Applications

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

—Winston Churchill

We started this book by talking about application programs—everything from Web browsers to videoconferencing tools—that people want to run over computer networks. In the intervening chapters, we have developed, one layer at a time, the networking infrastructure needed to make such applications possible. We have now come full circle, back to network applications. These applications are part network protocol (in the sense that they exchange messages with their peers on other machines) and part traditional application program (in the sense that they interact with the windowing system, the file system, and ultimately, the user). This chapter explores some of the most popular network applications available today, with a focus on their protocols. What you will quickly recognize is that a protocol is a protocol, no matter what layer it runs at. Said another way, the best way to prepare yourself to write network applications is to first understand how to design good network protocols.

PROBLEM

Applications Need Their Own Protocols

The first example we look at—a distributed name service—also happens to be the first application implemented on a network. Although it technically qualifies as a network application—it is, in effect, a distributed database built on top of the underlying transport protocols—it is not an application that users normally invoke explicitly. Nevertheless, it is an application that all other applications depend upon. This is because the name server is used to translate host names into host addresses; the existence of such an application allows the users of other applications to refer to remote hosts by name rather than by address. In other words, a name server is usually used by other applications, rather than by humans.

9

We then turn our attention to describing a variety of familiar, and not so familiar, network applications. These range from exchanging email and surfing the Web, to managing a set of network elements, to multimedia applications like vic and vat, to emerging peer-to-peer and content distribution networks. This list is by no means exhaustive, but it does serve to illustrate the trick in designing application-level protocols, which is to augment the underlying transport services of TCP and UDP so as to provide the precise communication service required by the application.

9.1 Name Service (DNS)

Up to this point, we have been using addresses to identify hosts. While perfectly suited for processing by routers, addresses are not exactly user friendly. It is for this reason that a unique *name* is also typically assigned to each host in a network. This section describes how a naming service can be developed to map user-friendly names into router-friendly addresses. Such a service is often the first application program implemented in a network since it frees other applications to identify hosts by name rather than by address. Name services are sometimes called *middleware* because they fill a gap between applications and the underlying network.

Host names differ from host addresses in two important ways. First, they are usually of variable length and mnemonic, thereby making them easier for humans to remember. (In contrast, fixed-length numeric addresses are easier for routers to process.) Second, names typically contain no information that helps the network locate (route packets toward) the host. Addresses, in contrast, sometimes have routing information embedded in them; *flat* addresses (those not divisible into component parts) are the exception.

Before getting into the details of how hosts are named in a network, we first introduce some basic terminology. First, a *name space* defines the set of possible names. A name space can be either *flat* (names are not divisible into components), or it can be *hierarchical* (Unix file names are the obvious example). Second, the naming system maintains a collection of *bindings* of names to values. The value can be anything we want the naming system to return when presented with a name; in many cases it is an address. Finally, a *resolution mechanism* is a procedure that, when invoked with a name, returns the corresponding value. A *name server* is a specific implementation of a resolution mechanism that is available on a network and that can be queried by sending it a message.

Because of its large size, the Internet has a particularly well-developed naming system in place—the *domain name system* (DNS). We therefore use DNS as a framework for discussing the problem of naming hosts. Note that the Internet did not always use DNS. Early in its history, when there were only a few hundred hosts on the Internet, a central authority called the Network Information Center (NIC) maintained a flat table of name-to-address bindings; this table was called `hosts.txt`. Whenever a site wanted to add a new host to the Internet, the site administrator sent email to the NIC giving the new host's name/address pair. This information was manually entered into the table, the modified table was mailed out to the various sites every few days, and the system administrator at each site installed the table on every host at the site. Name resolution was then simply implemented by a procedure that looked up a host's name in the local copy of the table and returned the corresponding address.

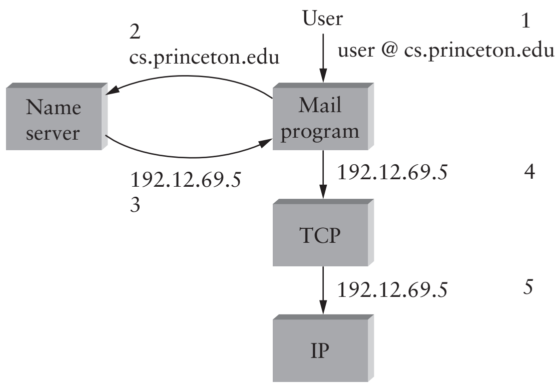


Figure 9.1 Names translated into addresses, where the numbers 1–5 show the sequence of steps in the process.

It should come as no surprise that the hosts.txt approach to naming did not work well as the number of hosts in the Internet started to grow. Therefore, in the mid-1980s, the domain naming system was put into place. DNS employs a hierarchical name space rather than a flat name space, and the "table" of bindings that implements this name space is partitioned into disjoint pieces and distributed throughout the Internet. These subtables are made available in name servers that can be queried over the network.

What happens in the Internet is that a user presents a host name to an application program (possibly embedded in a compound name such as an email address or URL), and this program engages the naming system to translate this name into a host address. The application then opens a connection to this host by presenting some transport protocol (e.g., TCP) with the host's IP address. This situation is illustrated (in the case of sending email) in Figure 9.1.

9.1.1 Domain Hierarchy

DNS implements a hierarchical name space for Internet objects. Unlike Unix file names, which are processed from left to right with the naming components separated with slashes, DNS names are processed from right to left and use periods as the separator. (Although they are "processed" from right to left, humans still "read" domain names from left to right.) An example domain name for a host is cicada.cs.princeton.edu. Notice that we said domain names are used to name Internet "objects." What we mean by this is that DNS is not strictly used to map host names into host addresses. It is more accurate to say that DNS maps domain names into values. For the time being, we assume that these values are IP addresses; we will come back to this issue later in this section.

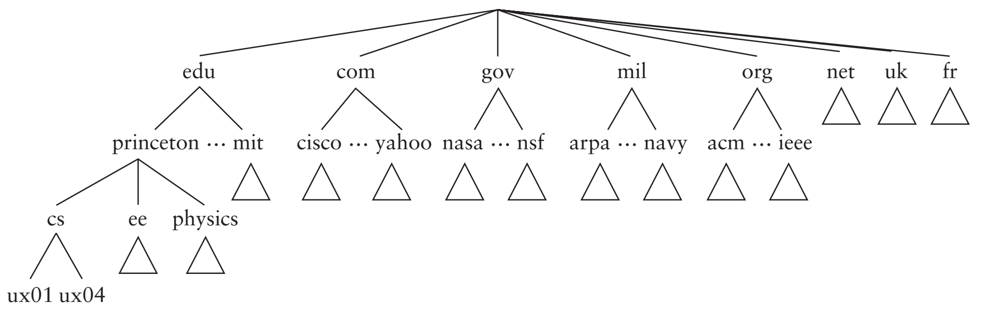


Figure 9.2 Example of a domain hierarchy.

Like the Unix file hierarchy, the DNS hierarchy can be visualized as a tree, where each node in the tree corresponds to a domain, and the leaves in the tree correspond to the hosts being named. Figure 9.2 gives an example of a domain hierarchy. Note that we should not assign any semantics to the term "domain" other than that it is simply a context in which additional names can be defined.

There was actually a substantial amount of discussion that took place when the domain name hierarchy was first being developed as to what conventions would govern the names that were to be handed out near the top of the hierarchy. Without going into that discussion in any detail, notice that the hierarchy is not very wide at the first level. There are domains for each country, plus the "big six" domains: edu, com, gov, mil, org, and net. These six domains are all based in the United States; the only domain names that don't explicitly specify a country are those in the United States. Aside from this U.S. bias, you might notice a military bias in the hierarchy. This is easy to explain, since the development of DNS was originally funded by ARPA, the major research arm of the U.S. Department of Defense.

9.1.2 Name Servers

The complete domain name hierarchy exists only in the abstract. We now turn our attention to the question of how this hierarchy is actually implemented. The first step is to partition the hierarchy into subtrees called *zones*. For example, Figure 9.3 shows how the hierarchy given in Figure 9.2 might be divided into zones. Each zone can be thought of as corresponding to some administrative authority that is responsible for that portion of the hierarchy. For example, the top level of the hierarchy forms a zone that is managed by the NIC. Below this is a zone that corresponds to Princeton University. Within this zone, some departments do not want the responsibility of managing the hierarchy (and so they remain in the university-level zone), while others, like the Department of Computer Science, manage their own department-level zone.

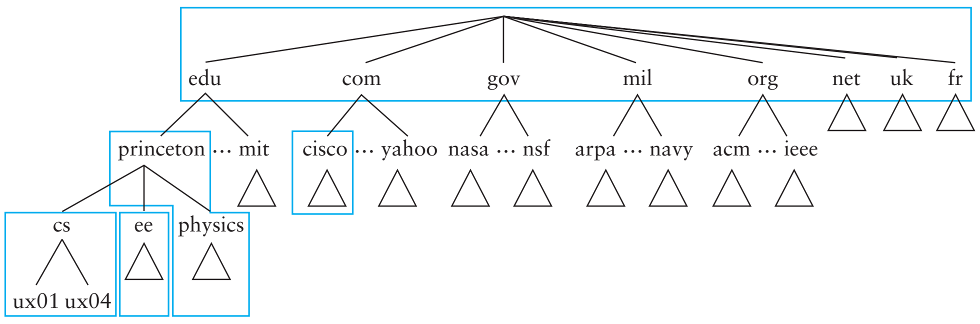


Figure 9.3 Domain hierarchy partitioned into zones.

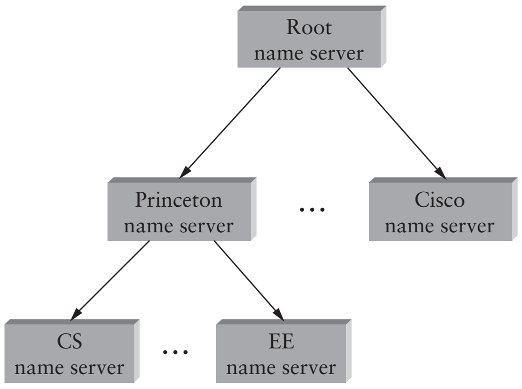


Figure 9.4 Hierarchy of name servers.

The relevance of a zone is that it corresponds to the fundamental unit of implementation in DNS—the name server. Specifically, the information contained in each zone is implemented in two or more name servers. Each name server, in turn, is a program that can be accessed over the Internet. Clients send queries to name servers, and name servers respond with the requested information. Sometimes the response contains the final answer that the client wants, and sometimes the response contains a pointer to another server that the client should query next. Thus, from an implementation perspective, it is more accurate to think of DNS as being represented by a hierarchy of name servers rather than by a hierarchy of domains, as illustrated in Figure 9.4.

Note that each zone is implemented in two or more name servers for the sake of redundancy; that is, the information is still available even if one name server fails. On the flip side, a given name server is free to implement more than one zone.

Each name server implements the zone information as a collection of *resource records*. In essence, a resource record is a name-to-value binding, or more specifically, a 5-tuple that contains the following fields:

⟨Name, Value, Type, Class, TTL⟩

The Name and Value fields are exactly what you would expect, while the Type field specifies how the Value should be interpreted. For example, Type = A indicates that the Value is an IP address. Thus, A records implement the name-to-address mapping we have been assuming. Other record types include

- NS: The Value field gives the domain name for a host that is running a name server that knows how to resolve names within the specified domain.
- CNAME: The Value field gives the canonical name for a particular host; it is used to define aliases.
- MX: The Value field gives the domain name for a host that is running a mail server that accepts messages for the specified domain.

The Class field was included to allow entities other than the NIC to define useful record types. To date, the only widely used Class is the one used by the Internet; it is denoted IN. Finally, the TTL field shows how long this resource record is valid. It is used by servers that cache resource records from other servers; when the TTL expires, the server must evict the record from its cache.

To better understand how resource records represent the information in the domain hierarchy, consider the following examples drawn from the domain hierarchy given in Figure 9.2. To simplify the examples, we ignore the TTL field and we give the relevant information for only one of the name servers that implement each zone.

First, the root name server contains an NS record for each second-level server. It also has an A record that translates this name into the corresponding IP address. Taken together, these two records effectively implement a pointer from the root name server to each of the second-level servers.

⟨princeton.edu, cit.princeton.edu, NS, IN⟩

⟨cit.princeton.edu, 128.196.128.233, A, IN⟩

⟨cisco.com, ns.cisco.com, NS, IN⟩

⟨ns.cisco.com, 128.96.32.20, A, IN⟩

⋮

Next, the domain princeton.edu has a name server available on host cit.princeton.edu that contains the following records. Note that some of these records give the final answer (e.g., the address for host saturn.physics.princeton.edu), while others point to third-level name servers.

⟨ cs.princeton.edu, gnat.cs.princeton.edu, NS, IN ⟩

⟨ gnat.cs.princeton.edu, 192.12.69.5, A, IN ⟩

⟨ ee.princeton.edu, helios.ee.princeton.edu, NS, IN ⟩

⟨ helios.ee.princeton.edu, 128.196.28.166, A, IN ⟩

⟨ jupiter.physics.princeton.edu, 128.196.4.1, A, IN ⟩

⟨ saturn.physics.princeton.edu, 128.196.4.2, A, IN ⟩

⟨ mars.physics.princeton.edu, 128.196.4.3, A, IN ⟩

⟨ venus.physics.princeton.edu, 128.196.4.4, A, IN ⟩

⋮

Finally, a third-level name server, such as the one managed by domain cs.princeton.edu, contains A records for all of its hosts. It might also define a set of aliases (CNAME records) for each of those hosts. Aliases are sometimes just convenient (e.g., shorter) names for machines, but they can also be used to provide a level of indirection. For example, www.cs.princeton.edu is an alias for the host named cicada.cs.princeton.edu. This allows the site's Web server to move to another machine without affecting remote users; they simply continue to use the alias without regard for what machine currently runs the domain's Web server. The mail exchange (MX) records serve the same purpose for the email application—it allows an administrator to change which host receives mail on behalf of the domain without having to change everyone's email address.

⟨ cs.princeton.edu, gnat.cs.princeton.edu, MX, IN ⟩

⟨ cicada.cs.princeton.edu, 192.12.69.60, A, IN ⟩

⟨ cic.cs.princeton.edu, cicada.cs.princeton.edu, CNAME, IN ⟩

⟨ gnat.cs.princeton.edu, 192.12.69.5, A, IN ⟩

⟨ gna.cs.princeton.edu, gnat.cs.princeton.edu, CNAME, IN ⟩

⟨ www.cs.princeton.edu, 192.12.69.35, A, IN ⟩

⟨ cicada.cs.princeton.edu, roach.cs.princeton.edu, CNAME, IN ⟩

⋮

Note that although resource records can be defined for virtually any type of object, DNS is typically used to name hosts (including servers) and sites. It is not used to name individual people, or other objects like files or directories; other naming systems are typically used to identify such objects. For example, X.500 is an ISO naming system designed to make it easier to identify people. It allows you to name a person by giving a set of attributes: name, title, phone number, postal address, and so on. X.500 proved too cumbersome—and in some sense, was usurped by powerful search engines now available on the Web—but it did eventually evolve into LDAP (Lightweight Directory Access Protocol). LDAP is a subset of X.500 originally designed as a PC front end to X.500. Today it is gaining in popularity, mostly at the enterprise level, as a system for learning information about users.

9.1.3 Name Resolution

Given a hierarchy of name servers, we now consider the issue of how a client engages these servers to resolve a domain name. To illustrate the basic idea, suppose the client wants to resolve the name cicada.cs.princeton.edu relative to the set of servers given in the previous subsection. The client first sends a query containing this name to the root server. The root server, unable to match the entire name, returns the best match it has—the NS record for princeton.edu. The server also returns all records that are related to this record, in this case, the A record for cit.princeton.edu. The client, having not received the answer it was after, next sends the same query to the name server at IP host 128.196.128.233. This server also cannot match the whole name, and so returns the NS and corresponding A records for the cs.princeton.edu domain. Finally, the client sends the same query as before to the server at IP host 192.12.69.5, and this time gets back the A record for cicada.cs.princeton.edu.

Naming Conventions

Our description of DNS focuses on the underlying *mechanisms*, that is, how the hierarchy is partitioned over multiple servers and how the resolution process works. There is an equally interesting, but much less technical, issue of the *conventions* that are used to decide the names to use in the mechanism. For example, it is by convention that all U.S. universities are under the edu domain, while English universities are under the ac (academic) subdomain of the uk (United Kingdom) domain. In fact, the very existence of the uk domain, rather than a gb (Great Britain) domain, was a source of great controversy in the early days of DNS, since the latter does not include Northern Ireland.

The thing to understand about conventions is that they are sometimes defined without anyone making an explicit decision. For example, by convention a site hides the exact host that serves as its mail exchange behind the MX record.

This example still leaves a couple of questions about the resolution process unanswered. The first question is, How did the client locate the root server in the first place, or said another way, How do you resolve the name of the server that knows how to resolve names? This is a fundamental problem in any naming system, and the answer is that the system has to be bootstrapped in some way. In this case, the name-to-address mapping for one or more root servers is well known, that is, published through some means outside the naming system itself.

In practice, however, not all clients know about the root servers. Instead, the client program running on each Internet host is initialized with the address of a *local* name server. For example, all the hosts in the Department of Computer Science at Princeton know about the server on gnat.cs.princeton.edu. This local name server, in turn, has resource records for one or more of the root servers, for example:

⟨ 'root', venera.isi.edu, NS, IN ⟩

⟨ venera.isi.edu, 128.9.0.32, A, IN ⟩

Thus, resolving a name actually involves a client querying the local server, which in turn acts as a client that queries the remote servers on the original client's behalf. This results in the client/server interactions illustrated in Figure 9.5. One advantage of this model is that all the hosts in the Internet do not have to be kept up-to-date on where the current root servers are located; only the servers have to know about the root. A second advantage is that the local server gets to see the answers that come back from queries that are posted by all the local clients. The local server caches these responses and is sometimes able to resolve future queries without having to go out over the network. The TTL field in the resource records returned by remote servers indicates how long each record can be safely cached.

An alternative would have been to adopt the convention of sending mail to user@mail.cs.princeton.edu, much as we expect to find a site's public FTP directory at ftp.cs.princeton.edu and its WWW server at www.cs.princeton.edu.

Conventions also exist at the local level, where an organization names its machines according to some consistent set of rules. Given that the host names venus, saturn, and mars are among the most popular in the Internet, it's not too hard to figure out one common naming convention. Some host naming conventions are more imaginative, however. For example, one site named its machines up, down, crashed, rebooting, and so on, resulting in confusing statements like "rebooting has crashed" and "up is down." Of course, there are also less imaginative names, such as those who name their machines after the integers.

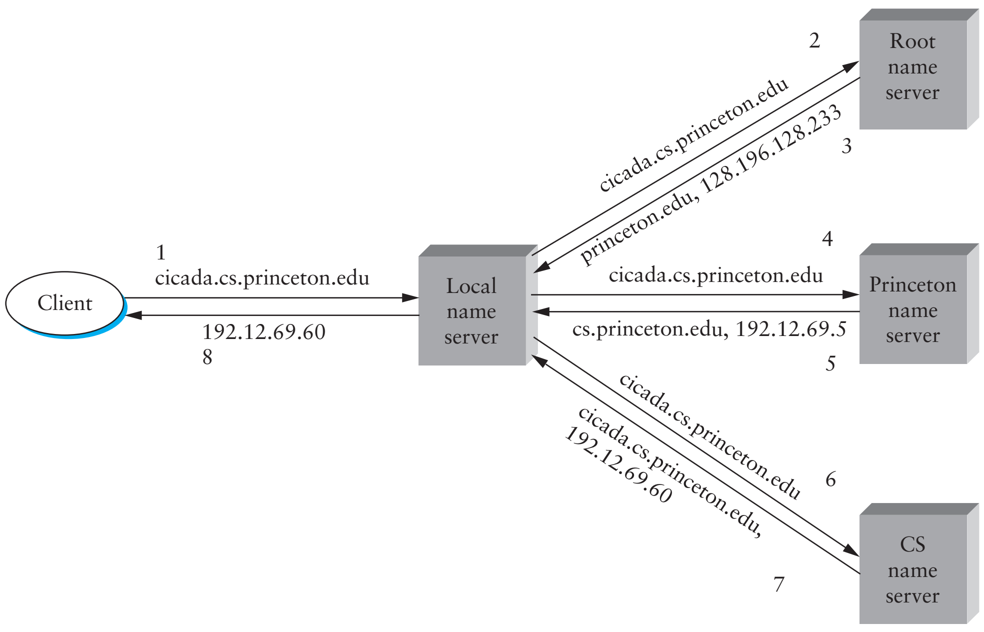


Figure 9.5 Name resolution in practice, where the numbers 1–8 show the sequence of steps in the process.

The second question is how the system works when a user submits a partial name (e.g., cicada) rather than a complete domain name (e.g., cicada.cs.princeton.edu). The answer is that the client program is configured with the local domain in which the host resides (e.g., cs.princeton.edu) and it appends this string to any simple names before sending out a query.

Just to make sure we are clear, we have now seen three different levels of identifiers—domain names, IP addresses, and physical network addresses—and the mapping of identifiers at one level into identifiers at another level happens at different points in the network architecture. First, users specify domain names when interacting with the application. Second, the application engages DNS to translate this name into an IP address; it is the IP address that is placed in each datagram, not the domain name. (As an aside, this translation process involves IP datagrams being sent over the Internet, but these datagrams are addressed to a host that runs a name server, not to the ultimate destination.) Third, IP does forwarding at each router, which often means that it maps one IP address into another; that is, it maps the ultimate destination's address into the address for the next hop router. Finally, IP engages ARP to translate the next hop IP address into the physical address for that machine; the next hop might be the ultimate destination or it might be an intermediate router. Frames sent over the physical network have these physical addresses in their headers.

9.2 Traditional Applications

The domain name system may be an essential Internet application, but it's one that users only indirectly interact with. We now turn our attention to those applications that are directly invoked by users, focusing on two of the most popular—the World Wide Web and email. We also look at network management, which although not so familiar to the average user, is the application of choice for system administrators. Like DNS, all three applications employ the request/reply paradigm—users send requests to servers, which then respond accordingly. We refer to these as "traditional" applications because they typify the sort of applications that have existed since the early days of computer networks. By contrast, the next two sections looks at a class of applications that have become feasible only relatively recently: streaming applications (e.g., multimedia applications like video and audio) and various overlay-based applications.

Before taking a close look at each of these applications, there are three general points that we need to make. The first is that it is important to distinguish between application *programs* and application *protocols*. For example, the HyperText Transport Protocol (HTTP) is an application protocol that is used to retrieve Web pages from remote servers. There can be many different application programs—that is, Web clients like Internet Explorer, Mosaic, and Netscape—that provide users with a different look and feel, but all of them use the same HTTP protocol to communicate with Web servers over the Internet. This section focuses on three application protocols:

- SMTP: Simple Mail Transfer Protocol is used to exchange electronic mail.
- HTTP: HyperText Transport Protocol is used to communicate between Web browsers and Web servers.
- SNMP: Simple Network Management Protocol is used to query (and sometimes modify) the state of remote network nodes.

The second point is that since all of the application protocols described in this section follow the same request/reply communication pattern, we would expect that they are all built on top of an RPC transport protocol. This is not the case, however, as they are all implemented on top of either TCP or UDP. In effect, each protocol reinvents a simple RPC-like mechanism on top of one of the existing transport protocols. We say "simple" because each protocol is not designed to support arbitrary remote procedure calls, but is instead designed to send and respond to a specific set of request messages.

In fact, it is no coincidence that two of the protocols have the word "Simple" in their name.

All three protocols have a companion protocol that specifies the format of the data that can be exchanged. This is one reason these protocols are relatively simple: Much of the complexity is managed in this companion document. For example, SMTP is a protocol for exchanging electronic mail messages, but RFC 822 (this specification has no other name) and MIME (Multipurpose Internet Mail Extensions) define the format of email messages. Similarly, HTTP is a protocol for fetching Web pages, but HTML (HyperText Markup Language) is a companion specification that defines the form of those pages. Finally, SNMP is a protocol for querying a network node, but MIB (management information base) defines the variables that can be queried.

9.2.1 Electronic Mail (SMTP, MIME, IMAP)

Email is one of the oldest network applications. After all, what could be more natural than wanting to send a message to the user at the other end of a crosscountry link you just managed to get running? In fact, the pioneers of the ARPANET had not really envisioned email as a key application when the network was created—remote access to computing resources was the main design goal—but it turned out to be a surprisingly successful application. Out of this work evolved the Internet's email system, which is now used by millions of people every day.

As with all the applications described in this section, the place to start in understanding how email works is (1) to distinguish the user interface (i.e., your mail reader) from the underlying message transfer protocol (in this case, SMTP), and (2) to distinguish between this transfer protocol and a companion protocol (RFC 822 and MIME) that defines the format of the messages being exchanged. We start by looking at the message format.

Message Format

RFC 822 defines messages to have two parts: a *header* and a *body*. Both parts are represented in ASCII text. Originally, the body was assumed to be simple text. This is still the case, although RFC 822 has been augmented by MIME to allow the message body to carry all sorts of data. This data is still represented as ASCII text, but because it may be an encoded version of, say, a JPEG image, it's not necessarily readable by human users. More on MIME in a moment.

The message header is a series of <CRLF>-terminated lines. (<CRLF> stands for carriage-return + line-feed, which are a pair of ASCII control characters often used to indicate the end of a line of text.) The header is separated from the message body by a blank line. Each header line contains a type and value separated by a colon. Many of these header lines are familiar to users since they are asked to fill them out when they compose an email message. For example, the To: header identifies the message recipient, and the Subject: header says something about the purpose of the message. Other headers are filled in by the underlying mail delivery system. Examples include Date: (when the message was transmitted), From: (what user sent the message), and Received: (each mail server that handled this message). There are, of course, many other header lines; the interested reader is referred to RFC 822.

RFC 822 was extended in 1993 (and updated again in 1996) to allow email messages to carry many different types of data: audio, video, images, Word documents, and so on. MIME consists of three basic pieces. The first piece is a collection of header lines that augment the original set defined by RFC 822. These header lines describe, in various ways, the data being carried in the message body. They include MIME-Version: (the version of MIME being used), Content-Description: (a human-readable description of what's in the message, analogous to the Subject: line), Content-Type: (the type of data contained in the message), and Content-Transfer-Encoding: (how the data in the message body is encoded).

The second piece is definitions for a set of content types (and subtypes). For example, MIME defines two different still image types, denoted image/gif and image/jpeg, each with the obvious meaning. As another example, text/plain refers to simple text you might find in a vanilla 822-style message, while text/richtext denotes a message that contains "marked up" text (e.g., text using special fonts, italics, etc.). As a third example, MIME defines an application type, where the subtypes correspond to the output of different application programs (e.g., application/postscript and application/msword).

MIME also defines a multipart type that says how a message carrying more than one data type is structured. This is like a programming language that defines both base types (e.g., integers and floats) and compound types (e.g., structures and arrays). One possible multipart subtype is mixed, which says that the message contains a set of independent data pieces in a specified order. Each piece then has its own header line that describes the type of that piece.

The third piece is a way to encode the various data types so they can be shipped in an ASCII email message. The problem is that for some data types (a JPEG image, for example), any given 8-bit byte in the image might contain one of 256 different values. Only a subset of these values are valid ASCII characters. It is important that email messages contain only ASCII, because they might pass through a number of intermediate systems (gateways, as described below) that assume all email is ASCII and would corrupt the message if it contained non-ASCII characters. To address this issue, MIME uses a straightforward encoding of binary data into the ASCII character set. The encoding is called base64. The idea is to map every three bytes of the original binary data into four ASCII characters. This is done by grouping the binary data into 24-bit units, and breaking each such unit into four 6-bit pieces. Each 6-bit piece maps onto one of 64 valid ASCII characters; for example, 0 maps onto *A*, 1 maps onto *B*, and so on. If you look at a message that has been encoded using the base64 encoding scheme, you'll notice only the 52 upper- and lowercase letters, the 10 digits 0 through 9, and the special characters + and /. These are the first 64 values in the ASCII character set.

As one aside, so as to make reading mail as painless as possible for those of us that insist on using text-only mail readers, a MIME message that consists of regular text only can be encoded using 7-bit ASCII. There's also a readable encoding for mostly ASCII data.

Putting this all together, a message that contains some plain text, a JPEG image, and a PostScript file would look something like this:

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="-------417CA6E2DE4ABCAFBC5"
From: Alice Smith <Alice@cisco.com>
To: Bob@cs.Princeton.edu
Subject: promised material
Date: Mon, 07 Sep 1998 19:45:19 -0400

---------417CA6E2DE4ABCAFBC5
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Bob,

Here's the jpeg image and draft report I promised.

--Alice

---------417CA6E2DE4ABCAFBC5
Content-Type: image/jpeg
Content-Transfer-Encoding: base64
```

... unreadable encoding of a jpeg figure

```
---------417CA6E2DE4ABCAFBC5
Content-Type: application/postscript; name="draft.ps"
Content-Transfer-Encoding: 7bit
```

... readable encoding of a PostScript document

In this example, the Content-Type line in the message header says that this message contains various pieces, each denoted by a character string that does not appear in the data itself. Each piece then has its own Content-Type and Content-Transfer-Encoding lines.

Message Transfer

Next, we look at SMTP—the protocol used to transfer messages from one host to another. To place SMTP in the right context, we need to identify the key players. First, users interact with a *mail reader* when they compose, file, search, and read their email. There are countless mail readers available, just like there are many Web browsers to choose from. In fact, most Web browsers now include a mail reader. Second, there is a *mail daemon* (or process) running on each host. You can think of this process as playing the role of a post office: Mail readers give the daemon messages they want to send to other users, the daemon uses SMTP running over TCP to transmit the message to a daemon running on another machine, and the daemon puts incoming messages into the user's *mailbox* (where that user's mail reader can later find it). Since SMTP is a protocol that anyone could implement, in theory there could be many different implementations of the mail daemon. It turns out, though, that the mail daemon running on most hosts is derived from the sendmail program originally implemented on Berkeley Unix.

While it is certainly possible that the sendmail program on a sender's machine establishes an SMTP/TCP connection to the sendmail program on the recipient's machine, in many cases the mail traverses one or more *mail gateways* on its route from the sender's host to the receiver's host. Like the end hosts, these gateways also run a sendmail process. It's not an accident that these intermediate nodes are called "gateways" since their job is to store and forward email messages, much like an "IP gateway" (which we have referred to as a router) stores and forwards IP datagrams. The only difference is that a mail gateway typically buffers messages on disk and is willing to try retransmitting them to the next machine for several days, while an IP router buffers datagrams in memory and is only willing to retry transmitting them for a fraction of a second. Figure 9.6 illustrates a two-hop path from the sender to the receiver.

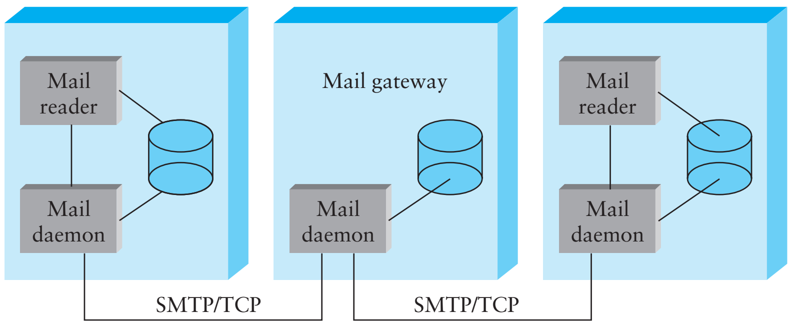


Figure 9.6 Sequence of mail gateways store and forward email messages.

Why, you might ask, are mail gateways necessary? Why can't the sender's host send the message to the receiver's host? One reason is that the recipient does not want to include the specific host on which he or she reads email in his or her address. For example, mail delivered to Bob@cs.princeton.edu is first sent to a mail gateway in the CS Department at Princeton (that is, to the host named cs.princeton.edu), and then forwarded—involving a second SMTP/TCP connection—to the specific machine on which Bob happens to be reading his email today. The forwarding gateway maintains a database that maps users into the machine on which they currently want to receive their mail; the sender need not be aware of this specific name. (The list of Received: header lines in the message will help you trace the mail gateways that a given message traversed.) Another reason is that the recipient's machine may not always be up, in which case the mail gateway holds the message until it can be delivered.

Independent of how many mail gateways are in the path, an independent SMTP connection is used between each host to move the message closer to the recipient. Each SMTP session involves a dialog between the two mail daemons, with one acting as the client and the other acting as the server. Multiple messages might be transferred between the two hosts during a single session. Since RFC 822 defines messages using ASCII as the base representation, it should come as no surprise to learn that SMTP is also ASCII based. This means it is possible for a human at a keyboard to pretend to be an SMTP client program.

SMTP is best understood by a simple example. The following is an exchange between sending host cs.princeton.edu and receiving host cisco.com. In this case, user Bob at Princeton is trying to send mail to users Alice and Tom at Cicso. The lines sent by cs.princeton.edu are shown in black and the lines sent by cisco.com are shown in green. Extra blank lines have been added to make the dialog more readable.

```
HELO cs.princeton.edu
250 Hello daemon@mail.cs.princeton.edu [128.12.169.24]

MAIL FROM:<Bob@cs.princeton.edu>
250 OK

RCPT TO:<Alice@cisco.com>
250 OK

RCPT TO:<Tom@cisco.com>
550 No such user here

DATA
354 Start mail input; end with <CRLF>.<CRLF>
Blah blah blah...
...etc. etc. etc.
```

```
<CRLF>.<CRLF>
250 OK

QUIT
221 Closing connection
```

As you can see, SMTP involves a sequence of exchanges between the client and the server. In each exchange, the client posts a command (e.g., HELO, MAIL, RCPT, DATA, QUIT) and the server responds with a code (e.g., 250, 550, 354, 221). The server also returns a human-readable explanation for the code (e.g., No such user here). In this particular example, the client first identifies itself to the server with the HELO command. It gives its domain name as an argument. The server verifies that this name corresponds to the IP address being used by the TCP connection; you'll notice the server states this IP address back to the client. The client then asks the server if it is willing to accept mail for two different users; the server responds by saying "yes" to one and "no" to the other. Then the client sends the message, which is terminated by a line with a single period (".") on it. Finally, the client terminates the connection.

There are, of course, many other commands and return codes. For example, the server can respond to a client's RCPT command with a 251 code, which indicates that the user does not have a mailbox on this host, but that the server promises to forward the message onto another mail daemon. In other words, the host is functioning as a mail gateway. As another example, the client can issue a VRFY operation to verify a user's email address, but without actually sending a message to the user.

The only other point of interest is the arguments to the MAIL and RCPT operations; for example, FROM:<Bob@cs.princeton.edu> and TO:<Alice@cisco.com>, respectively. These look a lot like 822 header fields, and in some sense, they are. What actually happens is that the mail daemon parses the message to extract the information it needs to run SMTP. The information it extracts is said to form an *envelope* for the message. The SMTP client uses this envelope to parameterize its exchange with the SMTP server. One historical note: The reason sendmail became so popular is that no one wanted to reimplement this message parsing function. While today's email addresses look pretty tame (e.g., Bob@cs.princeton.edu), this was not always the case. In the days before everyone was connected to the Internet, it was not uncommon to see email addresses of the form user%host@site!neighbor.

Mail Reader

The final step is for the user to actually retrieve his or her messages from the mailbox, read them, reply to them, and possibly save a copy for future reference. The user performs all these actions by interacting with a mail reader. In many cases, this reader is just a program running on the same machine as the user's mailbox resides, in which case it simply reads and writes the file that implements the mailbox. In other cases, the user accesses his or her mailbox from a remote machine using yet another protocol, such as the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP). It is beyond the scope of this book to discuss the user interface aspects of the mail reader, but it is definitely within our scope to talk about the access protocol. We consider IMAP, in particular.

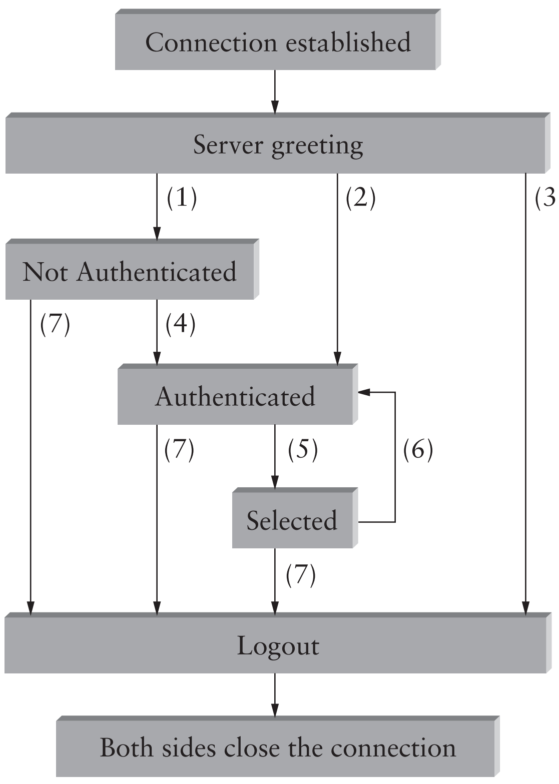
IMAP is similar to SMTP in many ways. It is a client/server protocol running over TCP, where the client (running on the user's desktop machine) issues commands in the form of <CRLF>-terminated ASCII text lines and the mail server (running on the machine that maintains the user's mailbox) responds in kind. The exchange begins with the client authenticating him- or herself, and identifying the mailbox he or she wants to access. This can be represented by the simple state transition diagram shown in Figure 9.7. In this diagram, LOGIN, AUTHENTICATE, SELECT, EXAMINE, CLOSE, and LOGOUT are example commands that the client can issue, while OK is one possible server response. Other common commands include FETCH, STORE, DELETE, and EXPUNGE, with the obvious meanings. Additional server responses include NO (client does not have permission to perform that operation) and BAD (command is ill-formed).

When the user asks to FETCH a message, the server returns it in MIME format and the mail reader decodes it. In addition to the message itself, IMAP also defines a set of message *attributes* that are exchanged as part of other commands, independent of transferring the message itself. Message attributes include information like the size of the message, but more interestingly, various *flags* associated with the message (e.g., Seen, Answered, Deleted, and Recent). These flags are used to keep the client and server synchronized; that is, when the user deletes a message in the mail reader, the client needs to report this fact to the mail server. Later, should the user decide to expunge all deleted messages, the client issues an EXPUNGE command to the server, which knows to actually remove all earlier deleted messages from the mailbox.

Finally, note that when the user replies to a message, or sends a new message, the mail reader does not forward the message from the client's desktop machine to the mail server using IMAP, but it instead uses SMTP. This means that the user's mail server is effectively the first mail gateway traversed along the path from the desktop to the recipient's mailbox.

9.2.2 World Wide Web (HTTP)

The World Wide Web has been so successful and has made the Internet accessible to so many people that sometimes it seems to be synonymous with the Internet. One helpful way to think of the Web is as a set of cooperating clients and servers, all of whom speak the same language: HTTP. Most people are exposed to the Web through

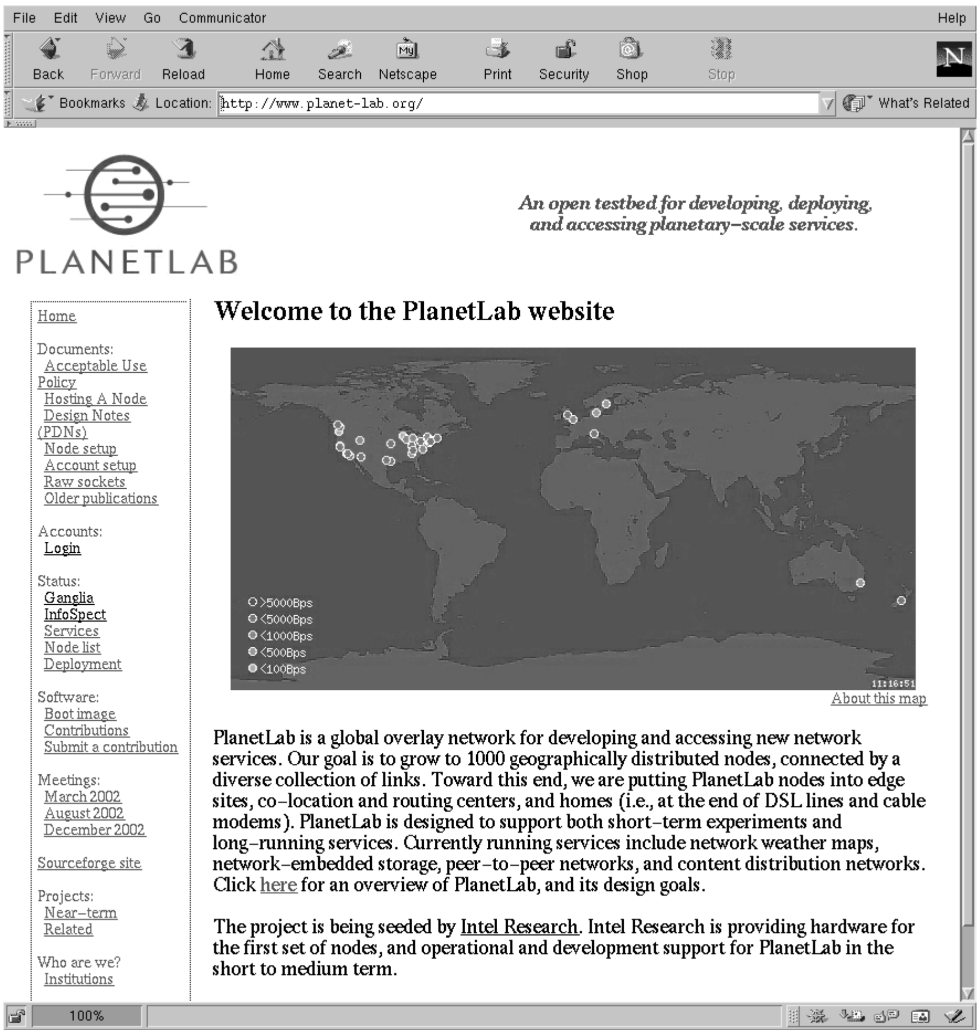


(1) connection without pre-authentication (OK greeting)
(2) pre-authenticated connection (PREAUTH greeting)
(3) rejected connection (BYE greeting)
(4) successful LOGIN or AUTHENTICATE command
(5) successful SELECT or EXAMINE command
(6) CLOSE command, or failed SELECT or EXAMINE command
(7) LOGOUT command, server shutdown, or connection closed

Figure 9.7 IMAP State Transition Diagram (needs to be translated to a real postcript figure).

a graphical client program, or Web browser, like Netscape or Explorer. Figure 9.8 shows the Netscape browser in use, displaying a page of information from Princeton University.

Any Web browser has a function that allows the user to “open a URL.” URLs (uniform resource locators) provide information about the location of objects on the



An open testbed for developing, deploying, and accessing planetary-scale services.

Welcome to the PlanetLab website

PlanetLab is a global overlay network for developing and accessing new network services. Our goal is to grow to 1000 geographically distributed nodes, connected by a diverse collection of links. Toward this end, we are putting PlanetLab nodes into edge sites, co-location and routing centers, and homes (i.e., at the end of DSL lines and cable modems). PlanetLab is designed to support both short-term experiments and long-running services. Currently running services include network weather maps, network-embedded storage, peer-to-peer networks, and content distribution networks. Click here for an overview of PlanetLab, and its design goals.

The project is being seeded by Intel Research. Intel Research is providing hardware for the first set of nodes, and operational and development support for PlanetLab in the short to medium term.

Figure 9.8 The Netscape Web browser.

Web; they look like the following:

http://www.cs.princeton.edu/index.html

If you opened that particular URL, your Web browser would open a TCP connection to the Web server at a machine called www.cs.princeton.edu and immediately retrieve and display the file called index.html. Most files on the Web contain images and text, and some have audio and video clips. They also include URLs that point to other files, and your Web browser will have some way in which you can recognize URLs and ask the browser to open them. These embedded URLs are called *hypertext links*. When you ask your Web browser to open one of these embedded URLs (e.g., by pointing and clicking on it with a mouse), it will open a new connection and retrieve and display a new file. This is called "following a link." It thus becomes very easy to hop from one machine to another around the network, following links to all sorts of information.

When you select to view a page, your browser (the client) fetches the page from the server using HTTP running over TCP. Like SMTP, HTTP is a text-oriented protocol. At its core, each HTTP message has the general form

```
START_LINE <CRLF>
MESSAGE_HEADER <CRLF>
<CRLF>
MESSAGE_BODY <CRLF>
```

where as before, <CRLF> stands for carriage-return-line-feed. The first line (START_LINE) indicates whether this is a request message or a response message. In effect, it identifies the "remote procedure" to be executed (in the case of a request message) or the "status" of the request (in the case of a response message). The next set of lines specify a collection of options and parameters that qualify the request or response. There are zero or more of these MESSAGE_HEADER lines—the set is terminated by a blank line—each of which looks like a header line in an email message. HTTP defines many possible header types, some of which pertain to request messages, some to response messages, and some to the data carried in the message body. Instead of giving the full set of possible header types, though, we just give a handful of representative examples. Finally, after the blank line comes the contents of the requested message (MESSAGE_BODY); this part of the message is typically empty for request messages.

Request Messages

The first line of an HTTP request message specifies three things: the operation to be performed, the Web page the operation should be performed on, and the version of HTTP being used. Although HTTP defines a wide assortment of possible request operations—including "write" operations that allow a Web page to be posted on a server—the two most common operations are GET (fetch the specified Web page) and HEAD (fetch status information about the specified Web page). The former is obviously used when your browser wants to retrieve and display a Web page. The latter is used to test the validity of a hypertext link or to see if a particular page has been

Operation	Description
OPTIONS	request information about available options
GET	retrieve document identified in URL
HEAD	retrieve metainformation about document identified in URL
POST	give information (e.g., annotation) to server
PUT	store document under specified URL
DELETE	delete specified URL
TRACE	loopback request message
CONNECT	for use by proxies

Table 9.1 HTTP request operations.

modified since the browser last fetched it. The full set of operations is summarized in Table 9.1.

For example, the START_LINE

```
GET http://www.cs.princeton.edu/index.html HTTP/1.1
```

says that the client wants the server on host www.cs.princeton.edu to return the page named index.html. This particular example uses an *absolute* URL. It is also possible to use a *relative* identifier and specify the host name in one of the MESSAGE_HEADER lines; for example,

```
GET index.html HTTP/1.1
Host: www.cs.princeton.edu
```

Here, Host is one of the possible MESSAGE_HEADER fields. One of the more interesting of these is If-Modified-Since, which gives the client a way to conditionally request a Web page—the server returns the page only if it has been modified since the time specified in that header line.

Response Messages

Like request messages, response messages begin with a single START_LINE. In this case, the line specifies the version of HTTP being used, a three-digit code indicating whether or not the request was successful, and a text string giving the reason for the response. For example, the START_LINE

```
HTTP/1.1 202 Accepted
```

Code	Type	Example Reasons
1xx	Informational	request received, continuing process
2xx	Success	action successfully received, understood, and accepted
3xx	Redirection	further action must be taken to complete the request
4xx	Client Error	request contains bad syntax or cannot be fulfilled
5xx	Server Error	server failed to fulfill an apparently valid request

Table 9.2 Five types of HTTP result codes.

indicates that the server was able to satisfy the request, while

```
HTTP/1.1 404 Not Found
```

indicates that it was not able to satisfy the request because the page was not found. There are five general types of response codes, with the first digit of the code indicating its type. Table 9.2 summarizes the five types of codes.

Also similar to request messages, response messages can contain one or more MESSAGE_HEADER lines. These lines relay additional information back to the client. For example, the Location header line specifies that the requested URL is available at another location. Thus, if the Princeton CS Department Web page had moved from http://www.cs.princeton.edu/index.html to http://www.princeton.edu/cs/index.html, for example, then the server at the original address might respond with

```
HTTP/1.1 301 Moved Permanently
Location: http://www.princeton.edu/cs/index.html
```

In the common case, the response message will also carry the requested page. This page is an HTML document, but since it may carry nontextual data (e.g., a GIF image), it is encoded using MIME (see Section 9.2.1). Certain of the MESSAGE_HEADER lines give attributes of the page contents, including Content-Length (number of bytes in the contents), Expires (time at which the contents are considered stale), and Last-Modified (time at which the contents were last modified at the server).

TCP Connections

The original version of HTTP (1.0) established a separate TCP connection for each data item retrieved from the server. It's not too hard to see how this was a very inefficient mechanism: connection setup and teardown messages had to be exchanged between the client and server even if all the client wanted to do was verify that it had the most recent copy of a page. Thus, retrieving a page that included some text and a dozen icons or other small graphics would result in 13 separate TCP connections being established and closed.

The most important improvement in the latest version of HTTP (1.1) is to allow *persistent connections*—the client and server can exchange multiple request/response messages over the same TCP connection. Persistent connections have two advantages. First, they obviously eliminate the connection setup overhead, thereby reducing the load on the server, the load on the network caused by the additional TCP packets, and the delay perceived by the user. Second, because a client can send multiple request messages down a single TCP connection, TCP's congestion window mechanism is able to operate more efficiently. This is because it's not necessary to go through the slow start phase for each page.

Persistent connections do not come without a price, however. The problem is that neither the client nor server necessarily knows how long to keep a particular TCP connection open. This is especially critical on the server, which might be asked to keep connections opened on behalf of thousands of clients. The solution is that the server must time out and close a connection if it has received no requests on the connection for a period of time. Also, both the client and server must watch to see if the other side has elected to close the connection, and they must use that information as a signal that they should close their side of the connection as well. (Recall that both sides must close a TCP connection before it is fully terminated.)

Caching

One of the most active areas of research (and entrepreneurship) in the Internet today is how to effectively cache Web pages. Caching has many benefits. From the client's perspective, a page that can be retrieved from a nearby cache can be displayed much more quickly than if it has to be fetched from across the world. From the server's perspective, having a cache intercept and satisfy a request reduces the load on the server.

Caching can be implemented in many different places. For example, a user's browser can cache recently accessed pages, and simply display the cached copy if the user visits the same page again. As another example, a site can support a single sitewide cache. This allows users to take advantage of pages previously downloaded by other users. Closer to the middle of the Internet, ISPs can cache pages. Note that in the second case, the users within the site most likely know what machine is caching pages on behalf of the site, and they configure their browsers to connect directly to the caching host. This node is sometimes called a *proxy*. In contrast, the sites that connect to the ISP are probably not aware that the ISP is caching pages. It simply happens to be the case that HTTP requests coming out of the various sites pass through a common ISP router. This router can peek inside the request message and look at the URL for the requested page. If it has the page in its cache, it returns it. If not, it forwards the request to the server and watches for the response to fly by in the other direction. When it does, the router saves a copy in the hope that it can use it to satisfy a future request.

No matter where pages are cached, the ability to cache Web pages is important enough that HTTP has been designed to make the job easier. The trick is that the cache needs to make sure it is not responding with an out-of-date version of the page. For example, the server assigns an expiration date (the Expires header field) to each page it sends back to the client (or to a cache between the server and client). The cache remembers this date and knows that it need not reverify the page each time it is requested until after that expiration date has passed. After that time (or if that header field is not set) the cache can use the HEAD or conditional GET operation (GET with If-Modified-Since header line) to verify that it has the most recent copy of the page. More generally, there are a set of "cache directives" that must be obeyed by all caching mechanisms along the request/response chain. These directives specify whether or not a document can be cached, how long it can be cached, how fresh a document must be, and so on.

9.2.3 Network Management (SNMP)

A network is a complex system, both in terms of the number of nodes that are involved and in terms of the suite of protocols that can be running on any one node. Even if you restrict yourself to worrying about the nodes within a single administrative domain, such as a campus, there might be dozens of routers and hundreds—or even thousands—of hosts to keep track of. If you think about all the state that is maintained and manipulated on any one of those nodes—for example, address translation tables, routing tables, TCP connection state, and so on—then it is easy to become depressed about the prospect of having to manage all of this information.

It is easy to imagine wanting to know about the state of various protocols on different nodes. For example, you might want to monitor the number of IP datagram reassemblies that have been aborted, so as to determine if the timeout that garbage collects partially assembled datagrams needs to be adjusted. As another example, you might want to keep track of the load on various nodes (i.e., the number of packets sent or received) so as to determine if new routers or links need to be added to the network. Of course, you also have to be on the watch for evidence of faulty hardware and misbehaving software.

What we have just described is the problem of network management, an issue that pervades the entire network architecture. Since the nodes we want to keep track of are distributed, our only real option is to use the network to manage the network.

This means we need a protocol that allows us to read, and possibly write, various pieces of state information on different network nodes. The most widely used protocol for this purpose is the Simple Network Management Protocol (SNMP).

SNMP is essentially a specialized request/reply protocol that supports two kinds of request messages: GET and SET. The former is used to retrieve a piece of state from some node, and the latter is used to store a new piece of state in some node. (SNMP also supports a third operation—GET-NEXT—which we explain below.) The following discussion focuses on the GET operation, since it is the one most frequently used.

SNMP is used in the obvious way. A system administrator interacts with a client program that displays information about the network. This client program usually has a graphical interface. You can think of this interface as playing the same role as a Web browser. Whenever the administrator selects a certain piece of information that he or she wants to see, the client program uses SNMP to request that information from the node in question. (SNMP runs on top of UDP.) An SNMP server running on that node receives the request, locates the appropriate piece of information, and returns it to the client program, which then displays it to the user.

There is only one complication to this otherwise simple scenario: Exactly how does the client indicate which piece of information it wants to retrieve, and likewise, how does the server know which variable in memory to read to satisfy the request? The answer is that SNMP depends on a companion specification called the management information base (MIB). The MIB defines the specific pieces of information—the MIB *variables*—that you can retrieve from a network node.

The current version of MIB, called MIB-II, organizes variables into 10 different *groups*. You will recognize that most of the groups correspond to one of the protocols described in this book, and nearly all of the variables defined for each group should look familiar. For example:

- System: general parameters of the system (node) as a whole, including where the node is located, how long it has been up, and the system's name.
- Interfaces: information about all the network interfaces (adaptors) attached to this node, such as the physical address of each interface, how many packets have been sent and received on each interface.
- Address translation: information about the Address Resolution Protocol (ARP), and in particular, the contents of its address translation table.
- IP: variables related to IP, including its routing table, how many datagrams it has successfully forwarded, and statistics about datagram reassembly. Includes counts of how many times IP drops a datagram for one reason or another.

- TCP: information about TCP connections, such as the number of passive and active opens, the number of resets, the number of timeouts, default timeout settings, and so on. Per-connection information persists only as long as the connection exists.
- UDP: information about UDP traffic, including the total number of UDP datagrams that have been sent and received.

There are also groups for ICMP, EGP, and SNMP itself. The 10th group is used by different media.

Returning to the issue of the client stating exactly what information it wants to retrieve from a node, having a list of MIB variables is only half the battle. Two problems remain. First, we need a precise syntax for the client to use to state which of the MIB variables it wants to fetch. Second, we need a precise representation for the values returned by the server. Both problems are addressed using ASN.1.

Consider the second problem first. As we already saw in Chapter 7, ASN.1/BER defines a representation for different data types, such as integers. The MIB defines the type of each variable, and then it uses ASN.1/BER to encode the value contained in this variable as it is transmitted over the network. As far as the first problem is concerned, ASN.1 also defines an object identification scheme; this identification system is not described in Chapter 7. The MIB uses this identification system to assign a globally unique identifier to each MIB variable. These identifiers are given in a "dot" notation, not unlike domain names. For example, 1.3.6.1.2.1.4.3 is the unique ASN.1 identifier for the IP-related MIB variable ipInReceives; this variable counts the number of IP datagrams that have been received by this node. In this example, the 1.3.6.1.2.1 prefix identifies the MIB database (remember, ASN.1 object IDs are for all possible objects in the world), the 4 corresponds to the IP group, and the final 3 denotes the third variable in this group.

Thus, network management works as follows. The SNMP client puts the ASN.1 identifier for the MIB variable it wants to get into the request message, and it sends this message to the server. The server then maps this identifier into a local variable (i.e., into a memory location where the value for this variable is stored), retrieves the current value held in this variable, and uses ASN.1/BER to encode the value it sends back to the client.

There is one final detail. Many of the MIB variables are either tables or structures. Such compound variables explain the reason for the SNMP GET-NEXT operation. This operation, when applied to a particular variable ID, returns the value of that variable plus the ID of the next variable, for example, the next item in the table or the next field in the structure. This aids the client in "walking through" the elements of a table or structure.

9.3 Multimedia Applications

Just like the traditional applications of the previous section, multimedia applications such as audio- and videoconferencing applications need application-layer protocols. Much of the initial experience in designing protocols for multimedia applications came from the "MBone tools"—applications such as vat and vic that were developed for use on the MBone, using IP multicast to enable multiparty conferencing. Initially, each application implemented its own protocol (or protocols), but it became apparent that many multimedia applications have common requirements. This ultimately led to the development of a number of general-purpose protocols for use by multimedia applications.

We have already seen one protocol that is of general use to multimedia applications in the form of RSVP (see Section 6.5.2.) That protocol can be used to request the allocation of resources in the network so that the desired quality of service (QoS) can be provided to an application. In addition to a QoS signalling protocol, many multimedia applications also need some sort of transport protocol, with rather different characteristics than TCP and with more functionality than UDP. The protocol that has been developed to meet those needs is called the Real-time Transport Protocol (RTP), described below.

A third class of protocol that many multimedia applications need is a *session control* protocol. For example, suppose that we wanted to be able to make IP-based telephone calls across the Internet. We would need some mechanism to notify the intended recipient of such a call that we wanted to talk to her, for example, by sending a message to some multimedia device that would cause it to make a ringing sound. We would also like to be able to support features like call forwarding, three-way calling, and so on. SIP (Session Initiation Protocol) and H.323 are examples of protocols that address the issues of session control; we discuss them in Section 9.3.2.

9.3.1 Real-time Transport Protocol (RTP)

You might wonder why a protocol whose name identifies it as a "transport protocol" appears in a chapter on application-layer issues. The reason for this is that RTP contains a considerable amount of functionality that is specific to multimedia applications. Furthermore, it typically runs on top of one of the transport-layer protocols described in Chapter 5—UDP—which provides some of the application-independent functions you usually associate with a transport protocol. RTP is nevertheless called a transport protocol because it provides common end-to-end functions to a number of applications. (Most application-layer protocols, like HTTP and SMTP, for example, are specific to a single application.) A point to note here is the difficulty of fitting real-world protocols into a strict layerist model.

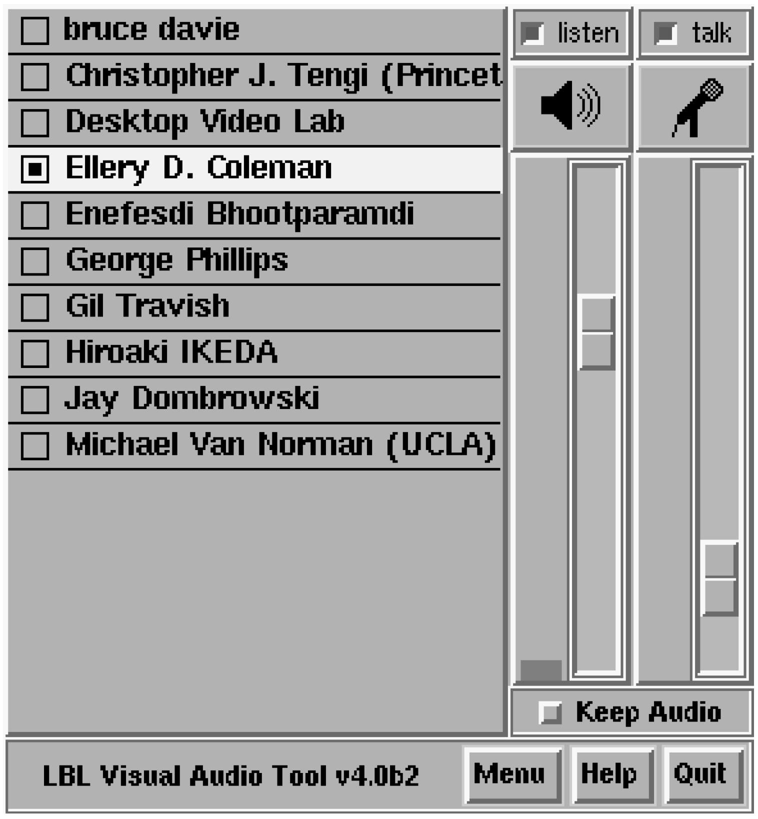


Figure 9.9 User interface of a vat audioconference.

Before we look at RTP in detail, it will help to consider some of the applications that might use it. Multimedia applications are sometimes divided into two classes—*conferencing* applications and *streaming* applications. A popular example of the former class is vat, the audioconferencing tool that is often used over networks supporting IP multicast. The control panel for a typical vat conference is shown in Figure 9.9. Another conferencing application is vic, the videoconferencing tool discussed in Chapter 1 and illustrated in Figure 1.1.

Streaming applications typically deliver audio or video streams from a server to a client, and are typified by such commercial products as Real Audio. Because of the lack of human interaction, such applications place somewhat different requirements on the underlying protocols. It should by now be apparent that designers of a transport

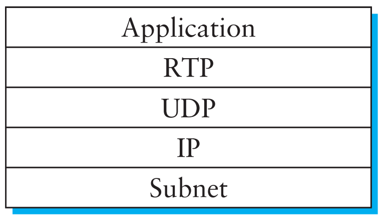


Figure 9.10 Protocol stack for multimedia applications using RTP.

protocol for multimedia applications face a real challenge in defining the requirements broadly enough to meet the needs of very different applications. They must also pay attention to the interactions among different applications, for example, the synchronization of audio and video streams. We will see how these concerns affected the design of RTP below.

Much of RTP actually derives from the application protocol that was originally embedded in vat. Newer versions of vat (and many other applications) run over RTP. RTP can run over many lower-layer protocols, but commonly runs over UDP. That leads to the protocol stack shown in Figure 9.10.

Requirements

The most basic requirement for a general-purpose multimedia protocol is that it allow similar applications to interoperate with each other. For example, it should be possible for two independently implemented audioconferencing applications to talk to each other. This immediately suggests that the applications had better use the same method of encoding and compressing voice; otherwise, the data sent by one party will be incomprehensible to the receiving party. Since there are quite a few different coding schemes for voice, each with its own trade-offs between quality, bandwidth requirements, and computational cost, it would probably be a bad idea to decree that only one such scheme can be used. Instead, our protocol should provide a way that a sender can tell a receiver which coding scheme it wants to use, and possibly negotiate until a scheme that is available to both parties is identified.

Just as with audio, there are many different video coding schemes. Thus, we see that the first common function that RTP can provide is the ability to communicate that choice of coding scheme. Note that this also serves to identify the type of application (e.g., audio or video); once we know what coding algorithm is being used, we know what type of data is being encoded as well.

Another important requirement for RTP is to enable the recipient of a data stream to determine the timing relationship among the received data. Recall from Section 6.5

that real-time applications need to place received data into a *playback buffer* to smooth out the jitter that may have been introduced into the data stream during transmission across the network. Thus, some sort of timestamping of the data will be necessary to enable the receiver to play it back at the appropriate time.

Related to the timing of a single media stream is the issue of synchronization of multiple media in a conference. The obvious example of this would be to synchronize an audio and video stream that are originating from the same sender. As we will see below, this is a slightly more complex problem than playback time determination for a single stream.

Another important function to be provided is an indication of packet loss. Note that an application with tight latency bounds generally cannot use a reliable transport like TCP because retransmission of data to correct for loss would probably cause the packet to arrive too late to be useful. Thus, the application must be able to deal with missing packets, and the first step in dealing with them is noticing that they are in fact missing. As an example, a video application using MPEG encoding will need to take different actions when a packet is lost, depending on whether the packet came from an I frame, a B frame, or a P frame.

Since multimedia applications generally do not run over TCP, they also miss out on the congestion-avoidance features of TCP (as described in Section 6.3). Yet many multimedia applications are capable of responding to congestion, for example, by changing the parameters of the coding algorithm to reduce the bandwidth consumed. Clearly, to make this work, the receiver needs to notify the sender that losses are occurring so that the sender can adjust its coding parameters.

Another common function across multimedia applications is the concept of frame boundary indication. A frame in this context is application-specific. For example, it may be helpful to notify a video application that a certain set of packets correspond to a single frame. In an audio application it is helpful to mark the beginning of a "talkspurt," which is a collection of sounds or words followed by silence. The receiver can then identify the silences between talkspurts and use them as opportunities to move the playback point. This follows the observation that slight shortening or lengthening of the spaces between words are not perceptible to users, whereas shortening or lengthening the words themselves is both perceptible and annoying.

A final function that we might want to put into the protocol is some way of identifying senders that is more user-friendly than an IP address. Tools such as vat and vic can display strings such as Joe User (user@domain.com) on their control panels, and thus the application protocol should support the association of such a string with a data stream.

In addition to the functionality that is required from our protocol, we note an additional requirement: It should make reasonably efficient use of bandwidth.

Put another way, we don't want to introduce a lot of extra bits that need to be sent with every packet in the form of a long header. The reason for this is that audio packets, which are one of the most common types of multimedia data, tend to be small, so as to reduce the time it takes to fill them with samples. Long audio packets would mean high latency due to packetization, which has a negative effect on the perceived quality of conversations. (Recall that this was one of the factors in choosing the length of ATM cells.) Since the data packets themselves are short, a large header would mean that a relatively large amount of link bandwidth would be used by headers, thus reducing the available capacity for "useful" data. We will see several aspects of the design of RTP that have been influenced by the necessity of keeping the header short.

RTP Details

Now that we have seen the rather long list of requirements for our application-layer protocol for multimedia, we turn to the details of the protocol that has been specified to meet those requirements. This protocol, RTP, was developed in the IETF and is in widespread use. The RTP standard actually defines a pair of protocols, RTP and the Real-time Transport Control Protocol (RTCP). The former is used for the exchange of multimedia data, while the latter is used to periodically send control information associated with a certain data flow. When running over UDP, the RTP data stream and the associated RTCP control stream use consecutive transport-layer ports. The RTP data uses an even port number and the RTCP control information uses the next higher (odd) port number.

Because RTP is designed to support a wide variety of applications, it provides a flexible mechanism by which new applications can be developed without repeatedly revising the RTP protocol itself. For each class of application (e.g., audio), RTP defines a *profile* and one or more *formats*. The profile provides a range of information that ensures a common understanding of the fields in the RTP header for that application class, as will be apparent when we examine the header in detail. The format specification explains how the data that follows the RTP header is to be interpreted. For example, the RTP header might just be followed by a sequence of bytes, each of which represents a single audio sample taken a defined interval after the previous one. Alternatively, the format of the data might be much more complex; an MPEG-encoded video stream, for example, would need to have a good deal of structure to represent all the different types of information.

The design of RTP embodies an architectural principle known as *Application Level Framing* (ALF). This principle was put forward by Clark and Tennenhouse in 1990 as a new way to design protocols for emerging multimedia applications. They recognized that these new applications were unlikely to be well served by existing protocols such as TCP, and that furthermore they might not be well served by any

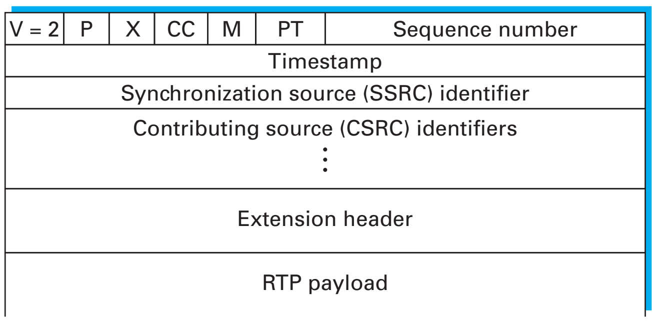

V = 2
P
X
CC
M
PT
Sequence number
Timestamp
Synchronization source (SSRC) identifier
Contributing source (CSRC) identifiers
Extension header
RTP payload

Figure 9.11 RTP header format.

sort of "one-size-fits-all" protocol. At the heart of this principle is the belief that an application understands its own needs best. For example, an MPEG video application knows how best to recover from lost frames, and how to react differently if an I frame or a B frame is lost. The same application also understands best how to segment the data for transmission—for example, it's better to send the data from different frames in different datagrams, so that a lost packet only corrupts a single frame, not two. It is for this reason that RTP leaves so many of the protocol details to the profile and format documents that are specific to an application.

Header Format

Figure 9.11 shows the header format used by RTP. The first 12 bytes are always present, whereas the contributing source identifiers are only used in certain circumstances. After this header there may be optional header extensions, as described below. Finally, the header is followed by the RTP payload, the format of which is determined by the application. The intention of this header is that it contain only the fields that are likely to be used by many different applications, since anything that is very specific to a single application would be more efficiently carried in the RTP payload for that application only.

The first two bits are a version identifier, which contains the value 2 in the RTP version deployed at the time of writing. You might think that the designers of the protocol were rather bold to think that 2 bits would be enough to contain all future versions of RTP, but recall that bits are at a premium in the RTP header. Furthermore, the use of profiles for different applications makes it less likely that many revisions to the base RTP protocol would be needed. In any case, if it turns out that another version of RTP is needed beyond version 2, it would be possible to consider a change to the header format so that more than one future version would be possible. For example,

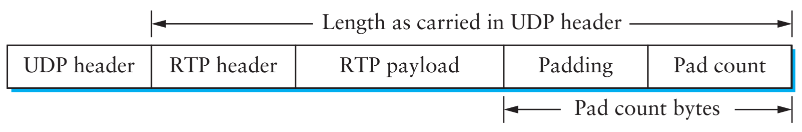


Figure 9.12 Padding of an RTP packet.

a new RTP header with the value 3 in the version field could have a "subversion" field somewhere else in the header.

The next bit is the "padding" (P) bit, which is set in circumstances in which the RTP payload has been padded for some reason. RTP data might be padded to fill up a block of a certain size as required by an encryption algorithm, for example. In such a case, the complete length of the RTP header, data, and padding would be conveyed by the lower-layer protocol header (e.g., the UDP header), and the last byte of the padding would contain a count of how many bytes should be ignored. This is illustrated in Figure 9.12. Note that this approach to padding removes any need for a length field in the RTP header (thus serving the goal of keeping the header short); in the common case of no padding, the length is deduced from the lower-layer protocol.

The extension (X) bit is used to indicate the presence of an extension header, which would be defined for a specific application and follow the main header. Such headers are rarely used, since it is generally possible to define a payload-specific header as part of the payload format definition for a particular application.

The X bit is followed by a 4-bit field that counts the number of "contributing sources," if any are included in the header. Contributing sources are discussed below.

We noted above the frequent need for some sort of frame indication; this is provided by the marker bit, which could be set at the beginning of a talkspurt, for example. The 7-bit payload type field follows; it indicates what type of multimedia data is carried in this packet. One possible use of this field would be to enable an application to switch from one coding scheme to another based on information about resource availability in the network or feedback on application quality. The exact usage of the marker bit and the payload type is determined by the application profile.

Note that the payload type is generally not used as a demultiplexing key to direct data to different applications (or to different streams within a single application, for example, the audio and video stream for a videoconference). This is because such demultiplexing is typically provided at a lower layer (e.g., by UDP, as described in Section 5.1). Thus, two media streams using RTP would typically use different UDP port numbers.

The sequence number is used to enable the receiver of an RTP stream to detect missing and misordered packets. The sender simply increments the value by one for each transmitted packet. Note that RTP does not do anything when it detects a lost packet, in contrast to TCP, which both corrects for the loss (by retransmission) and interprets the loss as a congestion indication (which may cause it to reduce its window size). Rather, it is left to the application to decide what to do when a packet is lost because this decision is likely to be highly application-dependent. For example, a video application might decide that the best thing to do when a packet is lost is to replay the last frame that was correctly received. Some applications might also decide to modify their coding algorithms to reduce bandwidth needs in response to loss, but this is not a function of RTP. It would not be sensible for RTP to decide that the sending rate should be reduced, as this might make the application useless.

The function of the timestamp field is to enable the receiver to play back samples at the appropriate intervals and to enable different media streams to be synchronized. Because different applications may require different granularities of timing, RTP itself does not specify the units in which time is measured. Instead, the timestamp is just a counter of "ticks," where the time between ticks is dependent on the encoding in use. For example, an audio application that samples data once every 125 μs could use that value as its clock resolution. The clock granularity is one of the details that is specified in the RTP profile or payload format for an application.

The timestamp value in the packet is a number representing the time at which the *first* sample in the packet was generated. The timestamp is not a reflection of the time of day; only the differences between timestamps are relevant. For example, if the sampling interval is 125 μs and the first sample in packet $n+1$ was generated 10 ms after the first sample in packet n, then the number of sampling instants between these two samples is

$$\begin{aligned}\textsf{TimeBetweenPackets} \div \textsf{TimePerSample} &= (10 \times 10^{-3}) \div (125 \times 10^{-6}) \\ &= 80\end{aligned}$$

Assuming the clock granularity is the same as the sampling interval, then the timestamp in packet $n+1$ would be greater than that in packet n by 80. Note that fewer than 80 samples might have been sent due to compression techniques such as silence detection, and yet the timestamp allows the receiver to play back the samples with the correct temporal relationship.

The synchronization source (SSRC) is a 32-bit number that uniquely identifies a single source of an RTP stream. In a given multimedia conference, each sender picks a random SSRC and is expected to resolve conflicts in the unlikely event that two sources pick the same value. By making the source identifier something other than the network or transport address of the source, RTP ensures independence from the lower-layer protocol. It also enables a single node with multiple sources (e.g., several cameras) to distinguish those sources. When a single node generates different media streams (e.g., audio and video), it is not required to use the same SSRC in each stream, as there are mechanisms in RTCP (described below) to allow intermedia synchronization.

The contributing source (CSRC) is used only when a number of RTP streams pass through a "mixer." A mixer can be used to reduce the bandwidth requirements for a conference by receiving data from many sources and sending it as a single stream. For example, the audio streams from several concurrent speakers could be decoded and recoded as a single audio stream. In this case, the mixer lists itself as the synchronization source but also lists the contributing sources—the SSRC values of the speakers who contributed to the packet in question.

Control Protocol

RTCP provides a control stream that is associated with a data stream for a multimedia application. This control stream provides three main functions:

1. Feedback on the performance of the application and the network
2. A way to correlate and synchronize different media streams that have come from the same sender
3. A way to convey the identity of a sender for display on a user interface (e.g., the `vat` interface shown in Figure 9.9)

The first function may be useful for rate-adaptive applications, which may use performance data to decide to use a more aggressive compression scheme to reduce congestion, or to send a higher-quality stream when there is little congestion. It can also be useful in diagnosing network problems.

You might think that the second function is already provided by the synchronization source ID of RTP, but in fact it is not. As already noted, multiple cameras from a single node might have different SSRC values. Furthermore, there is no requirement that an audio and video stream from the same node use the same SSRC. Because collisions of SSRC values may occur, it may be necessary to change the SSRC value of a stream. To deal with this problem, RTCP uses the concept of a "canonical name" (CNAME) that is assigned to a sender, which is then associated with the various SSRC values that might be used by that sender using RTCP mechanisms.

Simply correlating two streams is only part of the problem of intermedia synchronization. Because different streams may have completely different clocks (with different granularities and even different amounts of inaccuracy, or drift), there needs to be a way to accurately synchronize streams with each other. RTCP addresses this problem.

RTCP defines a number of different packet types, including

- sender reports, which enable active senders to a session to report transmission and reception statistics
- receiver reports, which receivers who are not senders use to report reception statistics
- source descriptions, which carry CNAMEs and other sender description information
- application-specific control packets

These different RTCP packet types are sent over the lower-layer protocol, which, as we have noted, is typically UDP. Several RTCP packets can be packed into a single PDU of the lower-level protocol. It is required that at least two RTCP packets are sent in every lower-level PDU: One of these is a report packet; the other is a source description packet. Other packets may be included up to the size limits imposed by the lower-layer protocols.

Before looking closely at the contents of an RTCP packet, we note that there is a potential problem with every member of a multicast group sending periodic control traffic. Unless we take some steps to limit it, this control traffic has the potential to be a significant consumer of bandwidth. For example, in an audioconference, no more than two or three senders are likely to send audio data at any instant, since there is no point in everyone talking at once. But there is no such social limit on everyone sending control traffic, and this could be a severe problem in a conference with thousands of participants. To deal with this problem, RTCP has a set of mechanisms by which the participants scale back their reporting frequency as the number of participants increases. These rules are somewhat complex, but the basic goal is this: Limit the total amount of RTCP traffic to a small percentage (typically 5%) of the RTP data traffic. To accomplish this goal, the participants should know how much data bandwidth is likely to be in use (e.g., the amount to send three audio streams) and the number of participants. They learn the former from means outside RTP (known as session management, discussed at the end of this section), and they learn the latter from the RTCP reports of other participants. Because RTCP reports might be sent at a very low rate, it might only be possible to get an approximate count of the current number of recipients, but that is typically sufficient. Also, it is recommended to allocate more RTCP bandwidth to active senders, on the assumption that most participants would like to see reports from them, for example, to find out who is speaking.

Once a participant has determined how much bandwidth it can consume with RTCP traffic, it sets about sending periodic reports at the appropriate rate.

Sender reports and receiver reports differ only in that the former include some extra information about the sender. Both types of reports contain information about the data that was received from all sources in the most recent reporting period.

The extra information in a sender report consists of

- a timestamp containing the actual time of day when this report was generated
- the RTP timestamp corresponding to the time when the report was generated
- cumulative counts of the packets and bytes sent by this sender since it began transmission

Note that the first two quantities can be used to enable synchronization of different media streams from the same source, even if those streams use different clock granularities in their RTP data streams, since it gives the key to convert time of day to the RTP timestamps.

Both sender and receiver reports contain one block of data per source that has been heard from since the last report. Each block contains the following statistics for the source in question:

- Its SSRC
- The fraction of data packets from this source that were lost since the last report was sent (calculated by comparing the number of packets received with the number of packets expected; this last value can be determined from the RTP sequence numbers)
- Total number of packets lost from this source since the first time it was heard from
- Highest sequence number received from this source (extended to 32 bits to account for wrapping of the sequence number)
- Estimated interarrival jitter for the source (calculated by comparing the interarrival spacing of received packets with the expected spacing at transmission time)
- Last actual timestamp received via RTCP for this source
- Delay since last sender report received via RTCP for this source

As you might imagine, the recipients of this information can learn all sorts of things about the state of the session. In particular, they can see if other recipients are getting much better quality from some sender than they are, which might be an indication that a resource reservation needs to be made, or that there is a problem in the network that needs to be attended to. In addition, if a sender notices that many receivers are experiencing high loss of its packets, it might decide that it should reduce its sending rate or use a coding scheme that is more resilient to loss.

The final aspect of RTCP that we will consider is the source description packet. Such a packet contains, at a minimum, the SSRC of the sender and the sender's CNAME. The canonical name is derived in such a way that all applications that generate media streams that might need to be synchronized (e.g., separately generated audio and video streams from the same user) will choose the same CNAME even though they might choose different SSRC values. This enables a receiver to identify the media stream that came from the same sender. The most common format of the CNAME is user@host, where host is the fully qualified domain name of the sending machine. Thus, an application launched by the user whose user name is jdoe running on the machine cicada.cs.princeton.edu would use the string jdoe@cicada.cs.princeton.edu as its CNAME. The large and variable number of bytes used in this representation would make it a bad choice for the format of an SSRC, since the SSRC is sent with every data packet and must be processed in real time. Allowing CNAMEs to be bound to SSRC values in periodic RTCP messages enables a compact and efficient format for the SSRC.

Other items may be included in the source description packet, such as the real name and email address of the user. These are used in user interface displays and to contact participants, but are less essential to the operation of RTP than the CNAME.

9.3.2 Session Control and Call Control (SDP, SIP, H.323)

To understand some of the issues of session control, consider the following problem. Suppose you want to hold a videoconference at a certain time and make it available to a wide number of participants. Perhaps you have decided to encode the video stream using the MPEG-2 standard, to use the multicast IP address 224.1.1.1 for transmission of the data, and to send it using RTP over UDP port number 4000. How would you make all that information available to the intended participants? One way would be to put all that information in an email and send it out, but ideally there should be a standard format and protocol for disseminating this sort of information. The IETF has a working group (the Multiparty Multimedia Session Control group) that has defined protocols for just this purpose. The protocols that have been defined include

- SDP (Session Description Protocol)
- SAP (Session Announcement Protocol)
- SIP (Session Initiation Protocol)
- SCCP (Simple Conference Control Protocol)

You might think that this is a lot of protocols for a seemingly simple task, but there are many aspects of the problem and several different situations in which it must be addressed. For example, there is a difference between announcing the fact that a certain conference session is going to be made available on the MBone (which would be done using SDP and SAP) and trying to make an internet phone call to a certain user at a particular time (which could be done using SDP and SIP). In the former case, you could consider your job done once you have sent all the session information in a standard format to a well-known multicast address. In the latter, you would need to locate one or more users, get a message to them announcing your desire to talk (analogous to ringing their phone), and perhaps negotiate a suitable audio encoding among all parties. We will look first at SDP, which is common to many applications, then at SIP, which is becoming widely used for a number of interactive applications such as internet telephony.

Session Description Protocol (SDP)

SDP is a rather general protocol that can be used in a variety of situations. It conveys the following information:

- The name and purpose of the session
- Start and end times for the session
- The media types (e.g., audio, video) that comprise the session
- Detailed information needed to receive the session (e.g., the multicast address to which data will be sent, the transport protocol to be used, the port numbers, the encoding schemes)

SDP provides this information formatted in ASCII using a sequence of lines of text, each of the form "<type>=<value>". An example of an SDP message will illustrate the main points.

```
v=0
o=larry 2890844526 2890842807 IN IP4 10.0.1.5
s=Networking 101
i=A class on computer networking
u=http://www.cs.princeton.edu/
e=larry@cs.princeton.edu
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
m=application 32416 udp wb
```

Note that SDP, like HTML, is fairly easy for a human to read, but has strict formatting rules that make it possible for machines to interpret the data unambiguously. For example, the SDP specification defines all the possible information "types" that are allowed to appear, the order in which they must appear, and the format and reserved words for every type that is defined.

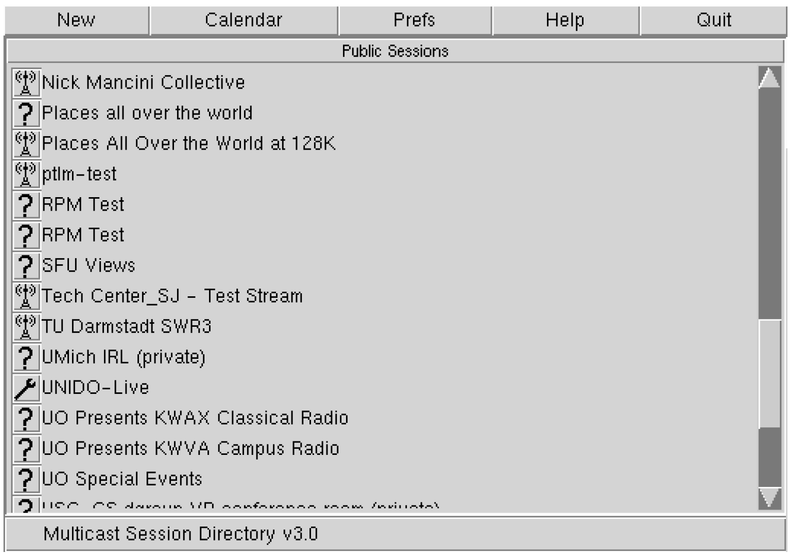
The first thing to notice is that each information "type" is identified by a single character. For example, the line v=0 tells us that "version" has the value zero; that is, this message is formatted according to version zero of SDP. The next line provides the "origin" of the session, which contains enough information to uniquely identify the session. larry is a username of the session creator, and 10.0.1.5 is the IP address of his computer. The number following larry is a session identifier that is chosen to be unique to that machine. This is followed by a "version" number for the SDP announcement; if the session information was updated by a later message, the version number would be increased.

The next three lines (s, i, and u) provide the session name, a session description, and a session *uniform resource identifier* (URI)—all provide information that would be helpful to a user in deciding whether to participate in this session. Such information could be displayed in the user interface of a "session directory" tool that shows current and upcoming events that have been advertised using SDP. The next line (e=...) contains an email address of a person to contact regarding the session. Figure 9.13 shows a screen shot of a session directory tool called sdr along with the descriptions of several sessions that had been announced at the time the picture was taken.

Next we get to the technical details that would enable an application program to participate in the session. The line beginning c=... provides the IP multicast address to which data for this session will be sent; a user would need to join this multicast group to receive the session. Next we see the start and end times for the session (encoded as integers according to the Network Time Protocol). Finally, we get to the information about the media for this session. This session has three media types available—audio, video, and a shared whiteboard application known as 'wb.' For each media type there is one line of information formatted as follows:

```
m=<media> <port> <transport> <format>
```

The media types are self-explanatory, and the port numbers in each case are UDP ports. When we look at the "transport" field, we can see that the wb application runs directly over UDP, while the audio and video are tranported using "RTP/AVP." This means that they run over RTP and use the *application profile* (as defined in Section 9.3.1) known as AVP. That application profile defines a number of different encoding schemes for audio and video; we can see in this case that the audio is using encoding 0 (which is an encoding using an 8-KHz sampling rate and 8 bits per



Nick Mancini Collective
Places all over the world
Places All Over the World at 128K
ptlm-test
RPM Test
RPM Test
SFU Views
Tech Center_SJ - Test Stream
TU Darmstadt SWR3
UMich IRL (private)
UNIDO-Live
UO Presents KWAX Classical Radio
UO Presents KWVA Campus Radio
UO Special Events

Multicast Session Directory v3.0

Figure 9.13 A session directory tool displays information extracted from SDP messages.

sample) and the video is using encoding 31, which represents the H.261 encoding scheme. These "magic numbers" for the encoding schemes are defined in the RFC that defines the AVP profile; it is also possible to describe nonstandard coding schemes in SDP.

Finally we see a description of the "wb" media type. All the encoding information for this data is specific to the wb application, and so it is sufficient just to provide the name of the application in the "format" field. This is analogous to putting application/wb in a MIME message.

Now that we know how to describe sessions, we can look at how they can be initiated. One way in which SDP is used is to announce multimedia conferences, by sending SDP messages to a well-known multicast address. The session directory tool shown in Figure 9.13 would function by joining that multicast group and displaying information that it gleans from received SDP messages.

SDP also plays an important role in conjunction with the Session Initiation Protocol (SIP). With the increased importance of "voice over IP" (VOIP, i.e., the support of telephony-like applications over IP networks), SIP has attracted a great deal of attention and now has its own working group at the IETF. While SIP can be used for many things other than IP telephony, that is certainly one of its driving applications.

SIP

SIP is an application-layer protocol that bears a certain resemblance to HTTP, being based on a similar request/response model. However, it is designed with rather different sorts of applications in mind, and thus provides quite different capabilities than HTTP. The capabilities provided by SIP can be grouped into five categories:

- User location: determining the correct device with which to communicate to reach a particular user
- User availability: determining if the user is willing or able to take part in a particular communication session
- User capabilities: determining such items as the choice of media and coding scheme to use
- Session setup: establishing session parameters such as port numbers to be used by the communicating parties
- Session management: a range of functions including transferring sessions (e.g., to implement "call forwarding") and modifying session parameters

Most of these functions are easy enough to understand, but the issue of location bears some further discussion. One important difference between SIP and, say, HTTP, is that SIP is primarily used for human-to-human communication. Thus, it is important to be able to locate individual *users*, not just machines. And unlike email, it's not good enough to just locate a server that the user will be checking on at some later date and dump the message there—we need to know where the user is right now if we want to be able to communicate with him in real time. This is further complicated by the fact that a user might choose to communicate using a range of different devices, for example, using his desktop PC when he's in the office and using a handheld device when traveling. Multiple devices might be active at the same time and might have widely different capabilities (e.g., an alphanumeric pager and a PC-based video "phone"). Ideally, it should be possible for other users to be able to locate and communicate with the appropriate device at any time. Furthermore, the user must be able to have control over when, where, and from whom he receives calls.

To enable a user to exercise the appropriate level of control over his calls, SIP introduces the notion of a proxy. A SIP proxy can be thought of as a point of contact for a user to which initial requests for communication with him are sent. Proxies also perform functions on behalf of callers. We can see how proxies work best through an example.

Consider the two users in Figure 9.14. The first thing to notice is that each user has a name in the format user@domain, very much like an email address. When user

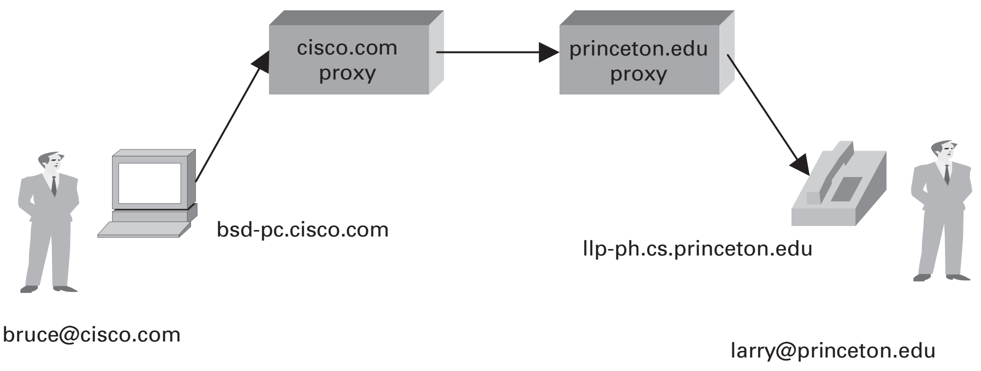


Figure 9.14 Establishing communication through SIP proxies.

Bruce wants to initiate a session with Larry, he sends his initial SIP message to the local proxy for his domain, cisco.com. Among other things, this initial message contains a *SIP URI*—these are a form of uniform resource identifier that look like this:

```
SIP:larry@princeton.edu
```

We saw an example of a different type of URI in Section 9.2.2. URLs (uniform resource locators) such as http://www.cs.princeton.edu are a particular type of URI that contains complete location information for a resource (e.g., a Web page). A SIP URI provides complete identification of a user, but does not provide his location, since that may change over time. We will see shortly how the location of a user can be determined.

Upon receiving the initial message from Bruce, the cisco.com proxy looks at the SIP URI and deduces that this message should be sent to the princeton.edu proxy. For now, we assume that the princeton.edu proxy has access to some database that enables it to obtain a mapping from the name larry@princeton.edu to the IP address of one or more devices at which Larry currently wishes to receive messages. The proxy can therefore forward the message on to Larry's chosen device(s). Sending the message to more than one device is called *forking* and may be done either in parallel or in series (e.g., send it to his cellphone if he doesn't answer the phone at his desk).

The initial message from Bruce to Larry is likely to be a SIP invite message, which looks something like the following:

```
INVITE sip:larry@princeton.edu SIP/2.0
Via: SIP/2.0/UDP bsd-pc.cisco.com;branch=z9hG4bK433yte4
To: Larry <sip:larry@princeton.edu>
```

```
From: Bruce <sip:bruce@cisco.com>;tag=55123
Call-ID: xy745jj210re3@bsd-pc.cisco.com
CSeq: 271828 INVITE
Contact: <sip:bruce@bsd-pc.cisco.com>
Content-Type: application/sdp
Content-Length: 142
```

The first line identifies the type of function to be performed (invite); the resource on which to perform it, that is, the called party (sip:larry@princeton.edu); and the protocol version (2.0). The subsequent header lines probably look somewhat familiar because of their resemblance to the header lines in an email message. SIP defines a large number of header fields, only some of which we describe here. Note that the Via: header in this example identifies the device from which this message originated. The Content-Type: and Content-Length: headers describe the contents of the message following the header, just as in a MIME-encoded email message. In this case, the content is an SDP (Session Description Protocol) message. That message would describe such things as the type of media (audio, video, etc.) that Bruce would like to exchange with Larry and other properties of the session such as codec types that he supports. Note that the Content-Type: field in SIP provides the capability to use any protocol for this purpose, although SDP is the most common.

Returning to the example, when the invite message arrives at the cisco.com proxy, the proxy not only forwards the message on toward princeton.edu, it also responds to the sender of the invite. Just as in HTTP, all responses have a response code, and the organization of codes is similar to that for HTTP, as shown in Table 9.2. In Figure 9.15 we can see a sequence of SIP messages and responses.

The first response message in this figure is the provisional response 100 trying, which indicates that the message was received without error by the caller's proxy. Once the invite is delivered to Larry's phone, it alerts Larry and responds with a 180 ringing message. The arrival of this message at Bruce's computer is a sign that it can generate a "ringtone." Assuming Larry is willing and able to communicate with Bruce, he could pick up his phone, causing the message 200 OK to be sent. Bruce's computer responds with an ACK, and at this point media (e.g., an RTP-encapsulated audio stream) can begin to flow between the two parties. Note that at this point the parties know each other's addresses, so the ACK can be sent directly, bypassing the proxies. At this point the proxies are no longer involved in the call. Note that the media will therefore typically take a different path through the network than the original signalling messages. Furthermore, even if one or both of the proxies were to crash at this point, the call could continue on normally. Finally, when one party wishes to end the session, it sends a BYE message, which elicits a 200 OK response under normal circumstances.

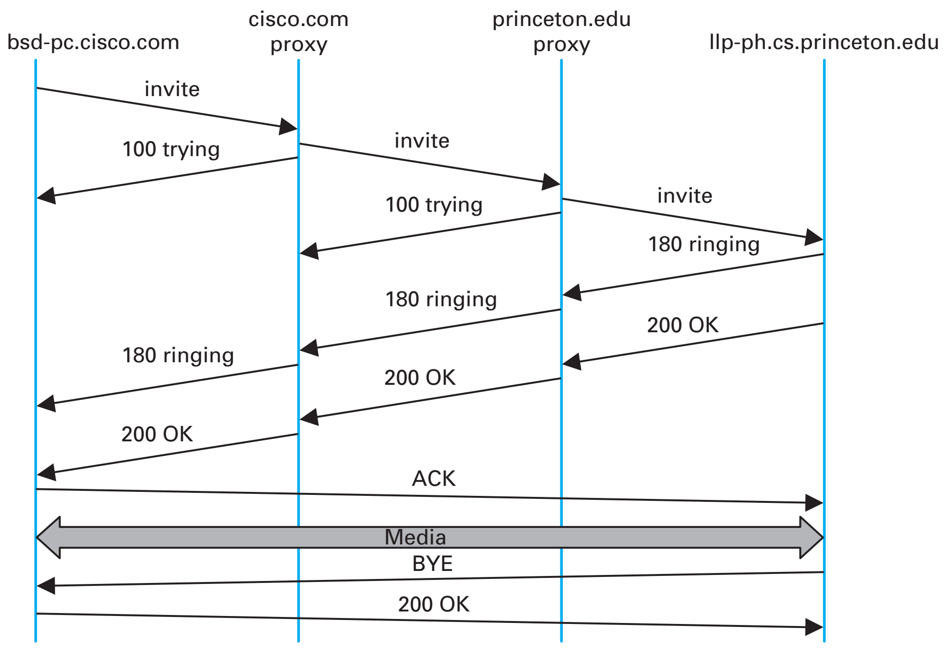


Figure 9.15 Message flow for a basic SIP session.

There are a few details that we have glossed over. One is the negotiation of session characteristics. Perhaps Bruce would have liked to communicate using both audio and video, but Larry's phone only supports audio. Thus Larry's phone would send an SDP message in its 200 OK describing the properties of the session that will be acceptable to Larry and the device, considering the options that were proposed in Bruce's invite. In this way, mutually acceptable session parameters are agreed before the media flow starts.

The other big issue we have glossed over is that of locating the correct device for Larry. First, Bruce's computer had to send its invite to the cisco.com proxy. This could have been a configured piece of information in the computer, or it could have been learned by DHCP. Then the cisco.com proxy had to find the princeton.edu proxy. This could be done using a special sort of DNS lookup that would return the IP address of the SIP proxy for the princeton.edu domain. Finally, the princeton.edu proxy had to find a device on which Larry could be contacted. Typically, a proxy server has access to a location database that can be populated in several ways. Manual configuration is one option, but a more flexible option is to use the *registration* capabilities of SIP.

A user can register with a location service by sending a SIP register message to the *registrar* for his domain. This message creates a binding between an *address of* *record* and a *contact address*. An address of record is likely to be a SIP URI that is the "well-known" address for the user (e.g., sip:larry@princeton.edu), and the contact address will be the address at which the user can currently be found (e.g., sip:larry@llp-ph.cs.princeton.edu). This is exactly the binding that was needed by the princeton.edu proxy in our example.

Note that a user may register at several locations and that multiple users may register at a single device. For example, one can imagine a group of people walking into a conference room that is equipped with an IP phone and all of them registering on it so that they can receive calls on that phone.

SIP is a very rich and flexible protocol that can support a wide range of complex calling scenarios as well as applications that have little or nothing to do with telephony. For example, SIP supports operations that enable a call to be routed to a "music-on-hold" server or a voicemail server. It is also easy to see how it could be used for applications like instant messaging; the SIMPLE working group at the IETF is defining standards in that area at the time of writing.

H.323

The ITU has also been very active in the call control area, which is not surprising given its relevance to telephony, the traditional realm of that body. Fortunately, there has been considerable coordination between the IETF and the ITU in this instance, so that the various protocols are somewhat interoperable. The major ITU recommendation for multimedia communication over packet networks is known as H.323, which ties together many other recommendations, including H.225 for call control. The full set of recommendations covered by H.323 runs to many hundreds of pages, and the protocol is known for its complexity, so it is only possible to give a brief overview of it here.

H.323 is popular as a protocol for Internet telephony, and we consider that application here. A device that originates or terminates calls is known as an H.323 terminal; this might be a workstation running an Internet telephony application, or it might be a specially designed "appliance"—a telephonelike device with networking software and an Ethernet port, for example. H.323 terminals can talk to each other directly, but the calls are frequently mediated by a device known as a *gatekeeper*. Gatekeepers perform a number of functions such as translating among the various address formats used for phone calls, and controlling how many calls can be placed at a given time to limit the bandwidth used by the H.323 applications. H.323 also includes the concept of a *gateway*, which connects the H.323 network to other types of networks. The most common use of a gateway is to connect an H.323 network to the public switched telephone network (PSTN) as illustrated in Figure 9.16. This enables a user running an H.323 application on a computer to talk to a person using a conventional phone on the public telephone network. One useful function performed by the gatekeeper is to help a terminal find a gateway, perhaps choosing among several

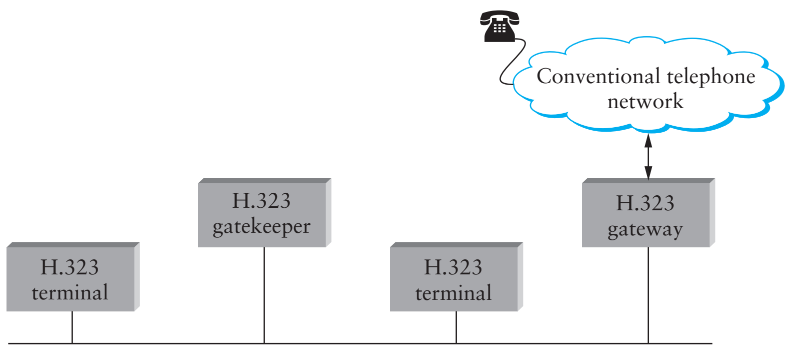


Figure 9.16 Devices in an H.323 network.

options to find one that is relatively close to the ultimate destination of the call. This is clearly useful in a world where conventional phones greatly outnumber PC-based phones. When an H.323 terminal makes a call to an endpoint that is a conventional phone, the gateway becomes the effective endpoint for the H.323 call and is responsible for performing the appropriate translation of both signalling information and the media stream that need to be carried over the telephone network.

An important part of H.323 is the H.245 protocol, which is used to negotiate the properties of the call, somewhat analogously to the use of SDP described above. An H.245 message might list a number of different audio codec standards that it can support, and the far endpoint of the call would reply with a list of its own supported codecs, and the two ends could pick a coding standard that they can both live with. H.245 can also be used to signal the UDP port numbers that will be used by RTP and RTCP for the media stream (or streams—a call might include both audio and video, for example) in this call. Once this is accomplished, the call can proceed, with RTP being used to transport the media streams and RTCP carrying the relevant control information.

9.4 Overlay Networks

From its inception, the Internet has adopted a clean model, in which the routers inside the network are responsible for forwarding packets from source to destination, and application programs run on the hosts connected to the edges of the network. The client/server paradigm illustrated by the applications discussed in the first two sections of this chapter certainly adhere to this model.

In the last few years, however, the distinction between *packet forwarding* and *application processing* has become less clear. New applications are being distributed across the Internet, and in many cases, these applications make their own forwarding decisions. These new hybrid applications can sometimes be implemented by extending

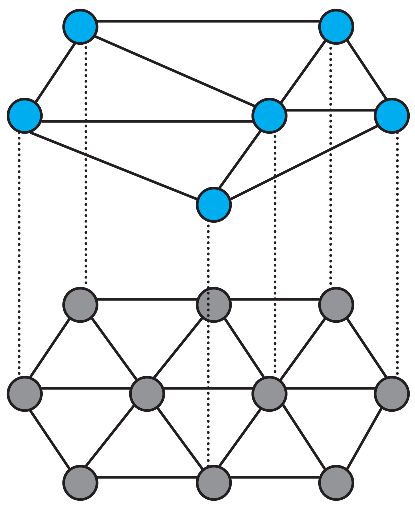


Figure 9.17 Overlay network layered on top of a physical network.

traditional routers and switches to support a modest amount of application-specific processing. For example, so-called *level-7 switches* sit in front of server clusters and forward HTTP requests to a specific server based on the requested URL. However, *overlay networks* are quickly emerging as the mechanism of choice for introducing new functionality into the Internet.

You can think of an overlay as a logical network implemented on top of a physical network. By this definition, the Internet itself is an overlay network, which is, in fact, a true statement. Figure 9.17 depicts an overlay implemented on top of an underlying network. Each node in the overlay also exists in the underlying network; it processes and forwards packets in an application-specific way. The links that connect the overlay nodes are implemented as tunnels through the underlying network. Multiple overlays networks can exist on top of the same underlying network—each implementing their own application-specific behavior—and overlays can be nested, one on top of another. For example, all of the example overlay networks discussed in this section treat today's Internet as the underlying network.

We have already seen examples of tunneling, for example, to implement virtual private networks (VPNs). As a brief refresher, the nodes on either end of a tunnel treat the multihop path between them as a single logical link, where the nodes that are "tunneled through" forward packets based on the outer header, never aware that the end nodes have attached an inner header. For example, Figure 9.18 shows three overlay nodes (A, B, and C) connected by a pair of tunnels. In this example, overlay node B might make a forwarding decision for packets from A to C based on the inner header

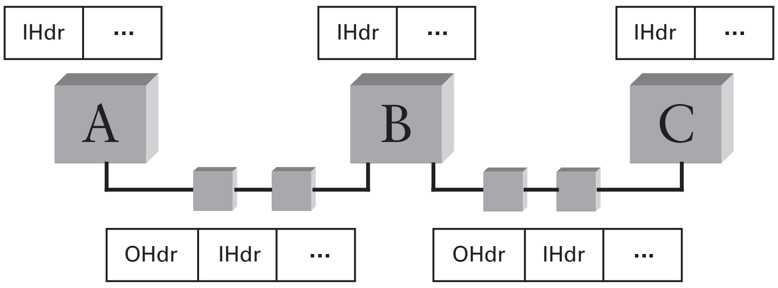


Figure 9.18 Overlay nodes tunnel through physical nodes.

(IHdr), and then attach an outer header (OHdr) that identifies C as the destination in the underlying network. Nodes A, B, and C are able to interpret both the inner and outer headers, whereas the intermediate routers understand only the outer header. Similarly, A, B, and C have addresses in both the overlay network and the underlying network, but they are not necessarily the same; for example, their underlying address might be a 32-bit IP address, while their overlay address might be an experimental 128-bit address. In fact, the overlay need not use conventional addresses at all, but may route based on URLs, domain names, an XML query, or even the content of the packet.

9.4.1 Routing Overlays

The simplest kind of overlay is one that exists purely to support an alternative routing strategy; no additional application-level processing is performed at the overlay nodes. You can view a virtual private network (see Section 4.1.8) as an example of a routing overlay, but one that doesn't so much define an alternative strategy or

Overlays and the Ossification of the Internet

Given its popularity and widespread use, it is easy to forget that at one time the Internet was a laboratory for researchers to experiment with packet-switched networking. The more the Internet has become a commercial success, however, the less useful it is as a platform for playing with new ideas. Today, commercial interests shape the Internet's continued development.

In fact, a recent report from the National Research Council points to the ossification of the Internet, both intellectually (pressure for compatibility with current standards stifles innovation) and in terms of the infrastructure itself (it is nearly impossible for researchers to affect the core infrastructure). The report goes on to observe that at the same time, a whole new set of challenges are emerging that algorithm, as it defines alternative routing table entries to be processed by the standard IP forwarding algorithm. In this particular case, the overlay is said to use “IP tunnels,” and the ability to utilize these VPNs is supported in most commercial routers.

Suppose, however, you wanted to use a routing algorithm that commercial router vendors were not willing to include in their products. How would you go about doing it? You could simply run your algorithm on a collection of end hosts and tunnel through the Internet routers. These hosts would behave like routers in the overlay network: as hosts they are likely connected to the Internet by only one physical link, but as a node in the overlay they would be connected to multiple neighbors via tunnels.

Since overlays, almost by definition, are a way to introduce new technologies independent of the standardization process, there are no standard overlays we can point to as examples. Instead, we illustrate the general idea of routing overlays by describing several experimental systems recently proposed by network researchers.

Experimental Versions of IP

Overlays are ideal for deploying experimental versions of IP that you hope will eventually take over the world. For example, IP multicast is an extension to IP that interprets class D addresses (those with the prefix 1110) as multicast addresses. IP multicast is used in conjunction with one of the multicast routing protocols, such as DVMRP, described in Section 4.4.

The MBone (multicast backbone) is an overlay network that implements IP multicast. One of the most popular applications run on top of the MBone is vic, a tool that supports multiparty videoconferencing. vic is used to broadcast both seminars and meetings across the Internet. For example, IETF meetings—which are a week long and attract thousands of participants—are generally broadcast over the MBone.

Like VPNs, the MBone uses both IP tunnels and IP addresses, but unlike VPNs, the MBone implements a different forwarding algorithm—it forwards

may require a fresh approach. The dilemma, according to the report, is that

> . . .successful and widely adopted technologies are subject to ossification, which makes it hard to introduce new capabilities or, if the current technology has run its course, to replace it with something better. Existing industry players are not generally motivated to develop or deploy disruptive technologies. . .

Finding the right way to introduce disruptive technologies is an interesting issue. Such innovations are likely to do some things very well, but overall they lag present technology in other important areas. For example, to introduce a new routing strategy into the Internet, one would have to build a router that not only supports this new strategy, but also competes with established vendors in terms of performance, packets to all downstream neighbors in the shortest-path multicast tree. As an overlay, multicast-aware routers tunnel through legacy routers, with the hope that one day there will be no more legacy routers.

The 6-Bone is a similar overlay that is used to incrementally deploy IPv6. Like the MBone, the 6-Bone uses tunnels to forward packets through IPv4 routers. Unlike the MBone, however, 6-Bone nodes do not simply provide a new interpertation of IPv4's 32-bit addresses. Instead, they forward packets based on IPv6's 128-bit address space. Moreover, since IPv6 supports multicast, so does the 6-Bone.

reliability, management toolset, and so on. This is an extremely tall order. What the innovator needs is a way to allow users to take advantage of the new idea without having to write the hundreds of thousands of lines of code needed to support just the base system.

Overlay networks provide exactly this opportunity. Overlay nodes can be programmed to support the new capability or feature, and then depend on conventional nodes to provide the underlying connectivity. Over time, if the idea deployed in the overlay proves useful, there may be economic motivation to migrate the functionality into the base system, that is, add it to the feature set of commercial routers. On the other hand, the functionality may be complex enough that an overlay layer may be exactly where it belongs.

End System Multicast

Although the MBone remains a popular overlay, IP multicast has failed to take over the world, and in response, multicast-based applications like videoconferencing have recently turned to an alternative strategy, called *end system multicast*. The idea of end system multicast is to accept that IP multicast will never become ubiquitous, and to instead let the end hosts that are participating in a particular multicast-based application implement their own multicast trees. (As an aside, there is a school of thought that says IP multicast never took off because it simply doesn't belong at the network layer, since it must support high-layer functionality such as error, flow, and congestion control, as well as membership management.)

Before describing how end system multicast works, it is important to first understand that, unlike VPNs and the MBone, end system multicast assumes that only Internet hosts (as opposed to Internet routers) participate in the overlay. Moreover, these hosts typically exchange messages with each other through UDP tunnels rather than IP tunnels, making it easy to implement as regular application programs. This makes it possible to view the underlying network as a fully connected graph, since every host in the Internet is able to send a message to every other host. Abstractly, then, end system multicast solves the following problem: starting with a fully connected

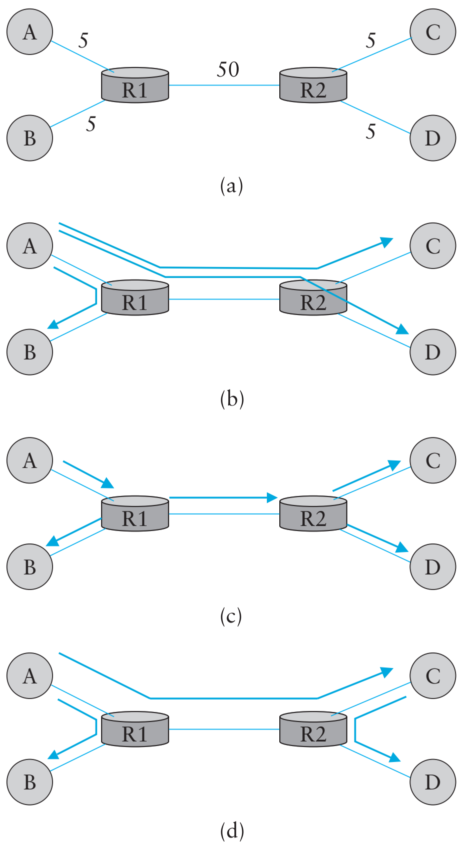


Figure 9.19 Alternative multicast trees mapped onto a physical topology (a) physical topology (b) naive unicast transmission (c) multicast tree constructed at the network level (by routers) (d) multicast tree constructed at the application level (by end hosts).

graph representing the Internet, the goal is to find the embedded multicast tree that spans all the group members.

Since we take the underlying Internet to be fully connected, a naive solution would be to have each source directly connected to each member of the group. In other words, end system multicast could be implemented by having each node send unicast messages to every group member. To see the problem in doing this, especially compared to implementing IP multicast in routers, consider the example topology in Figure 9.19. Figure 9.19(a) depicts an example physical toplogy, where R1 and R2 are routers connected by a low-bandwidth transcontinental link; A, B, C, and D are end hosts; and link delays are given as edge weights. Assuming A wants to send a multicast

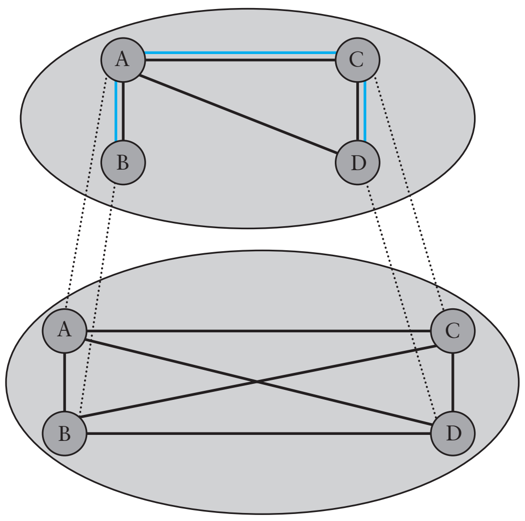


Figure 9.20 Multicast tree embedded in an overlay mesh.

message to the other three hosts, Figure 9.19(b) shows how naive unicast transmission would work. This is clearly undesirable because the same message must traverse the link A–R1 three times, and two copies of the message traverse R1–R2. Figure 9.19(c) depicts the IP multicast tree constructed by DVMRP. Clearly, this approach eliminates the redundant messages. Without support from the routers, however, the best one can hope for with end system multicast is a tree similar to the one shown in Figure 9.19(d). End system multicast defines an architecture for constructing this tree.

The general approach is to support multiple levels of overlay networks, each of which extracts a subgraph from the overlay below it, until we have selected the subgraph that the application expects. For end system multicast in particular, this happens in two stages: first we construct a simple *mesh* overlay on top of the fully connected Internet, and then we select a multicast tree within this mesh. The idea is illustrated in Figure 9.20, again assuming the four end hosts A, B, C, and D. The first step is the critical one: once we have selected a suitable mesh overlay, we simply run a standard multicast routing algorithm (e.g., DVMRP) on top of it to build the multicast tree. We also have the luxury of ignoring the scalability issue that Internet-wide multicast faces since the intermediate mesh can be selected to include only those nodes that want to participate in a particular multicast group.

The key to constructing the intermediate mesh overlay is to select a topology that roughly corresponds to the physical topology of the underlying Internet, but we have to do this without anyone telling us what the underlying Internet actually looks like since we are running only on end hosts and not routers. The general strategy is for the end hosts to measure the round-trip latency to other nodes and to decide to add links to the mesh only when they like what they see. This works as follows.

First, assuming a mesh already exists, each node exchanges the list of all other nodes it believes is part of the mesh with its directly connected neighbors. When a node receives such a membership list from a neighbor, it incorporates that information into its membership list and forwards the resulting list to its neighbors. This information eventually propagates through the mesh, much as in a distance vector routing protocol.

When a host wants to join the multicast overlay, it must know the IP address of at least one other node already in the overlay. It then sends a "join mesh" message to this node. This connects the new node to the mesh by an edge to the known node. In general, the new node might send a join message to multiple current nodes, thereby joining the mesh by multiple links. Once a node is connected to the mesh by a set of links, it periodically sends "keep alive" messages to its neighbors, letting it know that it still wants to be part of the group.

When a node leaves the group, it sends a "leave mesh" message to its directly connected neighbors, and this information is propagated to the other nodes in the mesh via the membership list described above. Alternatively, a node can fail, or just silently decide to quit the group, in which case its neighbors detect that it is no longer sending "keep alive" messages. Some node departures have little effect on the mesh, but should a node detect that the mesh has become partitioned due to a departing node, it creates a new edge to a node in the other partition by sending it a "join mesh" message. Note that multiple neighbors can simultaneously decide that a partition has occurred in the mesh, leading to multiple cross-partition edges being added to the mesh.

As described so far, we will end up with a mesh that is a subgraph of the original fully connected Internet, but it may have suboptimal performance because (1) initial neighbor selection adds random links to the topology, (2) partition repair might add edges that are essential at the moment but not useful in the long run, (3) group membership may change due to dynamic joins and departures, and (4) underlying network conditions may change. What needs to happen is that the system must evaluate the value of each edge, resulting in new edges being added to the mesh and existing edges being removed over time.

To add new edges, each node i periodically probes some random member j that it is not currently connected to in the mesh, measures the round-trip latency of edge (i, j), and then evaluates the utility of adding this edge. If the utility is above a certain threshold, link (i, j) is added to the mesh. Evaluating the utility of adding edge (i, j) might look something like this:

```
EvaluateUtility($j$)
  utility = 0
  for each member $m$ not equal to $i$
    CL = current latency to node $m$ along route through mesh
    NL = new latency to node $m$ along mesh if edge $(i, j)$ is added
    if (NL $<$ CL) then
      utility += (CL - NL)/CL
    return utility
```

Deciding to remove an edge is similar, except each node i computes the cost of each link to current neighbor j as follows:

```
EvaluateCost($j$)
  Cost$_{ij}$ = number of members for which $i$ uses $j$ as next hop
  Cost$_{ji}$ = number of members for which $j$ uses $i$ as next hop
  return max(Cost$_{ij}$, Cost$_{ji}$)
```

It then picks the neighbor with the lowest cost and drops it if the cost falls below a certain threshold.

Finally, since the mesh is maintained using what is essentially a distance vector protocol, it is trivial to run DVMRP to find an appropriate multicast tree in the mesh. Note that although it is not possible to prove that the protocol just described results in the optimum mesh network, thereby allowing DVMRP to select the best possible multicast tree, both simulation and extensive practical experience suggests that it does a good job.

Resilient Overlay Networks

Another routing overlay gaining in popularity is one that finds alternative routes for traditional unicast applications. Such overlays exploit the observation that the triangle inequality does not hold in the Internet. Figure 9.21 illustrates what we mean by this. It is not uncommon to find three sites in the Internet—call them A, B, and C—such that the latency between A and B is greater than the sum of the latencies from A to C, and from C to B. That is, sometimes you would be better off indirectly sending your packets via some intermediate node than sending them directly to the destination.

How can this be? Well, BGP never promised that it would find the *shortest* route between any two sites; it only tries to find *some* route. To make matters worse, there are countless opportunities for human-directed policies to override BGP's normal operation. This often happens, for example, at peering points between major backbone ISPs. In short, that the triangle inequality does not hold in the Internet should not come as a surprise.

How do we exploit this observation? The first step is to realize that there is a fundamental trade-off between the scalability and optimality of a routing algorithm.

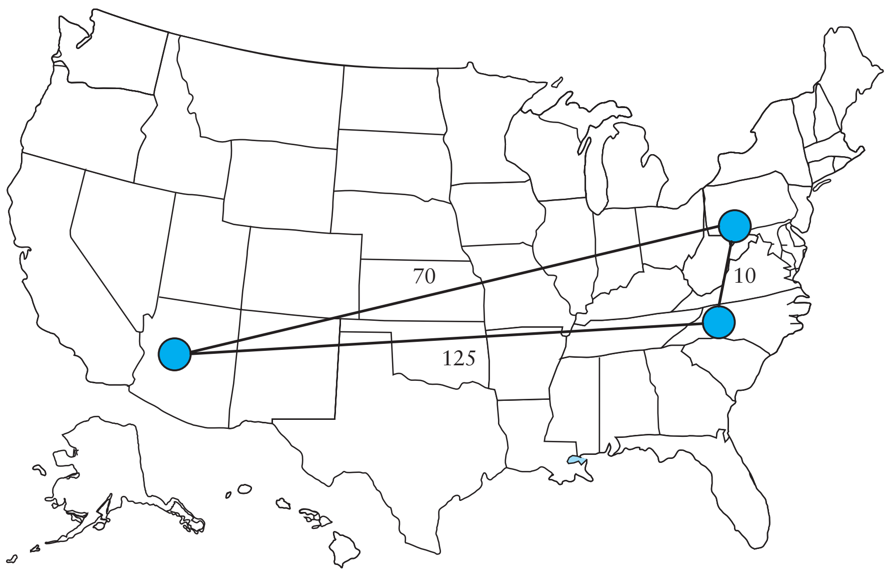


Figure 9.21 The triangle inequality does not necessarily hold in networks.

On the one hand, BGP scales to very large networks, but often does not select the best possible route and is slow to adapt to network outages. On the other hand, if you were only worried about finding the best route among a handful of sites, you could do a much better job of monitoring the quality of every path you might use, thereby allowing you to select the best possible route at any moment in time.

An experimental overlay, called RON (for resilient overlay network), does exactly this. RON scales to only a few dozen nodes because it uses an $N \times N$ strategy of closely monitoring (via active probes) three aspects of path quality—latency, available bandwidth, and loss probability—between every pair of sites. It is then able to both select the optimal route between any pair of nodes and rapidly change routes should network conditions change. Experience shows that RON is able to deliver modest performance improvements to applications, but more importantly, it recovers from network failures much more quickly. For example, during one 64-hour period in 2001, an instance of RON running on 12 nodes detected 32 outages lasting over 30 minutes, and it was able to recover from all of them in less than 20 seconds on average. This experiment also suggested that forwarding data through just one intermediate node is usually sufficient to recover from Internet failures.

Since RON does not scale, it is not possible to use RON to help random host A communicate with random host B; A and B have to know ahead of time that they are likely to communicate, and then join the same RON. However, RON seems like a good idea in certain settings, such as when connecting a few dozen corporate sites spread across the Internet, or allowing you and 50 of your friends to establish your own private overlay for the sake of running some application. The real question, though, is, What happens when everyone starts to run their own RON? Does the overhead of millions of RONs aggressively probing paths swamp the network, and does anyone see improved behavior when many RONs compete for the same paths? These questions are still unanswered.

All of these overlays illustrate a concept that is central to computer networks in general: *virtualization*. That is, it is possible to build a virtual network from abstract (logical) resources on top of a physical network constructed from physical resources. Moreover, it is possible to stack these virtualized networks on top of each other, and for multiple virtual network to co-exist at the same level. Each virtual network, in turn, provides new capabilities that are of value to some set of users, applications, or higher-level networks.

9.4.2 Peer-to-Peer Networks

Recent music-sharing applications like Napster and KaZaA have introduced the term "peer-to-peer" into the popular vernacular. But what exactly does it mean for a system to be "peer-to-peer"? Certainly in the context of sharing MP3 files it means not having to download music from a central site, but instead being able to access music files directly from whoever in the Internet happens to have a copy stored on their computer. More generally then, we could say that a peer-to-peer network allows a community of users to pool their resources (content, storage, network bandwidth, disk bandwidth, CPU), thereby providing access to larger archival stores, larger video/audio conferences, more complex searches and computations, and so on, than any one user could afford individually.

Quite often, attributes like *decentralized* and *self-organizing* are mentioned when discussing peer-to-peer networks, meaning that individual nodes organize themselves into a network without any centralized coordination. If you think about it, terms like these could be used to describe the Internet itself. Ironically, however, Napster is not a true peer-to-peer system by this definition since it depends on a central registery of known files, and users have to search this directory to find what machine offers a particular file. It is only the last step—actually downloading the file—that takes place between machines that belong to two users, but this is little more than a traditional client/server transaction. The only difference is that the server is owned by someone just like you rather than a large corporation.

So we are back to the original question: What's interesting about peer-to-peer networks? One answer is that both the process of locating an object of interest and

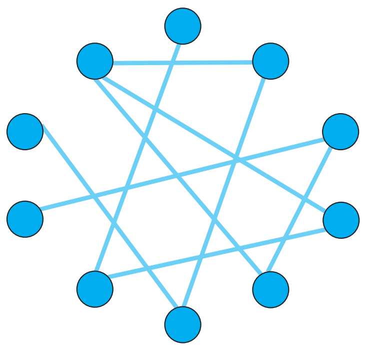


Figure 9.22 Example topology of a Gnutella peer-to-peer network.

the process of downloading that object onto your local machine happen without your having to contact a centralized authority, and at the same time, the system is able to scale to millions of nodes. A peer-to-peer system that can accomplish these two tasks in a decentralized manner turns out to be an overlay network, where the nodes are those hosts that are willing to share objects of interest (e.g., music and other assorted files), and the links (tunnels) connecting these nodes represent the sequence of machines that you have to visit to track down the object you want. This description will become more clear after we look at two examples.

Gnutella

Gnutella is an early peer-to-peer network that attempted to distinguish between exchanging music (which likely violates somebody's copyright) and the general sharing of files (which must be good since we've been taught to share since the age of two). What's interesting about Gnutella is that it was one of the first such systems to not depend on a centralized registry of objects. Instead Gnutella participants arrange themselves into an overlay network similar to the one shown in Figure 9.22. That is, each node that runs the Gnutella software (i.e., implements the Gnutella protocol) knows about some set of other machines that also run the Gnutella software. The relationship "A and B know each other" corresponds to the edges in this graph. (We'll talk about how this graph is formed in a moment.)

Whenever the user on a given node wants to find an object, Gnutella sends a QUERY message for the object—for example, specifying the file's name—to its neighbors in the graph. If one of the neighbors has the object, it responds to the node that sent it the query with a QUERY RESPONSE message, specifying where the object can be downloaded (e.g., an IP address and TCP port number). That node can subsequently use GET or PUT messages to access the object. If the node cannot resolve the query, it forwards the QUERY message to each of its neighbors (except the one that sent it the query), and the process repeats. In other words, Gnutella floods the overlay to locate the desired object. Gnutella sets a TTL on each query so this flood does not continue indefinitely.

In addition to the TTL and query string, each QUERY message contains a unique query identifier (QID), but it does not contain the identity of the original message source. Instead, each node maintains a record of the QUERY messages it has seen recently: both the QID and the neighbor that sent it the QUERY. It uses this history in two ways. First, if it ever receives a QUERY with a QID that matches one it has seen recently, the node does not forward the QUERY message. This serves to cut off forwarding loops more quickly than the TTL might have done. Second, whenever the node receives a QUERY RESPONSE from a downstream neighbor, it knows to forward the response to the upstream neighbor that originally sent it the QUERY message. In this way, the response works its way back to the original node without any of the intermediate nodes knowing who wanted to locate this particular object in the first place.

Returning to the question of how the graph evolves, a node certainly has to know about at least one other node when it joins a Gnutella overlay. The new node is attached to the overlay by at least this one link. After that, a given node learns about other nodes as the result of QUERY RESPONSE messages, both for objects it requested and for responses that just happen to pass through it. A node is free to decide which of the nodes it discovers in this way that it wants to keep as a neighbor. The Gnutella protocol provides PING and PONG messages by which a node probes whether or not a given neighbor still exists and by which that neighbor responds, respectively.

It should be clear that Gnutella is not a particularly clever protocol, and subsequent systems have tried to improve upon it. One dimension along which improvements are possible is in how queries are propogated. Flooding has the nice property that it is guaranteed to find the desired object in the fewest possible hops, but it does not scale well. It is possible to forward queries randomly, or according to the probability of success based on past results. A second dimension is to proactively replicate the objects, since the more copies of a given object there are, the easier it should be to find a copy. Alternatively, one could develop a completely different strategy, which is the topic we consider next.

Structured Overlays

At the same time file sharing systems have been fighting to fill the void left by Napster, the research community has been exploring an alternative design for peer-to-peer networks. We refer to these networks as *structured*, to contrast them with the essentially random (unstructured) way in which a Gnutella network evolves. Unstructured overlays like Gnutella employ trivial overlay construction and maintenance algorithms, but the best they can offer is unreliable, random search. In contrast, structured overlays are designed to conform to a particular graph structure that allows reliable and efficient (probabilistically bounded delay) object location, in return for additional complexity during overlay construction and maintenance.

If you think about what we are trying to do at a high level, there are two questions to consider: (1) how do we map objects onto nodes, and (2) how do we route requests to the node that is responsible for a given object? We start with the first question, which has a simple statement: How do we map an object with name x into the address of some node n that is able to serve that object? While traditional peer-to-peer networks have no control over which node hosts object x, if we could control how objects get distributed over the network, we might be able to do a better job of finding those objects at a later time.

A well-known technique for mapping names into addresses is to use a hash table, so that

$$hash(x) \longrightarrow n$$

implies object x is first placed on node n, and at a later time, a client trying to locate x would only have to perform the hash of x to determine that it is on node n. A hash-based approach has the nice property that it tends to spread the objects evenly across the set of nodes, but straightforward hashing algorithms suffer from a fatal flaw: how many possible values of n should we allow? (In hashing terminology, how many buckets should there be?) Naively, we could decide that there are, say, 101 possible hash values, and we use a modulo hash function; that is,

```
hash($x$)
  return $x$ % 101
```

Unfortunately, if there are more than 101 nodes willing to host objects, then we can't take advantage of all of them. On the other hand, if we select a number larger than the largest possible number of nodes, then there will be some values of x that will hash into an address for a node that does not exist. There is also the not-so-small issue of translating the value returned by the hash function into an actual IP address.

To address these issues, structured peer-to-peer networks use an algorithm known as *consistent hashing*, which hashes a set of objects x uniformly across a large id space. Figure 9.23 visualizes a 128-bit id space as a circle, where we use the algorithm to place both objects

$$hash(object\_name) \longrightarrow objid$$

and nodes

$$hash(IP\_addr) \longrightarrow nodeid$$

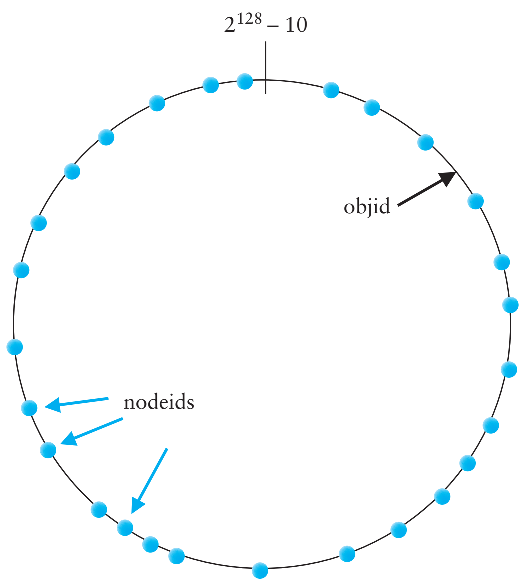


Figure 9.23 Both nodes and objects map (hash) onto the id space, where objects are maintained at the nearest node in this space.

onto this circle. Since a 128-bit id space is enormous, it is unlikely that an object will hash to exactly the same id as a machine's IP address hashes to. To account for this unlikelihood, each object is maintained on the node who's id is *closest*, in this 128-bit space, to the object id. In other words, the idea is to use a high-quality hash function to map both nodes and objects into the same large, sparse id space; you then map objects to nodes by numerical proximity of their respective identifiers. Like ordinary hashing, this distributes objects fairly evenly across nodes, but unlike ordinary hashing, only a small number of objects have to move when a node (hash bucket) joins or leaves.

We now turn to the second question—how does a user who wants to access object *x* know which node is closest in *x*'s id in this space? One possible answer is that each node keeps a complete table of node ids and their associated IP addresses, but this would not be practical for a large network. The alternative, which is the approach used by structured peer-to-peer networks, is to *route a message to this node!* In other words, if we construct the overlay in a clever way—which is the same as saying that we need to choose entries for a node's routing table in a clever way—then we find a node simply by routing toward it. Collectively, this approach is sometimes called *distributed hash tables* (DHT), since conceptually, the hash table is distributed over all the nodes in the network.

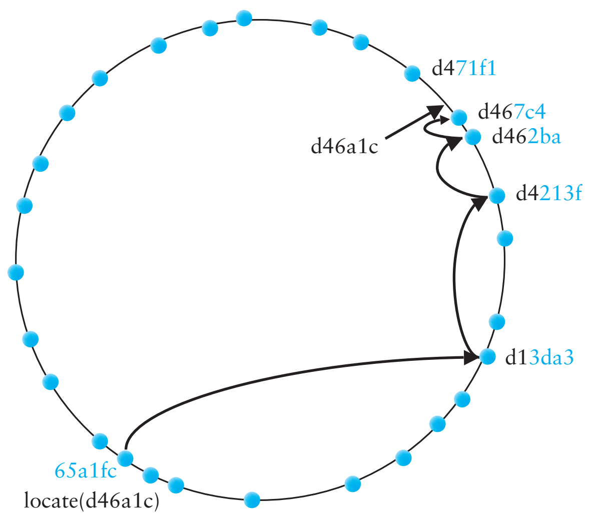


Figure 9.24 Objects are located by routing through the peer-to-peer overlay network.

Figure 9.24 illustrates what happens for a simple 28-bit id space. To keep the discussion as concrete as possible, we consider the approach used by a particular peer-to-peer network called Pastry. Other systems work in a similar manner. (See the papers cited at the end of the chapter for additional examples.)

Suppose you are at the node with id `65a1fc` (hex) and you are trying to locate the object with id `d46a1c`. You realize that your id shares nothing with the object's, but you know of a node that shares at least the prefix `d`. That node is closer than you in the 128-bit id space, so you forward the message to it. (We do not give the format of the message being forwarded, but you can think of it as saying, "Locate object `d46a1c`.") Assuming node `d13da3` knows of another node that shares an even longer prefix with the object, it forwards the message on. This process of moving closer in id space continues until you reach a node that knows of no closer node. This node is, by definition, the one that hosts the object. Keep in mind that as we logically move through "id space" the message is actually being forwarded, node to node, through the underlying Internet.

Each node maintains both a routing table (more below) and the IP addresses of a small set of numerically larger and smaller node ids. This is called the node's *leaf set*. The relevance of the leaf set is that once a message is routed to any node in the same leaf set as the node that hosts the object, that node can directly forward the message to the ultimate destination. Said another way, the leaf set facilitates correct and efficient delivery of a message to the numerically closest node, even though multiple nodes may exist that share a maximal length prefix with the object id. Moreover, the leaf set makes routing more robust because any of the nodes in a leaf set can route a message just as well as any other node in the same set. Thus, if one node is unable to make progress routing a message, one of its neighbors in the leaf set may be able to. In summary, the routing procedure is defined as follows:

```
Route($D$)
  if $D$ is within range of my leaf set
    forward to numerically closest member in leaf set
  else
    let $l$ = length of shared prefix
    let $d$ = value of $l$th digit in $D$'s address
    if RouteTab[$l$,$d$] exists
      forward to RouteTab[$l$,$d$]
    else
      forward to known node with at least as long a prefix
      and is numerically closer than this node
```

The routing table, denoted RouteTab, is a two-dimensional array. It has a row for every hex digit in an id (there such 32 digits in a 128-bit id) and a column for every hex value (there are obviously 16 such values). Every entry in row i shares a prefix of length i with this node, and within this row, the entry in column j has the hex value j in the $i + 1$th position. Figure 9.25 shows the first three rows of an example routing table for node 65a1fc$x$, where x denotes an unspecified suffix. This figure shows the id prefix matched by every entry in the table. It does not show the actual value contained in this entry—the IP address of the next node to route to.

Adding a node to the overlay works much like routing a "locate object message" to an object. The new node must know of at least one current member. It asks this member to route an "add node message" to the node numerically closest to the id of the joining node, as shown in Figure 9.26. It is through this routing process that the new node learns about other nodes with a shared prefix and is able to begin filling out its routing table. Over time, as additional nodes join the overlay, existing nodes also have the option of including information about the newly joining node in their routing tables. They do this when the new node adds a longer prefix than they currently have in their table. Neighbors in the leaf sets also exchange routing tables with each other, which means that over time routing information propagates through the overlay.

You may have noticed that although structured overlays provide a probabilistic bound on the number of routing hops required to locate a given object—the number of hops in Pastry is bounded by $\log_{16}N$, where N is the number of nodes in the overlay—each hop may contribute substantial delay. This is because each intermediate node may

Row 0	*0x*	*1x*	*2x*	*3x*	*4x*	*5x*		*7x*	*8x*	*9x*	*ax*	*bx*	*cx*	*dx*	*ex*	*fx*
Row 1	*60x*	*61x*	*62x*	*63x*	*64x*		*66x*	*67x*	*68x*	*69x*	*6ax*	*6bx*	*6cx*	*6dx*	*6ex*	*6fx*
Row 2	*650x*	*651x*	*652x*	*653x*	*654x*	*655x*	*656x*	*657x*	*658x*	*659x*		*65bx*	*65cx*	*65dx*	*65ex*	*65fx*
Row 3	*65a0x*		*65a2x*	*65a3x*	*65a4x*	*65a5x*	*65a6x*	*65a7x*	*65a8x*	*65a9x*	*65aax*	*65abx*	*65acx*	*65adx*	*65aex*	*65afx*

Figure 9.25 Example routing table at the node with id 65a1fc*x*.

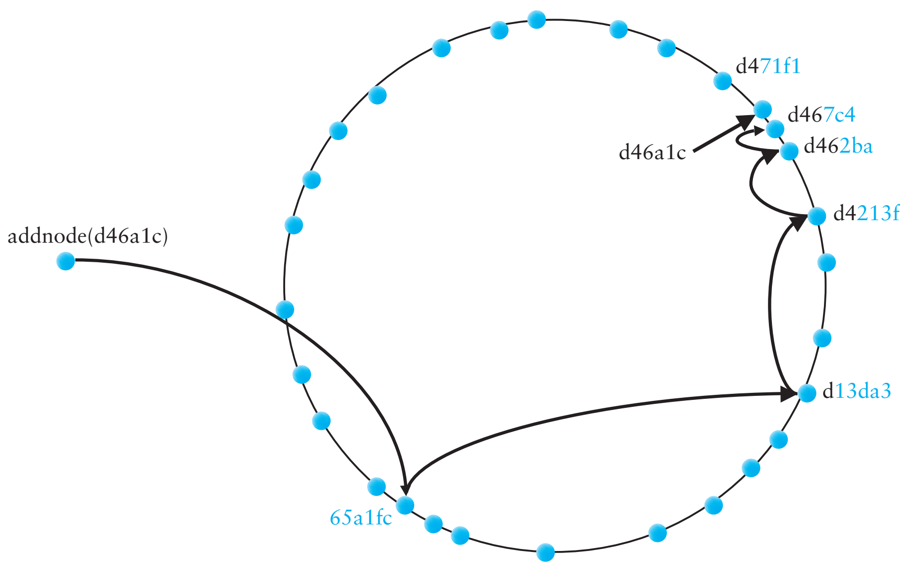


Figure 9.26 Adding a node to the network.

be at a random location in the Internet. (In the worst case, each node is on a different continent!) In fact, in a worldwide overlay network using the algorithm as described above, the expected delay of each hop is the average delay among all pairs of nodes in the Internet! Fortunately, we can do much better in practice. The idea is to choose each routing table entry such that it refers to a nearby node in the underlying physical network, among all nodes with an id prefix that is appropriate for the entry. It turns out that doing so achieves end-to-end routing delays that are within a small factor of the delay between source and destination node.

Finally, the discussion up to this point has focused on the general problem of locating objects in a peer-to-peer network. Given such a routing infrastructure, it is possible to build different services. For example, a file sharing service would use file names as object names. To locate a file, you first hash its name into a corresponding object id, and then route a "locate object message" to this id. The system might also replicate each file across multiple nodes to improve availability. Storing multiple copies on the leaf set of the node to which a given file normally routes would be one way of doing this. Keep in mind that even though these nodes are neighbors in the id space, they are likely to be physically distributed across the Internet. Thus, while a power-outage in an entire city might take down physically close replicas of a file in a traditional file system, one or more replicas would likely survive such a failure in a peer-to-peer network.

Services other than file-sharing can also be built on top of distributed hash tables. Consider multicast applications, for example. Instead of constructing a multicast tree from a mesh, one could construct the tree from edges in the structured overlay, thereby amortizing the cost of overlay construction and maintenance across several applications and multicast groups.

9.4.3 Content Distribution Networks

We have already seen how HTTP running over TCP allows Web browsers to retrieve pages from Web servers. However, anyone that has waited an eternity for a Web page to return knows that the system is far from perfect. Considering that the backbone of the Internet is now constructed from OC-192 (10-Gbps) links, it's not obvious why this should happen. It is generally agreed that when it comes to downloading Web pages, there are three potential bottlenecks in the system:

- The first mile: The Internet may have high-capacity links in it, but that doesn't help you download a Web page any faster when you're connected by a 56-Kbps modem.
- The last mile: The link that connects the server to the Internet, along with the server itself, can be overloaded by too many requests.

- Peering points: The handful of ISPs that collectively implement the backbone of the Internet may internally have high-bandwidth pipes, but they have little motivation to provide high-capacity connectivity to their peers. If you are connected to ISP A and the server is connected to ISP B, then the page you request may get dropped at the point A and B peer with each other.

There's not a lot anyone except you can do about the first problem, but it is possible to use replication to address the second and third problems. A system that does this is often called a *content distribution network* (CDN). Akamai and Digital Island are probably the two best-known CDNs.

The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than have millions of users wait forever to contact www.cnn.com when a big news story breaks—such a situation is known as a *flash crowd*—it is possible to spread this load across many servers. Moreover, rather than having to traverse multiple ISPs to reach www.cnn.com, if these surrogate servers happen to be spread across all the backbone ISPs, then it should be possible to reach one without having to cross a peering point. Clearly, maintaining thousands of surrogate servers all over the Internet is too expensive for any one site that wants to provide better access to its Web pages. Commercial CDNs provide this service for many sites, thereby amortizing the cost across many customers.

Although we call them surrogate servers, in fact, they can just as correctly be viewed as caches. If they don't have a page that has been requested by a client, they ask the backend server for it. In practice, however, the backend servers proactively replicate their data across the surrogates rather than wait for surrogates to request it on demand. It's also the case that only static pages, as opposed to dynamic content, are distributed across the surrogates. Clients have to go to the backend server for any content that either changes frequently (e.g., sports scores and stock quotes) or is produced as the result of some computation (e.g., a database query).

Having a large set of geographically distributed servers does not fully solve the problem. To complete the picture, CDNs also need to provide a set of *redirectors* that forward client requests to the most appropriate server, as shown in Figure 9.27. The primary objective of the redirectors is to select the server for each request that results in the best *response time* for the client. A secondary objective is for the system as a whole to process as many requests per second as the underlying hardware (network links and Web servers) is able to support. The average number of requests that can be satisfied in a given time period—known as the *system throughput*—is primarily an issue when the system is under heavy load, for example, when a flash crowd is accessing a small set of pages or a distributed denial of service (DDoS) attacker is targeting a particular site, as happened to CNN, Yahoo, and several other high-profile sites in February 2000.

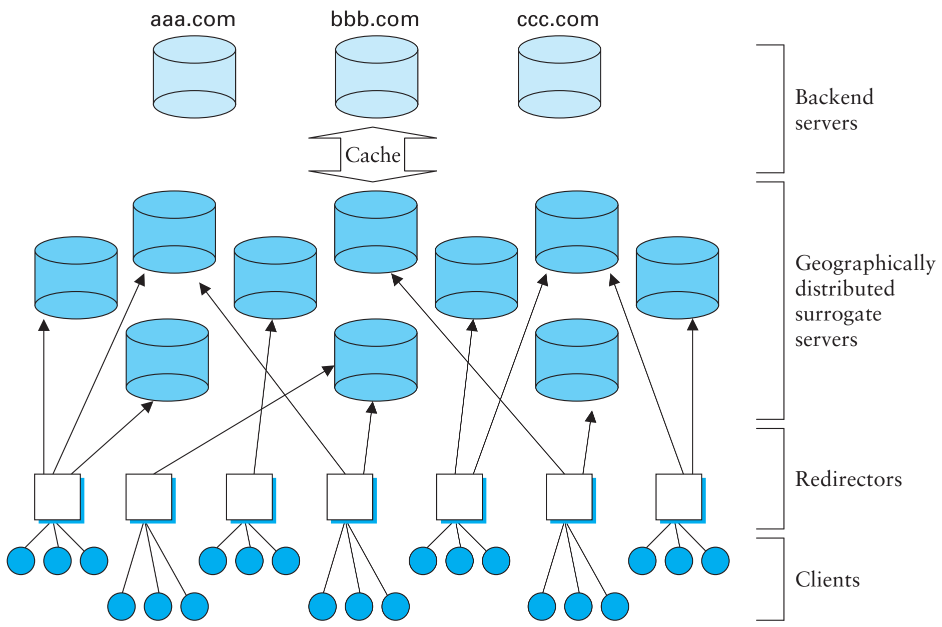


Figure 9.27 Components in a content distribution network (CDN).

CDNs use several factors to decide how to distribute client requests. For example, to minimize response time, a redirector might select a server based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration, that is, selects a server that is likely to already have the page being requested in its cache. The exact combination of factors that should be employed by a CDN is open to debate. This section considers some of the possibilities.

Mechanisms

As described so far, a redirector is just an abstract function, although it sounds like something a router might be asked to do since it logically forwards a request message much like a router forwards packets. In fact, there are several mechanisms that can be used to implement redirection. Note that for the purpose of this discussion we assume that each redirector knows the address of every available server. (From here on, we drop the "surrogate" qualifier and talk simply in terms of a set of servers.) In practice, some form of out-of-band communication takes place to keep this information up-to-date as servers come and go.

First, redirection could be implemented by augmenting DNS to return different server addresses to clients. For example, when a client asks to resolve the name www.cnn.com, the DNS server could return the IP address of a server hosting CNN's Web pages that is known to have the lightest load. Alternatively, for a given set of servers, it might just return addresses in a round-robin fashion. Note that the granularity of DNS-based redirection is usually at the level of a site (e.g., cnn.com) rather than a specific URL (e.g., http://www.cnn.com/2002/WORLD/europe/06/21/william.birthday/index.html). However, when returning an embedded link, the server can rewrite the URL, thereby effectively pointing the client at the most appropriate server for that specific object.

Commercial CDNs essentially use a combination of URL rewriting and DNS-based redirection. For scalability reasons, the high-level DNS server first points to a regional-level DNS server, which replies with the actual server address. In order to respond to changes quickly, the DNS servers tweak the TTL of the resource records they return to a very short period, such as 20 seconds. This is necessary so clients don't cache results, and thus fail to go back to the DNS server for the most recent URL-to-server mapping.

Another possibility is to use the HTTP redirect feature: the client sends a request message to a server, which responds with a new (better) server that the client should contact for the page. Unfortunately, server-based redirection incurs an additional round-trip time across the Internet, and even worse, servers can be vulnerable to being overloaded by the redirection task itself. Instead, if there is a node close to the client—for example, a local Web proxy—that is aware of the available servers, then it can intercept the request message and instruct the client to instead request the page from an appropriate server. In this case, either the redirector would need to be on a choke point so that all requests leaving the site pass through it, or the client would have to cooperate by explicitly addressing the proxy (as with a classical, rather than transparent, proxy).

At this point you may be wondering what CDNs have to do with overlay networks, and while viewing a CDN as an overlay is a bit of a stretch, they do share one very important trait in common. Like an overlay node, a proxy-based redirector makes an application-level routing decision. Rather than forward a packet based on an address and its knowledge of the network topology, it forwards HTTP requests based on a URL and its knowledge of the location and load of a set of servers. Today's Internet architecture does not support redirection directly—where by "directly" we mean the client sends the HTTP request to the redirector, which forwards it to the destination—so instead redirection is typically implemented indirectly by having the redirector return the appropriate destination address and the client contacts the server itself.

Policies

We now consider some example policies that redirectors might use to forward requests. Actually, we have already suggested one simple policy—round-robin. A similar scheme would be to simply select one of the available servers at random. Both of these approaches do a good job of spreading the load evenly across the CDN, but they do not do a particularly good job of lowering the client-perceived response time.

It's obvious that neither of these two schemes take network proximity into consideration, but just as importantly, they also ignore locality. That is, requests for the same URL are forwarded to different servers, making it less likely that the page will be served from the selected server's in-memory cache. This forces the server to retrieve the page from its disk, or possibly even from the backend server. How can a distributed set of redirectors cause requests for the same page to go to the same server (or small set of servers) without global coordination? The answer is surprisingly simple: all redirectors use some form of hashing to deterministically map URLs into a small range of values. The primary benefit of this approach is that no inter-redirector communication is required to achieve coordinated operation; no matter which redirector receives a URL, the hashing process produces the same output.

So what makes for a good hashing scheme? The classic *modulo* hashing scheme—which hashes each URL modulo the number of servers—is not suitable for this environment. This is because should the number of servers change, the modulo calculation will result in a diminishing fraction of the pages keeping their same server assignments. While we do not expect frequent changes in the set of servers, the fact that addition of new servers into the set will cause massive reassignment is undesirable.

An alternative is to use the same *consistent hashing* algorithm discussed in Section 9.4.2. Specifically, each redirector first hashes every server into the unit circle. Then for each URL that arrives, the redirector also hashes the URL to a value on the unit circle, and the URL is assigned to the server that lies closest on the circle to its hash value. If a node fails in this scheme, its load shifts to its neighbors (on the unit circle), so the addition/removal of a server only causes local changes in request assignments. Note that unlike the peer-to-peer case, where a message is routed from one node to another in order to find the server whose id is closest to the objects, each redirector knows how the set of servers map onto the unit circle, so they can each independently select the "nearest" one.

This strategy can easily be extended to take server load into account. Assume the redirector knows the current load of each of the available servers. This information may not be perfectly up-to-date, but we can imagine the redirector simply counting how many times it has forwarded a request to each server in the last few seconds, and using this count as an estimate of that server's current load. Upon receiving a URL, the redirector hashes the URL plus each of the available servers, and sorts the resulting values. This sorted list effectively defines the order in which the redirector will consider the available servers. The redirector then walks down this list until it finds a server whose load is below some threshold. The benefit of this approach compared to plain consistent hashing is that server order is different for each URL, so if one server fails, its load is distributed evenly among the other machines. This approach is the basis for the Cache Array Routing Protocol (CARP) and is shown in pseudocode below.

```
SelectServer($URL$, $S$)
   for = each server $s_i$ in server set $S$
      $weight_i$ = hash($URL$, address($s_i$))
   sort $weight$
   for each server $s_j$ in decreasing order of $weight_j$
      if = Load($s_j$) $<$ threshold then
         return $s_j$
   return server with highest weight
```

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some pages normally handled by "busy" servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual pages, servers hosting some popular pages may find more servers sharing their load than servers hosting collectively unpopular pages. In the process, some unpopular pages will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some pages become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

Finally, it is possible to introduce network proximity into the equation in at least two different ways. The first is to blur the distinction between server load and network proximity by monitoring how long a server takes to respond to requests, and using this measurement as the "server load" parameter in the preceding algorithm. This strategy tends to perfer nearby/lightly loaded servers over distant/heavily loaded servers. A second approach is to factor proximity into the decision at an earlier stage by limiting the candidate set of servers considered by the above algorithm (S) to only those that are nearby. The harder problem is deciding which of the potentially many servers are suitably close. One approach would be to select only those servers that are available on the same ISP as the client. A slightly more sophisticated approach would be to look at the map of autonoumous systems produced by BGP and select only those servers within some number of hops from the client as candidate servers. Finding the right balance between network proximity and server cache locality is a subject of ongoing research.

9.5 Summary

We have seen four client/server-based application protocols: the DNS protocol used by the domain naming system, SMTP used to exchange electronic mail, HTTP used to walk the World Wide Web, and SNMP used to query remote nodes for the sake of network management. We have also seen a collection of application-level protocols, including RTP and SIP, used to control and play streaming multimedia applications like vic and vat, as well as the emerging voice-over-IP service. Finally, we looked at emerging applications—including overlay, peer-to-peer, and content distribution networks—that blend application processing and packet forwarding in innovative ways.

Application protocols are a curious lot. In many ways, the traditional client/server applications are like another layer of transport protocol, except they have application-specific knowledge built into them. You could argue that they are just specialized transport protocols, and that transport protocols get layered on top of each other until producing the precise service needed by the application. Similarly, the overlay and peer-to-peer protocols can be viewed as providing an alternative routing infrastructure, but again, one that is tailored for a particular application's needs. The one sure lesson we draw from this observation is that designing application-level protocols is really no different than designing core network protocols, and that the more one understands about the latter, the better they will do designing the former.

OPEN ISSUE

New Network Architecture

It's difficult to put a finger on a specific open issue in the realm of application protocols—the entire field is open as new applications are invented every day, and the networking needs of these applications are, well, application-dependent. The real challenge to network designers is to recognize that what applications need from the network changes over time, and these changes drive the transport protocols we develop and the functionality we put into network routers.

Developing new transport protocols is a reasonably tractable problem. You may not be able to get the IETF to bless your transport protocol as an equal of TCP or UDP, but there's certainly nothing stopping you from designing the world's greatest multimedia application that comes bundled with a new end-to-end protocol that runs on top of UDP, much like happens with RTP.

On the other hand, pushing application-specific knowledge into the middle of the network—into the routers—is a much more difficult problem. This is because in order to effect a particular application, any new network service or functionality may need to be loaded into many, if not all, of the routers in the Internet. Overlay networks provide a way of introducing new functionality into the network without the cooperation of all (or even any) of the routers, but in the long run, we can expect the underlying network architecture will need to change to accommodate these overlays. We saw this issue with RON—how RON and BGP route selection interact with each other—and can expect it to be a general question as overlay networks become more prevalent.

One possibility is that an alternative *fixed* architecture does not evolve, but instead, the next network architecture will be highly adaptive. In the limit, rather than defining an infrastructure for carrying data packets, the network architecture might allow packets to carry both data and code (or possibly pointers to code) that tell the routers how they should process the packet. Such a network raises a host of issues, not the least of which is how to enforce security in a world where arbitrary applications can effectively program routers.

FURTHER READING

The seminal paper on application-layer protocols is that by Clark and Tennenhouse, which is cited by the designers of RTP as their guiding vision. The development of DNS is well described by Mockapetris and Dunlap. Although overlays and peer-to-peer networks are still an emerging area, the last six research papers provide a good place to start understanding the issues.

- Clark, D., and D. Tennenhouse. Architectural considerations for a new generation of protocols. *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.
- Mockapetris, P., and K. Dunlap. Development of the domain name system. *Proceedings of the SIGCOMM '88 Symposium*, pages 123–133, August 1988.
- Karger, D., E. Lehman, F.T. Leighton, R. Panigrahy, M. Levine, D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. *Proceedings of the ACM Symposium on Theory of Computing*, pages 654–663, 1997.
- Chu, Y., S. Rao, and H. Zhang. A case for end system multicast. *Proceedings of the ACM SIGMETRICS '00 Conference*, pages 1–12, June 2000.
- Andersen, D., H. Balakrishnan, F. Kaashoek, R. Morris. Resilient overlay networks. *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, October 2001.

- Rowstron, A. and P. Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 188–201, October 2001.
- Stoica, I., R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan. Chord: A peer-to-peer lookup service for Internet applications. *Proceedings of the ACM SIGCOMM Conference*, pages 149–160, August 2001.
- Ratnasamy, S., P. Francis, M. Handley, R. Karp, S. Shenker. A scalable content-addressable network. *Proceedings of ACM SIGCOMM '01*, pages 161–172, August 2001.

There are a wealth of papers on naming, as well as on the related issue of resource discovery (finding out what resources exist in the first place). General studies of naming can be found in Terry [Ter86], Comer and Peterson [CP89], Birrell et al. [BLNS82], Saltzer [Sal78], Shoch [Sho78], and Watson [Wat81]; attribute-based (descriptive) naming systems are described in Peterson [Pet88] and Bowman et al. [BPY90]; and resource discovery is the subject of Bowman et al. [BDMS94].

SMTP is originally defined in RFC 821 [Pos82], and of course, RFC 822 is RFC 822 [Cro82]. MIME is defined in a series of RFCs; the original specification was in RFC 1521 [BF93], and the most recent version is defined in RFC 2045 [FB96].

Version 1.0 of HTTP is specified in RFC 1945 [BLFF96], and the latest version (1.1) is defined in RFC 2068 [FGMBL97]. There are a wealth of papers written about Web performance, especially Web caching. A good example is a paper by Danzig on Web traffic and its implications on the effectiveness of caching [Dan98].

Network management is a sufficiently large and important field that the IETF devotes an entire area to it. There are well over 100 RFCs describing various aspects of SNMP and MIBs. The two key references, however, are Case et al. [CMRW93], which defines version 2 of SNMP (SNMPv2), and McCloghrie and Rose [MR91], which defines the second version of the mandatory MIB variables (MIB-II). Many of the other SNMP/MIB-related RFCs define extensions to the core set of MIB variables—for example, variables that are specific to a particular network technology or to a particular vendor's product. Perkins and Mcbinnis [PM97] provides a good introduction to SNMP and MIBS.

RTP is described in RFC 1889 [SCFJ 96], although much of the interesting detail is in Internet drafts that are yet to be published. McCanne and Jacobson [MJ95] describe vic, one of the applications to use RTP.

SIP is defined in RFC 3261 [SCJ$^+$02], which contains a helpful tutorial section as well as the detailed specification of the protocol.

The National Research Council report on the ossification of the Internet can be found in [NRC01], and a proposal to use overlay networks to introduce disruptive technology was made by Peterson et al. [PACR02]. The original case for overriding BGP routes is made by Savage et al. [SCH+99]. The idea of using DNS to load-balance a set of servers is described in RFC 1784 [Bri95]. The Cache Array Routing Protocol (CARP) is defined in an Internet Draft [CPVR97]. A comprehensive treatment of the issue of Web caching versus replicated servers can be found in Rabinovich and Spatscheck's book [RS02]. Wang et al. explore the design space for redirectors [WPP02].

Finally, we recommend the following live reference to help keep tabs on the rapid evolution of the Web:

- http://www.w3.org: World Wide Web Consortium

EXERCISES

1 ARP and DNS both depend on caches; ARP cache entry lifetimes are typically 10 minutes, while DNS cache is on the order of days. Justify this difference. What undesirable consequences might there be in having too long a DNS cache entry lifetime?

2 IPv6 simplifies ARP out of existence by allowing hardware addresses to be part of the IPv6 address. How does this complicate the job of DNS? How does this affect the problem of finding your local DNS server?

3 DNS servers also allow reverse lookup; given an IP address 128.112.169.4, it is reversed into a text string 4.169.112.128.in-addr.arpa and looked up using DNS PTR records (which form a hierarchy of domains analogous to that for the address domain hierarchy). Suppose you want to authenticate the sender of a packet based on its host name and are confident that the source IP *address* is genuine. Explain the insecurity in converting the source address to a name as above and then comparing this name to a given list of trusted hosts. Hint: Whose DNS servers would you be trusting?

4 What is the relationship between a domain name (e.g., cs.princeton.edu) and an IP subnet number (e.g., 192.12.69.0)? Do all hosts on the subnet have to be identified by the same name server? What about reverse lookup, as in the previous exercise?

5 Suppose a host elects to use a name server not within its organization for address resolution. When would this result in no more total traffic, for queries not found in any DNS cache, than with a local name server? When might this result in a better DNS cache hit rate and possibly less total traffic?

6 Figure 9.4 shows the hierarchy of name servers. How would you represent this hierarchy if one name server served multiple zones? In that setting, how does the name server hierarchy relate to the zone hierarchy? How do you deal with the fact that each zone may have multiple name servers?

7 Use the whois utility/service to find out who is in charge of your site, at least as far as the InterNIC is concerned. Look up your site both by DNS name and by IP network number; for the latter you may have to try an alternative whois server (e.g., whois -h whois.arin.net...). Try princeton.edu and cisco.com as well.

8 Many smaller organizations have their Web sites maintained by a third party. How could you use whois to find if this is the case, and if so, the identity of the third party?

9 One feature of the existing DNS .com hierarchy is that it is extremely "wide."

(a) Propose a more hierarchical reorganization of the .com hierarchy. What objections might you foresee to your proposal's adoption?

(b) What might be some of the consequences of having most DNS domain names contain four or more levels, versus the two of many existing names?

10 Suppose, in the other direction, we abandon any pretense at all of DNS hierarchy, and simply move all the .com entries to the root name server: www.cisco.com would become www.cisco, or perhaps just cisco. How would this affect root name server traffic in general? How would this affect such traffic for the specific case of resolving a name like cisco into a Web server address?

11 What DNS cache issues are involved in changing the IP address of, say, a Web server host name? How might these be minimized?

12 Take a suitable DNS-lookup utility (e.g., nslookup) and disable the recursive lookup feature (e.g., with set norecurse), so that when your utility sends a query to a DNS server, and that server is unable to fully answer the request from its own records, the server sends back the next DNS server in the lookup sequence rather than automatically forwarding the query to that next server.

Then carry out manually a name lookup such as that in Figure 9.5; try the host name www.cs.princeton.edu. List each intermediate name server contacted. You may also need to specify that queries are for NS records rather than the usual A records.

13 Discuss how you might rewrite SMTP or HTTP to make use of a hypothetical general-purpose request/reply protocol (perhaps something like CHAN RPC). Could an appropriate analog of persistent connections be moved from the application layer into such a transport protocol? What other application tasks might be moved into this protocol?

14 Most Telnet clients can be used to connect to port 25, the SMTP port, instead of to the Telnet port. Using such a tool, connect to an SMTP server and send yourself (or someone else, with permission) some forged email. Then examine the headers for evidence the message isn't genuine.

15 What features might be used by (or added to) SMTP and/or a mail daemon such as sendmail to provide some resistance to email forgeries as in the previous exercise?

16 Find out how SMTP hosts deal with unknown commands from the other side, and how in particular this mechanism allows for the evolution of the protocol (e.g., to "extended SMTP"). You can either read the RFC, or contact an SMTP server as in Exercise 14 and test its responses to nonexistent commands.

17 As presented in the text, SMTP involves the exchange of several small messages. In most cases, the server responses do not affect what the client sends subsequently. The client might thus implement *command pipelining:* sending multiple commands in a single message.

(a) For what SMTP commands *does* the client need to pay attention to the server's responses?

(b) Assume the server reads each client message with gets() or the equivalent, which reads in a string up to a <LF>. What would it have to do even to detect that a client had used command pipelining?

(c) Pipelining is nonetheless known to break with some servers; find out how a client can negotiate its use.

18 Find out what other features DNS MX records provide in addition to supplying an alias for a mail server; the latter could, after all, be provided by a DNS CNAME record. MX records are provided to support email; would an analogous WEB record be of use in supporting HTTP?

19 One of the central problems faced by a protocol such as MIME is the vast number of data formats available. Consult the MIME RFC to find out how MIME deals with new or system-specific image and text formats.

20 MIME supports multiple representations of the same content using the multipart/alternative syntax; for example, text could be sent as text/plain, text/richtext, and application/postscript. Why do you think plaintext is supposed to be the *first* format, even though implementations might find it easier to place plaintext after their native format?

21 Consult the MIME RFC to find out how base64 encoding handles binary data of a length not evenly divisible by three bytes.

22 In HTTP version 1.0, a server marked the end of a transfer by closing the connection. Explain why, in terms of the TCP layer, this was a problem for servers. Find out how HTTP version 1.1 avoids this. How might a general-purpose request/reply protocol address this?

23 Find out how to configure an HTTP server so as to eliminate the 404 not found message and have a default (and hopefully friendlier) message returned instead. Decide if such a feature is part of the protocol or part of an implementation, or is technically even permitted by the protocol. (Documentation for the apache HTTP server can be found at www.apache.org.)

24 Why does the HTTP GET command on page 612,

```
GET http://www.cs.princeton.edu/index.html HTTP/1.1
```

contain the name of the server being contacted? Wouldn't the server already know its name? Use Telnet, as in Exercise 14, to connect to port 80 of an HTTP server and find out what happens if you leave the host name out.

25 When an HTTP server initiates a close() at its end of a connection, it must then wait in TCP state FIN_WAIT_2 for the client to close the other end. What mechanism within the TCP protocol could help an HTTP server deal with noncooperative or poorly implemented clients that don't close from their end? If possible, find out about the programming interface for this mechanism, and indicate how an HTTP server might apply it.

26 The POP3 Post Office Protocol only allows a client to retrieve email, using a password for authentication. Traditionally, to *send* email a client would simply send it to its server and expect that it be relayed.

(a) Explain why email servers often no longer permit such relaying from arbitrary clients.

(b) Propose an SMTP option for remote client authentication.

(c) Find out what existing methods are available for addressing this issue.

27 Suppose a very large Web site wants a mechanism by which clients access whichever of multiple HTTP servers is "closest" by some suitable measure.

(a) Discuss developing a mechanism within HTTP for doing this.

(b) Discuss developing a mechanism within DNS for doing this.

Compare the two. Can either approach be made to work without upgrading the browser?

28 Find out if there is available to you an SNMP node that will answer queries you send it. If so, locate some SNMP utilities (e.g., the ucd-snmp suite) and try the following:

(a) Fetch the entire system group, using something like

```
snmpwalk  nodename  public system
```

Also try the above with 1 in place of system.

(b) Manually walk through the system group, using multiple SNMP GET-NEXT operations (e.g., using snmpgetnext or equivalent), retrieving one entry at a time.

29 Using the SNMP device and utilities of the previous exercise, fetch the tcp group (numerically group 6) or some other group. Then do something to make some of the group's counters change, and fetch the group again to show the change. Try to do this in such a way that you can be sure your actions were the cause of the change recorded.

30 What information provided by SNMP might be useful to someone planning the IP spoofing attack of Exercise 19 in Chapter 5? What other SNMP information might be considered sensitive?

31 Try to find situations where an RTP application might reasonably do the following:

- Send multiple packets at essentially the same time that need different timestamps.
- Send packets at different times that need the same timestamp.

Argue, in consequence, that RTP timestamps must, in at least some cases, be provided (at least indirectly) by the application. Hint: Think of cases where the sending rate and playback rate might not match.

32 Having the timestamp clock count time in units of one frame time or one voice sample time would be the minimum resolution to ensure accurate playback. But the time unit is usually considerably smaller; what is the purpose of this?

33 Suppose we want returning RTCP reports from receivers to amount to no more than 5% of the outgoing primary RTP stream. If each report is 84 bytes, and the RTP traffic is 20 KBps, and there are 1000 recipients, how often do individual receivers get to report? What if there are 10,000 recipients?

34 RFC 1889 specifies that the time interval between receiver RTCP reports include a randomization factor to avoid having all the receivers send at the same time. If all the receivers sent in the same 5% subinterval of their reply time interval, the arriving upstream RTCP traffic would rival the downstream RTP traffic.

(a) Video receivers might reasonably wait to send their reports until the higher-priority task of processing and displaying one frame is completed; this might mean their RTCP transmissions were synchronized on frame boundaries. Is this likely to be a serious concern?

(b) With 10 receivers, what is the probability of their all sending in one particular 5% subinterval?

(c) With 10 receivers, what is the probability half will send in one particular 5% subinterval? Multiply this by 20 for an estimate of the probability half will all send in the same arbitrary 5% subinterval. Hint: How many ways can we choose 5 receivers out of 10?

35 What might a server actually do with the packet-loss-rate data and jitter data in receiver reports?

36 Video applications typically run over UDP rather than TCP because they cannot tolerate retransmission delays. However, this means video applications are not constrained by TCP's congestion-control algorithm. What impact does this have on TCP traffic? Be specific about the consequences.

Fortunately, these video applications often use RTP, which results in RTCP "receiver reports" being sent from the sink back to the source. These reports are sent periodically (e.g., once a second) and include the percentage of packets successfully received in the last reporting period. Describe how the source might use this information to adjust its rate in a TCP-compatible way.

37 Suppose some receivers in a large conference can receive data at a significantly higher bandwidth than others. What sorts of things might be implemented to address this? Hint: Consider both the session announcement protocol (SAP) and the possibility of utilizing third-party "mixers."

38 Propose a mechanism for deciding when to report an RTP packet as lost. How does your mechanism compare with the TCP adaptive retransmission mechanisms of Section 5.2.6?

39 How might you encode audio (or video) data in two packets so that if one packet is lost, then the resolution is simply reduced to what would be expected with half the bandwidth? Explain why this is much more difficult if a JPEG-type encoding is used.

40 Explain the relationship between uniform resource locators (URLs) and uniform resource identifiers (URIs). Given an example of a URI that is *not* a URL.

41 What problem would a DNS-based redirection mechanism encounter if it wants to select an appropriate server based on current load information?