Web Mechanisms

Draft: 2/23/13 6:54 PM

Introduction

While it is perfectly possible to create web sites that work without knowing any of their underlying mechanisms, web developers must eventually develop an accurate model of how the web functions in order to work effectively.

This document gives a model of how the underpinnings of the web work. The model omits details and ignores exceptions to the rules, but is accurate in what it does say.

Internet Protocols

Computer networks provide a mechanism for transmitting bits of information from one computer to another. The physical medium for transmitting the bits may be wires (Ethernet, telephone, etc.); radio signals (Wi-Fi, for example); fiber optic cables (such as FiOS), or any other signaling mechanism one could imagine. For our purposes we'll just assume there is some way for a computer to transmit and receive bits. With that assumption in place, we'll look at how those bits are used to encode messages, and how those messages get delivered to the intended recipient.

The basic model used by the Internet is that computers exchange *messages* with each other. These messages generally contain plain text that could be read by people as well as by computers, but to make the messages unambiguous and easier for computers to process, the messages have to adhere to quite rigid structures, known as *protocols.*

At the lowest level, messages are broken into relatively small *packets*, which are structured according to the *Internet Protocol* (IP). An IP packet has two parts: a *header* and a *body.* The developers of the Internet thought in terms of postal mail, where letters are put inside envelopes, with the envelope containing the delivery address and a return address. By analogy, the header of an IP packet is an envelope and the body is the message that goes inside the envelope.

Just like postal mail envelopes, IP packet headers contain a delivery (destination) address and a return (originator) address. In the case of IP, these addresses are binary numbers. Every computer attached to the Internet has to have a unique binary IP address so there is no ambiguity about where a message is supposed to be delivered.

There are currently two versions of IP in use: version 4 (IPv4) uses 32-bit addresses, but the network is in the process of converting to version 6 (IPv6), which uses 128-bit addresses. The reason for the change is that 32-bit addresses allow "only" 4 billion (2^{32}) computers to be connected to the Internet at a time. The way binary

numbers work, changing to 128-bit addresses doesn't just quadruple the number of computer addresses ($32 \times 4 = 128$), it multiplies the number by 2^{96} ($2^{32} \times 2^{96} = 2^{128}$), which will suffice until computers are more ubiquitous than atoms in our galaxy.

To send an IP message, often called a "datagram," from one computer to another, the sender creates a packet with the receiver's IP address and its own IP address in the header, whatever information it wants to transmit in the packet body, and sends it off to some other processor that it is connected to (either by wire, wirelessly, or whatever), which starts a process in which routers and switches examine the header and pass the packet on to another router or switch that will decide how to get the packet closer and closer to the destination computer. The routing process is complex, and individual packets can take different routes to go between the same two computers, which means they might not be received in the same sequence in which they are sent. Furthermore, the process is not designed to be reliable: routers are allowed to discard packets if it is too busy to handle them.

Because datagram delivery is unreliable, there is another protocol for delivering messages reliably that is built "on top" of IP: the Transmission Control Protocol (TCP). The sending computer breaks TCP messages into IP packets, with the body part of the IP packets containing a TCP header and part of the message body. That is, a single TCP message is typically requires more than one IP packet. The TCP header includes housekeeping information that says something like, "I am packet number *x* of *y* packets that make up message *z*." The TCP handler at the receiving computer arranges the packets it receives in the correct order, sends back requests for retransmission of any packets that do not show up in a reasonable amount of time, and sends back an acknowledgement when it has received the entire message. It's this two-way communication, with provision for handling dropped packets, that enables TCP to provide reliable message passing using unreliable IP datagrams. Because so much Internet traffic uses these two protocols, they are typically lumped together as "TCP/IP."

The notion of TCP being built "on top of" IP is the basis for the *layered* nature of the internet: the IP layer provides a way to send datagrams between computers, but unreliably; the TCP layer adds a way to send messages between computers reliably. The TCP layer puts its messages inside the body sections of the IP layer packets.

But we need yet a third layer: one that allows not just computers, but *programs* to communicate with each other, web servers and web browsers in particular. Just like the lower layers, these programs use their own protocols to exchange messages, and in this case these messages are carried inside TCP message bodies. Web servers and web browsers use a protocol called the Hypertext Transfer Protocol (HTTP). Other protocols in common use include the Simple Mail Transfer Protocol (SMTP) for email, the Secure File Transfer Protocol (SFTP) for file transfers, the Remote Desktop Protocol (RDP) for remote login using a graphical user interface, and the Secure Shell protocol (SSH) for remote login using a command line interface.

To identify a program, TCP messages include a protocol identification number to tell which program on the other computer should receive the message. The common

name for these identification numbers is *port numbers*, and they are discussed further in the section on "Operating Systems and Internet Servers" below.

The Client-Server Model

HTTP, like most program-level Internet protocols, is based on what is called the *client-server model.* In this model, one program, called the client, sends a *request message* to another program, called the server, which sends a *reply message* back to the client. Once a request has been sent and a reply has been received, the message exchange is complete. Unlike a telephone conversation where you stay connected to the other party until one or the other of you hangs up ("breaks the connection") the internet is built on this "connectionless" model in which each message exchange is independent of any others. As you can imagine, this design makes some Internet activities tricky to build. For example Voice over IP (VoIP, like Skype) has to provide a continuous connection between the two sides of a telephone conversation even though the speech sounds are actually being delivered using IP datagram packets; VoIP uses IP, but it doesn't use TCP.

In the client-server model, the client has to have the server's IP address and port number in order to send a request to it. The server can then use the return address in the message to send its reply back to the client. To be complete we'll mention that the client program can use an arbitrary port number for receiving the reply: it sends its port number along with its IP address in the TCP header.

The terminology is that the server must use a "well-known" IP address and port number, meaning simply that there has to be some way for clients to know where to send their requests. We'll come back to this topic in the section on "Client Requests."

It's worth noting here that "getting a web page" can require many request-reply message exchanges between the client (a web browser, such as Internet Explorer, Firefox, Chrome, or Safari) and the server. For example, the browser would start with a request for an HTML document, and then, when it examines the code in that document, might find links to stylesheets, JavaScript code files, and image files, all of which have to be fetched from the server using separate request-reply message exchanges.

Operating Systems and Internet Servers

Microsoft Windows, Apple's OS X, and Linux are all operating systems that support both HTTP clients (browsers) and HTTP servers. In this section we'll look at how these operating systems support HTTP servers. There are two widely used HTTP servers, but one of them, called IIS, runs only on computers running the Windows operating system. Instead, we will use the Apache HTTP server as our example: it is freely available, runs on all three operating systems, and is more widely used than IIS.

When you use a laptop or desktop computer, you are familiar with the idea of running more than one program at a time, say a word processor, a web browser, and a music player. Likewise, server computers also typically run several "service"

programs at the same time, such as a web server, a mail server, a file server, and likely several other programs that all use the internet to communicate with remote clients.

To manage the task of running several programs at a time, operating systems handle two important tasks: (1) allocating resources, such as memory and access to the central processing unit (CPU), to the different programs as they need them, and (2) providing the low-level code needed for performing input and output (I/O) operations. It's a particular set of I/O operations that we are interested in here: receiving request messages from clients and sending reply messages back to them. With multiple server programs running at the same time, the operating system needs a way to know which server is to receive each request message that comes in over the network. Getting a server's reply messages back to the client is somewhat less complicated, but the client computer has the same issue to deal with: how to get incoming network messages to the correct problem.

Server programs typically start running when a computer is turned on and continue to run until the computer is turned off again. When a server program starts up, it tells the operating system that it wants to receive messages directed to a particular *port* (the protocol identification number mentioned above.) Each protocol has a standard port number that it uses. For HTTP, it's port number 80. For SMTP it's port number 25; SSH uses number 22; HTTPS (HTTP, but with encrypted request and reply messages for security) uses number 443; etc. The operating system will not allow more than one program to "listen" to a given port number.

There is a minor point to make here, which will show up in the next section when we deal with URLs. It is possible to run multiple copies of the Apache HTTP server program on the same computer simultaneously, but only one copy can use port 80. The others would have to use a different port number, even though they are running the same protocol.

Client Requests

So there are well-known port numbers associated with different protocols, such as port 80 for HTTP. But how does a server's binary IP address become "well known" enough for a browser to be able to direct our request to the proper host (server) computers? Google's IP address is 10101101 11000010 00101011 00100010. How does the browser know it should use that address when we click on "google.com?"

The answer is yet another Internet protocol called the Domain Name System (DNS) (which uses port number 53 if you're interested). There are computers connected to the internet that run DNS servers that know how to handle request messages containing a *fully qualified domain name* (FQDN), like google.com or babbage.cs.qc.cuny.edu. A DNS server sends back reply messages with the corresponding binary IP addresses.

As an aside, people sometimes have to deal with binary IP addresses, and there is a convention to break the binary numbers into smaller groups, which can be written in decimal (IPv4) or hexadecimal (IPv6). For example that binary address for Google

can be written using decimal numbers for successive groups of 8 bits. 10101101 in binary is 173 in decimal; 11000010 is 194; 00101000 is 43; and 00100010 is 34, so Google's IP address in "dotted decimal notation" is 173.194.43.34, which is easier for humans to deal with than the full 32-bit binary number. (Nobody says "easier" means "easy!")

If you think about it, computers can't use DNS to get the IP address of the DNS server they want to contact: those IP addresses have to be known, and entered into the operating system when its network connection is first set up. If you poke around in your computer's network settings, you will find the IP addresses of at least one DNS server stored there in dotted decimal notation.

URLs

The HTTP protocol uses Universal Resource Locators (URLs) to specify web servers, port numbers, and individual web pages. The full structure of a URL is quite complicated, but we will "parse" an example to see how they are structured in general:

`http://babbage.cs.qc.cuny.edu:80/courses/sample.php?name=vickery&passwd=secret`

`http`	The name of the protocol. Another example would be `https`.
`://`	Characters to separate the protocol from the next part.
`babbage.cs.qc.cuny.edu`	The FQDN of the "host" computer running the HTTP server.
`:80`	The port number of the HTTP server running on babbage.cs.qc.cuny.edu
`courses/sample.php`	The *path* to the resource being requested.
`?`	A character to separate the first part of the URL from the next part.
`name=vickery&passwd=secret`	A *query string* to be passed to the resource being requested.

There are several common variations on this structure: if the protocol name and/or port number are omitted, browsers will supply them for you automatically; the query string may be omitted if it is not needed or used, or it can be hidden inside the body of the HTTP request to keep it from appearing in the address bar of the browser; the *path* is normally a file system (disk) path to a file, but if it is omitted or ends with the name of a directory rather than a file, the HTTP server can be configured to produce the "correct" resource to return.

Some HTTP request types (GET, POST, and HEAD) request headers, (cookies and accept) and caching

HTTP Request Processing

Static and dynamic (scripted) pages. Where's the database?

Response headers and the response body.

HTTP Response Processing

The concepts of incremental rendering and caching.

Summary