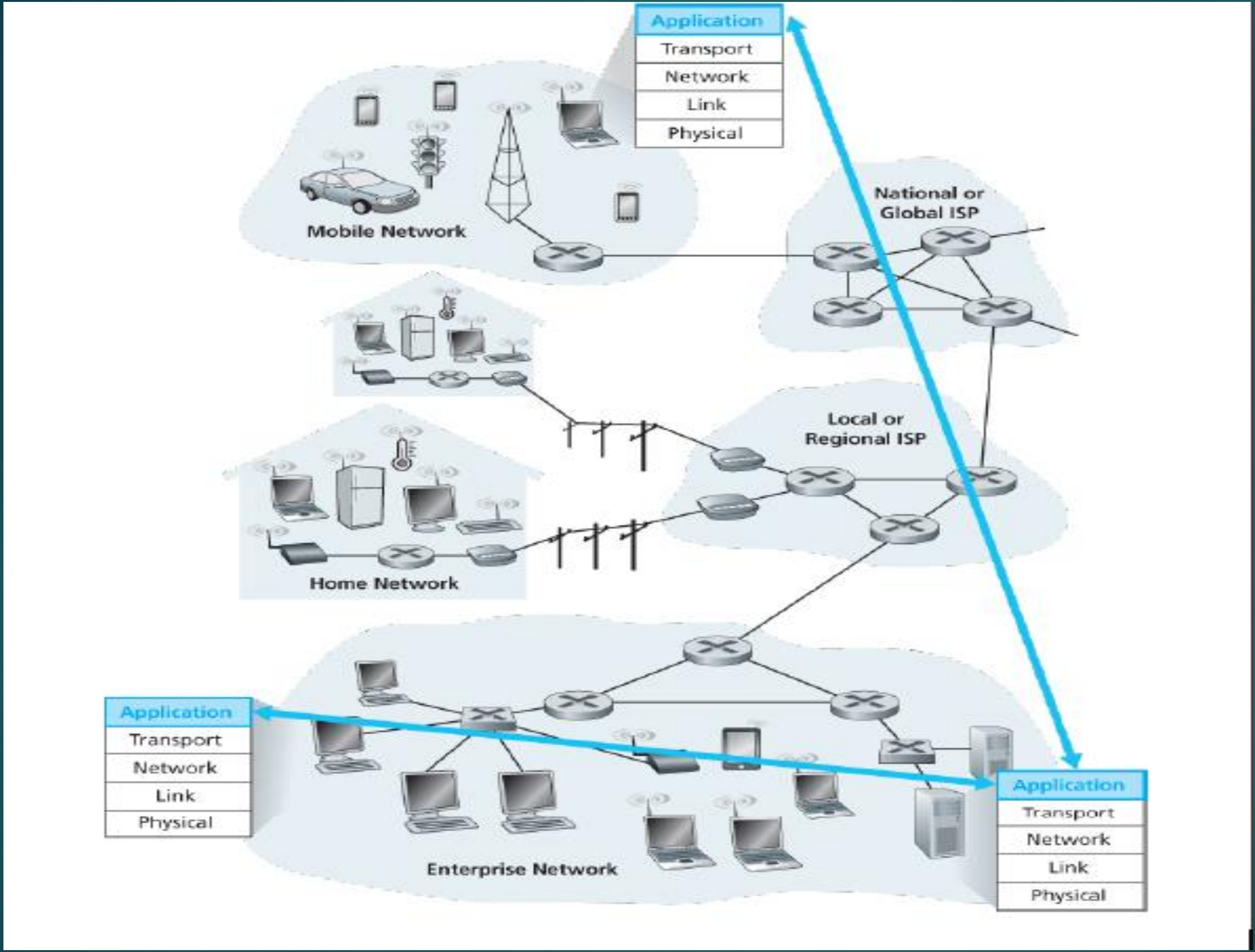
Application Layer

Chapter 2

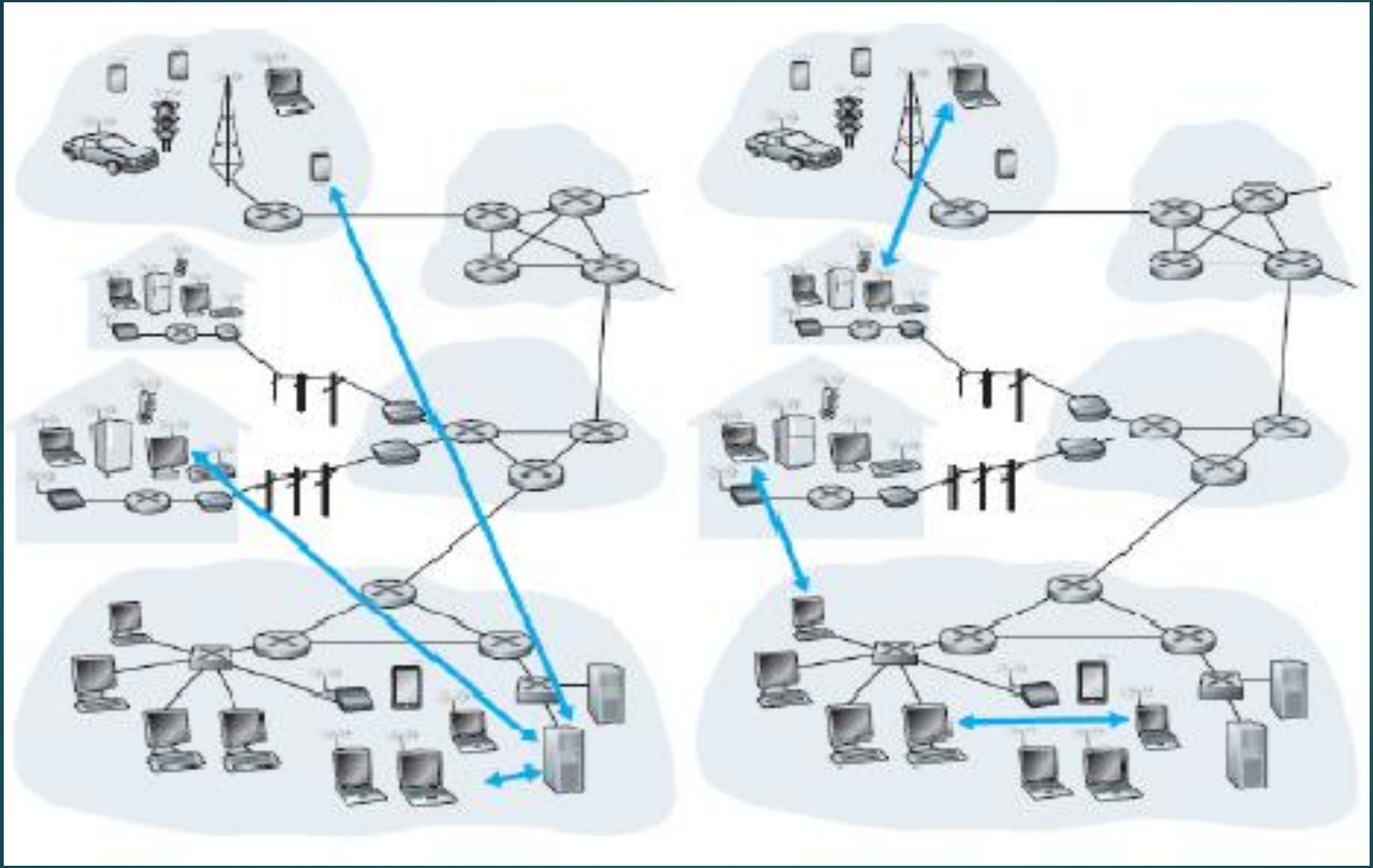
Principles of Network Application

At the core of network application development is writing programs that run on different end systems and work communicate with each other over the network. Ex: In the Web application there are two distinct programs that communicate with each other, the browser program running in the user’s host (desktop, Laptop etc…) and the web server program running in the web server host. When developing your new application, you need to write software that will run on multiple end systems. This software could be written, in C, Java, or Phyton. Importantly, you do not need to write software that runs on network-core devices, such as routers or link-layer switches.


Application
Transport
Network
Link
Physical
Mobile Network
National or Global ISP
Local or Regional ISP
Home Network
Application
Transport
Network
Link
Physical
Application
Transport
Network
Link
Physical
Enterprise Network

Network Application Architectures

Before diving into software coding, you should have a broad architectural plan for your application. From application developer’s perspective, the network architecture is fixed and provides a specific set of services to applications. The application architecture, on the other hand, is designed by the application developer and dictates how the application is structured over the various end systems. In choosing the application architecture, an application developer will likely draw on one of the two predominant architectural paradigms used in modern network applications, the client server architecture or peer to peer architecture. In Client-server architecture, there is an always-on host, called the server, which servers requests from many other hosts, called clients. When a Web services requests from browsers running on client hosts. When a Web server receives a request for an object from a client host, it responds by sending the requested object to the client host. Another characteristic of the client-server architecture is that the server has a fixed, well-known address, called IP-address. Because server has a fixed well-known IP address and because the server is always on, a client can always contact the server by sending a packet to the servers IP address.



Often in a client-server application, a single-server host is incapable of keeping up with all the requests from clients. Data center housing a large number of hosts, is often used to create a powerful virtual server. In a P2P architecture, there is minimal reliance on dedicated servers in data centers. Instead the application exploits direct communication between pairs of intermittently connected hosts, called peers. The peers are not owned by the service provider, but are instead desktops and laptops controlled by users, with most of the peers residing in homes, universities, and offices. One of the most compelling features of P2P architectures is their self-scalability. Example: in a P2P file-sharing application, although each peer generates workload by requesting files, each peer also adds service capacity to the system by distributing files to other peers. P2P architectures are also cost effective, since they normally don’t require significant server infrastructure and server bandwidth.

Processes Communicating

In the jargon of operating systems, it is not actually programs but processes that communicate. A process can be thought of as a program that is running within an end system. When processes are running on the same end system, they can communicate with each other with inter process communication, using rules that are governed by in how end system's operating system. Process on 2 different end systems communicate with each other by exchanging messages across the computer network. A sending process creates and sends messages into the network, a receiving process receives these messages and possibly responds by sending messages back.

Client and Server Processes

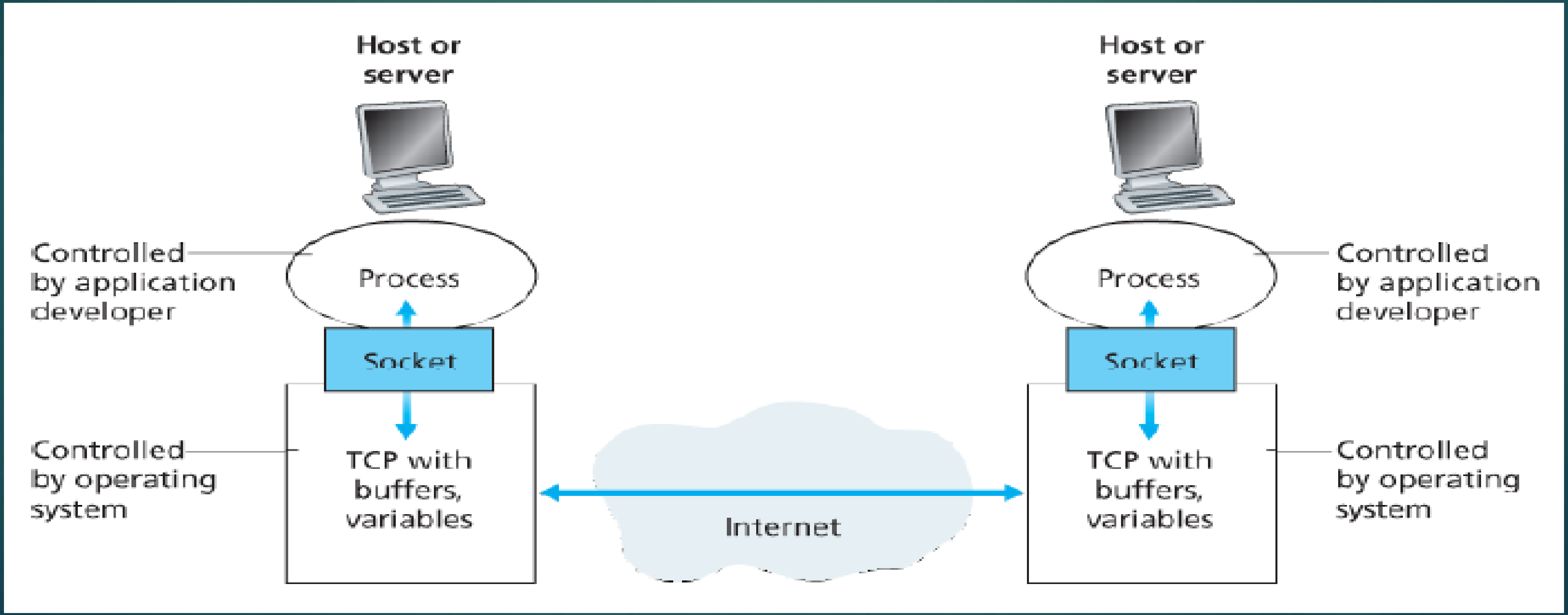
A network application consists of pairs of processes the send messages to each other over a network. In a P2P file-sharing system, a file is transferred from a process in one peer to process in another peer. For each pair of communicating processes, we typically label one of the 2 processes as the client and the other process as the server. In the Web, a browser process initializes contact with a Web server process hence the browser process is the client and the Web server process is the server.

The interface between the process and the computer network

Any message sent from one process to another must go through the underlying network. A process sends a messages into, and receives messages from, the network through a software interface called a socket. A process is analogous to a house and it’s socket is analogous to it’s door. When process wants to send a message to another process on another host, it shows the message out it’s door (socket). This sending process assumes that there is a transportation infrastructure on the other side of it’s door that will transport the message to the door of the destination process. Application Programming Interface (API) between the application and the network, since the socket is the programming interface with which network applications are built. The application developer has control of everything on the application-layer side of socket but has little control of the transport-layer side of the socket.

Addressing Processes

In order to send postal mail to a particular destination, the destination needs to have an address. Process running on one host to send packets to a process running on another host, the receiving process needs to have an address. To identify the receiving process, 2 pieces of information needs to be specified the address of the host and an identifier that specifies the receiving process in the destination host. In the internet the host is identified by IP address.


Host or server
Host or server
Controlled by application developer
Process
Process
Controlled by application developer
Socket
Socket
Controlled by operating system
TCP with buffers, variables
Internet
TCP with buffers, variables
Controlled by operating system

Transport Services Available to applications

Recall that socket is the interface between the application process and the transport-layer protocol. The application at the sending side pushes messages through the socket. At the other side of the socket, the transport-layer protocol has the responsibility of getting the messages to the socket of the receiving process. Many networks, including the internet, provide more than one transport-layer protocol.

Reliable Data Transfer

The applications has to be done to guaranteed data sent by one end of the application. If a protocol provides such a guaranteed data delivery service, it is said to provide reliable data transfer. One important service that a transport-layer protocol can potentially provide to an application is process-to-process reliable data transfer. When a transport protocol provides this service, the sending process can just pass it’s data into the socket and know with complete confidence that the data will arrive without errors at the receiving process.

Throughput

In the context of a communication session between 2 process along with path, is the rate at which the sending process can deliver bits to the receiving process. Because other sessions will be sharing the bandwidth along the network path, and because these other sessions will be coming and going, the available throughput can fluctuate with time. Which such a service, the application could request a guaranteed throughput of “r bits/sec” and the transport protocol would then ensure that the available throughput is always at least “r bits/sec”. Application that have throughput requirements are said to be bandwidth-sensitive applications. While bandwidth-sensitive applications have specific throughput requirements, elastic applications can make use of as much, or as little.

Timing

A transport-layer protocol can also provide timing guarantees. As with throughput guarantees, timing guarantees can come in many shapes and forms. A service would be applying to interactive real-time applications, such as Internet telephony, virtual environments, multiplayer games, all of which require tight timing constraints on data delivery in order to be effective.

Security

A transport protocol can provide an application with one or more security services. Example: In the sending host, a transport protocol can encrypt all data transmitted by the sending process, and in the receiving host, the transport-layer protocol can decrypt the data before delivering the data to the receiving process. Such a service would provide confidentiality between the two processes even if the data is somehow observed between sending and receiving processes. A transport protocol can also provide other security services in addition to confidentiality, including data integrity and end-point authentication.

Transport Services Provided by the Internet

The internet and, more generally, (TCP/IP networks) makes 2 transport protocols available to applications, UDP and TCP. When you create a new network application for the Internet, one of the first decisions you have to make is whether to use UDP or TCP.

TCP Services

The TCP service model includes a connection-oriented service and a reliable data transfer service. When an application invokes TCP as it’s transport protocol, the application receives both of these services from TCP.

Connection-oriented service. TCP has the client and server exchange transport-layer control information with each other before the application-level messages begin to flow. This so-called handshaking procedure alerts the client and server, allowing them to prepare for and onslaught of packets. After the handshaking phase, a TCP connection is said to exit between the sockets of the 2 processes. The connection is a full-duplex connection in that the 2 processes can send messages to each other over the connection at the same time .

Reliable data transfer service. The communicating processes can rely on TCP to deliver all data sent without error and in the proper order. When one side of the application passes a stream of bytes into a socket, it can count on TCP to deliver the same stream of bytes to the receiving socket, with no missing or duplicate bytes.

TCP also includes a congestion-control mechanism, a service for the general welfare of the internet rather than for the direct benefit of the communicating processes. The TCP congestion-control mechanism throttles a sending process (client or server) when the network is congested between sender and receiver.

UDP Services

UDP is a no-frills, lightweight transport protocol, providing minimal services. UDP is connectionless, so there is no handshaking before 2 processes start to communicate. UDP provides an unreliable data transfer service that is when a process sends a message into an UDP socket, UDP provides no guarantee that the message will ever reach the receiving process. Messages that do arrive at the receiving process may arrive out of order. UDP doesn’t include a congestion-control mechanism, so the sending side of UDP can pump data into the layer below (the network layer) at any rate it pleases, (However, that the actual end-to-end throughput may be less than this rate due to the limited transmission capacity of intervening links or due congestion)

Application-Layer Protocols

An application-layer protocols defines how an application’s processes, running on different end systems, pass messages to each other. In particular, an application-layer protocol defines:

1. The types of messages exchanged. Ex: Request messages and response messages
2. The syntax of the various message types, such as the fields in the message and how the fields are delineated.
3. The semantics of the fields, that is meaning of the information in the fields
4. Rules for determining when and how a process sends messages and responds to messages.

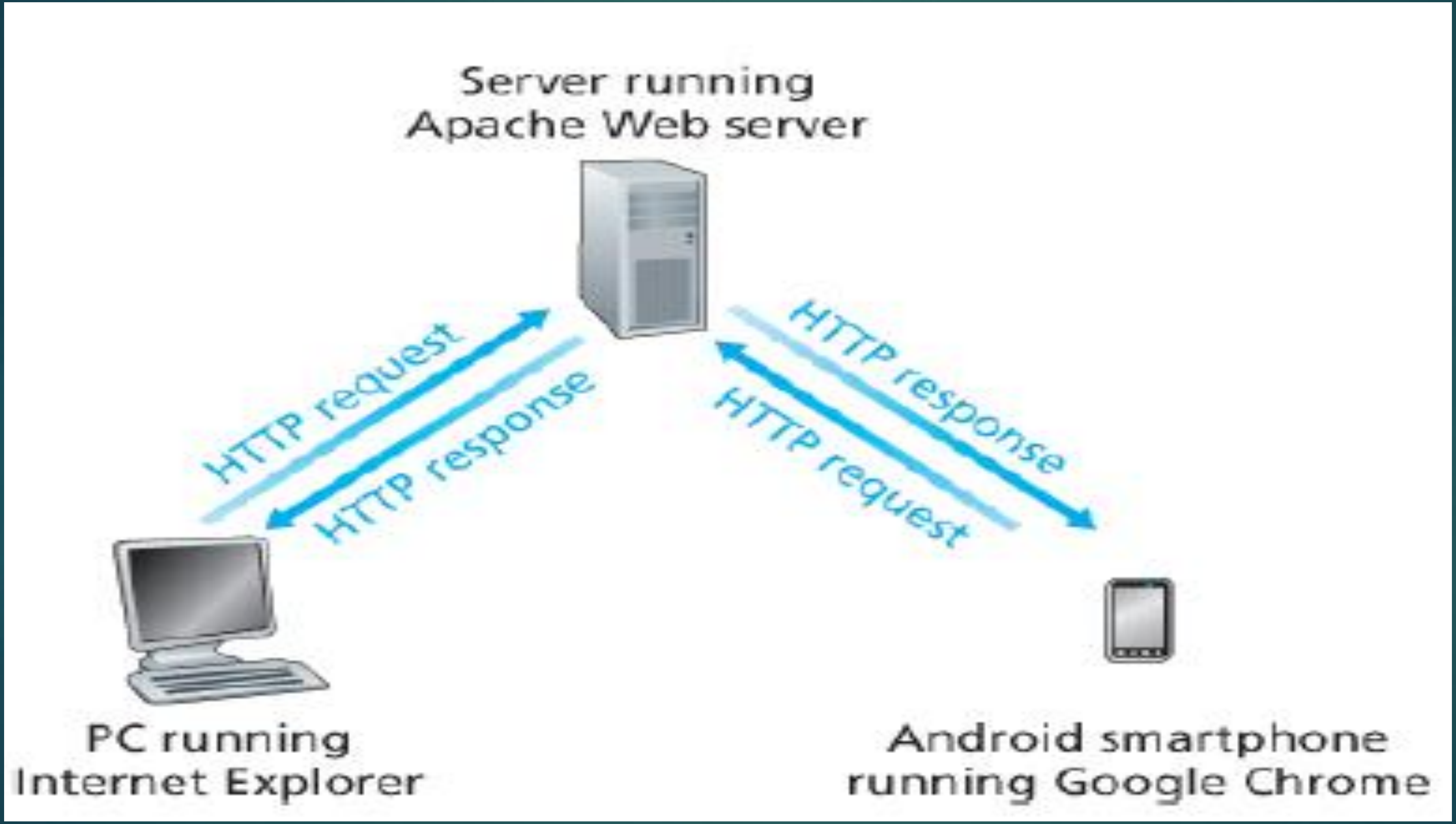
Some application-layer protocols are specified in RFCs and are therefore in the public domain. Many other application-layer protocols are proprietary and intentionally not available in the public domain. The important to distinguish between network application and application-layer protocols. An application-layer protocol is only one piece of a network application. The Web is a client-server application that allows users to obtain documents from Web servers on demand. The Web application consists of many components including a standard for document formats (HTML, Web browsers, Web Servers (Apache and Microsoft servers), and an application-layer protocol. The Web’s application-layer protocol, HTTP, defines the format and sequence of messages exchanged between browser and Web application. The application-layer protocol for electronic mail is SMTP. E-mail’s application-layer protocol, SMTP, is only one piece of the e-mail application.

The Web and HTTP

Until the early 1990’s the internet was used primarily by researchers, academics, and university students to log in to remote hosts, to transfer files from local hosts to remote hosts and vice versa, to receive and send news, and to receive and send electronic mail. These applications were extremely useful, the Internet was essentially unknown outside of the academic and research communities. It elevated the Internet from just one of many data networks to essentially the one and only data network. Perhaps what appeals the most to users is that the Web operates on demand. Users receive what they want, when they want it. This is unlike traditional broadcast radio and television, which force users to tune in when the content provider makes the content available. Hyperlinks and search engines help us navigate through an ocean of information. Forms, JavaScript, Java applets, and many other devices enable us to interact with pages and sites. The Web and it’s protocols serve as a platform for Youtube, Web-based e-mail, and most mobile internet applications, including Instagram and google maps.

Overview of HTTP

The HyperText Transfer Protocol (HTTP), the Web’s application-layer protocol, is at the heart of the Web. HTTP is implemented in 2 programs: a client program and a server program. The Client program and server program, executing on different end systems, talk to each other by exchanging HTTP messages, HTTP defines the structure of these messages and how the client and server exchange the messages. A Web page consists of objects. An object is simply a file such as an HTML file, a JPEG image, a Java applet, or a video clip that is addressable by a single URL. Most Web pages consist of a base HTML file and several referenced objects. The base HTML file references the other objects in the page with the objects URL. Each URL has 2 components the hostname of the server that houses the object and the object’s path name. Web servers which implement the server side of HTTP, house Web objects, each addressable by a URL. Popular Web servers include Apache and Microsoft Internet Information Server. HTTP uses TCP as it’s underlying transport protocol. The HTTP client first initiates a TCP connection with the server. Once the connection is established, the browser and the server processes access TCP through their socket interfaces. The important note is the server sends requested files to clients without storing any state information about the client. If a particular client asks for the same object twice in a period of a few seconds, the server does not respond by saying that it just server the object to the client, instead, the server resends the object, as it has completely forgotten what It did earlier. Because an HTTP server maintains no information about the clients, HTTP is said to be a stateless protocol.


Server running
Apache Web server
HTTP request
HTTP response
HTTP response
HTTP request
PC running
Internet Explorer
Android smartphone
running Google Chrome

Non-Persistent and Persistent Connections

In many internet applications the client and server communicate for and extended period of time, which the client making a series of requests and the server responding to each of the requests. Depending on the application and on how the application is being used, the series of requests may be made back-to-back, periodically at regular intervals, or intermittently. When this client-server interaction is taking place over TCP, the application developer needs to make an important decision—should each request/response pair be sent over a *separate* TCP connection, or should all of the requests and their corresponding responses be sent over the *same* TCP connection? In the former approach, the application is said to use non-persistent connections; and in the latter approach, persistent connections.

HTTP with Non-Persistent Connections

1. The HTTP client process initiates a TCP connection to the server www.someSchool.edu on port number 80, which is the default port number for HTTP. Associated with the TCP connection, there will be a socket at the client and a socket at the server.

2. The HTTP client sends an HTTP request message to the server via its socket. The request message includes the path name /someDepartment/home .index . (We will discuss HTTP messages in some detail below.)

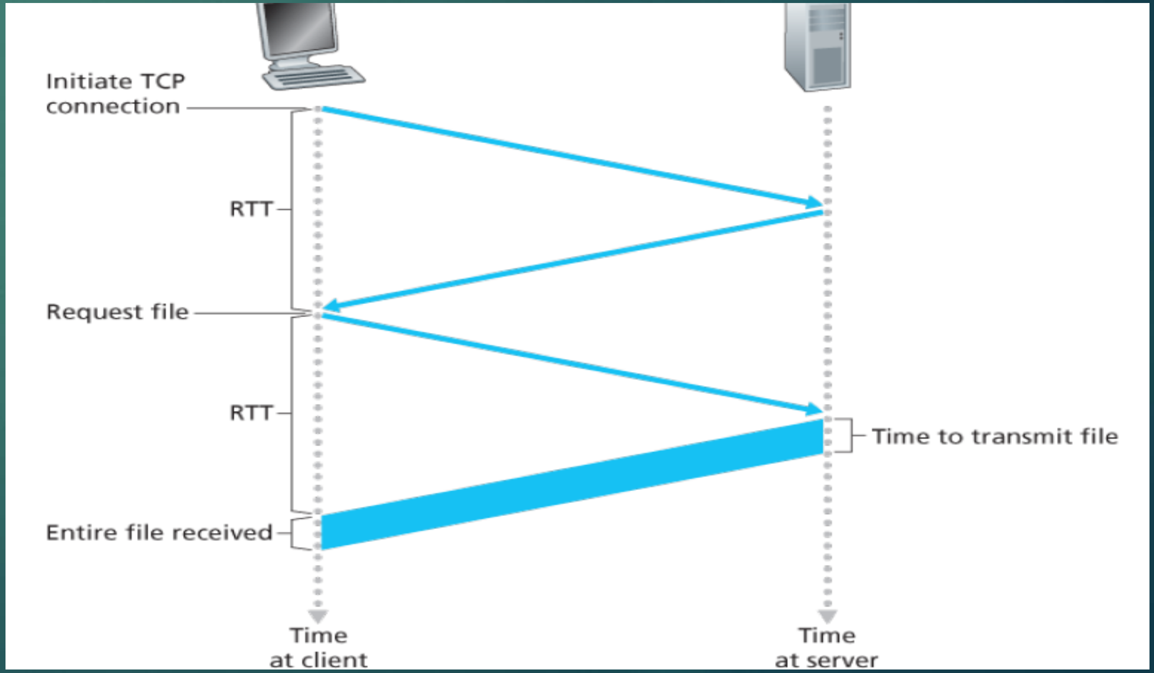
3. The HTTP server process receives the request message via its socket, retrieves the object /someDepartment/home.index from its storage (RAM or disk), encapsulates the object in an HTTP response message, and sends the response message to the client via its socket.

4. The HTTP server process tells TCP to close the TCP connection. (But TCP doesn’t actually terminate the connection until it knows for sure that the client has received the response message intact.)

5. The HTTP client receives the response message. The TCP connection terminates. The message indicates that the encapsulated object is an HTML file. The client extracts the file from the response message, examines the HTML file, and finds references to the 10 JPEG objects.

6. The first four steps are then repeated for each of the referenced JPEG objects.

As the browser receives the Web page, it displays the page to the user. Two different browsers may interpret (that is, display to the user) a Web page in somewhat different ways. HTTP has nothing to do with how a Web page is interpreted by a client. The steps above illustrate the use of non-persistent connections, where each TCP connection is closed after the server sends the object—the connection does not persist for other objects. Note that each TCP connection transports exactly one request message and one response message.



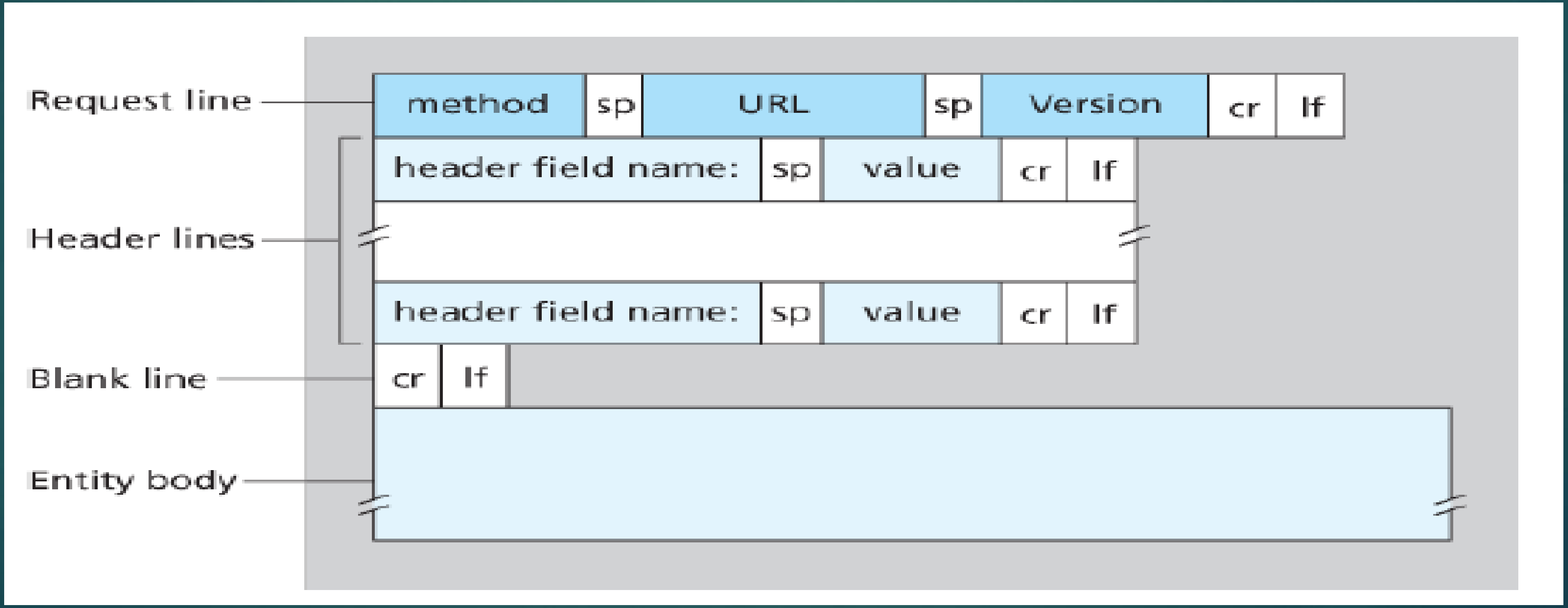
HTTP with Persistent Connections

Non-persistent connections have some shortcomings. First, a brand-new connection must be established and maintained for each requested object. For each of these connections, TCP buffers must be allocated and TCP variables must be kept in both the client and server. This can place a significant burden on the Web server, which may be serving requests from hundreds of different clients simultaneously. Second, as we just described, each object suffers a delivery delay of two RTTs—one RTT to establish the TCP connection and one RTT to request and receive an object. With HTTP 1.1 persistent connections, the server leaves the TCP connection open after sending a response. Subsequent requests and responses between the same client and server can be sent over the same connection. In particular, an entire Web page (in the example above, the base HTML file and the 10 images) can be sent over a single persistent TCP connection. Moreover, multiple Web pages residing on the same server can be sent from the server to the same client over a single persistent TCP connection. These requests for objects can be made back-to-back, without waiting for replies to pending requests (pipelining). Typically, the HTTP server closes a connection when it isn't used for a certain time (a configurable timeout interval). When the server receives the back-to-back requests, it sends the objects back-to-back. The default mode of HTTP uses persistent connections with pipelining.

HTTP Message Format

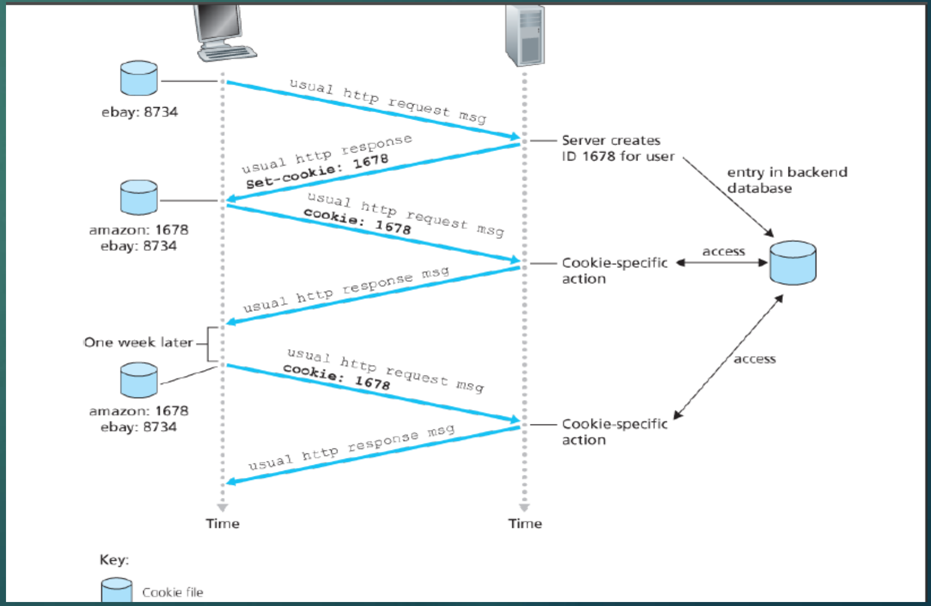
The HTTP specifications [RFC 1945; RFC 2616; RFC 7540] include the definitions of the HTTP message formats. There are two types of HTTP messages, request messages and response messages. First of all, we see that the message is written in ordinary ASCII text, so that your ordinary computer-literate human being can read it. Second, we see that the message consists of five lines, each followed by a carriage return and a line feed. The last line is followed by an additional carriage return and line feed. Although this particular request message has five lines, a request message can have many more lines or as few as one line. The first line of an HTTP request message is called the request line; the subsequent lines are called the header lines. The request line has three fields: the method field, the URL field, and the HTTP version field. The method field can take on several different values, including GET, POST, HEAD, PUT, and DELETE . The great majority of HTTP request messages use the GET method. The GET method is used when the browser requests an object, with the requested object identified in the URL field. In this example, the browser is requesting the object /somedir/page.html . The version is selfexplanatory; in this example, the browser implements version HTTP/1.1.

The entity body is empty with the GET method, but is used with the POST method. An HTTP client often uses the POST method when the user fills out a form—for example, when a user provides search words to a search engine. With a POST message, the user is still requesting a Web page from the server, but the specific contents of the Web page.



User-Server Interaction: Cookies

This simplifies server design and has permitted engineers to develop high-performance Web servers that can handle thousands of simultaneous TCP connections. However, it is often desirable for a Web site to identify users, either because the server wishes to restrict user access or because it wants to serve content as a function of the user identity. For these purposes, HTTP uses cookies. Cookies, defined in [RFC 6265], allow sites to keep track of users. The first time a user visits a site, the user can provide a user identification (possibly his or her name). During the subsequent sessions, the browser passes a cookie header to the server, thereby identifying the user to the server. Cookies can thus be used to create a user session layer on top of stateless HTTP.



Web Caching

A Web cache—also called a proxy server—is a network entity that satisfies HTTP requests on the behalf of an origin Web server. The Web cache has its own disk storage and keeps copies of recently requested objects in this storage.

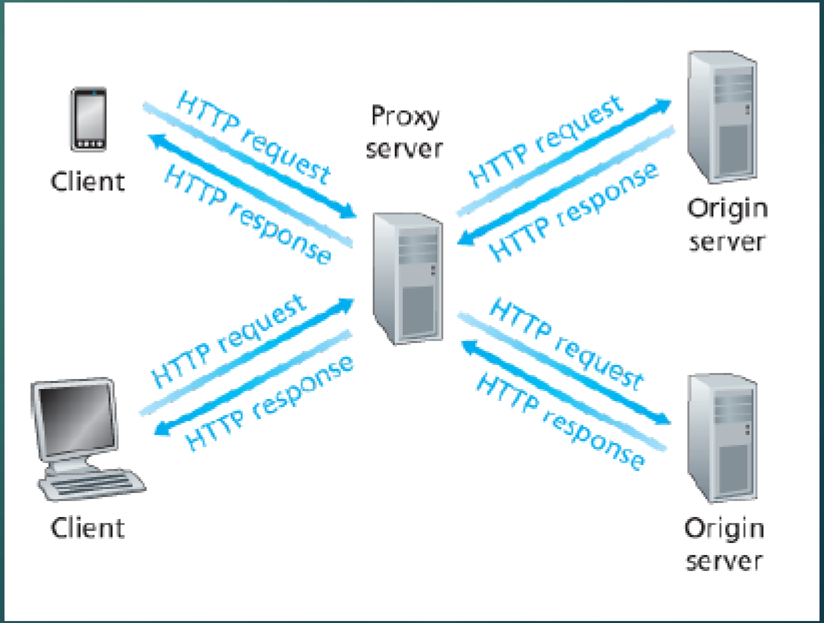
1. The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.

2. The Web cache checks to see if it has a copy of the object stored locally. If it does, the Web cache returns the object within an HTTP response message to the client browser.

3. If the Web cache does not have the object, the Web cache opens a TCP connection to the origin server, that is, to *www.someschool.edu* . The Web cache then sends an HTTP request for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the Web cache.

4. When the Web cache receives the object, it stores a copy in its local storage and sends a copy, within an HTTP response message, to the client browser (over the existing TCP connection between the client browser and the Web cache).

Typically a Web cache is purchased and installed by an ISP. For example, a university might install a cache on its campus network and configure all of the campus browsers to point to the cache. Or a major residential ISP (such as Comcast) might install one or more caches in its network and preconfigure its shipped browsers to point to the installed caches. Web caching has seen deployment in the Internet for two reasons. First, a Web cache can substantially reduce the response time for a client request, particularly if the bottleneck bandwidth between the client and the origin server is much less than the bottleneck bandwidth between the client and the cache. If there is a high-speed connection between the client and the cache, as there often is, and if the cache has the requested object, then the cache will be able to deliver the object rapidly to the client. Second, as we will soon illustrate with an example, Web caches can substantially reduce traffic on an institution's access link to the Internet.



Electronic Mail in the Internet

Electronic mail has been around since the beginning of the Internet. It remains one of the Internet’s most important and utilized applications. As with ordinary postal mail, e-mail is an asynchronous communication medium—people send and read messages when it is convenient for them, without having to coordinate with other people’s schedules. In contrast with postal mail, electronic mail is fast, easy to distribute, and inexpensive. Modern e-mail has many powerful features, including messages with attachments, hyperlinks, HTML-formatted text, and embedded photos. Mail servers form the core of the e-mail infrastructure. Each recipient, such as Bob, has a mailbox located in one of the mail servers. Bob’s mailbox manages and maintains the messages that have been sent to him. A typical message starts its journey in the sender’s user agent, travels to the sender’s mail server, and travels to the recipient’s mail server, where it is deposited in the recipient’s mailbox. SMTP is the principal application-layer protocol for Internet electronic mail. It uses the reliable data transfer service of TCP to transfer mail from the sender’s mail server to the recipient’s mail server. As with most application-layer protocols, SMTP has two sides: a client side, which executes on the sender’s mail server, and a server side, which executes on the recipient’s mail server. Both the client and server sides of SMTP run on every mail server. When a mail server sends mail to other mail servers, it acts as an SMTP client. When a mail server receives mail from other mail servers, it acts as an SMTP server.

SMTP

SMTP, defined in RFC 5321, is at the heart of Internet electronic mail. As mentioned above, SMTP transfers messages from senders' mail servers to the recipients' mail servers. SMTP is much older than HTTP. Although SMTP has numerous wonderful qualities, as evidenced by its ubiquity in the Internet, it is nevertheless a legacy technology that possesses certain archaic characteristics. The important to observe that SMTP does not normally use intermediate mail servers for sending mail, even when the two mail servers are located at opposite ends of the world. the SMTP protocol has many similarities with protocols that are used for face-to-face human interaction. First, the client SMTP (running on the sending mail server host) has TCP establish a connection to port 25 at the server SMTP (running on the receiving mail server host). If the server is down, the client tries again later. Once this connection is established, the server and client perform some application-layer handshaking—just as humans often introduce themselves before transferring information from one to another, SMTP clients and servers introduce themselves before transferring information. During this SMTP handshaking phase, the SMTP client indicates the email address of the sender (the person who generated the message) and the e-mail address of the recipient. Once the SMTP client and server have introduced themselves to each other, the client sends the message. SMTP can count on the reliable data transfer service of TCP to get the message to the server without errors. The client then repeats this process over the same TCP connection if it has other messages to send to the server; otherwise, it instructs TCP to close the connection.

Comparison with HTTP

Now briefly compare SMTP with HTTP. Both protocols are used to transfer files from one host to another: HTTP transfers files (also called objects) from a web server to a web client (typically a browser); SMTP transfers files (that is, e-mail messages) from one mail server to another mail server. When transferring the files, both persistent HTTP and SMTP use persistent connections. Thus, the two protocols have common characteristics. However, there are important differences. First, HTTP is mainly a pull protocol—someone loads information on a web server and users use HTTP to pull the information from the server at their convenience. In particular, the TCP connection is initiated by the machine that wants to receive the file. On the other hand, SMTP is primarily a push protocol—the sending mail server pushes the file to the receiving mail server. In particular, the TCP connection is initiated by the machine that wants to send the file. A second difference, which we alluded to earlier, is that SMTP requires each message, including the body of each message, to be in 7-bit ASCII format. If the message contains characters that are not 7-bit ASCII (for example, French characters with accents) or contains binary data (such as an image file), then the message has to be encoded into 7-bit ASCII. HTTP data does not impose this restriction. A third important difference concerns how a document consisting of text and images (along with possibly other media types) is handled. As we learned in **Section 2.2**, HTTP encapsulates each object in its own HTTP response message. SMTP places all of the message’s objects into one message.

Mail Message Formats

when an e-mail message is sent from one person to another, a header containing peripheral information precedes the body of the message itself. This peripheral information is contained in a series of header lines, which are defined in RFC 5322. The header lines and the body of the message are separated by a blank line (that is, by CRLF). RFC 5322 specifies the exact format for mail header lines as well as their semantic interpretations. As with HTTP, each header line contains readable text, consisting of a keyword followed by a colon followed by a value. Some of the keywords are required and others are optional. Every header must have a From: header line and a To: header line; a header may include a Subject: header line as well as other optional header lines

Mail Access Protocols

POP3

POP3 is an extremely simple mail access protocol. It is defined in [RFC 1939], which is short and quite readable. Because the protocol is so simple, its functionality is rather limited. POP3 begins when the user agent (the client) opens a TCP connection to the mail server (the server) on port 110. With the TCP connection established, POP3 progresses through three phases: authorization, transaction, and update. During the first phase, authorization, the user agent sends a username and a password (in the clear) to authenticate the user. During the second phase, transaction, the user agent retrieves messages; also during this phase, the user agent can mark messages for deletion, remove deletion marks, and obtain mail statistics. The third phase, update, occurs after the client has issued the quit command, ending the POP3 session; at this time, the mail server deletes the messages that were marked for deletion. In a POP3 transaction, the user agent issues commands, and the server responds to each command with a reply. There are two possible responses: +OK (sometimes followed by server-to-client data), used by the server to indicate that the previous command was fine; and -ERR , used by the server to indicate that something was wrong with the previous command. The authorization phase has two principal commands: user <username> and pass <password> . To illustrate these two commands, we suggest that you Telnet directly into a POP3 server, using port 110, and issue these commands. Suppose that mailServer is the name of your mail server.

During a POP3 session between a user agent and the mail server, the POP3 server maintains some state information; in particular, it keeps track of which user messages have been marked deleted. However, the POP3 server does not carry state information across POP3 sessions. This lack of state information across sessions greatly simplifies the implementation of a POP3 server.

IMAP

his paradigm— namely, folders and messages in the local machine—poses a problem for the nomadic user, who would prefer to maintain a folder hierarchy on a remote server that can be accessed from any computer. This is not possible with POP3—the POP3 protocol does not provide any means for a user to create remote folders and assign messages to folders. To solve this and other problems, the IMAP protocol, defined in **[RFC 3501]**, was invented. Like POP3, IMAP is a mail access protocol. It has many more features than POP3, but it is also significantly more complex. (And thus the client and server side implementations are significantly more complex.) An IMAP server will associate each message with a folder; when a message first arrives at the server, it is associated with the recipient’s INBOX folder. The recipient can then move the message into a new, user-created folder, read the message, delete the message, and so on. The IMAP protocol provides commands to allow users to create folders and move messages from one folder to another. IMAP also provides commands that allow users to search remote folders for messages matching specific criteria. Note that, unlike POP3, an IMAP server maintains user state information across IMAP sessions—for example, the names of the folders and which messages are associated with which folders.

DNS—The Internet’s Directory Service

One identifier for a host is its hostname. Hostnames—such as www.facebook.com, www.google.com ,and more they are therefore appreciated by humans. However, hostnames provide little, if any, information about the location within the Internet of the host. Furthermore, because hostnames can consist of variable-length alphanumeric characters, they would be difficult to process by routers. For these reasons, hosts are also identified by so-called IP addresses.

Services Provided by DNS

This is the main task of the Internet’s domain name system (DNS). The DNS is a distributed database implemented in a hierarchy of DNS servers, and an application-layer protocol that allows hosts to query the distributed database. The DNS servers are often UNIX machines running the Berkeley Internet Name Domain (BIND) software. The DNS protocol runs over UDP and uses port 53. DNS is commonly employed by other application-layer protocols—including HTTP and SMTP to translate user-supplied hostnames to IP addresses.

DNS provides a few other important services in addition to translating hostnames to IP addresses:

Host aliasing. A host with a complicated hostname can have one or more alias names. For example, a hostname such as relay1.west-coast.enterprise.com could have, say, two aliases such as enterprise.com and www.enterprise.com . DNS can be invoked by an application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host.

Mail server aliasing. For obvious reasons, it is highly desirable that e-mail addresses be mnemonic. For example, if Bob has an account with Yahoo Mail, Bob's e-mail address might be as simple as bob@yahoo.mail . DNS can be invoked by a mail application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host. In fact, the MX record permits a company's mail server and Web server to have identical (aliased) hostnames; for example, a company's Web server and mail server can both be called enterprise.com .

Load distribution. DNS is also used to perform load distribution among replicated servers, such as replicated Web servers. Busy sites, such as cnn.com , are replicated over multiple servers, with each server running on a different end system and each having a different IP address. For replicated Web servers, a set of IP addresses is thus associated with one canonical hostname. The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but rotates the ordering of the addresses within each reply. Because a client typically sends its HTTP request message to the IP address that is listed first in the set, DNS rotation distributes the traffic among the replicated servers. DNS rotation is also used for e-mail so that multiple mail servers can have the same alias name.

Overview of How DNS Works

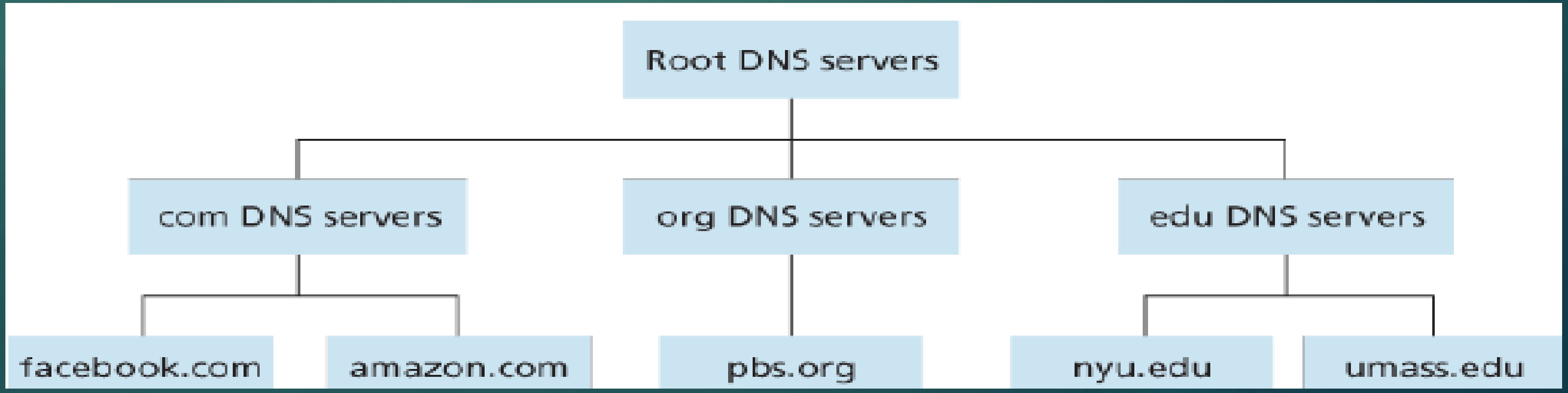
Suppose that some application (such as a Web browser or a mail reader) running in a user’s host needs to translate a hostname to an IP address. The application will invoke the client side of DNS, specifying the hostname that needs to be translated. (On many UNIX-based machines, gethostbyname() is the function call that an application calls in order to perform the translation.) DNS in the user’s host then takes over, sending a query message into the network. All DNS query and reply messages are sent within UDP datagrams to port 53. After a delay, ranging from milliseconds to seconds, DNS in the user’s host receives a DNS reply message that provides the desired mapping. This mapping is then passed to the invoking application. Thus, from the perspective of the invoking application in the user’s host, DNS is a black box providing a simple, straightforward translation service. But in fact, the black box that implements the service is complex, consisting of a large number of DNS servers distributed around the globe, as well as an application-layer protocol that specifies how the DNS servers and querying hosts communicate.

A simple design for DNS would have one DNS server that contains all the mappings. In this centralized design, clients simply direct all queries to the single DNS server, and the DNS server responds directly to the querying clients. Although the simplicity of this design is attractive, it is inappropriate for today's Internet, with its vast (and growing) number of hosts. The problems with a centralized design include:

A single point of failure. If the DNS server crashes, so does the entire Internet!

Traffic volume. A single DNS server would have to handle all DNS queries (for all the HTTP requests and e-mail messages generated from hundreds of millions of hosts).

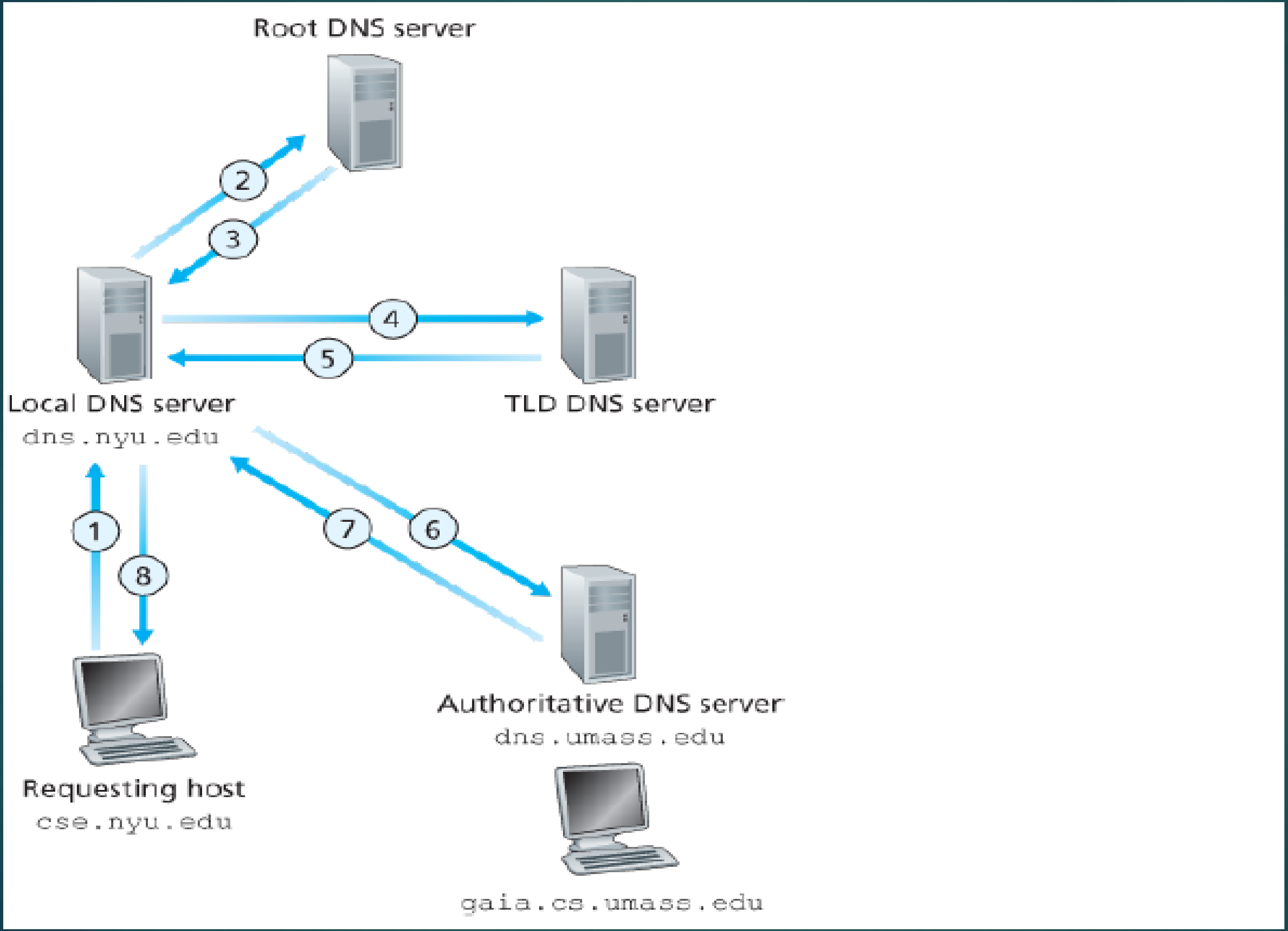
Distant centralized database. A single DNS server cannot be "close to" all the querying clients. If we put the single DNS server in New York City, then all queries from Australia must travel to the other side of the globe, perhaps over slow and congested links. This can lead to significant delays.
Maintenance. The single DNS server would have to keep records for all Internet hosts. Not only would this centralized database be huge, but it would have to be updated frequently to account for every new host.



Root DNS servers. There are over 400 root name servers scattered all over the world. These root name servers are managed by 13 different organizations. The full list of root name servers, along with the organizations that manage them and their IP addresses can be found at [Root Servers 2016]. Root name servers provide the IP addresses of the TLD servers.

Top-level domain (TLD) servers. For each of the top-level domains — top-level domains such as com, org, net, edu, and gov, and all of the country top-level domains such as uk, fr, ca, and jp — there is TLD server (or server cluster). The company Verisign Global Registry Services maintains the TLD servers for the com top-level domain, and the company Educause maintains the TLD servers for the edu top-level domain.

Authoritative DNS servers. Every organization with publicly accessible hosts (such as Web servers and mail servers) on the Internet must provide publicly accessible DNS records that map the names of those hosts to IP addresses. An organization’s authoritative DNS server houses these DNS records. An organization can choose to implement its own authoritative DNS server to hold these records; alternatively, the organization can pay to have these records stored in an authoritative DNS server of some service provider. Most universities and large companies implement and maintain their own primary and secondary (backup) authoritative DNS server.


Root DNS server
2
3
4
5
Local DNS server
dns.nyu.edu
TLD DNS server
1
8
7
6
Requesting host
cse.nyu.edu
Authoritative DNS server
dns.umass.edu
gaia.cs.umass.edu

DNS Caching

DNS extensively exploits DNS caching in order to improve the delay performance and to reduce the number of DNS messages ricocheting around the Internet. The idea behind DNS caching is very simple. In a query chain, when a DNS server receives a DNS reply (containing, for example, a mapping from a hostname to an IP address), it can cache the mapping in its local memory. hostname/IP address pair is cached in a DNS server and another query arrives to the DNS server for the same hostname, the DNS server can provide the desired IP address, even if it is not authoritative for the hostname. Because hosts and mappings between hostnames and IP addresses are by no means permanent, DNS servers discard cached information after a period of time.

DNS Records and Messages

The DNS servers that together implement the DNS distributed database store resource records (RRs), including RRs that provide hostname-to-IP address mappings. Each DNS reply message carries one or more resource records. A resource record is a four-tuple that contains the following fields:

(Name, Value, Type, TTL)

TTL is the time to live of the resource record; it determines when a resource should be removed from a cache. In the example records given below, we ignore the TTL field. The meaning of Name and Value depend on Type :

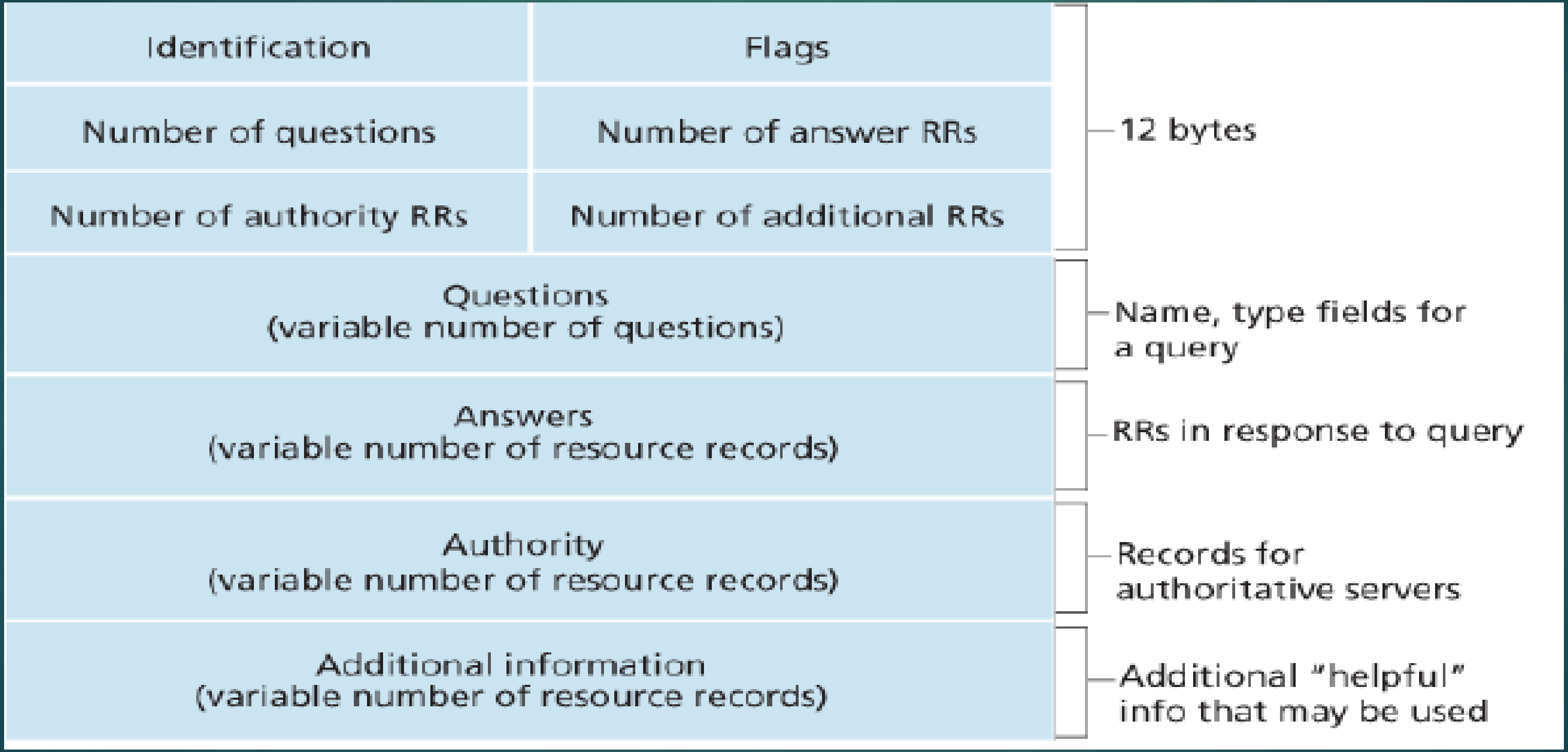
- If Type=A , then Name is a hostname and Value is the IP address for the hostname. Thus, a Type A record provides the standard hostname-to-IP address mapping. As an example, (relay1.bar.foo.com, 145.37.93.126, A) is a Type A record.
- If Type=NS , then Name is a domain (such as foo.com) and Value is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, (foo.com, dns.foo.com, NS) is a Type NS record.

- If Type=CNAME , then Value is a canonical hostname for the alias hostname Name . This record can provide querying hosts the canonical name for a hostname. As an example, (foo.com, relay1.bar.foo.com, CNAME) is a CNAME record.
- If Type=MX , then Value is the canonical name of a mail server that has an alias hostname Name . As an example, (foo.com, mail.bar.foo.com, MX) is an MX record. MX records allow the hostnames of mail servers to have simple aliases. Note that by using the MX record, a company can have the same aliased name for its mail server and for one of its other servers (such as its Web server). To obtain the canonical name for the mail server, a DNS client would query for an MX record; to obtain the canonical name for the other server, the DNS client would query for the CNAME record.

DNS Messages:

There are the only two kinds of DNS messages. Furthermore, both query and reply messages have the same format.

- The first 12 bytes is the header section, which has a number of fields. The first field is a 16-bit number that identifies the query. This identifier is copied into the reply message to a query, allowing the client to match received replies with sent queries. There are a number of flags in the flag field. A 1-bit query/reply flag indicates whether the message is a query (0) or a reply (1). A 1-bit authoritative flag is set in a reply message when a DNS server is an authoritative server for a queried name. A 1-bit recursion-desired flag is set when a client (host or DNS server) desires that the DNS server perform recursion when it doesn't have the record. A 1-bit recursion-available field is set in a reply if the DNS server supports recursion. In the header, there are also four number-of fields. These fields indicate the number of occurrences of the four types of data sections that follow the header.
- In a reply from a DNS server, the answer section contains the resource records for the name that was originally queried. Recall that in each resource record there is the Type (for example, A, NS, CNAME, and MX), the Value , and the TTL . A reply can return multiple RRs in the answer, since a hostname can have multiple IP addresses.
- The authority section contains records of other authoritative servers.
- The additional section contains other helpful records. For example, the answer field in a reply to an MX query contains a resource record providing the canonical hostname of a mail server. The additional section contains a Type A record providing the IP address for the canonical hostname of the mail server.

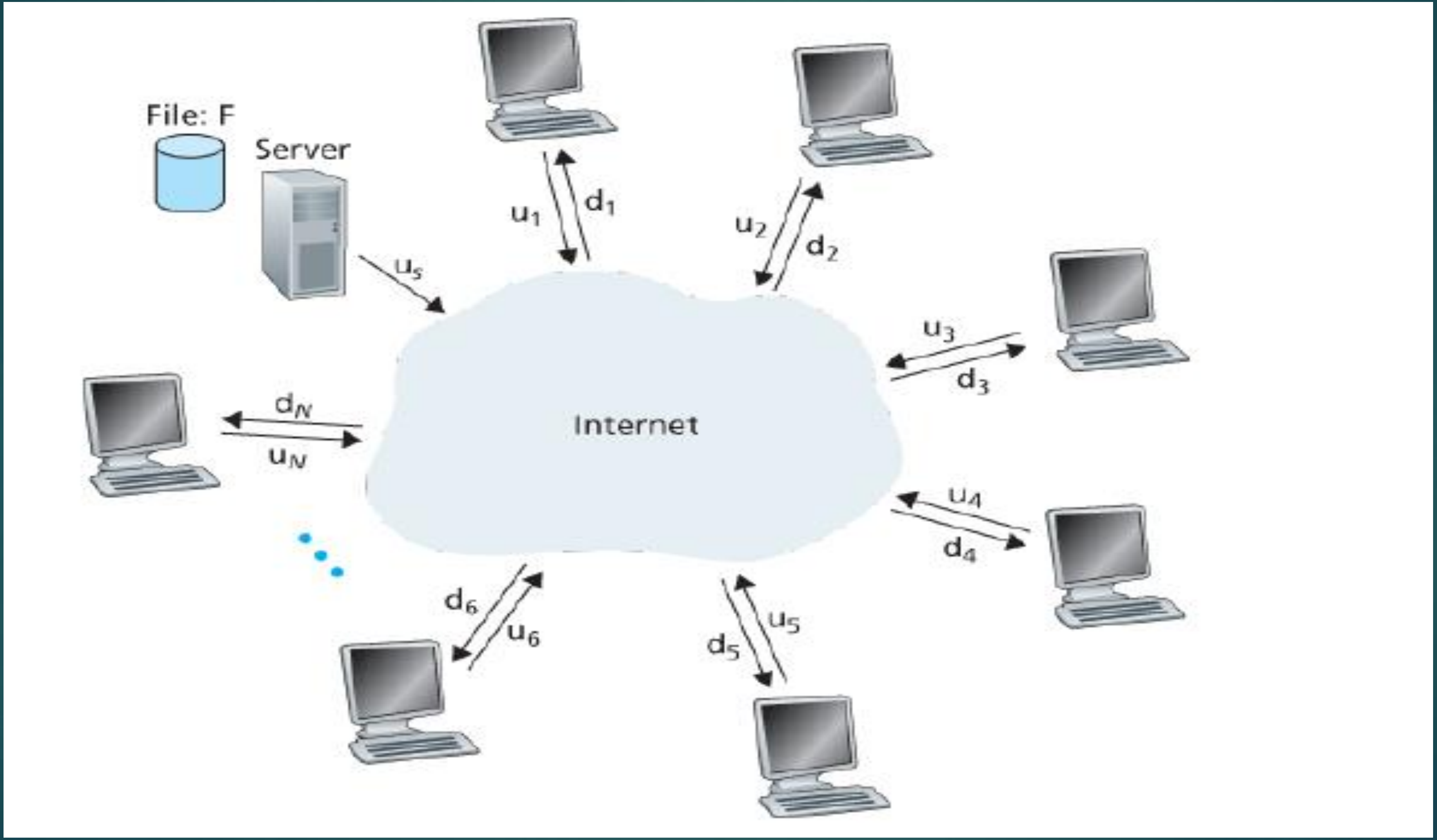

Identification
Flags
Number of questions
Number of answer RRs
Number of authority RRs
Number of additional RRs
12 bytes
Questions
(variable number of questions)
Name, type fields for a query
Answers
(variable number of resource records)
RRs in response to query
Authority
(variable number of resource records)
Records for authoritative servers
Additional information
(variable number of resource records)
Additional "helpful" info that may be used

Peer-to-Peer File Distribution

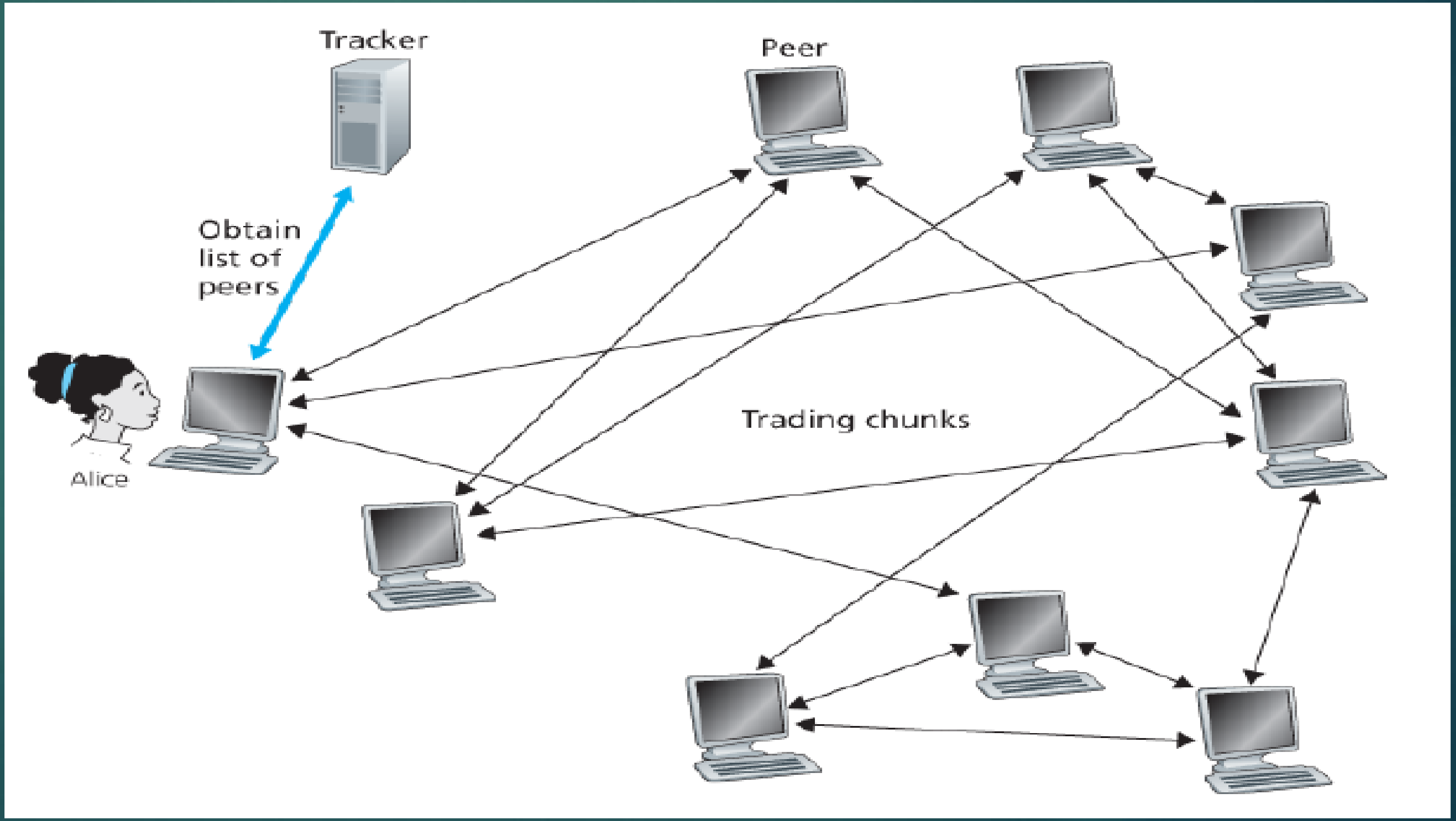
P2P architecture, there is minimal (or no) reliance on always-on infrastructure servers. Instead, pairs of intermittently connected hosts, called peers, communicate directly with each other. The peers are not owned by a service provider, but are instead desktops and laptops controlled by users. The file might be a new version of the Linux operating system, a software patch for an existing operating system or application, an MP3 music file, or an MPEG video file. In client-server file distribution, the server must send a copy of the file to each of the peers—placing an enormous burden on the server and consuming a large amount of server bandwidth. In P2P file distribution, each peer can redistribute any portion of the file it has received to any other peers, thereby assisting the server in the distribution process.

Scalability of P2P Architectures

To compare client-server architectures with peer-to-peer architectures, and illustrate the inherent self scalability of P2P, we now consider a simple quantitative model for distributing a file to a fixed set of peers for both architecture types. Denote the upload rate of the server's access link by u , the upload rate of the *i*th peer's access link by u, and the download rate of the *i*th peer's access link by d. Also denote the size of the file to be distributed (in bits) by F and the number of peers that want to obtain a copy of the file by N. The distribution time is the time it takes to get a copy of the file to all N peers.


File: F
Server
u_s
u_1
d_1
u_2
d_2
u_3
d_3
u_4
d_4
u_5
d_5
d_6
u_6
d_N
u_N
Internet

when a new peer, Alice, joins the torrent, the tracker randomly selects a subset of peers (for concreteness, say 50) from the set of participating peers, and sends the IP addresses of these 50 peers to Alice. Possessing this list of peers, Alice attempts to establish concurrent TCP connections with all the peers on this list. As time evolves, some of these peers may leave and other peers (outside the initial 50) may attempt to establish TCP connections with Alice. So a peer’s neighboring peers will fluctuate over time.



Video Streaming and Content Distribution Networks

Streaming prerecorded video now accounts for the majority of the traffic in residential ISPs in North America. We will see their implemented using application-level protocols and servers that function in some ways like a cache.

Internet Video

In streaming stored video applications, the underlying medium is prerecorded video, such as a movie, a television show, a prerecorded sporting event, or a prerecorded user-generated video (such as those commonly seen on YouTube). A video is a sequence of images, typically being displayed at a constant rate, for example, at 24 or 30 images per second. An uncompressed, digitally encoded image consists of an array of pixels, with each pixel encoded into a number of bits to represent luminance and color. An important characteristic of video is that it can be compressed, thereby trading off video quality with bit rate. From a networking perspective, perhaps the most salient characteristic of video is its high bit rate. Compressed Internet video typically ranges from 100 kbps for low-quality video to over 3 Mbps for streaming high-definition movies; 4K streaming envisions a bitrate of more than 10 Mbps. This can translate to huge amount of traffic and storage, particularly for high-end video.

HTTP Streaming and DASH

In HTTP streaming, the video is simply stored at an HTTP server as an ordinary file with a specific URL. When a user wants to see the video, the client establishes a TCP connection with the server and issues an HTTP *GET* request for that URL. The server then sends the video file, within an HTTP response message, as quickly as the underlying network protocols and traffic conditions will allow. On the client side, the bytes are collected in a client application buffer. Once the number of bytes in this buffer exceeds a predetermined threshold, the client application begins playback—specifically, the streaming video application periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user’s screen.

All clients receive the same encoding of the video, despite the large variations in the amount of bandwidth available to a client, both across different clients and also over time for the same client. This has led to the development of a new type of HTTP-based streaming, often referred to as **Dynamic Adaptive Streaming over HTTP (DASH)**. In DASH, the video is encoded into several different versions, with each version having a different bit rate and, correspondingly, a different quality level. The client dynamically requests chunks of video segments of a few seconds in length. When the amount of available bandwidth is high, the client naturally selects chunks from a high-rate version; and when the available bandwidth is low, it naturally selects from a low-rate version.

DASH allows clients with different Internet access rates to stream in video at different encoding rates. Clients with low-speed 3G connections can receive a low bit-rate (and low-quality) version, and clients with fiber connections can receive a high-quality version. DASH also allows a client to adapt to the available bandwidth if the available end-to-end bandwidth changes during the session. This feature is particularly important for mobile users. With DASH, each video version is stored in the HTTP server, each with a different URL. The HTTP server also has a **manifest file**, which provides a URL for each version along with its bit rate. The client first requests the manifest file and learns about the various versions. The client then selects one chunk at a time by specifying a URL and a byte range in an HTTP GET request message for each chunk. While downloading chunks, the client also measures the received bandwidth and runs a rate determination algorithm to select the chunk to request next. Naturally, if the client has a lot of video buffered and if the measured receive bandwidth is high, it will choose a chunk from a high-bitrate version. And naturally if the client has little video buffered and the measured received bandwidth is low, it will choose a chunk from a low-bitrate version. DASH therefore allows the client to freely switch among different quality levels.

Content Distribution Networks

Internet video company, perhaps the most straightforward approach to providing streaming video service is to build a single massive data center, store all of its videos in the data center, and stream the videos directly from the data center to clients worldwide. But there are three major problems with this approach. First, if the client is far from the data center, server-to-client packets will cross many communication links and likely pass through many ISPs, with some of the ISPs possibly located on different continents. If one of these links provides a throughput that is less than the video consumption rate, the end-to-end throughput will also be below the consumption rate, resulting in annoying freezing delays for the user.

A second drawback is that a popular video will likely be sent many times over the same communication links. Not only does this waste network bandwidth, but the Internet video company itself will be paying its provider ISP (connected to the data center) for sending the same bytes into the Internet over and over again.

A third problem with this solution is that a single data center represents a single point of failure—if the data center or its links to the Internet goes down, it would not be able to distribute any video streams.

Socket Programming: Creating Network Applications

There are two types of network applications. One type is an implementation whose operation is specified in a protocol standard, such as an RFC or some other standards document; such an application is sometimes referred to as “open,” since the rules specifying its operation are known to all. For such an implementation, the client and server programs must conform to the rules dictated by the RFC. If one developer writes code for the client program and another developer writes code for the server program, and both developers carefully follow the rules of the RFC, then the two programs will be able to interoperate. Indeed, many of today’s network applications involve communication between client and server programs that have been created by independent developers—for example, a Google Chrome browser communicating with an Apache Web server. The other type of network application is a proprietary network application. In this case the client and server programs employ an application-layer protocol that has *not* been openly published in an RFC or elsewhere. A single developer (or development team) creates both the client and server programs, and the developer has complete control over what goes in the code. But because the code does not implement an open protocol, other independent developers will not be able to develop code that interoperates with the application.

We introduce UDP and TCP socket programming by way of a simple UDP application and a simple TCP application. We present the simple UDP and TCP applications in Python 3. We could have written the code in Java, C, or C++, but we chose Python mostly because Python clearly exposes the key socket concepts. With Python there are fewer lines of code, and each line can be explained to the novice programmer without difficulty.

Socket Programming with UDP

write simple client-server programs that use UDP; in the following section, we’ll write similar programs that use TCP. Before the sending process can push a packet of data out the socket door, when using UDP, it must first attach a destination address to the packet. After the packet passes through the sender’s socket, the Internet will use this destination address to route the packet through the Internet to the socket in the receiving process. When the packet arrives at the receiving socket, the receiving process will retrieve the packet through the socket, and then inspect the packet’s contents and take appropriate action. By including the destination IP address in the packet, the routers in the Internet will be able to route the packet through the Internet to the destination host. But because a host may be running many network application processes, each with one or more sockets, it is also necessary to identify the particular socket in the destination host. When a socket is created, an identifier, called a port number, is assigned to it. So, as you might expect, the packet’s destination address also includes the socket’s port number.

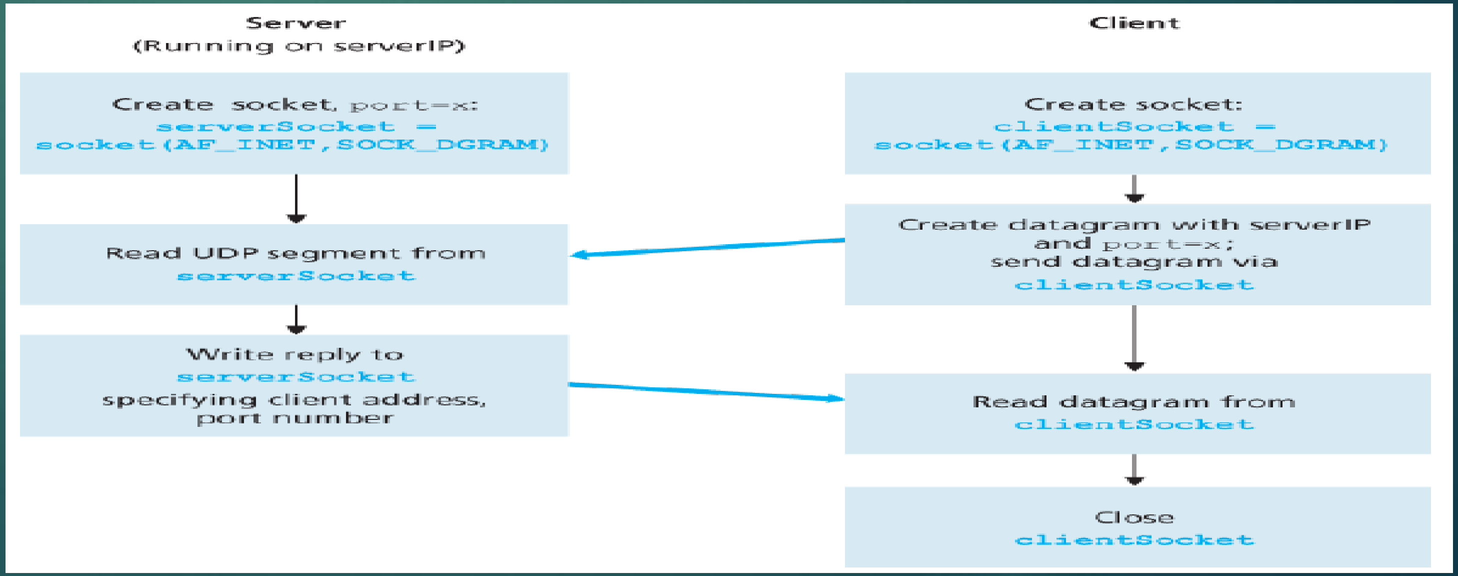
We'll use the following simple client-server application to demonstrate socket programming for both UDP and TCP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.

2. The server receives the data and converts the characters to uppercase.

3. The server sends the modified data to the client.

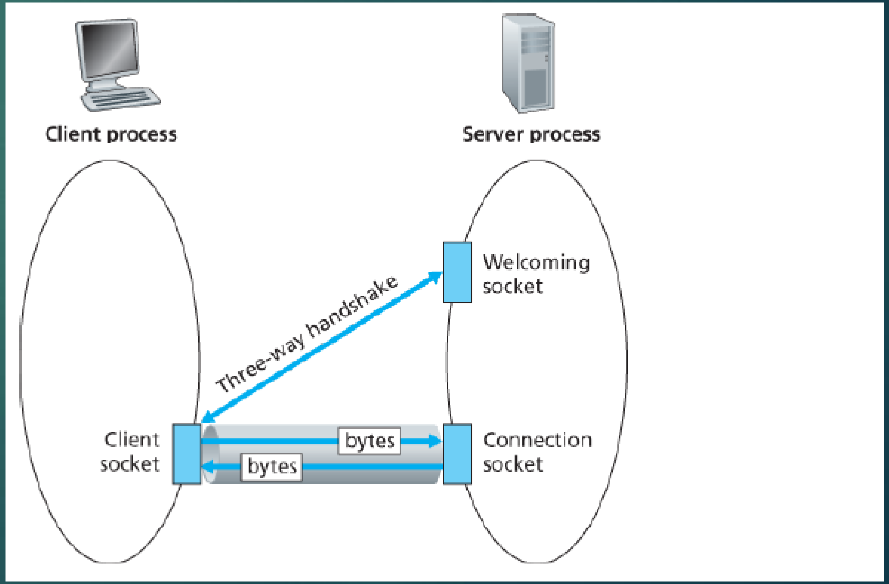
4. The client receives the modified data and displays the line on its screen.



Socket Programming with TCP

Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket. With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server programs.

During the three-way handshake, the client process knocks on the welcoming door of the server process. When the server “hears” the knocking, it creates a new door—more precisely, a new socket that is dedicated to that particular client. In our example below, the welcoming door is a TCP socket object that we call serverSocket ; the newly created socket dedicated to the client making the connection is called connectionSocket . Students who are encountering TCP sockets for the first time sometimes confuse the welcoming socket (which is the initial point of contact for all clients wanting to communicate with the server), and each newly created server-side connection socket that is subsequently created for communicating with each client.



www.dewhr.co
thanks!