalData  HeaderData,
        msgSecurityParameters  OCTET STRING,
        msgData  ScopedPduData
    }
```

Figure 30.7 The SNMP message format in ASN.1-style notation. Text following two consecutive dashes is a comment.

As the figure shows, each SNMP message consists of four main parts: an integer that identifies the protocol *version*, additional header data, a set of security parameters, and a data area that carries the payload. A precise definition must be supplied for each of the terms used. For example, Figure 30.8 illustrates how the contents of the *HeaderData* section can be specified.

```
HeaderData ::= SEQUENCE {
   msgID  INTEGER (0..2147483647),
     --  used to match responses with requests
   msgMaxSize  INTEGER (484..2147483647),
     --  maximum size reply the sender can accept
   msgFlags  OCTET STRING (SIZE(1)),
     --  Individual flag bits specify message characteristics
     --  bit 7 authorization used
     --  bit 6 privacy used
     --  bit 5 reportability (i.e., a response needed)
   msgSecurityModel  INTEGER (1..2147483647)
     -- determines exact format of security parameters that follow
}
```

Figure 30.8 The definition of the *HeaderData* area in an SNMP message.

The data area in an SNMP message is divided into *protocol data units* (*PDUs*). Each PDU consists of a request (sent by client) or a response (sent by an agent). SNMPv3 allows each PDU to be sent as plain text or to be encrypted for privacy. Thus, the grammar specifies a *CHOICE*. In programming language terminology, the concept is known as a *discriminated union*.

```
ScopedPduData ::= CHOICE {
   plaintext  ScopedPDU,
   encryptedPDU  OCTET STRING  -- encrypted ScopedPDU value
}
```

An encrypted PDU begins with an identifier of the *engine*† that produced it. The engine ID is followed by the name of the context and the octets of the encrypted message.

```
ScopedPDU ::= SEQUENCE {
   contextEngineID  OCTET STRING,
   contextName  OCTET STRING,
   data  ANY                 -- e.g., a PDU as defined below
}
```

The item labeled *data* in the *ScopedPDU* definition has a type *ANY* because field *contextName* defines the exact details of the item. The SNMPv3 Message Processing Model (*v3MP*) specifies that the data must consist of one of the SNMP PDUs as Figure 30.9 illustrates:

```
PDU ::=
   CHOICE {
      get-request
         GetRequest-PDU,
      get-next-request
         GetNextRequest-PDU,
      get-bulk-request
         GetBulkRequest-PDU,
      response
         Response-PDU,
      set-request
         SetRequest-PDU,
      inform-request
         InformRequest-PDU,
      snmpV2-trap
         SNMPv2-Trap-PDU,
      report
         Report-PDU,
   }
```

Figure 30.9 The ASN.1 definitions of an SNMP PDU. The syntax for each request type must be specified further.

The definition specifies that each protocol data unit consists of one of eight types. To complete the definition of an SNMP message, we must further specify the syntax of the eight individual types. For example, Figure 30.10 shows the definition of a *get-request*.

†SNMPv3 distinguishes between an *application* that uses the service SNMP supplies and an *engine*, which is the underlying software that transmits requests and receives responses.

```
GetRequest-PDU ::= [0]
    IMPLICIT SEQUENCE {
        request-id
            Integer32,
        error-status
            INTEGER (0..18),
        error-index
            INTEGER (0..max-bindings),
        variable-bindings
            VarBindList
    }
```

Figure 30.10 The ASN.1 definition of a *get-request* message. Formally, the message is defined to be a *GetRequest-PDU*.

Further definitions in the standard specify the remaining undefined terms. Both *error-status* and *error-index* are single octet integers which contain the value zero in a request. If an error occurs, the values sent in a response identify the cause of the error. Finally, *VarBindList* contains a list of object identifiers for which the client seeks values. In ASN.1 terms, the definitions specify that *VarBindList* is a sequence of pairs of object name and value. ASN.1 represents the pairs as a sequence of two items. Thus, in the simplest possible request, *VarBindList* is a sequence of two items: a name and a *null*.

30.11 Example Encoded SNMP Message

The encoded form of ASN.1 uses variable-length fields to represent items. In general, each field begins with a header that specifies the type of object and its length in bytes. For example, each *SEQUENCE* begins with an octet containing 30 (hexadecimal); the next octet specifies the number of following octets that comprise the sequence.

Figure 30.11 contains an example SNMP message that illustrates how values are encoded into octets. The message is a *get-request* that specifies data item *sysDescr* (numeric object identifier *1.3.6.1.2.1.1.1.0*). Because the example shows an actual message, it includes many details. In particular, the message contains a *msgSecurityParameters* section which has not been discussed above. This particular message uses the *UsmSecurityParameters* form of security parameters. It should be possible, however, to correlate other sections of the message with the definitions above.

```
   30       67       02       01       03
SEQUENCE len=103 INTEGER  len=1    vers=3

   30       0D       02       01       2A
SEQUENCE len=13 INTEGER  len=1   msgID=42

   02       02       08       00
INTEGER  len=2    maxmsgsize=2048

   04       01       04
 string  len=1  msgFlags=0x04 (bits mean noAuth, noPriv, reportable)

   02       01       03
INTEGER  len=1  used-based security

   04       25       30       23
 string  len=37 SEQUENCE len=35 UsmSecurityParameters

   04       0C       00       00       00       63       00       00       00
 string  len=12  msgAuthoritativeEngineID ...
   A1       C0       93       8E       23
engine is at IP address 192.147.142.35, port 161

   02       01       00
INTEGER  len=1  msgAuthoritativeEngineBoots=0

   02       01       00
INTEGER  len=1  msgAuthoritativeEngineTime=0

   04       09       43       6F       6D       65       72       42       6F
 string  len=9    -----msgUserName value is "ComerBook"-------------
   6F       6B
-------------

   04       00
 string  len=0   msgAuthenticationParameters (none)

   04       00
 string  len=0   msgPrivacyParameters (none)

   30       2C
SEQUENCE len=44   ScopedPDU

   04       0C       00       00       00       63       00       00
 string  len=12    -------------------contextEngineID-------
   00       A1       c0       93       8E       23
   -----------------------------------------

   04       00
 string  len=0  contextName = "" (default)
```

```
CONTEXT [0] IMPLICIT SEQUENCE

  A0      1A
 getreq. len=26

  02      02      4D      C6
 INTEGER len=2   request-id = 19910

  02      01      00
 INTEGER len=1 error-status = noError(0)

  02      01      00
 INTEGER len=1 error-index=0

  30      0E
SEQUENCE  len=14 VarBindList

  30      0C
SEQUENCE  len=12 VarBind

  06      08
 OBJECT IDENTIFIER name

  2B      06      01      02      01      01      01      00
 1.3  .   6   .   1   .   2   .   1   .   1   .   1   .   0 (sysDescr.0)

  05      00
 null    len=0 (no value specified)
```

Figure 30.11 The encoded form of an SNMPv3 *get-request* for data item *sysDescr* with octets shown in hexadecimal and a comment explaining their meaning below. Related octets have been grouped onto lines; they are contiguous in the message.

As Figure 30.11 shows, the message starts with a code for *SEQUENCE* which has a length of 103 octets†. The first item in the sequence is a 1-octet integer that specifies the protocol *version*; the value *3* indicates that this is an SNMPv3 message. Successive fields define a message ID and the maximum message size the sender can accept in a reply. Security information, including the name of the user (*ComerBook*) follows the message header.

The *GetRequest-PDU* occupies the tail of the message. The sequence labeled *ScopedPDU* specifies a context in which to interpret the remainder of the message. The octet *A0* specifies the operation as a *get-Request*. Because the high-order bit is turned on, the interpretation of the octet is *context specific*. That is, the hexadecimal value *A0* only specifies a *GetRequest-PDU* when used in context; it is not a universally reserved value. Following the request octet, the length octet specifies the request is *26* octets long. The request ID is *2* octets, but each of the error-status and error-index are one oc-

†Sequence items occur frequently in an SNMP message because SNMP uses *SEQUENCE* instead of conventional programming language constructs like *array* or *struct*.

tet. Finally, the sequence of pairs contains one binding, a single object identifier bound to a *null* value. The identifier is encoded as expected except that the first two numeric labels are combined into a single octet.

30.12 New Features In SNMPv3

We said that version 3 of SNMP represents an evolution that follows and extends the basic framework of earlier versions. The primary changes arise in the areas of security and administration. The goals are twofold. First, SNMPv3 is designed to have both general and flexible security policies, making it possible for the interactions between a manager and managed devices to adhere to the security policies an organization specifies. Second, the system is designed to make administration of security easy.

To achieve generality and flexibility, SNMPv3 includes facilities for several aspects of security, and allows each to be configured independently. For example, v3 supports *message authentication* to ensure that instructions originate from a valid manager, *privacy* to ensure that no one can read messages as they pass between a manager's station and a managed device, and *authorization* and *view-based access control* to ensure that only authorized managers access particular items. To make the security system easy to configure or change, v3 allows *remote configuration*, meaning that an authorized manager can change the configuration of security items listed above without being physically present at the device.

30.13 Summary

Network management protocols allow a manager to monitor and control routers and hosts. A network management client program executing on the manager's workstation contacts one or more servers, called agents, running on the devices to be controlled. Because an internet consists of heterogeneous machines and networks, TCP/IP management software executes as application programs and uses internet transport protocols (e.g., UDP) for communication between clients and servers.

The standard TCP/IP network management protocol is SNMP, the Simple Network Management Protocol. SNMP defines a low-level management protocol that provides two conceptual operations: fetch a value from a variable or store a value into a variable. In SNMP, other operations occur as side-effects of changing values in variables. SNMP defines the format of messages that travel between a manager's computer and a managed entity.

A set of companion standards to SNMP define the set of variables that a managed entity maintains. The set of variables comprise a Management Information Base (*MIB*). MIB variables are described using ASN.1, a formal language that provides a concise encoded form as well as a precise human-readable notation for names and objects. ASN.1 uses a hierarchical namespace to guarantee that all MIB names are globally unique while still allowing subgroups to assign parts of the namespace.