Principles of Network Applications

At the core of network application development is writing programs that run on different end systems and communicate with each other over the network.

When developing your new application, you need to write software that will run on multiple end systems. Network-core devices do not function at the application layer but instead function at lower layers.

2.1.1 Network Application Architectures

From the application developer's perspective, the network architecture is fixed and provides a specific set of services to applications. The application architecture is designed by the application developer and dictates how the application is structured over the various end systems. In choosing the application architecture there are two predominant architectural paradigms: the client-server architecture or the peer-to-peer (P2P) architecture.

In a client-server architecture, there is an always-on host, called the server, which services requests from many other hosts, called clients. 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. Note that with the client-server architecture, clients do not directly communicate with each other. Another characteristic of the client-server architecture is that the server has a fixed, well-known address, called an IP address. Because the server has a fixed, well-known address, and because the server is always on, a client can always contact the server by sending a packet to the server's IP address.

Often in a client-server application, a single-server host is incapable of keeping up with all the requests from clients. For this reason, a data center, housing a large number of hosts, is often used to create a powerful virtual server. A data center can have hundreds of thousands of servers, which must be powered and maintained. Additionally, the service providers must pay recurring interconnection and bandwidth costs for sending data from their data centers.

In a P2P architecture, there is minimal (or no) 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 laptop controlled by users. Because the peers communicate without passing through a dedicated server, the architecture is called peer-to-peer.

One of the most compelling features of P2P architectures is their self-scalability. 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. However, P2P applications face challenges of security, performance, and reliability due to their highly decentralized structure.

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 interprocess communication, using rules that are governed by the end system's operating system.

Processes on two 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 that send messages to each other over a network. In the Web application a client browser process exchanges messages with a Web server process. In a P2P file-sharing system, a file is transferred from a process in one peer to a process in another peer. For each pair of communicating processes. we typically label one of the two processes as the client and the other as the server. With P2P file sharing, the peer that is downloading the file is labeled as the client, and the peer that is uploading the file is labeled as the server.

> In the context of a communication session between a pair of processes, the process that initiates the communication is labeled as the client. The process that

waits to be contacted to begin the session 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 messages into, and receives messages from, the network through a software interface called a socket.

A socket is the interface between the application layer and the transport layer within a host. It is also referred to as the 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 the socket but has little control of the transport-layer side of the socket. The only control that the application developer has on the transport-layer side is (1) the choice of transport protocol and (2) perhaps the ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes.

Addressing Processes

In order for a 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, two pieces of information need to be specified: (1) the address of the host and (2) an identifier that specifies the receiving process in the destination host.

In the Internet, the host is identifies by its IP address (A unique 32-bit quantity). In addition to knowing the address of the host to which a message is destined, the sending process must also identify the receiving process (more specifically, the receiving socket) running in the host. This information is needed because in general a host could be running many network applications. A destination por number serves this purpose.

2.1.3 Transport Services Available to Application

Recall that a 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.

Reliable Data Transfer

For many application, data loss can have devastating consequences. Thus, to support these applications, something has to be done to guarantee that the data sent by one end of the application is delivered correctly and completely to the other end of the application. If a protocol provides such a guaranteed data delivery service, it is said to provide a 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 its data into the socket and know with complete confidence that the data will arrive without errors at the receiving process.

When a transport-layer protocol doesn’t provide reliable data transfer, some of the data sent by the sending process may never arrive at the receiving process. This may be acceptable for loss-tolerant applications.

Throughput

It is the rate at which the sending process can deliver bits to the receiving process. The available throughput can fluctuate with time. These observations lead to another natural service that a transport-layer protocol could provide, guaranteed available throughput at some specified rate. With 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. Applications that have throughput requirements are said to be bandwidth-sensitive applications.

When bandwidth-sensitive applications have specific throughput requirements, elastic applications can make use of as much, or as little, throughput as happens to be available.

Timing

A transport-layer protocol can also provide timing guarantees. Longs delays in Internet telephony, for example, tend to result in unnatural pauses in the conversation. For non-real-time application, lover delay is always preferable to higher delay, but no tight constraint is placed on the end-to-end delays.

Security

A transport protocol can provide an application with one or more security services. For 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.

2.1.4 Transport Services Provided by the Internet

The Internet makes two 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 its 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 an onslaught of packets. After the handshaking phase, a TCP connection is said to exist between the socket of the two processes. The connection is a full-duplex connection in that the two processes can send messages to each other over the connection at the same time. When the application finishes sending messages, it must tear down the connection.

> 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-contrrol mechanism. 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 the two processes start to communicate. UDP provides an unreliable data transfer service. UDP provides no guarantee that the message will ever reach the receiving process. Furthermore, messages that do arrive at the receiving process may arrive out of order.

UDP does not 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.

2.1.5 Application-Layer Protocols

An application-layer protocol defines:

- The types of messages exchanged, for example, request and response messages.
- The syntax of various message types, such as the fields in the message and how the fields are delineated.
- The semantics of the fields, that is, the meaning of the information in the fields.
- 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.

It is important to distinguish between network applications and application-layer protocols. An application-layer protocol is only one piece of a network application. The Web's application-layer protocol, HTTP, defines that the format and sequence of messages exchanged between browser and Web server. Thus, HTTP is only one piece (albeit, an important piece) of the Web application.

2.2 The Web and HTTP

2.1 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 two 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 Javascript file, a CCS style sheet file, or a video clip — that is addressable by a single URL.

http://www.someSchool.edu/someDepartment/picture.gif

has www.someSchool.edu for a hostname and /someDepartment/picture.gif for a path name. Web servers, which implement the server side of HTTP. Popular Web servers include Apache and Microsoft Internet Information Server.

HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients. When a user requests a Web page the browser sends HTTP request messages for the object in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects.

HTTP uses TCP as its 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 client sends HTPP request messages into its socket interface and receives HTTP response messages from its socket interface. Similarly, the HTTP server receives request messages from its socket interface and sends response messages into its socket interface. This implies that each HTTP request message sent by a client process eventually arrives intact at the server; similarly, each HTTP response message sent by the server process eventually arrives intact at the client. Here we see one of the great advantage of a layered architecture — HTTP need not worry about lost data or the details of how TCP recovers or reordering of data within the network. That is the job of TCP and the protocols in the lower layers of the protocol stack.

It is important to note that 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 served 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.

2.2.2 Non-Persistent and Persistent Connections

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. Although HTTP uses persistent connection in its default mode, HTTP clients and servers can be configured to use non-persistent connections instead.

HTTP with Non-Persisten Connections

1. The HTTP client process initiates a TCP connection to the server www.someSchool.edu on port number 80. 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 message to the server via its socket.
3. The HTTP server process receives the request message via its socket, retrieves the object from its storage, 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.
6. The first four steps are then repeated for each of the referenced JPEG objects.

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 objects. Note that each non-persisten TCP connection transports exactly one request message and one response message.

We define the round-trip time (RTT), which is the time it takes for a small packet to travel from client to server and then back to the client. The RTT includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays.

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. 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 (configurable timeout interval)

2.2.3 HTTP Message Format

HTTP Request Message

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

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.

The header line Host: www.someschool.edu specifies the host on which the object resides. The information provided by the host header line is required by Web proxy caches. By including the Connection: close header line, the browser is telling the server to close the connection after sending the requested object. The user-agent: header line specifies the user agent, that is, the browser type that is making the request to the server.

An HTTP client often uses the POST method when the user fills out a form.

HTTP Response Message

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 14:44:04 GMT
Server: Apache/2.2.4 (CentOS)
Last-Modified: Tue, 18 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html
(data data data data ...)
```

It has three sections: an initial status line, six header lines, and then the entity body. The entity body is the meat of the message—it contains the requested object itself. The status line has three fields: the protocol version field, a status code, and a corresponding status message.

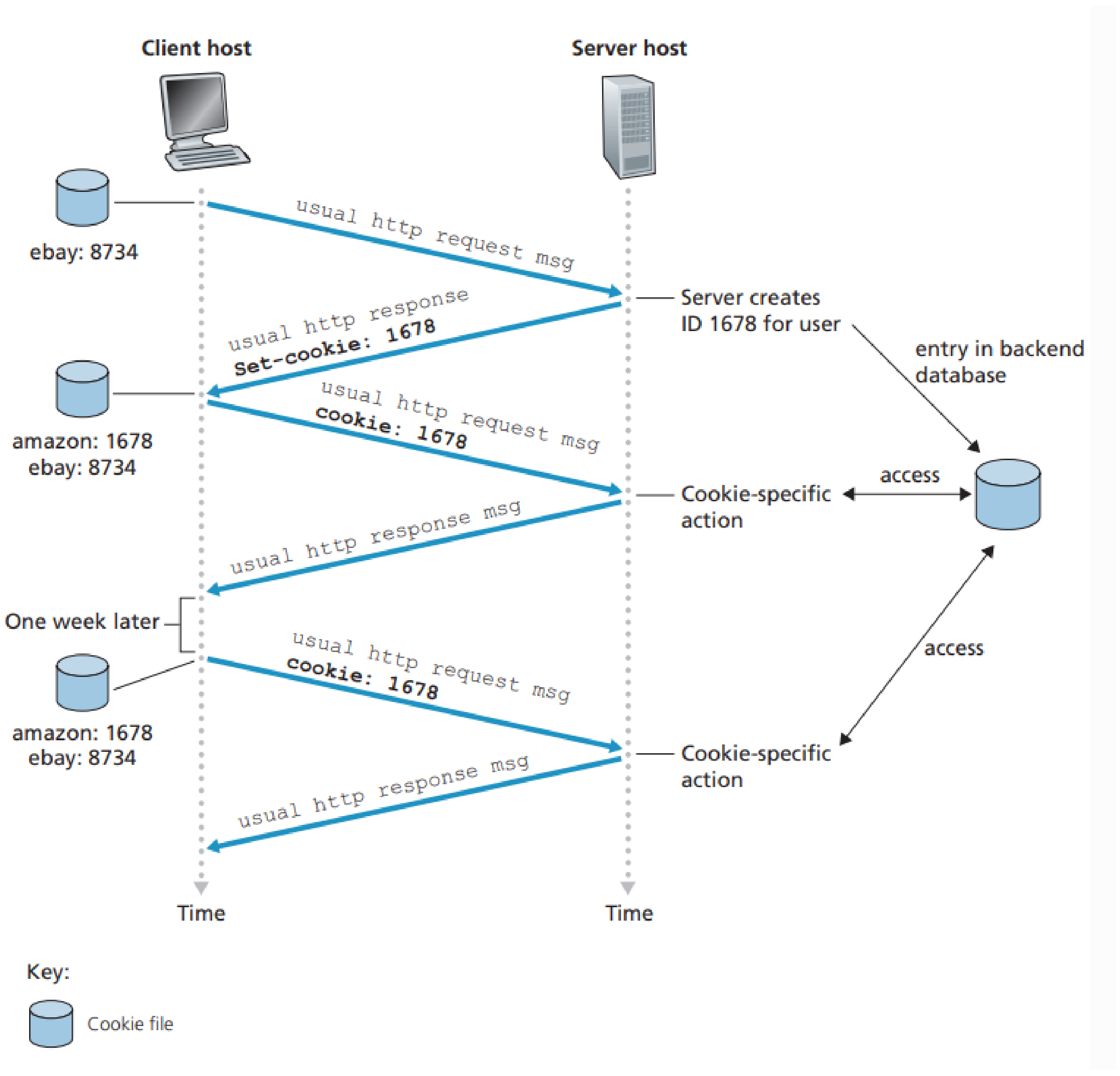
The server uses the Connection: close header line to tell the client that it is going to close the TCP connection after sending the message. The Date: header line indicates the time and date when the HTTP response was created and sent by the server. It is the time when the server retrieves the object from its file system, inserts the object into the response message, and sends the response message. The Server: header line indicates that the message was generated by an Apache Web server; it is analogous to the User-agent: header line in the HTTP request message. The Last-Modified: header line indicates the time and date when the object was created or last modified. The Content-Length: header line indicates the number of bytes in the object being sent. The Content-Type: header line indicates that the object in the entity body in HTML text.

- 200 OK: Request succeeded and the information is returned in the response
- 301 Moved Permanently: Requested object has been permanently moved; the new URL is specified in Location: header of the response message. The client software will automatically retrieve the new URL.
- 400 Bad Request: This is a generic error code indicating that the request could not be understood by the server.
- 404 Not Found: The requested document does not exist on this server
- 505 HTTP Version Not Supported: The requested HTTP protocol version is not supported by the server.

2.2.4 User-Server Interaction: Cookies

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], allows sites to keep track of users. Most major commercial Web sites use cookies today.

Cookie technology has four components: (1) a cookie header line in the HTTP response message; (2) a cookie header line in the HTTP request message; (3) a cookie file kept on the user’s end system and managed by the user’s browser; and (4) a back-end database at the Web site.



2.2.5 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. Once a browser is configured, each browser request for an object its first directed to the Web cache.

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. 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.

Note that a cache is both a server and a client at the same time. When it receives requests from and sends responses to a browser, it is a server. When it sends requests to and receives responses from an origin server, it is a client.

Typically a Web cache is purchased and installed by an ISP.

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. Second, Web caches can substantially reduce traffic on an institution's access link to the Internet. By reducing traffic, the institution does not have to upgrade bandwidth as quickly, thereby reducing costs. Furthermore, Web caches can substantially reduce Web traffic in the Internet as a whole, thereby improving performance for all applications.

Through the use of Content Distribution Networks (CDNs), Web caches are increasingly playing an important role in the Internet. A CDN company installs many geographically distributed caches throughout the Internet, thereby localizing much of the traffic.

The Conditional GET

Although caching can reduce user-perceived response times, it introduces a new problem—the copy of an object residing in the cache may be stale. In other words, the object housed in the Web server may have been modified since the copy was cached at the client. Fortunately, HTTP has a mechanism that allows a cache to verify that its objects are up to date. This mechanism is called the conditional GET [RFC 7232]. An HTTP request message is a so-called conditional GET message if (1) the request message uses the GET method and (2) the request message includes an If-Modified-Since: header line.

Note that the value of the If-modified-since: header line is exactly equal to the value of the Last-Modified: header line that was sent the object only if the object has been modified since the specified date.

We see that in response to the conditional GET, the Web server still sends a response message but does not include the requested object in the response message. Including the requested object would only waste bandwidth and increase user-perceived response time, particularly if the object is large.

2.2.6 HTTP/2

The primary goals for HTTP/2 are to reduce perceived latency by enabling request and response multiplexing over a single TCP connection, provide request prioritization and server push, and provide efficient compression of HTTP header fields. HTTP/2 changes how the data is formatted and transported between the client and server.

But developers of Web browsers quickly discovered that sending all the objects in a Web page over a single TCP connection has a Head of Line (HOL) blocking problem. Using a single TCP connection, the video clip will take a long time to pass through the bottleneck link, while the small objects are delayed as they wait behind the video clip; that is, the video clip at the head of the line blocks the small objects behind it.

TCP congestion control also provides browsers an unintended incentive to use multiple parallel TCP connections rather than a single persistent connection. Very roughly speaking, TCP congestion control aims to give each TCP connection sharing a bottleneck link an equal share of the available bandwidth of that link; so if there are n TCP connections operating over a bottleneck link, than each connection approximately gets l/nth of the bandwidth. By opening multiple parallel TCP connections to transport a single Web page, the browser can “cheat” and garb a large portion of the link bandwidth.

HTTP/2 Framing

The HTTP/2 solution for HOL blocking is to break each message into small frames, and interleave the request and response messages on the same TCP connection. Thus the HTTP/2 framing mechanism can significantly decrease user-perceived delay.

The ability to break down an HTTP message into independent frames, interleave them, and then reassemble them on the other end is the single most important enhancement of HTTP/2. The framing is done by the framing sub-layer of the HTTP/2 protocol. The frames of the response are
then interleaved by the framing sub-layer in the server with the frames of other responses and sent over the single persistent TCP connection. As the frames arrive at the client, they are first reassembled into the original response messages at the framing sub-layer and then processed by the browser as usual.

Response Message Prioritization and Server Pushing

Message priorization allows developers to customize the relative priority of requests to better optimize application performance. When a client sends concurrent requests to a server, it can prioritize the responses it is requesting by assigning a weight between 1 and 256 to each message. The higher number indicates higher priority. Using these weights, the server can send first the frames for the responses with the highest priority.

Another feature of HTTP/2 is the ability for a server to send multiple responses for a single client request.

HTTP/3

QUIC is a new “transport” protocol that is implemented in the application layer over the bare-bones UDP protocol. QUIC has several features that are desirable for HTTP.