8 Network Security

It is true greatness to have in one the frailty of a man and the security of a god.
–Seneca

Computer networks are typically a shared resource used by many applications representing different interests. The Internet is particularly widely shared, being used by competing businesses, mutually antagonistic governments, and opportunistic criminals. Unless security measures are taken, a network conversation or a distributed application may be compromised by an adversary.

Consider some threats to secure use of, for example, the World Wide Web. Suppose you are a customer using a credit card to order an item from a website. An obvious threat is that an adversary would eavesdrop on your network communica-

PROBLEM: SECURITY ATTACKS

tion, reading your messages to obtain your credit card information. How might that eavesdropping be accomplished? It is trivial on a broadcast network such as an Ethernet, where any node can be configured to receive all the message traffic on that network. Wireless communication can be monitored without any physical connection. More elaborate approaches include wiretapping and planting spy software on any of the chain of nodes involved. Only in the most extreme cases (e.g., national security) are serious measures taken to prevent such monitoring, and the Internet is not one of those cases. It is possible

Computer Networks: A Systems Approach. DOI: 10.1016/B978-0-12-385059-1.00008-9

and practical, however, to encrypt messages so as to prevent an adversary from understanding the message contents. A protocol that does so is said to provide *confidentiality*. Taking the concept a step farther, concealing the quantity or destination of communication is called *traffic confidentiality*—because merely knowing how much communication is going where can be useful to an adversary in some situations.

Even with confidentiality there still remains threats for the website customer. An adversary who can't read the contents of your encrypted message might still be able to change a few bits in it, resulting in a valid order for, say, a completely different item or perhaps 1000 units of the item. There are techniques to detect, if not prevent, such tampering. A protocol that detects such message tampering provides *data integrity*. The adversary could alternatively transmit an extra copy of your message in a *replay attack*. To the website, it would appear as though you had simply ordered another of the same item you ordered the first time. A protocol that detects replays provides *originality*. Originality would not, however, preclude the adversary intercepting your order, waiting a while, then transmitting it—in effect, delaying your order. The adversary could thereby arrange for the item to arrive on your doorstep while you are away on vacation, when it can be easily snatched. A protocol that detects such delaying tactics is said to provide *timeliness*. Data integrity, originality, and timeliness are considered aspects of the more general property of *integrity*.

Another threat to the customer is unknowingly being directed to a false website. This can result from a Domain Name System (DNS) attack, in which false information is entered in a DNS server or the name service cache of the customer's computer. This leads to translating a correct URL into an incorrect IP address—the address of a false website. A protocol that ensures that you really are talking to whom you think you're talking is said to provide *authentication*. Authentication entails integrity, since it is meaningless to say that a message came from a certain participant if it is no longer the same message.

The owner of the website can be attacked as well. Some websites have been defaced; the files that make up the website content have been remotely accessed and modified without authorization. That is an issue of *access control*: enforcing the rules regarding who is allowed to do what. Websites have also been subject to denial of service (DoS) attacks, during which would-be customers are unable to access the website because it is being overwhelmed by bogus requests. Ensuring a degree of access is called *availability*.

In addition to these issues, the Internet has notably been used as a means for deploying malicious code that exploits vulnerabilities in end systems. *Worms*, pieces of self-replicating code that spread over networks, have been known for several decades and continue to cause problems, as do their relatives, *viruses*, which are spread by the transmission of infected files. Infected machines can then be arranged

into *botnets*, which can be used to inflict further harm, such as launching DoS attacks.

Although the Internet was designed with the redundancy to survive problems such as the disruption of a link or router, it was not originally designed to provide the kind of security we have been discussing. Internet security mechanisms have essentially been patches. If a comprehensive redesign of the Internet were to take place, integrating security would likely be the foremost driving factor. That possibility makes this chapter all the more pertinent.

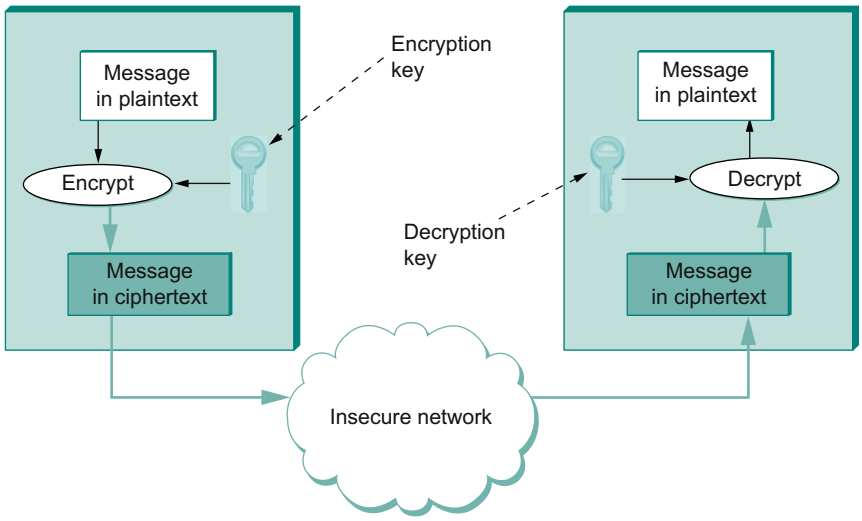
There are many tools today for securing networked systems, ranging from various forms of cryptography to specialized devices such as firewalls. This chapter will provide an introduction to these tools with a particular focus on the use of cryptographic methods to improve network security. Improving the security of networks continues to be a field of rapid change and considerable research effort.

8.1 CRYPTOGRAPHIC BUILDING BLOCKS

We introduce the concepts of cryptography-based security step by step. The first step is the cryptographic algorithms—ciphers and cryptographic hashes—that are introduced in this section. They are not a solution in themselves, but rather building blocks from which a solution can be built. Cryptographic algorithms are parameterized by *keys*, and Section 8.2 addresses the problem of distributing the keys. In the next step (Section 8.3), we describe how to incorporate the cryptographic building blocks into protocols that provide secure communication between participants who possess the correct keys. Finally, Section 8.4 examines several complete security protocols and systems in current use.

8.1.1 Principles of Ciphers

Encryption transforms a message in such a way that it becomes unintelligible to any party that does not have the secret of how to reverse the transformation. The sender applies an *encryption* function to the original *plaintext* message, resulting in a *ciphertext* message that is sent over the network, as shown in Figure 8.1. The receiver applies a secret *decryption* function—the inverse of the encryption function—to recover the original plaintext. The ciphertext transmitted across the network is unintelligible to any eavesdropper, assuming the eavesdropper doesn't know the



■ **FIGURE 8.1** Symmetric-key encryption and decryption.

decryption function. The transformation represented by an encryption function and its corresponding decryption function is called a *cipher*.

Cryptographers have been led to the principle, first stated in 1883, that encryption and decryption functions should be parameterized by a *key*, and furthermore that the functions should be considered public knowledge—only the key need be secret. Thus, the ciphertext produced for a given plaintext message depends on both the encryption function and the key. One reason for this principle is that if you depend on the cipher being kept secret, then you have to retire the cipher (not just the keys) when you believe it is no longer secret. This means potentially frequent changes of cipher, which is problematic since it takes a lot of work to develop a new cipher. Also, one of the best ways to know that a cipher is secure is to use it for a long time—if no one breaks it, it's probably secure. (Fortunately, there are plenty of people who will try to break ciphers and who will let it be widely known when they have succeeded, so no news is generally good news.) Thus, there is considerable cost and risk in deploying a new cipher. Finally, parameterizing a cipher with keys provides us with what is in effect a very large family of ciphers; by switching keys, we essentially switch ciphers, thereby limiting the amount of data that a *cryptanalyst* (code-breaker) can use to try to break our key/cipher and the amount she can read if she succeeds.

The basic requirement for an encryption algorithm is that it turn plaintext into ciphertext in such a way that only the intended recipient—the holder of the decryption key—can recover the plaintext. What this means is that encrypted messages cannot be read by people who do not hold the key.

It is important to realize that when a potential attacker receives a piece of ciphertext, he may have more information at his disposal than just the ciphertext itself. For example, he may know that the plaintext was written in English, which means that the letter *e* occurs more often in the plaintext that any other letter; the frequency of many other letters and common letter combinations can also be predicted. This information can greatly simplify the task of finding the key. Similarly, he may know something about the likely contents of the message; for example, the word "login" is likely to occur at the start of a remote login session. This may enable a *known plaintext* attack, which has a much higher chance of success than a *ciphertext only* attack. Even better is a *chosen plaintext* attack, which may be enabled by feeding some information to the sender that you know the sender is likely to transmit—such things have happened in wartime, for example.

The best cryptographic algorithms, therefore, can prevent the attacker from deducing the key even when the individual knows both the plaintext and the ciphertext. This leaves the attacker with no choice but to try all the possible keys—exhaustive, "brute force" search. If keys have n bits, then there are 2^n possible values for a key (each of the n bits could be either a zero or a one). An attacker could be so lucky as to try the correct value immediately, or so unlucky as to try every incorrect value before finally trying the correct value of the key, having tried all 2^n possible values; the average number of guesses to discover the correct value is halfway between those extremes, $2^n/2$. This can be made computationally impractical by choosing a sufficiently large key space and by making the operation of checking a key reasonably costly. What makes this difficult is that computing speeds keep increasing, making formerly infeasible computations feasible. Furthermore, although we are concentrating on the security of data as it moves through the network—that is, the data is sometimes vulnerable for only a short period of time—in general, security people have to consider the vulnerability of data that needs to be stored in archives for tens of years. This argues for a generously large key size. On the other hand, larger keys make encryption and decryption slower.

Most ciphers are *block ciphers*; they are defined to take as input a plaintext block of a certain fixed size, typically 64 to 128 bits. Using a block cipher to encrypt each block independently—known as *electronic codebook (ECB) mode* encryption—has the weakness that a given plaintext block value will always result in the same ciphertext block. Hence, recurring block values in the plaintext are recognizable as such in the ciphertext, making it much easier for a cryptanalyst to break the cipher.

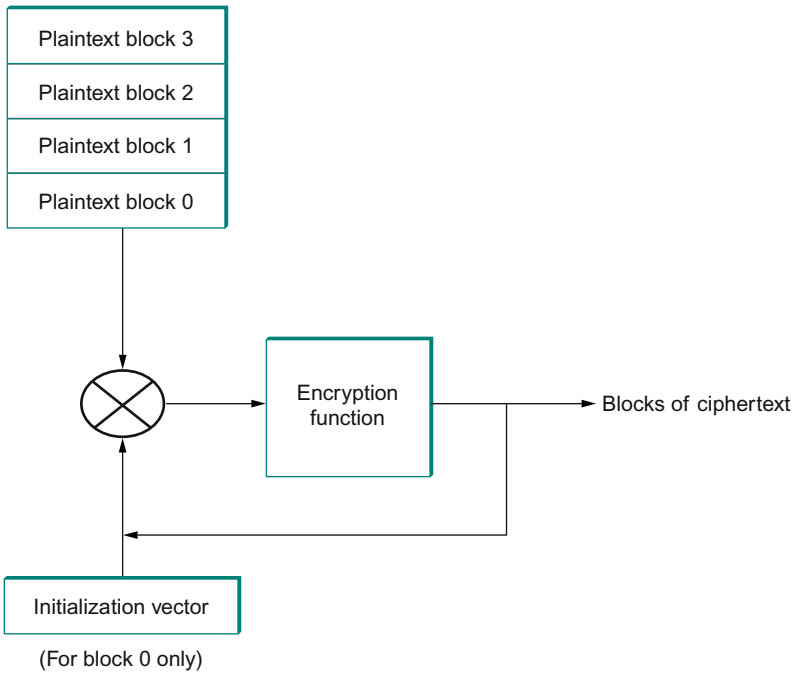
To prevent this, block ciphers are always augmented to make the ciphertext for a block vary depending on context. Ways in which a block cipher may be augmented are called *modes of operation*. A common mode of operation is *cipher block chaining* (CBC), in which each plaintext block is XORed with the previous block's ciphertext before being encrypted. The result is that each block's ciphertext depends in part on the preceding blocks (i.e., on its context). Since the first plaintext block has no preceding block, it is XORed with a random number. That random number, called an *initialization vector* (IV), is included with the series of ciphertext blocks so that the first ciphertext block can be decrypted. This mode is illustrated in Figure 8.2. Another mode of operation is *counter mode*, in which successive values of a counter (e.g., 1, 2, 3, ...) are incorporated into the encryption of successive blocks of plaintext.

8.1.2 Symmetric-Key Ciphers

In a symmetric-key cipher, both participants[1] in a communication share the same key. In other words, if a message is encrypted using a particular key, the same key is required for decrypting the message. If the cipher illustrated in Figure 8.1 were a symmetric-key cipher, then the encryption and decryption keys would be identical. Symmetric-key ciphers are also known as secret-key ciphers since the shared key must be known only to the participants. (We'll take a look at the alternative, public-key ciphers, shortly.)

The U.S. National Institute of Standards and Technology (NIST) has issued standards for a series of symmetric-key ciphers. *Data Encryption Standard* (DES) was the first, and it has stood the test of time in that no cryptanalytic attack better than brute force search has been discovered.

[1]We use the term *participant* for the parties involved in a secure communication since that is the term we have been using throughout the book to identify the two endpoints of a channel. In the security world, they are typically called *principals*.



■ **FIGURE 8.2** Cipher block chaining (CBC).

Brute force search, however, has gotten faster. DES's keys (56 independent bits) are now too small given current processor speeds. DES keys have 56 independent bits (although they have 64 bits in total; the last bit of every byte is a parity bit). As noted above, you would, on average, have to search half of the space of 2^{56} possible keys to find the right one, giving $2^{55} = 3.6 \times 10^{16}$ keys. That may sound like a lot, but such a search is highly parallelizable, so it's possible to throw as many computers at the task as you can get your hands on—and these days it's easy to lay your hands on thousands of computers (Amazon.com will rent them to you for a few cents an hour, for example). By the late 1990s, it was already possible to recover a DES key after a few hours. Consequently, NIST updated the DES standard in 1999 to indicate that DES should only be used for legacy systems.

NIST also standardized the cipher *Triple DES* (3DES), which leverages the cryptanalysis resistance of DES while in effect increasing the key size. A 3DES key has 168 $(= 3 \times 56)$ independent bits, and is used as three DES

keys; let's call them DES-key1, DES-key2, and DES-key3. 3DES encryption of a block is performed by first DES encrypting the block using DES-key1, then DES *de*crypting the result using DES-key2, and finally DES encrypting that result using DES-key3. Decryption involves decrypting using DES-key3, then encrypting using DES-key2, then decrypting using DES-key1.[2]

Although 3DES solves DES's key-length problem, it inherits some other shortcomings. Software implementations of DES/3DES are slow because it was originally designed, by IBM, for implementation in hardware. Also, DES/3DES uses a 64-bit block size; a larger block size is more efficient and more secure.

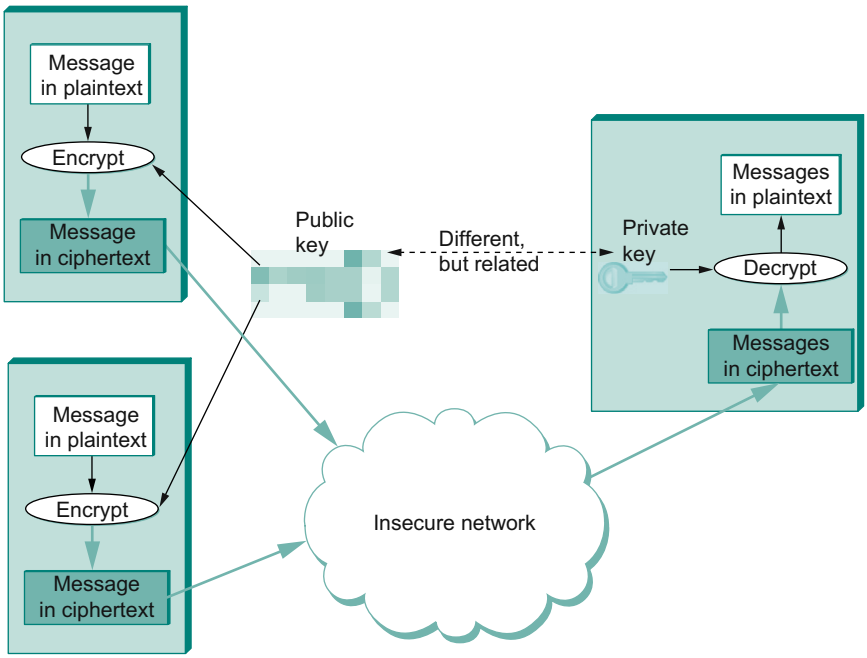
3DES is being superseded by the *Advanced Encryption Standard* (AES) standard issued by NIST in 2001. The cipher selected to become that standard (with a few minor modifications) was originally named Rijndael (pronounced roughly like "Rhine dahl") based on the names of its inventors, Daemen and Rijmen. AES supports key lengths of 128, 192, or 256 bits, and the block length is 128 bits. AES permits fast implementations in both software and hardware. It doesn't require much memory, which makes it suitable for small mobile devices. AES has some mathematically proven security properties and, as of the time of writing, has not suffered from any significant successful attacks.[3]

8.1.3 Public-Key Ciphers

An alternative to symmetric-key ciphers is asymmetric, or public-key, ciphers. Instead of a single key shared by two participants, a public-key cipher uses a pair of related keys, one for encryption and a different one for decryption. The pair of keys is "owned" by just one participant. The owner keeps the decryption key secret so that only the owner can decrypt messages; that key is called the *private key*. The owner makes the encryption key public, so that anyone can encrypt messages for the owner; that

[2]The reason 3DES encryption uses DES *de*cryption with DES-key2 is to interoperate with legacy DES systems. If a legacy DES system uses a single key, then a 3DES system can perform the same encryption function by using that key for each of DES-key1, DES-key2, and DES-key3; in the first two steps, we encrypt and then decrypt with the same key, producing the original plaintext, which we then encrypt again.

[3]Since anything that can recover the plaintext with less computational effort than sheer brute force is technically classified as an attack, there are some forms of attack on AES that have been published. While they do somewhat better than brute force, they remain computationally very expensive.



■ **FIGURE 8.3** Public-key encryption.

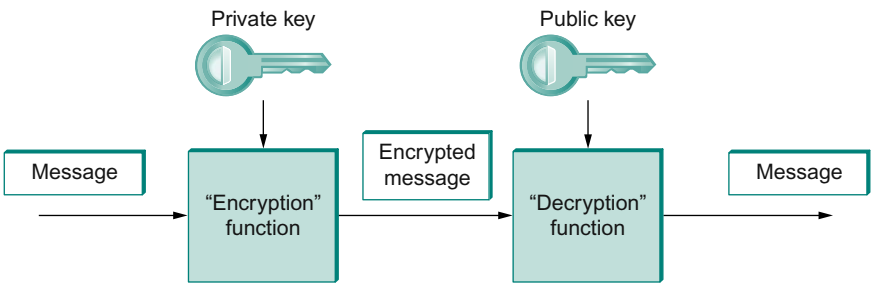
key is called the *public key*. Obviously, for such a scheme to work, it must not be possible to deduce the private key from the public key. Consequently, any participant can get the public key and send an encrypted message to the owner of the keys, and only the owner has the private key necessary to decrypt it. This scenario is depicted in Figure 8.3.

Because it is somewhat unintuitive, we emphasize that the public encryption key is useless for decrypting a message—you couldn't even decrypt a message that you yourself had just encrypted unless you had the private decryption key. If we think of keys as defining a communication channel between participants, then another difference between public-key and symmetric-key ciphers is the topology of the channels. A key for a symmetric-key cipher provides a channel that is two-way between two participants—each participant holds the same (symmetric) key that either one can use to encrypt or decrypt messages in either direction. A public/private key pair, in contrast, provides a channel that is one way and many to one from everyone who has the public key to the (unique) owner of the private key, as illustrated in Figure 8.3.

An important additional property of public-key ciphers is that the private "decryption" key can be used with the encryption algorithm to encrypt messages so that they can only be decrypted using the public "encryption" key. This property clearly wouldn't be useful for confidentiality since anyone with the public key could decrypt such a message. (Indeed, for two-way confidentiality between two participants, each participant needs its own pair of keys, and each encrypts messages using the other's public key.) This property is, however, useful for authentication since it tells the receiver of such a message that it could only have been created by the owner of the keys (subject to certain assumptions that we will get into later). This is illustrated in Figure 8.4. It should be clear from the figure that anyone with the public key can decrypt the encrypted message, and, assuming that the result of the decryption matches the expected result, it can be concluded that the private key must have been used to perform the encryption. Exactly how this operation is used to provide authentication is the topic of Section 8.3. As we will see, public-key ciphers are used primarily for authentication and to confidentially distribute symmetric keys, leaving the rest of confidentiality to symmetric-key ciphers.

A bit of interesting history: The concept of public-key ciphers was first published in 1976 by Diffie and Hellman. Subsequently, however, documents have come to light proving that Britain's Communications-Electronics Security Group had discovered public-key ciphers by 1970, and the U.S. National Security Agency (NSA) claims to have discovered them in the mid-1960s.

The best-known public-key cipher is RSA, named after its inventors: Rivest, Shamir, and Adleman. RSA relies on the high computational cost



■ **FIGURE 8.4** Authentication using public keys.

of factoring large numbers. The problem of finding an efficient way to factor numbers is one that mathematicians have worked on unsuccessfully since long before RSA appeared in 1978, and RSA's subsequent resistance to cryptanalysis has further bolstered confidence in its security. Unfortunately, RSA needs relatively large keys, at least 1024 bits, to be secure. This is larger than keys for symmetric-key ciphers because it is faster to break an RSA private key by factoring the large number on which the pair of keys is based than by exhaustively searching the key space.

Another public-key cipher is ElGamal. Like RSA, it relies on a mathematical problem, the discrete logarithm problem, for which no efficient solution has been found, and requires keys of at least 1024 bits. There is a variation of the discrete logarithm problem, arising when the input is an elliptic curve, that is thought to be even more difficult to compute; cryptographic schemes based on this problem are referred to as *elliptic curve cryptography*.

Public-key ciphers are, unfortunately, several orders of magnitude slower than symmetric-key ciphers. Consequently, symmetric-key ciphers are used for the vast majority of encryption, while public-key ciphers are reserved for use in authentication (Section 8.1.4) and session key establishment (Section 8.2).

8.1.4 Authenticators

Encryption alone does not provide data integrity. For example, just randomly modifying a ciphertext message could turn it into something that decrypts into valid-looking plaintext, in which case the tampering would be undetectable by the receiver. Nor does encryption alone provide authentication. It is not much use to say that a message came from a certain participant if the contents of the message have been modified after that participant created it. In a sense, integrity and authentication are fundamentally inseparable.

An *authenticator* is a value, to be included in a transmitted message, that can be used to verify simultaneously the authenticity and the data integrity of a message. We will see how authenticators can be used in protocols to Section 8.3. For now, we focus on the algorithms that produce authenticators.

You should recall that in Section 2.4.3 we looked at checksums and cyclic redundancy checks (CRCs)—added pieces of information sent with

the original message—as ways of detecting when a message has been inadvertently modified by bit errors. A similar concept applies to authenticators, with the added challenge that the corruption of the message is likely to be deliberately performed by someone who wants the corruption to go undetected. To support authentication, an authenticator includes some proof that whoever created the authenticator knows a secret that is known only to the alleged sender of the message; for example, the secret could be a key, and the proof could be some value encrypted using the key. There is a mutual dependency between the form of the redundant information and the form of the proof of secret knowledge. We discuss several workable combinations.

We initially assume that the original message need not be confidential—that a transmitted message will consist of the plaintext of the original message plus an authenticator. Later we will consider the case where confidentiality is desired.

One kind of authenticator combines encryption and a *cryptographic hash function*. Cryptographic hash algorithms are treated as public knowledge, as with cipher algorithms. A cryptographic hash function (also known as a *cryptographic checksum*) is a function that outputs sufficient redundant information about a message to expose any tampering. Just as a checksum or CRC exposes bit errors introduced by noisy links, a cryptographic checksum is designed to expose deliberate corruption of messages by an adversary. The value it outputs is called a *message digest* and, like an ordinary checksum, is appended to the message. All the message digests produced by a given hash have the same number of bits regardless of the length of the original message. Since the space of possible input messages is larger than the space of possible message digests, there will be different input messages that produce the same message digest, like collisions in a hash table.

An authenticator can be created by encrypting the message digest. The receiver computes a digest of the plaintext part of the message and compares that to the decrypted message digest. If they are equal, then the receiver would conclude that the message is indeed from its alleged sender (since it would have to have been encrypted with the right key) and has not been tampered with. No adversary could get away with sending a bogus message with a matching bogus digest because she would not have the key to encrypt the bogus digest correctly. An adversary could, however, obtain the plaintext original message and its encrypted digest

by eavesdropping. The adversary could then (since the hash function is public knowledge) compute the digest of the original message and generate alternative messages looking for one with the same message digest. If she finds one, she could undetectably send the new message with the old authenticator. Therefore, security requires that the hash function have the *one-way* property: It must be computationally infeasible for an adversary to find any plaintext message that has the same digest as the original.

For a hash function to meet this requirement, its outputs must be fairly randomly distributed. For example, if digests are 128 bits long and randomly distributed, then you would need to try 2^{127} messages, on average, before finding a second message whose digest matches that of a given message. If the outputs are not randomly distributed—that is, if some outputs are much more likely than others—then for some messages you could find another message with the same digest much more easily than this, which would reduce the security of the algorithm. If you were instead just trying to find any *collision*—any two messages that produce the same digest—then you would need to compute the digests of only 2^{64} messages, on average. This surprising fact is the basis of the "birthday attack"—see the exercises for more details.

There are several common cryptographic hash algorithms, including Message Digest 5 (MD5) and Secure Hash Algorithm 1 (SHA-1). MD5 outputs a 128-bit digest, and SHA-1 outputs a 160-bit digest. Weaknesses of MD5 have been known for some time, which led to recommendations to shift from MD5 to SHA-1. More recently, researchers have discovered techniques that find SHA-1 collisions somewhat more efficiently than brute force, but they are not yet computationally feasible. Although *collision attacks* (attacks based on finding any collision) are not as great a risk as *preimage attacks* (attacks based on finding a second message that collides with a given first message), these are nonetheless serious weaknesses. NIST recommended phasing out SHA-1 by 2010, in favor of four variants of SHA that are collectively known as SHA-2. There is an ongoing competition to devise a new hash known as SHA-3.

When generating an encrypted message digest, the digest encryption could use either a symmetric-key cipher or a public-key cipher. If a public-key cipher is used, the digest would be encrypted using the sender's private key (the one we normally think of as being used for decryption), and the receiver—or anyone else—could decrypt the digest using the sender's public key.

A digest encrypted with a public key algorithm but using the private key is called a *digital signature* because it provides nonrepudiation like a written signature. The receiver of a message with a digital signature can prove to any third party that the sender really sent that message, because the third party can use the sender's public key to check for herself. (Symmetric-key encryption of a digest does not have this property because only the two participants know the key; furthermore, since both participants know the key, the alleged receiver could have created the message herself.) Any public-key cipher can be used for digital signatures. *Digital Signature Standard* (DSS) is a digital signature format that has been standardized by NIST. DSS signatures may use any one of three public-key ciphers, one based on RSA, another on ElGamal, and a third called the *Elliptic Curve Digital Signature Algorithm*.

Another kind of authenticator is similar, but instead of encrypting a hash it uses a hash-like function that takes a secret value (known to only the sender and the receiver) as a parameter, as illustrated in Figure 8.5. Such a function outputs an authenticator called a *message authentication code* (MAC). The sender appends the MAC to her plaintext message. The receiver recomputes the MAC using the plaintext and the secret value and compares that recomputed MAC to the received MAC.

A common variation on MACs is to apply a cryptographic hash (such as MD5 or SHA-1) to the concatenation of the plaintext message and

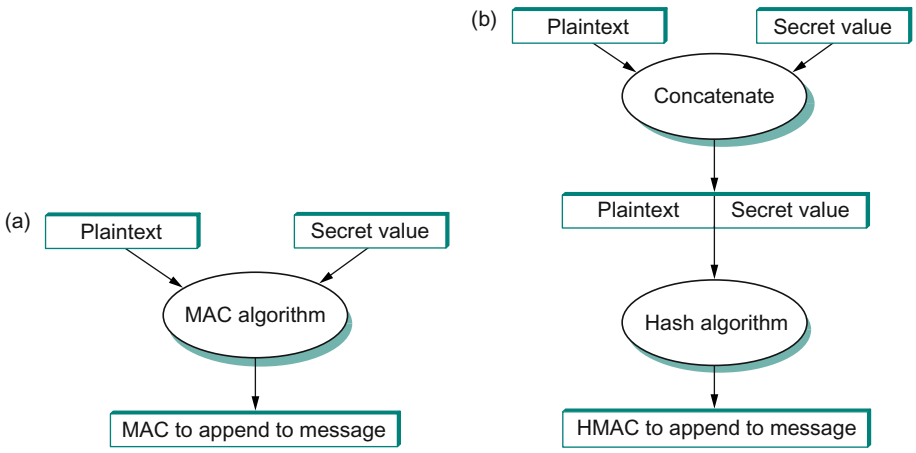


FIGURE 8.5 Computing a MAC (a) versus computing an HMAC (b).

the secret value, as illustrated in Figure 8.5. The resulting digest is called a *hashed message authentication code* (HMAC) since it is essentially a MAC. The HMAC, but not the secret value, is appended to the plaintext message. Only a receiver who knows the secret value can compute the correct HMAC to compare with the received HMAC. If it weren't for the one-way property of the hash, an adversary might be able to find the input that generated the HMAC and compare it to the plaintext message to determine the secret value.

Up to this point, we have been assuming that the message wasn't confidential, so the original message could be transmitted as plaintext. To add confidentiality to a message with an authenticator, it suffices to encrypt the concatenation of the entire message including its authenticator—the MAC, HMAC, or encrypted digest. Remember that, in practice, confidentiality is implemented using symmetric-key ciphers because they are so much faster than public-key ciphers. Furthermore, it costs little to include the authenticator in the encryption, and it increases security. A common simplification is to encrypt the message with its (raw) digest, such that the digest is only encrypted once; in this case, the entire ciphertext message is considered to be an authenticator.

Although authenticators may seem to solve the authentication problem, we will see in Section 8.3 that they are only the foundation of a solution. First, however, we address the issue of how participants obtain keys in the first place.

8.2 KEY PREDISTRIBUTION

To use ciphers and authenticators, the communicating participants need to know what keys to use. In the case of a symmetric-key cipher, how does a pair of participants obtain the key they share? In the case of a public-key cipher, how do participants know what public key belongs to a certain participant? The answer differs depending on whether the keys are short-lived *session keys* or longer-lived *predistributed keys*.

A session key is a key used to secure a single, relatively short episode of communication: a session. Each distinct session between a pair of participants uses a new session key, which is always a symmetric key for speed. The participants determine what session key to use by means of a protocol—a session key establishment protocol. A session key establishment protocol needs its own security (so that, for example, an adversary

cannot learn the new session key); that security is based on the longer-lived predistributed keys.

There are several motivations for this division of labor between session keys and predistributed keys:

- Limiting the amount of time a key is used results in less time for computationally intensive attacks, less ciphertext for cryptanalysis, and less information exposed should the key be broken.
- Predistribution of symmetric keys is problematic.
- Public key ciphers are generally superior for authentication and session key establishment but too slow to use for encrypting entire messages for confidentiality.

This section explains how predistributed keys are distributed, and Section 8.3 will explain how session keys are then established. We henceforth use "Alice" and "Bob" to designate participants, as is common in the cryptography literature. Bear in mind that although we tend to refer to participants in anthropomorphic terms, we are more frequently concerned with the communication between software or hardware entities such as clients and servers that often have no direct relationship with any particular person.

8.2.1 Predistribution of Public Keys

The algorithms to generate a matched pair of public and private keys are publicly known, and software that does it is widely available. So, if Alice wanted to use a public-key cipher, she could generate her own pair of public and private keys, keep the private key hidden, and publicize the public key. But, how can she publicize her public key—assert that it belongs to her—in such a way that other participants can be sure it really belongs to her? Not via email or Web, because an adversary could forge an equally plausible claim that key x belongs to Alice when x really belongs to the adversary.

A complete scheme for certifying bindings between public keys and identities—what key belongs to whom—is called a *Public Key Infrastructure* (PKI). A PKI starts with the ability to verify identities and bind them to keys out of band. By "out of band," we mean something outside the network and the computers that comprise it, such as in the following scenarios. If Alice and Bob are individuals who know each other, then

they could get together in the same room and Alice could give her public key to Bob directly, perhaps on a business card. If Bob is an organization, Alice the individual could present conventional identification, perhaps involving a photograph or fingerprints. If Alice and Bob are computers owned by the same company, then a system administrator could configure Bob with Alice's public key.

Establishing keys out of band doesn't sound like it would scale well, but it suffices to bootstrap a PKI. Bob's knowledge that Alice's key is x can be widely, scalably disseminated using a combination of digital signatures and a concept of trust. For example, suppose that you have received Bob's public key out of band and that you know enough about Bob to trust him on matters of keys and identities. Then Bob could send you a message asserting that Alice's key is x and—since you already know Bob's public key—you could authenticate the message as having come from Bob. (Remember that to digitally sign the statement Bob would append a cryptographic hash of it that has been encrypted using his private key.) Since you trust Bob to tell the truth, you would now know that Alice's key is x, even though you had never met her or exchanged a single message with her. Using digital signatures, Bob wouldn't even have to send you a message; he could simply create and publish a digitally signed statement that Alice's key is x. Such a digitally signed statement of a public key binding is called a *public key certificate*, or simply a certificate. Bob could send Alice a copy of the certificate, or post it on a website. If and when someone needs to verify Alice's public key, they could do so by getting a copy of the certificate, perhaps directly from Alice—as long as they trust Bob and know his public key. You can see how starting from a very small number of keys (in this case, just Bob's) you could build up a large set of trusted keys over time. Bob in this case is playing the role often referred to as a *certification authority* (CA), and much of today's Internet security depends on CAs. VeriSign® is one well-known commercial CA. We return to this topic below.

One of the major standards for certificates is known as X.509. This standard leaves a lot of details open, but specifies a basic structure. A certificate clearly must include:

- The identity of the entity being certified
- The public key of the entity being certified
- The identity of the signer

- The digital signature
- A digital signature algorithm identifier (which cryptographic hash and which cipher)

An optional component is an expiration time for the certificate. We will see a particular use of this feature below.

Since a certificate creates a binding between an identity and a public key, we should look more closely at what we mean by "identity." For example, a certificate that says, "This public key belongs to John Smith," may not be terribly useful if you can't tell which of the thousands of John Smiths is being identified. Thus, certificates must use a well-defined name space for the identities being certified; for example, certificates are often issued for email addresses and DNS domains.

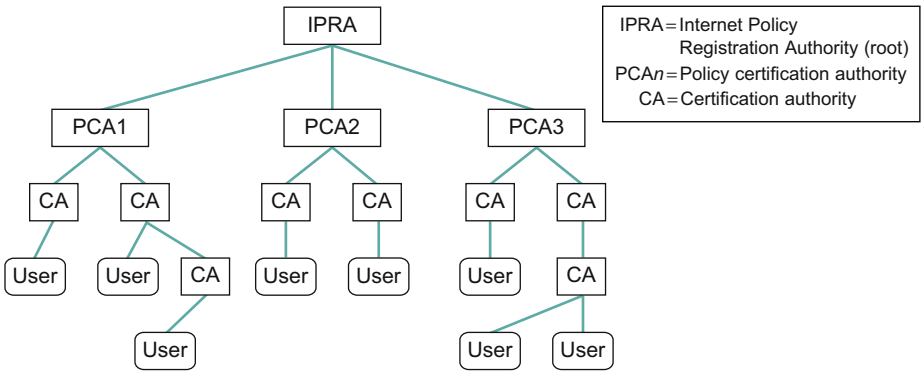
There are different ways a PKI could formalize the notion of trust. We discuss the two main approaches.

Certification Authorities

In this model of trust, trust is binary; you either trust someone completely or not at all. Together with certificates, this allows the building of *chains of trust*. If X certifies that a certain public key belongs to Y, and then Y goes on to certify that another public key belongs to Z, then there exists a chain of certificates from X to Z, even though X and Z may have never met. If you know X's key—and you trust X and Y—then you can believe the certificate that gives Z's key. In other words, all you need is a chain of certificates, all signed by entities you trust, as long as it leads back to an entity whose key you already know.

A *certification authority* or *certificate authority* (CA) is an entity claimed (by someone) to be trustworthy for verifying identities and issuing public key certificates. There are commercial CAs, governmental CAs, and even free CAs. To use a CA, you must know its own key. You can learn that CA's key, however, if you can obtain a chain of CA-signed certificates that starts with a CA whose key you already know. Then you can believe any certificate signed by that new CA.

A common way to build such chains is to arrange them in a tree-structured hierarchy, as shown in Figure 8.6. If everyone has the public key of the root CA, then any participant can provide a chain of certificates to another participant and know that it will be sufficient to build a chain of trust for that participant.



■ **FIGURE 8.6** Tree-structured certification authority hierarchy.

There are some significant issues with building chains of trust. Most importantly, even if you are certain that you have the public key of the root CA, you need to be sure that every CA from the root on down is doing its job properly. If just one CA in the chain is willing to issue certificates to entities without verifying their identities, then what looks like a valid chain of certificates becomes meaningless. For example, a root CA might issue a certificate to a second-tier CA and thoroughly verify that the name on the certificate matches the business name of the CA, but that second-tier CA might be willing to sell certificates to anyone who asks, without verifying their identity. This problem gets worse the longer the chain of trust. X.509 certificates provide the option of restricting the set of entities that the subject of a certificate is, in turn, trusted to certify.

There can be more than one root to a certification tree, and this is common in securing Web transactions today, for example. Web browsers such as Firefox and Internet Explorer come pre-equipped with certificates for a set of CAs; in effect, the browser's producer has decided these CAs and their keys can be trusted. A user can also add CAs to those that their browser recognizes as trusted. These certificates are accepted by Secure Socket Layer (SSL)/Transport Layer Security (TLS), the protocol most often used to secure Web transactions, which we discuss below in Section 8.4.3. (If you are curious, you can poke around in the preferences settings for your browser and find the "view certificates" option to see how many CAs your browser is configured to trust.)

Web of Trust

An alternative model of trust is the *web of trust* exemplified by Pretty Good Privacy (PGP), which is further discussed in Section 8.4.3. PGP is a security system for email, so email addresses are the identities to which keys are bound and by which certificates are signed. In keeping with PGP's roots as protection against government intrusion, there are no CAs. Instead, every individual decides whom they trust and how much they trust them—in this model, trust is a matter of degree. In addition, a public key certificate can include a confidence level indicating how confident the signer is of the key binding claimed in the certificate, so a given user may have to have several certificates attesting to the same key binding before he is willing to trust it.

For example, suppose you have a certificate for Bob provided by Alice; you can assign a moderate level of trust to that certificate. However, if you have additional certificates for Bob that were provided by C and D, each of whom is also moderately trustworthy, that might considerably increase your level of confidence that the public key you have for Bob is valid. In short, PGP recognizes that the problem of establishing trust is quite a personal matter and gives users the raw material to make their own decisions, rather than assuming that they are all willing to trust in a single hierarchical structure of CAs. To quote Phil Zimmerman, the developer of PGP, "PGP is for people who prefer to pack their own parachutes."

PGP has become quite popular in the networking community, and PGP key-signing parties are a regular feature of IETF meetings. At these gatherings, an individual can

- Collect public keys from others whose identity he knows.
- Provide his public key to others.
- Get his public key signed by others, thus collecting certificates that will be persuasive to an increasingly large set of people.
- Sign the public key of other individuals, thus helping them build up their set of certificates that they can use to distribute their public keys.
- Collect certificates from other individuals whom he trusts enough to sign keys.

Thus, over time, a user will collect a set of certificates with varying degrees of trust.

Certificate Revocation

One issue that arises with certificates is how to revoke, or undo, a certificate. Why is this important? Suppose that you suspect that someone has discovered your private key. There may be any number of certificates in the universe that assert that you are the owner of the public key corresponding to that private key. The person who discovered your private key thus has everything he needs to impersonate you: valid certificates and your private key. To solve this problem, it would be nice to be able to revoke the certificates that bind your old, compromised key to your identity, so that the impersonator will no longer be able to persuade other people that he is you.

The basic solution to the problem is simple enough. Each CA can issue a *certificate revocation list* (CRL), which is a digitally signed list of certificates that have been revoked. The CRL is periodically updated and made publicly available. Because it is digitally signed, it can just be posted on a website. Now, when Alice receives a certificate for Bob that she wants to verify, she will first consult the latest CRL issued by the CA. As long as the certificate has not been revoked, it is valid. Note that, if all certificates have unlimited life spans, the CRL would always be getting longer, since you could never take a certificate off the CRL for fear that some copy of the revoked certificate might be used. For this reason, it is common to attach an expiration date to a certificate when it is issued. Thus, we can limit the length of time that a revoked certificate needs to stay on a CRL. As soon as its original expiration date is passed, it can be removed from the CRL.

8.2.2 Predistribution of Symmetric Keys

If Alice wants to use a secret-key cipher to communicate with Bob, she can't just pick a key and send it to to him because, without already having a key, they can't encrypt this key to keep it confidential and they can't authenticate each other. As with public keys, some predistribution scheme is needed. Predistribution is harder for symmetric keys than for public keys for two obvious reasons:

- While only one public key per entity is sufficient for authentication and confidentiality, there must be a symmetric key for each pair of entities who wish to communicate. If there are N entities, that means $N(N-1)/2$ keys.
- Unlike public keys, secret keys must be kept secret.

In summary, there are a lot more keys to distribute, and you can't use certificates that everyone can read.

The most common solution is to use a *Key Distribution Center* (KDC). A KDC is a trusted entity that shares a secret key with each other entity. This brings the number of keys down to a more manageable $N - 1$, few enough to establish out of band for some applications. When Alice wishes to communicate with Bob, that communication does not travel via the KDC. Rather, the KDC participates in a protocol that authenticates Alice and Bob—using the keys that the KDC already shares with each of them—and generates a new session key for them to use. Then Alice and Bob communicate directly using their session key. Kerberos (Section 8.3.3) is a widely used system based on this approach.

8.3 AUTHENTICATION PROTOCOLS

We described how to encrypt messages and build authenticators in Section 8.1 and how to predistribute the necessary keys in Section 8.2. It might seem as if all we have to do to make a protocol secure is append an authenticator to every message and, if we want confidentiality, encrypt the message.

There are two main reasons why it's not that simple. First, there is the problem of a *replay attack*: an adversary retransmitting a copy of a message that was previously sent. If the message was an order you had placed on a website, for example, then the replayed message would appear to the website as though you had ordered more of the same. Even though it wasn't the original incarnation of the message, its authenticator would still be valid; after all, the message was created by you, and it wasn't modified. In a variation of this attack called a *suppress-replay attack*, an adversary might merely delay your message (by intercepting and later replaying it), so that it is received at a time when it is no longer appropriate. For example, an adversary could delay your order to buy stock from an auspicious time to a time when you would not have wanted to buy. Although this message would in a sense be the original, it wouldn't be timely. Originality and timeliness may be considered aspects of integrity. Ensuring them will in most cases require a nontrivial, back-and-forth protocol.

The other problem we have not yet solved is how to establish a session key. A session key is a symmetric-key cipher key generated on the fly and

used for just one session as described in Section 8.2. This too involves a nontrivial protocol.

What these two issues have in common is authentication. If a message is not original and timely, then from a practical standpoint we want to consider it as not being authentic, not being from whom it claims to be. And, obviously, when you are arranging to share a new session key with someone, you want to know you are sharing it with the right person. Usually, authentication protocols establish a session key at the same time, so that at the end of the protocol Alice and Bob have authenticated each other and they have a new symmetric key to use. Without a new session key, the protocol would just authenticate Alice and Bob at one point in time; a session key allows them to efficiently authenticate subsequent messages. Generally, session key establishment protocols perform authentication (a notable exception is Diffie–Hellman; see Section 8.3.4), so the terms *authentication protocol* and *session key establishment protocol* are almost synonymous.

There is a core set of techniques used to ensure originality and timeliness in authentication protocols. We describe those techniques before moving on to particular protocols.

8.3.1 Originality and Timeliness Techniques

We have seen that authenticators alone do not enable us to detect messages that are not original or timely. One approach is to include a timestamp in the message. Obviously the timestamp itself must be tamperproof, so it must be covered by the authenticator. The primary drawback to timestamps is that they require distributed clock synchronization. Since our system would then depend on synchronization, the clock synchronization itself would need to be defended against security threats, in addition to the usual challenges of clock synchronization. Another issue is that distributed clocks are synchronized to only a certain degree—a certain margin of error. Thus, the timing integrity provided by timestamps is only as good as the degree of synchronization.

Another approach is to include a *nonce*—a random number used only once—in the message. Participants can then detect replay attacks by checking whether a nonce has been used previously. Unfortunately, this requires keeping track of past nonces, of which a great many could accumulate. One solution is to combine the use of timestamps and nonces, so that nonces are required to be unique only within a certain span of time.

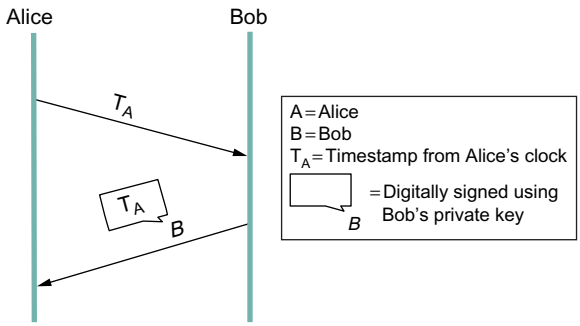


FIGURE 8.7 A challenge-response protocol.

That makes ensuring uniqueness of nonces manageable while requiring only loose synchronization of clocks.

Another solution to the shortcomings of timestamps and nonces is to use one or both of them in a *challenge–response* protocol. Suppose we use a timestamp. In a challenge–response protocol, Alice sends Bob a timestamp, challenging Bob to encrypt it in a response message (if they share a symmetric key) or digitally sign it in a response message (if Bob has a public key, as in Figure 8.7). The encrypted timestamp is like an authenticator that additionally proves timeliness. Alice can easily check the timeliness of the timestamp in a response from Bob since that timestamp comes from Alice's own clock—no distributed clock synchronization needed. Suppose instead that the protocol uses nonces. Then Alice need only keep track of those nonces for which responses are currently outstanding and haven't been outstanding too long; any purported response with an unrecognized nonce must be bogus.

The beauty of challenge–response, which might otherwise seem excessively complex, is that it combines timeliness and authentication; after all, only Bob (and possibly Alice, if it's a symmetric-key cipher) knows the key necessary to encrypt the never before seen timestamp or nonce. Timestamps or nonces are used in most of the authentication protocols that follow.

8.3.2 Public-Key Authentication Protocols

In the following discussion, we assume that Alice and Bob's public keys have been predistributed to each other via some means such as a PKI

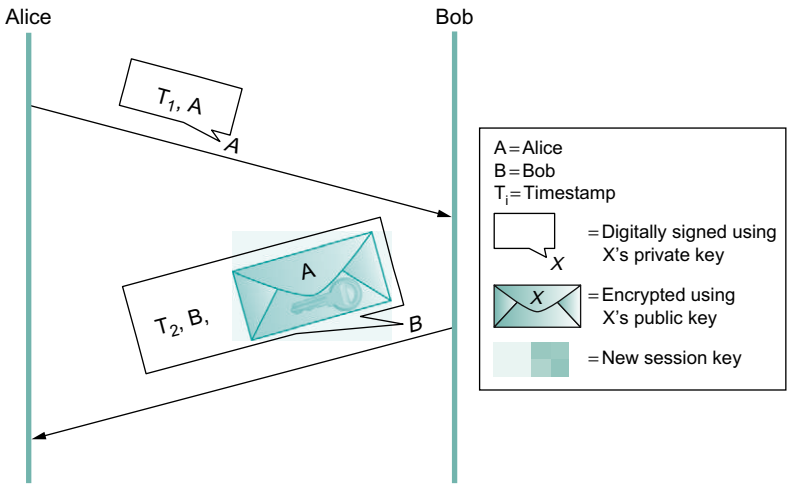
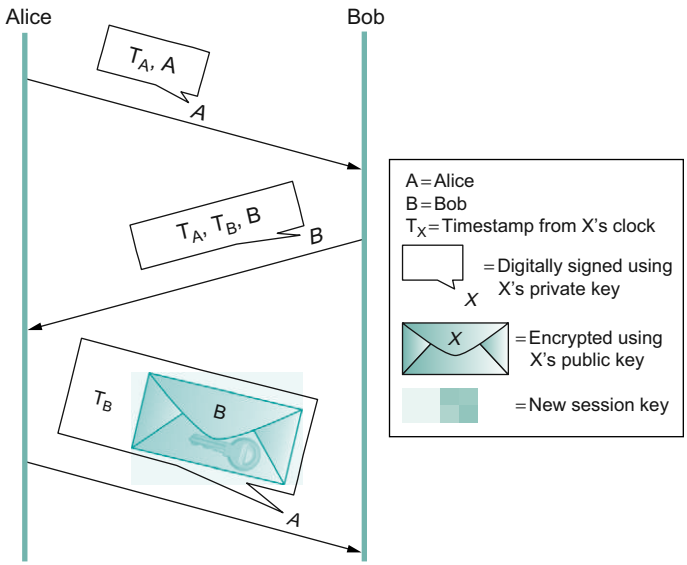


FIGURE 8.8 A public-key authentication protocol that depends on synchronization.

(Section 8.2.1). We mean this to include the case where Alice includes her certificate in her first message to Bob, and the case where Bob searches for a certificate about Alice when he receives her first message.

This first protocol (Figure 8.8) relies on Alice and Bob's clocks being synchronized. Alice sends Bob a message with a timestamp and her identity in plaintext plus her digital signature. Bob uses the digital signature to authenticate the message and the timestamp to verify its freshness. Bob sends back a message with a timestamp and his identity in plaintext, as well as a new session key encrypted (for confidentiality) using Alice's public key, all digitally signed. Alice can verify the authenticity and freshness of the message, so she knows she can trust the new session key. To deal with imperfect clock synchronization, the timestamps could be augmented with nonces.

The second protocol (Figure 8.9) is similar but does not rely on clock synchronization. In this protocol, Alice again sends Bob a digitally signed message with a timestamp and her identity. Because their clocks aren't synchronized, Bob cannot be sure that the message is fresh. Bob sends back a digitally signed message with Alice's original timestamp, his own new timestamp, and his identity. Alice can verify the freshness of Bob's reply by comparing her current time against the timestamp that originated with her. She then sends Bob a digitally signed message with his



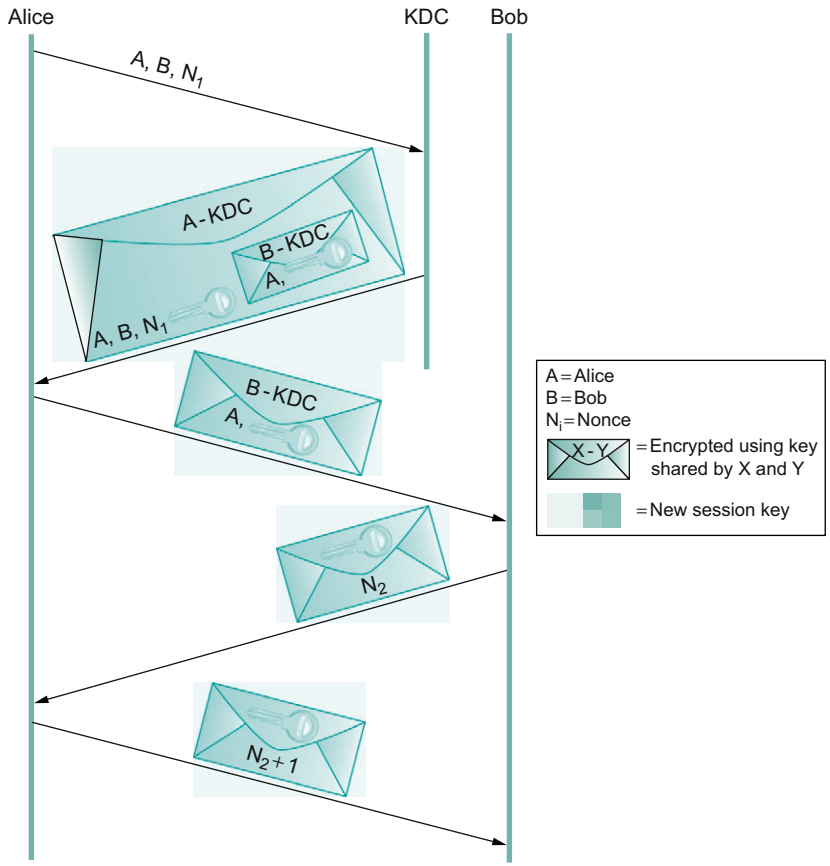
■ **FIGURE 8.9** A public-key authentication protocol that does not depend on synchronization. Alice checks her own timestamp against her own clock, and likewise for Bob.

original timestamp and a new session key encrypted using Bob's public key. Bob can verify the freshness of the message because the timestamp came from his clock, so he knows he can trust the new session key. The timestamps essentially serve as convenient nonces, and indeed this protocol could use nonces instead.

8.3.3 Symmetric-Key Authentication Protocols

As explained in Section 8.2.2, only in fairly small systems is it practical to predistribute symmetric keys to every pair of entities. We focus here on larger systems, where each entity would have its own *master key* shared only with a Key Distribution Center (KDC). In this case, symmetric-key-based authentication protocols involve three parties: Alice, Bob, and a KDC. The end product of the authentication protocol is a session key shared between Alice and Bob that they will use to communicate directly, without involving the KDC.

The Needham–Schroeder authentication protocol is illustrated in Figure 8.10. Note that the KDC doesn't actually authenticate Alice's initial message and doesn't communicate with Bob at all. Instead, the KDC



■ **FIGURE 8.10** The Needham–Schroeder authentication protocol.

uses its knowledge of Alice's and Bob's master keys to construct a reply that would be useless to anyone other than Alice (because only Alice can decrypt it) and contains the necessary ingredients for Alice and Bob to perform the rest of the authentication protocol themselves.

The nonce in the first two messages is to assure Alice that the KDC's reply is fresh. The second and third messages include the new session key and Alice's identifier, encrypted together using Bob's master key. It is a sort of symmetric-key version of a public-key certificate; it is in effect a signed statement by the KDC (because the KDC is the only entity besides Bob who knows Bob's master key) that the enclosed session key is owned by Alice and Bob. Although the nonce in the last two messages is intended

to assure Bob that the third message was fresh, there is a flaw in this reasoning—see Exercise 4.

Kerberos

Kerberos is an authentication system based on the Needham–Schroeder protocol and specialized for client/server environments. Originally developed at MIT, it has been standardized by the IETF and is available as both open source and commercial products. We will focus here on some of Kerberos's interesting innovations.

Kerberos clients are generally human users, and users authenticate themselves using passwords. Alice's master key, shared with the KDC, is derived from her password—if you know the password, you can compute the key. Kerberos assumes anyone can physically access any client machine; therefore, it is important to minimize the exposure of Alice's password or master key not just in the network but also on any machine where she logs in. Kerberos takes advantage of Needham–Schroeder to accomplish this. In Needham-Schroeder, the only time Alice needs to use her password is when decrypting the reply from the KDC. Kerberos client-side software waits until the KDC's reply arrives, prompts Alice to enter her password, computes the master key and decrypts the KDC's reply, and then erases all information about the password and master key to minimize its exposure. Also note that the only sign a user sees of Kerberos is when the user is prompted for a password.

In Needham–Schroeder, the KDC's reply to Alice plays two roles: It gives her the means to prove her identity (only Alice can decrypt the reply), and it gives her a sort of symmetric-key certificate or "ticket" to present to Bob—the session key and Alice's identifier, encrypted with Bob's master key. In Kerberos, those two functions—and the KDC itself, in effect—are split up (Figure 8.11). A trusted server called an Authentication Server (AS) plays the first KDC role of providing Alice with something she can use to prove her identity—not to Bob this time, but to a second trusted server called a Ticket Granting Server (TGS). The TGS plays the second KDC role, replying to Alice with a ticket she can present to Bob. The attraction of this scheme is that if Alice needs to communicate with several servers, not just Bob, then she can get tickets for each of them from the TGS without going back to the AS.

In the client/server application domain for which Kerberos is intended, it is reasonable to assume a degree of clock synchronization. This

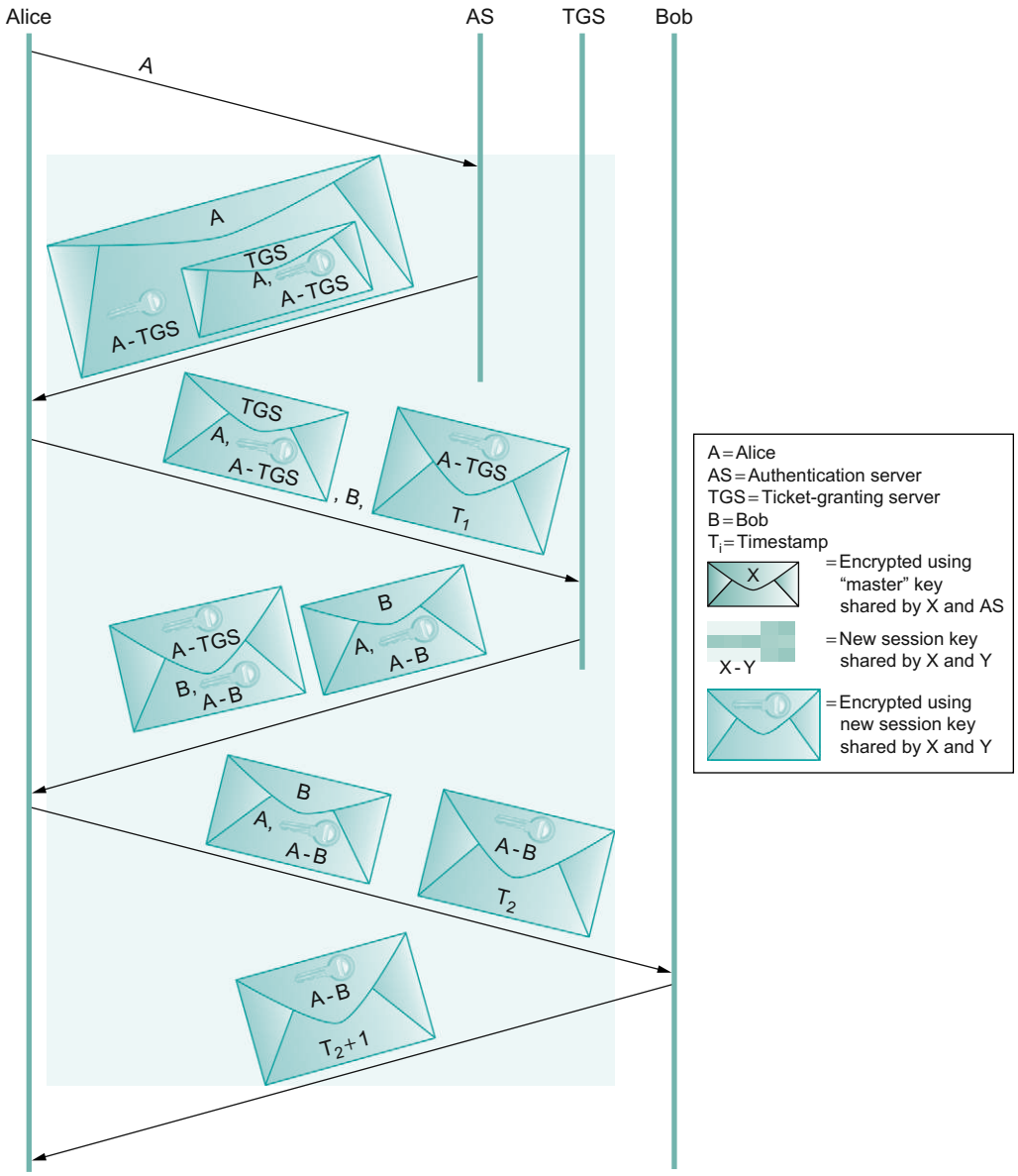


FIGURE 8.11 Kerberos authentication.

allows Kerberos to use timestamps and lifespans instead of Needham–Shroeder's nonces, and thereby eliminate the Needham-Schroeder security weakness explored in Exercise 4. Kerberos supports a choice of

cryptographic algorithms including the hashes SHA-1 and MD5 and the symmetric-key ciphers AES, 3DES, and DES.

8.3.4 Diffie–Hellman Key Agreement

The Diffie–Hellman key agreement protocol establishes a session key without using any predistributed keys. The messages exchanged between Alice and Bob can be read by anyone able to eavesdrop, and yet the eavesdropper won't know the session key that Alice and Bob end up with. On the other hand, Diffie–Hellman doesn't authenticate the participants. Since it is rarely useful to communicate securely without being sure whom you're communicating with, Diffie–Hellman is usually augmented in some way to provide authentication. One of the main uses of Diffie–Hellman is in the Internet Key Exchange (IKE) protocol, a central part of the IP Security (IPsec) architecture.

The Diffie–Hellman protocol has two parameters, p and g, both of which are public and may be used by all the users in a particular system. Parameter p must be a prime number. The integers $\bmod p$ (short for modulo p) are 0 through $p-1$, since $x \bmod p$ is the remainder after x is divided by p, and form what mathematicians call a *group* under multiplication. Parameter g (usually called a generator) must be a *primitive root* of p: For every number n from 1 through $p-1$ there must be some value k such that $n = g^k \bmod p$. For example, if p were the prime number 5 (a real system would use a much larger number), then we might choose 2 to be the generator g since:

$$1 = 2^0 \bmod p$$

$$2 = 2^1 \bmod p$$

$$3 = 2^3 \bmod p$$

$$4 = 2^2 \bmod p$$

Suppose Alice and Bob want to agree on a shared symmetric key. Alice and Bob, and everyone else, already know the values of p and g. Alice generates a random private value a and Bob generates a random private value b. Both a and b are drawn from the set of integers $\{1, \ldots, p-1\}$. Alice and Bob derive their corresponding public values—the values they will send to each other unencrypted—as follows. Alice's public value is

$$g^a \bmod p$$

and Bob's public value is

$$g^b \bmod p$$

They then exchange their public values. Finally, Alice computes

$$g^{ab} \bmod p = (g^b \bmod p)^a \bmod p$$

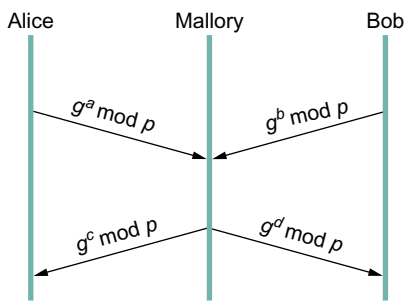
and Bob computes

$$g^{ba} \bmod p = (g^a \bmod p)^b \bmod p.$$

Alice and Bob now have $g^{ab} \bmod p$ (which is equal to $g^{ba} \bmod p$) as their shared symmetric key.

Any eavesdropper would know p, g, and the two public values $g^a \bmod p$ and $g^b \bmod p$. If only the eavesdropper could determine a or b, she could easily compute the resulting key. Determining a or b from that information is, however, computationally infeasible for suitably large p, a, and b; it is known as the *discrete logarithm problem*.

On the other hand, there is the problem of Diffie–Hellman's lack of authentication. One attack that can take advantage of this is the *man-in-the-middle attack*. Suppose Mallory is an adversary with the ability to intercept messages. Mallory already knows p and g since they are public, and she generates random private values c and d to use with Alice and Bob, respectively. When Alice and Bob send their public values to each other, Mallory intercepts them and sends her own public values, as in Figure 8.12. The result is that Alice and Bob each end up unknowingly sharing a key with Mallory instead of each other.



■ **FIGURE 8.12** A man-in-the-middle attack.

A variant of Diffie–Hellman sometimes called *fixed Diffie–Hellman* supports authentication of one or both participants. It relies on certificates that are similar to public key certificates but instead certify the Diffie–Hellman public parameters of an entity. For example, such a certificate would state that Alice's Diffie–Hellman parameters are p, g, and $g^a \bmod p$ (note that the value of a would still be known only to Alice). Such a certificate would assure Bob that the other participant in Diffie–Hellman is Alice—or else the other participant won't be able to compute the secret key, because she won't know a. If both participants have certificates for their Diffie–Hellman parameters, they can authenticate each other. If just one has a certificate, then just that one can be authenticated. This is useful in some situations; for example, when one participant is a web server and the other is an arbitrary client, the client can authenticate the web server and establish a session key for confidentiality before sending a credit card number to the web server.

8.4 EXAMPLE SYSTEMS

At this point, we have seen many of the components that are required to provide one or two aspects of security. These components include cryptographic algorithms, key predistribution mechanisms, and authentication protocols. In this section, we examine some complete systems that use these components.

These systems can be roughly categorized by the protocol layer at which they operate. Systems that operate at the application layer include Pretty Good Privacy (PGP), which provides electronic mail security, and Secure Shell (SSH), a secure remote login facility. At the transport layer, there is the IETF's Transport Layer Security (TLS) standard and the older protocol from which it derives, Secure Socket Layer (SSL). The IPsec (IP Security) protocols, as their name implies, operate at the IP (network) layer. 802.11i provides security at the link layer of wireless networks. This section describes the salient features of each of these approaches.

You might reasonably wonder why security has to be provided at so many different layers. One reason is that different threats require different defensive measures, and this often translates into securing a different protocol layer. For example, if your main concern is with a person in the building next door snooping on your traffic as it flows between your laptop and your 802.11 access point, then you probably want security at the

link layer. However, if you want to be really sure you are connected to your bank's website and preventing all the data that you send to the bank from being read by curious employees of some Internet service provider, then something that extends all the way from your machine to the bank's server—like the transport layer—may be the right place to secure the traffic. As is often the case, there is no one-size-fits-all solution.

The security systems described below have the ability to vary which cryptographic algorithms they use. The idea of making a security system algorithm independent is a very good one, because you never know when your favorite cryptographic algorithm might be proved to be insufficiently strong for your purposes. It would be nice if you could quickly change to a new algorithm without having to change the protocol specification or implementation. Note the analogy to being able to change keys without changing the algorithm; if one of your cryptographic algorithms turns out to be flawed, it would be great if your entire security architecture didn't need an immediate redesign.

8.4.1 Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) is a widely used approach to providing security for electronic mail. It provides authentication, confidentiality, data integrity, and nonrepudiation. Originally devised by Phil Zimmerman, it has evolved into an IETF standard known as OpenPGP. As we saw in Section 8.2, PGP is notable for using a "web of trust" model for distribution of keys rather than a tree-like hierarchy.

PGP's confidentiality and receiver authentication depend on the receiver of an email message having a public key that is known to the sender. To provide sender authentication and nonrepudiation, the sender must have a public key that is known by the receiver. These public keys are predistributed using certificates and a web-of-trust PKI as described in Section 8.2.1. PGP supports RSA and DSS for public key certificates. These certificates may additionally specify which cryptographic algorithms are supported or preferred by the key's owner. The certificates provide bindings between email addresses and public keys.

Consider the following example of PGP being used to provide both sender authentication and confidentiality. Suppose Alice has a message to email to Bob. Alice's PGP application goes through the steps illustrated in Figure 8.13. First, the message is digitally signed by Alice; MD5, SHA-1, and the SHA-2 family are among the hashes that may be used in

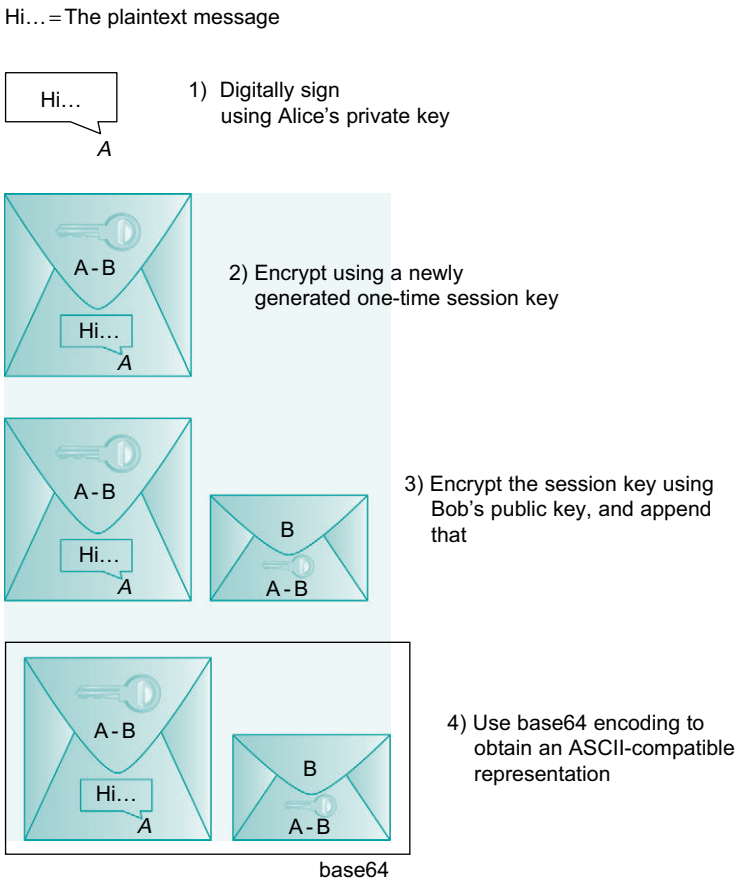


FIGURE 8.13 PGP's steps to prepare a message for emailing from Alice to Bob.

the digital signature. Her PGP application then generates a new session key for just this one message; AES and 3DES are among the supported symmetric-key ciphers. The digitally signed message is encrypted using the session key, then the session key itself is encrypted using Bob's public key and appended to the message. Alice's PGP application reminds her of the level of trust she had previously assigned to Bob's public key, based on the number of certificates she has for Bob and the trustworthiness of the individuals who signed the certificates. Finally, not for security but because email messages have to be sent in ASCII, a base64 encoding (as described in Section 9.1.1) is applied to the message to convert it to an ASCII-compatible representation. Upon receiving the PGP message in an

email, Bob's PGP application reverses this process step-by-step to obtain the original plaintext message and confirm Alice's digital signature—and reminds Bob of the level of trust he has in Alice's public key.

Email has particular characteristics that allow PGP to embed an adequate authentication protocol in this one-message data transmission protocol, avoiding the need for any prior message exchange (and sidestepping some of the complexities described earlier in Section 8.3). Alice's digital signature suffices to authenticate her. Although there is no proof that the message is timely, legitimate email isn't guaranteed to be timely either. There is also no proof that the message is original, but Bob is an email user and probably a fault-tolerant human who can recover from duplicate emails (which, again, are not out of the question under normal operation anyway). Alice can be sure that only Bob could read the message because the session key was encrypted with his public key. Although this protocol doesn't prove to Alice that Bob is actually there and received the email, an authenticated email from Bob back to Alice could do this.

The preceding discussion gives a good example of why application-layer security mechanisms can be helpful. Only with a full knowledge of how the application works can you make the right choices about which attacks to defend against (like forged email) versus which to ignore (like delayed or replayed email).

8.4.2 Secure Shell (SSH)

The Secure Shell (SSH) protocol is used to provide a remote login service and is intended to replace the less secure Telnet and rlogin programs used in the early days of the Internet. (SSH can also be used to remotely execute commands and transfer files, like the Unix rsh and rcp commands, respectively, but we will focus first on how SSH supports remote login.) SSH is most often used to provide strong client/server authentication/message integrity—where the SSH client runs on the user's desktop machine and the SSH server runs on some remote machine that the user wants to log into—but it also supports confidentiality. Telnet and rlogin provide none of these capabilities. Note that "SSH" is often used to refer to both the SSH protocol and applications that use it; you need to figure out which from the context.

To better appreciate the importance of SSH on today's Internet, consider a couple of the scenarios where it is used. Telecommuters, for

example, often subscribe to ISPs that offer high-speed cable modem or DSL service, and they use these ISPs, and some chain of other ISPs as well, to reach machines operated by their employer. This means that when a telecommuter logs into a machine inside his employer's data center, both the passwords and all the data sent or received potentially passes through any number of untrusted networks. SSH provides a way to encrypt the data sent over these connections and to improve the strength of the authentication mechanism used to log in. A similar usage of SSH is remote login to a router, perhaps to change its configuration or read its log files; clearly, a network administrator wants to be sure that he can log into a router securely and that unauthorized parties can neither log in nor intercept the commands sent to the router or output sent back to the administrator.

The latest version of SSH, version 2, consists of three protocols:

- SSH-TRANS, a transport layer protocol
- SSH-AUTH, an authentication protocol
- SSH-CONN, a connection protocol

We focus on the first two, which are involved in remote login. We briefly discuss the purpose of SSH-CONN at the end of the section.

SSH-TRANS provides an encrypted channel between the client and server machines. It runs on top of a TCP connection. Any time a user uses an SSH application to log into a remote machine, the first step is to set up an SSH-TRANS channel between those two machines. The two machines establish this secure channel by first having the client authenticate the server using RSA. Once authenticated, the client and server establish a session key that they will use to encrypt any data sent over the channel. This high-level description skims over several details, including the fact that the SSH-TRANS protocol includes a negotiation of the encryption algorithm the two sides are going to use. For example, AES is commonly selected. Also, SSH-TRANS includes a message integrity check of all data exchanged over the channel.

The one issue we can't skim over is how the client came to possess the server's public key that it needs to authenticate the server. Strange as it may sound, the server tells the client its public key at connection time. The first time a client connects to a particular server, the SSH application warns the user that it has never talked to this machine before and asks if the user wants to continue. Although it is a risky thing to do, because

SSH is effectively not able to authenticate the server, users often say "yes" to this question. The SSH application then remembers the server's public key, and the next time the user connects to that same machine it compares this saved key with the one the server responds with. If they are the same, SSH authenticates the server. If they are different, however, the SSH application again warns the user that something is amiss, and the user is then given an opportunity to abort the connection. Alternatively, the prudent user can learn the server's public key through some out-of-band mechanism, save it on the client machine, and thus never take the "first time" risk.

Once the SSH-TRANS channel exists, the next step is for the user to actually log into the machine, or more specifically, authenticate himself or herself to the server. SSH allows three different mechanisms for doing this. First, since the two machines are communicating over a secure channel, it is OK for the user to simply send his or her password to the server. This is not a safe thing to do when using Telnet since the password would be sent in the clear, but in the case of SSH the password is encrypted in the SSH-TRANS channel. The second mechanism uses public-key encryption. This requires that the user has already placed his or her public key on the server. The third mechanism, called *host-based authentication*, basically says that any user claiming to be so-and-so from a certain set of trusted hosts is automatically believed to be that same user on the server. Host-based authentication requires that the client *host* authenticate itself to the server when they first connect; standard SSH-TRANS only authenticates the server by default.

The main thing you should take away from this discussion is that SSH is a fairly straightforward application of the protocols and algorithms we have seen throughout this chapter. However, what sometimes makes SSH a challenge to understand is all the keys a user has to create and manage, where the exact interface is operating system dependent. For example, the OpenSSH package that runs on most Unix machines supports a `ssh-keygen` command that can be used to create public/private key pairs. These keys are then stored in various files in directory `.ssh` in the user's home directory. For example, file `/.ssh/known_hosts` records the keys for all the hosts the user has logged into, file `/.ssh/authorized_keys` contains the public keys needed to authenticate the user when he or she logs into this machine (i.e., they are used on the server side), and file `/.ssh/identity` contains the private keys needed to authenticate the user on remote machines (i.e., they are used on the client side).

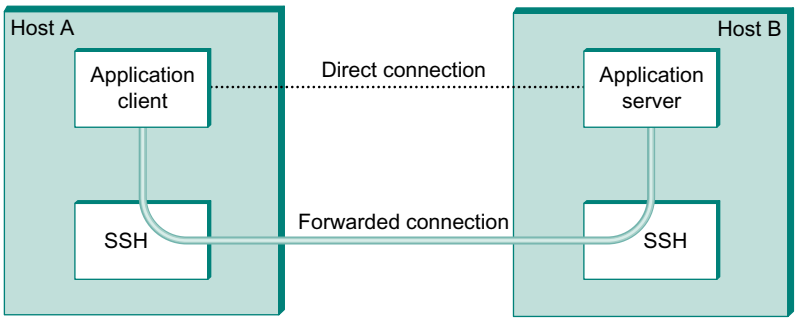


FIGURE 8.14 Using SSH port forwarding to secure other TCP-based applications.

Finally, SSH has proven so useful as a system for securing remote login, it has been extended to also support other applications, such as sending and receiving email. The idea is to run these applications over a secure "SSH tunnel." This capability is called *port forwarding*, and it uses the SSH-CONN protocol. The idea is illustrated in Figure 8.14, where we see a client on host A indirectly communicating with a server on host B by forwarding its traffic through an SSH connection. The mechanism is called *port forwarding* because when messages arrive at the well-known SSH port on the server, SSH first decrypts the contents and then "forwards" the data to the actual port at which the server is listening. This is just another sort of tunnel of the sort introduced in Section 3.2.9, which in this case happens to provide confidentiality and authentication. It's possible to provide a form of virtual private network (VPN) using SSH tunnels in this way.

8.4.3 Transport Layer Security (TLS, SSL, HTTPS)

To understand the design goals and requirements for the Transport Layer Security (TLS) standard and the Secure Socket Layer (SSL) on which TLS is based, it is helpful to consider one of the main problems that they are intended to solve. As the World Wide Web became popular and commercial enterprises began to take an interest in it, it became clear that some level of security would be necessary for transactions on the Web. The canonical example of this is making purchases by credit card. There are several issues of concern when sending your credit card information to a computer on the Web. First, you might worry that the information would be intercepted in transit and subsequently used to make unauthorized

purchases. You might also worry about the details of a transaction being modified, such as changing the purchase amount. And you would certainly like to know that the computer to which you are sending your credit card information is in fact one belonging to the vendor in question and not some other party. Thus, we immediately see a need for confidentiality, integrity, and authentication in Web transactions. The first widely used solution to this problem was SSL, originally developed by Netscape and subsequently the basis for the IETF's TLS standard.

The designers of SSL and TLS recognized that these problems were not specific to Web transactions (i.e., those using HTTP) and instead built a general-purpose protocol that sits between an application protocol such as HTTP and a transport protocol such as TCP. The reason for calling this "transport layer security" is that, from the application's perspective, this protocol layer looks just like a normal transport protocol except for the fact that it is secure. That is, the sender can open connections and deliver bytes for transmission, and the secure transport layer will get them to the receiver with the necessary confidentiality, integrity, and authentication. By running the secure transport layer on top of TCP, all of the normal features of TCP (reliability, flow control, congestion control, etc.) are also provided to the application. This arrangement of protocol layers is depicted in Figure 8.15.

When HTTP is used in this way, it is known as HTTPS (Secure HTTP). In fact, HTTP itself is unchanged. It simply delivers data to and accepts data from the SSL/TLS layer rather than TCP. For convenience, a default TCP port has been assigned to HTTPS (443). That is, if you try to connect to a server on TCP port 443, you will likely find yourself talking to the SSL/TLS protocol, which will pass your data through to HTTP provided all goes well with authentication and decryption. Although standalone implementations of SSL/TLS are available, it is more common for an

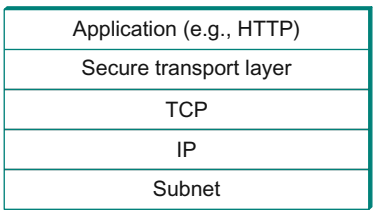


FIGURE 8.15 Secure transport layer inserted between application and TCP layers.

implementation to be bundled with applications that need it, primarily web browsers.

In the remainder of our discussion of transport layer security, we focus on TLS. Although SSL and TLS are unfortunately not interoperable, they differ in only minor ways, so nearly all of this description of TLS applies to SSL.

The Handshake Protocol

A pair of TLS participants negotiate at runtime which cryptography to use. The participants negotiate a choice of:

- Data integrity hash (MD5, SHA-1, etc.), used to implement HMACs
- Symmetric-key cipher for confidentiality (among the possibilities are DES, 3DES, and AES)
- Session key establishment approach (among the possibilities are Diffie–Hellman, fixed Diffie–Hellman, and public-key authentication protocols using RSA or DSS)

Interestingly, the participants may also negotiate the use of a compression algorithm, not because this offers any security benefits, but because it's easy to do when you're negotiating all this other stuff and you've already decided to do some expensive per-byte operations on the data.

In TLS, the confidentiality cipher uses two keys, one for each direction, and similarly two initialization vectors. The HMACs are likewise keyed with different keys for the two participants. Thus, regardless of the choice of cipher and hash, a TLS session requires effectively six keys. TLS derives all of them from a single shared *master secret*. The master secret is a 384-bit (48-byte) value that in turn is derived in part from the "session key" that results from TLS's session key establishment protocol.

The part of TLS that negotiates the choices and establishes the shared master secret is called the *handshake protocol*. (Actual data transfer is performed by TLS's *record protocol*.) The handshake protocol is at heart a session key establishment protocol, with a master secret instead of a session key. Since TLS supports a choice of approaches to session key establishment, these call for correspondingly different protocol variants. Furthermore, the handshake protocol supports a choice between mutual authentication of both participants, authentication of just one participant (this is the most common case, such as authenticating a website but not a user), or no authentication at all (anonymous Diffie–Hellman). Thus,

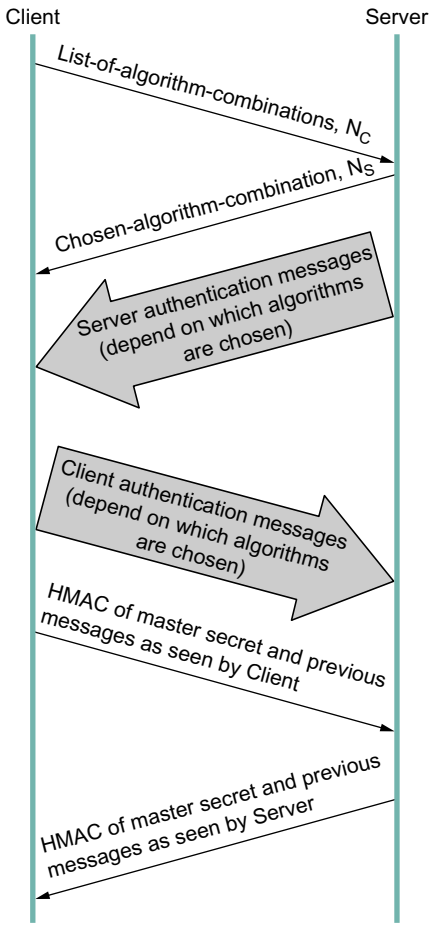


FIGURE 8.16 Handshake protocol to establish TLS session.

the handshake protocol knits together several session key establishment protocols into a single protocol.

Figure 8.16 shows the handshake protocol at a high level. The client initially sends a list of the combinations of cryptographic algorithms that it supports, in decreasing order of preference. The server responds, giving the single combination of cryptographic algorithms it selected from those listed by the client. These messages also contain a *client nonce* and a *server nonce*, respectively, that will be incorporated in generating the master secret later.

At this point, the negotiation phase is complete. The server now sends additional messages based on the negotiated session key establishment protocol. That could involve sending a public-key certificate or a set of Diffie–Hellman parameters. If the server requires authentication of the client, it sends a separate message indicating that. The client then responds with its part of the negotiated key exchange protocol.

Now the client and server each have the information necessary to generate the master secret. The "session key" that they exchanged is not in fact a key, but instead what TLS calls a *pre-master secret*. The master secret is computed (using a published algorithm) from this pre-master secret, the client nonce, and the server nonce. Using the keys derived from the master secret, the client then sends a message that includes a hash of all the preceding handshake messages, to which the server responds with a similar message. This enables them to detect any discrepancies between the handshake messages they sent and received, such as would result, for example, if a man in the middle modified the initial unencrypted client message to weaken its choices of cryptographic algorithms.

The Record Protocol

Within a session established by the handshake protocol, TLS's record protocol adds confidentiality and integrity to the underlying transport service. Messages handed down from the application layer are:

1. Fragmented or coalesced into blocks of a convenient size for the following steps
2. Optionally compressed
3. Integrity-protected using an HMAC
4. Encrypted using a symmetric-key cipher
5. Passed to the transport layer (normally TCP) for transmission

The record protocol uses an HMAC as an authenticator. The HMAC uses whichever hash algorithm (MD5, SHA-1, etc.) was negotiated by the participants. The client and server have different keys to use when computing HMACs, making them even harder to break. Furthermore, each record protocol message is assigned a sequence number, which is included when the HMAC is computed—even though the sequence number is never explicit in the message. This implicit sequence number prevents replays or reorderings of messages. This is needed because, although TCP can deliver sequential, unduplicated messages to the layer

above it under normal assumptions, those assumptions do not include an adversary that can intercept TCP messages, modify messages, or send bogus ones. On the other hand, it is TCP's delivery guarantees that make it possible for TLS to rely on a legitimate TLS message having the next implicit sequence number in order.

Another interesting feature of the TLS protocol, which is quite a useful feature for Web transactions, is the ability to resume a session. To understand the motivation for this, it is helpful to understand how HTTP makes use of TCP connections. (The details of HTTP are presented in Section 9.1.2.) Each HTTP operation, such as getting a page of text or an image from a server, requires a new TCP connection to be opened. Retrieving a single page with a number of embedded graphical objects might take many TCP connections. Recall from Section 5.2 that opening a TCP connection requires a three-way handshake before data transmission can start. Once the TCP connection is ready to accept data, the client would then need to start the TLS handshake protocol, taking at least another two round-trip times (and consuming some amount of processing resources and network bandwidth) before actual application data could be sent. The resumption capability of TLS alleviates this problem.

Session resumption is an optimization of the handshake that can be used in those cases where the client and the server have already established some shared state in the past. The client simply includes the session ID from a previously established session in its initial handshake message. If the server finds that it still has state for that session, and the resumption option was negotiated when that session was originally created, then the server can reply to the client with an indication of success, and data transmission can begin using the algorithms and parameters previously negotiated. If the session ID does not match any session state cached at the server, or if resumption was not allowed for the session, then the server will fall back to the normal handshake process.

8.4.4 IP Security (IPsec)

Probably the most ambitious of all the efforts to integrate security into the Internet happens at the IP layer. Support for IPsec, as the architecture is called, is optional in IPv4 but mandatory in IPv6.

IPsec is really a framework (as opposed to a single protocol or system) for providing all the security services discussed throughout this chapter. IPsec provides three degrees of freedom. First, it is highly modular,

allowing users (or more likely, system administrators) to select from a variety of cryptographic algorithms and specialized security protocols. Second, IPsec allows users to select from a large menu of security properties, including access control, integrity, authentication, originality, and confidentiality. Third, IPsec can be used to protect narrow streams (e.g., packets belonging to a particular TCP connection being sent between a pair of hosts) or wide streams (e.g., all packets flowing between a pair of routers).

When viewed from a high level, IPsec consists of two parts. The first part is a pair of protocols that implement the available security services. They are the Authentication Header (AH), which provides access control, connectionless message integrity, authentication, and antireplay protection, and the Encapsulating Security Payload (ESP), which supports these same services, plus confidentiality. AH is rarely used so we focus on ESP here. The second part is support for key management, which fits under an umbrella protocol known as the Internet Security Association and Key Management Protocol (ISAKMP).

The abstraction that binds these two pieces together is the *security association* (SA). An SA is a simplex (one-way) connection with one or more of the available security properties. Securing a bidirectional communication between a pair of hosts—corresponding to a TCP connection, for example—requires two SAs, one in each direction. Although IP is a connectionless protocol, security depends on connection state information such as keys and sequence numbers. When created, an SA is assigned an ID number called a *security parameters index* (SPI) by the receiving machine. A combination of this SPI and the destination IP addresses uniquely identifies an SA. An ESP header includes the SPI so the receiving host can determine which SA an incoming packet belongs to and, hence, what algorithms and keys to apply to the packet.

SAs are established, negotiated, modified, and deleted using ISAKMP. It defines packet formats for exchanging key generation and authentication data. These formats aren't terribly interesting because they provide a framework only—the exact form of the keys and authentication data depends on the key generation technique, the cipher, and the authentication mechanism that is used. Moreover, ISAKMP does not specify a particular key exchange protocol, although it does suggest the Internet Key Exchange (IKE) as one possibility, and IKE is what is used in practice.

ESP is the protocol used to securely transport data over an established SA. In IPv4, the ESP header follows the IP header; in IPv6, it is an extension

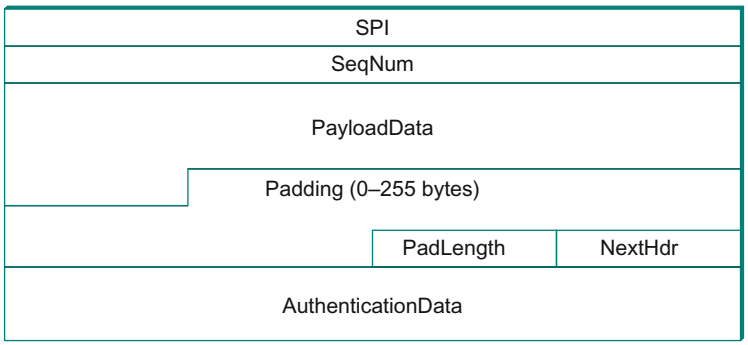
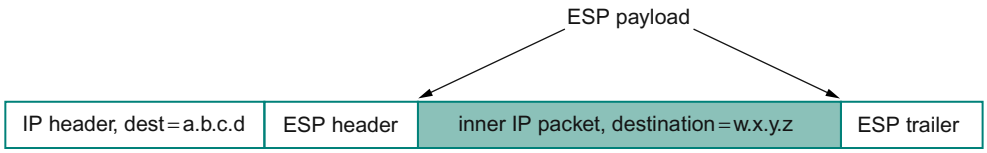


FIGURE 8.17 IPsec's ESP format.

header. Its format uses both a header and a trailer, as shown in Figure 8.17. The SPI field lets the receiving host identify the security association to which the packet belongs. The SeqNum field protects against replay attacks. The packet's PayloadData contains the data described by the NextHdr field. If confidentiality is selected, then the data is encrypted using whatever cipher was associated with the SA. The PadLength field records how much padding was added to the data; padding is sometimes necessary because, for example, the cipher requires the plaintext to be a multiple of a certain number of bytes or to ensure that the resulting ciphertext terminates on a 4-byte boundary. Finally, the AuthenticationData carries the authenticator.

IPsec supports a *tunnel mode* as well as the more straightforward *transport mode*. Each SA operates in one or the other mode. In a transport mode SA, ESP's payload data is simply a message for a higher layer such as UDP or TCP. In this mode, IPsec acts as an intermediate protocol layer, much like SSL/TLS does between TCP and a higher layer. When an ESP message is received, its payload is passed to the higher level protocol.

In a tunnel mode SA, however, ESP's payload data is itself an IP packet, as in Figure 8.18. The source and destination of this inner IP packet may be different from those of the outer IP packet. When an ESP message is received, its payload is forwarded on as a normal IP packet. The most common way to use the ESP is to build an "IPsec tunnel" between two routers, typically firewalls. For example, a corporation wanting to link two sites using the Internet could open a pair of tunnel-mode SAs between a router at one site and a router at the other site, as we discussed in Section 3.2.9. An IP packet outgoing from one site would, at the outgoing



■ **FIGURE 8.18** An IP packet with a nested IP packet encapsulated using ESP in tunnel mode. Note that the inner and outer packets have different addresses.

router, become the payload of an ESP message sent to the other site's router. The receiving router would unwrap the payload IP packet and forward it on to its true destination.

These tunnels may also be configured to use ESP with confidentiality and authentication, thus preventing unauthorized access to the data that traverses this virtual link and ensuring that no spurious data is received at the far end of the tunnel. Furthermore, tunnels can provide traffic confidentiality, since multiplexing multiple flows through a single tunnel obscures information about how much traffic is flowing between particular endpoints. A network of such tunnels can be used to implement an entire virtual private network (see Section 3.2.9). Hosts communicating over a VPN need not even be aware that it exists.

8.4.5 Wireless Security (802.11i)

Wireless links (Section 2.7) are particularly exposed to security threats due to the lack of any physical security on the medium. While the convenience of 802.11 has prompted widespread acceptance of the technology, lack of security has been a recurring problem. For example, it is all too easy for an employee of a corporation to connect an 802.11 access point to the corporate network. Since radio waves pass through most walls, if the access point lacks the correct security measures, an attacker can now gain access to the corporate network from outside the building. Similarly, a computer with a wireless network adaptor inside the building could connect to an access point outside the building, potentially exposing it to attack, not to mention the rest of the corporate network if that same computer has, say, an Ethernet connection as well.

Consequently, there has been considerable work on securing Wi-Fi links. Somewhat surprisingly, one of the early security techniques developed for 802.11, known as Wired Equivalent Privacy (WEP), turned out to be seriously flawed and quite easily breakable.

The IEEE 802.11i standard provides authentication, message integrity, and confidentiality to 802.11 (Wi-Fi) at the link layer. *WPA2* (Wi-Fi Protected Access 2) is often used as a synonym for 802.11i, although it is technically a trademark of the Wi-Fi Alliance® that certifies product compliance with 802.11i.

For backward compatibility, 802.11i includes definitions of first-generation security algorithms—including WEP—that are now known to have major security flaws. We will focus here on 802.11i's newer, stronger algorithms.

802.11i authentication supports two modes. In either mode, the end result of successful authentication is a shared Pairwise Master Key. *Personal mode*, also known as *Pre-Shared Key (PSK) mode*, provides weaker security but is more convenient and economical for situations like a home 802.11 network. The wireless device and the Access Point (AP) are preconfigured with a shared *passphrase*—essentially a very long password—from which the Pairwise Master Key is cryptographically derived.

802.11i's stronger authentication mode is based on the IEEE 802.1X framework for controlling access to a LAN, which uses an Authentication Server (AS) as in Figure 8.19. The AS and AP must be connected by a secure channel and could even be implemented as a single box, but they are logically separate. The AP forwards authentication messages between the wireless device and the AS. The protocol used for authentication is called the *Extensible Authentication Protocol* (EAP). EAP is designed to support multiple authentication methods—smart cards, Kerberos, one-time passwords, public key authentication, and so on—as well as both one-sided and mutual authentication. So EAP is better thought of as an authentication framework than a protocol. Specific EAP-compliant protocols, of which there are many, are called *EAP methods*. For example, EAP-TLS is an EAP method based on TLS authentication (see Section 8.4.3).

802.11i does not place any restrictions on what the EAP method can use as a basis for authentication. It does, however, require an EAP method that performs *mutual* authentication, because not only do we want to prevent an adversary from accessing the network via our AP, we also want to prevent an adversary from fooling our wireless devices with a bogus, malicious AP. The end result of a successful authentication is a Pairwise Master Key shared between the wireless device and the AS, which the AS then conveys to the AP.

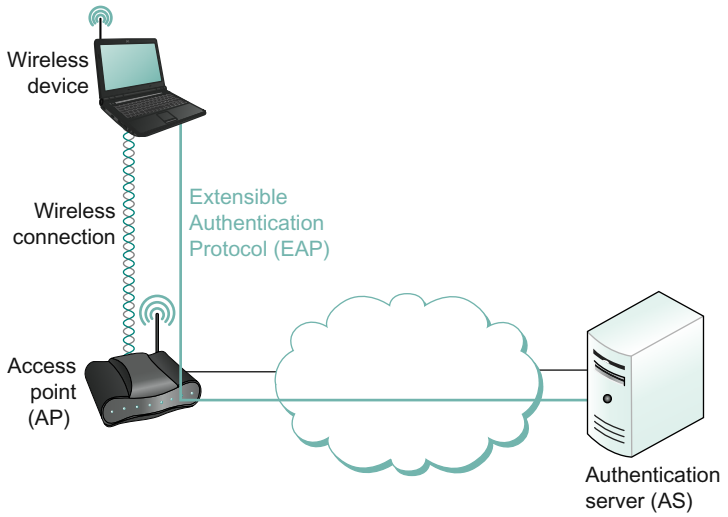


FIGURE 8.19 Use of an Authentication Server in 802.11i.

One of the main differences between the stronger AS-based mode and the weaker personal mode is that the former readily supports a unique key per client. This in turn makes it easier to change the set of clients that can authenticate themselves (e.g., to revoke access to one client) without needing to change the secret stored in every client.

With a Pairwise Master Key in hand, the wireless device and the AP execute a session key establishment protocol called the 4-way handshake to establish a Pairwise Transient Key. This Pairwise Transient Key is really a collection of keys that includes a session key called a *Temporal Key*. This session key is used by the protocol, called *CCMP*, that provides 802.11i's data confidentiality and integrity.

CCMP stands for CTR (Counter Mode) with CBC-MAC (Cipher-Block Chaining with Message Authentication Code) Protocol. CCMP uses AES in counter mode to encrypt for confidentiality. Recall that in counter mode encryption successive values of a counter are incorporated into the encryption of successive blocks of plaintext (Section 8.1.1).

CCMP uses a Message Authentication Code (MAC) as an authenticator. The MAC algorithm is based on CBC (Section 8.1.1), even though CCMP doesn't use CBC in the confidentiality encryption. In effect, CBC is performed without transmitting any of the CBC-encrypted blocks, solely so that the last CBC-encrypted block can be used as a MAC (only its first

8 bytes are actually used). The role of initialization vector is played by a specially constructed first block that includes a 48-bit packet number—a sequence number. (The packet number is also incorporated in the confidentiality encryption and serves to expose replay attacks.) The MAC is subsequently encrypted along with the plaintext in order to prevent birthday attacks, which depend on finding different messages with the same authenticator (Section 8.1.4).

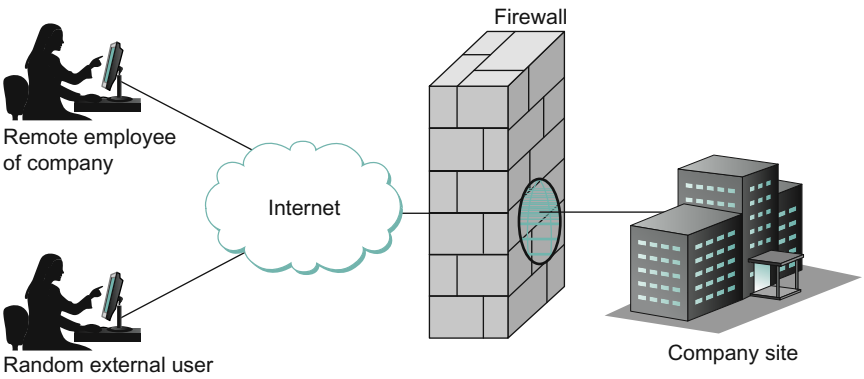
8.5 FIREWALLS

Whereas much of this chapter has focused on the uses of cryptography to provide such security features as authentication and confidentiality, there is a whole set of security issues that are not readily addressed by cryptographic means. For example, worms and viruses spread by exploiting bugs in operating systems and application programs (and sometimes human gullibility as well), and no amount of cryptography can help you if your machine has unpatched vulnerabilities. So other approaches are often used to keep out various forms of potentially harmful traffic. Firewalls are one of the most common ways to do this.

A firewall is a system that typically sits at some point of connectivity between a site it protects and the rest of the network, as illustrated in Figure 8.20. It is usually implemented as an "appliance" or part of a router, although a "personal firewall" may be implemented on an end-user machine. Firewall-based security depends on the firewall being the only connectivity to the site from outside; there should be no way to bypass the firewall via other gateways, wireless connections, or dial-up connections. The wall metaphor is somewhat misleading in the context of networks since a great deal of traffic passes through a firewall. One way to think of a firewall is that by default it blocks traffic unless that traffic is specifically allowed to pass through. For example, it might filter out all incoming messages except those addresses to a particular set of IP addresses or to particular TCP port numbers.

In effect, a firewall divides a network into a more-trusted zone internal to the firewall and a less-trusted zone external to the firewall.[4] This is

[4]The location of a firewall also often happens to be the dividing line between globally addressable regions and those that use local addresses. Hence, Network Address Translation (NAT; see Section 4.1.3) functionality and firewall functionality often are found in device, even though they are logically separate.



■ **FIGURE 8.20** A firewall filters packets flowing between a site and the rest of the Internet.

useful if you do not want external users to access a particular host or service within your site. Much of the complexity comes from the fact that you want to allow different kinds of access to different external users, ranging from the general public, to business partners, to remotely located members of your organization. A firewall may also impose restrictions on outgoing traffic to prevent certain attacks and to limit losses if an adversary succeeds in getting access inside the firewall.

Firewalls may be used to create multiple *zones of trust*, such as a hierarchy of increasingly trusted zones. A common arrangement involves three zones of trust: the internal network, the *DMZ* ("demilitarized zone"); and the rest of the Internet. The DMZ is used to hold services such as DNS and email servers that need to be accessible to the outside. Both the internal network and the outside world can access the DMZ, but hosts in the DMZ cannot access the internal network; therefore, an adversary who succeeds in compromising a host in the exposed DMZ still cannot access the internal network. The DMZ can be periodically restored to a clean state.

Firewalls filter based on IP, TCP, and UDP information, among other things. They are configured with a table of addresses that characterize the packets they will, and will not, forward. By addresses, we mean more than just the destination's IP address, although that is one possibility. Generally, each entry in the table is a 4-tuple: It gives the IP address and TCP (or UDP) port number for both the source and destination.

For example, a firewall might be configured to filter out (not forward) all packets that match the following description:

⟨ 192.12.13.14, 1234, 128.7.6.5, 80 ⟩

This pattern says to discard all packets from port 1234 on host 192.12.13.14 addressed to port 80 on host 128.7.6.5. (Port 80 is the well-known TCP port for HTTP.) Of course, it's often not practical to name every source host whose packets you want to filter, so the patterns can include wildcards. For example,

⟨ *, *, 128.7.6.5, 80 ⟩

says to filter out all packets addressed to port 80 on 128.7.6.5, regardless of what source host or port sent the packet. Notice that address patterns like these require the firewall to make forwarding/filtering decisions based on level 4 port numbers, in addition to level 3 host addresses. It is for this reason that network layer firewalls are sometimes called *level 4 switches*.

In the preceding discussion, the firewall forwards everything except where specifically instructed to filter out certain kinds of packets. A firewall could also filter out everything unless explicitly instructed to forward it, or use a mix of the two strategies. For example, instead of blocking access to port 80 on host 128.7.6.5, the firewall might be instructed to only allow access to port 25 (the SMTP mail port) on a particular mail server, such as

⟨ *, *, 128.19.20.21, 25 ⟩

but to block all other traffic. Experience has shown that firewalls are very frequently configured incorrectly, allowing unsafe access. Part of the problem is that filtering rules can overlap in complex ways, making it hard for a system administrator to correctly express the intended filtering. A design principle that maximizes security is to configure a firewall to discard all packets other than those that are explicitly allowed. Of course, this means that some valid applications might be accidentally disabled; presumably users of those applications eventually notice and ask the system administrator to make the appropriate change.

Many client/server applications dynamically assign a port to the client. If a client inside a firewall initiates access to an external server, the server's response would be addressed to the dynamically assigned port. This poses a problem: How can a firewall be configured to allow an arbitrary server's response packet but disallow a similar packet for which

there was no client request? This is not possible with a *stateless firewall*, which evaluates each packet in isolation. It requires a *stateful firewall*, which keeps track of the state of each connection. An incoming packet addressed to a dynamically assigned port would then be allowed only if it is a valid response in the current state of a connection on that port.

Modern firewalls also understand and filter based on many specific application-level protocols such as HTTP, Telnet, or FTP. They use information specific to that protocol, such as URLs in the case of HTTP, to decide whether to discard a message.

8.5.1 Strengths and Weaknesses of Firewalls

At best, a firewall protects a network from undesired access from the rest of the Internet; it cannot provide security to legitimate communication between the inside and the outside of the firewall. In contrast, the cryptography-based security mechanisms described in this chapter are capable of providing secure communication between any participants anywhere. This being the case, why are firewalls so common? One reason is that firewalls can be deployed unilaterally, using mature commercial products, while cryptography-based security requires support at both endpoints of the communication. A more fundamental reason for the dominance of firewalls is that they encapsulate security in a centralized place, in effect factoring security out of the rest of the network. A system administrator can manage the firewall to provide security, freeing the users and applications inside the firewall from security concerns—at least some kinds of security concerns.

Unfortunately, firewalls have serious limitations. Since a firewall does not restrict communication between hosts that are inside the firewall, the adversary who does manage to run code internal to a site can access all local hosts. How might an adversary get inside the firewall? The adversary could be a disgruntled employee with legitimate access, or the adversary's software could be hidden in some software installed from a CD or downloaded from the Web. It might be possible to bypass the firewall by using wireless communication or dial-up connections.

Another problem is that any parties granted access through your firewall, such as business partners or externally located employees, become a security vulnerability. If their security is not as good as yours, then an adversary could penetrate your security by penetrating their security.

On of the most serious problems for firewalls is their vulnerability to the exploitation of bugs in machines inside the firewall. Such bugs are discovered regularly, so a system administrator has to constantly monitor announcements of them. Administrators frequently fail to do so, since firewall security breaches routinely exploit security flaws that have been known for some time and have straightforward solutions.

Malware (for "malicious software") is the term for software that is designed to act on a computer in ways concealed from and unwanted by the computer's user. Viruses, worms, and spyware are common types of malware. (*Virus* is sometimes used synonymously with *malware*, but we will use it in the narrower sense in which it refers to only a particular kind of malware.) Malware code need not be natively executable object code; it could as well be interpreted code such as a script or an executable macro such as those used by Microsoft® Word.

Viruses and *worms* are characterized by the ability to make and spread copies of themselves; the difference between them is that a worm is a complete program that replicates itself, while a virus is a bit of code that is inserted (and inserts copies of itself) into another piece of software or a file, so that it is executed as part of the execution of that piece of software or as a result of opening the file. Viruses and worms typically cause problems such as consuming network bandwidth as mere side effects of attempting to spread copies of themselves. Even worse, they can also deliberately damage a system or undermine its security in various ways. They could, for example, install a *backdoor*—software that allows remote access to the system without the normal authentication. This could lead to a firewall exposing a service that should be providing its own authentication procedures but has been undermined by a backdoor.

Spyware is software that, without authorization, collects and transmits private information about a computer system or its users. Usually spyware is secretly embedded in an otherwise useful program and is spread by users deliberately installing copies. The problem for firewalls is that the transmission of the private information looks like legitimate communication.

A natural question to ask is whether firewalls (or cryptographic security) could keep malware out of a system in the first place. Most malware is indeed transmitted via networks, although it may also be transmitted via portable storage devices such as CDs and memory sticks. Certainly this

is one argument in favor of the "block everything not explicitly allowed" approach taken by many administrators in their firewall configurations.

One approach that is used to detect malware is to search for segments of code from known malware, sometimes called a *signature*. This approach has its own challenges, as cleverly designed malware can tweak its representation in various ways. There is also a potential impact on network performance to perform such detailed inspection of data entering a network. Cryptographic security cannot eliminate the problem either, although it does provide a means to authenticate the originator of a piece of software and detect any tampering, such as when a virus inserts a copy of itself.

Related to firewalls are systems known as *intrusion detection systems* (IDS) and *intrusion prevention systems* (IPS). These systems try to look for anomalous activity, such as an unusually large amount of traffic targeting a given host or port number, for example, and generate alarms for network managers or perhaps even take direct action to limit a possible attack. While there are commercial products in this space today, it is still a developing field.

8.6 SUMMARY

Networks such as the Internet are shared by parties with conflicting interests, a situation that was not entirely foreseeable in the early days of networking. The job of network security is to keep some set of users from spying on or interfering with other users of the network. Confidentiality is achieved by encrypting messages. Data integrity can be assured using cryptographic hashing. The two techniques can be combined to guarantee authenticity of messages.

Symmetric-key ciphers such as AES and 3DES use the same secret key for both encryption and decryption, so sender and receiver must share the same key. Public-key ciphers such as RSA use a public key for encryption and a secret, private key for decryption. This means that any party can use the public key to encrypt a message such that it is readable only by the holder of the private key. The fastest technique known for breaking established ciphers such as AES and RSA is brute force search of the space of possible keys, which is made computationally infeasible by the use of large keys. Most encryption for confidentiality uses symmetric-key ciphers due to their vastly superior speed, while public-key ciphers are usually reserved for authentication and session key establishment.

An authenticator is a value attached to a message to verify the authenticity and data integrity of the message. One way to generate an authenticator is to encrypt a message digest that is output by a cryptographic hash function such as MD5 or one of the SHA family of hashes. If the message digest is encrypted using the private key of a public-key cipher, the resulting authenticator is considered a digital signature, since the public key can be used to verify that only the holder of the private key could have generated it. Another kind of authenticator is a Message Authentication Code, which is output by a hash-like function that takes a shared secret value as a parameter. A hashed MAC is a MAC computed by applying a cryptographic hash to the concatenation of the plaintext message and the secret value.

A session key is used to secure a relatively short episode of communication. The dynamic establishment of a session key depends on longer-lived predistributed keys. The ownership of a predistributed public key by a certain party can be attested to by a public key certificate that is digitally signed by a trusted party. A Public Key Infrastructure is a complete scheme for certifying such bindings and depends on a chain or web of trust. Predistribution of keys for symmetric-key ciphers is different because public certificates can't be used and because symmetric-key ciphers need a unique key for each pair of participants. A Key Distribution Center is a trusted entity that shares a predistributed secret key with each other participant, so that they can use session keys, not predistributed keys, between themselves.

Authentication and session key establishment require a protocol to assure the timeliness and originality of messages. Timestamps or nonces are used to guarantee the freshness of the messages. We saw two authentication protocols that use public-key ciphers, one that required synchronized clocks and one that did not. Needham–Schroeder is a protocol for authenticating two participants who each share a master symmetric-key cipher key with a Key Distribution Center. Kerberos is an authentication system based on the Needham–Schroeder protocol and specialized for client/server environments. The Diffie–Hellman key agreement protocol establishes a session key without predistributed keys and authentication.

We discussed several systems that provide security based on these cryptographic algorithms and protocols. At the application level, PGP can be used to protect email messages and SSH can be used to securely connect to a remote machine. At the transport level, TLS can be used to protect commercial transactions on the World Wide Web. At the network

level, the IPsec architecture can be used to secure communication among any set of hosts or routers on the Internet.

A firewall filters the messages that pass between the site it protects and the rest of the network. Firewalls filter based on IP, TCP, and UDP addresses, as well as fields of some application protocols. A stateful firewall keeps track of the state of each connection so that it can allow valid responses to be delivered to dynamically assigned ports. Although firewall security has important limitations, it has the advantage of shifting some responsibility for security from users and applications to system administrators.

If you ask any Internet researcher "What would be the most important feature to include in a future Internet if we could start from scratch?" there is a pretty good chance the answer will include something about better security. One way to think about the problem is that the Internet was designed by a fairly small community who wanted access to each other's computers; today's Internet is used by a large, global community, including a reasonable number of criminals who would also like to gain access to a lot of other computers. Thus, the design of making open access the default isn't clearly a good match to today's world.

There are a lot of theories about how this situation might be improved. One problem, of course, is that the Internet

WHAT'S NEXT: COMING TO GRIPS WITH SECURITY

is such a fundamental part of everyday life now that we can't easily imagine replacing it with a new, designed-from-scratch version. There is, however, a fair amount of "clean-slate" research underway, based on the theory that working on a future Internet unhindered by questions of incremental deployment might lead to some new insights that could later be retrofitted to the current Internet. (See the Further Reading section of Chapter 3 for some references.)

The near-term outlook seems to be for a continual playing out of the cat-and-mouse game that has gone on for some time. Firewalls, intrusion detection systems, and DoS-mitigation systems get more sophisticated; attackers find new ways of working around the defenses of these systems; the systems evolve to become better at defending against the

new attacks, and so on. On the positive side, many security systems work quite well; there wouldn't be nearly as much e-commerce on the Web as there is were it not for the effectiveness of Transport Layer Security and all the cryptographic methods on which it depends.

One phrase that has been used to describe a vision for a future Internet is "An Internet deserving of society's trust." It is clear that realizing that vision is a significant challenge and that securing networks will be an area of research and innovation for some time to come.

FURTHER READING

The first two security-related papers, taken together, give a good overview of the topic. The article by Lampson et al. contains a formal treatment of security, while the Satyanarayanan paper gives a nice description of how a secure system is designed in practice. The third paper is a thorough and somewhat alarming overview of how worms and viruses spread and how a well-planned attack could be sped up.

- Lampson, B. et al. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10(4):265–310, November 1992.
- Satyanarayanan, M. Integrating security in a large distributed system. *ACM Transactions on Computer Systems* 7(3):247–280, August 1989.
- Staniford, S., V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. *USENIX Security Symposium 2002*, pp. 149–167. San Francisco, CA, August 2002.

There are several good books covering the full gamut of network security. We recommend Schneier [Sch95], Stallings [Sta03], and Kaufman et al. [KPS02]. The first two give comprehensive treatments of the topic, while the last gives a very readable overview of the subject. The full IPSec architecture is defined in a series of RFCs: [Ken05a], [Eas05], [MG98a], [MG98b], [MD98], [Ken05b], and [Kau05]. The Open PGP standard is defined in [Cal07], and the latest TLS standard is [DR08]. A book by Barrett and Silverman [Bar01] gives a thorough description of SSH. Menezes et al. [MvOV96] is a comprehensive cryptography reference (a copy can be freely downloaded from the URL listed below).

A discussion of the problem of recognizing and defending against denial-of-service attacks can be found in Moore et al. [MVS01], Spatscheck and Peterson [SP99], and Qiexh et al. [QPP02]. Recent techniques used to identify the source of attacks can be found in papers by Savage et al. [SWKA00] and Snoeren et al. [SPS+01]. The increasing threat of DDoS attacks is discussed by Garber [Gar00] and Harrison [Har00], and early approaches to defending against such attacks are reported in a paper by Park and Lee [PL01]. A novel approach to DoS prevention which falls in the "clean slate" category is the TVA approach by Yang et al. [YWA08].

Finally, we recommend the following live references:

- http://www.cert.org/: The website of CERT, an organization focused on computer security issues.
- http://www.cacr.math.uwaterloo.ca/hac/: Downloadable copy of [MvOV96] a comprehensive cryptography reference.

EXERCISES

1. Find or install an encryption utility (e.g., the Unix des command or pgp) on your system. Read its documentation and experiment with it. Measure how fast it is able to encrypt and decrypt data. Are these two rates the same? Try to compare these timing results using different key sizes; for example, compare AES with triple-DES.
2. Diagram cipher block chaining as described in Section 8.1.1.
3. Learn about a key escrow, or key surrender, scheme (for example, Clipper). What are the pros and cons of key escrow?
4. A good cryptographic hashing algorithm should produce random outputs; that is, the probability of any given hash value should be approximately the same as any other for randomly chosen input data. What would be the consequence of using a hash algorithm whose outputs were not random? Consider, for example, the case where some hash values are twice as likely to occur as others.
5. Suppose Alice uses the Needham–Schroeder authentication protocol described in Section 8.3.3 to initiate a session with Bob. Further suppose that an adversary is able to eavesdrop on the

authentication messages and, long after the session has completed, discover the (unencrypted) session key. How could the adversary deceive Bob into authenticating the adversary as Alice?

6. One mechanism for resisting replay attacks in password authentication is to use *one-time passwords*: A list of passwords is prepared, and once $password[N]$ has been accepted the server decrements N and prompts for $password[N-1]$ next time. At $N = 0$ a new list is needed. Outline a mechanism by which the user and server need only remember one master password mp and have available locally a way to compute $password[N] = f(mp, N)$. Hint: Let g be an appropriate one-way function (e.g., MD5) and let $password[N] = g^N(mp) = g$ applied N times to mp. Explain why knowing $password[N]$ doesn't help reveal $password[N-1]$.

7. Suppose a user employs one-time passwords as above (or, for that matter, reusable passwords), but that the password is transmitted sufficiently slowly.
   (a) Show that an eavesdropper can gain access to the remote server with a relatively modest number of guesses. (Hint: The eavesdropper starts guessing after the original user has typed all but one character of the password.)
   (b) To what other attacks might a user of one-time passwords be subject?

8. The Diffie–Hellman key exchange protocol is vulnerable to a "man-in-the-middle" attack as shown in Section 8.3.4 and Figure 8.12. Outline how Diffie–Hellman can be extended to protect against this possibility.

9. Suppose we have a very short secret s (e.g., a single bit or even a Social Security number), and we wish to send someone else a message m now that will not reveal s but that can be used later to verify that we did know s. Explain why $m = \mathrm{MD5}(s)$ or $m = \mathrm{E}(s)$ with RSA encryption would not be secure choices, and suggest a better choice.

10. Suppose two people want to play poker over the network. To deal the cards they need a mechanism for fairly choosing a random number x between them; each party stands to lose if the other

party can unfairly influence the choice of x. Describe such a mechanism. Hint: You may assume that if either of two bit strings x_1 and x_2 are random, then the exclusive-OR $x = x_1 \oplus x_2$ is random.

11. Estimate the probabilities of finding two messages with the same MD5 checksum, given total numbers of messages of 2^{63}, 2^{64}, and 2^{65}. Hint: This is the Birthday Problem again, as in Exercise 48 in Chapter 2, and again the probability that the $k+1$th message has a different checksum from each of the preceding k is $1 - k/2^{128}$. However, the approximation in the hint there for simplifying the product fails rather badly now. So, instead, take the log of each side and use the approximation $\log(1 - k/2^{128}) \approx -k/2^{128}$.

12. Suppose we wanted to encrypt a Telnet session with, say, 3DES. Telnet sends lots of 1-byte messages, while 3DES encrypts in blocks of 8 bytes at a time. Explain how 3DES might be used securely in this setting.

13. Consider the following simple UDP protocol (based loosely on TFTP, *Request for Comments* 1350) for downloading files:

- Client sends a file request.
- Server replies with first data packet.
- Client sends ACK, and the two proceed using stop-and-wait.

Suppose client and server possess keys K_C and K_S, respectively, and that these keys are known to each other.

(a) Extend the file downloading protocol, using these keys and MD5, to provide sender authentication and message integrity. Your protocol should also be resistant to replay attacks.

(b) How does the extra information in your revised protocol protect against the arrival of late packets from prior connection incarnations and sequence number wraparound?

14. Using the browser of your choice, find out what certification authorities for HTTPS your browser is configured by default to trust. Do you trust these agencies? Find out what happens when you disable trust of some or all of these certification authorities.

15. Use an OpenPGP implementation such as GnuPG to do the following. Note that no email is involved—you are working exclusively with files on a single machine.

(a) Generate a public–private key pair.

(b) Use your public key to encrypt a file, as if for secure storage, and then use your private key to decrypt it.

(c) Use your key pair to digitally sign an unencrypted file and then, as if you were someone else, verify your signature using your public key.

(d) Consider the first public–private key pair as belonging to Alice, and generate a second public–private key pair, for Bob. Playing the role of Alice, encrypt and sign a file intended for Bob. (Be sure to sign as Alice, not Bob.) Then, playing the role of Bob, verify Alice's signature and decrypt the file.

16. Consider a certification hierarchy as described in Section 8.2.1. A root CA signs a certificate for a second-tier CA, and the second-tier CA signs a certificate for Alice. Bob has the public key for the root CA, so he can verify the certificate of the second-tier CA. Why might Bob still not trust that the certificate for Alice truly establishes Alice as the owner of the public key in the certificate?

17. PuTTY (pronounced "putty") is a popular free SSH client—an application that implements the client side of SSH connections—for Unix and Windows. Its documentation is accessible on the Web.

(a) How does PuTTY handle authentication of a server that it has not previously connected to?

(b) How are clients authenticated to servers?

(c) PuTTY supports several ciphers. How does it determine which one to use for a particular connection?

(d) PuTTY supports ciphers, such as DES, that might be considered too weak for some—or any—situations. Why? How does PuTTY determine which ciphers are weak, and how does it use that information?

(e) For a given connection, PuTTY lets a user specify a maximum amount of time and/or transmitted data after which PuTTY will initiate the establishment of a new session key, which the

documentation refers to as a *key exchange* or *rekeying*. What is the motivation behind this feature?

(f) Use PuTTYgen, the PuTTY key generator, to generate a public–private key pair for one of the PuTTY-supported public key ciphers.

18. Suppose you want your firewall to block all incoming Telnet connections but to allow outbound Telnet connections. One approach would be to block all inbound packets to the designated Telnet port (23).

(a) We might want to block inbound packets to other ports as well, but what inbound TCP connections *must* be permitted in order not to interfere with outbound Telnet?

(b) Now suppose your firewall is allowed to use the TCP header Flags bits in addition to the port numbers. Explain how you can achieve the desired Telnet effect here while at the same time allowing no inbound TCP connections.

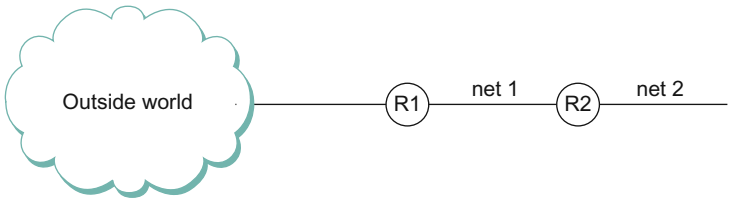
19. Suppose a firewall is configured to allow outbound TCP connections but inbound connections only to specified ports. The FTP protocol now presents a problem: When an inside client contacts an outside server, the outbound TCP control connection can be opened normally but the TCP data connection traditionally is inbound.

(a) Look up the FTP protocol in, for example, *Request for Comments* 959. Find out how the PORT command works. Discuss how the client might be written so as to limit the number of ports to which the firewall must grant inbound access. Can the number of such ports be limited to one?

(b) Find out how the FTP PASV command can be used to solve this firewall problem.

20. Suppose filtering routers are arranged as in Figure 8.21; the primary firewall is R1. Explain how to configure R1 and R2 so that outsiders can Telnet to net 2 but not to hosts on net 1. To avoid "leapfrogging" break-ins to net 1, also disallow Telnet connections from net 2 to net 1.

21. Why might an Internet Service Provider want to block certain *outbound* traffic?



■ **FIGURE 8.21** Diagram for Exercise 18.

22. It is said that IPsec may not work with Network Address Translation (NAT) (RFC 1631). However, whether IPsec will work with NAT depends on which mode of IPsec and NAT we use. Suppose we use true NAT, where only IP addresses are translated (without port translation). Will IPsec and NAT work in each of the following cases? Explain why or why not.

(a) IPsec uses ESP transport mode.

(b) IPsec uses ESP tunnel mode.

(c) What if we use PAT (Port Address Translation), also known as Network Address/Port Translation (NAPT) in NAT, where in addition to IP addresses port numbers will be translated to share one IP address from outside the private network?


A SYSTEMS APPROACH
A SYSTEMS APPROACH
A SYSTEMS APPROACH
A SYSTEMS APPROACH
A SYSTEMS APPROACH

9 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 building block 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.

PROBLEM: APPLICATIONS NEED THEIR OWN PROTOCOLS

Looking at applications drives home the *systems approach* that we have emphasized throughout this book. That is, the best way to build effective networked applications is to understand the building blocks that a network can provide and how those blocks can interact with each other. Thus, for example, a particular networked application might need to make use of a

Computer Networks: A Systems Approach. DOI: 10.1016/B978-0-12-385059-1.00009-0

reliable transport protocol, authentication and privacy mechanisms, and resource allocation capabilities of the underlying network. Applications often work best when the application developer knows how to make the best use of these facilities (and there are also plenty of counter-examples of applications making poor use of available networking capabilities). Applications typically need their own protocols, too, in many cases using the same principles that we have seen in our prior examination of lower layer protocols. Thus, our focus in this chapter is on how to put together the ideas and techniques already described to build effective networked applications. Said another way, if you ever imagine yourself writing a network application, then you will by definition also become a protocol designer (and implementer).

We proceed by examining a variety of familiar, and not so familiar, network applications. These range from exchanging email and surfing the Web, to integrating applications across businesses, to multimedia applications like vic and vat, to managing a set of network elements, to emerging peer-to-peer and content distribution networks. This list is by no means exhaustive, but it does serve to illustrate many of the key principles of designing and building applications. Applications need to pick and choose the appropriate building blocks that are available at other layers either inside the network or in the host protocol stacks and then augment those underlying services to provide the precise communication service required by the application.

9.1 TRADITIONAL APPLICATIONS

We begin our discussion of applications by focusing on two of the most popular—the World Wide Web and email. Broadly speaking, both of these applications use 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 (although the Web is a lot newer than email but has its roots in file transfers that predated it). By contrast, later sections will look 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. (Note that there is a bit of a blurring between these classes, as you can of course get access to streaming multimedia data over the Web, but for now we'll focus on the general usage of the Web to request pages, images, etc.)

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. Many different application programs—that is, web clients like Internet Explorer, Chrome, Firefox, and Safari—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. Indeed, it is the fact that the protocol is published and standardized that enables application programs developed by many different companies and individuals to interoperate. That is how so many browsers are able to interoperate with all the web servers (of which there are also many varieties).

This section looks at two very widely used, standardized application protocols:

- Simple Mail Transfer Protocol (SMTP) is used to exchange electronic mail.
- HyperText Transport Protocol (HTTP) is used to communicate between web browsers and web servers.

We'll also look at how custom application protocols are defined in the *Web Services* architecture.

The second point is that, since the application protocols described in this section follow the same request/reply communication pattern, you might expect that they would be built on top of a Remote Procedure Call (RPC) transport protocol. This is not the case, however, as they are instead implemented on top of TCP. In effect, each protocol reinvents a simple RPC-like mechanism on top of a reliable transport protocol (TCP). We say "simple" because each protocol is not designed to support arbitrary remote procedure calls of the sort discussed in Section 5.3, but is instead designed to send and respond to a specific set of request messages.

Finally, we observe that many application layer protocols, including HTTP and SMTP, have a companion protocol that specifies the format of the data that can be exchanged. This is one reason WHY 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 and Multipurpose Internet Mail Extensions (MIME) define the format of email messages. Similarly, HTTP

is a protocol for fetching Web pages, but HyperText Markup Language (HTML) is a companion specification that defines the basic form of those pages.

9.1.1 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 cross-country link you just managed to get running? Surprisingly, 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 useful application that continues to be extremely popular.

As noted above, it is important (1) to distinguish the user interface (i.e., your mail reader) from the underlying message transfer protocols (such as SMTP or IMAP), 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 quite a few times since then) to allow email messages to carry many different types of data: audio, video, images, PDF 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 (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 who still 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

For many years, the majority of email was moved from host to host using only SMTP. While SMTP continues to play a central role, it is now just one email protocol of several, Internet Message Access Protocol (IMAP) and Post Office Protocol (POP) being two other important protocols for retrieving mail messages. We'll begin our discussion by looking at SMTP, and move on to IMAP below.

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. Countless mail readers are available, just like there are many web browsers to choose from. In the early days of the Internet, users typically logged into the machine on which their *mailbox* resided, and the mail reader they invoked was a local application program that extracted messages from the file system. Today, of course, users remotely access their mailbox from their laptop or smartphone; they do not first log into the host that stores their mail (a mail server). A second mail transfer protocol, such as POP or IMAP, is used to remotely download email from a mail server to the user's device.

Second, there is a *mail daemon* (or process) running on each host that holds a mailbox. You can think of this process, also called a *message transfer agent* (MTA), as playing the role of a post office: Users (or their 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 them). 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 there are only a few popular

implementations, with the old sendmail program from Berkeley Unix and postfix being the most widespread.

While it is certainly possible that the MTA on a sender's machine establishes an SMTP/TCP connection to the MTA on the recipient's mail server, 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 message transfer agent 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 9.1 illustrates a two-hop path from the sender to the receiver.

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. Another is scale: In large organizations, it's often the case that a number of different machines hold the *mailboxes* for the organization. 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 connection—to the specific machine on which Bob has a mailbox. The forwarding gateway maintains a database that maps users into the machine on which their mailbox resides; the

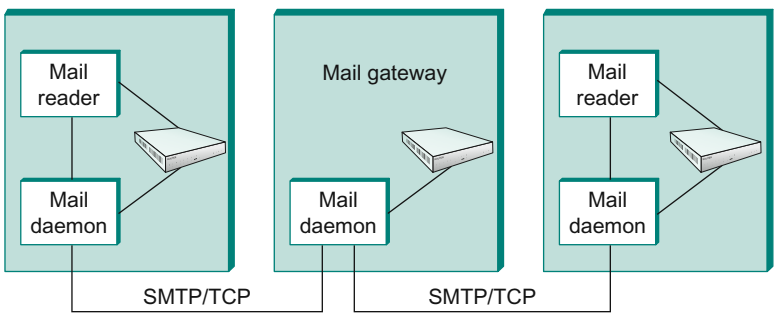


FIGURE 9.1 Sequence of mail gateways store and forward email messages.

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.) Yet another reason, particularly true in the early days of email, is that the machine that hosts any given user's mailbox may not always be up or reachable, 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 Cisco. The lines sent by cs.princeton.edu are shown in black and the lines sent by cisco.com are shown in teal. 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. As pointed out earlier, this reader was originally just a program running on the same machine as the user's mailbox, in which case it could simply read and write the file that implements the mailbox. This was the common case in the pre-laptop era. Today, most often the user accesses his or her mailbox from a remote machine using yet another protocol, such as POP or 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 9.2. 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 and, 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.

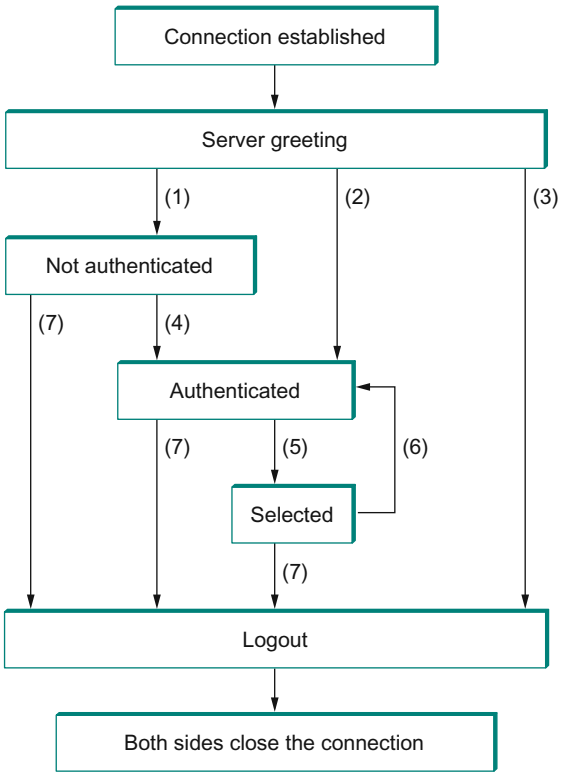


FIGURE 9.2 IMAP state transition diagram.

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 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.

9.1.2 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. In fact, the design of the system that became the Web started around 1989, long after the Internet had become a widely deployed system. The original goal of the Web was to find a way to organize and retrieve information, drawing on ideas about hypertext—interlinked documents—that had been around since at least the 1960s.[1] The core idea of hypertext is that one document can link to another document, and the protocol (HTTP) and document language (HTML) were designed to meet that goal.

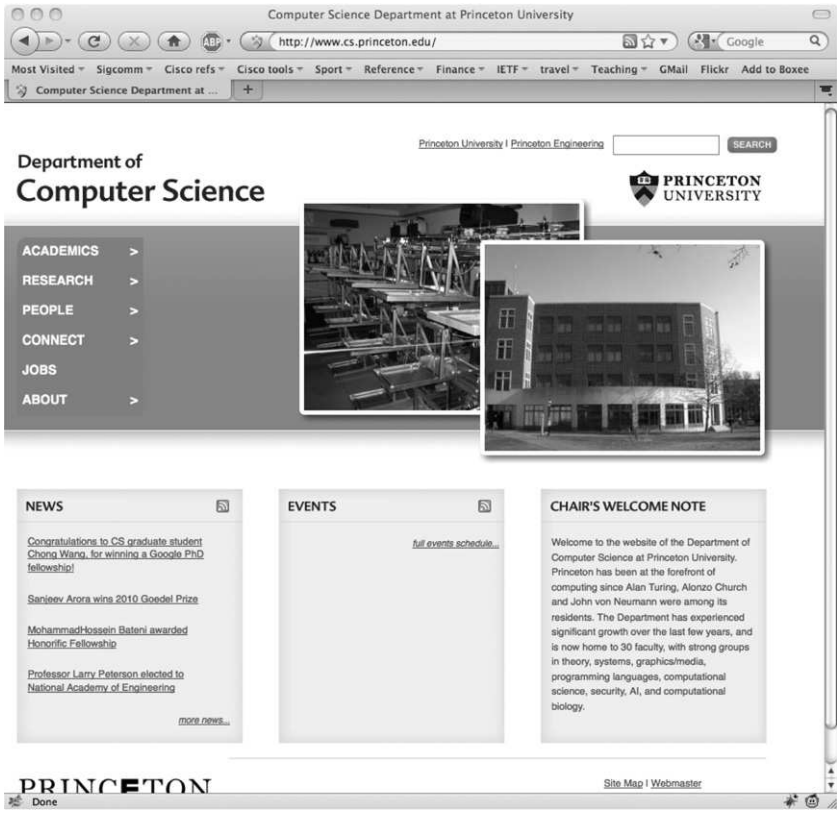
One helpful way to think of the Web is as a set of cooperating clients and servers, all of whom speak the same language: HTTP. Most people are exposed to the Web through a graphical client program or web browser like Safari, Chrome, Firefox, or Internet Explorer. Figure 9.3 shows the Firefox browser in use, displaying a page of information from Princeton University.

Clearly, if you want to organize information into a system of linked documents or objects, you need to be able to retrieve one document to get started. Hence, any web browser has a function that allows the user to obtain an object by opening a URL. Uniform Resource Locators (URLs) are so familiar to most of us by now that it's easy to forget that they haven't been around forever. They provide information that allows objects on the Web to be located, and 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 many have other objects such as audio and video clips, pieces of code, etc. They also frequently include URLs that point to other files that may be located on other machines, which is the core of the "hypertext" part of HTTP and HTML. A web browser has some way in which you can recognize URLs (often by highlighting or underlining some text) and then you can 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

[1] A short history of the Web provided by the World Wide Web consortium traces its roots to a 1945 article describing links between microfiche documents.



■ **FIGURE 9.3** The Firefox web browser.

the network, following links to all sorts of information. Once you have a means to embed a link in a document and allow a user to follow that link to get another document, you have the basis of a hypertext system.

When you ask your browser 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, HTTP is a request/response protocol, where every 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 specifies 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 where a server would place the requested page when responding to a request, and it is typically empty for request messages.

Why does HTTP run over TCP? The designers didn't have to do it that way, but TCP does provide a pretty good match to what HTTP needs, particularly by providing reliable delivery (who wants a Web page with missing data?), flow control, and congestion control. However, as we'll see below, there are a few issues that can arise from building a request/response protocol on top of TCP, especially if you ignore the subtleties of the interactions between the application and transport layer protocols.

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 modified since the browser last fetched it. The full set of operations is summarized in Table 9.1. As innocent as it sounds, the POST command enables much mischief (including spam) on the Internet.

Table 9.1 HTTP Request Operations

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

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
```

Table 9.2 Five Types of HTTP Result Codes

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

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.

As with the unexpected consequences of the POST request message, it is sometimes surprising how various response messages are used in practice. For example, request redirection (specifically code 302) turns out to be a powerful mechanism that plays a big role in Content Distribution Networks (CDNs) (see Section 9.4.3) by redirecting requests to a nearby cache.

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 9.1.1). 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).

Uniform Resource Identifiers

The URLs that HTTP uses as addresses are one type of *Uniform Resource Identifier* (URI). A URI is a character string that identifies a resource, where a resource can be anything that has identity, such as a document, an image, or a service.

The format of URIs allows various more specialized kinds of resource identifiers to be incorporated into the URI space of identifiers. The first part of a URI is a *scheme* that names a particular way of identifying a certain kind of resource, such as mailto for email addresses or file for file names. The second part of a URI, separated from the first part by a colon, is the *scheme-specific part*. It is a resource identifier consistent with the scheme in the first part, as in the URIs

```
mailto:santa@northpole.org
```

and

```
file:///C:/foo.html
```

A resource doesn't have to be retrievable or accessible. We saw an example of this earlier in Section 7.1.3—extensible markup language (XML) namespaces are identified by URIs that look an awful lot like URLs, but strictly speaking they are not *locators* because they don't tell you how to locate something; they just provide a globally unique identifier for the namespace. There is no requirement that you can retrieve anything at the URI given as the target namespace of an XML document. We'll see another example of a URI that is not a URL in Section 9.2.1.

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.

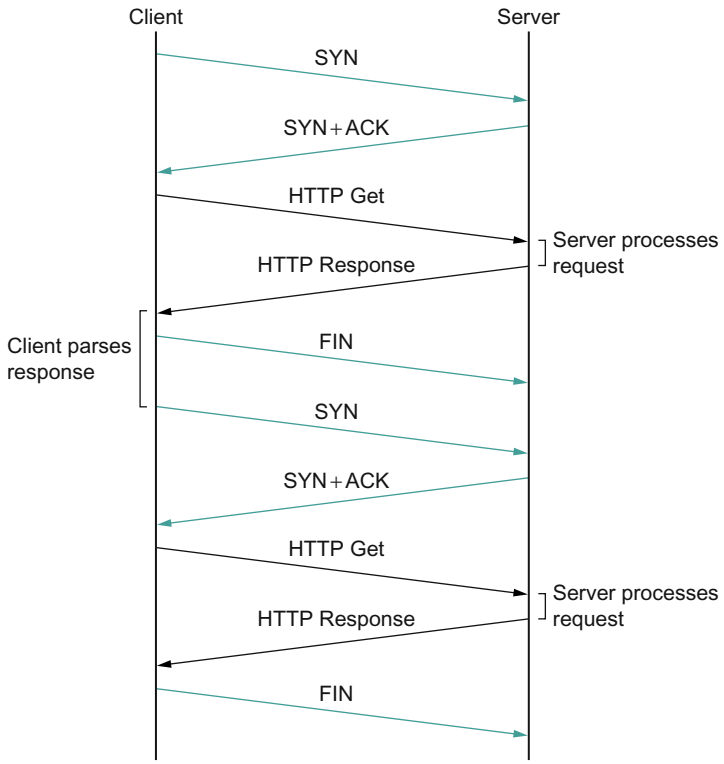


FIGURE 9.4 HTTP 1.0 behavior.

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. Figure 9.4 shows the sequence of events for fetching a page that has just a single embedded object. Colored lines indicate TCP messages, while black lines indicate the HTTP requests and responses. (Some of the TCP ACKs are not shown to avoid cluttering the picture.) You can see two round trip times are spent setting up TCP connections while another two (at least) are spent getting the page and image. As well as the latency impact, there is also processing cost on the server to handle the extra TCP connection establishment and termination.

To overcome this situation, HTTP version 1.1 introduced *persistent connections*—the client and server can exchange multiple request/response messages over the same TCP connection. Persistent connections have many advantages. First, they obviously eliminate the

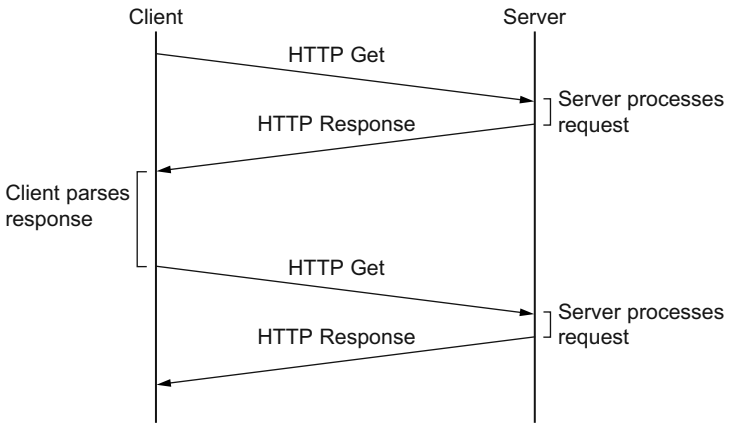


FIGURE 9.5 HTTP 1.1 behavior with persistent connections.

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. Figure 9.5 shows the transaction from Figure 9.4 using a persistent connection in the case where the connection is already open (presumably due to some prior access of the same server).

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.) Concerns about this added complexity may be one reason why persistent connections were not used from the outset, but today it is widely accepted that the benefits of persistent connections more than offset the drawbacks.

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 site-wide cache. This allows users to take advantage of pages previously downloaded by other users. Closer to the middle of the Internet, Internet Service Providers (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.[2]

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

[2]There are quite a few issues with this sort of caching, ranging from the technical to the regulatory. One example of a technical challenge is the effect of *asymmetric paths*, when the request to the server and the response to the client do not follow the same sequence of router hops.

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. We'll look at the related issue of CDNs—which are effectively distributed caches—in Section 9.4.3.

9.1.3 Web Services

So far we have focused on interactions between a human and a machine. For example, a human uses a web browser to interact with a server, and the interaction proceeds in response to input from the user (e.g., by clicking on links). However, there is increasing demand for direct computer-to-computer interaction. And, just as the applications described above need protocols, so too do the applications that communicate directly with each other. We conclude this section by looking at the challenges of building large numbers of application-to-application protocols and some of the proposed solutions.

Much of the motivation for enabling direct application-to-application communication comes from the business world. Historically, interactions between enterprises—businesses or other organizations—have involved some manual steps such as filling out an order form or making a phone call to determine whether some product is in stock. Even within a single enterprise it is common to have manual steps between software systems that cannot interact directly because they were developed independently. Increasingly, such manual interactions are being replaced with direct application-to-application interaction. An ordering application at enterprise A would send a message to an order fulfillment application at enterprise B, which would respond immediately indicating whether the order can be filled. Perhaps, if the order cannot be filled by B, the application at A would immediately order from another supplier or solicit bids from a collection of suppliers.

Here is a simple example of what we are talking about. Suppose you buy a book at an online retailer like Amazon.com. Once your book has been shipped, Amazon could send you the tracking number in an email, and then you could head over to the website for the shipping company—http://www.fedex.com, perhaps—and track the package. However, you can also track your package directly from the Amazon.com website. In order to make this happen, Amazon has to be able to send a query to FedEx,

in a format that FedEx understands, interpret the result, and display it in a Web page that perhaps contains other information about your order. Underlying the user experience of getting all the information about the order served up at once on the Amazon.com Web page is the fact that Amazon and FedEx had to have a protocol for exchanging the information needed to track packages—call it the Package Tracking Protocol. It should be clear that there are so many potential protocols of this type that we'd better have some tools to simplify the task of specifying them and building them.

Network applications, even those that cross organization boundaries, are not new—email and web browsing cross such boundaries. What is new about this problem is the scale. Not scale in the size of the network, but scale in the number of different kinds of network applications. Both the protocol specifications and the implementations of those protocols for traditional applications like electronic mail and file transfer have typically been developed by a small group of networking experts. To enable the vast number of potential network applications to be developed quickly, it was necessary to come up with some technologies that simplify and automate the task of application protocol design and implementation.

Two architectures have been advocated as solutions to this problem. Both architectures are called *Web Services*, taking their name from the term for the individual applications that offer a remotely accessible service to client applications to form network applications.[3] The terms used as informal shorthand to distinguish the two Web Services architectures are *SOAP* and *REST* (as in, "the SOAP vs. REST debate"). We will discuss the technical meanings of those terms shortly.

The SOAP architecture's approach to the problem is to make it feasible, at least in theory, to generate protocols that are customized to each network application. The key elements of the approach are a framework for protocol specification, software toolkits for automatically generating protocol implementations from the specifications, and modular partial specifications that can be reused across protocols.

The REST architecture's approach to the problem is to regard individual Web Services as World Wide Web resources—identified by URIs and accessed via HTTP. Essentially, the REST architecture is just the Web

[3]The name *Web Services* is unfortunately so generic sounding that many mistakenly assume that it includes any sort of service associated with the Web.

architecture. The Web architecture's strengths include stability and a demonstrated scalability (in the network-size sense). It could be considered a weakness that HTTP is not well suited to the usual procedural or operation-oriented style of invoking a remote service. REST advocates argue, however, that rich services can nonetheless be exposed using a more data-oriented or document-passing style for which HTTP is well suited.

Although both architectures are being actively adopted, they are still new enough that we don't yet have much empirical data about their real-world use. One architecture may come to dominate, or they may merge in some way, or we may find that one architecture is better suited to certain kinds of applications while the other architecture is better for others.

Custom Application Protocols (WSDL, SOAP)

The architecture informally referred to as SOAP is based on *Web Services Description Language* (WSDL) and *SOAP*.[4] Both of these standards are issued by the World Wide Web Consortium (W3C). This is the architecture that people usually mean when they use the term Web Services without any preceding qualifier. As these standards are still evolving, our discussion here is effectively a snapshot.

WSDL and SOAP are frameworks for specifying and implementing application protocols and transport protocols, respectively. They are generally used together, although that is not strictly required. WSDL is used to specify application-specific details such as what operations are supported, the formats of the application data to invoke or respond to those operations, and whether an operation involves a response. SOAP's role is to make it easy to define a transport protocol with exactly the desired semantics regarding protocol features such as reliability and security.

Both WSDL and SOAP consist primarily of a protocol specification language. Both languages are based on XML (Section 7.1.3) with an eye toward making specifications accessible to software tools such as stub compilers and directory services. In a world of many custom protocols, support for automating generation of implementations is crucial to avoid the effort of manually implementing each protocol. Support software generally takes the form of toolkits and application servers developed

[4] Although the name *SOAP* originated as an acronym, it officially no longer stands for anything.

by third-party vendors, which allows developers of individual Web Services to focus more on the business problem they need to solve (such as tracking the package purchased by a customer).

Defining Application Protocols

WSDL has chosen a procedural *operation* model of application protocols. An abstract Web Service interface consists of a set of named operations, each representing a simple interaction between a client and the Web Service. An operation is analogous to a remotely callable procedure in an RPC system. An example from W3C's WSDL Primer is a hotel reservation Web Service with two operations, CheckAvailability and MakeReservation.

Each operation specifies a *Message Exchange Pattern* (MEP) that gives the sequence in which the messages are to be transmitted, including the fault messages to be sent when an error disrupts the message flow. Several MEPs are predefined, and new custom MEPs can be defined, but it appears that in practice only two MEPs are being used: **In-Only** (a single message from client to service) and **In-Out** (a request from client and a corresponding reply from service). These patterns should be very familiar, and suggest that the costs of supporting MEP flexibility perhaps outweigh the benefits.

MEPs are templates that have placeholders instead of specific message types or formats, so part of the definition of an operation involves specifying which message formats to map into the placeholders in the pattern. Message formats are not defined at the bit level that is typical of protocols we have discussed. They are instead defined as an abstract data model using XML Schema (Section 7.1.3). XML Schema provides a set of primitive data types and ways to define compound data types. Data that conforms to an XML Schema-defined format—its abstract data model—can be concretely represented using XML, or it can use another representation, such as the "binary" representation Fast Infoset.

WSDL nicely separates the parts of a protocol that can be specified abstractly—operations, MEPs, abstract message formats—from the parts that must be concrete. WSDL's concrete part specifies an underlying protocol, how MEPs are mapped onto it, and what bit-level representation is used for messages on the wire. This part of a specification is known as a *binding*, although it is better described as an implementation, or a mapping onto an implementation. WSDL has predefined bindings for HTTP and SOAP-based protocols, with parameters that allow the

protocol designer to fine-tune the mapping onto those protocols. There is a framework for defining new bindings, but SOAP protocols dominate.

A crucial aspect of how WSDL mitigates the problem of specifying large numbers of protocols is through reuse of what are essentially specification modules. The WSDL specification of a Web Service may be composed of multiple WSDL documents, and individual WSDL documents may also be used in other Web Service specifications. This modularity makes it easier to develop a specification and easier to ensure that, if two specifications are supposed to have some elements that are identical (for example, so that they can be supported by the same tool), then those elements are indeed identical. This modularity, together with WSDL's defaulting rules, also helps keep specifications from becoming overwhelmingly verbose for human protocol designers.

WSDL modularity should be familiar to anyone who has developed moderately large pieces of software. A WSDL document need not be a complete specification; it could, for example, define a single message format. The partial specifications are uniquely identified using XML Namespaces (Section 7.1.3); each WSDL document specifies the URI of a *target namespace*, and any new definitions in the document are named in the context of that namespace. One WSDL document can incorporate components of another by *including* the second document if both share the same target namespace or *importing* it if the target namespaces differ.

Defining Transport Protocols

Although SOAP is sometimes called a protocol, it is better thought of as a framework for defining protocols. As the SOAP 1.2 specification explains, "SOAP provides a simple messaging framework whose core functionality is concerned with providing extensibility." SOAP uses many of the same strategies as WSDL, including message formats defined using XML Schema, bindings to underlying protocols, Message Exchange Patterns, and reusable specification elements identified using XML namespaces.

SOAP is used to define transport protocols with exactly the features needed to support a particular application protocol. SOAP aims to make it feasible to define many such protocols by using reusable components. Each component captures the header information and logic that go into implementing a particular feature. To define a protocol with a certain set of features, just compose the corresponding components. Let's look more closely at this aspect of SOAP.

SOAP 1.2 introduced a *feature* abstraction, which the specification describes thus: *A SOAP feature is an extension of the SOAP messaging framework. Although SOAP poses no constraints on the potential scope of such features, example features may include "reliability," "security," "correlation," "routing," and message exchange patterns (MEPs) such as request/response, one-way, and peer-to-peer conversations.* A SOAP feature specification must include:

- A URI that identifies the feature
- The state information and processing, abstractly described, that is required at each SOAP node to implement the feature
- The information to be relayed to the next node
- (If the feature is a MEP) the life cycle and temporal/causal relationships of the messages exchanged—for example, responses follow requests and are sent to the originator of the request

Note that this formalization of the concept of a protocol feature is rather low level; it is almost a design.

Given a set of features, there are two strategies for defining a SOAP protocol that will implement them. One is by layering: binding SOAP to an underlying protocol in such a way as to derive the features. For example, we could obtain a request/response protocol by binding SOAP to HTTP, with a SOAP request in an HTTP request and a SOAP reply in an HTTP response. Because this is such a common example, it happens that SOAP has a predefined binding to HTTP; new bindings may be defined using the SOAP Protocol Binding Framework.

The second and more flexible way to implement features involves *header blocks*. A SOAP message consists of an Envelope, which contains a Header that contains header blocks, and a Body, which contains the payload destined for the ultimate receiver. This message structure is illustrated in Figure 9.6.

It should be a familiar notion by now that certain header information corresponds to particular features. A digital signature is used to implement authentication, a sequence number is used for reliability, and a checksum is used to detect message corruption. A SOAP header block is intended to encapsulate the header information that corresponds to a particular feature. The correspondence is not always one-to-one since multiple header blocks could be involved in a single feature, or a single

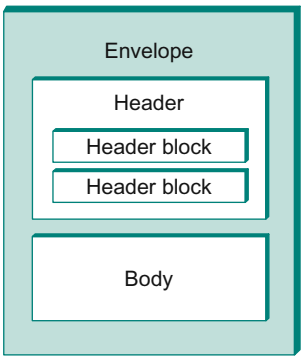


FIGURE 9.6 SOAP message structure.

header block could be used in multiple features. A *SOAP module* is a specification of the syntax and the semantics of one or more header blocks. Each module is intended to provide one or more features and must declare the features it implements.

The goal behind SOAP modules is to be able to compose a protocol with a set of features by simply including each of the corresponding module specifications. If your protocol is required to have at-most-once semantics and authentication, include the corresponding modules in your specification. This represents a novel approach to modularizing protocol services, an alternative to the protocol layering we have seen throughout this book. It is bit like flattening a series of protocol layers into a single protocol, but in a structured way. It remains to be seen how well SOAP features and modules, introduced in version 1.2 of SOAP, will work in practice. The main weakness of this scheme is that modules may well interfere with each other. A module specification is required to specify any *known* interactions with other SOAP modules, but clearly that doesn't do much to alleviate the problem. On the other hand, a core set of features and modules that provides the most important properties may be small enough to be well known and well understood.

Standardizing Web Services Protocols

As we've said, WSDL and SOAP aren't protocols; they are standards for *specifying* protocols. For different enterprises to implement Web Services that interoperate with each other, it is not enough to agree to use WSDL and SOAP to define their protocols; they must agree on—standardize—specific protocols. For example, you could imagine that online retailers

and shipping companies might like to standardize a protocol by which they exchange information, along the lines of the simple package tracking example at the start of this section. This standardization is crucial for tool support as well as interoperability. And, yet, different network applications in this architecture must necessarily differ in at least the message formats and operations they use.

This tension between standardization and customization is tackled by establishing partial standards called *profiles*. A profile is a set of guidelines that narrow or constrain choices available in WSDL, SOAP, and other standards that may be referenced in defining a protocol. They may at the same time resolve ambiguities or gaps in those standards. In practice, a profile often formalizes an emerging *de facto* standard.

The broadest and most widely adopted profile is known as the *WS-I Basic Profile*. It was proposed by the Web Services Interoperability Organization (WS-I), an industry consortium, while WSDL and SOAP are specified by the World Wide Web Consortium (W3C). The Basic Profile resolves some of the most basic choices faced in defining a Web Service. Most notably it requires that WSDL be bound exclusively to SOAP and SOAP be bound exclusively to HTTP and use the HTTP POST method. It also specifies which versions of WSDL and SOAP must be used.

The *WS-I Basic Security Profile* adds security constraints to the Basic Profile by specifying how the SSL/TLS layer (Section 8.4.3) is to be used and requiring conformance to *WS-Security* (Web Services Security). WS-Security specifies how to use various existing techniques such as X.509 public key certificates (Section 8.2.1) and Kerberos (Section 8.3.3) to provide security features in SOAP protocols.

WS-Security is just the first of a growing suite of SOAP-level standards established by the industry consortium OASIS (Organization for the Advancement of Structured Information Standards). The standards known collectively as *WS-** include WS-Reliability, WS-ReliableMessaging, WS-Coordination, and WS-AtomicTransaction.

A Generic Application Protocol (REST)

The WSDL/SOAP Web Services architecture is based on the assumption that the best way to integrate applications across networks is via protocols that are customized to each application. That architecture is designed to make it practical to specify and implement all those protocols. In contrast, the REST Web Services architecture is based on the assumption that the best way to integrate applications across networks is by re-applying the

model underlying the World Wide Web architecture (Section 9.1.2). This model, articulated by Web architect Roy Fielding, is known as *REpresentational State Transfer* (REST). There is no need for a new REST architecture for Web Services—the existing Web architecture is sufficient, although a few extensions are probably necessary. In the Web architecture, individual Web Services are regarded as resources identified by URIs and accessed via HTTP—a single generic application protocol with a single generic addressing scheme.

Where WSDL has user-defined operations, REST uses the small set of available HTTP methods, such as GET and POST (see Table 9.1). So how can these simple methods provide an interface to a rich Web Service? By employing the REST model, in which the complexity is shifted from the protocol to the payload. The payload is a representation of the abstract state of a resource. For example, a GET could return a representation of the current state of the resource, and a POST could send a representation of a desired state of the resource.

The representation of a resource state is abstract; it need not resemble how the resource is actually implemented by a particular Web Service instance. It is not necessary to transmit a complete resource state in each message. The size of messages can be reduced by transmitting just the parts of a state that are of interest (e.g., just the parts that are being modified). And, because Web Services share a single protocol and address space with other web resources, parts of states can be passed by reference—by URI—even when they are other Web Services.

This approach is best summarized as a data-oriented or document-passing style, as opposed to a procedural style. Defining an application protocol in this architecture consists of defining the document structure (i.e., the state representation). XML and the lighter-weight JavaScript Object Notation (JSON) are the most frequently used presentation languages (Section 7.1) for this state. Interoperability depends on agreement, between a Web Service and its client, on the state representation. Of course, the same is true in the SOAP architecture; a Web Service and its client have to be in agreement on payload format. The difference is that in the SOAP architecture interoperability additionally depends on agreement on the protocol; in the REST architecture, the protocol is always HTTP, so that source of interoperability problems is eliminated.

One of the selling features of REST is that it leverages the infrastructure that has been deployed to support the Web. For example, Web proxies

can enforce security or cache information. Existing content distribution networks (CDNs) can be used to support RESTful applications.

In contrast with WSDL/SOAP, the Web has had time for standards to stabilize and to demonstrate that it scales very well. It also comes with some security in the form of Secure Socket Layer (SSL)/Transport Layer Security (TLS). The Web and REST may also have an advantage in evolvability. Although the WSDL and SOAP *frameworks* are highly flexible with regard to what new features and bindings can go into the definition of a protocol, that flexibility is irrelevant once the protocol is defined. Standardized protocols such as HTTP are designed with a provision for being extended in a backward-compatible way. HTTP's own extensibility takes the form of headers, new methods, and new content types. Protocol designers using WSDL/SOAP need to design such extensibility into each of their custom protocols. Of course, the designers of state representations in a REST architecture also have to design for evolvability.

An area where WSDL/SOAP may have an advantage is in adapting or wrapping previously written, "legacy" applications to conform to Web Services. This is an important point since most Web Services will be based on legacy applications for the near future at least. These applications usually have a procedural interface that maps more easily into WSDL's operations than REST states. The REST versus WSDL/SOAP competition may very well hinge on how easy or difficult it turns out to be to devise REST-style interfaces for individual Web Services. We may find that some Web Services are better served by WSDL/SOAP and others by REST.

The online retailer Amazon.com, as it happens, was an early adopter (2002) of Web Services. Interestingly, Amazon made its systems publicly accessible via *both* of the Web Services architectures, and according to some reports a substantial majority of developers use the REST interface. Of course, this is just one data point and may well reflect factors specific to Amazon.

9.2 MULTIMEDIA APPLICATIONS

Just like the traditional applications described earlier in this chapter, multimedia applications such as telephony and videoconferencing need their own 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, an overlay network that supports IP multicast to enable multiparty conferencing. (More on overlay networks including the MBone in the next section.) 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 a number of protocols that multimedia applications use. The Real-Time Transport Protocol (RTP; described in Section 5.4) provides many of the functions that are common to multimedia applications such as conveying timing information and identifying the coding schemes and media types of an application.

The Resource Reservation Protocol (RSVP; see Section 6.5.2) 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. We will see how resource allocation interacts with other aspects of multimedia applications in Section 9.2.2.

In addition to these protocols for multimedia transport and resource allocation, many multimedia applications also need a signalling or *session control* protocol. For example, suppose that we wanted to be able to make telephone calls across the Internet (Voice over IP, or VoIP). We would need some mechanism to notify the intended recipient of such a call that we wanted to talk to her, such as 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, etc. The Session Initiation Protocol (SIP) and H.323 are examples of protocols that address the issues of session control; we begin our discussion of multimedia applications by examining these protocols.

9.2.1 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 defined protocols for just this purpose. The protocols that have been defined include

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

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 widely used for a number of interactive applications such as Internet telephony.

Session Description Protocol (SDP)

The Session Description Protocol (SDP) is a rather general protocol that can be used in a variety of situations and is typically used in conjunction with one or more other protocols (e.g., SIP). 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 required 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 scheme)

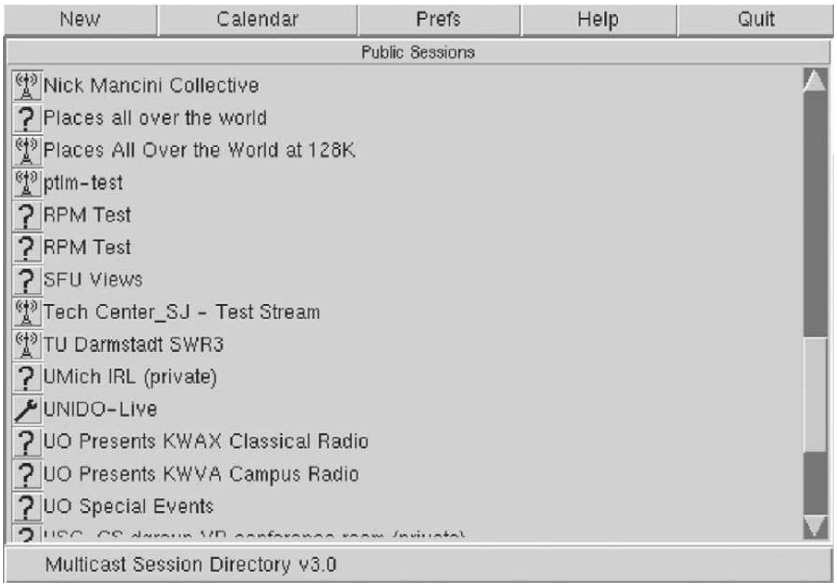
SDP provides this information formatted in ASCII using a sequence of lines of text, each of the form "<type>=<value>." An example of an SDP message will illustrate the main points.

```
v=0
o=larry 2890844526 2890842807 IN IP4 128.112.136.10
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 128.112.136.10 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, as described in Section 9.1.2)—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



■ **FIGURE 9.7** A session directory tool displays information extracted from SDP messages.

session. Figure 9.7 shows a screen shot of a (somewhat archaic) 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 transported using "RTP/AVP." This means that they run over RTP and

use the *application profile* (as defined in Section 5.4) 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 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 9.7 would function by joining that multicast group and displaying information that it gleans from received SDP messages. SDP is also used in the delivery of entertainment video of IP (often called IPTV) to provide information about the video content on each TV channel.

SDP also plays an important role in conjunction with the Session Initiation Protocol (SIP). With the widespread adoption of Voice over IP (i.e., the support of telephony-like applications over IP networks) and IP-based video conferencing, SIP is now one of the more important members of the Internet protocol suite.

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 just to 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, such as 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 9.8. The first thing to notice is that each user has a name in the format user@domain, very much like an email

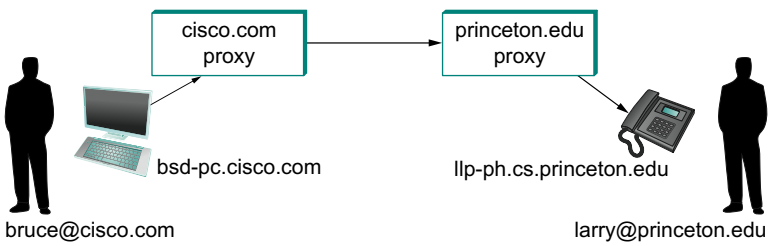


FIGURE 9.8 Establishing communication through SIP proxies.

address. When user 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 which look like this:

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

A SIP URI provides complete identification of a user, but (unlike a URL) 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 mobile phone 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:

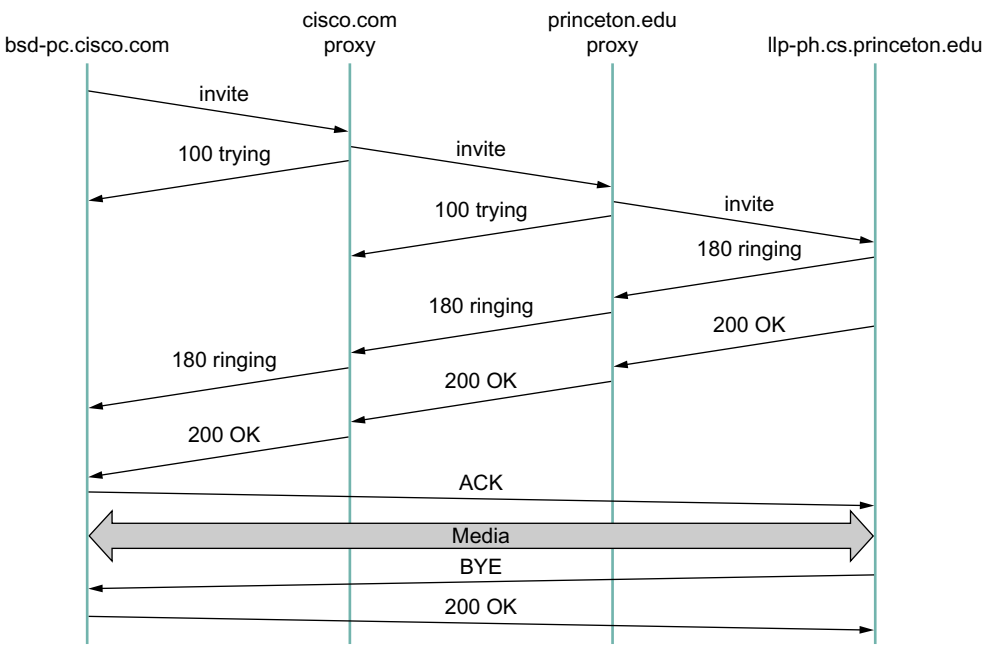
```
INVITE sip:larry@princeton.edu SIP/2.0
Via: SIP/2.0/UDP bsd-pc.cisco.com;branch=z9hG4bK433yte4
To: Larry <sip:larry@princeton.edu>
From: Bruce <sip:bruce@cisco.com>;tag=55123
Call-ID: xy745jj210re3@bsd-pc.cisco.com
CSeq: 271828 INVITE
Contact: <sip:bruce@bsd-pc.cisco.com>
Content-Type: application/sdp
Content-Length: 142
```

The first line identifies the type of function to be performed (invite); the resource on which to perform it, 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 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, not only does the proxy forward the message on toward princeton.edu, but 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 9.9 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



■ **FIGURE 9.9** Message flow for a basic SIP session.

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 media (e.g., an RTP-encapsulated audio stream) can now begin to flow between the two parties. Note that at this point the parties know each others' addresses, so the ACK can be sent directly, bypassing the proxies. The proxies are now 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.

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 to 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. (We'll discuss how DNS can do this in Section 9.3.1.) 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 voice-mail server. It is also easy to see how it could be used for applications like instant messaging, and standardization of SIP extensions for such purposes is ongoing at the time of writing.

H.323

The International Telecommunication Union (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, including video calls, and we consider that class of 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 telephone-like 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

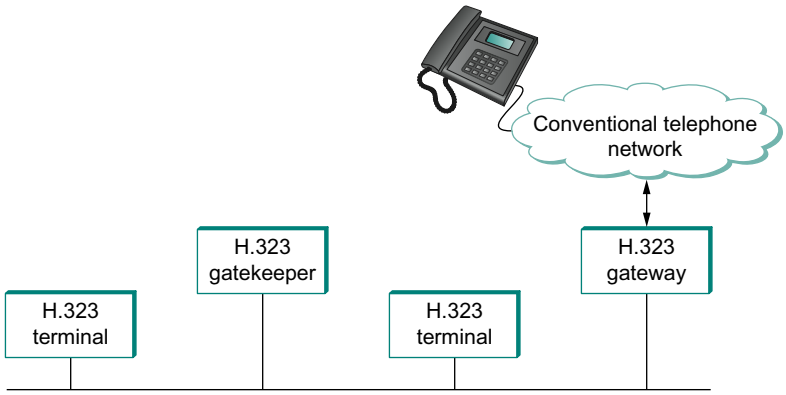


FIGURE 9.10 Devices in an H.323 network.

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 9.10. 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 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. H.245 messages might list a number of different audio codec standards that it can support; 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 Real-Time Control Protocol (RTCP) for the media stream (or streams—a call might include both audio and video, for example) for 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.

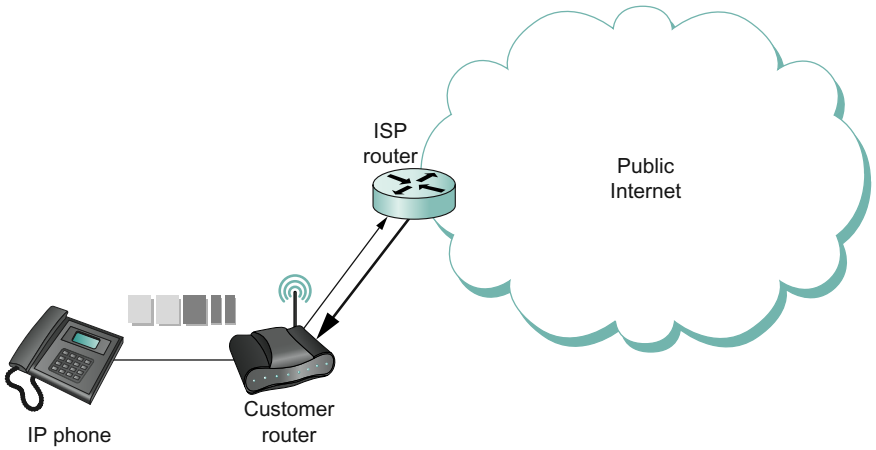
9.2.2 Resource Allocation for Multimedia Applications

As we have just seen, session control protocols like SIP and H.323 can be used to initiate and control communication in multimedia applications, while RTP provides transport-level functions for the data streams of the applications. A final piece of the puzzle in getting multimedia applications to work is making sure that suitable resources are allocated inside the network to ensure that the quality of service needs of the application are met. We presented a number of methods for resource allocation in Chapter 6. The motivation for developing these technologies was largely for the support of multimedia applications. So how do applications take advantage of the underlying resource allocation capabilities of the network?

It is worth noting that many multimedia applications run successfully over "best-effort" networks, such as the public Internet. The wide array of commercial VOIP services (such as Skype) are testimony to the fact that you only have to worry about resource allocation when resources are not abundant—and in many parts of today's Internet, resource abundance is the norm.

A protocol like RTCP (Section 5.4) can help applications in best-effort networks, by giving the application detailed information about the quality of service that is being delivered by the network. Recall that RTCP carries information about the loss rate and delay characteristics between participants in a multimedia application. An application can use this information to change its coding scheme—changing to a lower bitrate codec, for example, when bandwidth is scarce. Note that, while it might be tempting to change to a codec that sends additional, redundant information when loss rates are high, this is frowned upon; it is analogous to *increasing* the window size of TCP in the presence of loss, the exact opposite of what is required to avoid congestion collapse.

As discussed in Section 6.5.3, Differentiated Services (DiffServ) can be used to provide fairly basic and scalable resource allocation to applications. A multimedia application can set the differentiated services code point (DSCP) in the IP header of the packets that it generates in an effort to ensure that both the media and control packets receive appropriate quality of service. For example, it is common to mark voice media packets as "EF" (expedited forwarding) to cause them to be placed in a low-latency or priority queue in routers along the path, while the call signalling (e.g., SIP) packets are often marked with some sort of



■ **FIGURE 9.11** Differentiated Services applied to a VOIP application. DiffServ queueing is applied only on the upstream link from customer router to ISP.

"AF" (assured forwarding) to enable them to be queued separately from best-effort traffic and thus reduce their risk of loss.

Of course, it only makes sense to mark the packets inside the sending host or appliance if network devices such as routers pay attention to the DSCP. In general, routers in the public Internet ignore the DSCP, providing best-effort service to all packets. However, enterprise or corporate networks have the ability to use DiffServ for their internal multimedia traffic, and frequently do so. Also, even residential users of the Internet can often improve the quality of VOIP or other multimedia applications just by using DiffServ on the outbound direction of their Internet connections, as illustrated in Figure 9.11. This is effective because of the asymmetry of many broadband Internet connections: If the outbound link is substantially slower (i.e., more resource constrained) than the inbound, then resource allocation using DiffServ on that link may be enough to make all the difference in quality for latency- and loss-sensitive applications.

While DiffServ is appealing for its simplicity, it is clear that it cannot meet the needs of applications under all conditions. For example, suppose the upstream bandwidth in Figure 9.11 is only 100 kbps, and the customer attempts to place two VOIP calls, each with a 64-kbps codec. Clearly the upstream link is now more than 100% loaded, which will lead to large queueing delays and lost packets. No amount of clever queueing in the customer's router can fix that.

The characteristics of many multimedia applications are such that, rather than try to squeeze too many calls into a too-narrow pipe, it would be better to block one call while allowing another to proceed. That is, it is better to have one person carrying on a conversation successfully while another hears a busy signal than to have both callers experiencing unacceptable audio quality at the same time. We sometimes refer to such applications as having a *steep utility curve*, meaning that the utility (usefulness) of the application drops rapidly as the quality of service provided by the network degrades. Multimedia applications often have this property, whereas many traditional applications do not. Email, for example, continues to work quite well even if delays run into the hours.

Applications with steep utility curves are often well suited to some form of admission control. If you cannot be sure that sufficient resources will always be available to support the offered load of the applications, then admission control provides a way to say "no" to some applications while allowing others to get the resources they need.

We saw one way to do admission control using RSVP in Section 6.5.2, and we will return to that shortly, but multimedia applications that use session control protocols provide some other admission control options. The key point to observe here is that session control protocols like SIP or H.323 often involve some sort of message exchange between an endpoint and another entity (SIP proxy or H.323 gatekeeper) at the beginning of a call or session. This can provide a handy means to say "no" to a new call for which sufficient resources are not available.

As an example, consider the network in Figure 9.12. Suppose the wide area link from the branch office to the head office has enough bandwidth to accommodate three VOIP calls simultaneously using 64-kbps codecs. Each phone already needs to communicate with the local SIP proxy or H.323 gatekeeper when it begins to place a call, so it is easy enough for the proxy/gatekeeper to send back a message that tells the IP phone to play a busy signal if that link is already fully loaded. The proxy or gatekeeper can even deal with the possibility that a particular IP phone might be making multiple calls at the same time and that different codec speeds might be used. However, this scheme will work only if no other device can overload the link without first talking to the gatekeeper or proxy. DiffServ queueing can be used to ensure that, for example, a PC engaged in a file transfer doesn't interfere with the VOIP calls. But, suppose some VOIP application that doesn't first talk to the gatekeeper or proxy is enabled in the

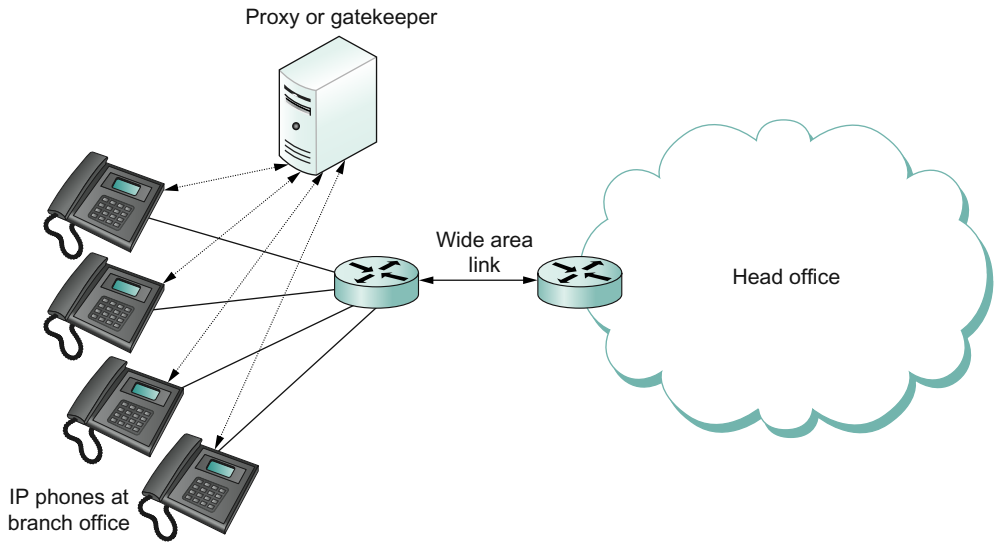


FIGURE 9.12 Admission control using session control protocol.

remote office. Such an application, if it can get its packets marked appropriately and in the same queue as the existing VOIP traffic, can clearly drive the link to the point of overload with no feedback from the proxy or gatekeeper.

Another problem with the approach just described is that it depends on the gatekeeper or proxy having knowledge of the path that each application will use. In the simple topology of Figure 9.12 this isn't a big issue, but in more complex networks it can quickly become unmanageable. We only need to imagine the case where the remote office has two different connections to the outside world to see that we are asking the proxy or gatekeeper to understand not just SIP or H.323 but also routing, link failures, and current network conditions. This can quickly become unmanageable.

We refer to the sort of admission control just described as *off-path*, in the sense that the device making admission control decisions does not sit on the data path where resources need to be allocated. The obvious alternative is *on-path* admission control, and the standard example of a protocol that does on-path admission control in IP networks is the Resource Reservation Protocol (RSVP). We saw in Section 6.5.2 how RSVP can be used to ensure that sufficient resources are allocated along a path,

and it is straightforward to use RSVP in applications like those described in this section. The one detail that still needs to be filled in is how the admission control protocol interacts with the session control protocol.

Coordinating the actions of an admission control (or resource reservation) protocol and a session control protocol is not rocket science, but it does require some attention to details. As an example, consider a simple telephone call between two parties. Before you can make a reservation, you need to know how much bandwidth the call is going to use, which means you need to know what codecs are to be used. That implies you need to do some of the session control first, to exchange information about the codecs supported by the two phones. However, you can't do *all* the session control first, because you wouldn't want the phone to ring before the admission control decision had been made, in case admission control failed. Figure 9.13 illustrates this situation where SIP is used for session control and RSVP is used to make the admission control decision (successfully in this case).

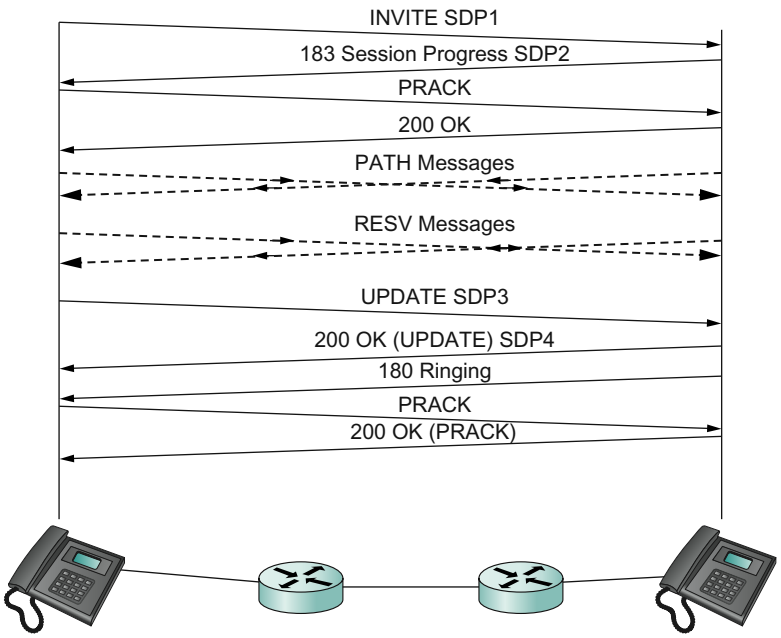


FIGURE 9.13 Co-ordination of SIP signalling and resource reservation.

The main thing to notice here is the interleaving of session control and resource allocation tasks. Solid lines represent SIP messages, dashed lines represent RSVP messages. Note that SIP messages are transmitted direction from phone to phone in this example (i.e., we have not shown any SIP proxies), whereas the RSVP messages are also processed by the routers in the middle as the check for sufficient resources to admit the call.

We being with an initial exchange of codec information in the first two SIP messages (recall that SDP is used to list available codecs, among other things). PRACK is a "provisional acknowledgment." Once these messages have been exchanged, RSVP PATH messages, which contain a description of the amount of resources that will be required, can be sent as the first step in reserving resources in both directions of the call. Next, RESV messages can be sent back to actually reserve the resources. Once a RESV is received by the initiating phone, it can send an updated SDP message reporting the fact that resources have been reserved in one direction. When the called phone has received both that message and the RESV from the other phone, it can start to ring and tell the other phone that resources are now reserved in both directions (with the SDP message) and also notify the calling phone that it is ringing. From here on, normal SIP signalling and media flow, similar to that shown in Figure 9.9, proceeds.

Again we see how building applications requires us to understand the interaction between different building blocks (SIP and RSVP, in this case). The designers of SIP actually made some changes to the protocol to enable this interleaving of functions between protocols with different jobs, hence our repeated emphasis in this book on focusing on complete systems rather than just looking at one layer or component in isolation from the other parts of the system.

9.3 INFRASTRUCTURE SERVICES

There are some protocols that are essential to the smooth running of the Internet but that don't fit neatly into the strictly layered model. One of these is the Domain Name System (DNS)—not an application that users normally invoke explicitly, but rather a service that almost all other applications depend upon. This is because the name service 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 service is usually used by other applications, rather than by humans.

A second critical function is network management, which although not so familiar to the average user, is the operation performed most often by system administrators. Network management is widely considered one of the hard problems of networking and continues to be the focus of much research. We'll look at some of the issues and approaches to the problem below.

9.3.1 Name Service (DNS)

In most of this book, 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. Already in this section we have seen application protocols like HTTP using names such as www.princeton.edu. We now describe how a naming service can be developed to map user-friendly names into router-friendly addresses. 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 *hierarchical* (Unix file names are an 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.

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 namespace 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 9.14. While this picture makes the name resolution task look simple enough, there is a bit more to it, as we shall see.

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

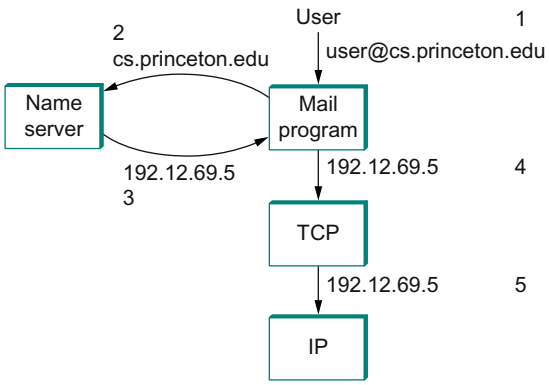


FIGURE 9.14 Names translated into addresses, where the numbers 1 to 5 show the sequence of steps in the process.

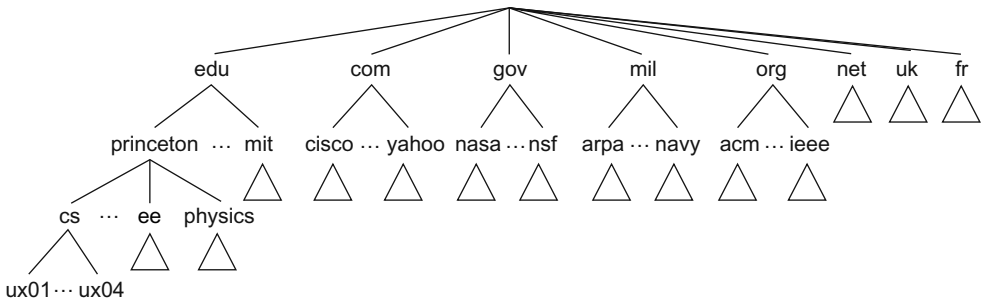


FIGURE 9.15 Example of a domain hierarchy.

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.

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 9.15 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.[5]

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 were all originally based in the United States (where the Internet and DNS were invented); for example, only U.S.-accredited educational institutions can register an .edu domain name. In recent years, the number of top-level domains has been expanded, partly to deal with the high demand for .com domains names. The newer top-level domains include .biz, .coop, and .info. Another recent development has been the support of domain names that are represented in character sets other than the Latin alphabet, such as Arabic and Chinese.

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*. Figure 9.16 shows how the hierarchy given in Figure 9.15 might

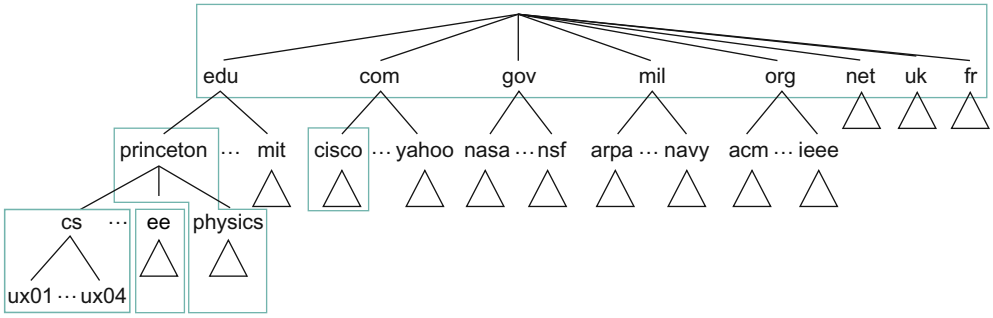
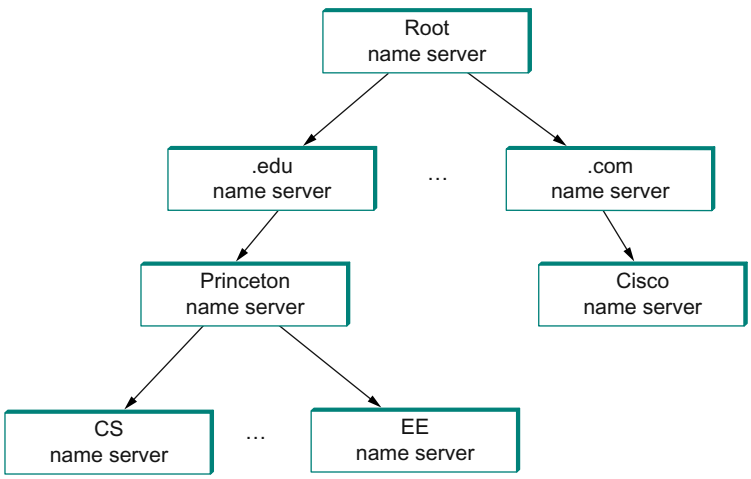


FIGURE 9.16 Domain hierarchy partitioned into zones.

[5]Confusingly, the word *domain* is also used in Internet routing, where it means something different than it does in DNS, being roughly equivalent to the term *autonomous system*.

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 Internet Corporation for Assigned Names and Numbers (ICANN). 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.

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 9.17.



■ **FIGURE 9.17** Hierarchy of name servers.

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 time-to-live (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 9.15. To simplify the example, we ignore the TTL field and we give the relevant information for only one of the name servers that implement each zone.

First, a root name server contains an NS record for each top-level domain (TLD) name server. This identifies a server that can resolve

queries for this part of the DNS hierarchy (.edu and .com in this example). It also has A records that translates these names into the corresponding IP addresses. Taken together, these two records effectively implement a pointer from the root name server to one of the TLD servers.

⟨edu, a3.nstld.com, NS, IN⟩

⟨a3.nstld.com, 192.5.6.32, A, IN⟩

⟨com, a.gtld-servers.net, NS, IN⟩

⟨a.gtld-servers.net, 192.5.6.30, A, IN⟩

⋮

Moving our way down the hierarchy by one level, the a3.nstld.com server has records for .edu domains like this:

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

⟨dns.princeton.edu, 128.112.129.15, A, IN⟩

⋮

In this case, we get an NS record and an A record for the name server that is responsible for the princeton.edu part of the hierarchy. That server might be able to directly resolve some queries (e.g., for email.princeton.edu) while it would redirect others to a server at yet another layer in the hierarchy (e.g., for a query about penguins.cs .princeton.edu).

⟨email.princeton.edu, 128.112.198.35, A, IN⟩

⟨penguins.cs.princeton.edu, dns1.cs.princeton.edu, NS, IN⟩

⟨dns1.cs.princeton.edu, 128.112.136.10, 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 coreweb.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—they allow an administrator to change which host receives mail on behalf of the domain without having to change everyone's email address.

⟨penguins.cs.princeton.edu, 128.112.155.166, A, IN⟩

⟨www.cs.princeton.edu, coreweb.cs.princeton.edu, CNAME, IN⟩

⟨coreweb.cs.princeton.edu, 128.112.136.35, A, IN⟩

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

⟨mail.cs.princeton.edu, 128.112.136.72, A, 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 the Lightweight Directory Access Protocol (LDAP). 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.

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 penguins.cs.princeton.edu relative to the set of servers given in the previous subsection. The client could first send a query containing this name to one of the root servers (as we'll see below, this rarely happens in practice but will suffice to illustrate the basic operation for now). The root server, unable to match the entire name, returns the best match it

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.

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. 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. This last one is so prevalent that many people do not even realize it is just a convention.

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.

has—the NS record for edu which points to the TLD server a3.nstld.com. The server also returns all records that are related to this record, in this case, the A record for a3.nstld.com. The client, having not received the answer it was after, next sends the same query to the name server at IP host 192.5.6.32. This server also cannot match the whole name and so returns the NS and corresponding A records for the princeton.edu domain. Once again, the client sends the same query as before to the server at IP host 128.112.129.15, and this time gets back the NS record and corresponding A record for the cs.princeton.edu domain. This time, the server that can fully resolve the query has been reached. A final query to the server at 128.112.136.10 yields the A record for penguins.cs.princeton.edu, and the client learns that the corresponding IP address is 128.112.155.166.

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, put 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, it 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 dns1.cs.princeton.edu. This local name server, in turn, has resource records for one or more of the root servers, for example:

⟨'root', a.root-servers.net, NS, IN⟩
⟨a.root-servers.net, 198.41.0.4, 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 9.18. 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. This caching mechanism can be used further up the hierarchy as well, reducing the load on the root and TLD servers.

The second question is how the system works when a user submits a partial name (e.g., penguins) rather than a complete domain name (e.g., penguins.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.

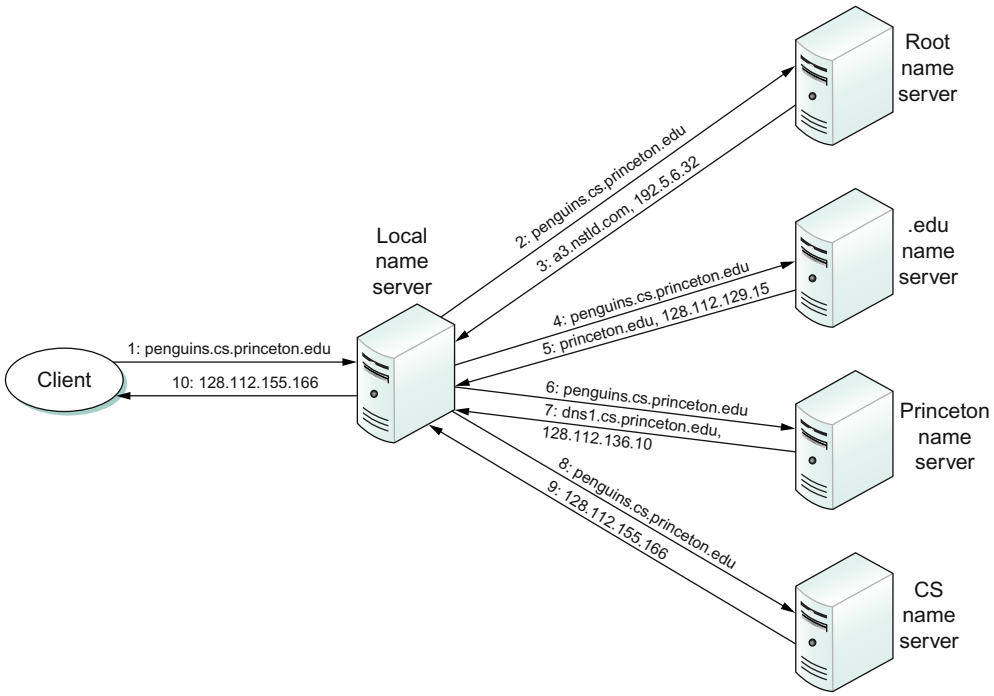


FIGURE 9.18 Name resolution in practice, where the numbers 1 to 10 show the sequence of steps in the process.

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 the Address Resolution Protocol (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.

9.3.2 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—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 and how many packets have been sent and received on each interface
- Address translation—Information about the Address Resolution Protocol, 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 Internet Control Message Protocol (ICMP), Exterior Gateway Protocol (EGP), and SNMP itself. The tenth 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 Abstract Syntax Notation One (ASN.1).

Consider the second problem first. As we already saw in Chapter 7, ASN.1/Basic Encoding Rules (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.

9.4 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 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 some underlying network. By this definition, the Internet started out as an overlay network on top of the links provided by the old telephone network. Figure 9.19 depicts an overlay implemented on top of an underlying network. Each node in the overlay also exists in the underlying network;

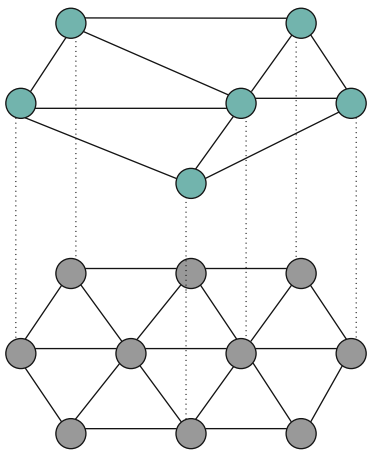


FIGURE 9.19 Overlay network layered on top of a physical network.

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 overlay networks can exist on top of the same underlying network—each implementing its 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 multi-hop path between them as a single logical link, the nodes that are tunneled through forward packets based on the outer header, never aware that the end nodes have attached an inner header. Figure 9.20 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 (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 header, 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.

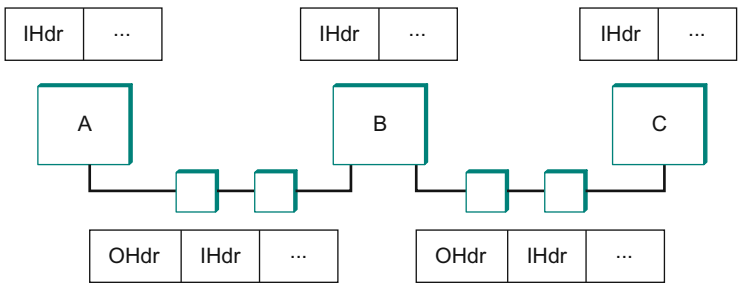


FIGURE 9.20 Overlay nodes tunnel through physical nodes.

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, as far back as 2001, a report from the National Research Council pointed 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 went on to observe that, at the same time, a whole new set of challenges were emerging that 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 current 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, 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.

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 algorithm as it does 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 many 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 probably 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 that have been built 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 (Section 4.2) started off as an extension to IP and even today is not enabled in many Internet routers. The MBone (multicast backbone) was an overlay network that implemented IP multicast on top of the unicast routing provided by the Internet. A number of multimedia conference tools were developed for and deployed on the Mbone. For example, IETF meetings—which are a week long and attract thousands of participants—were for many years broadcast over the MBone.

Like VPNs, the MBone used both IP tunnels and IP addresses, but unlike VPNs, the MBone implemented a different forwarding algorithm—forwarding 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 was a similar overlay that was used to incrementally deploy IPv6. Like the MBone, the 6-BONE used tunnels to forward packets through IPv4 routers. Unlike the MBone, however, 6-BONE nodes did not simply provide a new interpretation of IPv4's 32-bit addresses. Instead, they forwarded packets based on IPv6's 128-bit address space. The 6-BONE also supported IPv6 multicast.

End System Multicast

Although IP multicast is popular with researchers and certain segments of the networking community, its deployment in the global Internet has been limited at best. In response, multicast-based applications like video-conferencing 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.

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 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 9.21. Figure 9.21(a) depicts an example physical topology, where

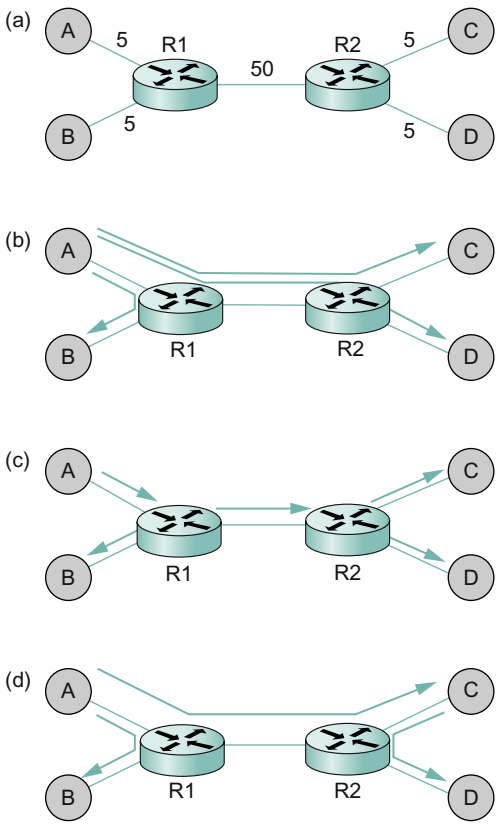


FIGURE 9.21 Alternative multicast trees mapped onto a physical topology.

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 message to the other three hosts, Figure 9.21(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 9.21(c) depicts the IP multicast tree constructed by the Distance Vector Multicast Routing Protocol (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 9.21(d). End system multicast defines an architecture for constructing this tree.

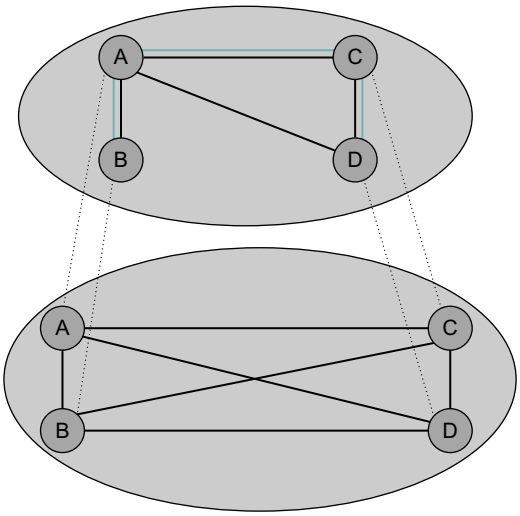


FIGURE 9.22 Multicast tree embedded in an overlay mesh.

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 9.22, 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 roundtrip latency to other nodes and 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 them 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 function that can be performed by an overlay is to find alternative routes for traditional unicast applications. Such overlays exploit the observation that the triangle inequality does not hold in the Internet. Figure 9.23 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.

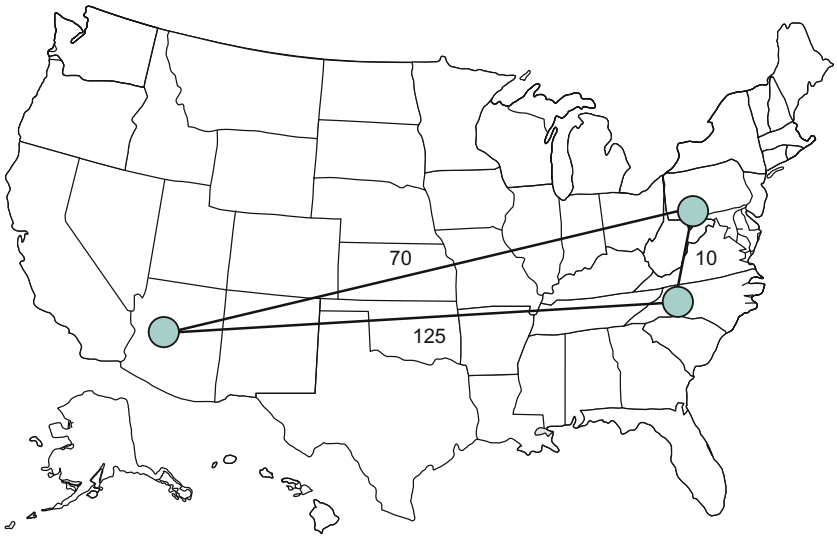


FIGURE 9.23 The triangle inequality does not necessarily hold in networks.

How can this be? Well, the Border Gateway Protocol (BGP) never promised that it would find the *shortest* route between any two sites; it only tries to find *some* route. To make matters more complex, BGP's routes are heavily influenced by policy issues, such as who is paying whom to carry their traffic. 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 tradeoff between the scalability and optimality of a routing algorithm. 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 the Resilient Overlay Network (RON), 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 is not designed to be a scalable approach, 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*.[6] 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 coexist 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

Music-sharing applications like Napster® and KaZaA 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,

[6]The term *virtualization* is used a lot these days in the context of data center computing to refer to the virtualization of servers using hypervisors and similar technologies but it's really a much broader concept.

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 a larger archival store, 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 was not a true peer-to-peer system by this definition since it depended on a central registry of known files, and users had to search this directory to find what machine offered a particular file. It was only the last step—actually downloading the file—that took 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 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 clearer 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

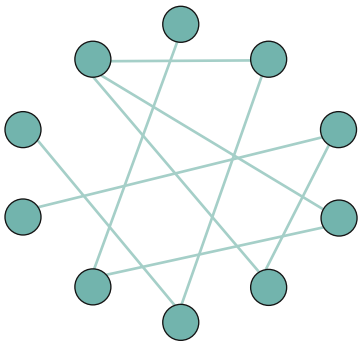


FIGURE 9.24 Example topology of a Gnutella peer-to-peer network.

registry of objects. Instead, Gnutella participants arrange themselves into an overlay network similar to the one shown in Figure 9.24. 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 that neighbor's response, respectively.

It should be clear that Gnutella as described here 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 propagated. 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 a request 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 an address 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
```

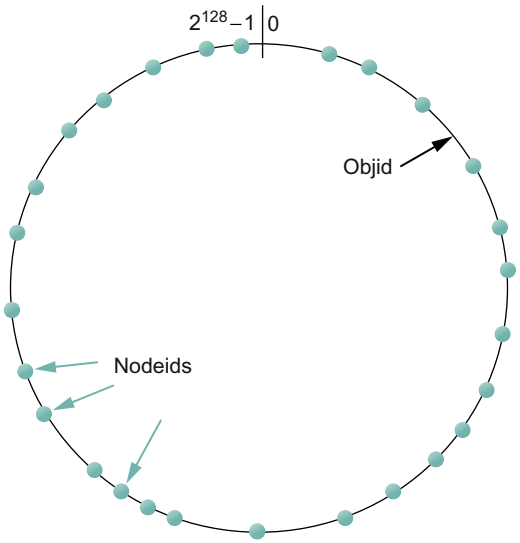
Unfortunately, if there are more than 101 nodes willing to host objects, then we can't take advantage of all of them. On the other hand, if we select a number larger than the largest possible number of nodes, then there will be some values of x that will hash into an address for a node that does not 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 9.25 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

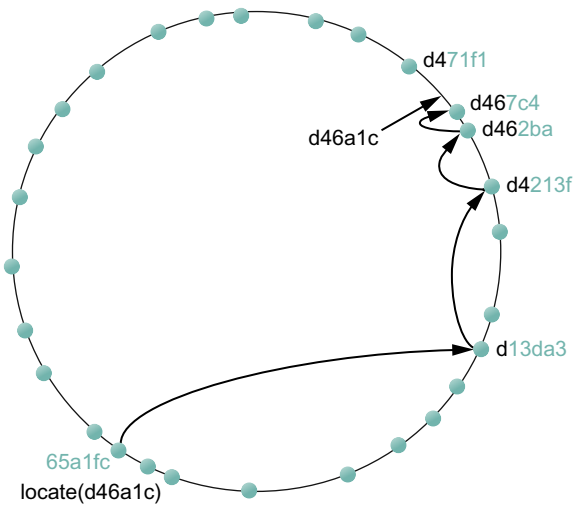
$$hash(IP\_addr) \longrightarrow nodeid$$



■ **FIGURE 9.25** 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 whose 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 that 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



■ **FIGURE 9.26** Objects are located by routing through the peer-to-peer overlay network.

sometimes called a *distributed hash table* (DHT), since conceptually, the hash table is distributed over all the nodes in the network.

Figure 9.26 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 a both 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 shared prefix
      and 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 9.27 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 9.28. It is through this routing process that the new node

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 9.27 Example routing table at the node with ID 65a1fcx.

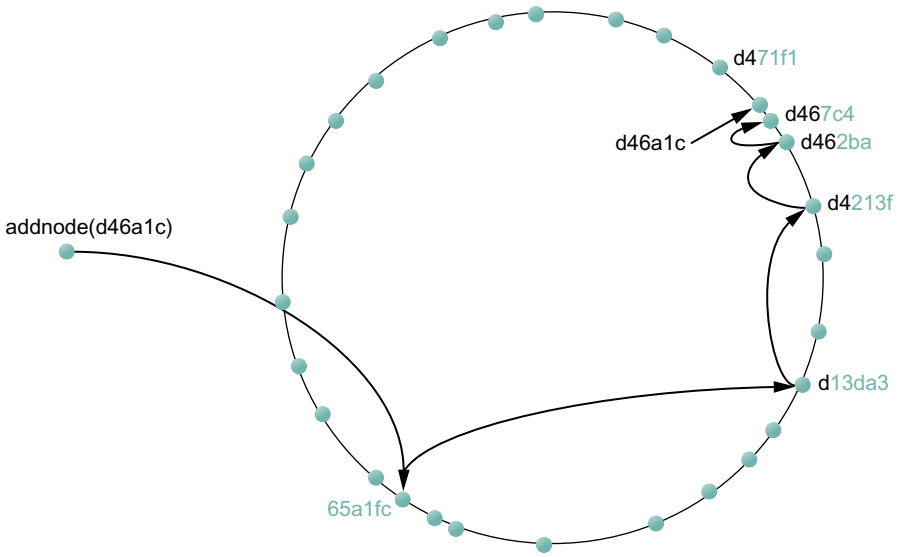


FIGURE 9.28 Adding a node to the network.

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 joined 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.

The reader 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 be at a random location in the Internet. (In the worst case, each node is on a different continent!) In fact, in a world-wide 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, one 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.

BitTorrent

BitTorrent is a peer-to-peer file sharing protocol devised by Bram Cohen. It is based on replicating the file or, rather, replicating segments of the file, which are called *pieces*. Any particular piece can usually be downloaded from multiple peers, even if only one peer has the entire file. The primary benefit of BitTorrent's replication is avoiding the bottleneck of having only one source for a file. This is particularly useful when you consider that any given computer has a limited speed at which it can serve files over its uplink to the Internet, often quite a low limit due to the asymmetric nature of most broadband networks. The beauty of BitTorrent is that replication is a natural side effect of the downloading process: As soon as a peer downloads a particular piece, it becomes another source for that piece. The more peers downloading pieces of the file, the more piece replication occurs, distributing the load proportionately, and the more total bandwidth is available to share the file with others. Pieces are downloaded in random order to avoid a situation where peers find themselves lacking the same set of pieces.

Each file is shared via its own independent BitTorrent network, called a *swarm*. (A swarm could potentially share a set of files, but we describe the single file case for simplicity.) The lifecycle of a typical swarm is as follows. The swarm starts as a singleton peer with a complete copy of the file. A node that wants to download the file joins the swarm, becoming its second member, and begins downloading pieces of the file from the original peer. In doing so, it becomes another source for the pieces it has downloaded, even if it has not yet downloaded the entire file. (In fact, it is common for peers to leave the swarm once they have completed their downloads, although they are encouraged to stay longer.) Other nodes join the swarm and begin downloading pieces from multiple peers, not just the original peer. See Figure 9.29.

If the file remains in high demand, with a stream of new peers replacing those who leave the swarm, the swarm could remain active indefinitely; if not, it could shrink back to include only the original peer until new peers join the swarm.

Now that we have an overview of BitTorrent, we can ask how requests are routed to the peers that have a given piece. To make requests, a would-be downloader must first join the swarm. It starts by downloading a .torrent file containing meta-information about the file and swarm. The .torrent file, which may be easily replicated, is typically downloaded

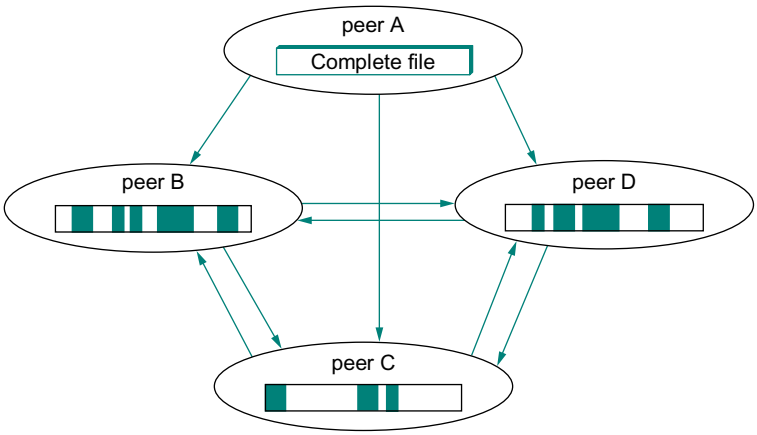


FIGURE 9.29 Peers in a BitTorrent swarm download from other peers that may not yet have the complete file.

from a web server and discovered by following links from Web pages. It contains:

- The target file's size
- The piece size
- SHA-1 hash values (Section 8.1.4) precomputed from each piece
- The URL of the swarm's *tracker*

A tracker is a server that tracks a swarm's current membership. We'll see later that BitTorrent can be extended to eliminate this point of centralization, with its attendant potential for bottleneck or failure.

The would-be downloader then joins the swarm, becoming a peer, by sending a message to the tracker giving its network address and a peer ID that it has generated randomly for itself. The message also carries a SHA-1 hash of the main part of the .torrent file, which is used as a swarm ID.

Let's call the new peer P. The tracker replies to P with a partial list of peers giving their IDs and network addresses, and P establishes connections, over TCP, with some of these peers. Note that P is directly connected to just a subset of the swarm, although it may decide to contact additional peers or even request more peers from the tracker. To establish a BitTorrent connection with a particular peer after their TCP connection has been established, P sends P's own peer ID and swarm ID, and the peer replies with its peer ID and swarm ID. If the swarm IDs don't match, or the reply peer ID is not what P expects, the connection is aborted.

The resulting BitTorrent connection is symmetric: Each end can download from the other. Each end begins by sending the other a bitmap reporting which pieces it has, so each peer knows the other's initial state. Whenever a downloader (D) finishes downloading another piece, it sends a message identifying that piece to each of its directly connected peers, so those peers can update their internal representation of D's state. This, finally, is the answer to the question of how a download request for a piece is routed to a peer that has the piece, because it means that each peer knows which directly connected peers have the piece. If D needs a piece that none of its connections has, it could connect to more or different peers (it can get more from the tracker) or occupy itself with other pieces in hopes that some of its connections will obtain the piece from their connections.

How are objects—in this case, pieces—mapped onto peer nodes? Of course each peer eventually obtains all the pieces, so the question is really about which pieces a peer has at a given time before it has all the pieces or, equivalently, about the order in which a peer downloads pieces. The answer is that they download pieces in random order, to keep them from having a strict subset or superset of the pieces of any of their peers.

The BitTorrent described so far utilizes a central tracker that constitutes a single point of failure for the swarm and could potentially be a performance bottleneck. Also, providing a tracker can be a nuisance for someone who would like to make a file available via BitTorrent. Newer versions of BitTorrent additionally support "trackerless" swarms that use a DHT-based implementation. BitTorrent client software that is trackerless capable implements not just a BitTorrent peer but also what we'll call a *peer finder* (the BitTorrent terminology is simply *node*), which the peer uses to find peers.

Peer finders form their own overlay network, using their own protocol over UDP to implement a DHT. Furthermore, a peer finder network includes peer finders whose associated peers belong to different swarms. In other words, while each swarm forms a distinct network of BitTorrent peers, a peer finder network instead spans swarms.

Peer finders randomly generate their own finder IDs, which are the same size (160 bits) as swarm IDs. Each finder maintains a modest table containing primarily finders (and their associated peers) whose IDs are close to its own, plus some finders whose IDs are more distant. The following algorithm ensures that finders whose IDs are close to a given swarm

ID are likely to know of peers from that swarm; the algorithm simultaneously provides a way to look them up. When a finder F needs to find peers from a particular swarm, it sends a request to the finders in its table whose IDs are close to that swarm's ID. If a contacted finder knows of any peers for that swarm, it replies with their contact information. Otherwise, it replies with the contact information of the finders in its table that are close to the swarm, so that F can iteratively query those finders.

After the search is exhausted, because there are no finders closer to the swarm, F inserts the contact information for itself and its associated peer into the finders closest to the swarm. The net effect is that peers for a particular swarm get entered in the tables of the finders that are close to that swarm.

The above scheme assumes that F is already part of the finder network, that it already knows how to contact some other finders. This assumption is true for finder installations that have run previously, because they are supposed to save information about other finders, even across executions. If a swarm uses a tracker, its peers are able to tell their finders about other finders (in a reversal of the peer and finder roles) because the BitTorrent peer protocol has been extended to exchange finder contact information. But, how can a newly installed finder discover other finders? The .torrent files for trackerless swarms include contact information for one or a few finders, instead of a tracker URL, for just that situation.

An unusual aspect of BitTorrent is that it deals head-on with the issue of fairness, or good "network citizenship." Protocols often depend on the good behavior of individual peers without being able to enforce it. For example, an unscrupulous Ethernet peer could get better performance by using a backoff algorithm that is more aggressive than exponential backoff, or an unscrupulous TCP peer could get better performance by not cooperating in congestion control.

The good behavior that BitTorrent depends on is peers uploading pieces to other peers. Since the typical BitTorrent user just wants to download the file as quickly as possible, there is a temptation to implement a peer that tries to download all the pieces while doing as little uploading as possible—this is a bad peer. To discourage bad behavior, the BitTorrent protocol includes mechanisms that allow peers to reward or punish each other. If a peer is misbehaving by not nicely uploading to another peer, the second peer can *choke* the bad peer: It can decide to stop uploading to the bad peer, at least temporarily, and send it a message saying so.

There is also a message type for telling a peer that it has been unchoked. The choking mechanism is also used by a peer to limit the number of its active BitTorrent connections, to maintain good TCP performance. There are many possible choking algorithms, and devising a good one is an art.

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 who 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 four 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 or a poorly performing 3G wireless link.
- *The last mile.* The link that connects the server to the Internet can be overloaded by too many requests, even if the aggregate bandwidth of that link is quite high.
- *The server itself.* A server has a finite amount of resources (CPU, memory, disk bandwidth, etc.) and can be overloaded by too many concurrent 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 where 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 remaining problems. Systems that do this are often called *Content Distribution Networks* (CDNs). Akamai operates what is probably the best-known CDN.

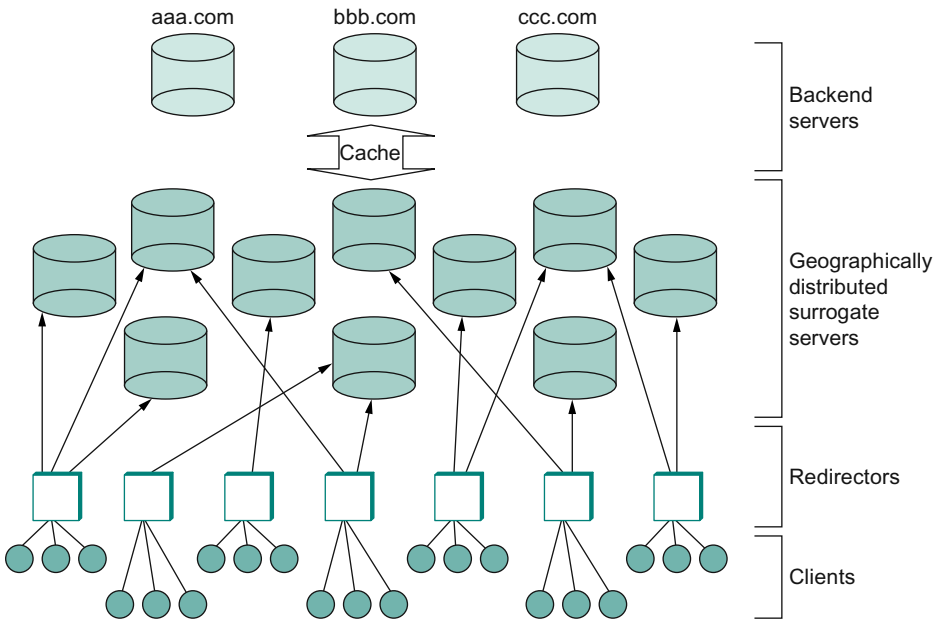
The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than having millions of users wait forever to contact

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 9.30. 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, such as 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.

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, it selects



■ **FIGURE 9.30** Components in a Content Distribution Network (CDN).

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 what 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 (e.g., 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 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 takes 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 the 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 or 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 prefer 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 algorithms (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 autonomous 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 two of the most widely used client/server-based application protocols: SMTP used to exchange electronic mail and HTTP used to walk the World Wide Web. We have seen how application-to-application communication is driving the creation of new protocol development frameworks such as SOAP and REST. And we have examined session control protocols, such as SIP and H.323, which are used to control multimedia applications such as Voice over IP. In addition to these application protocols, we looked at some critical supporting protocols: the DNS protocol used by the Domain Naming System and SNMP used to query remote nodes for the sake of network management. 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. 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. We also observe that the systems approach—understanding how functions and components interact to build a complete system—applies at least as much in the design of applications as in any other aspect of networking.

It's difficult to put a finger on a specific 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.

WHAT'S NEXT: NEW NETWORK ARCHITECTURE

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 that 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 router how it 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. Another possibility is that virtualization of networks becomes the norm, with perhaps some "slices" providing robust, well-understood, and fully debugged services while others are used for more experimental functions. This is one direction the research community is currently pursuing.

FURTHER READING

Our first article provides an interesting perspective on the early design and implementation of the World Wide Web, written by its inventors before it had taken the world by storm. The development of DNS is well described by Mockapetris and Dunlap. Overlays, CDNs, and peer-to-peer networks have been extensively researched in recent years, and the last six research papers provide a good place to start understanding the issues.

- Berners-Lee, T., R. Caillia, A. Luotonen, H. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM* 37(8), pages 76–82, August 1994.
- Mockapetris, P., and K. Dunlap. Development of the Domain Name System. *Proceedings of the SIGCOMM '88 Symposium*, pages 123–133, August 1988.
- Karger, D. et al. 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, May 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. et al. 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. et al. Chord: A scalable peer-to-peer lookup service for Internet applications. *Proceedings of the ACM SIGCOMM Conference*, pages 149–160, August 2001.
- Ratnasamy, S. et al. A scalable content-addressable network. *Proceedings of ACM SIGCOMM '01*, pages 161–172, August 2001.

SMTP was originally defined in RFC 821 [Pos82], and, of course, RFC 822 is RFC 822 [Cro82]. They have been, in IETF terminology, "obsoleted" by [Kle01] and [Res01], respectively. MIME is defined in a series of RFCs; the most recent version is defined in RFC 2045 [FB96], with several additional RFCs filling in details.

Version 1.0 of HTTP is specified in RFC 1945 [BLFF96], and the latest version (1.1) is defined in RFC 2616 [FGM+99]. Mogul [Mog95] made the case for the persistent connections in HTTP 1.1. There is 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]. Roy Fielding's Ph.D. thesis [Fie00] is the ultimate reference for REST.

SIP is defined in RFC 3261 [SCJ+02], which contains a helpful tutorial section as well as the detailed specification of the protocol. As with MIME, there are many other RFCs that extend the protocol.

There is 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].

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 RFC 2578 [MPS99], which defines the structure of management information for version 2 of SNMP (SNMPv2), and RFC 3416 [Pre02], which defines the protocol operations for SNMPv2. 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 et al. [PM97] provides a good introduction to SNMP and MIBS.

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., Anderson, Culler, and Roscoe [PACR02]. The original case for overriding BGP routes was made by Savage et al., Collins, Hoffman, Snell, and Anderson [SCH+99]. The idea of using DNS to load-balance a set of servers is described in RFC 1794 [Bri95]. A comprehensive treatment of the issue of web caching versus replicated servers can be found in Rabinovich and Spatscheck's book [RS02]. Wang, Pai, and Peterson 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 and for a wealth of information related to Web-related standards and history:

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

EXERCISES

1. Discuss how you might rewrite SMTP or HTTP to make use of a hypothetical general-purpose request/reply protocol. 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?

2. 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.

3. 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?

4. 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 2, above, and test its responses to nonexistent commands.

5. 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.

6. 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.

7. 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?

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

9. 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 (using SMTP) 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.

10. 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?

11. Find out how to configure an HTTP server so as to eliminate the 404 not found message and have a default (and 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.)

12. Why does the HTTP GET command on page 712,

```
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 2, above to connect to port 80 of an HTTP server and find out what happens if you leave the host name out.

13. 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.

14. Suppose a very large website 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?

15. Application protocols such as FTP and SMTP were designed from scratch, and they seem to work reasonably well. What is it about Business to Business and Enterprise Application Integration protocols that calls for a Web Services protocol framework?

16. Choose a Web Service with equivalent REST and SOAP interfaces, such as those offered by Amazon.com. Compare how equivalent operations are implemented in the two styles.

17. Get the WSDL for some SOAP-style Web Service and choose an operation. In the messages that implement that operation, identify the fields.

18. 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.")

19. 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.

20. Explain the relationship between Uniform Resource Locators (URLs) and Uniform Resource Identifiers (URIs). Give an example of a URI that is *not* a URL.

21. 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?

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

23. 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?

24. 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?)

25. 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?

26. 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?

27. Figure 9.17 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?

28. 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.

29. Many smaller organizations have their websites 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?

30. 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 levels of many existing names?

31. 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?

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

33. Take a suitable DNS-lookup utility (e.g., dig) and disable the recursive lookup feature (e.g., with +norecursive), 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 9.18; 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.

34. 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.

35. 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.

36. What information provided by SNMP might be useful to someone planning the IP spoofing attack of Exercise 17 in Chapter 5? What other SNMP information might be considered sensitive?

37. How would one design a CDN redirection mechanism using only HTTP 302 redirects or only DNS? What are the limitations of each approach? Is a combination of the two mechanisms feasible?

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

39. Imagine a situation in which multiple CDNs exist and want to peer with each other (analogous to the way autonomous systems peer with each at the IP layer) for the purpose of delivering content to a larger set of end users. For example, CDN A might serve content on behalf of one set of content providers and CDN B might serve content on behalf of another set of content providers, where both A and B have a physical footprint that allows them to deliver that content to disjoint sets of end users. Sketch how CDNs A and B can use a combination of DNS redirection and HTTP 302 redirects to deliver content from CDN A's content providers to CDN B's end users (and *vice versa*).

40. Imagine a CDN configured as a *caching hierarchy*, with end users accessing content from edge caches, which fetch the content for a parent cache upon a cache miss, and so on up to a root cache, which ultimately fetches content from an origin server. What metrics would guide provisioning decisions to (a) add more storage capacity to a given cache versus (b) adding an additional level to the caching hierarchy.

41. A multicast overlay effectively *pushes* streaming content from a single source to multiple destinations, with no caching of the stream at the intermediate nodes. A CDN effectively *pulls* content (including videos) down a caching hierarchy, caching it at the intermediate nodes. Show by example how these two can be viewed as duals of each other. Explain why a CDN can be viewed as equivalent to *asynchronous multicast*. (Hint: Think TiVo.)

42. Consider the following simplified BitTorrent scenario. There is a swarm of 2^n peers and, during the time in question, no peers join or leave the swarm. It takes a peer 1 unit of time to upload or download a piece, during which time it can only do one or the

other. Initially, one peer has the whole file and the others have nothing.

(a) If the swarm's target file consists of only 1 piece, what is the minimum time necessary for all the peers to obtain the file? Ignore all but upload/download time.

(b) Let x be your answer to the preceding question. If the swarm's target file instead consisted of 2 pieces, would it be possible for all the peers to obtain the file in less than $2x$ time units? Why or why not?