Mr. Ali bin Tahir Email: ali@biit.edu.pk

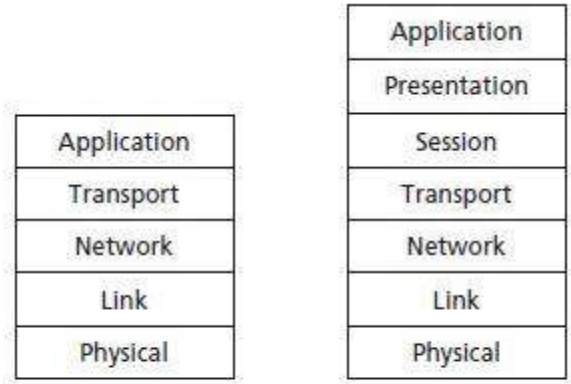
Week # 14 – Lecture # 27 – Application Layer

Recommended Reading:

Book: Computer Networking, A Top-Down Approach 6th edition, Authors: James F. Kurose, K W. Ross

Application Layer

In our last lecture, we have completed our discussion on transport layer. Today, we will start a new chapter (chapter 2 in the book) which is about application layer. It is the top layer in the TCP/IP protocol stack (and in OSI model as well).



a. Five-layer Internet protocol stack

b. Seven-layer ISO OSI reference model

Here is the outline for today's lecture:

- **Principles of Network Applications**
- **Network Application Architecture**
- **Process Level Communication**
- **Application Layer Protocols**

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. For example, 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, tablet, smart phone, and so on); and the **Web server program** running in the Web server host. As another example, in a **P2P (peer to peer) file-sharing system there is a program in each**

host that participates in the file-sharing community. In this case, the programs in the various hosts may be similar or identical.

Thus, when developing your new application, you need to write software that will run on multiple end systems. This software could be written, for example, in C, Java, or Python. Importantly, you do not need to write software that runs on network core devices, such as routers or link-layer switches. Network-core devices do not function at the application layer but instead function at lower layers—specifically at the network layer and below. This basic design—namely, limiting application software to the end systems—as shown in Figure 2.1, has facilitated the rapid development and deployment of a vast array of network applications.

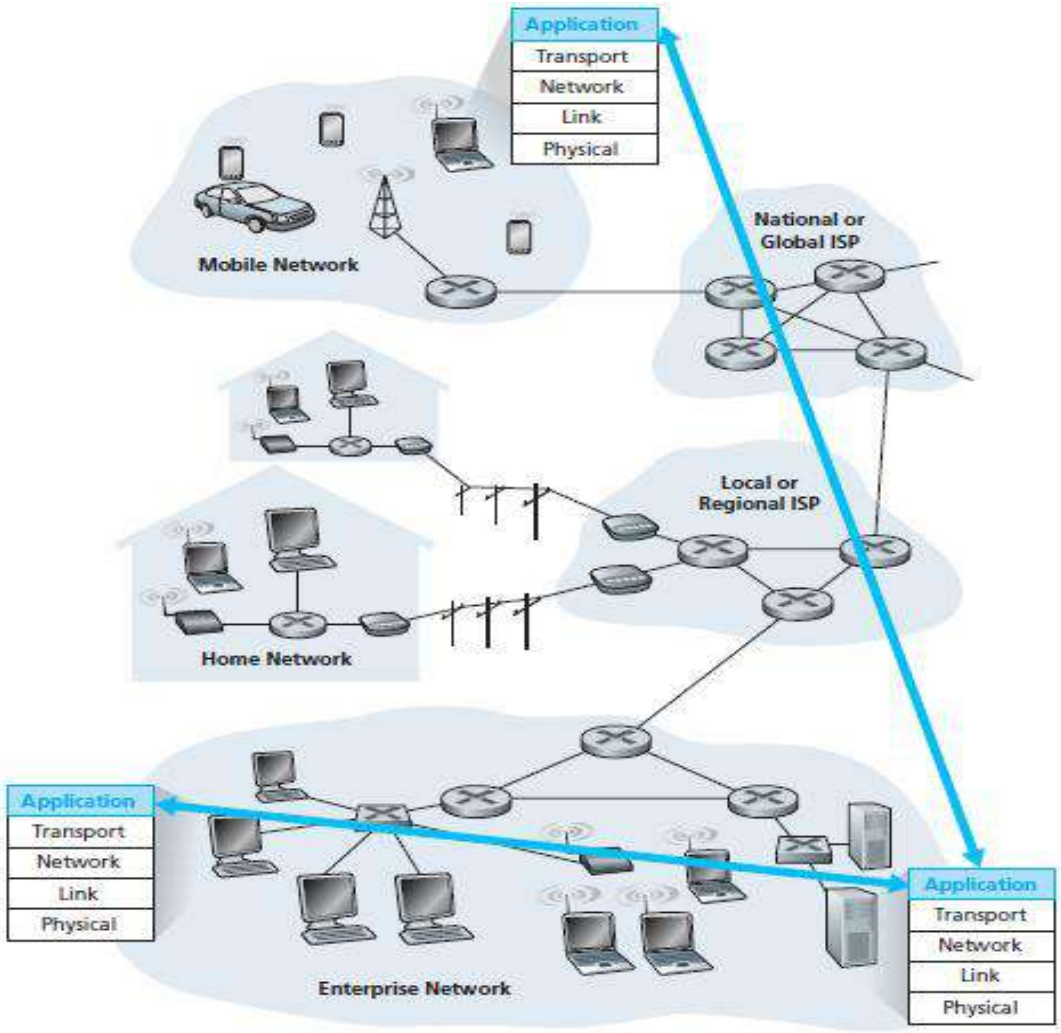


Figure 2.1 • Communication for a network application takes place between end systems at the application layer

Network Application Architecture

Before diving into software coding, you should select an architectural for your network application. Keep in mind that an application's architecture is different from the network architecture (e.g., the five-layer Internet architecture discussed in Chapter 1). From the 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**.

There are two widely deployed network application architectures:

- **Client-server architecture**
- **Peer-to-peer (P2P) architecture**

Client-server Architecture

In a client-server architecture, **there is an always-on host, called the *server***, which **services requests from many other hosts, called *clients***. A classic example is the Web application for which an always-on Web server 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**. Note that with the client-server architecture, **clients do not directly communicate with each other**; for example, in the Web application, two browsers do not directly communicate. Another characteristic of the client-server architecture is that the **server has a fixed, well-known IP 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**. Some of the better-known applications with a client-server architecture include the **Web, FTP, Telnet, and e-mail**. The client-server architecture is shown in Figure 2.2(a).

P2P Architecture

In a P2P architecture, there is no reliance on dedicated servers. Instead, the **application exploits direct communication between pairs of 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. Because the peers communicate without passing through a dedicated server, the architecture is called peer-to-peer. Many of today's most popular and traffic-intensive applications are based on P2P architectures. These applications

include **file sharing (e.g., Bit Torrent), peer-assisted download acceleration (e.g., Internet Download Manager IDM), Internet Telephony (e.g., Skype), and IPTV (e.g., Kankan and PPstream)**. The P2P architecture is illustrated in Figure 2.2(b).

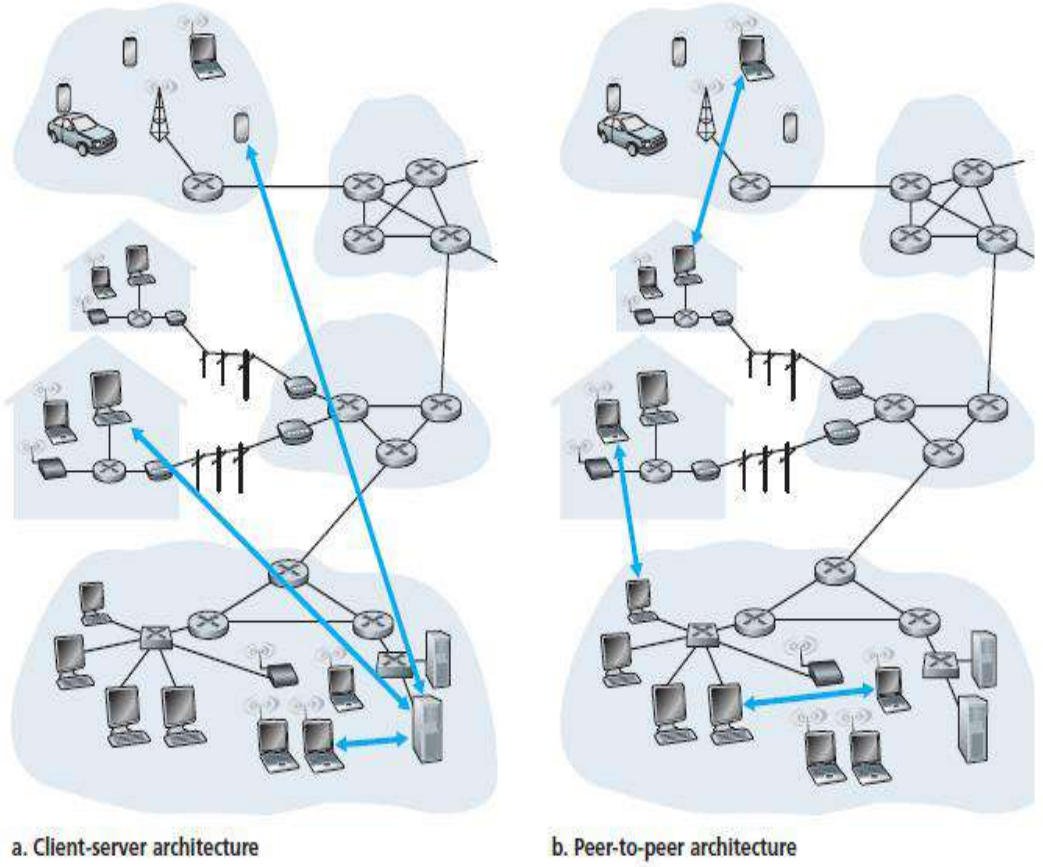


Figure 2.2 ♦ (a) Client-server architecture; (b) P2P architecture

Process Level Communication

Before building your network application, you also need a basic understanding of how the programs, running in multiple end systems, communicate with each other. In the field 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 the end system's operating system. But we are not interested in how processes in the same host communicate, but instead in how processes running on *different* hosts (with potentially different operating systems) communicate.

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. Figure 2.1 illustrates that processes communicating with each other reside in the application layer of the five-layer protocol stack.

A network application consists of pairs of processes that send messages to each other over a network. For example, in the Web application a client browser process exchanges messages with a Web server process. For each pair of communicating processes, we typically label one of the two processes as the client and the other process as the server. With the Web, a browser is a client process and a Web server is a server process. We define the client and server processes as follows:

In the context of a communication session between a pair of processes, ***the process that initiates the communication (that is, initially contacts the other process at the beginning of the session) is labelled as the client. The process that waits to be contacted to begin the session is the server****.*

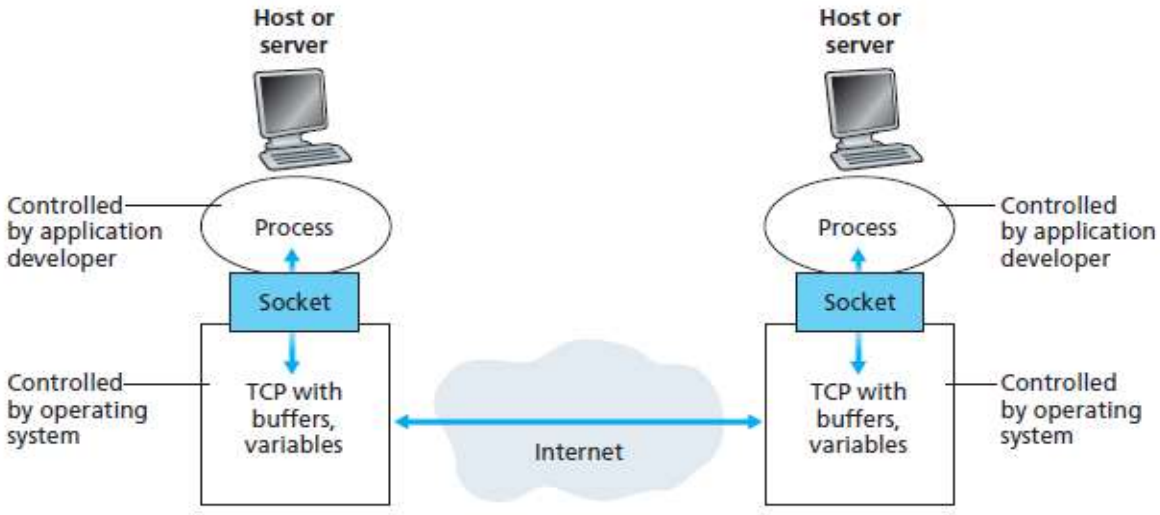


Figure 2.3 ♦ Application processes, sockets, and underlying transport protocol

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** (**Socket is also called Application Programming Interface API**). Figure 2.3 illustrates socket communication between two processes that communicate over the Internet. (Figure 2.3 assumes that the underlying transport protocol used by the processes is the Internet's TCP protocol.) As shown in this figure, **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**. Once the application developer chooses a transport protocol, the application is built using the transport-layer services provided by that protocol.

Application Layer Protocols

We have learned that network processes communicate with each other by sending messages into sockets. But how are these messages structured? What are the meanings of the various fields in the messages? When do the processes send the messages? These questions bring us into the domain of application-layer protocols. An application-layer protocol defines how an application's processes, running on different end systems, pass messages to each other. In particular, an application-layer protocol defines:

- The types of messages exchanged, for example, **request messages and response messages**.
- **The syntax of the various message types, such as the fields in the message and meaning of the information in these fields**.
- **Rules for determining when and how a process sends messages and responds to messages**.

It is important to distinguish between network applications and application-layer protocols. An application-layer protocol is only one piece of a network application. Let's look at a couple of examples.

Example 1

The Web application 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 (that is, **HTML**), Web browsers (for example, Firefox and Microsoft Internet Explorer), **Web servers** (for example, **Apache** and **Microsoft IIS**), and an application-layer protocol. The Web's

application-layer protocol, HTTP, defines the format and sequence of messages exchanged between browser and Web server. Thus, HTTP is only one piece (an important piece indeed) of the Web application.

Example 2

As another example, an Internet **e-mail application** also has many components, including mail server that keeps user mailboxes; mail clients (such as Microsoft Outlook) that allow users to read and create messages; a standard for defining the structure of an e-mail message; and application-layer protocols that define how messages are passed between servers, how messages are passed between servers and mail clients, and how the contents of message headers are to be interpreted. The principal application-layer protocol for electronic mail is **SMTP** (Simple Mail Transfer Protocol). Thus, e-mail's principal application-layer protocol, SMTP, is only one piece of the e-mail application.

SMTP is the protocol for **sending email** whether it is from the client or in between servers for propagating the email towards the intended destination. In comparison, **IMAP** is a protocol that deals with managing and retrieving email messages from the server

IMAP and **POP3** are the two most commonly used **Internet mail protocols** for **retrieving emails**. Both protocols are supported by all modern email clients and web servers. While the **POP3 protocol** assumes that your email is being accessed only from one application, **IMAP** allows simultaneous access by multiple clients.

Differences between IMAP and POP

	IMAP	POP
Flexibility	Delete or move a message without having to download it.	Must download all messages.
	Download only the body of a message.	Must download entire message, including attachments.
Synchronisation	View messages in all folders.	Only view messages from the Inbox.

	Changes made on the web and on your devices stay in sync everywhere.	Once downloaded, changes can only be made on your home computer.
	Access messages at home, work, and through the web.	Access messages only from a single device.
Backups	All messages kept with multiple backup copies on mail servers.	Once downloaded, the message only exists on your local computer. If it crashes, the message is lost.

Protocol	Non-Encrypted Port	SSL/TLS port
SMTP	**25**	465
IMAP	**143**	993
POP3	**110**	995

Week # 14 – Lecture # 28 – Application Layer

Recommended Reading:

Book: Computer Networking, A Top-Down Approach 6th edition, Authors: James F. Kurose, K W. Ross

Application Layer

In previous lectures we have introduced some basic concepts related to application layer, such as network application architecture and application layer protocols. Now we will focus on particular applications, and the first application which we will discuss today is the web application.

Here is the outline for today's lecture:

- **Web Application and HTTP**
- **Non-Persistent and Persistent Connections**
- **HTTP Message Format**

Web Application and HTTP

A Web page (also called a document) 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. For example, **if a Web page contains HTML text and five JPEG images, then the Web page has six objects**: the base HTML file plus the five images. The base HTML file references the other objects in the page with the objects' URLs. Each URL has two components: the hostname of the server that has the object and the object's path name. For example, the URL

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

has **BIIT.edu** for a hostname and **/someDepartment/** for a path name and picture.gif is a filename.

The Hyper Text 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. Web browsers (such as Internet Explorer and Firefox) implement the client side of HTTP. Web servers implements the server side of HTTP, contains Web objects, each addressable by a URL. Popular Web servers include Apache v2.4.x and Microsoft Internet Information Server v10.

HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients. The general idea is illustrated in Figure 2.6. **When a user requests a Web page (for example, clicks on a hyperlink), the browser sends HTTP request messages for the objects in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects**.

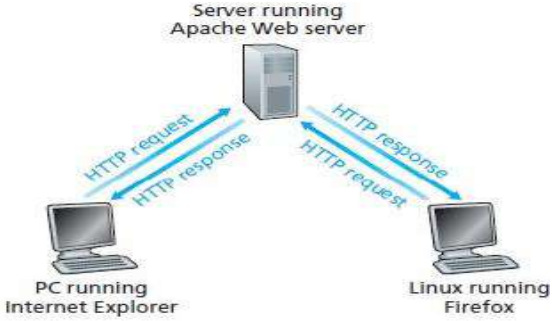


Figure 2.6 ♦ HTTP request-response behavior

HTTP uses TCP as its underlying transport protocol (rather than running on top of UDP). **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 HTTP 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**.

Once the client sends a message into its socket interface, the message is out of the client's hands and is "in the hands" of TCP. Recall from transport layer chapter that TCP provides a reliable data transfer service to HTTP. This implies that each HTTP request message sent by a client process eventually arrives at the server; similarly, each HTTP response message sent by the server process eventually arrives at the client. Here we see one of the great advantages of a layered architecture—HTTP need not worry about lost data or the details of how TCP recovers from loss or reordering of data within the network. That is the job of TCP and the protocols in the lower layers of the protocol stack.

Non-Persistent and Persistent Connections

When 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.

To gain a deep understanding of this design issue, let's examine the advantages and disadvantages of persistent connections in the context of a specific application, namely, HTTP, which can use both non-persistent connections and persistent connections. Although HTTP uses persistent connections in its default mode, HTTP clients and servers can be configured to use non-persistent connections instead.

HTTP with Non-Persistent Connections

Let's walk through the steps of transferring a Web page from server to client for the case of non-persistent connections. Let's suppose the page consists of a base HTML file and 10 JPEG images, and that all 11 of these objects reside on the same server.

Further suppose the URL for the base HTML file is

http://www.BIIT.edu/someDepartment/home.index

In case of non-persistent connections, 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. Thus, in this example, when a user requests the Web page, 11 TCP connections are generated.

Disadvantage of Non-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, TCP connection establishment takes some time (since packets are exchanged between sender and receiver to establish a connection) which causes delay in web communication.

HTTP with Persistent Connections

With 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.

HTTP Message Format

The HTTP specifications include the definitions of the HTTP message formats. There are two types of HTTP messages, **request messages and response messages**, both of which are discussed below.

HTTP Request Message

Below we provide a typical HTTP request message:

GET /somedir/page.html HTTP/1.1

Host: www.BIIT.edu

Connection: close

User-agent: Mozilla/5.0

Accept-language: fr

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 first line of an HTTP request message is called the request line; the subsequent lines are called the header lines**.

Request line

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 self-explanatory; in this example, the browser implements version HTTP/1.1.

Header lines

Now let's look at the header lines in the example. The header line **Host: www.BIIT.edu** specifies the host on which the object resides. By including the **Connection: close** header line, the browser is telling the server that it doesn't want to bother with persistent connections; **it wants 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**. Here the user agent is **Mozilla/5.0, a Firefox browser**. This header line is useful because the server can actually send different versions of the same object to different types of user agents. (Each of the versions is addressed by the same URL.) Finally, the **Accept language**: header indicates that the user prefers to receive a **French version** of the object, if such an object exists on the server; otherwise, the server should send its default version. The Accept-language: header is just one of many content negotiation headers available in HTTP.

HTTP Response Message

Below we provide a typical HTTP response message. This response message could be the response to the example request message just discussed.

HTTP/1.1 200 OK

Connection: close

Date: Tue, 09 Aug 2011 15:44:04 GMT

Server: Apache/2.2.3 (CentOS)

Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT

Content-Length: 6821

Content-Type: text/html

(data data data data data ...)

Let's take a careful look at this response message. **It has three sections: an initial status line, six header lines, and then the entity body**. The entity body contains the requested object itself (represented by data data data data data ...).

Status line

The status line has **three fields**: the **protocol version field, a status code, and a corresponding status message**. In this example, the status line indicates that the server is using HTTP/1.1 and that everything is OK (that is, the server has found, and is sending, the requested object).

Let's say a few additional words about status codes and their phrases. The status code and associated phrase indicate the result of the request. Some common **status codes and associated phrases** include:

- **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.**

Header lines

Now let's look at the header lines. 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**. Note that this is not the time when the object was created or last modified; 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 Last-Modified: header, which we will soon cover in more detail, is critical for object caching, both in the local client and in network cache servers (also known as proxy servers). 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 is HTML text.