7

THE APPLICATION LAYER

Having finished all the preliminaries, we now come to the layer where all the applications are found. The layers below the application layer are there to provide transport services, but they do not do real work for users. In this chapter, we will study some real network applications.

Even at the application layer there is a need for support protocols, to allow many applications to function. Accordingly, we will look at an important one of these before starting with the applications themselves. The item in question is the DNS (Domain Name System), which maps Internet names to IP addresses. After that, we will examine three real applications: electronic mail, the World Wide Web (generally referred to simply as ''the Web''), and multimedia, including modern video streaming. We will finish the chapter by discussing content distribution, including peer-to-peer networks and content delivery networks.

7.1 THE DOMAIN NAME SYSTEM (DNS)

Although programs theoretically could refer to Web pages, mailboxes, and other resources by using the network (i.e., IP) addresses of the computers where they are stored, these addresses are difficult for people to remember. Also, browsing a company's Web pages from *128.111.24.41* is brittle: if the company moves the Web server to a different machine with a different IP address, everyone needs to be told the new IP address. Although moving a Web site from one IP address to

another might seem far-fetched, in practice this general notion occurs quite often, in the form of load balancing. Specifically, many modern Web sites host their content on multiple machines, often geographically distributed clusters. The organization hosting the content may wish to ''move'' a client's communication from one Web server to another. The DNS is typically the most convenient way to do this.

High-level, readable names decouple machine names from machine addresses. An organization's Web server could thus be referred to as *www.cs.uchicago.edu*, regardless of its IP address. Because the devices along a network path forward traffic to its destination based on IP address, these human-readable domain names must be converted to IP addresses; the **DNS** (**Domain Name System**) is the mechanism that does so. In the subsequent sections, we will study how DNS performs this mapping, as well as how it has evolved over the past decades. In particular, one of the most significant developments in the DNS in recent years is its implications for user privacy. We will explore these implications and various recent developments in DNS encryption that are related to privacy.

7.1.1 History and Overview

Back in the ARPANET days, a file, *hosts.txt*, listed all the computer names and their IP addresses. Every night, all of the hosts would fetch it from the site at which it was maintained. For a network of a few hundred large timesharing machines, this approach worked reasonably well.

However, well before many millions of PCs were connected to the Internet, everyone involved with it realized that this approach could not continue to work forever. For one thing, the size of the file would become too large. Even more importantly, host name conflicts would occur constantly unless names were centrally managed, something unthinkable in a huge international network due to the load and latency. The Domain Name System was invented in 1983 to address these problems, and it has been a key part of the Internet ever since.

DNS is a hierarchical naming scheme and a distributed database system that implements this naming scheme. It is primarily used for mapping host names to IP addresses, but it has several other purposes, which we will outline in more detail below. DNS is one of the most actively evolving protocols in the Internet. DNS is defined in RFC 1034, RFC 1035, RFC 2181, and further elaborated in many other RFCs.

7.1.2 The DNS Lookup Process

DNS operates as follows. To map a name onto an IP address, an application program calls a library procedure, (typically *gethostbyname* or the equivalent) passing this function the name as a parameter. This process is sometimes referred to as the **stub resolver**. The stub resolver sends a query containing the name to a local DNS resolver, often called the **local recursive resolver** or simply the **local**

resolver, which subsequently performs a so-called **recursive lookup** for the name against a set of DNS resolvers. The local recursive resolver ultimately returns a response with the corresponding IP address to the stub resolver, which then passes the result to the function that issued the query in the first place. The query and response messages are sent as UDP packets. Given knowledge of the IP address, the program can then communicate with the host corresponding to the DNS name that it had looked up. We will explore this process in more detail later in this chapter.

Typically, the stub resolver issues a recursive lookup to the local resolver, meaning that it simply issues the query and waits for the response from the local resolver. The local resolver, on the other hand, issues a sequence of queries to the respective name servers for each part of the name hierarchy; the name server that is responsible for a particular part of the hierarchy is often called the **authoritative name server** for that domain. As we will see later, DNS uses caching, but caches can be out of date. The authoritative name server is, well, authoritative. It is by definition always correct. Before describing more detailed operation of DNS, we describe the DNS name server hierarchy and how names are allocated.

When a host's stub resolver sends a query to the local resolver, the local resolver handles the resolution until it has the desired answer, or no answer. It does *not* return partial answers. On the other hand, the root name server (and each subsequent name server) does not recursively continue the query for the local name server. It just returns a partial answer and moves on to the next query. The local resolver is responsible for continuing the resolution by issuing further iterative queries.

The name resolution process typically involves both mechanisms. A recursive query may always seem preferable, but many name servers (especially the root) will not handle them. They are too busy. Iterative queries put the burden on the originator. The rationale for the local name server supporting a recursive query is that it is providing a service to hosts in its domain. Those hosts do not have to be configured to run a full name server, just to reach the local one. A 16-bit transaction identifier is included in each query and copied to the response so that a name server can match answers to the corresponding query, even if multiple queries are outstanding at the same time.

All of the answers, including all the partial answers returned, are cached. In this way, if a computer at *cs.vu.nl* queries for *cs.uchicago.edu*, the answer is cached. If shortly thereafter, another host at *cs.vu.nl* also queries *cs.uchicago.edu*, the answer will already be known. Even better, if a host queries for a different host in the same domain, say *noise.cs.uchicago.edu*, the query can be sent directly to the authoritative name server for *cs.uchicago.edu*. Similarly, queries for other domains in *uchicago.edu* can start directly from the *uchicago.edu* name server. Using cached answers greatly reduces the steps in a query and improves performance. The original scenario we sketched is in fact the worst case that occurs when no useful information is available in the cache.

Cached answers are not authoritative, since changes made at *cs.uchicago.edu* will not be propagated to all the caches in the world that may know about it. For this reason, cache entries should not live too long. This is the reason that the *Time_to_live* field is included in each DNS resource record, a part of the DNS database we will discuss shortly. It tells remote name servers how long to cache records. If a certain machine has had the same IP address for years, it may be safe to cache that information for one day. For more volatile information, it might be safer to purge the records after a few seconds or a minute.

DNS queries have a simple format that includes various information, including the name being queried (QNAME), as well as other auxiliary information, such as a transaction identifier; the transaction identifier is often used to map queries to responses. Initially, the transaction ID was only 16 bits, and the queries and responses were not secured; this design choice left DNS vulnerable to a variety of attacks including something called a cache poisoning attack, whose details we discuss further in Chap. 8. When performing a series of iterative lookups, a recursive DNS resolver might send the entire QNAME to the sequence of authoritative name servers returning the responses. At some point, protocol designers pointed out that sending the entire QNAME to every authoritative name server in a sequence of iterative resolvers constituted a privacy risk. As a result, many recursive resolvers now use a process called **QNAME minimization**, whereby the local resolver only sends the part of the query that the respective authoritative name server has the information to resolve. For example, with QNAME minimization, given a name to resolve such as *www.cs.uchicago.edu*, a local resolver would send only the string *cs.uchicago.edu* to the authoritative name server for *uchicago.edu*, as opposed to the fully qualified domain name (FQDN), to avoid revealing the entire FQDN to the authoritative name server. For more information on QNAME minimization, see RFC 7816.

Until very recently, DNS queries and responses relied on UDP as its transport protocol, based on the rationale that DNS queries and responses needed to be fast and lightweight, and could not handle the corresponding overhead of a TCP three-way handshake. However, various developments, including the resulting insecurity of the DNS protocol and the myriad subsequent attacks that DNS has been subject to, ranging from cache poisoning to distributed denial-of-service (DDoS) attacks, has resulted in an increasing trend towards the use of TCP as the transport protocol for DNS. Using TCP as the transport protocol for DNS has subsequently allowed DNS to leverage modern secure transport and application-layer protocols, resulting in DNS-over-TLS (DoT) and DNS-over-HTTPS (DoH). We discuss these developments in more detail later in this chapter.

If the DNS stub resolver does not receive a response within some relatively short period of time (a timeout period), the DNS client repeats the query, trying another server for the domain after a small number of retries. This process is designed to handle the case of the server being down as well as the query or response packet getting lost.

7.1.3 The DNS Name Space and Hierarchy

Managing a large and constantly changing set of names is challenging. In the postal system, name management is done by requiring letters to specify (implicitly or explicitly) the country, state or province, city, street address, and name of the addressee. Using this kind of hierarchical addressing ensures that there is no confusion between the Marvin Anderson on Main St. in White Plains, N.Y. and the Marvin Anderson on Main St. in Austin, Texas. DNS works the same way.

For the Internet, the top of the naming hierarchy is managed by an organization called **ICANN** (**Internet Corporation for Assigned Names and Numbers**). ICANN was created for this purpose in 1998, as part of the maturing of the Internet to a worldwide, economic concern. Conceptually, the Internet is divided into over 250 **top-level domains**, where each domain covers many hosts. Each domain is partitioned into subdomains, and these are further partitioned, and so on. All of these domains constitute a namespace hierarchy, which can be represented by a tree, as shown in Fig. 7-1. The leaves of the tree represent domains that have no subdomains (but do contain machines, of course). A leaf domain may contain a single host, or it may represent a company and contain thousands of hosts.

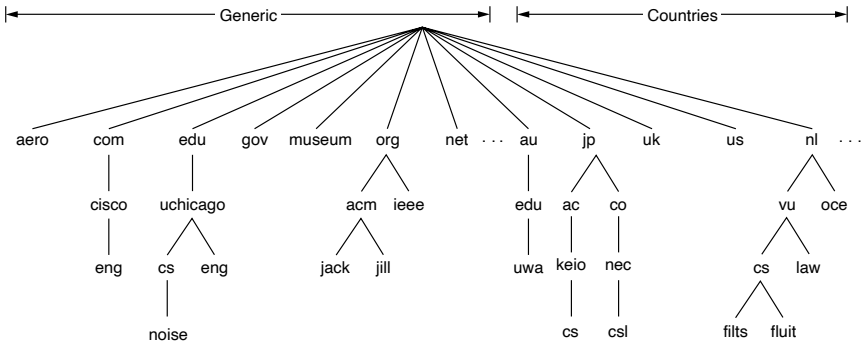

Figure 7-1. A portion of the Internet domain name space.

The top-level domains have several different types: **gTLD** (**generic Top Level Domain**), **ccTLD** (**country code Top Level Doman**), and others. Some of the original generic TLDs, listed in Fig. 7-2, include original domains from the 1980s, plus additional top-level domains introduced to ICANN. The country domains include one entry for every country, as defined in ISO 3166. Internationalized country domain names that use non-Latin alphabets were introduced in 2010. These domains let people name hosts in Arabic, Chinese, Cyrillic, Hebrew, or other languages.

In 2011, there were only 22 gTLDs, but in June 2011, ICANN voted to end restrictions on the creation of additional gTLDs, allowing companies and other

organizations to select essentially arbitrary top-level domains, including TLDs that include non-Latin characters (e.g., Cyrillic). ICANN began accepting applications for new TLDs at the beginning of 2012. The initial cost of applying for a new TLD was nearly 200,000 dollars. Some of the first new gTLDs became operational in 2013, and in July 2013, the first four new gTLDs were launched based on agreement that was signed in Durban, South Africa. All four were based on non-Latin characters: the Arabic word for ''Web,'' the Russian word for ''online,'' the Russian word for ''site,'' and the Chinese word for ''game.'' Some tech giants have applied for many gTLDs: Google and Amazon, for example, have each applied for about 100 new gTLDs. Today, some of the most popular gTLDs include *top*, *loan*, *xyz*, and so forth.

Domain	Intended use	Start date	Restricted?
com	Commercial	1985	No
edu	Educational institutions	1985	Yes
gov	Government	1985	Yes
int	International organizations	1988	Yes
mil	Military	1985	Yes
net	Network providers	1985	No
org	Non-profit organizations	1985	No
aero	Air transport	2001	Yes
biz	Businesses	2001	No
coop	Cooperatives	2001	Yes
info	Informational	2002	No
museum	Museums	2002	Yes
name	People	2002	No
pro	Professionals	2002	Yes
cat	Catalan	2005	Yes
jobs	Employment	2005	Yes
mobi	Mobile devices	2005	Yes
tel	Contact details	2005	Yes
travel	Travel industry	2005	Yes
xxx	Sex industry	2010	No

Figure 7-2. The original generic TLDs, as of 2010. As of 2020, there are more than 1,200 gTLDs.

Getting a second-level domain, such as *name-of-company.com*, is easy. The top-level domains are operated by companies called **registries**. They are appointed by ICANN. For example, the registry for *com* is Verisign. One level down, **registrars** sell domain names directly to users. There are many of them and they compete on price and service. Common registrars include Domain.com, GoDaddy, and

NameCheap. Fig. 7-3 shows the relationship between registries and registrars as far as registering a domain name is concerned.

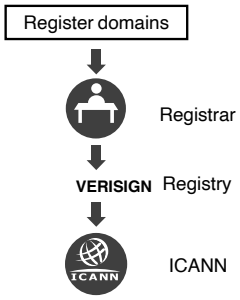

Figure 7-3. The relationship between registries and registrars.

The domain name that a machine aims to look up is typically called a **FQDN** (**Fully Qualified Domain Name**) such as *www.cs.uchicago.edu* or *cisco.com*. The FQDN starts with the most specific part of the domain name, and each part of the hierarchy is separated by a ''.'' (Technically, all FQDNs end with a ''.'' as well, signifying the root of the DNS hierarchy, although most operating systems complete that portion of the domain name automatically.)

Each domain is named by the path upward from it to the (unnamed) root. The components are separated by periods (pronounced ''dot''). Thus, the engineering department at Cisco might be *eng.cisco.com.*, rather than a UNIX-style name such as */com/cisco/eng*. Notice that this hierarchical naming means that *eng.cisco.com.* does not conflict with a potential use of *eng* in *eng.uchicago.edu.*, which might be used by the English department at the University of Chicago.

Domain names can be either absolute or relative. An absolute domain name always ends with a period (e.g., *eng.cisco.com.*), whereas a relative one does not. Relative names have to be interpreted in some context to uniquely determine their true meaning. In both cases, a named domain refers to a specific node in the tree and all the nodes under it.

Domain names are case-insensitive, so *edu*, *Edu*, and *EDU* mean the same thing. Component names can be up to 63 characters long, and full path names must not exceed 255 characters. The fact that DNS in case insensitive has been used to defend against various DNS attacks, including DNS cache poisoning attacks, using a technique called 0x20 encoding (Dagon et al., 2008), which we will discuss in more detail later in this chapter.

In principle, domains can be inserted into the hierarchy in either the generic or the country domains. For example, the domain *cc.gatech.edu* could equally well be (and are often) listed under the *us* country domain as *cc.gt.atl.ga.us*. In practice, however, most organizations in the United States are under generic domains,

and most outside the United States are under the domain of their country. There is no rule against registering under multiple top-level domains. Large companies often do so (e.g., *sony.com*, *sony.net*, and *sony.nl*).

Each domain controls how it allocates the domains under it. For example, Japan has domains *ac.jp* and *co.jp* that mirror *edu* and *com*. The Netherlands does not make this distinction and puts all organizations directly under *nl*. Austraian universities are all in *edu.au*. Thus, all three of the following are university CS and EE departments:

1. *cs.chicago.edu* (University of Chicago, in the U.S.).
2. *cs.vu.nl* (Vrije Universiteit, in The Netherlands).
3. *ee.uwa.edu.au* (University of Western Australia).

To create a new domain, permission is required of the domain in which it will be included. For example, if a security research group at the University of Chicago wants to be known as *security.cs.uchicago.edu*, it has to get permission from whoever manages *cs.uchicago.edu*. (Fortunately, that person is typically not far away, thanks to the federated management architecture of DNS) Similarly, if a new university is chartered, say, the University of Northern South Dakota, it must ask the manager of the *edu* domain to assign it *unsd.edu* (if that is still available). In this way, name conflicts are avoided and each domain can keep track of all its subdomains. Once a new domain has been created and registered, it can create subdomains, such as *cs.unsd.edu*, without getting permission from anybody higher up the tree.

Naming follows organizational boundaries, not physical networks. For example, if the computer science and electrical engineering departments are located in the same building and share the same LAN, they can nevertheless have distinct domains. Similarly, even if computer science is split over Babbage Hall and Turing Hall, the hosts in both buildings will normally belong to the same domain.

7.1.4 DNS Queries and Responses

We now turn to the structure, format, and purpose of DNS queries, and how the DNS servers answer those queries.

DNS Queries

As previously discussed, a DNS client typically issues a query to a local recursive resolver, which performs an iterative query to ultimately resolve the query. The most common query type is an *A* record query, which asks for a mapping from a domain name to an IP address for a corresponding Internet endpoint. DNS has a range of other resource records (with corresponding queries), as we discuss further in the next section on resource records (i.e., responses).

Although the primary mechanism for DNS has long been to map human readable names to IP addresses, over the years, DNS queries have been used for a variety of other purposes. Another common use for DNS queries is to look up domains in a **DNSBL** (**DNS-based blacklist**), which are lists that are commonly maintained to keep track of IP addresses associated with spammers and malware. To look up a domain name in a DNSBL, a client might send a DNS A-record query to a special DNS server, such as *pbl.spamhaus.org* (a ''policy blacklist''), which corresponds to a list of IP addresses that are not supposed to be making connections to mail servers. To look up a particular IP address, a client simply reverses the octets for the IP address and prepends the result to *pbl.spamhaus.org*.

For example, to look up 127.0.0.2, a client would simply issue a query for *2.0.0.127.pbl.spamhaus.org*. If the corresponding IP address was in the list, the DNS query would return an IP address that typically encodes some additional information, such as the provenance of that entry in the list. If the IP address is not contained in the list, the DNS server would indicate that by responding with the corresponding NXDOMAIN response, corresponding to ''no such domain.''

Extensions and Enhancements to DNS Queries

DNS queries have become more sophisticated and complex over time, as the needs to serve clients with increasingly specific and relevant information over time has increased, and as security concerns have grown. Two significant extensions to DNS queries in recent years has been the use of the **EDNS0 CS Extended DNS Client Subnet** or simply **EDNS Client Subnet** option, whereby a client's local recursive resolver passes the IP address subnet of the stub resolver to the authoritative name server.

The EDNS0 CS mechanism allows the authoritative name server for a domain name to know the IP address of the client that initially performed the query. Knowing this information can typically allow an authoritative DNS server to perform a more effective mapping to a nearby copy of a replicated service. For example, if a client issues a query for google.com, the authoritative name server for Google would typically want to return a name that corresponds to a front-end server that is close to the client. The ability to do so of course depends on knowing where on the network (and, ideally, where in the world, geographically) the client is located. Ordinarily, an authoritative name server might only see the IP address of the local recursive resolver.

If the client that initiated the query happens to be located near its respective local resolver, then the authoritative server for that domain could determine an appropriate client mapping simply from the location of the DNS local recursive. Increasingly, however, clients have begun to use local recursive resolvers that may have IP addresses that make it difficult to locate the client. For example, Google and Cloudflare both operate public DNS resolvers (8.8.8.8 and 1.1.1.1, respectively). If a client is configured to use one of these local recursive resolvers, then

the authoritative name server does not learn much useful information from the IP address of the recursive resolver. EDNS0 CS solves this problem by including the IP subnet in the query from the local recursive, so that the authoritative can see the IP subnet of the client that initiated the query.

As previously noted, the names in DNS queries are not case sensitive. This characteristic has allowed modern DNS resolvers to include additional bits of a transaction ID in the query by setting each character in a QNAME to an arbitrary case. A 16-bit transaction ID is vulnerable to various cache poisoning attacks, including the Kaminsky attack described in Chap. 8. This vulnerability partially arises because the DNS transaction ID is only 16 bits. Increasing the number of bits in the transaction ID would require changing the DNS protocol specification, which is a massive undertaking.

An alternative was developed, usually called **0x20 encoding**, whereby a local recursive would toggle the case on each QNAME (e.g., *uchicago.edu* might become *uCHicaGO.EDu* or similar), allowing each letter in the domain name to encode an additional bit for the DNS transaction ID. The catch, of course, is that no other resolver should alter the case of the QNAME in subsequent iterative queries or responses. If the casing is preserved, then the corresponding reply contains the QNAME with the original casing indicated by the local recursive resolver, effectively acting adding bits to the transaction identifier. The whole thing is an ugly hack, but such is the nature of trying to change widely deployed software while maintaining backward compatibility.

DNS Responses and Resource Records

Every domain, whether it is a single host or a top-level domain, can have a set of **resource records** associated with it. These records are the DNS database. For a single host, the most common resource record is just its IP address, but many other kinds of resource records also exist. When a resolver gives a domain name to DNS, what it gets back are the resource records associated with that name. Thus, the primary function of DNS is to map domain names onto resource records.

A resource record is a five-tuple. Although resource records are encoded in binary, in most expositions resource records are presented as ASCII text, with one line per resource record, as follows:

```
Domain_name  Time_to_live  Class  Type  Value
```

The *Domain_name* tells the domain to which this record applies. Normally, many records exist for each domain, and each copy of the database holds information about multiple domains. This field is thus the primary search key used to satisfy queries. The order of the records in the database is not significant.

The *Time_to_live* field gives an indication of how stable the record is. Information that is highly stable is assigned a large value, such as 86400 (the number of seconds in 1 day). Information that is volatile (like stock prices), or that operators

may want to change frequently (e.g., to enable load balancing a single name across multiple IP addresses) may be assigned a small value, such as 60 seconds (1 minute). We will return to this point later when we have discussed caching.

The third field of every resource record is the *Class*. For Internet information, it is always *IN*. For non-Internet information, other codes can be used, but in practice these are rarely seen.

The *Type* field tells what kind of record this is. There are many kinds of DNS records. The important types are listed in Fig. 7-4.

Type	Meaning	Value
SOA	Start of authority	Parameters for this zone
A	IPv4 address of a host	32-Bit integer
AAAA	IPv6 address of a host	128-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
SPF	Sender policy framework	Text encoding of mail sending policy
SRV	Service	Host that provides it
TXT	Text	Descriptive ASCII text

Figure 7-4. The principal DNS resource record types.

An *SOA* record provides the name of the primary source of information about the name server's zone (described below), the email address of its administrator, a unique serial number, and various flags and timeouts.

Common Record Types

The most important record type is the *A* (Address) record. It holds a 32-bit IPv4 address of an interface for some host. The corresponding *AAAA*, or ''quad A,'' record holds a 128-bit IPv6 address. Every Internet host must have at least one IP address so that other machines can communicate with it. Some hosts have two or more network interfaces, so they will have two or more type *A* or *AAAA* resource records. Additionally, a single service (e.g., *google.com*) may be hosted on many geographically distributed machines around the world (Calder et al., 2013). In these cases, a DNS resolver might return multiple IP addresses for a single domain name. In the case of a geographically distributed service, a resolver may return to its client one or more IP addresses of a server that is close to the client (geographically or topologically), to improve performance, and for load balancing.

An important record type is the *NS* record. It specifies a name server for the domain or subdomain. This is a host that has a copy of the database for a domain. It is used as part of the process to look up names, which we will describe shortly.

Another record type is the *MX* record. It specifies the name of the host prepared to accept email for the specified domain. It is used because not every machine is prepared to accept email. If someone wants to send email to, as an example, *bill@microsoft.com*, the sending host needs to find some mail server located at *microsoft.com* that is willing to accept email. The *MX* record can provide this information.

CNAME records allow aliases to be created. For example, a person familiar with Internet naming in general and wanting to send a message to user *paul* in the computer science department at the University of Chicago might guess that *paul@cs.chicago.edu* will work. Actually, this address will not work, because the domain for the computer science department is *cs.uchicago.edu*. As a service to people who do not know this, the University of Chicago could create a *CNAME* entry to point people and programs in the right direction. An entry like this one might do the job:

```
www.cs.uchicago.edu  120 IN  CNAME hnd.cs.uchicago.edu
```

*CNAME*s are commonly used for Web site aliases, because the common Web server addresses (which often start with *www*) tend to be hosted on machines that serve multiple purposes and whose primary name is not *www*.

The *PTR* record points to another name and is typically used to associate an IP address with a corresponding name. PTR lookups that associate a name with a corresponding IP address are typically called **reverse lookups**.

SRV is a newer type of record that allows a host to be identified for a given service in a domain. For example, the Web server for *www.cs.uchicago.edu* could be identified as *hnd.cs.uchicago.edu*. This record generalizes the *MX* record that performs the same task but it is just for mail servers.

SPF lets a domain encode information about what machines in the domain will send mail to the rest of the Internet. This helps receiving machines check that mail is valid. If mail is being received from a machine that calls itself *dodgy* but the domain records say that mail will only be sent out of the domain by a machine called *smtp*, chances are that the mail is forged junk mail.

Last on the list, *TXT* records were originally provided to allow domains to identify themselves in arbitrary ways. Nowadays, they usually encode machine-readable information, typically the *SPF* information.

Finally, we have the *Value* field. This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type. A short description of the *Value* fields for each of the principal record types is given in Fig. 7-4.

DNSSEC Records

The original deployment of DNS did not consider the security of the protocol. In particular, DNS name servers or resolvers could manipulate the contents of any DNS record, thus causing the client to receive incorrect information. RFC 3833

highlights some of the various security threats to DNS and how DNSSEC addresses these threats. DNSSEC records allow responses from DNS name servers to carry digital signatures, which the local or stub resolver can subsequently verify to ensure that the DNS records were not modified or tampered with. Each DNS server computes a hash (a kind of long checksum) of the **RRSET** (**Resource Record Set**) for each set of resource records of the same type, with its private cryptographic keys. Corresponding public keys can be used to verify the signatures on the RRSETs. (For those not familiar with cryptography, Chap. 8 provides some technical background.)

Verifying the signature of an RRSET with the name server's corresponding public key of course requires verifying the authenticity of that server's public key. This verification can be accomplished if the public key of one authoritative name server's public key is signed by the parent name server in the name hierarchy. For example, the *.edu* authoritative name server might sign the public key corresponding to the chicago.edu authoritative name server, and so forth.

DNSSEC has two resource records relating to public keys: (1) the RRSIG record, which corresponds to a signature over the RRSET, signed with the corresponding authoritative name server's private key, and (2) the DNSKEY record, which is the public key for the corresponding RRSET, which is signed by the parent's private key. This hierarchical structure for signatures allows DNSSEC public keys for the name server hierarchy to be distributed in band. Only the root-level public keys must be distributed out-of-band, and those keys can be distributed in the same way that resolvers come to know about the IP addresses of the root name servers. Chap. 8 discusses DNSSEC in more detail.

DNS Zones

Fig. 7-5. shows an example of the type of information that might be available in a typical DNS resource record for a particular domain name. This figure depicts part of a (hypothetical) database for the *cs.vu.nl* domain shown in Fig. 7-1, which is often called a **DNS zone file** or sometimes simply **DNS zone** for short. This zone file contains seven types of resource records.

The first noncomment line of Fig. 7-5 gives some basic information about the domain, which will not concern us further. Then come two entries giving the first and second places to try to deliver email sent to *person@cs.vu.nl*. The *zephyr* (a specific machine) should be tried first. If that fails, the *top* should be tried as the next choice. The next line identifies the name server for the domain as *star*.

After the blank line (added for readability) come lines giving the IP addresses for the *star*, *zephyr*, and *top*. These are followed by an alias, *www.cs.vu.nl*, so that this address can be used without designating a specific machine. Creating this alias allows *cs.vu.nl* to change its World Wide Web server without invalidating the address people use to get to it. A similar argument holds for *ftp.cs.vu.nl*.

```
; Authoritative data for cs.vu.nl
cs.vu.nl.       86400   IN  SOA     star boss (9527,7200,7200,241920,86400)
cs.vu.nl.       86400   IN  MX      1 zephyr
cs.vu.nl.       86400   IN  MX      2 top
cs.vu.nl.       86400   IN  NS      star

star            86400   IN  A       130.37.56.205
zephyr          86400   IN  A       130.37.20.10
top             86400   IN  A       130.37.20.11
www             86400   IN  CNAME   star.cs.vu.nl
ftp             86400   IN  CNAME   zephyr.cs.vu.nl

flits           86400   IN  A       130.37.16.112
flits           86400   IN  A       192.31.231.165
flits           86400   IN  MX      1 flits
flits           86400   IN  MX      2 zephyr
flits           86400   IN  MX      3 top

rowboat                 IN  A       130.37.56.201
                        IN  MX      1 rowboat
                        IN  MX      2 zephyr

little-sister           IN  A       130.37.62.23

laserjet                IN  A       192.31.231.216
```

Figure 7-5. A portion of a possible DNS database (zone file) for *cs.vu.nl*.

The section for the machine *flits* lists two IP addresses and three choices are given for handling email sent to *flits.cs.vu.nl*. First choice is naturally the *flits* itself, but if it is down, the *zephyr* and *top* are the second and third choices.

The next three lines contain a typical entry for a computer, in this example, *rowboat.cs.vu.nl*. The information provided contains the IP address and the primary and secondary mail drops. Then comes an entry for a computer that is not capable of receiving mail itself, followed by an entry that is likely for a printer (laserjet) that is connected to the Internet.

In theory at least, a single name server could contain the entire DNS database and respond to all queries about it. In practice, this server would be so overloaded as to be useless. Furthermore, if it ever went down, the entire Internet would be crippled.

To avoid the problems associated with having only a single source of information, the DNS name space is divided into nonoverlapping **zones**. One possible way to divide the name space of Fig. 7-1 is shown in Fig. 7-6. Each circled zone contains some part of the tree.

Where the zone boundaries are placed within a zone is up to that zone's administrator. This decision is made in large part based on how many name servers are

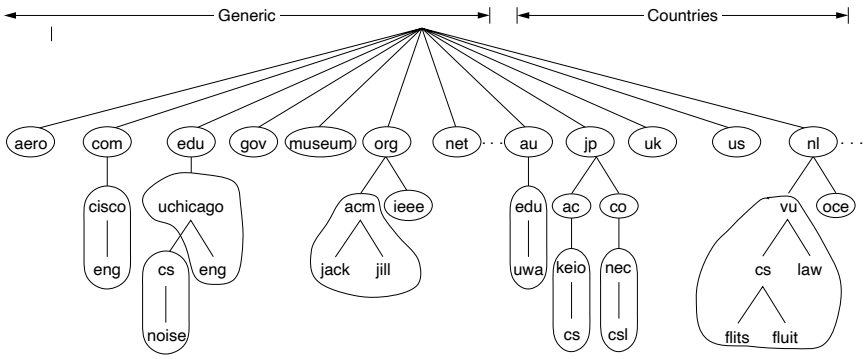

Figure 7-6. Part of the DNS name space divided into zones (which are circled).

desired, and where. For example, in Fig. 7-6, the University of Chicago has a zone for *chicago.edu* that handles traffic to *cs.uchicago.edu*. However, it does not handle *eng.uchicago.edu*. That is a separate zone with its own name servers. Such a decision might be made when a department such as English does not wish to run its own name server, but a department such as Computer Science does.

7.1.5 Name Resolution

Each zone is associated with one or more name servers. These are hosts that hold the database for the zone. Normally, a zone will have one primary name server, which gets its information from a file on its disk, and one or more secondary name servers, which get their information from the primary name server. To improve reliability, some of the name servers can be located outside the zone.

The process of looking up a name and finding an address is called **name resolution**. When a resolver has a query about a domain name, it passes the query to a local name server. If the domain being sought falls under the jurisdiction of the name server, such as *top.cs.vu.nl* falling under *cs.vu.nl*, it returns the authoritative resource records. An **authoritative record** is one that comes from the authority that manages the record and is thus always correct. Authoritative records are in contrast to **cached records**, which may be out of date.

What happens when the domain is remote, such as when *flits.cs.vu.nl* wants to find the IP address of *cs.uchicago.edu* at the University of Chicago? In this case, and if there is no cached information about the domain available locally, the name server begins a remote query. This query follows the process shown in Fig. 7-7. Step 1 shows the query that is sent to the local name server. The query contains the domain name sought, the type (*A*), and the class(*IN*).

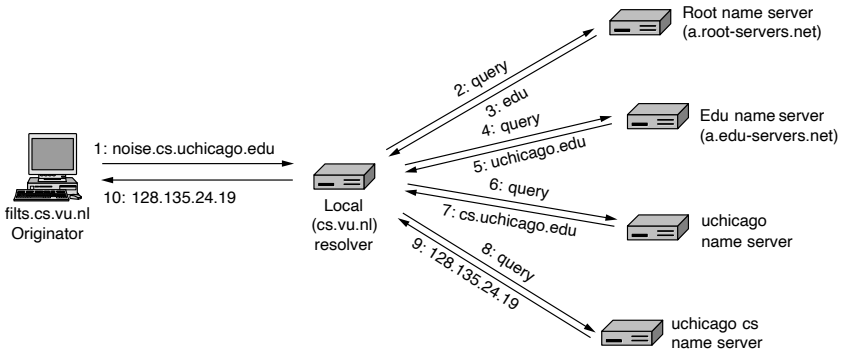

Figure 7-7. Example of a resolver looking up a remote name in 10 steps.

The next step is to start at the top of the name hierarchy by asking one of the **root name servers**. These name servers have information about each top-level domain. This is shown as step 2 in Fig. 7-7. To contact a root server, each name server must have information about one or more root name servers. This information is normally present in a system configuration file that is loaded into the DNS cache when the DNS server is started. It is simply a list of *NS* records for the root and the corresponding *A* records.

There are 13 root DNS servers, unimaginatively called *a.root-servers.net* through *m.root-servers.net*. Each root server could logically be a single computer. However, since the entire Internet depends on the root servers, they are powerful and heavily replicated computers. Most of the servers are present in multiple geographical locations and reached using anycast routing, in which a packet is delivered to the nearest instance of a destination address; we described anycast in Chap. 5. The replication improves reliability and performance.

The root name server is very unlikely to know the address of a machine at *uchicago.edu*, and probably does not know the name server for *uchicago.edu* either. But it must know the name server for the *edu* domain, in which *cs.uchicago.edu* is located. It returns the name and IP address for that part of the answer in step 3.

The local name server then continues its quest. It sends the entire query to the *edu* name server (*a.edu-servers.net*). That name server returns the name server for *uchicago.edu*. This is shown in steps 4 and 5. Closer now, the local name server sends the query to the *uchicago.edu* name server (step 6). If the domain name being sought was in the English department, the answer would be found, as the *uchicago.edu* zone includes the English department. The Computer Science department has chosen to run its own name server. The query returns the name and IP address of the *uchicago.edu* Computer Science name server (step 7).

Finally, the local name server queries the *uchicago.edu* Computer Science name server (step 8). This server is authoritative for the domain *cs.uchicago.edu*, so it must have the answer. It returns the final answer (step 9), which the local name server forwards as a response to *flits.cs.vu.nl* (step 10).

7.1.6 Hands on with DNS

You can explore this process using standard tools such as the *dig* program that is installed on most UNIX systems. For example, typing

```
dig ns @a.edu-servers.net cs.uchicago.edu
```

will send a query for *cs.uchicago.edu* to the *a.edu-servers.net* name server and print out the result for its name servers. This will show you the information obtained in Step 4 in the example above, and you will learn the name and IP address of the *uchicago.edu* name servers. Most organizations will have multiple name servers in case one is down. Half a dozen is not unusual. If you have access to a UNIX, Linux, or MacOS system, try experimenting with the *dig* program to see what it can do. You can learn a lot about DNS from using it. (The *dig* program is also available for Windows, but you may have to install it yourself.)

Even though its purpose is simple, it should be clear that DNS is a large and complex distributed system that is comprised of millions of name servers that work together. It forms a key link between human-readable domain names and the IP addresses of machines. It includes replication and caching for performance and reliability and is designed to be highly robust.

Some applications need to use names in more flexible ways, for example, by naming content and resolving to the IP address of a nearby host that has the content. This fits the model of searching for and downloading a movie. It is the movie that matters, not the computer that has a copy of it, so all that is wanted is the IP address of *any* nearby computer that has a copy of the movie. Content delivery networks are one way to accomplish this mapping. We will describe how they build on the DNS later in this chapter, in Sec. 7.5.

7.1.7 DNS Privacy

Historically, DNS queries and responses have not been encrypted. As a result, any other device or eavesdropper on the network (e.g., other devices, a system administrator, a coffee shop network) could conceivably observe a user's DNS traffic and determine information about that user. For example, a lookup to a site like *uchicago.edu* might indicate that a user was browsing the University of Chicago Web site. While such information might seem innocuous, DNS lookups to Web sites such as *webmd.com* might indicate that a user was performing medical research. Combinations of lookups combined with other information can often even reveal more specific information, possibly even the precise Web site that a user is visiting.

Privacy issues associated with DNS queries have become more contentious when considering emerging applications, such as the Internet of Things (IoT) and smart homes. For example, the DNS queries that a device issues can reveal information about the type of devices that users have in their smart homes and the extent to which they are interacting with those devices. For example, the DNS queries that an Internet-connected camera or sleep monitor issues can uniquely identify the device (Apthorpe et al., 2019). Given the increasingly sensitive activities that people perform on Internet-connected devices, from browsers to Internet-connected ''smart'' devices, there is an increasing desire to encrypt DNS queries and responses.

Several recent developments are poised to potentially reshape DNS entirely. The first is the movement toward encrypting DNS queries and responses. Various organizations, including Cloudflare, Google, and others are now offering users the opportunity to direct their DNS traffic to their own local recursive resolvers, and additionally offering support for encrypted transport (e.g., TLS, HTTPS) between the DNS stub resolver and their local resolver. In some cases, these organizations are partnering with Web browser manufacturers (e.g., Mozilla) to potentially direct all DNS traffic to these local resolvers by default.

If all DNS queries and responses are exchanged with cloud providers over encrypted transport by default, the implications for the future of the Internet architecture could be extremely significant. Specifically, Internet service providers will no longer have the ability to observe DNS queries from their subscribers' home networks, which has, in the past, been one of the primary ways that ISPs monitor these networks for infections and malware (Antonakakis et al., 2010). Other functions, such as parental controls and various other services that ISPs offer, also depend on seeing DNS traffic.

Ultimately, two somewhat orthogonal issues are at play. The first is the shift of DNS towards encrypted transport, which almost everyone would agree is a positive change (there were initial concerns about performance, which have mostly now been addressed). The second issue is thornier: it involves who gets to operate the local recursive resolvers. Previously, the local recursive resolver was generally operated by a user's ISP; if DNS resolution moves to the browser, however, via DoH, then the browsers (the two most popular of which are at this point largely controlled by a single dominant provider, Google) can control who is in a position to observe DNS traffic. Ultimately, the operator of the local recursive resolver can see the DNS queries from the user and associate those with an IP address; whether the user wants their ISP or a large advertising company to see their DNS traffic should be their choice, but the default settings in the browser may ultimately determine who ends up seeing the majority of this traffic. Presently, a wide range of organizations, from ISPs to content providers and advertising companies are trying to establish what are being called **TRR**s (**Trusted Recursive Resolvers**), which are local recursive resolvers that use DoT or DoH to resolve queries for clients. Time will tell how these developments ultimately reshape the DNS architecture.

Even DoT and DoH do not completely resolve all DNS-related privacy concerns, because the operator of the local resolver must still be trusted with sensitive information: namely, the DNS queries and the IP addresses of the clients that issued those queries. Other recent enhancements to DNS and DoH have been proposed, including **oblivious DNS** (Schmitt et al., 2019) and **oblivious DoH** (Kinnear et al., 2019), whereby the stub resolver encrypts the original query to the local recursive resolver, which in turn sends the encrypted query to an authoritative name serve that can decrypt and resolve the query, but does not know the identity or IP address of the stub resolver that initiated the query. Figure 7-8 shows this relationship.

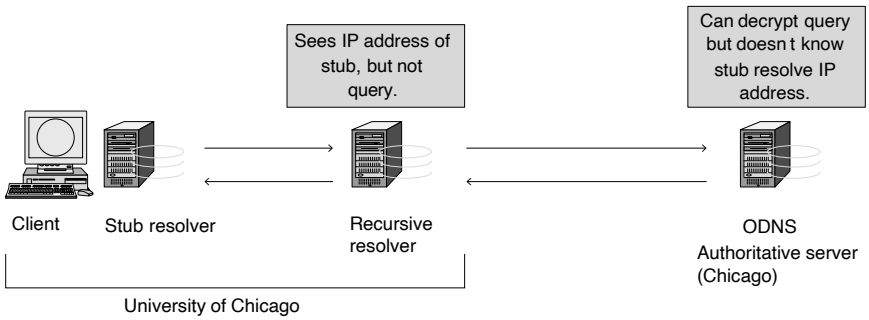

Figure 7-8. Oblivious DNS.

Most of these implementations are still nascent, in the forms of early prototypes and draft standards being discussed in the DNS privacy working group at IETF.

7.1.8 Contention Over Names

As the Internet has become more commercial and more international, it has also become more contentious, especially in matters related to naming. This controversy includes ICANN itself. For example, the creation of the *xxx* domain took several years and court cases to resolve. Is voluntarily placing adult content in its own domain a good or a bad thing? (Some people did not want adult content available at all on the Internet while others wanted to put it all in one domain so nanny filters could easily find and block it from children.) Some of the domains self-organize, while others have restrictions on who can obtain a name, as noted in Fig. 7-8. But what restrictions are appropriate? Take the *pro* domain, for example. It is for qualified professionals. But who, exactly, is a professional? Doctors and lawyers clearly are professionals. But what about freelance photographers, piano teachers, magicians, plumbers, barbers, exterminators, tattoo artists, mercenaries, and prostitutes? Are these occupations eligible? According to whom?

There is also money in names. Tuvalu (a tiny island country midway between Hawaii and Australia) sold a lease on its *tv* domain for $50 million, all because the country code is well-suited to advertising television sites. Virtually every common (English) word has been taken in the *com* domain, along with the most common misspellings. Try household articles, animals, plants, body parts, etc. The practice of registering a domain only to turn around and sell it off to an interested party at a much higher price even has a name. It is called **cybersquatting**. Many companies that were slow off the mark when the Internet era began found their obvious domain names already taken when they tried to acquire them. In general, as long as no trademarks are being violated and no fraud is involved, it is first-come, first-served with names. Nevertheless, policies to resolve naming disputes are still being refined.

7.2 ELECTRONIC MAIL

Electronic mail, or more commonly **email**, has been around for over four decades. Faster and cheaper than paper mail, email has been a popular application since the early days of the Internet. Before 1990, it was mostly used in academia. During the 1990s, it became known to the public at large and grew exponentially, to the point where the number of emails sent per day now is vastly more than the number of **snail mail** (i.e., paper) letters. Other forms of network communication, such as instant messaging and voice-over-IP calls have expanded greatly in use over the past decade, but email remains the workhorse of Internet communication. It is widely used within industry for intracompany communication, for example, to allow far-flung employees all over the world to cooperate on complex projects. Unfortunately, like paper mail, the majority of email—some 9 out of 10 messages—is junk mail or **spam**. While mail systems can remove much of it nowadays, a lot still gets through and research into detecting it all is ongoing, for example, see Dan et al. (2019) and Zhang et al. (2019).

Email, like most other forms of communication, has developed its own conventions and styles. It is very informal and has a low threshold of use. People who would never dream of calling up or even writing a letter to a Very Important Person do not hesitate for a second to send a sloppily written email to him or her. By eliminating most cues associated with rank, age, and gender, email debates often focus on content, not status. With email, a brilliant idea from a summer student can have more impact than a dumb one from an executive vice president.

Email is full of jargon such as BTW (By The Way), ROTFL (Rolling On The Floor Laughing), and IMHO (In My Humble Opinion). Many people also use little ASCII symbols called **smileys**, starting with the ubiquitous ''**:-)**''. This symbol and other **emoticons** help to convey the tone of the message. They have spread to other terse forms of communication, such as instant messaging, typically as graphical **emoji**. Many smartphones have hundreds of emojis available.

The email protocols have evolved during the period of their use, too. The first email systems simply consisted of file transfer protocols, with the convention that the first line of each message (i.e., file) contained the recipient's address. As time went on, email diverged from file transfer and many features were added, such as the ability to send one message to a list of recipients. Multimedia capabilities became important in the 1990s to send messages with images and other non-text material. Programs for reading email became much more sophisticated too, shifting from text-based to graphical user interfaces and adding the ability for users to access their mail from their laptops wherever they happen to be. Finally, with the prevalence of spam, email systems now pay attention to finding and removing unwanted email.

In our description of email, we will focus on the way that mail messages are moved between users, rather than the look and feel of mail reader programs. Nevertheless, after describing the overall architecture, we will begin with the user-facing part of the email system, as it is familiar to most readers.

7.2.1 Architecture and Services

In this section, we will provide an overview of how email systems are organized and what they can do. The architecture of the email system is shown in Fig. 7-9. It consists of two kinds of subsystems: the **user agents**, which allow people to read and send email, and the **message transfer agents**, which move the messages from the source to the destination. We will also refer to message transfer agents informally as **mail servers**.

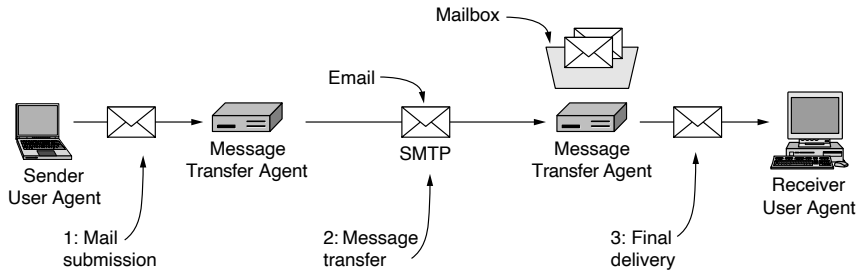

Figure 7-9. Architecture of the email system.

The user agent is a program that provides a graphical interface, or sometimes a text- and command-based interface that lets users interact with the email system. It includes a means to compose messages and replies to messages, display incoming messages, and organize messages by filing, searching, and discarding them. The act of sending new messages into the mail system is called **mail submission**.

Some of the user agent processing may be done automatically, anticipating what the user wants. For example, incoming mail may be filtered to extract or deprioritize messages that are likely spam. Some user agents include advanced features, such as arranging for automatic email responses (''I'm having a wonderful vacation and it will be a while before I get back to you.''). A user agent runs on the same computer on which a user reads her mail. It is just another program and may be run only some of the time.

The message transfer agents are typically system processes. They run in the background on mail server machines and are intended to be always available. Their job is to automatically move email through the system from the originator to the recipient with SMTP (Simple Mail Transfer Protocol), discussed in Sec. 7.2.4. This is the message transfer step.

SMTP was originally specified as RFC 821 and revised to become the current RFC 5321. It sends mail over connections and reports back the delivery status and any errors. Numerous applications exist in which confirmation of delivery is important and may even have legal significance (''Well, Your Honor, my email system is just not very reliable, so I guess the electronic subpoena just got lost somewhere'').

Message transfer agents also implement **mailing lists**, in which an identical copy of a message is delivered to everyone on a list of email addresses. Additional advanced features are carbon copies, blind carbon copies, high-priority email, secret (encrypted) email, alternative recipients if the primary one is not currently available, and the ability for assistants to read and answer their bosses' email.

Linking user agents and message transfer agents are the concepts of mailboxes and a standard format for email messages. **Mailboxes** store the email that is received for a user. They are maintained by mail servers. User agents simply present users with a view of the contents of their mailboxes. To do this, the user agents send the mail servers commands to manipulate the mailboxes, inspecting their contents, deleting messages, and so on. The retrieval of mail is the final delivery (step 3) in Fig. 7-9. With this architecture, one user may use different user agents on multiple computers to access one mailbox.

Mail is sent between message transfer agents in a standard format. The original format, RFC 822, has been revised to the current RFC 5322 and extended with support for multimedia content and international text. This scheme is called MIME. People still refer to Internet email as RFC 822, though.

A key idea in the message format is the clear distinction between the **envelope** and the contents of the envelope. The envelope encapsulates the message. Furthermore, it contains all the information needed for transporting the message, such as the destination address, priority, and security level, all of which are distinct from the message itself. The message transport agents use the envelope for routing, just as the post office does.

The message inside the envelope consists of two separate parts: the **header** and the **body**. The header contains control information for the user agents. The body

is entirely for the human recipient. None of the agents care much about it. Envelopes and messages are illustrated in Fig. 7-10.

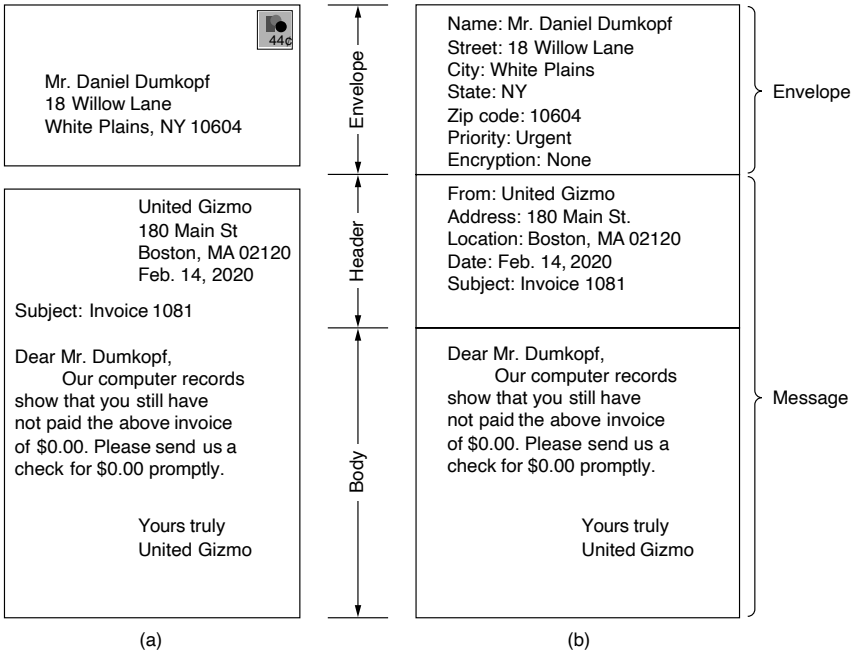

Figure 7-10. Envelopes and messages. (a) Paper mail. (b) Electronic mail.

We will examine the pieces of this architecture in more detail by looking at the steps that are involved in sending email from one user to another. This journey starts with the user agent.

7.2.2 The User Agent

A user agent is a program (sometimes called an **email reader**) that accepts a variety of commands for composing, receiving, and replying to messages, as well as for manipulating mailboxes. There are many popular user agents, including Google Gmail, Microsoft Outlook, Mozilla Thunderbird, and Apple Mail. They can vary greatly in their appearance. Most user agents have a menu- or icon-driven graphical interface that requires a mouse, or a touch interface on smaller mobile devices. Older user agents, such as Elm, mh, and Pine, provide text-based interfaces and expect one-character commands from the keyboard. Functionally, these are the same, at least for text messages.

The typical elements of a user agent interface are shown in Fig. 7-11. Your mail reader is likely to be much flashier, but probably has equivalent functions. When a user agent is started, it will usually present a summary of the messages in the user's mailbox. Often, the summary will have one line for each message in some sorted order. It highlights key fields of the message that are extracted from the message envelope or header.

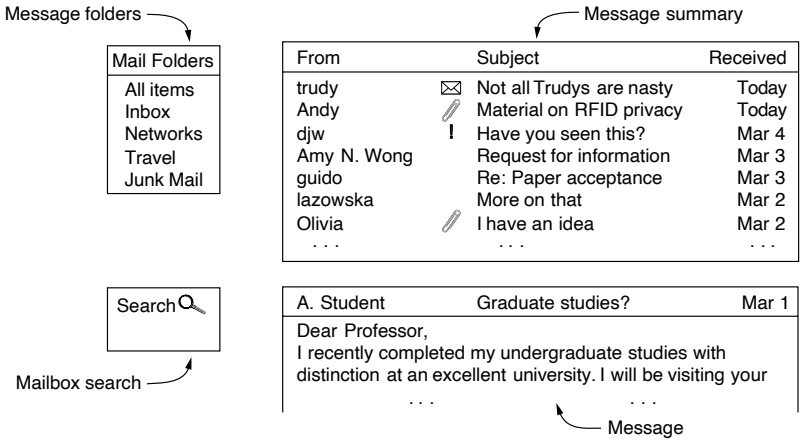

Figure 7-11. Typical elements of the user agent interface.

Seven summary lines are shown in the example of Fig. 7-11. The lines use the *From*, *Subject*, and *Received* fields, in that order, to display who sent the message, what it is about, and when it was received. All the information is formatted in a user-friendly way rather than displaying the literal contents of the message fields, but it is based on the message fields. Thus, people who fail to include a *Subject* field often discover that responses to their emails tend not to get the highest priority.

Many other fields or indications are possible. The icons next to the message subjects in Fig. 7-11 might indicate, for example, unread mail (the envelope), attached material (the paperclip), and important mail, at least as judged by the sender (the exclamation point).

Many sorting orders are also possible. The most common is to order messages based on the time that they were received, most recent first, with some indication as to whether the message is new or has already been read by the user. The fields in the summary and the sort order can be customized by the user according to her preferences.

User agents must also be able to display incoming messages as needed so that people can read their email. Often a short preview of a message is provided, as in

Fig. 7-11, to help users decide when to read further and when to hit the SPAM button. Previews may use small icons or images to describe the contents of the message. Other presentation processing includes reformatting messages to fit the display, and translating or converting contents to more convenient formats (e.g., digitized speech to recognized text).

After a message has been read, the user can decide what to do with it. This is called **message disposition**. Options include deleting the message, sending a reply, forwarding the message to another user, and keeping the message for later reference. Most user agents can manage one mailbox for incoming mail with multiple folders for saved mail. The folders allow the user to save message according to sender, topic, or some other category.

Filing can be done automatically by the user agent as well, even before the user reads the messages. A common example is that the fields and contents of messages are inspected and used, along with feedback from the user about previous messages, to determine if a message is likely to be spam. Many ISPs and companies run software that labels mail as important or spam so that the user agent can file it in the corresponding mailbox. The ISP and company have the advantage of seeing mail for many users and may have lists of known spammers. If hundreds of users have just received a similar message, it is probably spam, although it could be a message from the CEO to all employees. By presorting incoming mail as ''probably legitimate'' and ''probably spam,'' the user agent can save users a fair amount of work separating the good stuff from the junk.

And the most popular spam? It is generated by collections of compromised computers called **botnets** and its content depends on where you live. Fake diplomas are common in Asia, and cheap drugs and other dubious product offers are common in the U.S. Unclaimed Nigerian bank accounts still abound. Pills for enlarging various body parts are common everywhere.

Other filing rules can be constructed by users. Each rule specifies a condition and an action. For example, a rule could say that any message received from the boss goes to one folder for immediate reading and any message from a particular mailing list goes to another folder for later reading. Several folders are shown in Fig. 7-11. The most important folders are the Inbox, for incoming mail not filed elsewhere, and Junk Mail, for messages that are thought to be spam.

7.2.3 Message Formats

Now we turn from the user interface to the format of the email messages themselves. Messages sent by the user agent must be placed in a standard format to be handled by the message transfer agents. First we will look at basic ASCII email using RFC 5322, which is the latest revision of the original Internet message format as described in RFC 822 and its many updates. After that, we will look at multimedia extensions to the basic format.

RFC 5322—The Internet Message Format

Messages consist of a primitive envelope (described as part of SMTP in RFC 5321), some number of header fields, a blank line, and then the message body. Each header field (logically) consists of a single line of ASCII text containing the field name, a colon, and, for most fields, a value. The original RFC 822 was designed decades ago and did not clearly distinguish the envelope fields from the header fields. Although it has been revised to RFC 5322, completely redoing it was not possible due to its widespread usage. In normal usage, the user agent builds a message and passes it to the message transfer agent, which then uses some of the header fields to construct the actual envelope, a somewhat old-fashioned mixing of message and envelope.

The principal header fields related to message transport are listed in Fig. 7-12. The *To:* field gives the email address of the primary recipient. Having multiple recipients is also allowed. The *Cc:* field gives the addresses of any secondary recipients. In terms of delivery, there is no distinction between the primary and secondary recipients. It is entirely a psychological difference that may be important to the people involved but is not important to the mail system. The term *Cc:* (Carbon copy) is a bit dated, since computers do not use carbon paper, but it is well established. The *Bcc:* (Blind carbon copy) field is like the *Cc:* field, except that this line is deleted from all the copies sent to the primary and secondary recipients. This feature allows people to send copies to third parties without the primary and secondary recipients knowing this.

Header	Meaning
To:	Email address(es) of primary recipient(s)
Cc:	Email address(es) of secondary recipient(s)
Bcc:	Email address(es) for blind carbon copies
From:	Person or people who created the message
Sender:	Email address of the actual sender
Received:	Line added by each transfer agent along the route
Return-Path:	Can be used to identify a path back to the sender

Figure 7-12. RFC 5322 header fields related to message transport.

The next two fields, *From:* and *Sender:*, tell who wrote and actually sent the message, respectively. These two fields need not be the same. For example, a business executive may write a message, but her assistant may be the one who actually transmits it. In this case, the executive would be listed in the *From:* field and the assistant in the *Sender:* field. The *From:* field is required, but the *Sender:* field may be omitted if it is the same as the *From:* field. These fields are needed in case the message is undeliverable and must be returned to the sender.

A line containing *Received:* is added by each message transfer agent along the way. The line contains the agent's identity, the date and time the message was received, and other information that can be used for debugging the routing system.

The *Return-Path:* field is added by the final message transfer agent and was intended to tell how to get back to the sender. In theory, this information can be gathered from all the *Received:* headers (except for the name of the sender's mailbox), but it is rarely filled in as such and typically just contains the sender's address.

In addition to the fields of Fig. 7-12, RFC 5322 messages may also contain a variety of header fields used by the user agents or human recipients. The most common ones are listed in Fig. 7-13. Most of these are self-explanatory, so we will not go into all of them in much detail.

Header	Meaning
Date:	The date and time the message was sent
Reply-To:	Email address to which replies should be sent
Message-Id:	Unique number for referencing this message later
In-Reply-To:	Message-Id of the message to which this is a reply
References:	Other relevant Message-Ids
Keywords:	User-chosen keywords
Subject:	Short summary of the message for the one-line display

Figure 7-13. Some fields used in the RFC 5322 message header.

The *Reply-To:* field is sometimes used when neither the person composing the message nor the person sending the message wants to see the reply. For example, a marketing manager may write an email message telling customers about a new product. The message is sent by an assistant, but the *Reply-To:* field lists the head of the sales department, who can answer questions and take orders. This field is also useful when the sender has two email accounts and wants the reply to go to the other one.

The *Message-Id:* is an automatically generated number that is used to link messages together (e.g., when used in the *In-Reply-To:* field) and to prevent duplicate delivery.

The RFC 5322 document explicitly says that users are allowed to invent optional headers for their own private use. By convention since RFC 822, these headers start with the string *X-*. It is guaranteed that no future headers will use names starting with *X-*, to avoid conflicts between official and private headers. Sometimes wiseguy undergraduates make up fields like *X-Fruit-of-the-Day:* or *X-Disease-of-the-Week:*, which are legal, although not always illuminating.

After the headers comes the message body. Users can put whatever they want here. Some people terminate their messages with elaborate signatures, including quotations from greater and lesser authorities, political statements, and disclaimers

of all kinds (e.g., The XYZ Corporation is not responsible for my opinions; in fact, it cannot even comprehend them).

MIME—The Multipurpose Internet Mail Extensions

In the early days of the ARPANET, email consisted exclusively of text messages written in English and expressed in ASCII. For this environment, the early RFC 822 format did the job completely: it specified the headers but left the content entirely up to the users. In the 1990s, the worldwide use of the Internet and demand to send richer content through the mail system meant that this approach was no longer adequate. The problems included sending and receiving messages in languages with diacritical marks (e.g., French and German), non-Latin alphabets (e.g., Hebrew and Russian), or no alphabets (e.g., Chinese and Japanese), as well as sending messages not containing text at all (e.g., audio, images, or binary documents and programs).

The solution was the development of **MIME** (**Multipurpose Internet Mail Extensions**). It is widely used for mail messages that are sent across the Internet, as well as to describe content for other applications such as Web browsing. MIME is described in RFC 2045, and the ones following it as well as RFC 4288 and 4289.

The basic idea of MIME is to continue to use the RFC 822 format but to add structure to the message body and define encoding rules for the transfer of non-ASCII messages. Not deviating from RFC 822 allowed MIME messages to be sent using the existing mail transfer agents and protocols (based on RFC 821 then, and RFC 5321 now). All that had to be changed were the sending and receiving programs, which users could do for themselves.

MIME defines five new message headers, as shown in Fig. 7-14. The first of these simply tells the user agent receiving the message that it is dealing with a MIME message, and which version of MIME it uses. Any message not containing a *MIME-Version:* header is assumed to be an English plaintext message (or at least one using only ASCII characters) and is processed as such.

Header	Meaning
MIME-Version:	Identifies the MIME version
Content-Description:	Human-readable string telling what is in the message
Content-Id:	Unique identifier
Content-Transfer-Encoding:	How the body is wrapped for transmission
Content-Type:	Type and format of the content

Figure 7-14. Message headers added by MIME.

The *Content-Description:* header is an ASCII string telling what is in the message. This header is needed so the recipient will know whether it is worth decoding and reading the message. If the string says ''Photo of Aron's hamster'' and the

person getting the message is not a big hamster fan, the message will probably be discarded rather than decoded into a high-resolution color photograph.

The *Content-Id:* header identifies the content. It uses the same format as the standard *Message-Id:* header.

The *Content-Transfer-Encoding:* tells how the body is wrapped for transmission through the network. A key problem at the time MIME was developed was that the mail transfer (SMTP) protocols expected ASCII messages in which no line exceeded 1000 characters. ASCII characters use 7 bits out of each 8-bit byte. Binary data such as executable programs and images use all 8 bits of each byte, as do extended character sets. There was no guarantee this data would be transferred safely. Hence, some method of carrying binary data that made it look like a regular ASCII mail message was needed. Extensions to SMTP since the development of MIME do allow 8-bit binary data to be transferred, though even today binary data may not always go through the mail system correctly if unencoded.

MIME provides five transfer encoding schemes, plus an escape to new schemes—just in case. The simplest scheme is just ASCII text messages. ASCII characters use 7 bits and can be carried directly by the email protocol, provided that no line exceeds 1000 characters.

The next simplest scheme is the same thing, but using 8-bit characters, that is, all values from 0 up to and including 255 are allowed. Messages using the 8-bit encoding must still adhere to the standard maximum line length.

Then there are messages that use a true binary encoding. These are arbitrary binary files that not only use all 8 bits but also do not adhere to the 1000-character line limit. Executable programs fall into this category. Nowadays, mail servers can negotiate to send data in binary (or 8-bit) encoding, falling back to ASCII if both ends do not support the extension.

The ASCII encoding of binary data is called **base64 encoding**. In this scheme, groups of 24 bits are broken up into four 6-bit units, with each unit being sent as a legal ASCII character. The coding is "A" for 0, "B" for 1, and so on, followed by the 26 lowercase letters, the 10 digits, and finally + and / for 62 and 63, respectively. The == and = sequences indicate that the last group contained only 8 or 16 bits, respectively. Carriage returns and line feeds are ignored, so they can be inserted at will in the encoded character stream to keep the lines short enough. Arbitrary binary text can be sent safely using this scheme, albeit inefficiently. This encoding was very popular before binary-capable mail servers were widely deployed. It is still commonly seen.

The last header shown in Fig. 7-14 is really the most interesting one. It specifies the nature of the message body and has had an impact well beyond email. For instance, content downloaded from the Web is labeled with MIME types so that the browser knows how to present it. So is content sent over streaming media and real-time transports such as voice over IP.

Initially, seven MIME types were defined in RFC 1521. Each type has one or more available subtypes. The type and subtype are separated by a slash, as in

''Content-Type: video/mpeg''. Since then, over 2700 subtypes have been added, along two new types (font and model). Additional entries are being added all the time as new types of content are developed. The list of assigned types and subtypes is maintained online by IANA at *www.iana.org/assignments/media-types*. The types, along with several examples of commonly used subtypes, are given in Fig. 7-15.

Type	Example subtypes	Description
text	plain, html, xml, css	Text in various formats
image	gif, jpeg, tiff	Pictures
audio	basic, mpeg, mp4	Sounds
video	mpeg, mp4, quicktime	Movies
font	otf, ttf Fonts for typesetting	
model	vrml	3D model
application	octet-stream, pdf, javascript, zip	Data produced by applications
message	http, RFC 822	Encapsulated message
multipart	mixed, alternative, parallel, digest	Combination of multiple types

Figure 7-15. MIME content types and example subtypes.

The MIME types in Fig. 7-15 should be self-explanatory except perhaps the last one. It allows a message with multiple attachments, each with a different MIME type.

7.2.4 Message Transfer

Now that we have described user agents and mail messages, we are ready to look at how the message transfer agents relay messages from the originator to the recipient. The mail transfer is done with the SMTP protocol.

The simplest way to move messages is to establish a transport connection from the source machine to the destination machine and then just transfer the message. This is how SMTP originally worked. Over the years, however, two different uses of SMTP have been differentiated. The first use is **mail submission**, step 1 in the email architecture of Fig. 7-9. This is the means by which user agents send messages into the mail system for delivery. The second use is to transfer messages between message transfer agents (step 2 in Fig. 7-9). This sequence delivers mail all the way from the sending to the receiving message transfer agent in one hop. Final delivery is accomplished with different protocols that we will describe in the next section.

In this section, we will describe the basics of the SMTP protocol and its extension mechanism. Then we will discuss how it is used differently for mail submission and message transfer.

SMTP (Simple Mail Transfer Protocol) and Extensions

Within the Internet, email is delivered by having the sending computer establish a TCP connection to port 25 of the receiving computer. Listening to this port is a mail server that speaks **SMTP** (**Simple Mail Transfer Protocol**). This server accepts incoming connections, subject to some security checks, and accepts messages for delivery. If a message cannot be delivered, an error report containing the first part of the undeliverable message is returned to the sender.

SMTP is a simple ASCII protocol. This is not a weakness but a feature. Using ASCII text makes protocols easy to develop, test, and debug. They can be tested by sending commands manually, and records of the messages are easy to read. Most application-level Internet protocols now work this way (e.g., HTTP).

We will walk through a simple message transfer between mail servers that delivers a message. After establishing the TCP connection to port 25, the sending machine, operating as the client, waits for the receiving machine, operating as the server, to talk first. The server starts by sending a line of text giving its identity and telling whether it is prepared to receive mail. If it is not, the client releases the connection and tries again later.

If the server is willing to accept email, the client announces whom the email is coming from and whom it is going to. If such a recipient exists at the destination, the server gives the client the go-ahead to send the message. Then the client sends the message and the server acknowledges it. No checksums are needed because TCP provides a reliable byte stream. If there is more email, that is now sent. When all the email has been exchanged in both directions, the connection is released. A sample dialog is shown in Fig. 7-16. The lines sent by the client (i.e., the sender) are marked *C:*. Those sent by the server (i.e., the receiver) are marked *S:*.

The first command from the client is indeed meant to be *HELO*. Of the various four-character abbreviations for *HELLO*, this one has numerous advantages over its biggest competitor. Why all the commands had to be four characters has been lost in the mists of time.

In Fig. 7-16, the message is sent to only one recipient, so only one *RCPT* command is used. Such commands are allowed to send a single message to multiple receivers. Each one is individually acknowledged or rejected. Even if some recipients are rejected (because they do not exist at the destination), the message can be sent to the other ones.

Finally, although the syntax of the four-character commands from the client is rigidly specified, the syntax of the replies is less rigid. Only the numerical code really counts. Each implementation can put whatever string it wants after the code.

The basic SMTP works well, but it is limited in several respects. It does not include authentication. This means that the *FROM* command in the example could give any sender address that it pleases. This is quite useful for sending spam. Another limitation is that SMTP transfers ASCII messages, not binary data. This is

```
                    S: 220 ee.uwa.edu.au SMTP service ready
C: HELO abcd.com
                    S: 250 cs.uchicago.edu says hello to ee.uwa.edu.au
C: MAIL FROM: <alice@cs.uchicago.edu>
                    S: 250 sender ok
C: RCPT TO: <bob@ee.uwa.edu.au>
                    S: 250 recipient ok
C: DATA
                    S: 354 Send mail; end with "." on a line by itself
C: From: alice@cs.uchicago.edu
C: To: bob@ee.uwa.edu.au
C: MIME-Version: 1.0
C: Message-Id: <0704760941.AA00747@ee.uwa.edu.au>
C: Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
C: Subject: Earth orbits sun integral number of times
C:
C: This is the preamble. The user agent ignores it. Have a nice day.
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: text/html
C:
C: <p>Happy birthday to you
C: Happy birthday to you
C: Happy birthday dear <bold> Bob </bold>
C: Happy birthday to you
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: message/external-body;
C:     access-type="anon-ftp";
C:     site="bicycle.cs.uchicago.edu";
C:     directory="pub";
C:     name="birthday.snd"
C:
C: content-type: audio/basic
C: content-transfer-encoding: base64
C: --qwertyuiopasdfghjklzxcvbnm
C: .
                    S: 250 message accepted
C: QUIT
                    S: 221 ee.uwa.edu.au closing connection
```

Figure 7-16. A message from *alice cs.uchicago.edu* to *bob ee.uwa.edu.au*.

why the base64 MIME content transfer encoding was needed. However, with that encoding the mail transmission uses bandwidth inefficiently, which is an issue for large messages. A third limitation is that SMTP sends messages in the clear. It has no encryption to provide a measure of privacy against prying eyes.

To allow these and many other problems related to message processing to be addressed, SMTP was revised to have an extension mechanism. This mechanism

is a mandatory part of the RFC 5321 standard. The use of SMTP with extensions is called **ESMTP** (**Extended SMTP**).

Clients wanting to use an extension send an *EHLO* message instead of *HELO* initially. If this is rejected, the server is a regular SMTP server, and the client should proceed in the usual way. If the *EHLO* is accepted, the server replies with the extensions that it supports. The client may then use any of these extensions. Several common extensions are shown in Fig. 7-17. The figure gives the keyword as used in the extension mechanism, along with a description of the new functionality. We will not go into extensions in further detail.

Keyword	Description
AUTH	Client authentication
BINARYMIME	Server accepts binary messages
CHUNKING	Server accepts large messages in chunks
SIZE	Check message size before trying to send
STARTTLS	Switch to secure transport (TLS; see Chap. 8)
UTF8SMTP	Internationalized addresses

Figure 7-17. Some SMTP extensions.

To get a better feel for how SMTP and some of the other protocols described in this chapter work, try them out. In all cases, first go to a machine connected to the Internet. On a UNIX (or Linux) system, in a shell, type

```
telnet mail.isp.com 25
```

substituting the DNS name of your ISP's mail server for *mail.isp.com*. On a Windows machine, you may have to first install the telnet program (or equivalent) and then start it yourself. This command will establish a telnet (i.e., TCP) connection to port 25 on that machine. Port 25 is the SMTP port; see Fig. 6-34 for the ports for other common protocols. You will probably get a response something like this:

```
Trying 192.30.200.66...
Connected to mail.isp.com
Escape character is '^]'.
220 mail.isp.com Smail #74 ready at Thu, 25 Sept 2019 13:26 +0200
```

The first three lines are from telnet, telling you what it is doing. The last line is from the SMTP server on the remote machine, announcing its willingness to talk to you and accept email. To find out what commands it accepts, type

```
HELP
```

From this point on, a command sequence such as the one in Fig. 7-16 is possible if the server is willing to accept mail from you. You may have to type quickly, though, since the connection may time out if it is inactive too long. Also, not every mail server will accept a telnet connection from an unknown machine.

Mail Submission

Originally, user agents ran on the same computer as the sending message transfer agent. In this setting, all that is required to send a message is for the user agent to talk to the local mail server, using the dialog that we have just described. However, this setting is no longer the usual case.

User agents often run on laptops, home PCs, and mobile phones. They are not always connected to the Internet. Mail transfer agents run on ISP and company servers. They are always connected to the Internet. This difference means that a user agent in Boston may need to contact its regular mail server in Seattle to send a mail message because the user is traveling.

By itself, this remote communication poses no problem. It is exactly what the TCP/IP protocols are designed to support. However, an ISP or company usually does not want any remote user to be able to submit messages to its mail server to be delivered elsewhere. The ISP or company is not running the server as a public service. In addition, this kind of **open mail relay** attracts spammers. This is because it provides a way to launder the original sender and thus make the message more difficult to identify as spam.

Given these considerations, SMTP is normally used for mail submission with the *AUTH* extension. This extension lets the server check the credentials (username and password) of the client to confirm that the server should be providing mail service.

There are several other differences in the way SMTP is used for mail submission. For example, port 587 can be used in preference to port 25 and the SMTP server can check and correct the format of the messages sent by the user agent. For more information about the restricted use of SMTP for mail submission, please see RFC 4409.

Physical Transfer

Once the sending mail transfer agent receives a message from the user agent, it will deliver it to the receiving mail transfer agent using SMTP. To do this, the sender uses the destination address. Consider the message in Fig. 7-16, addressed to *bob@ee.uwa.edu.au*. To what mail server should the message be delivered?

To determine the correct mail server to contact, DNS is consulted. In the previous section, we described how DNS contains multiple types of records, including the *MX*, or mail exchanger, record. In this case, a DNS query is made for the *MX* records of the domain *ee.uwa.edu.au*. This query returns an ordered list of the names and IP addresses of one or more mail servers.

The sending mail transfer agent then makes a TCP connection on port 25 to the IP address of the mail server to reach the receiving mail transfer agent, and uses SMTP to relay the message. The receiving mail transfer agent will then place mail for the user *bob* in the correct mailbox for Bob to read it at a later time. This local

delivery step may involve moving the message among computers if there is a large mail infrastructure.

With this delivery process, mail travels from the initial to the final mail transfer agent in a single hop. There are no intermediate servers in the message transfer stage. It is possible, however, for this delivery process to occur multiple times. One example that we have described already is when a message transfer agent implements a mailing list. In this case, a message is received for the list. It is then expanded as a message to each member of the list that is sent to the individual member addresses.

As another example of relaying, Bob may have graduated from M.I.T. and also be reachable via the address *bob@alum.mit.edu*. Rather than reading mail on multiple accounts, Bob can arrange for mail sent to this address to be forwarded to *bob@ee.uwa.edu*. In this case, mail sent to *bob@alum.mit.edu* will undergo two deliveries. First, it will be sent to the mail server for *alum.mit.edu*. Then, it will be sent to the mail server for *ee.uwa.edu.au*. Each of these legs is a complete and separate delivery as far as the mail transfer agents are concerned.

7.2.5 Final Delivery

Our mail message is almost delivered. It has arrived at Bob's mailbox. All that remains is to transfer a copy of the message to Bob's user agent for display. This is step 3 in the architecture of Fig. 7-9. This task was straightforward in the early Internet, when the user agent and mail transfer agent ran on the same machine as different processes. The mail transfer agent simply wrote new messages to the end of the mailbox file, and the user agent simply checked the mailbox file for new mail.

Nowadays, the user agent on a PC, laptop, or mobile, is likely to be on a different machine than the ISP or company mail server and certain to be on a different machine for a mail provider such as Gmail. Users want to be able to access their mail remotely, from wherever they are. They want to access email from work, from their home PCs, from their laptops when on business trips, and from cybercafes when on so-called vacation. They also want to be able to work offline, then reconnect to receive incoming mail and send outgoing mail. Moreover, each user may run several user agents depending on what computer it is convenient to use at the moment. Several user agents may even be running at the same time.

In this setting, the job of the user agent is to present a view of the contents of the mailbox, and to allow the mailbox to be remotely manipulated. Several different protocols can be used for this purpose, but SMTP is not one of them. SMTP is a push-based protocol. It takes a message and connects to a remote server to transfer the message. Final delivery cannot be achieved in this manner both because the mailbox must continue to be stored on the mail transfer agent and because the user agent may not be connected to the Internet at the moment that SMTP attempts to relay messages.

IMAP—The Internet Message Access Protocol

One of the main protocols that is used for final delivery is **IMAP** (**Internet Message Access Protocol**). Version 4 of the protocol is defined in RFC 3501 and in its many updates. To use IMAP, the mail server runs an IMAP server that listens to port 143. The user agent runs an IMAP client. The client connects to the server and begins to issue commands from those listed in Fig. 7-18.

Command	Description
CAPABILITY	List server capabilities
STARTTLS	Start secure transport (TLS; see Chap. 8)
LOGIN	Log on to server
AUTHENTICATE	Log on with other method
SELECT	Select a folder
EXAMINE	Select a read-only folder
CREATE	Create a folder
DELETE	Delete a folder
RENAME	Rename a folder
SUBSCRIBE	Add folder to active set
UNSUBSCRIBE	Remove folder from active set
LIST	List the available folders
LSUB	List the active folders
STATUS	Get the status of a folder
APPEND	Add a message to a folder
CHECK	Get a checkpoint of a folder
FETCH	Get messages from a folder
SEARCH	Find messages in a folder
STORE	Alter message flags
COPY	Make a copy of a message in a folder
EXPUNGE	Remove messages flagged for deletion
UID	Issue commands using unique identifiers
NOOP	Do nothing
CLOSE	Remove flagged messages and close folder
LOGOUT	Log out and close connection

Figure 7-18. IMAP (version 4) commands.

First, the client will start a secure transport if one is to be used (in order to keep the messages and commands confidential), and then log in or otherwise authenticate itself to the server. Once logged in, there are many commands to list folders and messages, fetch messages or even parts of messages, mark messages

with flags for later deletion, and organize messages into folders. To avoid confusion, please note that we use the term ''folder'' here to be consistent with the rest of the material in this section, in which a user has a single mailbox made up of multiple folders. However, in the IMAP specification, the term *mailbox* is used instead. One user thus has many IMAP mailboxes, each of which is typically presented to the user as a folder.

IMAP has many other features, too. It has the ability to address mail not by message number, but by using attributes (e.g., give me the first message from Alice). Searches can be performed on the server to find the messages that satisfy certain criteria so that only those messages are fetched by the client.

IMAP is an improvement over an earlier final delivery protocol, **POP3** (**Post Office Protocol, version 3**), which is specified in RFC 1939. POP3 is a simpler protocol but supports fewer features and is less secure in typical usage. Mail is usually downloaded to the user agent computer, instead of remaining on the mail server. This makes life easier on the server, but harder on the user. It is not easy to read mail on multiple computers, plus if the user agent computer breaks, all email may be lost permanently. Nonetheless, you will still find POP3 in use.

Proprietary protocols can also be used because the protocol runs between a mail server and user agent that can be supplied by the same company. Microsoft Exchange is a mail system with a proprietary protocol.

Webmail

An increasingly popular alternative to IMAP and SMTP for providing email service is to use the Web as an interface for sending and receiving mail. Widely used **Webmail** systems include Google Gmail, Microsoft Hotmail and Yahoo! Mail. Webmail is one example of software (in this case, a mail user agent) that is provided as a service using the Web.

In this architecture, the provider runs mail servers as usual to accept messages for users with SMTP on port 25. However, the user agent is different. Instead of being a standalone program, it is a user interface that is provided via Web pages. This means that users can use any browser they like to access their mail and send new messages.

When the user goes to the email Web page of the provider, say, Gmail, a form is presented in which the user is asked for a login name and password. The login name and password are sent to the server, which then validates them. If the login is successful, the server finds the user's mailbox and builds a Web page listing the contents of the mailbox on the fly. The Web page is then sent to the browser for display.

Many of the items on the page showing the mailbox are clickable, so messages can be read, deleted, and so on. To make the interface responsive, the Web pages will often include JavaScript programs. These programs are run locally on the client in response to local events (e.g., mouse clicks) and can also download and

upload messages in the background, to prepare the next message for display or a new message for submission. In this model, mail submission happens using the normal Web protocols by posting data to a URL. The Web server takes care of injecting messages into the traditional mail delivery system that we have described. For security, the standard Web protocols can be used as well. These protocols concern themselves with encrypting Web pages, not whether the content of the Web page is a mail message.

7.3 THE WORLD WIDE WEB

The Web, as the World Wide Web is popularly known, is an architectural framework for accessing linked content spread out over millions of machines all over the Internet. In 10 years it went from being a way to coordinate the design of high-energy physics experiments in Switzerland to the application that millions of people think of as being ''The Internet.'' Its enormous popularity stems from the fact that it is easy for beginners to use and provides access with a rich graphical interface to an enormous wealth of information on almost every conceivable subject, from aardvarks to Zulus.

The Web began in 1989 at CERN, the European Center for Nuclear Research. The initial idea was to help large teams, often with members in a dozen or more countries and time zones, collaborate using a constantly changing collection of reports, blueprints, drawings, photos, and other documents produced by experiments in particle physics. The proposal for a Web of linked documents came from CERN physicist Tim Berners-Lee. The first (text-based) prototype was operational 18 months later. A public demonstration given at the Hypertext '91 conference caught the attention of other researchers, which led Marc Andreessen at the University of Illinois to develop the first graphical browser. It was called Mosaic and released in February 1993.

The rest, as they say, is now history. Mosaic was so popular that a year later Andreessen left to form a company, Netscape Communications Corp., whose goal was to develop Web software. For the next three years, Netscape Navigator and Microsoft's Internet Explorer engaged in a ''browser war,'' each one trying to capture a larger share of the new market by frantically adding more features (and thus more bugs) than the other one.

Through the 1990s and 2000s, Web sites and Web pages, as Web content is called, grew exponentially until there were millions of sites and billions of pages. A small number of these sites became tremendously popular. Those sites and the companies behind them largely define the Web as people experience it today. Examples include: a bookstore (Amazon, started in 1994), a flea market (eBay, 1995), search (Google, 1998), and social networking (Facebook, 2004). The period through 2000, when many Web companies became worth hundreds of millions of dollars overnight, only to go bust practically the next day when they turned

out to be hype, even has a name. It is called the **dot com era**. New ideas are still striking it rich on the Web. Many of them come from students. For example, Mark Zuckerberg was a Harvard student when he started Facebook, and Sergey Brin and Larry Page were students at Stanford when they started Google. Perhaps you will come up with the next big thing.

In 1994, CERN and M.I.T. signed an agreement setting up the **W3C** (**World Wide Web Consortium**), an organization devoted to further developing the Web, standardizing protocols, and encouraging interoperability between sites. Berners-Lee became the director. Since then, several hundred universities and companies have joined the consortium. Although there are now more books about the Web than you can shake a stick at, the best place to get up-to-date information about the Web is (naturally) on the Web itself. The consortium's home page is at *www.w3.org*. Interested readers are referred there for links to pages covering all of the consortium's numerous documents and activities.

7.3.1 Architectural Overview

From the users' point of view, the Web comprises a vast, worldwide collection of content in the form of **Web pages**. Each page typically contains links to hundreds of other objects, which may be hosted on any server on the Internet, anywhere in the world. These objects may be other text and images, but nowadays also include a wide variety of objects, including advertisements and tracking scripts. A page may also **link** to other Web pages; users can follow a link by clicking on it, which then takes them to the page pointed to. This process can be repeated indefinitely. The idea of having one page point to another, now called **hypertext**, was invented by a visionary M.I.T. professor of electrical engineering, Vannevar Bush, in 1945 (Bush, 1945). This was long before the Internet was invented. In fact, it was before commercial computers existed although several universities had produced crude prototypes that filled large rooms and had millions of times less computing power than a smart watch but consumed more electrical power than a small factory.

Pages are generally viewed with a program called a **browser**. Brave, Chrome, Edge, Firefox, Opera, and Safari are examples of popular browsers. The browser fetches the page requested, interprets the content, and displays the page, properly formatted, on the screen. The content itself may be a mix of text, images, and formatting commands, in the manner of a traditional document, or other forms of content such as video or programs that produce a graphical interface for users.

Figure 7-19 shows an example of a Web page, which contains many objects. In this case, the page is for the U.S. Federal Communications Commission. This page shows text and graphical elements (which are mostly too small to read here). Many parts of the page include references and links to other pages. The **index page**, which the browser loads, typically contains instructions for the browser

concerning the locations of other objects to assemble, as well as how and where to render those objects on the page.

A piece of text, icon, graphic image, photograph, or other page element that can be associated with another page is called a **hyperlink**. To follow a link, a desktop or notebook computer user places the mouse cursor on the linked portion of the page area (which causes the cursor to change shape) and clicks. On a smartphone or tablet, the user taps the link. Following a link is simply a way of telling the browser to fetch another page. In the early days of the Web, links were highlighted with underlining and colored text so that they would stand out. Now, the creators of Web pages can use **style sheets** to control the appearance of many aspects of the page, including hyperlinks, so links can effectively appear however the designer of the Web site wishes. The appearance of a link can even be dynamic, for example, it might change its appearance when the mouse passes over it. It is up to the creators of the page to make the links visually distinct to provide a good user experience.

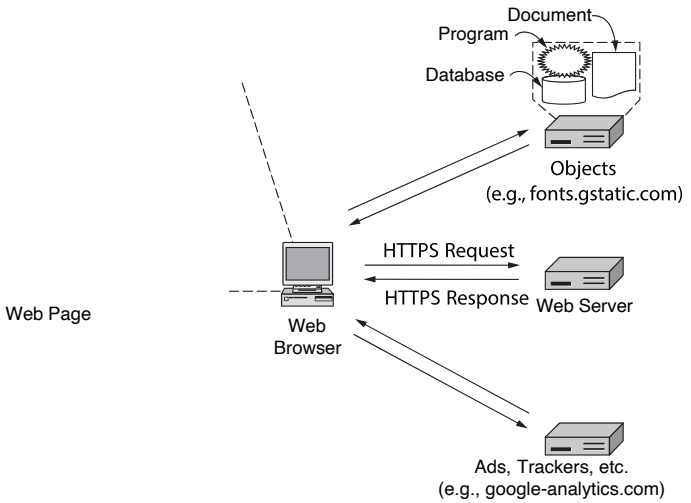

Figure 7-19. Fetching and rendering a Web page involves HTTP/HTTPS requests to many servers.

Readers of this page might find a story of interest and click on the area indicated, at which point the browser fetches the new page and displays it. Dozens of other pages are linked off the first page besides this example. Every other page can consist of content on the same machine(s) as the first page, or on machines halfway around the globe. The user cannot tell. The browser typically fetches whatever objects the user indicates to the browser through a series of clicks. Thus, moving between machines while viewing content is seamless.

The browser is displaying a Web page on the client machine. Each page is fetched by sending a request to one or more servers, which respond with the contents of the page. The request-response protocol for fetching pages is a simple text-based protocol that runs over TCP, just as was the case for SMTP. It is called **HTTP** (**HyperText Transfer Protocol**). The secure version of this protocol, which is now the predominant mode of retrieving content on the Web today, is called **HTTPS** (**Secure HyperText Transfer Protocol**). The content may simply be a document that is read off a disk, or the result of a database query and program execution. The page is a **static page** if it is a document that is the same every time it is displayed. In contrast, if it was generated on demand by a program or contains a program it is a **dynamic page**.

A dynamic page may present itself differently each time it is displayed. For example, the front page for an electronic store may be different for each visitor. If a bookstore customer has bought mystery novels in the past, upon visiting the store's main page, the customer is likely to see new thrillers prominently displayed, whereas a more culinary-minded customer might be greeted with new cookbooks. How the Web site keeps track of who likes what is a story to be told shortly. But briefly, the answer involves cookies (even for culinarily challenged visitors).

In the browser contacts a number of servers to load the Web page. The content on the index page might be loaded directly from files hosted at *fcc.gov*. Auxiliary content, such as an embedded video, might be hosted at a separate server, still at *fcc.gov*, but perhaps on infrastructure that is dedicated to hosting the content. The index page may also contain references to other objects that the user may not even see, such as tracking scripts, or advertisements that are hosted on third-party servers. The browser fetches all of these objects, scripts, and so forth and assembles them into a single page view for the user.

Display entails a range of processing that depends on the kind of content. Besides rendering text and graphics, it may involve playing a video or running a script that presents its own user interface as part of the page. In this case, the *fcc.gov* server supplies the main page, the *fonts.gstatic.com* server supplies additional objects (e.g., fonts), and the *google-analytics.com* server supplies nothing that the user can see but tracks visitors to the site. We will investigate trackers and Web privacy later in this chapter.

The Client Side

Let us now examine the Web browser side in Fig. 7-19 in more detail. In essence, a browser is a program that can display a Web page and capture a user's request to ''follow'' other content on the page. When an item is selected, the browser follows the hyperlink and retrieves the object that the user indicates (e.g., with a mouse click, or by tapping the link on the screen of a mobile device).

When the Web was first created, it was immediately apparent that having one page point to another Web page required mechanisms for naming and locating

pages. In particular, three questions had to be answered before a selected page could be displayed:

1. What is the page called?
2. Where is the page located?
3. How can the page be accessed?

If every page were somehow assigned a unique name, there would not be any ambiguity in identifying pages. Nevertheless, the problem would not be solved. Consider a parallel between people and pages. In the United States, almost every adult has a Social Security number, which is a unique identifier, as no two people are supposed to have the same one. Nevertheless, if you are armed only with a social security number, there is no way to find the owner's address, and certainly no way to tell whether you should write to the person in English, Spanish, or Chinese. The Web has basically the same problems.

The solution chosen identifies pages in a way that solves all three problems at once. Each page is assigned a **URL** (**Uniform Resource Locator**) that effectively serves as the page's worldwide name. URLs have three parts: the protocol (also known as the **scheme**), the DNS name of the machine on which the page is located, and the path uniquely indicating the specific page (a file to read or program to run on the machine). In the general case, the path has a hierarchical name that models a file directory structure. However, the interpretation of the path is up to the server; it may or may not reflect the actual directory structure.

As an example, the URL of the page shown in Fig. 7-19 is

https://fcc.gov/

This URL consists of three parts: the protocol (*https*), the DNS name of the host (*fcc.gov*), and the path name (/, which the Web server often treats as some default index object).

When a user selects a hyperlink, the browser carries out a series of steps in order to fetch the page pointed to. Let us trace the steps that occur when our example link is selected:

1. The browser determines the URL (by seeing what was selected).
2. The browser asks DNS for the IP address of the server *fcc.gov*.
3. DNS replies with 23.1.55.196.
4. The browser makes a TCP connection to that IP address; given that the protocol is HTTPS, the secure version of HTTP, the TCP connection would by default be on port 443 (the default port for HTTP, which is used far less often now, is port 80).
5. It sends an HTTPS request asking for the page //, which the Web server typically assumes is some index page (e.g., index.html, index.php, or similar, as configured by the Web server at *fcc.gov*).

6. The server sends the page as an HTTPS response, for example, by sending the file */index.html*, if that is determined to be the default index object.
7. If the page includes URLs that are needed for display, the browser fetches the other URLs using the same process. In this case, the URLs include multiple embedded images also fetched from that server, embedded objects from *gstatic.com*, and a script from *google-analytics.com* (as well as a number of other domains that are not shown).
8. The browser displays the page */index.html* as it appears in Fig. 7-19.
9. The TCP connections are released if there are no other requests to the same servers for a short period.

Many browsers display which step they are currently executing in a status line at the bottom of the screen. In this way, when the performance is poor, the user can see if it is due to DNS not responding, a server not responding, or simply page transmission over a slow or congested network.

A more detailed way to explore and understand the performance of the Web page is through a so-called **waterfall diagram**, as shown in Fig. 7-20.

The figure shows a list of all of the objects that the browser loads in the process of loading this page (in this case, 64, but many pages have hundreds of objects), as well as the timing dependencies associated with loading each request, and the operations associated with each page load (e.g., a DNS lookup, a TCP connection, the downloading of actual content, and so forth). These waterfall diagrams can tell us a lot about the behavior of a Web browser; for example, we can learn about the number of parallel connections that a browser makes to any given server, as well as whether connections are being reused. We can also learn about the relative time for DNS lookups versus actual object downloads, as well as other potential performance bottlenecks.

The URL design is open-ended in the sense that it is straightforward to have browsers use multiple protocols to retrieve different kinds of resources. In fact, URLs for various other protocols have been defined. Slightly simplified forms of the common ones are listed in Fig. 7-21.

Let us briefly go over the list. The *http* protocol is the Web's native language, the one spoken by Web servers. **HTTP** stands for **HyperText Transfer Protocol**. We will examine it in more detail later in this section, with a particular focus on HTTPS, the secure version of this protocol, which is now the predominant protocol used to serve objects on the Web today.

The *ftp* protocol is used to access files by FTP, the Internet's file transfer protocol. FTP predates the Web and has been in use for more than four decades. The Web makes it easy to obtain files placed on numerous FTP servers throughout the world by providing a simple, clickable interface instead of the older command-line

Figure 7-20. Waterfall diagram for *fcc.gov*.

interface. This improved access to information is one reason for the spectacular growth of the Web.

It is possible to access a local file as a Web page by using the *file* protocol, or more simply, by just naming it. This approach does not require having a server. Of course, it works only for local files, not remote ones.

The *mailto* protocol does not really have the flavor of fetching Web pages, but is still useful anyway. It allows users to send email from a Web browser. Most

Name	Used for	Example
http	Hypertext (HTML)	https://www.ee.uwa.edu/~rob/ (https://www.ee.uwa.edu/~rob/)
https	Hypertext with security	https://www.bank.com/accounts/ (https://www.bank.com/accounts/)
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README (ftp://ftp.cs.vu.nl/pub/minix/README)
file	Local file	file:///usr/nathan/prog.c
mailto	Sending email	mailto:JohnUser@acm.org
rtsp	Streaming media	rtsp://youtube.com/montypython.mpg
sip	Multimedia calls	sip:eve@adversary.com
about	Browser information	about:plugins

Figure 7-21. Some common URL schemes.

browsers will respond when a *mailto* link is followed by starting the user's mail agent to compose a message with the address field already filled in.

The *rtsp* and *sip* protocols are for establishing streaming media sessions and audio and video calls.

Finally, the *about* protocol is a convention that provides information about the browser. For example, following the *about:plugins* link will cause most browsers to show a page that lists the MIME types that they handle with browser extensions called plug-ins. Many browsers have very interesting information in the about: section; an interesting example in the Firefox browser is *about:telemetry*, which shows all of the performance and user activity information that the browser gathers about the user. *about:preferences* shows user preferences, and *about:config* shows many interesting aspects of the browser configuration, including whether the browser is performing DNS-over-HTTPS lookups (and to which trusted recursive resolvers), as described in the previous section on DNS.

The URLs themselves have been designed not only to allow users to navigate the Web, but to run older protocols such as FTP and email as well as newer protocols for audio and video, and to provide convenient access to local files and browser information. This approach makes all the specialized user interface programs for those other services unnecessary and integrates nearly all Internet access into a single program: the Web browser. If it were not for the fact that this idea was thought of by a British physicist working a multinational European research lab in Switzerland (CERN), it could easily pass for a plan dreamed up by some software company's advertising department.

The Server Side

So much for the client side. Now let us take a look at the server side. As we saw above, when the user types in a URL or clicks on a line of hypertext, the browser parses the URL and interprets the part between *https://* and the next slash as a DNS name to look up. Armed with the IP address of the server, the browser can

establish a TCP connection to port 443 on that server. Then it sends over a command containing the rest of the URL, which is the path to the page on that server. The server then returns the page for the browser to display.

To a first approximation, a simple Web server is similar to the server of Fig. 6-6. That server is given the name of a file to look up and return via the network. In both cases, the steps that the server performs in its main loop are:

1. Accept a TCP connection from a client (a browser).
2. Get the path to the page, which is the name of the file requested.
3. Get the file (from disk).
4. Send the contents of the file to the client.
5. Release the TCP connection.

Modern Web servers have more features, but in essence, this is what a Web server does for the simple case of content that is contained in a file. For dynamic content, the third step may be replaced by the execution of a program (determined from the path) that generates and returns the contents.

However, Web servers are implemented with a different design to serve hundreds or thousands of requests per second. One problem with the simple design is that accessing files is often the bottleneck. Disk reads are very slow compared to program execution, and the same files may be read repeatedly from disk using operating system calls. Another problem is that only one request is processed at a time. If the file is large, other requests will be blocked while it is transferred.

One obvious improvement (used by all Web servers) is to maintain a cache in memory of the n most recently read files or a certain number of gigabytes of content. Before going to disk to get a file, the server checks the cache. If the file is there, it can be served directly from memory, thus eliminating the disk access. Although effective caching requires a large amount of main memory and some extra processing time to check the cache and manage its contents, the savings in time are nearly always worth the overhead and expense.

To tackle the problem of serving more than a single request at a time, one strategy is to make the server **multithreaded**. In one design, the server consists of a front-end module that accepts all incoming requests and k processing modules, as shown in Fig. 7-22. The $k + 1$ threads all belong to the same process, so the processing modules all have access to the cache within the process' address space. When a request comes in, the front end accepts it and builds a short record describing it. It then hands the record to one of the processing modules.

The processing module first checks the cache to see if the requested object is present. If so, it updates the record to include a pointer to the file in the record. If it is not there, the processing module starts a disk operation to read it into the cache (possibly discarding some other cached file(s) to make room for it). When the file comes in from the disk, it is put in the cache and also sent back to the client.

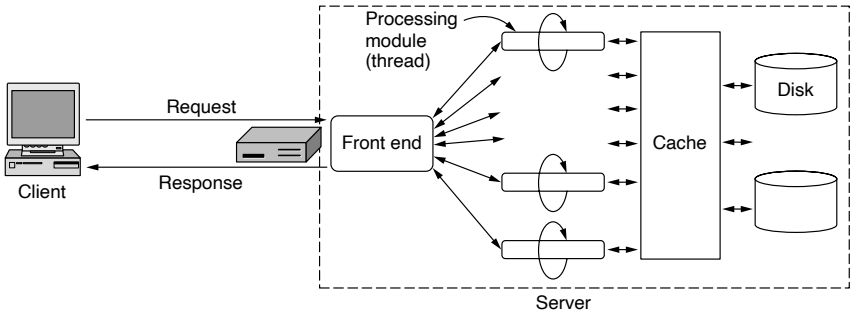

Figure 7-22. A multithreaded Web server with a front end and processing modules.

The advantage of this approach is that while one or more processing modules are blocked waiting for a disk or network operation to complete (and thus consuming no CPU time), other modules can be actively working on other requests. With k processing modules, the throughput can be as much as k times higher than with a single-threaded server. Of course, when the disk or network is the limiting factor, it is necessary to have multiple disks or a faster network to get any real improvement over the single-threaded model.

Essentially all modern Web architectures are now designed as shown above, with a split between the front end and a back end. The front-end Web server is often called a **reverse proxy**, because it retrieves content from other (typically back-end) servers and serves those objects to the client. The proxy is called a ''reverse'' proxy because it is acting on behalf of the servers, as opposed to acting on behalf of clients.

When loading a Web page, a client will often first be directed (using DNS) to a reverse proxy (i.e., front end server), which will begin returning static objects to the client's Web browser so that it can begin loading some of the page contents as quickly as possible. While those (typically static) objects are loading, the back end can perform complex operations (e.g., performing a Web search, doing a database lookup, or otherwise generating dynamic content), which it can serve back to the client via the reverse proxy as those results and content becomes available.

7.3.2 Static Web Objects

The basis of the Web is transferring Web pages from server to client. In the simplest form, Web objects are static. However, these days, almost any page that you view on the Web will have some dynamic content, but even on dynamic Web pages, a significant amount of the content (e.g., the logo, the style sheets, the header and footer) remains static. Static objects are just files sitting on some server that present themselves in the same way each time they are fetched and viewed. They

are generally amenable to caching, sometimes for a very long time, and are thus often placed on object caches that are close to the user. Just because they are static does not mean that the pages are inert at the browser, however. A video is a static object, for example.

As mentioned earlier, the lingua franca of the Web, in which most pages are written, is HTML. The home pages of university instructors are generally static objects; in some cases, companies may have dynamic Web pages, but the end result of the dynamic-generation process is a page in HTML. **HTML** (**HyperText Markup Language**) was introduced with the Web. It allows users to produce Web pages that include text, graphics, video, pointers to other Web pages, and more. HTML is a markup language, or language for describing how documents are to be formatted. The term ''markup'' comes from the old days when copyeditors actually marked up documents to tell the printer—in those days, a human being—which fonts to use, and so on. Markup languages thus contain explicit commands for formatting. For example, in HTML, ** means start boldface mode, and ** means leave boldface mode. Also, <h1> means to start a level 1 heading here. LaTeX and TeX are other examples of markup languages that are well known to most academic authors. In contrast, Microsoft Word is *not* a markup language because the formatting commands are *not* embedded in the text.

The key advantage of a markup language over one with no explicit markup is that it separates content from how it should be presented. Most modern Webpages use **style sheets** to define the typefaces, colors, sizes, padding, and many other attributes of text, lists, tables, headings, ads, and other page elements. Style sheets are written in a language called **CSS** (**Cascading Style Sheets**).

Writing a browser is then straightforward: the browser simply has to understand the markup commands and style sheet and apply them to the content. Embedding all the markup commands within each HTML file and standardizing them makes it possible for any Web browser to read and reformat any Web page. That is crucial because a page may have been produced in a 3840 × 2160 window with 24-bit color on a high-end computer but may have to be displayed in a 640 × 320 window on a mobile phone. Just scaling it down linearly is a bad idea because then the letters would be so small that no one could read them.

While it is certainly possible to write documents like this with any plain text editor, and many people do, it is also possible to use word processors or special HTML editors that do most of the work (but correspondingly give the user less direct control over the details of the final result). There are also many programs available for designing Web pages, such as Adobe Dreamweaver.

7.3.3 Dynamic Web Pages and Web Applications

The static page model we have used so far treats pages as (multimedia) documents that are conveniently linked together. It was a good model back in the early days of the Web, as vast amounts of information were put online. Nowadays,

much of the excitement around the Web is using it for applications and services. Examples include buying products on e-commerce sites, searching library catalogs, exploring maps, reading and sending email, and collaborating on documents.

These new uses are like conventional application software (e.g., mail readers and word processors). The twist is that these applications run inside the browser, with user data stored on servers in Internet data centers. They use Web protocols to access information via the Internet, and the browser to display a user interface. The advantage of this approach is that users do not need to install separate application programs, and user data can be accessed from different computers and backed up by the service operator. It is proving so successful that it is rivaling traditional application software. Of course, the fact that these applications are offered for free by large providers helps. This model is a prevalent form of **cloud computing**, where computing moves off individual desktop computers and into shared clusters of servers in the Internet.

To act as applications, Web pages can no longer be static. Dynamic content is needed. For example, a page of the library catalog should reflect which books are currently available and which books are checked out and are thus not available. Similarly, a useful stock market page would allow the user to interact with the page to see stock prices over different periods of time and compute profits and losses. As these examples suggest, dynamic content can be generated by programs running on the server or in the browser (or in both places).

The general situation is as shown in Fig. 7-23. For example, consider a map service that lets the user enter a street address and presents a corresponding map of the location. Given a request for a location, the Web server must use a program to create a page that shows the map for the location from a database of streets and other geographic information. This action is shown as steps 1 through 3. The request (step 1) causes a program to run on the server. The program consults a database to generate the appropriate page (step 2) and returns it to the browser (step 3).

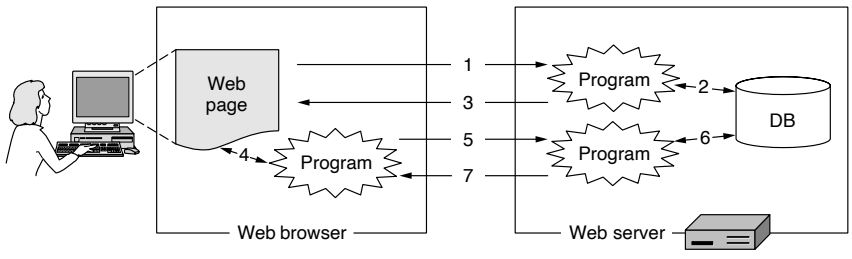

Figure 7-23. Dynamic pages.

There is more to dynamic content, however. The page that is returned may itself contain programs that run in the browser. In our map example, the program

would let the user find routes and explore nearby areas at different levels of detail. It would update the page, zooming in or out as directed by the user (step 4). To handle some interactions, the program may need more data from the server. In this case, the program will send a request to the server (step 5) that will retrieve more information from the database (step 6) and return a response (step 7). The program will then continue updating the page (step 4). The requests and responses happen in the background; the user may not even be aware of them because the page URL and title typically do not change. By including client-side programs, the page can present a more responsive interface than with server-side programs alone.

Server-Side Dynamic Web Page Generation

Let us look briefly at the case of server-side content generation. When the user clicks on a link in a form, for example in order to buy something, a request is sent to the server at the URL specified with the form along with the contents of the form as filled in by the user. These data must be given to a program or script to process. Thus, the URL identifies the program to run; the data are provided to the program as input. The page returned by this request will depend on what happens during the processing. It is not fixed like a static page. If the order succeeds, the page returned might give the expected shipping date. If it is unsuccessful, the returned page might say that widgets requested are out of stock or the credit card was not valid for some reason.

Exactly how the server runs a program instead of retrieving a file depends on the design of the Web server. It is not specified by the Web protocols themselves. This is because the interface can be proprietary and the browser does not need to know the details. As far as the browser is concerned, it is simply making a request and fetching a page.

Nonetheless, standard APIs have been developed for Web servers to invoke programs. The existence of these interfaces makes it easier for developers to extend different servers with Web applications. We will briefly look at two APIs to give you a sense of what they entail.

The first API is a method for handling dynamic page requests that has been available since the beginning of the Web. It is called the **CGI** (**Common Gateway Interface**) and is defined in RFC 3875. CGI provides an interface to allow Web servers to talk to back-end programs and scripts that can accept input (e.g., from forms) and generate HTML pages in response. These programs may be written in whatever language is convenient for the developer, usually a scripting language for ease of development. Pick Python, Ruby, Perl, or your favorite language.

By convention, programs invoked via CGI live in a directory called *cgi-bin*, which is visible in the URL. The server maps a request to this directory to a program name and executes that program as a separate process. It provides any data sent with the request as input to the program. The output of the program gives a Web page that is returned to the browser.

The second API is quite different. The approach here is to embed little scripts inside HTML pages and have them be executed by the server itself to generate the page. A popular language for writing these scripts is **PHP** (**PHP: Hypertext Preprocessor**). To use it, the server has to understand PHP, just as a browser has to understand CSS to interpret Web pages with style sheets. Usually, servers identify Web pages containing PHP from the file extension *php* rather than *html* or *htm*. PHP is simpler to use than CGI and is widely used.

Although PHP is easy to use, it is actually a powerful programming language for interfacing the Web and a server database. It has variables, strings, arrays, and most of the control structures found in C, but much more powerful I/O than just *printf*. PHP is open source code, freely available, and widely used. It was designed specifically to work well with Apache, which is also open source and is the world's most widely used Web server.

Client-Side Dynamic Web Page Generation

PHP and CGI scripts solve the problem of handling input and interactions with databases on the server. They can all accept incoming information from forms, look up information in one or more databases, and generate HTML pages with the results. What none of them can do is respond to mouse movements or interact with users directly. For this purpose, it is necessary to have scripts embedded in HTML pages that are executed on the client machine rather than the server machine. Starting with HTML 4.0, such scripts were permitted using the tag <script>. The current HTML standard is now generally referred to as **HTML5**. HTML5 includes many new syntactic features for incorporating multimedia and graphical content, including <video>, <audio>, and <canvas> tags. Notably, the canvas element facilitates dynamic rendering of two-dimensional shapes and bitmap images. Interestingly, the canvas element also has various privacy considerations, because the HTML canvas properties are often unique on different devices. The privacy concerns are significant, because the uniqueness of canvases on individual user devices allows Web site operators to track users, even if the users delete all tracking cookies and block tracking scripts.

The most popular scripting language for the client side is **JavaScript**, so we will now take a quick look at it. Many books have been written about it (e.g., Coding, 2019; and Atencio, 2020). Despite the similarity in names, JavaScript has almost nothing to do with the Java programming language. Like other scripting languages, it is a very high-level language. For example, in a single line of JavaScript it is possible to pop up a dialog box, wait for text input, and store the resulting string in a variable. High-level features like this make JavaScript ideal for designing interactive Web pages. On the other hand, the fact that it is mutating faster than a fruit fly trapped in an X-ray machine makes it difficult to write JavaScript programs that work on all platforms, but maybe some day it will stabilize.

It is important to understand that while PHP and JavaScript look similar in that they both embed code in HTML files, they are processed totally differently. With PHP, after a user has clicked on the *submit* button, the browser collects the information into a long string and sends it off to the server as a request for a PHP page. The server loads the PHP file and executes the PHP script that is embedded in to produce a new HTML page. That page is sent back to the browser for display. The browser cannot even be sure that it was produced by a program. This processing is shown as steps 1 to 4 in Fig. 7-24(a).

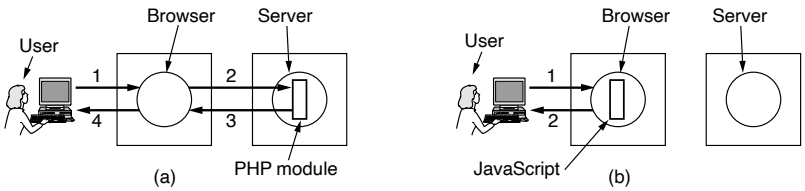

Figure 7-24. (a) Server-side scripting with PHP. (b) Client-side scripting with JavaScript.

With JavaScript, when the *submit* button is clicked the browser interprets a JavaScript function contained on the page. All the work is done locally, inside the browser. There is no contact with the server. This processing is shown as steps 1 and 2 in Fig. 7-24(b). As a consequence, the result is displayed virtually instantaneously, whereas with PHP there can be a delay of several seconds before the resulting HTML arrives at the client.

This difference does not mean that JavaScript is better than PHP. Their uses are completely different. PHP is used when interaction with a database on the server is needed. JavaScript (and other client-side languages) is used when the interaction is with the user at the client computer. It is certainly possible to combine them, as we will see shortly.

7.3.4 HTTP and HTTPS

Now that we have an understanding of Web content and applications, it is time to look at the protocol that is used to transport all this information between Web servers and clients. It is **HTTP** (**HyperText Transfer Protocol**), as specified in RFC 2616. Before we get into too many details, it is worth noting some distinctions between HTTP and its secure counterpart, **HTTPS** (**Secure HyperText Transfer Protocol**). Both protocols essentially retrieve objects in the same way, and the HTTP standard to retrieve Web objects is evolving essentially independently from its secure counterpart, which effectively uses the HTTP protocol over a secure transport protocol called **TLS** (**Transport Layer Security**). In this chapter, we will focus on the protocol details of HTTP and how it has evolved from early

versions, to the more modern versions of this protocol in what is now known as HTTP/3. Chapter 8 discusses TLS in more detail, which effectively is the transport protocol that transports HTTP, constituting what we think of as HTTPS. For the remainder of this section, we will talk about HTTP; you can think of HTTPS as simply HTTP that is transported over TLS.

Overview

HTTP is a simple request-response protocol; conventional versions of HTTP typically run over TCP, although the most modern version of HTTP, HTTP/3, now commonly runs over UDP as well. It specifies what messages clients may send to servers and what responses they get back in return. The request and response headers are given in ASCII, just like in SMTP. The contents are given in a MIME-like format, also like in SMTP. This simple model was partly responsible for the early success of the Web because it made development and deployment straightforward.

In this section, we will look at the more important properties of HTTP as it is used today. Before getting into the details we will note that the way it is used in the Internet is evolving. HTTP is an application layer protocol because it runs on top of TCP and is closely associated with the Web. That is why we are covering it in this chapter. In another sense, HTTP is becoming more like a transport protocol that provides a way for processes to communicate content across the boundaries of different networks. These processes do not have to be a Web browser and Web server. A media player could use HTTP to talk to a server and request album information. Antivirus software could use HTTP to download the latest updates. Developers could use HTTP to fetch project files. Consumer electronics products like digital photo frames often use an embedded HTTP server as an interface to the outside world. Machine-to-machine communication increasingly runs over HTTP. For example, an airline server might contact a car rental server and make a car reservation, all as part of a vacation package the airline was offering.

Methods

Although HTTP was designed for use in the Web, it was intentionally made more general than necessary with an eye to future object-oriented uses. For this reason, operations, called **methods**, other than just requesting a Web page are supported.

Each request consists of one or more lines of ASCII text, with the first word on the first line being the name of the method requested. The built-in methods are listed in Fig. 7-25. The names are case sensitive, so *GET* is allowed but not *get*.

The *GET* method requests the server to send the page. (When we say ''page'' we mean ''object'' in the most general case, but thinking of a page as the contents of a file is sufficient to understand the concepts.) The page is suitably encoded in

Method	Description
GET	Read a Web page
HEAD	Read a Web page's header
POST	Append to a Web page
PUT	Store a Web page
DELETE	Remove the Web page
TRACE	Echo the incoming request
CONNECT	Connect through a proxy
OPTIONS	Query options for a page

Figure 7-25. The built-in HTTP request methods.

MIME. The vast majority of requests to Web servers are *GET*s and the syntax is simple. The usual form of *GET* is

```
GET filename HTTP/1.1
```

where *filename* names the page to be fetched and 1.1 is the protocol version.

The *HEAD* method just asks for the message header, without the actual page. This method can be used to collect information for indexing purposes, or just to test a URL for validity.

The *POST* method is used when forms are submitted. Like *GET*, it bears a URL, but instead of simply retrieving a page it uploads data to the server (i.e., the contents of the form or parameters). The server then does something with the data that depends on the URL, conceptually appending the data to the object. The effect might be to purchase an item, for example, or to call a procedure. Finally, the method returns a page indicating the result.

The remaining methods are not used much for browsing the Web. The *PUT* method is the reverse of *GET*: instead of reading the page, it writes the page. This method makes it possible to build a collection of Web pages on a remote server. The body of the request contains the page. It may be encoded using MIME, in which case the lines following the *PUT* might include authentication headers, to prove that the caller indeed has permission to perform the requested operation.

DELETE does what you might expect: it removes the page, or at least it indicates that the Web server has agreed to remove the page. As with *PUT*, authentication and permission play a major role here.

The *TRACE* method is for debugging. It instructs the server to send back the request. This method is useful when requests are not being processed correctly and the client wants to know what request the server actually got.

The *CONNECT* method lets a user make a connection to a Web server through an intermediate device, such as a Web cache.

The *OPTIONS* method provides a way for the client to query the server for a page and obtain the methods and headers that can be used with that page.

Every request gets a response consisting of a status line, and possibly additional information (e.g., all or part of a Web page). The status line contains a three-digit status code telling whether the request was satisfied and, if not, why not. The first digit is used to divide the responses into five major groups, as shown in Fig. 7-26.

Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

Figure 7-26. The status code response groups.

The 1xx codes are rarely used in practice. The 2xx codes mean that the request was handled successfully and the content (if any) is being returned. The 3xx codes tell the client to look elsewhere, either using a different URL or in its own cache (discussed later). The 4xx codes mean the request failed due to a client error such an invalid request or a nonexistent page. Finally, the 5xx errors mean the server itself has an internal problem, either due to an error in its code or to a temporary overload.

Message Headers

The request line (e.g., the line with the *GET* method) may be followed by additional lines with more information. They are called **request headers**. This information can be compared to the parameters of a procedure call. Responses may also have **response headers**. Some headers can be used in either direction. A selection of the more important ones is given in Fig. 7-27. This list is not short, so as you might imagine there are often several headers on each request and response.

The *User-Agent* header allows the client to inform the server about its browser implementation (e.g., *Mozilla/5.0* and *Chrome/74.0.3729.169*). This information is useful to let servers tailor their responses to the browser, since different browsers can have widely varying capabilities and behaviors.

The four *Accept* headers tell the server what the client is willing to accept in the event that it has a limited repertoire of what is acceptable to it. The first header specifies the MIME types that are welcome (e.g., *text/html*). The second gives the character set (e.g., *ISO-8859-5* or *Unicode-1-1*). The third deals with compression methods (e.g., *gzip*). The fourth indicates a natural language (e.g., Spanish). If the server has a choice of pages, it can use this information to supply the one the client is looking for. If it is unable to satisfy the request, an error code is returned and the request fails.

Header	Type	Contents
User-Agent	Request	Information about the browser and its platform
Accept	Request	The type of pages the client can handle
Accept-Charset	Request	The character sets that are acceptable to the client
Accept-Encoding	Request	The page encodings the client can handle
Accept-Language	Request	The natural languages the client can handle
If-Modified-Since	Request	Time and date to check freshness
If-None-Match	Request	Previously sent tags to check freshness
Host	Request	The server's DNS name
Authorization	Request	A list of the client's credentials
Referrer	Request	The previous URL from which the request came
Cookie	Request	Previously set cookie sent back to the server
Set-Cookie	Response	Cookie for the client to store
Server	Response	Information about the server
Content-Encoding	Response	How the content is encoded (e.g., *gzip*)
Content-Language	Response	The natural language used in the page
Content-Length	Response	The page's length in bytes
Content-Type	Response	The page's MIME type
Content-Range	Response	Identifies a portion of the page's content
Last-Modified	Response	Time and date the page was last changed
Expires	Response	Time and date when the page stops being valid
Location	Response	Tells the client where to send its request
Accept-Ranges	Response	Indicates the server will accept byte range requests
Date	Both	Date and time the message was sent
Range	Both	Identifies a portion of a page
Cache-Control	Both	Directives for how to treat caches
ETag	Both	Tag for the contents of the page
Upgrade	Both	The protocol the sender wants to switch to

Figure 7-27. Some HTTP message headers.

The *If-Modified-Since* and *If-None-Match* headers are used with caching. They let the client ask for a page to be sent only if the cached copy is no longer valid. We will describe caching shortly.

The *Host* header names the server. It is taken from the URL. This header is mandatory. It is used because some IP addresses may serve multiple DNS names and the server needs some way to tell which host to hand the request to.

The *Authorization* header is needed for pages that are protected. In this case, the client may have to prove it has a right to see the page requested. This header is used for that case.

The client uses the (misspelled) *Referer* [sic] header to give the URL that referred to the URL that is now requested. Most often this is the URL of the previous page. This header is particularly useful for tracking Web browsing, as it tells servers how a client arrived at the page.

Cookies are small files that servers place on client computers to remember information for later. A typical example is an e-commerce Web site that uses a client-side cookie to keep track of what the client has ordered so far. Every time the client adds an item to her shopping cart, the cookie is updated to reflect the new item ordered. Although cookies are dealt with in RFC 2109 rather than RFC 2616, they also have headers. The *Set-Cookie* header is how servers send cookies to clients. The client is expected to save the cookie and return it on subsequent requests to the server by using the *Cookie* header. (Note that there is a more recent specification for cookies with newer headers, RFC 2965, but this has largely been rejected by industry and is not widely implemented.)

Many other headers are used in responses. The *Server* header allows the server to identify its software build if it wishes. The next five headers, all starting with *Content-*, allow the server to describe properties of the page it is sending.

The *Last-Modified* header tells when the page was last modified, and the *Expires* header tells for how long the page will remain valid. Both of these headers play an important role in page caching.

The *Location* header is used by the server to inform the client that it should try a different URL. This can be used if the page has moved or to allow multiple URLs to refer to the same page (possibly on different servers). It is also used for companies that have a main Web page in the *com* domain but redirect clients to a national or regional page based on their IP addresses or preferred language.

If a page is large, a small client may not want it all at once. Some servers will accept requests for byte ranges, so the page can be fetched in multiple small units. The *Accept-Ranges* header announces the server's willingness to handle this.

Now we come to headers that can be used either way. The *Date* header can be used in both directions and contains the time and date the message was sent, while the *Range* header tells the byte range of the page that is provided by the response.

The *ETag* header gives a short tag that serves as a name for the content of the page. It is used for caching. The *Cache-Control* header gives other explicit instructions about how to cache (or, more usually, how not to cache) pages.

Finally, the *Upgrade* header is used for switching to a new communication protocol, such as a future HTTP protocol or a secure transport. It allows the client to announce what it can support and the server to assert what it is using.

Caching

People often return to Web pages that they have viewed before, and related Web pages often have the same embedded resources. Some examples are the images that are used for navigation across the site, as well as common style sheets

and scripts. It would be very wasteful to fetch all of these resources for these pages each time they are displayed because the browser already has a copy.

Squirreling away pages that are fetched for subsequent use is called **caching**. The advantage is that when a cached page can be reused, it is not necessary to repeat the transfer. HTTP has built-in support to help clients identify when they can safely reuse pages. This support improves performance by reducing both network traffic and latency. The trade-off is that the browser must now store pages, but this is nearly always a worthwhile trade-off because local storage is inexpensive. The pages are usually kept on disk so that they can be used when the browser is run at a later date.

The difficult issue with HTTP caching is how to determine that a previously cached copy of a page is the same as the page would be if it was fetched again. This determination cannot be made solely from the URL. For example, the URL may give a page that displays the latest news item. The contents of this page will be updated frequently even though the URL stays the same. Alternatively, the contents of the page may be a list of the gods from Greek and Roman mythology. This page should change somewhat less rapidly.

HTTP uses two strategies to tackle this problem. They are shown in Fig. 7-28 as forms of processing between the request (step 1) and the response (step 5). The first strategy is page validation (step 2). The cache is consulted, and if it has a copy of a page for the requested URL that is known to be fresh (i.e., still valid), there is no need to fetch it anew from the server. Instead, the cached page can be returned directly. The *Expires* header returned when the cached page was originally fetched and the current date and time can be used to make this determination.

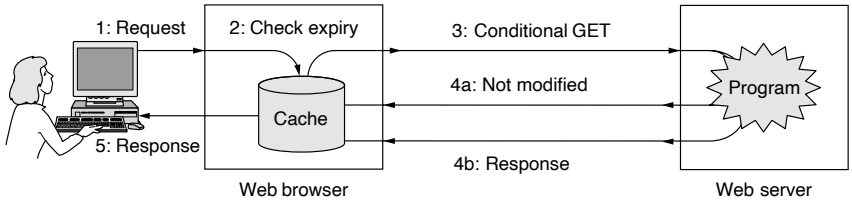

Figure 7-28. HTTP caching.

However, not all pages come with a convenient *Expires* header that tells when the page must be fetched again. After all, making predictions is hard—especially about the future. In this case, the browser may use heuristics. For example, if the page has not been modified in the past year (as told by the *Last-Modified* header) it is a fairly safe bet that it will not change in the next hour. There is no guarantee, however, and this may be a bad bet. For example, the stock market might have closed for the day so that the page will not change for hours, but it will change rapidly once the next trading session starts. Thus, the cacheability of a page may

vary wildly over time. For this reason, heuristics should be used with care, though they often work well in practice.

Finding pages that have not expired is the most beneficial use of caching because it means that the server does not need to be contacted at all. Unfortunately, it does not always work. Servers must use the *Expires* header conservatively, since they may be unsure when a page will be updated. Thus, the cached copies may still be fresh, but the client does not know.

The second strategy is used in this case. It is to ask the server if the cached copy is still valid. This request is a **conditional GET**, and it is shown in Fig. 7-28 as step 3. If the server knows that the cached copy is still valid, it can send a short reply to say so (step 4a). Otherwise, it must send the full response (step 4b).

More header fields are used to let the server check whether a cached copy is still valid. The client has the time a cached page was most recently updated from the *Last-Modified* header. It can send this time to the server using the *If-Modified-Since* header to ask for the page if and only if it has been changed in the meantime. There is much more to say about caching because it has such a big effect on performance, but this is not the place to say it. Not surprisingly, there are many tutorials on the Web that you can find easily by searching for ''Web caching.''

HTTP/1 and HTTP/1.1

The usual way for a browser to contact a server is to establish a TCP connection to port 443 for HTTPS (or port 80 for HTTP) on the server's machine, although this procedure is not formally required. The value of using TCP is that neither browsers nor servers have to worry about how to handle long messages, reliability, or congestion control. All of these matters are handled by the TCP implementation.

Early in the Web, with HTTP/1.0, after the connection was established a single request was sent over and a single response was sent back. Then the TCP connection was released. In a world in which the typical Web page consisted entirely of HTML text, this method was adequate. Quickly, the average Web page grew to contain large numbers of embedded links for content such as icons and other eye candy. Establishing a separate TCP connection to transport each single icon became a very expensive way to operate.

This observation led to HTTP/1.1, which supports **persistent connections**. With them, it is possible to establish a TCP connection, send a request and get a response, and then send additional requests and get additional responses. This strategy is also called **connection reuse**. By amortizing the TCP setup, startup, and release costs over multiple requests, the relative overhead due to TCP is reduced per request. It is also possible to pipeline requests, that is, send request 2 before the response to request 1 has arrived.

The performance difference between these three cases is shown in Fig. 7-29. Part (a) shows three requests, one after the other and each in a separate connection.

Let us suppose that this represents a Web page with two embedded images on the same server. The URLs of the images are determined as the main page is fetched, so they are fetched after the main page. Nowadays, a typical page has around 40 other objects that must be fetched to present it, but that would make our figure far too big so we will use only two embedded objects.

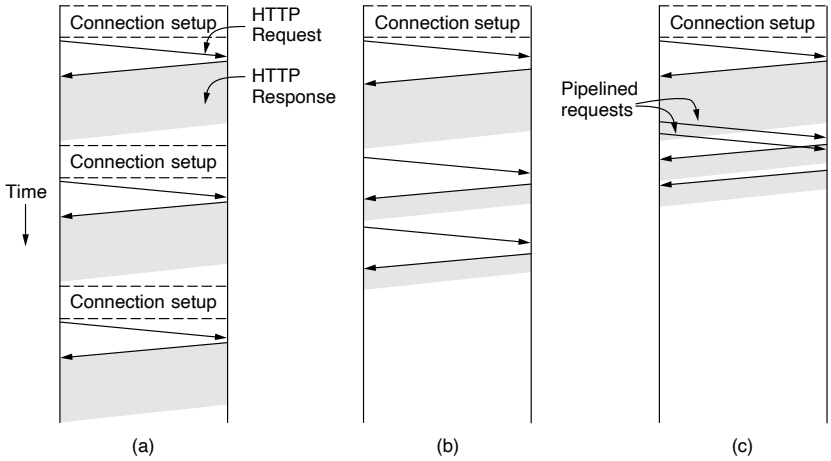

Figure 7-29. HTTP with (a) multiple connections and sequential requests. (b) A persistent connection and sequential requests. (c) A persistent connection and pipelined requests.

In Fig. 7-29(b), the page is fetched with a persistent connection. That is, the TCP connection is opened at the beginning, then the same three requests are sent, one after the other as before, and only then is the connection closed. Observe that the fetch completes more quickly. There are two reasons for the speedup. First, time is not wasted setting up additional connections. Each TCP connection requires at least one round-trip time to establish. Second, the transfer of the same images proceeds more quickly. Why is this? It is because of TCP congestion control. At the start of a connection, TCP uses the slow-start procedure to increase the throughput until it learns the behavior of the network path. The consequence of this warmup period is that multiple short TCP connections take disproportionately longer to transfer information than one longer TCP connection.

Finally, in Fig. 7-29(c), there is one persistent connection and the requests are pipelined. Specifically, the second and third requests are sent in rapid succession as soon as enough of the main page has been retrieved to identify that the images must be fetched. The responses for these requests follow eventually. This method cuts down the time that the server is idle, so it further improves performance.

Persistent connections do not come for free, however. A new issue that they raise is when to close the connection. A connection to a server should stay open while the page loads. What then? There is a good chance that the user will click on a link that requests another page from the server. If the connection remains open, the next request can be sent immediately. However, there is no guarantee that the client will make another request of the server any time soon. In practice, clients and servers usually keep persistent connections open until they have been idle for a short time (e.g., 60 seconds) or they have a large number of open connections and need to close some.

The observant reader may have noticed that there is one combination that we have left out so far. It is also possible to send one request per TCP connection, but run multiple TCP connections in parallel. This **parallel connection** method was widely used by browsers before persistent connections. It has the same disadvantage as sequential connections—extra overhead—but much better performance. This is because setting up and ramping up the connections in parallel hides some of the latency. In our example, connections for both of the embedded images could be set up at the same time. However, running many TCP connections to the same server is discouraged. The reason is that TCP performs congestion control for each connection independently. As a consequence, the connections compete against each other, causing added packet loss, and in aggregate are more aggressive users of the network than an individual connection. Persistent connections are superior and used in preference to parallel connections because they avoid overhead and do not suffer from congestion problems.

HTTP/2

HTTP/1.0 was around from the start of the Web and HTTP/1.1 was written in 2007. By 2012 it was getting a bit long in tooth, so IETF set up a working group to create what later became HTTP/2. The starting point was a protocol Google had devised earlier, called SPDY. The final product was published as RFC 7540 in May 2015.

The working group had several goals it tried to achieve, including:

1. Allow clients and servers to choose which HTTP version to use.
2. Maintain compatibility with HTTP/1.1 as much as possible.
3. Improve performance with multiplexing, pipelining, compression, etc.
4. Support existing practices used in browsers, servers, proxies, delivery networks, and more.

A key idea was to maintain backward compatibility. Existing applications had to work with HTTP/2, but new ones could take advantage of the new features to improve performance. For this reason, the headers, URLs, and general semantics

did not change much. What changed was the way everything is encoded and the way the clients and servers interact. In HTTP/1.1, a client opens a TCP connection to a server, sends over a request as text, waits for a response, and in many cases then closes the connection. This is repeated as often as needed to fetch an entire Web page. In HTTP/2 A TCP connection is set up and many requests can be sent over, in binary, possibly prioritized, and the server can respond to them in any order it wants to. Only after all requests have been answered is the TCP connection torn down.

Through a mechanism called **server push**, HTTP/2 allows the server to push out files that it knows will be needed but which the client may not know initially. For example, if a client requests a Web page and the server sees that it uses a style sheet and a JavaScript file, the server can send over the style sheet and the JavaScript before they are even requested. This eliminates some delays. An example of getting the same information (a Web page, its style sheet, and two images) in HTTP/1.1 and HTTP/2 is shown in Fig. 7-30.

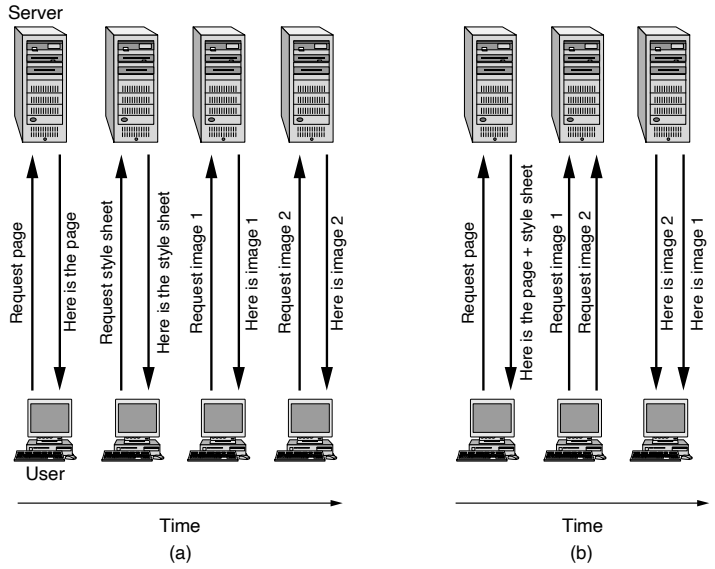

Figure 7-30. (a) Getting a Web page in HTTP/1.1. (b) Getting the same page in HTTP/2.

Note that Fig. 7-30(a) is the best case for HTTP/1.1, where multiple requests can be sent consecutively over the same TCP connection, but the rules are that they must be processed in order and the results sent back in order. In HTTP/2 [Fig. 7-30(b)], the responses can come back in any order. If it turns out, for example, that image 1 is very large, the server could back image 2 first so the browser

can start displaying the page with image 2 even before image 1 is available. That is not allowed in HTTP/1.1. Also note that in Fig. 7-30(b) the server sent the style sheet without the browser asking for it.

In addition to the pipelining and multiplexing of requests over the same TCP connection, HTTP/2 compresses the headers and sends them in binary to reduce bandwidth usage and latency. An HTTP/2 session consists of a series of frames, each with a separate identifier. Responses may come back in a different order than the requests, as in Fig. 7-30(b), but since each response carries the identifier of the request, the browser can determine which request each response corresponds to.

Encryption was a sore point during the development of HTTP/2. Some people wanted it badly, and others opposed it equally badly. The opposition was mostly related to Internet-of-Things applications, in which the ''thing'' does not have a lot of computing power. In the end, encryption was not required by the standard, but all browsers require encryption, so de facto it is there anyway, at least for Web browsing.

HTTP/3

HTTP/3 or simply **H3** is the third major revision of HTTP, designed as a successor to HTTP/2. The major distinction for HTTP/3 is the transport protocol that it uses to support the HTTP messages: rather than relying on TCP, it relies on an augmented version of UDP called **QUIC**, which relies on user-space congestion control running on top of UDP. HTTP/3 started out simply as HTTP-over-QUIC and has become the latest proposed major revision to the protocol. Many open-source libraries that support client and server logic for QUIC and HTTP/3 are available, in languages that include C, C++, Python, Rust, and Go. Popular Web servers including nginx also now support HTTP/3 through patches.

The QUIC transport protocol supports stream multiplexing and per-stream flow control, similar to that offered in HTTP/2. Stream-level reliability and connection-wide congestion control can dramatically improve the performance of HTTP, since congestion information can be shared across sessions, and reliability can be amortized across multiple connections fetching objects in parallel. Once a connection exists to a server endpoint, HTTP/3 allows the client to reuse that same connection with multiple different URLs.

HTTP/3, running HTTP over QUIC, promises many possible performance enhancements over HTTP/2, primarily because of the benefits that QUIC offers for HTTP vs. TCP. In some ways, QUIC could be viewed as the next generation of TCP. It offers connection setup with no additional round trips between client and server; in the case when a previous connection has been established between client and server, a zero-round-trip connection re-establishment is possible, provided that a secret from the previous connection was established and cached. QUIC guarantees reliable, in-order delivery of bytes within a single stream, but it does not

provide any guarantees with respect to bytes on other QUIC streams. QUIC does permit out-of-order delivery within a stream, but HTTP/3 does not make use of this feature. HTTP/3 over QUIC will be performed exclusively using HTTPS; requests to (the increasingly deprecated) HTTP URLs will not be upgraded to use HTTP/3. For more details on HTTP/3, see *https://http3.net*.

7.3.5 Web Privacy

One of the most significant concerns in recent years has been the privacy concerns associated with Web browsing. Web sites, Web applications, and other third parties often use mechanisms in HTTP to track user behavior, both within the context of a single Web site or application, or across the Internet. Additionally, attackers may exploit various information side channels in the browser or device to track users. This section describes some of the mechanisms that are used to track users and fingerprint individual users and devices.

Cookies

One conventional way to implement tracking is by placing a **cookie** (effectively a small amount of data) on client devices, which the clients may then send back upon subsequent visits to various Web sites. When a user requests a Web object (e.g., a Web page), a Web server may place a piece of persistent state, called a cookie, on the user's device, using the ''set-cookie'' directive in HTTP. The data passed to the client's device using this directive is subsequently stored locally on the device. When the device visits that Web domain in the future, the HTTP request passes the cookie, in addition to the request itself.

''First-party'' HTTP cookies (i.e., those set by the domain of the Web site that the user intends to visit, such as a shopping or news Web site) are useful for improving user experience on many Web sites. For example, cookies are often used to preserve state across a Web ''session.'' They allow a Web site to track useful information about a user's ongoing behavior on a Web site, such as whether they recently logged into the Web site, or what items they have placed in a shopping cart.

Cookies set by one domain are generally only visible to the same domain that set the cookie in the first place. For example, one advertising network may set a cookie on a user device, but no other third party can see the cookie that was set. This Web security policy, called the **same-origin policy**, prevents one party from reading a cookie that was set by another party and in some sense can limit how information about an individual user is shared.

Although first-party cookies are often used to improve the user experience, third parties, such as advertisers and tracking companies can also set cookies on client devices, which can allow those third parties to track the sites that users visit

as they navigate different Web sites across the entire Internet. This tracking takes place as follows:

1. When a user visits a Web site, in addition to the content that the user requests directly, the device may load content from third-party sites, including from the domains of advertising networks. Loading an advertisement or script from a third party allows that party to set a unique cookie on the user's device.
2. That user may subsequently visit different sites on the Internet that load Web objects from the same third party that set tracking information on a different site.

A common example of this practice might be two different Web sites that use the same advertising network to serve ads. In this case, the advertising network would see: (1) the user's device return the cookie that it set on a different Web site; (2) the HTTP *referer* request header that accompanies the request to load the object from the advertiser, indicating the original site that the user's device was visiting. This practice is commonly referred to as cross-site tracking.

Super cookies, and other locally stored tracking identifiers, that a user cannot control as they would regular cookies, can allow an intermediary to track a user across Web sites over time. Unique identifiers can include things such as third-party tracking identifiers encoded in HTTP (specifically **HSTS** (**HTTP Strict Transport Security**) headers that are not cleared when a user clears their cookies and tags that an intermediate third party such as a mobile ISP can insert into unencrypted Web traffic that traverses a network segment. This enables third parties, such as advertisers, to build up a profile of a user's browsing across a set of Web sites, similar to the Web tracking cookies used by ad networks and application providers.

Third-Party Trackers

Web cookies that originate from a third-party domain that are used across many sites can allow an advertising network or other third parties to track a user's browsing habits on any site where that tracking software is deployed (i.e., any site that carries their advertisements, sharing buttons, or other embedded code). Advertising networks and other third parties typically track a user's browsing patterns across the range of Web sites that the user browses, often using browser-based tracking software. In some cases, a third party may develop its own tracking software (e.g., Web analytics software). In other cases, they may use a different third-party service to collect and aggregate this behavior across sites.

Web sites may permit advertising networks and other third-party trackers to operate on their site, enabling them to collect analytics data, advertise on other Web sites (called re-targeting), or monetize the Web site's available advertising space via placement of carefully targeted ads. The advertisers collect data about

users by using various tracking mechanisms, such as HTTP cookies, HTML5 objects, JavaScript, device fingerprinting, browser fingerprinting, and other common Web technologies. When a user visits multiple Web sites that leverage the same advertising network, that advertising network recognizes the user's device, enabling them to track user Web behavior over time.

Using such tracking software, a third party or advertising network can discover a user's interactions, social network and contacts, likes, interests, purchases, and so on. This information can enable precise tracking of whether an advertisement resulted in a purchase, mapping of relationships between people, creation of detailed user tracking profiles, conduct of highly targeted advertising, and significantly more due to the breadth and scope of tracking.

Even in cases where someone is not a registered user of a particular service (e.g., social media site, search engine), has ceased using that service, or has logged out of that service, they often are still being uniquely tracked using third-party (and first-party) trackers. Third-party trackers are increasingly becoming concentrated with a few large providers.

In addition to third-party tracking with cookies, the same advertisers and third-party trackers can track user browsing behavior with techniques such as canvas fingerprinting (a type of browser fingerprinting), session replay (whereby a third party can see a playback of every user interaction with a particular Webpage), and even exploitation of a browser or password manager's ''auto-fill'' feature to send back data from Web forms, often before a user even fills out the form. These more sophisticated technologies can provide detailed information about user behavior and data, including fine-grained details such as the user's scrolls and mouse-clicks and even in some instances the user's username and password for a given Web site (which can be either intentional on the part of the user or unintentional on the part of the Web site).

A recent study suggests that specific instances of third-party tracking software are pervasive. The same study also discovered that news sites have the largest number of tracking parties on any given first-party site; other popular categories for tracking include arts, sports, and shopping Web sites. Cross-device tracking refers to the practice of linking activities of a single user across multiple devices (e.g., smartphones, tablets, desktop machines, other ''smart devices''); the practice aims to track a user's behavior, even as they use different devices.

Certain aspects of cross-device tracking may improve user experience. For example, as with cookies on a single device or browser, cross-device tracking can allow a user to maintain a seamless experience when moving from one device to the next (e.g., continuing to read a book or watch a movie from the place where the user left off). Cross-device tracking can also be useful for preventing fraud; for example, a service provider may notice that a user has logged in from an unfamiliar device in a completely new location. When a user attempts a login from an unrecognized device, a service provider can take additional steps to authenticate the user (e.g., two-factor authentication).

Cross-device tracking is most common by first-party services, such as email service providers, content providers (e.g., streaming video services), and commerce sites, but third parties are also becoming increasingly adept at tracking users across devices.

1. Cross-device tracking may be deterministic, based on a persistent identifier such as a login that is tied to a specific user.
2. Cross-device tracking may also be probabilistic; the IP address is one example of a probabilistic identifier that can be used to implement cross-device tracking. For example, technologies such as network address translation can cause multiple devices on a network to have the same public IP address. Suppose that a user visits a Web site from a mobile device (e.g., a smartphone) and uses that device at both home and work. A third party can set IP address information in the device's cookies. That user may then appear from two public IP addresses, one at work, and one at home, and those two IP addresses may be linked by the same third party cookie; if the user then visits that third party from different devices that share either of those two IP addresses, then those additional devices can be linked to the same user with high confidence.

Cross-device tracking often uses a combination of deterministic and probabilistic techniques; many of these techniques do not require the user to be logged into any site to enable this type of tracking. For example, some parties offer ''analytics'' services that, when embedded across many first-party Web sites, allow the third-party to track a user across Web sites and devices. Third parties often work together to track users across devices and services using a practice called **cookie syncing**, described in more detail later in this section.

Cross-device tracking enables more sophisticated inference of higher-level user activities, since data from different devices can be combined to build a more comprehensive picture of an individual user's activity. For example, data about a user's location (as collected from a mobile device) can be combined with a user's search history, social network activity (such as ''likes'') to determine for example whether a user has physically visited a store following an online search or online advertising exposure.

Device and Browser Fingerprinting

Even when users disable common tracking mechanisms such as third-party cookies, Web sites and third parties can still track users based on environmental, contextual, and device information that the device returns to the server. Based on a collection of this information, a third party may be able to uniquely identify, or ''fingerprint,'' a user across different sites and over time.

One well-known fingerprinting method is a technique called **canvas fingerprinting**, whereby the HTML canvas is used to identify a device. The HTML canvas allows a Web application to draw graphics in real time. Differences in font rendering, smoothing, dimensions, and some other features may cause each device to draw an image differently, and the resulting pixels can serve as a device fingerprint. The technique was first discovered in 2012, but not brought to public attention until 2014. Although there was a backlash at that time, many trackers continue to use canvas fingerprinting and related techniques such as canvas font fingerprinting, which identifies a device based on the browser's font list; a recent study found that these techniques are still present on thousands of sites. Web sites can also use browser APIs to retrieve other information for tracking devices, including information such as the battery status, which can be used to track a user based on battery charge level and discharge time. Other reports describe how knowing the battery status of a device can be used to track a device and therefore associate a device with a user (Olejnik et al., 2015)

Cookie Syncing

When different third-party trackers share information with each other, these parties can track an individual user even as they visit Web sites that have different tracking mechanisms installed. **Cookie syncing** is difficult to detect and also facilitates merging of datasets about individual users between disparate third parties, creating significant privacy concerns. A recent study suggests that the practice of cookie syncing is widespread among third-party trackers.

7.4 STREAMING AUDIO AND VIDEO

Email and Web applications are not the only major uses of networks. For many people, audio and video are the holy grail of networking. When the word ''multimedia'' is mentioned, both the propellerheads and the suits begin salivating as if on cue. The former see immense technical challenges in providing good quality voice over IP and 8K video-on-demand to every computer. The latter see equally immense profits in it.

While the idea of sending audio and video over the Internet has been around since the 1970s at least, it is only since roughly 2000 that **real-time audio** and **real-time video** traffic has grown with a vengeance. Real-time traffic is different from Web traffic in that it must be played out at some predetermined rate to be useful. After all, watching a video in slow motion with fits and starts is not most people's idea of fun. In contrast, the Web can have short interruptions, and page loads can take more or less time, within limits, without it being a major problem.

Two things happened to enable this growth. First, computers have became much more powerful and are equipped with microphones and cameras so that they can input, process, and output audio and video data with ease. Second, a flood of

Internet bandwidth has come to be available. Long-haul links in the core of the Internet run at many gigabits/sec, and broadband and 802.11ac wireless reaches users at the edge of the Internet. These developments allow ISPs to carry tremendous levels of traffic across their backbones and mean that ordinary users can connect to the Internet 100–1000 times faster than with a 56-kbps telephone modem.

The flood of bandwidth caused audio and video traffic to grow, but for different reasons. Telephone calls take up relatively little bandwidth (in principle 64 kbps but less when compressed) yet telephone service has traditionally been expensive. Companies saw an opportunity to carry voice traffic over the Internet using existing bandwidth to cut down on their telephone bills. Startups such as Skype saw a way to let customers make free telephone calls using their Internet connections. Upstart telephone companies saw a cheap way to carry traditional voice calls using IP networking equipment. The result was an explosion of voice data carried over the Internet and called Internet telephony and discussed in Sec. 7.4.4.

Unlike audio, video takes up a large amount of bandwidth. Reasonable quality Internet video is encoded with compression resulting in a stream of around 8 Mbps for 4K (which is 7 GB for a 2-hour movie) Before broadband Internet access, sending movies over the network was prohibitive. Not so any more. With the spread of broadband, it became possible for the first time for users to watch decent, streamed video at home. People love to do it. Around a quarter of the Internet users on any given day are estimated to visit YouTube, the popular video sharing site. The movie rental business has shifted to online downloads. And the sheer size of videos has changed the overall makeup of Internet traffic. The majority of Internet traffic is already video, and it is estimated that 90% of Internet traffic will be video within a few years.

Given that there is enough bandwidth to carry audio and video, the key issue for designing streaming and conferencing applications is network delay. Audio and video need real-time presentation, meaning that they must be played out at a predetermined rate to be useful. Long delays mean that calls that should be interactive no longer are. This problem is clear if you have ever talked on a satellite phone, where the delay of up to half a second is quite distracting. For playing music and movies over the network, the absolute delay does not matter, because it only affects when the media starts to play. But the variation in delay, called **jitter**, still matters. It must be masked by the player or the audio will sound unintelligible and the video will look jerky.

As an aside, the term **multimedia** is often used in the context of the Internet to mean video and audio. Literally, multimedia is just two or more media. That definition makes this book a multimedia presentation, as it contains text and graphics (the figures). However, that is probably not what you had in mind, so we use the term ''multimedia'' to imply two or more **continuous media**, that is, media that have to be played during some well-defined time interval. The two media are normally video with audio, that is, moving pictures with sound. Audio and smell may take a while. Many people also refer to pure audio, such as Internet telephony or

Internet radio, as multimedia as well, which it is clearly not. Actually, a better term for all these cases is **streaming media**. Nonetheless, we will follow the herd and consider real-time audio to be multimedia as well.

7.4.1 Digital Audio

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

The frequency range of the human ear runs from 20 Hz to 20,000 Hz. Some animals, notably dogs, can hear higher frequencies. The ear hears loudness logarithmically, so the ratio of two sounds with power A and B is conventionally expressed in **dB** (**decibels**) as the quantity $10 \log_{10}(A/B)$. If we define the lower limit of audibility (a sound pressure of about 20 μPascals) for a 1-kHz sine wave as 0 dB, an ordinary conversation is about 50 dB and the pain threshold is about 120 dB. The dynamic range is a factor of more than 1 million.

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

Digital audio is a digital representation of an audio wave that can be used to recreate it. Audio waves can be converted to digital form by an **ADC** (**Analog-to-Digital Converter**). An ADC takes an electrical voltage as input and generates a binary number as output. In Fig. 7-31(a) we see an example of a sine wave. To represent this signal digitally, we can sample it every ΔT seconds, as shown by the bar heights in Fig. 7-31(b). If a sound wave is not a pure sine wave but a linear superposition of sine waves where the highest frequency component present is f, the Nyquist theorem (see Chap. 2) states that it is sufficient to make samples at a frequency $2f$. Sampling more often is of no value since the higher frequencies that such sampling could detect are not present.

The reverse process takes digital values and produces an analog electrical voltage. It is done by a **DAC** (**Digital-to-Analog Converter**). A loudspeaker can then convert the analog voltage to acoustic waves so that people can hear sounds.

Audio Compression

Audio is often compressed to reduce bandwidth needs and transfer times, even though audio data rates are much lower than video data rates. All compression systems require two algorithms: one is used for compressing the data at the source,

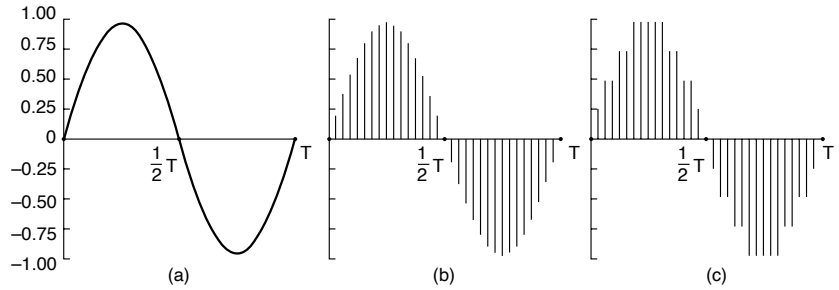

Figure 7-31. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

and another is used for decompressing it at the destination. In the literature, these algorithms are referred to as the **encoding** and **decoding** algorithms, respectively. We will use this terminology too.

Compression algorithms exhibit certain asymmetries that are important to understand. Even though we are considering audio first, these asymmetries hold for video as well. The first asymmetry applies to encoding the source material. For many applications, a multimedia document will only be encoded once (when it is stored on the multimedia server) but will be decoded thousands of times (when it is played back by customers). This asymmetry means that it is acceptable for the encoding algorithm to be slow and require expensive hardware provided that the decoding algorithm is fast and does not require expensive hardware.

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

Many audio compression algorithms have been developed. Probably the most popular formats are **MP3** (**MPEG audio layer 3**) and **AAC** (**Advanced Audio Coding**) as carried in **MP4** (**MPEG-4**) files. To avoid confusion, note that MPEG provides audio and video compression. MP3 refers to the audio compression portion (part 3) of the MPEG-1 standard, not the third version of MPEG, which has been replaced by MPEG-4. AAC is the successor to MP3 and the default audio encoding used in MPEG-4. MPEG-2 allows both MP3 and AAC audio. Is that clear now? The nice thing about standards is that there are so many to choose from. And if you do not like any of them, just wait a year or two.

Audio compression can be done in two ways. In **waveform coding**, the signal is transformed mathematically by a Fourier transform into its frequency components. In Chap. 2, we showed an example function of time and its Fourier amplitudes in Fig. 2-12(a). The amplitude of each component is then encoded in a minimal way. The goal is to reproduce the waveform fairly accurately at the other end in as few bits as possible.

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

Perceptual encoding dominates modern multimedia systems, so let us take a look at it. A key property is that some sounds can mask other sounds. For example, imagine that you are broadcasting a live flute concert on warm summer day. Then all of a sudden, a crew of workmen show up with jackhammers and start tearing up the street to replace it. No one can hear the flute any more, so you can just transmit the frequency of the jackhammers and the listeners will get the same musical experience as if you also had broadcast the flute as well, and you can save bandwidth to boot. This is called **frequency masking**.

When the jackhammers stop, you don't have to start broadcasting the flute frequency for a small period of time because the ear turns down its gain when it picks up a loud sound and it takes a bit of time to reset it. Transmission of low-amplitude sounds during this recovery period are pointless and omitting them can save bandwidth. This is called **temporal masking**. Perceptual encoding relies heavily on not encoding or transmitting audio that the listeners are not going to perceive anyway.

7.4.2 Digital Video

Now that we know all about the ear, it is time to move on to the eye. (No, this section is not followed by one on the nose.) The human eye has the property that when an image appears on the retina, the image is retained for some number of milliseconds before decaying. If a sequence of images is drawn at 50 images/sec, the eye does not notice that it is looking at discrete images. All video systems since the Lumière brothers invented the movie projector in 1895 exploit this principle to produce moving pictures.

The simplest digital representation of video is a sequence of frames, each consisting of a rectangular grid of picture elements, or **pixels**. Common sizes for screens range from 1280 × 720 (called **720p**), 1920 × 1080 (called **1080p** or **HD video**), 3840 × 2160 (called **4K**) and 7680 × 4320 (called **8K**).

Most systems use 24 bits per pixel, with 8 bits each for the red, blue, and green (RGB) components. Red, blue, and green are the primary additive colors and every other color can be made from superimposing them in the appropriate intensity.

Older frame rates vary from 24 frames/sec, which traditional film-based movies used, through 25.00 frames/sec (the PAL system used in most of the world), to 30 frames/sec (the American NTSC system). Actually, if you want to get picky, NTSC uses 29.97 frames/sec instead of 30 due to a hack the engineers introduced during the transition from black-and-white television to color. A bit of bandwidth was needed for part of the color management so they took it by reducing the frame rate by 0.03 frame/sec. PAL used color from its inception, so the rate really is exactly 25.00 frame/sec. In France, a slightly different system, called SECAM, was developed in part, to protect French companies from German television manufacturers. It also runs at exactly 25.00 frames/sec. During the 1950s, the Communist countries of Eastern Europe adopted SECAM to prevent their people from watching West German (PAL) television and getting Bad Ideas.

To reduce the amount of bandwidth required to broadcast television signals over the air, television stations adopted a scheme in which frames were divided into two **fields**, one with the odd-numbered rows and one with the even-numbered rows, which were broadcast alternately. This meant that 25 frames/sec was actually 50 fields/sec. This scheme is called **interlacing**, and gives less flicker than broadcasting entire frames one after another. Modern video does not use interlacing and and just sends entire frames in sequence, usually at 50 frames/sec (PAL) or 59.94 frames/sec (NTSC). This is called **progressive video**.

Video Compression

It should be obvious from our discussion of digital video that compression is critical for sending video over the Internet. Even 720p PAL progressive video requires 553 Mbps of bandwidth and HD, 4K, and 8K require a lot more. To produce a standard for compressing video that could be used over all platforms and by all manufacturers, the standards' committees created a group called **MPEG** (**Motion Picture Experts Group**) to come up with a worldwide standard. Very briefly, the standards it came up with, known as MPEG-1, MPEG-2, and MPEG-4, work like this. Every few seconds a complete video frame is transmitted. The frame is compressed using something like the familiar JPEG algorithm that is used for digital still pictures. Then for the next few seconds, instead of sending out full frames, the transmitter sends out differences between the current frame and the base (full) frame it most recently sent out.

First let us briefly look at the **JPEG** (**Joint Photographic Experts Group**) algorithm for compressing a single still image. Instead of working with the RGB components, it converts the image into **luminance** (brightness) and **chrominance** (color) components because the eye is much more sensitive to luminance than chrominance, allowing fewer bits to be used to encode the chrominance without loss of perceived image quality. The image is then broken up into blocks of typically 8×8 or 10×10 pixels, each of which is processed separately. Separately, the

luminance and chrominance are run through a kind of Fourier transform (technically a discrete cosine transformation) to get the spectrum. High-frequency amplitudes can then be discarded. The more amplitudes that are discarded, the fuzzier the image and the smaller the compressed image is. Then standard lossless compress techniques like run-length encoding and Huffman encoding are applied to the remaining amplitudes. If this sounds complicated, it is, but computers are pretty good at carrying out complicated algorithms.

Now on to the MPEG part, described below in a simplified way. The frame following a full JPEG (base) frame is likely to be very similar to the JPEG frame, so instead of encoding the full frame, only the blocks that differ from the base frame are transmitted. A block containing, say, a piece of blue sky is likely to be the same as it was 20 msec earlier, so there is no need to transmit it again. Only the blocks that have changed need to be retransmitted.

As an example, consider the situation of a a camera mounted securely on a tripod with an actor walking toward a stationary tree and house. The first three frames are shown in Fig. 7-32. The encoding of the second frame just sends the blocks that have changed. Conceptually, the receiver starts out producing the second frame by copying the first frame into a buffer and then applying the changes. It then stores the second frame uncompressed for display. It also uses the second frame as the base for applying the changes that come describing the difference between the third frame and the second one.

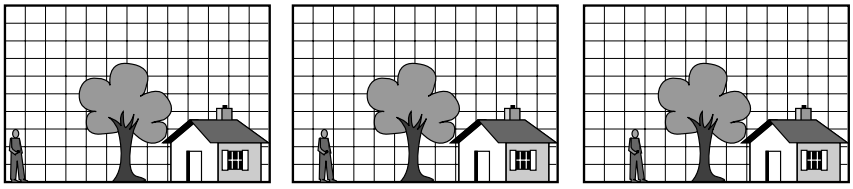

Figure 7-32. Three consecutive frames.

It is slightly more complicated than this, though. If a block (say, the actor) is present in the second frame but has moved, MPEG allows the encoder to say, in effect, ''block 29 from the previous frame is present in the new frame offset by a distance $(\Delta x, \Delta y)$ and furthermore the sixth pixel has changed to *abc* and the 24th pixel is now *xyz*.'' This allows even more compression.

We mentioned symmetries between encoding and decoding before. Here we see one. The encoder can spend as much time as it wants searching for blocks that have moved and blocks that have changed somewhat to determine whether it is better to send a list of updates to the previous frame or a complete new JPEG frame. Finding a moved block is a lot more work than simply copying a block from the previous image and pasting it into the new one at a known $(\Delta x, \Delta y)$ offset.

To be a bit more complete, MPEG actually has *three* different kinds of frames, not just two:

1. I (Intracoded) frames that are self-contained compressed still images.
2. P (Predictive) frames that are difference with the *previous* frame.
3. B (Bidirectional) frames that code differences with the *next* I-frame.

The B-frames require the receiver to stop processing until the next I-frame arrives and then work backward from it. Sometimes this gives more compression, but having the encoder constantly check to see if differences with the previous frame or differences with any one of the next 30, 50, or 80 frames gives the smallest result is time consuming on the encoding side but not time consuming on the decoding side. This asymmetry is exploited to the maximum to give the smallest possible encoded file. The MPEG standards do not specify how to search, how far to search, or how good a match has to be in order to send differences or a complete new block. This is up to each implementation.

Audio and video are encoded separately as we have described. The final MPEG-encoded file consists of chunks containing some number of compressed images and the corresponding compressed audio to be played while the frames in that chunk are displayed. In this way, the video and audio are kept synchronized.

Note that this is a rather simplified description. In reality, even more tricks are used to get better compression, but the basic ideas given above are essentially correct. The most recent format is MPEG-4, also called MP4. It is formally defined in a standard known as H.264. It's successor (defined for resolutions up to 8K) is H.265. H.264 is the format most consumer video cameras produce. Because the camera has to record the video on the SD card or other medium in real time, it has very little time to hunt for blocks that have moved a little. Consequently, the compression is not nearly as good as what a Hollywood studio can do when it dynamically allocates 10,000 computers at a cloud server to encode its latest production. This is encoding/decoding asymmetry in action.

7.4.3 Streaming Stored Media

Let us now move on to network applications. Our first case is streaming a video that is already stored on a server somewhere, for example, watching a YouTube or Netflix video. The most common example of this is watching videos over the Internet. This is one form of **VoD** (**Video on Demand**). Other forms of video on demand use a provider network that is separate from the Internet to deliver the movies (e.g., the cable TV network).

The Internet is full of music and video sites that stream stored multimedia files. Actually, the easiest way to handle stored media is *not* to stream it. The straightforward way to make the video (or music track) available is just to treat the

pre-encoded video (or audio) file as a very big Web page and let the browser download it. The sequence of four steps is shown in Fig. 7-33.

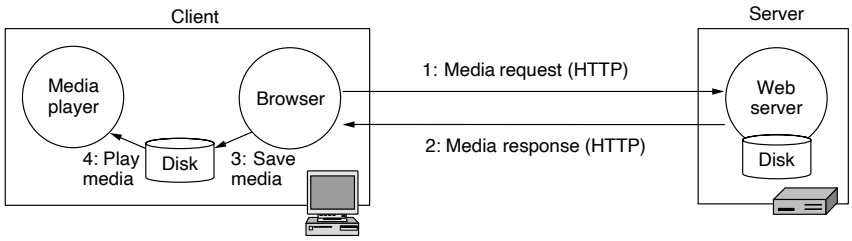

Figure 7-33. Playing media over the Web via simple downloads.

The browser goes into action when the user clicks on a movie. In step 1, it sends an HTTP request for the movie to the Web server to which the movie is linked. In step 2, the server fetches the movie (which is just a file in MP4 or some other format) and sends it back to the browser. Using the MIME type, the browser looks up how it is supposed to display the file. The browser then saves the entire movie to a scratch file on disk in step 3. It then starts the media player, passing it the name of the scratch file. Finally, in step 4 the media player starts reading the file and playing the movie. Conceptually, this is no different than fetching and displaying a static Web page, except that the downloaded file is ''displayed'' by using a media player instead of just writing pixels to a monitor.

In principle, this approach is completely correct. It will play the movie. There is no real-time network issue to address either because the download is simply a file download. The only trouble is that the entire video must be transmitted over the network before the movie starts. Most customers do not want to wait an hour for their ''video on demand'' to start, so something better is needed.

What is needed is a media player that is designed for streaming. It can either be part of the Web browser or an external program called by the browser when a video needs to be played. Modern browsers that support HTML5 usually have a built-in media player.

A media player has five major jobs to do:

1. Manage the user interface.
2. Handle transmission errors.
3. Decompress the content.
4. Eliminate jitter.
5. Decrypt the file.

Most media players nowadays have a glitzy user interface, sometimes simulating a stereo unit, with shiny buttons, knobs, sliders, and visual displays. Often there are

interchangeable front panels, called **skins**, that the user can drop onto the player. The media player has to manage all this and interact with the user.

The next three are related and depend on the network protocols. We will go through each one in turn, starting with handling transmission errors. Dealing with errors depends on whether a TCP-based transport like HTTP is used to transport the media, or a UDP-based transport like **RTP** (**Real Time Protocol**) is used. If a TCP-based transport is being used then there are no errors for the media player to correct because TCP already provides reliability by using retransmissions. This is an easy way to handle errors, at least for the media player, but it does complicate the removal of jitter in a later step because timing out and asking for retransmissions introduces uncertain and variable delays in the movie.

Alternatively, a UDP-based transport like RTP can be used to move the data. With these protocols, there are no retransmissions. Thus, packet loss due to congestion or transmission errors will mean that some of the media does not arrive. It is up to the media player to deal with this problem. One way is to ignore the problem and just have bits of video and audio be wrong. If errors are infrequent, this works fine and almost no one will notice. Another possibility is to use **forward error correction**, such as encoding the video file with some redundancy, such as a Hamming code or a Reed-Solomon code. Then the media player will have enough information to correct errors on its own, without having to ask for retransmissions or skip bits of damaged movies.

The downside here is that adding redundancy to the file makes it bigger. Another approach involves using selective retransmission of the parts of the video stream that are most important to play back the content. For example, in a compressed video sequence, a packet loss in an I-frame is much more consequential, since the decoding errors that result from the loss can propagate throughout the group of pictures. On the other hand, losses in derivative frames, including P-frames and B-frames, are easier to recover from. Similarly, the value of a retransmission also depends on whether the retransmission of the content would arrive in time for playback. As a result, some retransmissions can be far more valuable than others, and selectively retransmitting certain packets (e.g., those within I-frames that would arrive before playback) is one possible strategy. Protocols have been built on top of RTP and QUIC to provide unequal loss protection when videos are streamed over UDP (Feamster et al., 2000; and Palmer et al., 2018).

The media player's third job is decompressing the content. Although this task is computationally intensive, it is fairly straightforward. The thorny issue is how to decode media if the underlying network protocol does not correct transmission errors. In many compression schemes, later data cannot be decompressed until the earlier data has been decompressed, because the later data is encoded relative to the earlier data. Recall that a P-frame is based upon the most recent I-frame (and other I-frames following it). If the I-frame is damaged and cannot be decoded, all the subsequent P-frames are useless. The media player will then be forced to wait for the next I-frame and simply skip a few seconds of video.

This reality forces the encoder to make a decision. If I-frames are spaced closely, say, one per second, the gap when an error occurs will be fairly small, but the video will be bigger because I-frames are much bigger than P- or B-frames. If I-frames are, say, 5 seconds apart, the video file will be much smaller but there will be 5-second gap if an I-frame is damaged and a smaller gap if a P-frame is damaged. For this reason, when the underlying protocol is TCP, I-frames can be spaced much further apart than if RTP is used. Consequently, many video-streaming sites use TCP to allow a smaller encoded file with widely spaced I-frames and less bandwidth needed for smooth playback.

The fourth job is to eliminate jitter, the bane of all real-time systems. Using TCP makes this much worse, because it introduces random delays whenever retransmissions are needed. The general solution that all streaming systems use is a playout buffer. before starting to play the video, the system collects 5–30 seconds worth of media, as shown in Fig. 7-34. Playing drains media regularly from the buffer so that the audio is clear and the video is smooth. The startup delay gives the buffer a chance to fill to the **low-water mark**. The idea is that data should now arrive regularly enough that the buffer is never completely emptied. If that were to happen, the media playout would stall.

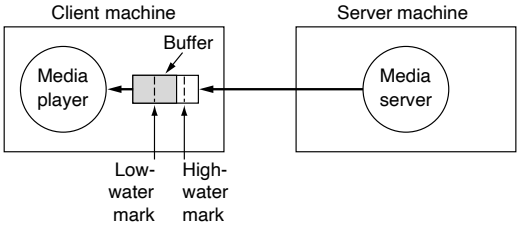

Figure 7-34. The media player buffers input from the media server and plays from the buffer rather than directly from the network.

Buffering introduces a new complication. The media player needs to keep the buffer partly full, ideally between the low-water mark and the high-water mark. This means when the buffer passes the high-water mark, the player needs to tell the source to stop sending, lest it lose data for lack of a place to put it. The high-water mark has to be before the end of the buffer because data will continue to stream in until the *Stop* request gets to the media server. Once the server stops sending and the pipeline is empty, the buffer will start draining. When it hits the low-water mark, the player sends a *Start* command to the server to start streaming again.

By using a protocol in which the media player can command the server to stop and start, the media player can keep enough, but not too much, media in the buffer to ensure smooth playout. Since RAM is fairly cheap these days, a media player, even on a smartphone, could allocate enough buffer space to hold a minute or more of media, if need be.

The start-stop mechanism has another nice feature. It decouples the server's transmission rate from the playout rate. Suppose, for example, that the player has to play out the video at 8 Mbps. When the buffer drops to the low-water mark, the player will tell the server to deliver more data. If the server is capable of delivering it at 100 Mbps, that is not a problem. It just comes in and is stored in the buffer. When the high-water mark is reached, the player tells the server to stop. In this way, the server's transmission rate and the playout rate are completely decoupled. What started out as a real-time system has become a simple nonreal-time file transfer system. Getting rid of all the real-time transmission requirements is another reason YouTube, Netflix, Hulu, and other streaming servers use TCP. It makes the whole system design much simpler.

Determining the size of the buffer is a bit tricky. If lots of RAM is available, at first glance it sounds like it might make sense to have a large buffer and allow the server to keep it almost full, just in case the network suffers some congestion later on. However, users are sometimes finicky. If a user finds a scene boring and uses the buttons on the media player's interface to skip forward, that might render most or all of the buffer useless. In any event, jumping forward (or backward) to a specific point in time is unlikely to work unless that frame happens to be an I-frame. If not, the player has to search for a nearby I-frame. If the new play point is outside the buffer, the entire buffer has to be cleared and reloaded. In effect, users who skip around a lot (and there are many of them), waste network bandwidth by invalidating precious data in their buffers. Systemwide, the existence of users who skip around a lot argues for limiting the buffer size, even if there is plenty of RAM available. Ideally, a media player could observe the user's behavior and pick a buffer size to match the user's viewing style.

All commercial videos are encrypted to prevent piracy, so media players have to be able to decrypt them as them come in. That is the fifth task in the list above.

DASH and HLS

The plethora of devices for viewing media introduces some complications we need to look at now. Someone who buys a bright, shiny, and very expensive 8K monitor will want movies delivered in 7680×4320 resolution at 100 or 120 frames/sec. But if halfway through an exciting movie she has to go to the doctor and wants to finish watching it in the waiting room on a 1280×720 smartphone that can handle at most 25 frames/sec, she has a problem. From the streaming site's point of view, this raises the question of what at resolution and frame rate should movies be encoded.

The easy answer is to use every possible combination. At most it wastes disk space to encode every movie at seven screen resolutions (e.g., smartphone, NTSC, PAL, 720p, HD, 4K, and 8K) amd six frame rates (e.g., 25, 30, 50, 60, 100, and 120), for a total of 42 variants, but disk space is not very expensive. A bigger, but

related problem. is what happens when the viewer is stationary at home with her big, shiny monitor, but due to network congestion, the bandwidth between her and the server is changing wildly and cannot always support the full resolution.

Fortunately, several solutions have been already implemented. One solution is **DASH** (**Dynamic Adaptive Streaming over HTTP**). The basic idea is simple and it is compatible with HTTP (and HTTPS), so it can be streamed on a Web page. The streaming server first encodes its movies at multiple resolutions and frame rates and has them all stored in its disk farm. Each version is not stored as a single file, but as many files, each storing, say, 10 seconds of video and audio. This would mean that a 90-minute movie with seven screen resolutions and six frame rates (42 variants) would require $42 \times 540 = 22{,}680$ separate files, each with 10 seconds worth of content. In other words, each file holds a segment of the movie at one specific resolution and frame rate. Associated with the movie is a manifest, officially known as an **MPD** (**Media Presentation Description**), which lists the names of all these files and their properties, including resolution, frame rate, and frame number in the movie.

To make this approach work, both the player and server must both use the DASH protocol. The user side could either be the browser itself, a player shipped to the browser as a JavaScript program, or a custom application (e.g., for a mobile device, or a streaming set top box). The first thing it does when it is time to start viewing the movie is fetch the manifest for the movie, which is just a small file, so a normal *GET* HTTPS request is all that is needed.

The player then interrogates the device where it is running to discover its maximum resolution and possibly other characteristics, such as what audio formats it can handle and how many speakers it has. Then it begins running some tests by sending test messages to the server to try to estimate how much bandwidth is available. Once it has figured out what resolution the screen has and how much bandwidth is available, the player consults the manifest to find the first, say, 10 seconds of the movie that gives the best quality for the screen and available bandwidth.

But that's not the end of the story. As the movie plays, the player continues to run bandwidth tests. Every time it needs more content, that is, when the amount of media in the buffer hits the low-water mark, it again consults the manifest and orders the appropriate file depending where it is in the movie and which resolution and frame rate it wants. If the bandwidth varies wildly during playback, the movie shown may change from 8K at 100 frames/sec to HD at 25 frames/sec and back several times a minute. In this way, the system adapts rapidly to changing network conditions and allows the best viewing experience consistent with the available resources. Companies such as Netflix have published information about how they adapt the bitrate of a video stream based on the playback buffer occupancy (Huang et al., 2014). An example is shown in Fig. 7-35.

In Fig. 7-35, as the bandwidth decreases, the player decides to ask for increasingly low resolution versions. However, it could also have compromised in other ways. For example, sending out 300 frames for a 10-second playout requires less

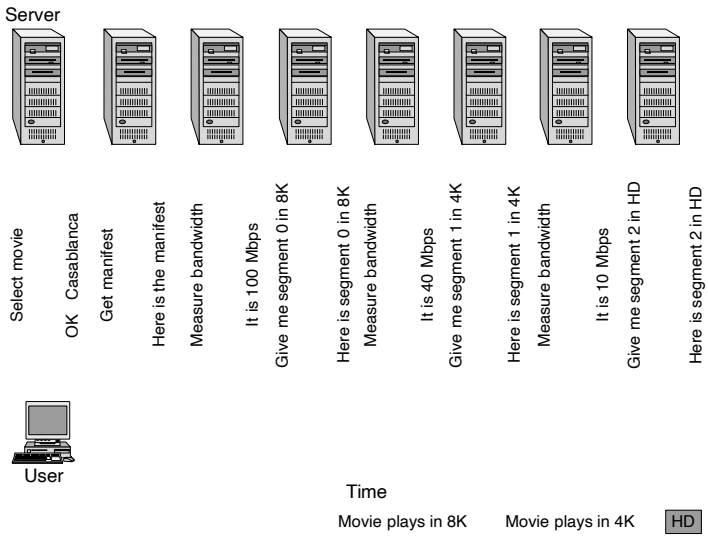

Figure 7-35. DASH being used to change format while watching a movie.

bandwidth than sending out 600 or 1200 frames for a 10-second playout, even with good compression. In a real pinch, it could also have asked for a 10 frames/sec version at 480 × 320 in black-and-white with monaural sound if that is on the manifest. DASH allows the player to adapt to changing circumstances to give the user the best possible experience for the current circumstances. The behavior of the player and how it requests segments varies depending on the nature of the playback service and the device. Services whose goal is to avoid rebuffering events might requests a large number of segments before playing back video and to request segments in batches; other services whose goal is interactivity might fetch DASH segments at a more consistent, steady pace.

DASH is still evolving. For example, work is going on to reduce the latency (Le Feuvre et al., 2015), improve the robustness (Wang and Ren, 2019), fairness (Altamini, S., and Shirmohammadi, S, 2019), support virtual reality (Ribezzo et al., 2018), and handle 4K videos well (Quinlan and Sreenan, 2018).

DASH is the most common method for streaming video today, although there are some alternatives worth discussing. Apple's **HLS** (**HTTP Live Streaming**) also works in a browser using HTTP. It is the preferred method for viewing video in Safari on iPhones, iPads, MacBooks, and all Apple devices. It is also widely used by browsers such as Microsoft Edge, Firefox, and Chrome, on Windows, Linux, and Android platforms. It is also supported by many game consoles, smart TVs and other devices that can play multimedia content.

Like DASH, HLS requires the server to encode the movie in multiple resolutions and frame rates, with each segment covering only a few seconds of video to provide for rapid adaptation to changing conditions. HLS also has other features, including fast forward, fast backward, subtitles in multiple languages, and more. It is described in RFC 8216.

While the basic principles are the same, DASH and HLS differ in some ways. DASH is codec agnostic, which means works with videos using any encoding algorithm. HLS works only with algorithms that Apple supports, but since these include H.264 and H.265, this difference is minor because almost all videos use one of these. DASH allows third parties to easily insert ads into the video stream, which HLS does not. DASH can handle arbitrary digital rights management schemes, whereas HLS supports only Apple's own system.

DASH is an open official standard, whereas HLS is a proprietary product. But that cuts both ways. Because HLS has a powerful sponsor behind it, it is available on many more platforms than DASH and the implementations are extremely stable. On the other hand, YouTube and Netflix both use DASH. However, DASH is not natively supported on iOS devices. Most likely the two protocols will continue to coexist for years to come.

Video streaming has been a major force driving the Internet for decades. For a retrospective, see Li et al. (2013).

An ongoing challenge with streaming video is estimating user **QoE** (**Quality of Experience**) which is, informally, how happy a user is with the performance of the video streaming application. Obviously, measuring QoE directly is challenging (it requires asking users about their experience), but network operators are increasingly aiming to determine when video streaming applications experience conditions that may affect a user's experience. Generally speaking, the parameters that operators aim to estimate are the startup delay (how long a video takes to start playing), the resolution of the video, and any instances of stalling (''rebuffering''). It can be challenging to identify these events in an encrypted video stream, particularly for an ISP that does not have access to the client software; machine learning techniques are increasingly being used to infer application quality from encrypted video traffic streams (Mangla et al., 2018; and Bronzino et al., 2020).

7.4.4 Real-Time Streaming

It is not only recorded videos that are tremendously popular on the Web. Real-time streaming is very popular too. Once it became possible to stream audio and video over the Internet, commercial radio and TV stations got the idea of broadcasting their content over the Internet as well as over the air. Not so long after that, college stations started putting their signals out over the Internet. Then college *students* started their own Internet broadcasts.

Today, people and companies of all sizes stream live audio and video. The area is a hotbed of innovation as the technologies and standards evolve. Live streaming

is used for an online presence by major television stations. This is called **IPTV** (**IP TeleVision**). It is also used to broadcast radio stations. This is called **Internet radio**. Both IPTV and Internet radio reach audiences worldwide for events ranging from fashion shows to World Cup soccer and test matches live from the Newlands Cricket Ground. Live streaming over IP is used as a technology by cable providers to build their own broadcast systems. And it is widely used by low-budget operations from adult sites to zoos. With current technology, virtually anyone can start live streaming quickly and with little expense.

One approach to live streaming is to record programs to disk. Viewers can connect to the server's archives, pull up any program, and download it for listening. A **podcast** is an episode retrieved in this manner.

Streaming live events adds new complications to the mix, at least sometimes. For sports, news broadcasts, and politicians giving long boring speeches, the method of Fig. 7-34 still works. When a user logs onto the Web site covering the live event, no video is shown for the first few seconds while the buffer fills. After that, it is the same as watching a movie. The player pulls data out of the buffer, which is continuously filled by the feed from the live event. The only real difference is that when streaming a movie from a server, the server can potentially load 10 seconds worth of movie in one second if the connection is fast enough. With a live event, that is not possible.

Voice over IP

A good example of real-time streaming where buffering is not possible is using the Internet to transmit telephone calls (possibly with video, as Skype, FaceTime, and many other services do). Once upon a time, voice calls were carried over the public switched telephone network, and network traffic was primarily voice traffic, with a little bit of data traffic here and there. Then came the Internet, and the Web. The data traffic grew and grew, until by 1999 there was as much data traffic as voice traffic (since voice is now digitized, both can be measured in bits). By 2002, the volume of data traffic was an order of magnitude more than the volume of voice traffic and still growing exponentially, with voice traffic staying almost flat. Now the data traffic is orders of magnitude more than the voice traffic.

The consequence of this growth has been to flip the telephone network on its head. Voice traffic is now carried using Internet technologies, and represents only a tiny fraction of the network bandwidth. This disruptive technology is known as **voice over IP**, and also as **Internet telephony**. (As an aside, ''Telephony'' is pronounced ''te-LEF-ony.'') It is also called that when the calls include video or are multiparty, that is, videoconferencing.

The biggest difference streaming a movie over the Internet and Internet telephony is the need for low latency. The telephone network allows a one-way latency of up to 150 msec for acceptable usage, after which delay begins to be perceived as annoying by the participants. (International calls may have a latency of up to 400 msec, by which point they are far from a positive user experience.)

This low latency is difficult to achieve. Certainly, buffering 5–10 seconds of media is not going to work (as it would for broadcasting a live sports event). Instead, video and voice-over-IP systems must be engineered with a variety of techniques to minimize latency. This goal means starting with UDP as the clear choice rather than TCP, because TCP retransmissions introduce at least one round-trip worth of delay.

Some forms of latency cannot be reduced, however, even with UDP. For example, the distance between Seattle and Amsterdam is close to 8,000 km. The speed-of-light propagation delay for this distance in optical fiber is 40 msec. Good luck beating that. In practice, the propagation delay through the network will be longer because it will cover a larger distance (the bits do not follow a great circle route) and have transmission delays as each IP router stores and forwards a packet. This fixed delay eats into the acceptable delay budget.

Another source of latency is related to packet size. Normally, large packets are the best way to use network bandwidth because they are more efficient. However, at an audio sampling rate of 64 kbps, a 1-KB packet would take 125 msec to fill (and even longer if the samples are compressed). This delay would consume most of the overall delay budget. In addition, if the 1-KB packet is sent over a broadband access link that runs at just 1 Mbps, it will take 8 msec to transmit. Then add another 8 msec for the packet to go over the broadband link at the other end. Clearly, large packets will not work.

Instead, voice-over-IP systems use short packets to reduce latency at the cost of bandwidth efficiency. They batch audio samples in smaller units, commonly 20 msec. At 64 kbps, this is 160 bytes of data, less with compression. However, by definition the delay from this packetization will be 20 msec. The transmission delay will be smaller as well because the packet is shorter. In our example, it would reduce to around 1 msec. By using short packets, the minimum one-way delay for a Seattle-to-Amsterdam packet has been reduced from an unacceptable 181 msec (40 + 125 + 16) to an acceptable 62 msec (40 + 20 + 2).

We have not even talked about the software overhead, but it, too, will eat up some of the delay budget. This is especially true for video, since compression is usually needed to fit video into the available bandwidth. Unlike streaming from a stored file, there is no time to have a computationally intensive encoder for high levels of compression. The encoder and the decoder must both run quickly.

Buffering is still needed to play out the media samples on time (to avoid unintelligible audio or jerky video), but the amount of buffering must be kept very small since the time remaining in our delay budget is measured in milliseconds. When a packet takes too long to arrive, the player will skip over the missing samples, perhaps playing ambient noise or repeating a frame to mask the loss to the user. There is a trade-off between the size of the buffer used to handle jitter and the amount of media that is lost. A smaller buffer reduces latency but results in more loss due to jitter. Eventually, as the size of the buffer shrinks, the loss will become noticeable to the user.

Observant readers may have noticed that we have said nothing about the *network layer* protocols so far in this section. The network can reduce latency, or at least jitter, by using quality of service mechanisms. The reason that this issue has not come up before is that streaming is able to operate with substantial latency, even in the live streaming case. If latency is not a major concern, a buffer at the end host is sufficient to handle the problem of jitter. However, for real-time conferencing, it is usually important to have the network reduce delay and jitter to help meet the delay budget. The only time that it is not important is when there is so much network bandwidth that everyone gets good service.

In Chap. 5, we described two quality of service mechanisms that help with this goal. One mechanism is DS (Differentiated Services), in which packets are marked as belonging to different classes that receive different handling within the network. The appropriate marking for voice-over-IP packets is low delay. In practice, systems set the DS codepoint to the well-known value for the *Expedited Forwarding* class with *Low Delay* type of service. This is especially useful over broadband access links, as these links tend to be congested when Web traffic or other traffic competes for use of the link. Given a stable network path, delay and jitter are increased by congestion. Every 1-KB packet takes 8 msec to send over a 1-Mbps link, and a voice-over-IP packet will incur these delays if it is sitting in a queue behind Web traffic. However, with a low delay marking the voice-over-IP packets will jump to the head of the queue, bypassing the Web packets and lowering their delay.

The second mechanism that can reduce delay is to make sure that there is sufficient bandwidth. If the available bandwidth varies or the transmission rate fluctuates (as with compressed video) and there is sometimes not sufficient bandwidth, queues will build up and add to the delay. This will occur even with DS. To ensure sufficient bandwidth, a reservation can be made with the network. This capability is provided by integrated services.

Unfortunately, it is not widely deployed. Instead, networks are engineered for an expected traffic level or network customers are provided with service-level agreements for a given traffic level. Applications must operate below this level to avoid causing congestion and introducing unnecessary delays. For casual videoconferencing at home, the user may choose a video quality as a proxy for bandwidth needs, or the software may test the network path and select an appropriate quality automatically.

Any of the above factors can cause the latency to become unacceptable, so real-time conferencing requires that attention be paid to all of them. For an overview of voice over IP and analysis of these factors, see Sun et al. (2015).

Now that we have discussed the problem of latency in the media streaming path, we will move on to the other main problem that conferencing systems must address. This problem is how to set up and tear down calls. We will look at two protocols that are widely used for this purpose, H.323 and SIP. Skype and FaceTime are other important systems, but their inner workings are proprietary.

H.323

One thing that was clear to everyone before voice and video calls were made over the Internet was that if each vendor designed its own protocol stack, the system would never work. To avoid this problem, a number of interested parties got together under ITU auspices to work out standards. In 1996, ITU issued recommendation **H.323**, entitled ''Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-Guaranteed Quality of Service.'' Only the telephone industry would come up with such a name. After some criticism, It was changed to ''Packet-based Multimedia Communications Systems'' in the 1998 revision. H.323 was the basis for the first widespread Internet conferencing systems. It is still widely used.

H.323 is more of an architectural overview of Internet telephony than a specific protocol. It references a large number of specific protocols for speech coding, call setup, signaling, data transport, and other areas rather than specifying these things itself. The general model is depicted in Fig. 7-36. At the center is a **gateway** that connects the Internet to the telephone network. It speaks the H.323 protocols on the Internet side and the PSTN protocols on the telephone side. The communicating devices are called **terminals**. A LAN may have a **gatekeeper**, which controls the end points under its jurisdiction, called a **zone**.

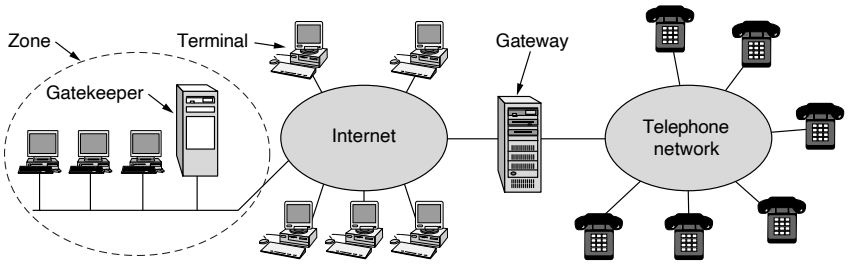

Figure 7-36. The H.323 architectural model for Internet telephony.

A telephone network needs a number of protocols. To start with, there is a protocol for encoding and decoding audio and video. Standard telephony representations of a single voice channel as 64 kbps of digital audio (8000 samples of 8 bits per second) are defined in ITU recommendation **G.711**. All H.323 systems must support G.711. Other encodings that compress speech are permitted, but not required. They use different compression algorithms and make different trade-offs between quality and bandwidth. For video, the MPEG forms of video compression that we described above are supported, including H.264.

Since multiple compression algorithms are permitted, a protocol is needed to allow the terminals to negotiate which one they are going to use. This protocol is called **H.245**. It also negotiates other aspects of the connection such as the bit rate.

RTCP is need for the control of the RTP channels. Also required is a protocol for establishing and releasing connections, providing dial tones, making ringing sounds, and the rest of the standard telephony. ITU **Q.931** is used here. The terminals need a protocol for talking to the gatekeeper (if present) as well. For this purpose, **H.225** is used. The PC-to-gatekeeper channel it manages is called the **RAS** (**Registration/Admission/Status**) channel. This channel allows terminals to join and leave the zone, request and return bandwidth, and provide status updates, among other things. Finally, a protocol is needed for the actual data transmission. RTP over UDP is used for this purpose. It is managed by RTCP, as usual. The positioning of all these protocols is shown in Fig. 7-37.

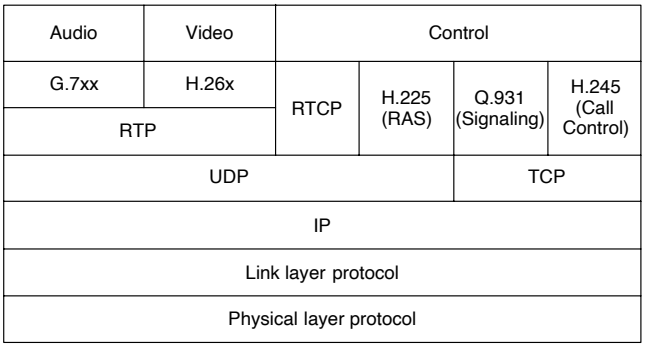

Figure 7-37. The H.323 protocol stack.

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

As an aside, the telephone system does the same thing. When you pick up the receiver, a signal is sent to the local end office. If the office has enough spare capacity for another call, it generates a dial tone. If not, you hear nothing. Nowadays, the system is so overdimensioned that the dial tone is nearly always instantaneous, but in the early days of telephony, it often took a few seconds. So if your grandchildren ever ask you ''Why are there dial tones?'' now you know. Except by then, probably telephones will no longer exist.

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

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

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

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

The H.245 protocol is now used to negotiate the parameters of the call. It uses the H.245 control channel, which is always open. Each side starts out by announcing its capabilities, for example, whether it can handle video (H.323 can handle video) or conference calls, which codecs it supports, etc. Once each side knows what the other one can handle, two unidirectional data channels are set up and a codec and other parameters are assigned to each one. Since each side may have different equipment, it is entirely possible that the codecs on the forward and reverse channels are different. After all negotiations are complete, data flow can begin using RTP. It is managed using RTCP, which plays a role in congestion control. If video is present, RTCP handles the audio/video synchronization. The various channels are shown in Fig. 7-38. When either party hangs up, the Q.931 call signaling channel is used to tear down the connection after the call has been completed in order to free up resources no longer needed.

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

We have not said anything about quality of service for H.323, even though we have said it is an important part of making real-time conferencing a success. The reason is that QoS falls outside the scope of H.323. If the underlying network is capable of producing a stable, jitter-free connection from the calling PC to the gateway, the QoS on the call will be good; otherwise, it will not be. However, any

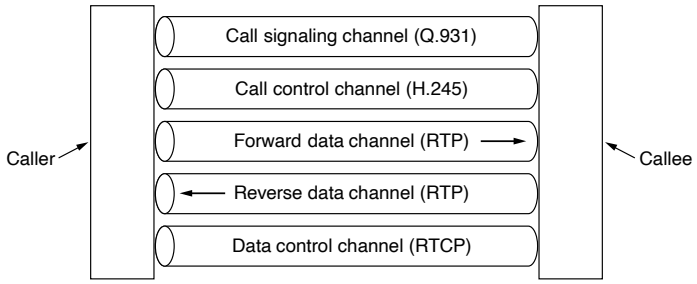

Figure 7-38. Logical channels between the caller and callee during a call.

portion of the call on the telephone side will be jitter-free, because that is how the telephone network is designed.

SIP—The Session Initiation Protocol

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

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

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

Telephone numbers in SIP are represented as URLs using the *sip* scheme, for example, *sip:ilse@cs.university.edu* for a user named Ilse at the host specified by

the DNS name *cs.university.edu*. SIP URLs may also contain IPv4 addresses, IPv6 addresses, or actual telephone numbers.

The SIP protocol is a text-based protocol modeled on HTTP. One party sends a message in ASCII text consisting of a method name on the first line, followed by additional lines containing headers for passing parameters. Many of the headers are taken from MIME to allow SIP to interwork with existing Internet applications. The six methods defined by the core specification are listed in Fig. 7-39.

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

Figure 7-39. SIP methods.

To establish a session, the caller either creates a TCP connection with the callee and sends an *INVITE* message over it or sends the *INVITE* message in a UDP packet. In both cases, the headers on the second and subsequent lines describe the structure of the message body, which contains the caller's capabilities, media types, and formats. If the callee accepts the call, it responds with an HTTP-type reply code (a three-digit number using the groups of Fig. 7-26, 200 for acceptance). Following the reply-code line, the callee also may supply information about its capabilities, media types, and formats.

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

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

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

The *REGISTER* method relates to SIP's ability to track down and connect to a user who is away from home. This message is sent to a SIP location server that keeps track of who is where. That server can later be queried to find the user's current location. The operation of redirection is illustrated in Fig. 7-40. Here, the caller sends the *INVITE* message to a proxy server to hide the possible redirection. The proxy then looks up where the user is and sends the *INVITE* message there. It then acts as a relay for the subsequent messages in the three-way handshake. The *LOOKUP* and *REPLY* messages are not part of SIP; any convenient protocol can be used, depending on what kind of location server is used.

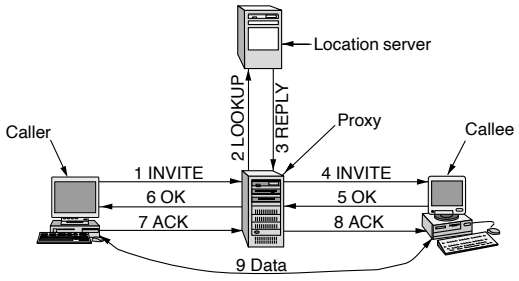

Figure 7-40. Use of a proxy server and redirection with SIP.

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

Comparison of H.323 and SIP

Both H.323 and SIP allow two-party and multiparty calls using both computers and telephones as end points. Both support parameter negotiation, encryption, and the RTP/RTCP protocols. A summary of their similarities and differences is given in Fig. 7-41.

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

In contrast, SIP is a typical Internet protocol that works by exchanging short lines of ASCII text. It is a lightweight module that interworks well with other Internet protocols but less well with existing telephone system signaling protocols. Because the IETF model of voice over IP is highly modular, it is flexible and can be adapted to new applications easily. The downside is that is has suffered from interoperability problems as people try to interpret what the standard means.

7.5 CONTENT DELIVERY

The Internet used to be all about point-to-point communication, much like the telephone network. Early on, academics would communicate with remote computers, logging in over the network to perform tasks. People have used email to

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

Figure 7-41. Comparison of H.323 and SIP.

communicate with each other for a long time, and now use video and voice over IP as well. Since the Web grew up, however, the Internet has become more about content than communication. Many people use the Web to find information, and there is a tremendous amount of downloading of music, videos, and other material. The switch to content has been so pronounced that the majority of Internet bandwidth is now used to deliver stored videos.

Because the task of distributing content is different from that of point-to-point communication, it places different requirements on the network. For example, if Sally wants to talk to John, she may make a voice-over-IP call to his mobile. The communication must be with a particular computer; it will do no good to call Paul's computer. But if John wants to watch his team's latest cricket match, he is happy to stream video from whichever computer can provide the service. He does not mind whether the computer is Sally's or Paul's, or, more likely, an unknown server in the Internet. That is, location does not matter for content, except as it affects performance (and legality).

The other difference is that some Web sites that provide content have become tremendously popular. YouTube is a prime example. It allows users to share videos of their own creation on every conceivable topic. Many people want to do this. The rest of us want to watch. Internet traffic today is upwards of 70% streaming

video, with the vast majority of that streaming video traffic being delivered by a small number of content providers.

No single server is powerful or reliable enough to handle such a startling level of demand. Instead, YouTube, Netflix, and other large content providers build their own content distribution networks. These networks use data centers spread around the world to serve content to an extremely large number of clients with good performance and availability.

The techniques that are used for content distribution have been developed over time. Early in the growth of the Web, its popularity was almost its undoing. More demands for content led to servers and networks that were frequently overloaded. Many people began to call the WWW the World Wide Wait. To reduce the endless delays, researchers developed different architectures to use the bandwidth for distributing content.

A common architecture for distributing content architecture is a **CDN** (**Content Delivery Network**), sometimes also called a **Content Distribution Network**. A CDN is effectively a very large distributed set of caches, which typically serves content directly to clients. CDNs were once exclusively the purview of only the large content providers; a content provider with popular content might pay a CDN such ask Akamai to distribute their content, effectively prepopulating its caches with the content that needed to be distributed. Today, many large content providers, including Netflix, Google, and even many ISPs that host their own content (e.g., Comcast) now operate their own CDNs.

Another way to distribute content is via a **P2P** (**Peer-to-Peer**) network, whereby computers serve content to each other, typically without separately provisioned servers or any central point of control. This idea has captured people's imagination because, by acting together, many little players can pack an enormous punch.

7.5.1 Content and Internet Traffic

To design and engineer networks that work well, we need an understanding of the traffic that they must carry. With the shift to content, for example, servers have migrated from company offices to Internet data centers that provide large numbers of machines with excellent network connectivity. To run even a small server nowadays, it is easier and cheaper to rent a virtual server hosted in an Internet data center than to operate a real machine in a home or office with broadband connectivity to the Internet.

Internet traffic is highly skewed. Many properties with which we are familiar are clustered around an average. For instance, most adults are close to the average height. There are some tall people and some short people, but few are very tall or very short. Similarly, most novels are a few hundred pages; very few are 20 pages or 10,000 pages. For these kinds of properties, it is possible to design for a range that is not very large but nonetheless captures the majority of the population.

Internet traffic is not like this. For a long time, it has been known that there are a small number of Web sites with massive traffic (e.g., Google, YouTube, and Facebook) and a vast number of Web sites with much smaller traffic.

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

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

Thus, the most popular movie is seven times as popular as the number seven movie. This result is known as **Zipf's law** (Zipf, 1949). It is named after George Zipf, a professor of linguistics at Harvard University who noted that the frequency of a word's usage in a large body of text is inversely proportional to its rank. For example, the 40th most common word is used twice as much as the 80th most common word and three times as much as the 120th most common word.

A Zipf distribution is shown in Fig. 7-42(a). It captures the notion that there are a small number of popular items and a great many unpopular items. To recognize distributions of this form, it is convenient to plot the data on a log scale on both axes, as shown in Fig. 7-42(b). The result should be a straight line.

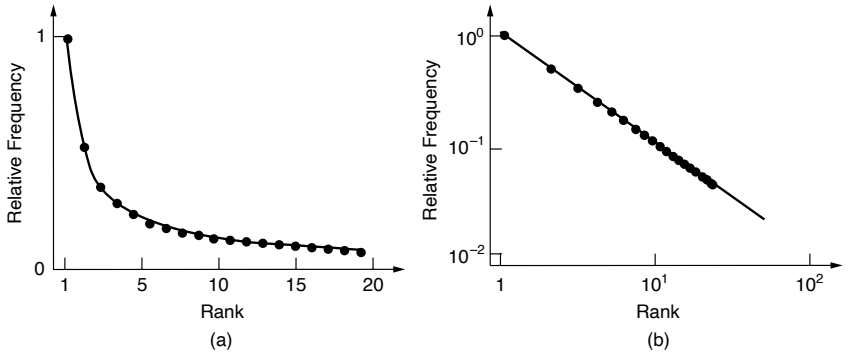

Figure 7-42. Zipf distribution (a) On a linear scale. (b) On a log-log scale.

When people first looked at the popularity of Web pages, it also turned out to roughly follow Zipf's law (Breslau et al., 1999). A Zipf distribution is one example in a family of distributions known as **power laws**. Power laws are evident in many human phenomena, such as the distribution of city populations and of wealth. They have the same propensity to describe a few large players and a great many smaller players, and they too appear as a straight line on a log-log plot. It was soon discovered that the topology of the Internet could be roughly described with power laws (Siganos et al., 2003). Next, researchers began plotting every

imaginable property of the Internet on a log scale, observing a straight line, and shouting: ''Power law!''

However, what matters more than a straight line on a log-log plot is what these distributions mean for the design and use of networks. Given the many forms of content that have Zipf or power law distributions, it seems fundamental that Web sites on the Internet are Zipf-like in popularity. This in turn means that an *average* site is not a useful representation. Sites are better described as either popular or unpopular. Both kinds of sites matter. The popular sites obviously matter, since a few popular sites may be responsible for most of the traffic on the Internet. Perhaps surprisingly, the unpopular sites can matter too. This is because the total amount of traffic directed to the unpopular sites can add up to a large fraction of the overall traffic. The reason is that there are so many unpopular sites. The notion that, collectively, many unpopular choices can matter has been popularized by books such as *The Long Tail* (Anderson, 2008a).

To work effectively in this skewed world, we must be able to build both kinds of Web sites. Unpopular sites are easy to handle. By using DNS, many different sites may actually point to the same computer in the Internet that runs all of the sites. On the other hand, popular sites are difficult to handle. There is no single computer even remotely powerful enough, and using a single computer would make the site inaccessible for millions of users when (*not* if) it fails. To handle these sites, we must build content distribution systems. We will start on that quest next.

7.5.2 Server Farms and Web Proxies

The Web designs that we have seen so far have a single server machine talking to multiple client machines. To build large Web sites that perform well, we can speed up processing on either the server side or the client side. On the server side, more powerful Web servers can be built with a server farm, in which a cluster of computers acts as a single server. On the client side, better performance can be achieved with better caching techniques. In particular, proxy caches provide a large shared cache for a group of clients.

We will describe each of these techniques in turn. However, note that neither technique is sufficient to build the largest Web sites. Those popular sites require the content distribution methods that we describe in the following sections, which combine computers at many different locations.

Server Farms

No matter how much computing capacity and bandwidth one machine has, it can only serve so many Web requests before the load is too great. The solution in this case is to use more than one computer to make a Web server. This leads to the **server farm** model of Fig. 7-43.

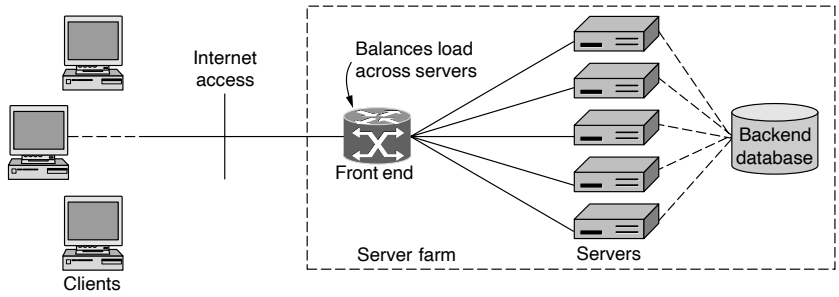

Figure 7-43. A server farm.

The difficulty with this seemingly simple model is that the set of computers that make up the server farm must look like a single logical Web site to clients. If they do not, we have just set up different Web sites that run in parallel.

There are several possible solutions to make the set of servers appear to be one Web site. All of the solutions assume that any of the servers can handle a request from any client. To do this, each server must have a copy of the Web site. The servers are shown as connected to a common back-end database by a dashed line for this purpose.

Perhaps the most common solution is to use DNS to spread the requests across the servers in the server farm. When a DNS request is made for the DNS domain in the corresponding Web URL, the DNS server returns a DNS response that redirects the client to a CDN service (typically by a NS-record referral to a name server that is authoritative for that domain), which in turn aims to return an IP address to the client that corresponds to a server replica that is close to the client. If multiple IP addresses are returned in the response, the client typically attempts to connect to the first IP address in the provided set of responses. The effect is that different clients contact different servers to access the same Web site, just as intended, hopefully one that is close to the client. Note that this process, which is sometimes referred to as **client mapping**, relies on the authoritative name server to know the topological or geographic location for the client. We will discuss DNS-based client mapping in more detail when we describe CDNs.

Another popular approach for load balancing today is to use **IP anycast**. Briefly, IP anycast is the process by which a single IP address can be advertised from multiple different network attachment points (e.g., a network in Europe and a network in the United States). If all goes well, a client that seeks to contact a particular IP address would end up having its traffic routed to the closest network endpoint. Of course, as we know, interdomain routing on the Internet doesn't always pick the shortest (or even the best) path, and so this method is far more coarse-grained and difficult to control than DNS-based client mapping. Nevertheless,

some large CDNs such as Cloudflare use IP anycast in conjunction with DNS-based client mapping.

Other less common solutions rely on a **front end** that distributes incoming requests over the pool of servers in the server farm. This happens even when the client contacts the server farm using a single destination IP address. The front end is usually a link-layer switch or an IP router, that is, a device that handles frames or packets. All of the solutions are based on it (or the servers) peeking at the network, transport, or application layer headers and using them in nonstandard ways. A Web request and response are carried as a TCP connection. To work correctly, the front end must distribute all of the packets for a request to the same server.

A simple design is for the front end to broadcast all of the incoming requests to all of the servers. Each server answers only a fraction of the requests by prior agreement. For example, 16 servers might look at the source IP address and reply to the request only if the last 4 bits of the source IP address match their configured selectors. Other packets are discarded. While this is wasteful of incoming bandwidth, often the responses are much longer than the request, so it is not nearly as inefficient as it sounds.

In a more general design, the front end may inspect the IP, TCP, and HTTP headers of packets and arbitrarily map them to a server. The mapping is called a **load balancing** policy as the goal is to balance the workload across the servers. The policy may be simple or complex. A simple policy might be to use the servers one after the other in turn, or round-robin. With this approach, the front end must remember the mapping for each request so that subsequent packets that are part of the same request will be sent to the same server. Also, to make the site more reliable than a single server, the front end should notice when servers have failed and stop sending them requests.

Web Proxies

Caching improves performance by shortening the response time and reducing the network load. If the browser can determine that a cached page is fresh by itself, the page can be fetched from the cache immediately, with no network traffic at all. However, even if the browser must ask the server for confirmation that the page is still fresh, the response time is shortened and the network load is reduced, especially for large pages, since only a small message needs to be sent.

However, the best the browser can do is to cache all of the Web pages that the user has previously visited. From our discussion of popularity, you may recall that as well as a few popular pages that many people visit repeatedly, there are many, many unpopular pages. In practice, this limits the effectiveness of browser caching because there are a large number of pages that are visited just once by a given user. These pages always have to be fetched from the server.

One strategy to make caches more effective is to share the cache among multiple users. That way, a page already fetched for one user can be returned to another

user when that user requests the same page again. Without browser caching, both users would need to fetch the page from the server. Of course, this sharing cannot be done for encrypted traffic, pages that require authentication, and uncacheable pages (e.g., current stock prices) that are returned by programs. Dynamic pages created by programs, especially, are a growing case for which caching is not effective. Nonetheless, there are plenty of Web pages that are visible to many users and look the same no matter which user makes the request (e.g., images).

A **Web proxy** is used to share a cache among users. A proxy is an agent that acts on behalf of someone else, such as the user. There are many kinds of proxies. For instance, an ARP proxy replies to ARP requests on behalf of a user who is elsewhere (and cannot reply for himself). A Web proxy fetches Web requests on behalf of its users. It normally provides caching of the Web responses, and since it is shared across users it has a substantially larger cache than a browser.

When a proxy is used, the typical setup is for an organization to operate one Web proxy for all of its users. The organization might be a company or an ISP. Both stand to benefit by speeding up Web requests for its users and reducing its bandwidth needs. While flat pricing, independent of usage, is common for home users, most companies and ISPs are charged according to the bandwidth that they use.

This setup is shown in Fig. 7-44. To use the proxy, each browser is configured to make page requests to the proxy instead of to the page's real server. If the proxy has the page, it returns the page immediately. If not, it fetches the page from the server, adds it to the cache for future use, and returns it to the client that requested it.

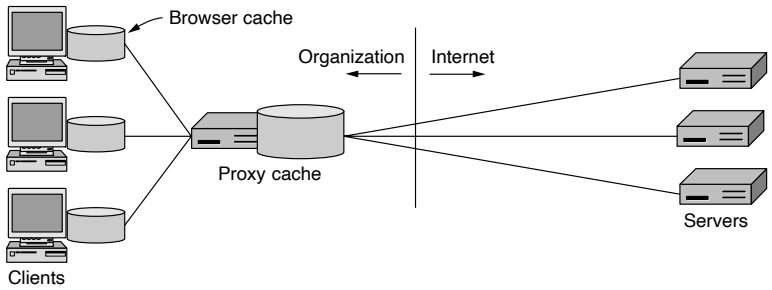

Figure 7-44. A proxy cache between Web browsers and Web servers.

As well as sending Web requests to the proxy instead of the real server, clients perform their own caching using its browser cache. The proxy is only consulted after the a browser has tried to satisfy the request from its own cache. That is, the proxy provides a second level of caching.

Further proxies may be added to provide additional levels of caching. Each proxy (or browser) makes requests via its **upstream proxy**. Each upstream proxy

caches for the **downstream proxies** (or browsers). Thus, it is possible for browsers in a company to use a company proxy, which uses an ISP proxy, which contacts Web servers directly. However, the single level of proxy caching we have shown in Fig. 7-44 is often sufficient to gain most of the potential benefits, in practice. The problem again is the long tail of popularity. Studies of Web traffic have shown that shared caching is especially beneficial until the number of users reaches about the size of a smallish company (say, 100 people). As the number of people grows larger, the benefits of sharing a cache become marginal because of the unpopular requests that cannot be cached due to lack of storage space.

Web proxies provide additional benefits that are often a factor in the decision to deploy them. One benefit is to filter content. The administrator may configure the proxy to blacklist sites or otherwise filter the requests that it makes. For example, many administrators frown on employees watching YouTube videos (or worse yet, pornography) on company time and set their filters accordingly. Another benefit of having proxies is privacy or anonymity, when the proxy shields the identity of the user from the server.

7.5.3 Content Delivery Networks

Server farms and Web proxies help to build large sites and to improve Web performance, but they are not sufficient for truly popular Web sites that must serve content on a global scale. For these sites, a different approach is needed.

CDNs (**Content Delivery Networks**) turn the idea of traditional Web caching on its head. Instead, of having clients look for a copy of the requested page in a nearby cache, provider places a copy of the page in a set of nodes at different locations and directs the client to use a nearby node as the server.

The techniques for using DNS for content distribution were pioneered by Akamai starting in 1998, when the Web was groaning under the load of its early growth. Akamai was the first major CDN and soon became the industry leader. Probably even more clever than the idea of using DNS to connect clients to nearby nodes was the model and incentive structure of its business. Companies pay Akamai to deliver their content to clients, so that they have responsive Web sites that customers like to use. The CDN nodes must be placed at network locations with good connectivity, which initially meant inside ISP networks. In practice a CDN node consists of a standard 19-inch equipment rack containing a computer and a lot of disks, with an optical fiber coming out of it to connect to the ISP's internal LAN.

For the ISPs, there is a benefit to having a CDN node in their networks, namely that the CDN node cuts down the amount of upstream network bandwidth that they need (and must pay for). In addition, the CDN node reduces latency to the content the ISP's customers, Thus, the content provider, the ISP, and the customers all benefit and the CDN makes money. Since 1998, many companies, including Cloudflare, Limelight, Dyn, and others, have gotten into the business, so it is now a

competitive industry with multiple providers. As mentioned, many large content providers such as YouTube, Facebook, and Netflix operate their own CDNs.

The largest CDNs have hundreds of thousands of servers deployed in countries all over the world. This large capacity can also help Web sites defend against DDoS attacks. If an attacker manages to send hundreds or thousands of requests per second to a site that uses a CDN, there is a good chance that the CDN will be able to reply to them all. In this way, the attacked site will be able to survive the flood of requests. That is, the CDN can quickly scale up a site's serving capacity. Some CDNs even advertise their ability to handle large-scale DDoS attacks as a selling point to attract content providers.

The CDN nodes pictured in our example are normally clusters of machines. DNS redirection is done with two levels: one to map clients to the approximate network location, and another to spread the load over nodes in that location. Both reliability and performance are concerns. To be able to shift a client from one machine in a cluster to another, DNS replies at the second level are given with short TTLs so that the client will repeat the resolution after a short while. Finally, while we have concentrated on distributing static objects like images and videos, CDNs can also support dynamic page creation, streaming media, and more. CDNs are also commonly used to distribute video.

Populating CDN Cache Nodes

An example of the path that data follows when it is distributed by a CDN is shown in Fig. 7-45. It is a tree. The origin server in the CDN distributes a copy of the content to other nodes in the CDN, in Sydney, Boston, and Amsterdam, in this example. This is shown with dashed lines. Clients then fetch pages from the ''nearest'' node in the CDN. This is shown with solid lines. In this way, the clients in Sydney both fetch the page copy that is stored in Sydney; they do not both fetch the page from the origin server, which may be in Europe.

Using a tree structure has three advantages. First, the content distribution can be scaled up to as many clients as needed by using more nodes in the CDN, and more levels in the tree when the distribution among CDN nodes becomes the bottleneck. No matter how many clients there are, the tree structure is efficient. The origin server is not overloaded because it talks to the many clients via the tree of CDN nodes; it does not have to answer each request for a page by itself. Second, each client gets good performance by fetching pages from a nearby server instead of a distant server. This is because the round-trip time for setting up a connection is shorter, TCP slow-start ramps up more quickly because of the shorter round-trip time, and the shorter network path is less likely to pass through regions of congestion in the Internet. Finally, the total load that is placed on the network is also kept at a minimum. If the CDN nodes are well placed, the traffic for a given page should pass over each part of the network only once. This is important because someone pays for network bandwidth, eventually.

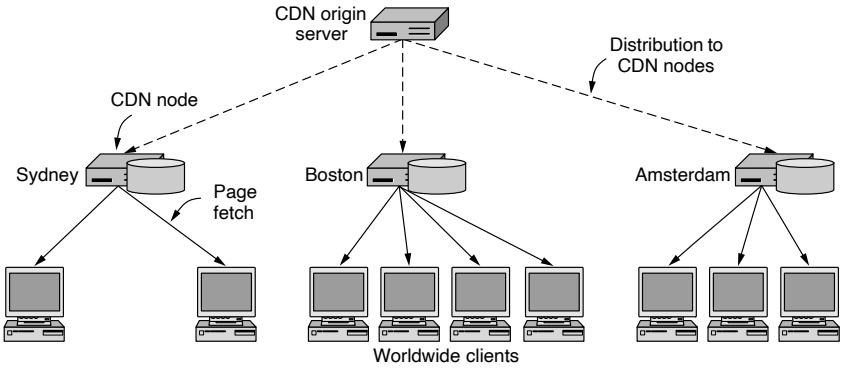

Figure 7-45. CDN distribution tree.

With the growth of encryption on the Web, and particularly with the rise of HTTPS for distributing Web content, serving content from CDNs has become more complex. Suppose, for example, that you wanted to retrieve *https://nytimes.com/*. A DNS lookup for this domain might give you a referral to a name server at Dyn, such as *ns1.p24.dynect.net*, which would in turn redirect you to an IP address hosted on the Dyn CDN. But, now that server has to deliver content to you that is authenticated by the *New York Times*. To do so, it might need the secret keys for the *New York Times*, or the ability to serve a certificate for *nytimes.com* (or both). As a result, the CDN would need to be trusted with sensitive information from the content provider, and the server has to be configured to effectively act as an agent of *nytimes.com*. An alternative is to direct all client requests back to the origin server, which could serve the HTTPS certificates and content, but doing so would negate essentially all of the performance benefits of a CDN. The typical solution typically involves somewhat of a middle ground, where the CDN generates a certificate on behalf of the content provider and serves the content from the CDN using that certificate, acting as the organization. This achieves the most commonly desired goal of encrypting the content between the CDN and the user, and authenticating the content for the user. More complex options, which require deploying certificates at the origin server, can allow content to also be encrypted between the origin and the cache nodes. Cloudflare has a good summary of these options on its website at *https://cloudflare.com/ssl/*.

DNS Redirection and Client Mapping

The idea of using a distribution tree is straightforward. What is less simple is how to map clients to the appropriate cache node in this tree. For example, proxy servers would seem to provide a solution. Looking at Fig. 7-45, if each client was

configured to use the Sydney, Boston, or Amsterdam CDN node as a caching Web proxy, the distribution would follow the tree.

The most common way to map or direct clients to nearby CDN cache nodes, as we briefly discussed earlier, is using **DNS redirection**. We now describe the approach in detail. Suppose that a client wants to fetch a page with the URL *https://www.cdn.com/page.html*. To fetch the page, the browser will use DNS to resolve *www.cdn.com* to an IP address. This DNS lookup proceeds in the usual manner. By using the DNS protocol, the browser learns the IP address of the name server for *cdn.com*, then contacts the name server to ask it to resolve *www.cdn.com*. At this point, however, because the name server is run by the CDN. instead of returning the same IP address for each request, it will look at the IP address of the client making the request and return different answers depending on where the client is located. The answer will be the IP address of the CDN node that is nearest to the client. That is, if a client in Sydney asks the CDN name server to resolve *www.cdn.com*, the name server will return the IP address of the Sydney CDN node, but if a client in Amsterdam makes the same request, the name server will return the IP address of the Amsterdam CDN node instead.

This strategy is perfectly appropriate, according to the semantics of DNS. We have previously seen that name servers may return changing lists of IP addresses. After the name resolution, the Sydney client will fetch the page directly from the Sydney CDN node. Further pages on the same ''server'' will be fetched directly from the Sydney CDN node as well because of DNS caching. The overall sequence of steps is shown in Fig. 7-46.

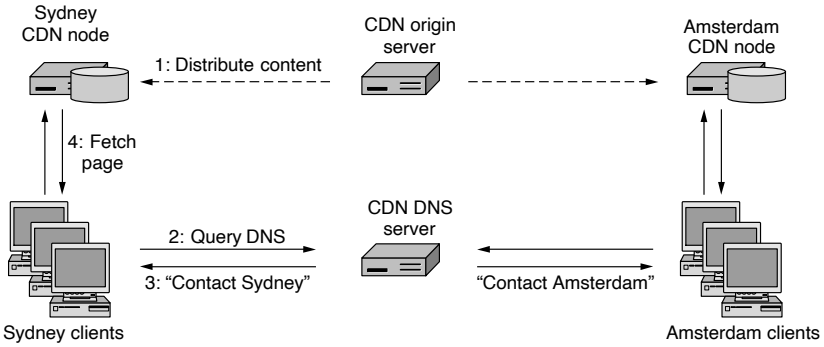

Figure 7-46. Directing clients to nearby CDN nodes using DNS.

A complex question in the above process is what it means to find the nearest CDN node, and how to go about it (this is the **client mapping** problem that we discussed earlier). There are at least two factors to consider in mapping a client to a CDN node. One factor is the network distance. The client should have a short and high-capacity network path to the CDN node. This situation will produce quick

downloads. CDNs use a map they have previously computed to translate between the IP address of a client and its network location. The CDN node that is selected might be the one with the shortest distance as the crow flies, or it might not. What matters is some combination of the length of the network path and any capacity limits along it.

The second factor is the load that is already being carried by the CDN node. If the CDN nodes are overloaded, they will deliver slow responses, just like the overloaded Web server that we sought to avoid in the first place. Thus, it may be necessary to balance the load across the CDN nodes, mapping some clients to nodes that are slightly further away but more lightly loaded.

The ability of a CDN's authoritative DNS server to map a client to a nearby CDN cache node depends on the ability to determine the client's location. As previously discussed in the DNS section, modern extensions to DNS, such as EDNS0 Client Subnet make it possible for the authoritative name server to see the client's IP address. The potential move to DNS-over-HTTPS also may introduce new challenges, given that the IP address of the local recursive resolver may be nowhere near the client; if the DNS local recursive does not pass on the IP address of the client (as is typically the case, given that the whole purpose is to preserve the privacy of the client), then CDNs who do not also resolve DNS for their clients are likely to face greater difficulties in performing client mapping. On the other hand, CDNs who also operate a DoH resolver (as Cloudflare and Google now do) may reap significant benefits, as they will have direct knowledge of the client IP addresses that are issuing DNS queries, often for content on their own CDNs! The centralization of DNS is indeed poised to reshape content distribution once again over the coming few years.

This section presented a simplified description of how CDNs work. There are many more details that matter in practice. For example, the CDN nodes' disks will eventually fill up so they have to purged regularly. Much work has been done on determining on which files to discard and when, for example Basu et al. (2018).

7.5.4 Peer-to-Peer Networks

Not everyone can set up a 1000-node CDN at locations around the world to distribute their content. (Actually, it is not hard to rent 1000 virtual machines around the globe because of the well-developed and competitive hosting industry. However, setting up a CDN only starts with getting the nodes.) Luckily, there is an alternative for the rest of us that is simple to use and can distribute a tremendous amount of content. It is a P2P (Peer-to-Peer) network.

P2P networks burst onto the scene starting in 1999. The first widespread application was for mass crime: 50 million Napster users were exchanging copyrighted songs without the copyright owners' permission until Napster was shut down by the courts amid great controversy. Nevertheless, peer-to-peer technology has many interesting and legal uses. Other systems continued development, with

such great interest from users that P2P traffic quickly eclipsed Web traffic. Today, BitTorrent remains the most popular P2P protocol. It is used so widely to share (licensed and public domain) videos, as well as other large content (e.g., operating system disk images), that it still accounts for a significant fraction of all Internet traffic, despite the growth of video. We will look at it later in this section.

Overview

The basic idea of a **P2P** (**Peer-to-Peer**) file-sharing network is that many computers come together and pool their resources to form a content distribution system. The computers are often simply home computers. They do not need to be machines in Internet data centers. The computers are called **peers** because each one can alternately act as a client to another peer, fetching its content, and as a server, providing content to other peers. What makes peer-to-peer systems interesting is that there is no dedicated infrastructure, unlike in a CDN. Everyone participates in the task of distributing content, and there is often no central point of control. Many use cases exist (Karagiannis et al., 2019).

Many people are excited about P2P technology because it is seen as empowering the little guy. The reason is not only that it takes a large company to run a CDN, while anyone with a computer can join a P2P network. It is that P2P networks have a formidable capacity to distribute content that can match the largest of Web sites.

Early Peer-to-Peer Networks: Napster

As previously discussed, early peer-to-peer networks such as Napster were based on a centralized directory service. Users installed client software that scaned their local storage for files to share and, after inspecting the contents, uploaded metadata information about the shared files (e.g., file names, sizes, identity of the user sharing the content) to a centralized directory service. Users who wished to retrieve files from the Napster network would subsequently search the centralized directory server and could learn about other users who had that file. The server would inform the user searching for content about the IP address of a peer that was sharing the file that the user was looking for, at which point the user's client software could contact that host directly and download the file in question.

A side-effect of Napster's centralized directory server was that it made it relatively easy for others to search the network and exhaustively determine who was sharing which files, effectively crawling the entire network. It became clear at some point that a significant fraction of all content on Napster was copyrighted material, which ultimately resulted in injunctions that shut the service down. Another side-effect of the centralized directory service that became clear was that to disable the service, one needed only to disable the directory server. Without it, Napster became effectively unusable. In response, designers of new peer-to-peer

networks began to design systems that could be more robust to shutdown or failure. The general approach to doing so was to decentralize the directory or search process. Next-generation peer-to-peer systems, such as Gnutella, took this approach.

Decentralizing the Directory: Gnutella

Gnutella was released in 2000; it attempted to solve some of the problems that a centralized directory service that Napster suffered from, effectively by implementing a fully distributed search function. In Gnutella, a peer that joined the network would attempt to discover other connected peers through an ad hoc discovery process; the peer would start by contacting a few well-known Gnutella peers which it had to discover through some bootstrapping process. One way of doing so was to ship some set of IP addresses of Gnutella peers with the software itself. Upon discovering a set of peers, the Gnutella peer could then issue search queries to these neighboring peers, who would then pass the query on to their neighbors, and so forth. This general approach to searching a peer-to-peer network is often referred to as **gossip**.

Although the gossip approach solved some of the problems faced by semi-centralized services such as Napster, it quickly faced other problems. One problem is that in the Gnutella network, peers were continually joining and leaving the network; peers were simply other users' computers, and thus they were continually entering and leaving the network. In particular, users had no particular reason to stay on the network after retrieving the files that they were interested in, and thus so called **free-riding** behavior was common, with 70% of the users contributing no content (Adar and Huberman, 2000). Second, the flooding-based Specifically, the gossip approach scaled very poorly, particularly as Gnutella became popular. Specifically, the number of gossip messages grew exponentially with the number of participants in the network. The protocol thus scaled particularly poorly. Users with limited network capacity basically found the network completely unusable. Gnutella's introduction of so-called **ultra-peers** mitigated these scalability challenges somewhat, but in general Gnutella was fairly wasteful of available network resources. The lack of scalability in Gnutella's lookup process inspired the invention of **DHTs** (**Distributed Hash Tables**) whereby a lookup is routed to the appropriate a peer-to-peer network based on the corresponding hash value of the lookup; each node in the peer-to-peer network is responsible only for maintaining information about some subset of the overall lookup space, and the DHT is responsible for routing the query to the appropriate peer that can resolve the lookup. DHTs are used in many modern peer-to-peer networks, including eDonkey (which uses a DHT for lookup) and BitTorrent (which uses a DHT to scale the tracking of peers in the network, as we describe in the next section).

Finally, Gnutella did not automatically verify file contents that users were downloading, resulting in a significant amount of bogus content on the network. Why would a peer-to-peer network have so much fake content, you might wonder.

There are many possible reasons. One simple reason is that, just as any Internet service might be subject to a denial-of-service attack, Gnutella itself also became a target, and one of the easiest ways to launch a denial of service attack on the network was to mount so-called **pollution attacks**, which flooded the network with fake content. One group that was particularly interested in rendering these networks useless was the recording industry (notably the Recording Industry Association of America), who was found to be polluting peer-to-peer networks such as Gnutella with large amounts of fake content to dissuade people from using the networks to exchange copyrighted content.

Thus, peer-to-peer networks were faced with a number of challenges: scaling, convincing users to stick around after downloading the content they were searching for, and verifying the content they downloaded. BitTorrent's design addressed all three challenges, as we discuss next.

Coping with Scaling, Incentives, and Verification: BitTorrent

The BitTorrent protocol was developed by Bram Cohen in 2001 to let a set of peers share files quickly and easily. There are dozens of freely available clients that speak this protocol, just as there are many browsers that speak the HTTP protocol to Web servers. The protocol is available as an open standard at *bittorrent.org*.

In a typical peer-to-peer system, like that formed with BitTorrent, the users each have some information that may be of interest to other users. This information may be free software, music, videos, photographs, and so on. There are three problems that need to be solved to share content in this setting:

1. How does a peer find other peers that have the content it wants to download?
2. How is content replicated by peers to provide high-speed downloads for everyone?
3. How do peers encourage each other to upload content to others as well as download content for themselves?

The first problem exists because not all peers will have all of the content. The approach taken in BitTorrent is for every content provider to create a content description called a **torrent**. The torrent is much smaller than the content, and is used by a peer to verify the integrity of the data that it downloads from other peers. Other users who want to download the content must first obtain the torrent, say, by finding it on a Web page advertising the content.

The torrent is just a file in a specified format that contains two key kinds of information. One kind is the name of a tracker, which is a server that leads peers to the content of the torrent. The other kind of information is a list of equal-sized

pieces, or **chunks**, that make up the content. In early versions of BitTorrent, the tracker was a centralized server; as with Napster, centralizing the tracker resulted in a single point of failure for a BitTorrent network. As a result, modern versions of BitTorrent commonly decentralize the tracker functionality using a DHT. Different chunk sizes can be used for different torrents; they typically range from 64 KB to 512 KB. The torrent file contains the name of each chunk, given as a 160-bit SHA-1 hash of the chunk. We will cover cryptographic hashes such as SHA-1 in Chap. 8. For now, you can think of a hash as a longer and more secure checksum. Given the size of chunks and hashes, the torrent file is at least three orders of magnitude smaller than the content, so it can be transferred quickly.

To download the content described in a torrent, a peer first contacts the tracker for the torrent. The **tracker** is a server (or group of servers, organized by a DHT) that maintains a list of all the other peers that are actively downloading and uploading the content. This set of peers is called a **swarm**. The members of the swarm contact the tracker regularly to report that they are still active, as well as when they leave the swarm. When a new peer contacts the tracker to join the swarm, the tracker tells it about other peers in the swarm. Getting the torrent and contacting the tracker are the first two steps for downloading content, as shown in Fig. 7-47.

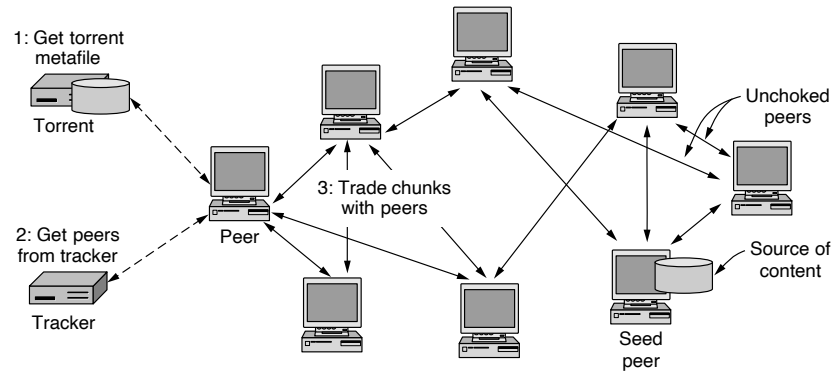

Figure 7-47. BitTorrent.

The second problem is how to share content in a way that gives rapid downloads. When a swarm is first formed, some peers must have all of the chunks that make up the content. These peers are called **seeders**. Other peers that join the swarm will have no chunks; they are the peers that are downloading the content.

While a peer participates in a swarm, it simultaneously downloads chunks that it is missing from other peers, and uploads chunks that it has to other peers who need them. This trading is shown as the last step of content distribution in Fig. 7-47. Over time, the peer gathers more chunks until it has downloaded all of the content. The peer can leave the swarm (and return) at any time. Normally a

peer will stay for a short period after finishes its own download. With peers coming and going, the rate of churn in a swarm can be quite high.

For the above method to work well, each chunk should be available at many peers. If everyone were to get the chunks in the same order, it is likely that many peers would depend on the seeders for the next chunk. This would create a bottleneck. Instead, peers exchange lists of the chunks they have with each other. Then they preferentially select rare chunks that are hard to find to download. The idea is that downloading a rare chunk will result in the creation of another copy of it, which will make the chunk easier for other peers to find and download. If all peers do this, after a short while all chunks will be widely available.

The third problem involves incentives. CDN nodes are set up exclusively to provide content to users. P2P nodes are not. They are users' computers, and the users may be more interested in getting a movie than helping other users with their downloads; in other words, there can sometimes be incentives for users to cheat the system. Nodes that take resources from a system without contributing in kind are called **free-riders** or **leechers**. If there are too many of them, the system will not function well. Earlier P2P systems were known to host them (Saroiu et al., 2003) so BitTorrent sought to minimize them.

BitTorrent attempts to address this problem by rewarding peers who show good upload behavior. Each peer randomly samples the other peers, retrieving chunks from them while it uploads chunks to them. The peer continues to trade chunks with only a small number of peers that provide the highest download performance, while also randomly trying other peers to find good partners. Randomly trying peers also allows newcomers to obtain initial chunks that they can trade with other peers. The peers with which a node is currently exchanging chunks are said to be **unchoked**.

Over time, this algorithm aims to match peers with comparable upload and download rates with each other. The more a peer is contributing to the other peers, the more it can expect in return. Using a set of peers also helps to saturate a peer's download bandwidth for high performance. Conversely, if a peer is not uploading chunks to other peers, or is doing so very slowly, it will be cut off, or **choked**, sooner or later. This strategy discourages adversarial behavior in which peers freeride on the swarm.

The choking algorithm is sometimes described as implementing the **tit-for-tat** strategy that encourages cooperation in repeated interactions; the theory behind the incentives for cooperation are rooted in the famous tit-for-tat game in game theory, whereby players have incentives to cheat unless (1) they repeatedly play the game with each other (as is the case in BitTorrent, where peers must repeatedly swap chunks) and (2) peers are punished for not cooperating (as is the case with choking). Despite this design, in actual practice BitTorrent does not prevent clients from gaming the system in various ways (Piatek et al., 2007). For example, BitTorrent's algorithm whereby a client favors selecting rare pieces can create incentives for a peer to lie about which chunks of the file it has (e.g., claiming that it has

rare pieces when it does not) (Liogkas et al., 2006). Software also exists whereby clients can lie to the tracker about its ratio of upload to download, effectively saying that it performed uploads that it did not perform. For these reasons, it is critical for a peer to verify each chunk that they download from other peers. It can do so by comparing the SHA-1 hash value of each chunk that is present in the torrent file against the corresponding SHA-1 hash value that they can compute for each corresponding chunk that it downloads.

Another challenge involves creating incentives for peers to stay around in the BitTorrent swarm as seeders, even after they have completed downloading the entire file. If they do not, then the possibility exists that nobody in the swarm has the entire file, and (worse), that a swarm may collectively be missing pieces of the entire file, thus making it impossible for anyone to download the complete file. This problem is particularly acute for files that are less popular (Menasche et al., 2013). Various approaches have been developed to address these incentive issues (Ramachandran et al., 2007).

As you can see from our discussion, BitTorrent comes with a rich vocabulary. There are torrents, swarms, leechers, seeders, and trackers, as well as snubbing, choking, lurking, and more. For more information see the short paper on BitTorrent (Cohen, 2003).

7.5.5 Evolution of the Internet

As we described in Chap. 1, the Internet has had a strange history, starting as an academic research project for a few dozen American universities with an ARPA contract. It is even hard to define the moment it began. Was that Nov. 21, 1969, when two ARPANET nodes, UCLA and SRI, were connected? Was it on Dec. 17, 1972 when the Hawaiian AlohaNet connected to the ARPANET to form an internetwork? Was it Jan. 1, 1983, when ARPA officially adopted TCP/IP as the protocol? Was it in 1989, when Tim Berners-Lee proposed what is now the World Wide Web? It is hard to say. What is easy to say, however, is that a huge amount has changed since the early days of the ARPANET and fledgling Internet, much of it do the widespread adoption of CDNs and cloud computing. Below we will take a quick look.

The fundamental model behind the ARPANET and the early Internet is shown in Fig. 7-48. It consists of three components:

1. Hosts (the computers that did the work for the users).
2. Routers (called IMPs in the ARPANET) that switched the packets.
3. Transmission lines (originally 56-kbps leased telephone lines).

Each router was connected to one or more computers.

The conceptual model of the early Internet architecture was dominated by the basic idea of point-to-point communications. The host computers were all seen as

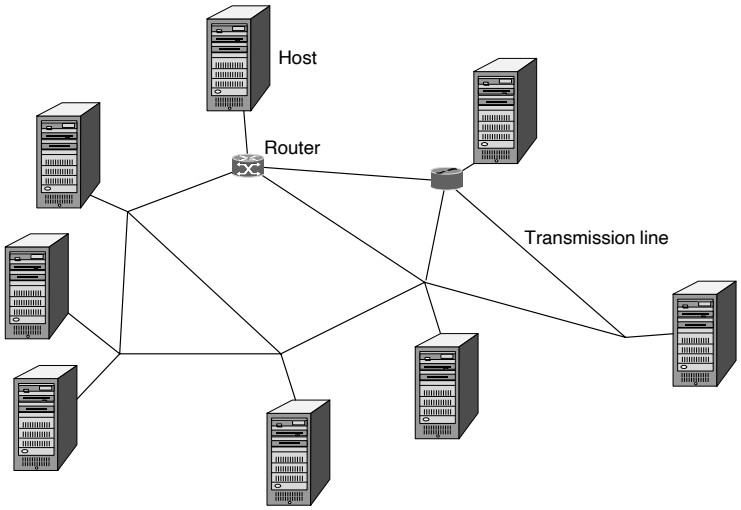

Figure 7-48. The early Internet involved primarily point-to-point communications

equals (although some were much more powerful than others) and any computer could send packets to any other computer since every computer had a unique address. With the introduction of TCP/IP these were all 32 bits, which at the time seemed like an excellent approximation to infinity. Now it seems closer to zero than to infinity. The transmission model was that of a simple stateless, datagram system, with each packet containing its destination address. Once a packet passed through a router, it was completely forgotten. Routing was done hop by hop. Each packet was routed based on its destination address and information in the router's tables about which transmission line to use for the packet's destination.

Things began to change when the Internet surged past its academic beginnings and went commercial. That led to the development of the backbone networks, which used very high-speed links and were operated by large telecom companies like AT&T and Verizon. Each company ran its own backbone, but the companies connected to each other at peering exchanges. Internet service providers sprung up to connect homes and businesses to the Internet and regional networks connected the ISPs to the backbones. This situation is shown in Fig. 1-17. The next step was the introduction of national ISPs and CDNs, as shown in Fig. 1-18.

Cloud computing and very large CDNs have again disrupted the structure of the Internet, much as we described in Chap. 1. Modern cloud data centers, like those run by Amazon and Microsoft, have hundreds of thousands of computers in the same building, allowing users (typically large companies) to allocate 100 or 1000 or 10,000 machines within seconds. When Walmart has a big sale on Cyber

Monday (the Monday after Thanksgiving), if it needs 10,000 machines to handle the load, it just requests them automatically from its cloud provider as needed and they will be available within seconds. On Back-to-Normal Tuesday, it can give them all back. Almost all large companies that deal with millions of consumers use cloud services to be able to expand or contract their computing capacity almost instantaneously, as needed. As a side benefit, as mentioned above, clouds also provide fairly good protection against DDoS attacks because the cloud is so big that it can absorb thousands of requests/sec, answer them all, and keep on functioning, thus defeating the intent of the DDoS attack.

CDNs are hierarchical, with a master site (possibly replicated two or three times for reliability) and many caches all over the world to which content is pushed. When a user requests content, it is served from the closest cache. This reduces latency and spreads the workload. Akamai, the first large commercial CDN, has over 200,000 cache nodes in more than 1500 networks in more than 120 countries. Similarly, Cloudflare now has cache nodes in more than 90 countries. In many cases, CDN cache nodes are co-located with ISP offices, so data can travel from the CDN to the ISP over a very fast piece of optical fiber perhaps only 5 meters long. This new world has led to the Internet architecture shown in Fig. 7-49, where the vast majority of Internet traffic is carried between access (e.g., regional) networks and distributed cloud infrastructure (i.e., either CDNs or cloud services).

Users send requests to large servers to do something and the server does it and creates a Web page showing what it did. Examples of requests are:

1. Buy a product at an e-commerce store.
2. Fetch an email message from an email provider.
3. Issue a payment order to a bank.
4. Request a song or movie to be streamed to a user's device.
5. Update a Facebook page.
6. Ask an online newspaper to display an article.

Nearly all Internet traffic today now follows this model. The proliferation of cloud services and CDNs have upended the conventional client-server model of Internet traffic, whereby a client would retrieve or exchange content with a single server. Today, the vast majority of content and communications operates on distributed cloud services; many access ISPs for example send the majority of their traffic to distributed cloud services. In most developed regions, there is simply no need for users to access massive amounts of content over long-haul transit infrastructure: CDNs have by and large placed much of that popular content close to the user, often geographically nearby and across a direct network interconnect to their access ISP. Thus an increasing amount of content is delivered via CDNs that are

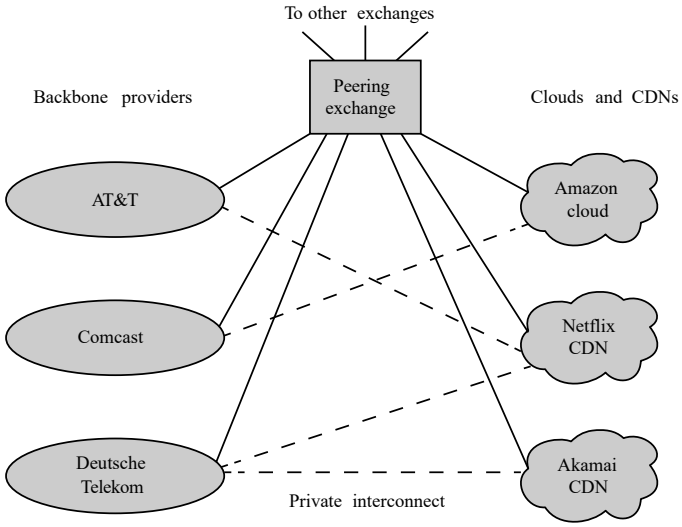

Figure 7-49. Most Internet traffic today is from clouds and CDNs, with a significant amount of traffic being exchanged between access networks and ISPs over private interconnects.

hosted either directly over private interconnects to access networks, or even on CDNs, where cache nodes are located within the access network itself.

Backbone networks allow the many clouds and CDNs to interconnect via peering exchanges for those cases where there is no private dedicated interconnection. The DE-CIX exchange in Frankfurt connects about 2000 networks. The AMS-IX exchange in Amsterdam and the LINX exchange in London each connect about 1,000 networks. The larger exchanges in the United States each connect hundreds of networks. These exchanges are themselves interconnected with one or more OC-192 and/or OC-768 fiber links running at 9.6 and 38.5 Gbps, respectively. The peering exchanges and the larger carrier networks that meet at them form the Internet backbone to which most clouds and CDNs directly connect.

Content and cloud providers are increasingly connecting directly to access ISPs over private interconnects to put the content closer to the users; in some cases, they even place the content on servers directly in the access ISP network. One example of this is Akamai, which has over 200,000 servers, most inside ISP networks, as mentioned above. This trend will continue to reshape the Internet in years to come. Other CDNs, such as Cloudflare, are also becoming increasingly pervasive. Finally, providers of content and services are themselves deploying CDNs; Netflix has deployed its own CDN called Open Connect, for example, where Netflix content is deployed on cache nodes either at IXPs or directly inside

an access ISP network. The extent to which Internet paths traverse a separate backbone network or **IXP** (**Internet Exchange Point**) depends on a variety of factors, including cost, available connectivity in the region, and economies of scale. IXPs are extremely popular in Europe and other parts of the world; in contrast, in the United States, direct connection over private interconnects tend to be more popular and prevalent.

7.6 SUMMARY

Naming in the ARPANET started out in a very simple way: an ASCII text file listed the names of all the hosts and their corresponding IP addresses. Every night all the machines downloaded this file. But when the ARPANET morphed into the Internet and exploded in size, a far more sophisticated and dynamic naming scheme was required. The one used now is a hierarchical approach called the Domain Name System. It organizes all the machines on the Internet into a set of trees. At the top level are the well-known generic domains, including *com* and *edu*, as well as about 200 country domains. DNS is implemented as a distributed database with servers all over the world. By querying a DNS server, a process can map an Internet domain name onto the IP address used to communicate with a computer for that domain. DNS is used for a variety of purposes; recent developments have created privacy concerns around DNS, resulting in a move to encrypt DNS with TLS or HTTPS. The resulting potential centralization of DNS is poised to change fundamental aspects of the Internet architecture.

Email is the original killer app of the Internet. It is still widely used by everyone from small children to grandparents. Most email systems in the world use the mail system now defined in RFC 5321 and RFC 5322. Messages have simple ASCII headers, and many kinds of content can be sent using MIME. Mail is submitted to message transfer agents for delivery and retrieved from them for presentation by a variety of user agents, including Web applications. Submitted mail is delivered using SMTP, which works by making a TCP connection from the sending message transfer agent to the receiving one.

The Web is the application that most people think of as being the Internet. Originally, it was a system for seamlessly linking hypertext pages (written in HTML) across machines. The pages are downloaded by making a TCP connection from the browser to a server and using HTTP. Nowadays, much of the content on the Web is produced dynamically, either at the server (e.g., with PHP) or in the browser (e.g., with JavaScript). When combined with back-end databases, dynamic server pages allow Web applications such as e-commerce and search. Dynamic browser pages are evolving into full-featured applications, such as email, that run inside the browser and use the Web protocols to communicate with remote servers. With the growth of the advertising industry, tracking on the Web has become very pervasive, through a variety of techniques, from cookies to canvas fingerprinting.

While there are ways to block certain types of tracking mechanisms such as cookies, doing so can sometimes hamper the functionality of a Web site, and some tracking mechanisms (e.g., canvas fingerprinting) are incredibly difficult to block.

Digital audio and video have been key drivers for the Internet since 2000. The majority of Internet traffic today is video. Much of it is streamed from Web sites over a mix of protocols although TCP is also very widely used. Live media is streamed to many consumers. It includes Internet radio and TV stations that broadcast all manner of events. Audio and video are also used for real-time conferencing. Many calls use voice over IP, rather than the traditional telephone network, and include videoconferencing.

There are a small number of tremendously popular Web sites, as well as a very large number of less popular ones. To serve the popular sites, content distribution networks have been deployed. CDNs use DNS to direct clients to a nearby server; the servers are placed in data centers all around the world. Alternatively, P2P networks let a collection of machines share content such as movies among themselves. They provide a content distribution capacity that scales with the number of machines in the P2P network and which can rival the largest of sites.

PROBLEMS

1. Many business computers have three distinct and worldwide unique identifiers. What are they?

2. In Fig. 7-5, there is no period after *laserjet*. Why not?

3. Give an example, similar to the one shown in Fig. 7-6, of a resolver looking up the domain name *courses.cs.vu.nl* in two steps. In which scenario would this happen in practice?

4. Which DNS record verifies the key that is used to sign the DNS records for an authoritative name server?

5. Which DNS record verifies the signature of the DNS records for an authoritative name server?

6. Describe the process of client mapping, by which some part of the DNS infrastructure would identify a content server that is close to the client that issued the DNS query. Explain any assumptions involved in determining the location of the client.

7. Consider a situation in which a cyberterrorist makes all the DNS servers in the world crash simultaneously. How does this change one's ability to use the Internet?

8. Explain the advantages and disadvantages of using TCP instead of UDP for DNS queries and responses.

9. Assuming that caching behavior for DNS lookups is as normal and DNS is not encryp-

ted, which of the following parties can see all of your DNS lookups from your local device? If DNS is encrypted with DoH or DoT, who can see the DNS lookups?

10. Nathan wants to have an original domain name and uses a randomized program to generate a secondary domain name for him. He wants to register this domain name in the *com* generic domain. The domain name that was generated is 253 characters long. Will the *com* registrar allow this domain name to be registered?

11. Can a machine with a single DNS name have multiple IP addresses? How could this occur?

12. The number of companies with a Web site has grown explosively in recent years. As a result, thousands of companies are registered in the *com* domain, causing a heavy load on the top-level server for this domain. Suggest a way to alleviate this problem without changing the naming scheme (i.e., without introducing new top-level domain names). It is permitted that your solution requires changes to the client code.

13. Some email systems support a *Content Return:* header field. It specifies whether the body of a message is to be returned in the event of nondelivery. Does this field belong to the envelope or to the header?

14. You receive a suspicious email, and suspect that it has been sent by a malicious party. The FROM field in the email says the email was sent by someone you trust. Can you trust the contents of the email? What more can you do to check its authenticity?

15. Electronic mail systems need directories so people's email addresses can be looked up. To build such directories, names should be broken up into standard components (e.g., first name, last name) to make searching possible. Discuss some problems that must be solved for a worldwide standard to be acceptable.

16. A large law firm, which has many employees, provides a single email address for each employee. Each employee's email address is *<login>@lawfirm.com*. However, the firm did not explicitly define the format of the login. Thus, some employees use their first names as their login names, some use their last names, some use their initials, etc. The firm now wishes to make a fixed format, for example:

firstname.lastname@lawfirm.com,

that can be used for the email addresses of all its employees. How can this be done without rocking the boat too much?

17. A binary file is 4560 bytes long. How long will it be if encoded using base64 encoding, with a CR+LF pair inserted after every 110 bytes sent and at the end?

18. A 100-byte ASCII string is encoded using base64. How long is the resulting string?

19. Your fellow student encodes the ASCII string ''ascii'' using base64, resulting in ''YXNjaWJ''. Show what went wrong during encoding, and give the correct encoding of the string.

20. You are building an instant messaging application for your computer networks lab assignment. The application must be able to transfer ASCII text and binary files. Unfortunately, another student on your team already handed in the server code without

implementing a feature for transferring binary files. Can you still implement this feature by only changing the client code?

21. In any standard, such as RFC 5322, a precise grammar of what is allowed is needed so that different implementations can interwork. Even simple items have to be defined carefully. The SMTP headers allow white space between the tokens. Give *two* plausible alternative definitions of white space between tokens.

22. Name five MIME types not listed in this book. You can check your browser or the Internet for information.

23. Suppose that you want to send an MP3 file to a friend, but your friend's ISP limits the size of each incoming message to 1 MB and the MP3 file is 4 MB. Is there a way to handle this situation by using RFC 5322 and MIME?

24. IMAP allows users to fetch and download email from a remote mailbox. Does this mean that the internal format of mailboxes has to be standardized so any IMAP program on the client side can read the mailbox on any mail server? Discuss your answer.

25. Although it was not mentioned in the text, an alternative form for a URL is to use the IP address instead of its DNS name. Use this information to explain why a DNS name cannot end with a digit.

26. Imagine that someone in the math department at Stanford has just written a new document including a proof that he wants to distribute by FTP for his colleagues to review. He puts the program in the FTP directory */ftp/pub/forReview/newProof.pdf*. What is the URL for this program likely to be?

27. Imagine a Web page that takes 3 sec. to load using HTTP with a persistent connection and sequential requests. Of these 3 seconds, 150 msec is spent setting up the connection and obtaining the first response. Loading the same page using pipelined requests takes 200 msec. Assume that sending a request is instantaneous, and that the time between the request and reply is equal for all requests. How many requests are performed when fetching this Web page?

28. You are building a networked application for your computer networks lab assignment. Another student on your team says that, because your system communicates via HTTP, which runs over TCP, your system does not need to take into account the possibility that communication between hosts breaks down. What do you say to your teammate?

29. For each of the following applications, tell whether it would be (1) possible and (2) better to use a PHP script or JavaScript, and why:
(a) Displaying a calendar for any requested month since September 1752.
(b) Displaying the schedule of flights from Amsterdam to New York.
(c) Graphing a polynomial from user-supplied coefficients.

30. The *If-Modified-Since* header can be used to check whether a cached page is still valid. Requests can be made for pages containing images, sound, video, and so on, as well as HTML. Do you think the effectiveness of this technique is better or worse for JPEG images as compared to HTML? Think carefully about what ''effectiveness'' means and explain your answer.

31. You request a Web page from a server. The server's reply includes an Expires header,

whose value is set to one day in the future. After five minutes, you request the same page from the same server. Can the server send you a newer version of the page? Explain why (not).

32. Does it make sense for a single ISP to function as a CDN? If so, how would that work? If not, what is wrong with the idea?

33. Audio CDs encode the music at 44,000 Hz with 16-bit samples. Would it make sense to produce higher-quality audio by sampling at 176,000 Hz with 16-bit samples? What about 44,000 Hz with 24-bit samples?

34. Assume that compression is not used for audio CDs. How many MB of data must the compact disc contain in order to be able to play 2hours of music?

35. Could a psychoacoustic model be used to reduce the bandwidth needed for Internet telephony? If so, what conditions, if any, would have to be met to make it work? If not, why not?

36. A server hosting a popular chat room sends data to its clients at a rate of 32 kbps. If this data arrives at the clients every 100 msec, what is the packet size used by the server? What is the packet size if the clients receive data every second?

37. An audio streaming server has a one-way ''distance'' of 100 msec to a media player. It outputs at 1 Mbps. If the media player has a 2-MB buffer, what can you say about the position of the low-water mark and the high-water mark?

38. You are streaming a five-minute video and receive 80 Mbps of encoded data per second, with a compression ratio of 200:1. The video has a resolution of 2000 × 1000 pixels, uses 20 bits per pixel, and is played at 60 frames per second. After 40 sec., your Internet connection breaks down. Can you watch the video to completion?

39. Suppose that a wireless transmission medium loses a lot of packets. How could uncompressed CD-quality audio be transmitted so that a lost packet resulted in a lower quality sound but no gap in the music?

40. In the text we discussed a buffering scheme for video that is shown in Fig. 7-34. Would this scheme also work for pure audio? Why or why not?

41. Real-time audio and video streaming has to be smooth. End-to-end delay and packet jitter are two factors that affect the user experience. Are they essentially the same thing? Under what circumstances does each one come into play? Can either one be combatted, and if so, how?

42. What is the bit rate for transmitting uncompressed 1200 × 800 pixel color frames with 16 bits/pixel at 50 frames/sec?

43. What is the compression ratio needed to send a 4K video over a 80 Mbps channel? Assume that the video plays at a rate of 60 frames per second, and every pixel value is stored in 3 bytes.

44. Suppose an DASH system with 50 frames/sec breaks a video up into 10-second segments, each with exactly 500 frames, Do you see any problems here? (*Hint:* think

about the kind of frames used in MPEG) If you see a problem, how could it be fixed?

45. Can a 1-bit error in an MPEG frame affect more than the frame in which the error occurs? Explain your answer.

46. Imagine that a video streaming service decides to use UDP instead of TCP. UDP packets can arrive in a different order than the one in which they were sent. What problem does this cause and how can it be solved? What complication does your solution introduce, if any?

47. While working at a game-streaming company, a colleague suggests creating a new transport-layer protocol that overcomes the shortcomings of TCP and UDP, and guarantees low latency and jitter for multimedia applications. Explain why this will not work.

48. Consider a 50,000-customer video server, where each customer watches three movies per month. Two-thirds of the movies are served at 9 P.M. How many movies does the server have to transmit at once during this time period? If each movie requires 6 Mbps, how many OC-12 connections does the server need to the network?

49. Suppose that Zipf's law holds for accesses to a 10,000-movie video server. If the server holds the most popular 1000 movies in memory and the remaining 9000 on disk, give an expression for the fraction of all references that will be to memory. Write a little program to evaluate this expression numerically.

50. A popular Web page hosts 2 billion videos. If the video popularity follows a Zipf distribution, what fraction of views goes to the top 10 videos?

51. One of the advantages of peer-to-peer systems is that there is often no central point of control, making these systems resilient to failures. Explain why BitTorrent is not fully decentralized.

52. Some cybersquatters have registered domain names that are misspellings of common corporate sites, for example, *www.microsfot.com*. Make a list of at least five such domains.

53. Numerous people have registered DNS names that consist of *www.word.com*, where *word* is a common word. For each of the following categories, list five such Web sites and briefly summarize what it is (e.g., *www.stomach.com* belongs to a gastroenterologist on Long Island). Here is the list of categories: animals, foods, household objects, and body parts. For the last category, please stick to body parts above the waist.

54. Explain some reasons why a BitTorrent client might cheat or lie, and how it might do so.