Website Architecture

Lezione 7

The Client-Server Model

An introduction to servers and a generic look at the capabilities of a Web server, including HTTP requests and responses.

Michael Serritella

Summer 2010

Intro to Servers

So far, we have worked with websites that reside on the same computer as the browser. This is *not* usually the case. Servers are the machines that store the websites and then deliver them, one page at a time, to people who request them over a network such as the Internet.

Anatomy of a server

Servers are different from regular desktops and laptops in terms of both hardware and software, but you can still apply the vast majority of your computer science knowledge to servers. Simply put, the key characteristic of a server is that it is listening all the time, always ready to provide a service.

Software differences

Let's say that a server needs to listen for requests for Web pages, images, CSS files, etc., and send them back over the network. And let's start from a desktop computer and examine the differences. A desktop could do this, but do you really need a desktop? Do you need.. a mouse? A GUI? A printer? A webcam?

Servers have operating systems similar to personal computers, but they are different flavors of those operating systems that have stripped out the support for these unnecessary things. On top of that, they may add modules to manage lots of simultaneous requests more efficiently, to enforce different security policies that servers might typically have (large-scale virtualization for multiple users), etc. But if you have Ubuntu on your home machine and Ubuntu Server on your server, you will find that they are conceptually compatible in most ways.

Worst-case, though, you could still run a Web server from a good desktop with a typical OS - or a cluster of old desktops - as long as the demand for their services is not too high. It's simply a matter of economics that we usually use specialized machines for this.

Hardware differences

Seeing as how you don't need a mouse, printer, sound card, and probably even a keyboard, servers may look different because their internals are more streamlined for the remaining components that they do need. They are architecturally very similar, however. Servers are heavily parallelized, so they will have multi-core CPUs, but so does the modern personal computer; the differences in that regard are getting smaller.

At advanced levels, you will want to consider the hardware characteristics of a server to make informed decisions on software algorithms. Web servers in particular will try to service as many people as possible - they aim for very high throughput, even if the average processing speed is a little slow. The speed bottleneck of the user experience is likely the network delay if the algorithms are not too terrible. So, raw CPU speed is not extremely important. Each program running on a server should try to minimize the RAM it uses, since they will all occupy space simutlaneously and this is a significant limiting factor in the number of concurrent users that the machine can service. So, keep in mind when designing algorithms that you should keep your RAM usage relatively low; streaming algorithms are strongly preferred over aggressively-buffered algorithms.

Software differences again

The term "server" is not well-regulated.. It can be used to refer to the hardware ("a Cell Blade server"), the OS ("Ubuntu Server"), or a particular program running on the server that performs the principal task ("Apache HTTP Server"; "I'm running a server on that computer."). In this course, we are most interested in the last one - the software which makes it a server.

Intro to Web page requests

Say that you want to download **http://www.CIS4930.com/index.html**. You first need to find the server that has this index.html file on its hard drive. We'll get to that. Then, you have to ask the server for that specific file. The interesting thing is that you don't communicate directly with the operating system and ask it for a file path. You cannot just reach into the filesystem in that way. You ask a particular program - a Web server - for one of the files that it manages.

This actually opens up a world of possibilities; the program has autonomy over what to do with that request. It could..

- Honor the request; send the contents of the file.
- Reject it ("directory is private").
- Return an error message; return a message that redirects the user to another location.
- Give back an entirely different file!
- Run a program and return the output of the program!

Basic networking

In order to go further, we should see an introduction to networking. If you want to request a Web page, how do you find the location of the server? How do you talk to the specific program running on the machine? What does your request look like? We'll look at networking in general in order to begin to explain these answers.

Communication paths

How do you reach across an Internet? You have to find a pretty good path from your computer to the destination computer. This involves "hopping" on several waypoints, including even some smaller computers along the way. While your message is being transmitted across the Internet, anyone along the way could technically read it if they were to configure their software in a malicious manner. You never know what path your messages will take, and you never know whether people will be listening. Therefore, you have to assume the worst case: all transmissions are publicly readable, unless you employ cryptography to encode and decode them.

Out of curiosity: How do you even know the physical location of **http://www.CIS4930.com**? Servers called Domain Name Servers (DNSes) are different types of servers - their sole purpose is to provide the answer to that question. They keep track of domain names such as "CIS4930.com" and map them to IP addresses, like "74.50.13.8", which help find the physical location.

Your computer (and/or browser) has a few of these DNSes hard-coded, and from there, they can figure out the rest. After your browser has resolved an IP address, it will cache the IP address. So, if you have never been to a domain before, it will be slower the first time, since browser has to find the physical location of the server. This is one reason why you do not want too many subdomains or different domains in your Content Distribution Network (CDN).

Program-to-program communication

So, you've found a physical computer; you've found the server. How do you talk to the specific server program you want? How do you talk to the Web server? When you connect to the server, on top of the IP address, you must specify a port number. The port number corresponds to a program running on the computer. When the operating system sees a connection for a certain port, it forwards the connection to the program that has claimed the port.

There is no strict correspondence of port numbers to program types, but when a program runs, it can try to claim a port number. By convention, Web servers run on port 80. Other examples: secure Web connections (HTTPS; SSL) run on port 443. FTP is on 21. If you want to specify a server address with a port, you specify it like this: "74.50.13.8**:80**", "74.50.13.8**:443**", "74.50.13.8**:21**", etc.

Message formats

Messages sent over the Internet are broken up into fixed-size pieces [1]. Networking technology is layered, so that some machines and programs are only concerned with a particular layer of abstraction - i.e. how to get the message across the country;

[1] At one layer or another; sizes are at least bounded above.

how to get it to the right process running on the computer. Each layer wraps the messages of the upper layers into its own message format, so you have a sort of matryoshka doll of messages. Each of them has their own metadata. The top layer (most abstract) is concerned with "packets", for instance.

There are two important results here. First: Packets have a maximum size, so messages may be broken across packets (typically around 1,500 bytes per packet). We may see later that you don't want to send a 1,600-byte message and waste time and space. Second: There is a layered separation of information. Messages are conceptually put into envelopes, which are put into envelopes, each of which contains routing information and so forth. The operating system opens the last "envelope" of the message when it handles the packet and decides which process will receive it. When the process receives the data, it receives the contents of the last envelope, not the envelope itself, so it no longer has information about routing[2].

[2]Unless you include some of the relevant data in the message itself, as in HTTP.

The Hypertext Transfer Protocol

Once you are connected to the server program, now what? You have to speak in a language that it understands. You have to make your request in a conventional manner, and it will give you a response in a conventional manner.

The convention of the Internet for Web pages is Hypertext Transfer Protocol (HTTP). Minimal HTTP messages are extremely simple, and even the more advanced options are not too involved.

You can see a few examples online[3] [4], as well as the **standard**. Fortunately, the Firefox extension Firebug also shows you a great view of the raw HTTP requests and responses; this is probably the easiest to get going.

[3] **What's in an HTTP Request?** : Examines request for the page itself, with general explanations for each field.
[4] **web-sniffer.net**: Examine HTTP requests and responses for any URL.

Getting started

You may want to write a "raw" request, without the help of a browser, and you may want to read the "raw" response, including headers. There are a few ways to do this, and undoubtedly, there are new Web apps that would let you do this (and expose your requests & responses to whoever wrote the Web app). But there are some old-standby tools on the Unix console that you may want to use instead.

The most famous old battleaxe is **telnet**. Using telnet, you can get a basic connection going between one application (telnet) and another (a Web server), sending the plain bytes back and forth. You would invoke it like this:

```
telnet www.example.com 80
(now inside the telnet prompt..)
GET /subfolder/whatever.html HTTP/1.1
Host: www.example.com
(newline a couple times)
```

You might like to use the program **nc** instead, since you can redirect input to the program with "<"; with telnet, you literally have to type it out. So you can do this:

```
nc www.example.com 80 < preparedReq.txt > theResponse.txt
```

Anatomy of a request

A request is all in plain text, and only the first line is required. This is sometimes called the **request line**. The first line simply gives the name of the resource (file) requested, along with some other things that we'll see. Here's an example:

GET /articles/news.html HTTP/1.1

The first word ("GET") is the **request method**; then there's the resource relative URI; then there's the protocol name and version.

After the first line, there may be any number of extra lines which pass along extra information, mostly about the client and the client's browser. These lines together are called the

"the request headers"; this term is sometimes used vaguely to include the request line. Most requests have the request line and a few headers. We will see later how a request may contain a "body" as well.

Looking ahead, you should be especially interested in headers for two reasons. 1) They tell you what information is sent along with a request, and information management in general is important; you should know what data is being sent to whom. 2) We mentioned how the Web server may decide run a program and send its output as a response to the request. Of course, you are going to be writing those programs. These are the inputs to your program!

Headers each appear on their own line(s) and have the following format:

```
Header-Name-actually-case-insensitive: Header value
```

Let's see a small subset of some of the most typical and useful headers.

The Host header

Quick note: Nowadays, practically all requests to real web servers will have at least one header: the `Host` header, which simply has the value *`www.example.com`* or whatever domain name is used. So if you're wondering why your homegrown request isn't working, that may be it.

Client information

The browser sends a long string that identifies the browser and perhaps even more information about the computer, like the operating system. The browser information is similar to the data provided in the DOM via the **navigator** object. These strings are typically so long and specific (including things like the build number) that you wouldn't compare it for equality with an expected value; you would search for substrings within them. The string likely provides the layout engine as well as the browser, which is nice.

Example (*ellipsis added*):

```
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US;..
      rv:1.9.0.8) Gecko/2009033100 Ubuntu/9.04 (jaunty)..
      Firefox/3.0.8
```

These are called "user-agent strings" or "UA strings". You can get a feel for some typical values at **UserAgentString.com**.

MIME types

Let's take a small detour. Back when email was invented, people wanted to be able to attach binary files with the email. And it became helpful to preface the attachments by stating the name and type of the file. So, those in power decided to write a standard list of strings that identified different file types. You've seen them before: "image/jpeg", "text/javascript", "text/html", "application/xml", and more. These are known as **MIME types**. Or they were, at least; they have since been re-branded as "Internet media types," though "MIME type" is still very common. They are now pervasive in personal computing, including websites and even operating systems.

Client preferences

Now, guess what? MIME types are incorporated into HTTP requests. When making a request, the client may send along the MIME types that it can handle, so the server may make an informed choice of what to send back. For example, the client may request a Web page, and it may expect a response format of HTML, XHTML, or even general XML. So it puts this expectation into the request. It is up to the application developer to actually serve variable content - most every site just serves the same content no matter what - but it is possible to make an informed choice. Along with the possible media types, the client also sends a preference indicator, called a "quality factor" (or "Q-value"), which should help sort the list.

Example (*ellipsis added*):

```
Accept: text/html,application/xhtml+xml,..
      application/xml;q=0.9,*/*;q=0.8
```

In addition to the preferred media type, the client will also send a preferred set of languages and text encodings; you can see this in `Accept-Language`, `Accept-Encoding`, and `Accept-Charset`. We will see `Accept-Encoding` later.

Refer to the standard's section on this topic for a more thorough(ly messy) explanation. Note that there are libraries which can decipher this list for you.

Referring page

Perhaps one of the most interesting headers is `Referer` [sic - spelled wrong in the standard]. This tells you the URI of the page that the user *came from* in order to get to the requested page. If the user went to the page directly (like with a bookmark or by typing into the address bar), then the `Referer` is typically omitted; maybe it's just blank. Everything about HTTP headers is browser-dependent, since the browser ultimately writes them and may do it any way it wants, so you cannot depend on this. Hey - perhaps a bare-bones browser does not even send `Referer`; it's the designer's choice. But `Referer` is good to know. Servers can build logs of which referring pages are the most common, for instance, and they can do it on a per-resource basis (for each page, image, etc.).

Authenticity of headers

None of the header information should be trusted for a correctness-critical feature of your site. A browser typically writes these headers, but any home-grown program may connect to a Web server at port 80 and send a custom request. There are even Firefox extensions which let you spoof these headers and write whatever you want; they've been around for years. The headers are accurate 99% of the time, since most people don't spoof their headers, so you can still use them for value-added features, like referrer tracking for fun and statistics (*not* for proof of site membership or other authorization), for decent browser identification, etc.

In addition to the dishonesty of the client, you have to worry about the incompetence of the browser. Some browsers incorrectly report their preferences for content type - for example, saying they support HTML and XHTML equally, when this is not the case. Some browsers, such as browsers on embedded systems, may masquerade as other browsers via their UA string, in hopes of getting a content format that was meant for the other browser. Simply know that this is not to be trusted, and if you depend on any of these features for your site, look up what happens in the most common cases (e.g. what Internet Explorer sends).

Miscellaneous: URL encoding

If you look at the first line of an HTTP request, you notice that the fields are separated by spaces. So, what if a Web address contains a space? It can't! And there are a few other characters it can't contain. To get around this, there is an encoding scheme which simply translates these special characters into a sort of escaped form. It is known as **URL encoding**.

Anatomy of a response

When the server accepts a request, it then sends a response. The response may similarly contain metadata, and it may even contain important information, like error messages, instead of the expected file. There are typically more headers in the response than in the request. The format of the header is the same, and this time, we have a message body. The header and body are separated by a blank line. After the header, the contents of the requested file (or program output, etc.) are sent over the wire.

Status code

The first line of a response gives the "status", which includes a numeric code. If you've received a "404 error" or "500 error", you've seen a status code. The code for success is 200. Let's start with an example:

```
HTTP/1.1 200 OK
```

The most important part, of course, is the "200 OK". The "OK" - the reason phrase - is nonstandard and unimportant; the numeric code is universally recognized. There are standard code numbers that sparsely occupy the range of 100-505. You may view **the standard** for more.

The error messages in particular bring up an interesting topic. We've all seen a few different styles of 404 errors. Some of them even have browser styles and logos. So: Who writes a 404 error page? The browser? The server? A site author?

Well.. all three. When the browser receives a 404 status code in the response headers, it knows that the page is not found. If the server continues to send a message body, then the browser will display that message body, because, hey, maybe it is helpful. Here, there are two possibilities: The server program could have written a standard 404 message, or the server could be configured to show a certain page upon a 404. This example may help show you how the server can intervene and change the user experience.

In this case, the most important thing for the browser is the status code; it may remove that URI from the browsing history or the cache or something else, because it treats the address as invalid.

As shown in the **standard**, the status code may also be used to redirect the user to another page, among other things.

Content type and disposition

While the client may use a set of MIME types in writing the request, the server uses a MIME type to specify the type of the response. It is much more important in the response, of course, since it may be anything from "text/html" to "application/zip".

Examples:

```
Content-type: image/png
```

```
Content-type: text/html; charset=utf-8
```

Some header (sub-)values may be separated by a semicolon. The types of Web pages should be amended with their character encoding, which is basically always UTF-8[5].

Now - you know that if you go straight to the URI of an image, you get the image rendered in your browser. But every once in a while, you see a prompt to *download* the image. Why is that? There was an extra header in the response which told the browser to do so. This is the `Content-Disposition` header. Here is an example:

```
Content-Disposition: attachment; filename="cats_353.jpg"
```

[5]The uppercase name is the official one, but the standards organizations tolerate it in lowercase within this context.

If you include this header (with "attachment"), then the file will prompt a download with the given name, which overrides any file name implied by the request.

Content length and transfer encoding

The header `Content-Length` gives the length of the content in bytes. Hey, that's great. Example:

```
Content-Length: 14276
```

But what about the case when the length is not known in advance? Perhaps part of the requested page is a news ticker from elsewhere, and we don't know how long that is. For these purposes - and others, in general - there is the "chunked" transfer format. This allows the server to send the response in pieces, where each piece (chunk) has some mini-header information (basically the length), then the content itself, then a mini-footer, and then a blank line. You can see a nice, quick summary **here**.

This has some strange consequences in rare cases. If you are writing some low-level tools to process HTTP requests & responses (i.e. "roll your own"), you will need to assemble these chunks into the proper response body. Otherwise, you will see these mini-headers and mini-footers in the middle of your response, which might look like just a couple hex digits ("3a") that shouldn't be there.

Compression

One great benefit of having a Web server as a middleman is that you can compress the content you send, using gzip. The server sends the content compressed, and the browser decompresses it and renders it. We will see the details when we see specific server software, but you should know that it is possible and how it is communicated. When looking at HTTP requests, we saw the `Accept-Encoding` field. This indicated whether the browser is able to accept compressed content. Most modern browsers do, and the *de facto* standard encoding is gzip. You can roughly expect 50% compression, which is very nice. And it's a streaming algorithm, which is best for Web. How does it work?

The server can compress content like CSS files, JavaScript files, markup, and possibly images, and it can keep extra copies of them (e.g. myStyles.css.gzip). Then, when the request comes in, it serves the contents of that gzipped version. And it adds a header to the response:

```
Content-Encoding: gzip
```

And the `Content-Length` field is given, which is the gzipped length. So, that's great, too. But what if the length is not known in advance? The entire file would have to be gzipped before the length is known. Fortunately, chunked transfers allow us to gzip on the fly and send out chunks when they're ready.

Finally, would you really want to do that? Compressing static content makes a lot of sense, because the compression only occurs once and saves everyone time and energy. But it might not make sense to compress everything on the fly, especially for dynamically generated content that would only be written once and read once. Perhaps if you have some very long document that is conducive to gzip compression (lots of pattern-based text?). Run tests to determine whether this is right for your site. More universally, however: you probably don't want to compress images, such as JPEGs, since they are already compressed in JPEG format and there probably isn't much of an opportunity to find patterns and redundant information in that file.

Caching and concurrency

We alluded to this a few lessons back, and now we get to see how it works: caching. It helps if the server gives the browser some hints about the file it's serving, so that the browser knows when it should evict the file from its cache.

There are actually a few different *types* of hints. You can tell the browser exactly when the file will expire, which is a "hard" limit, in that the browser should not re-fetch the file until that date. Or you can tell it when the file was last modified, which is a more "soft" limit. Or you can do both.

The fields **`Expires`** and **`Last-Modified`** are used for this purpose. See them in the standard here and here.

Finally[6], we have `Cache-Control`. See the standard here. This gives directives to the browser that it "MUST" obey. Of the most interesting values are *public* and *private*. This introduces yet another topic: proxies. In this context, a client's proxy is actually the Internet service provider. The ISP maintains its own cache, and it's in the ISP's best interest to cache common things, like Google's logo, to avoid going out to the Internet and fetching them a million times per day. However, this means that the same thing served to you may be shared for someone else, and that won't work if the content is private for you. Using `Cache-Control`, you can specify whether the content in the HTTP response is eligible for caching for the public.

Ninja mode?

Check out the value of *no-store* for `Cache-Control`. Avoid the thought police!

[6]For our current practical purposes

Performance tuning in general

Google has a comprehensive site which gives tips for increasing speed. Some of them are ludicrious in that they break software-engineering principles, but most of them are quite useful. Knowledge of HTTP is required for many of their optimizations, and now that you know enough, you can begin reading; read Google's tips at **Let's make the web faster**.

Submitting information to the server

Once in a while, you want to help pollute the Internet yourself. You want to send some data along with a request. The canonical example is with an HTML form, which we haven't yet seen in detail. Any way you submit the data, the data generally has "**name**=value" syntax, and multiple names are strung together into one string, like "**name1**=value&**name2**=value..".

Once you pass data in this way, it is directly accessible by any program that is running via the Web server - like your PHP program.

There are actually a few different types of requests. We saw earlier that "GET" was the first word of our example request. This indicates the request method. Let's see the three most common types. We will see more possibilities in later lessons, such as when we learn about Web services.

GET requests

GET requests are for when you get stuff and you don't even want to send any stuff. Almost. If you have the name/value pairs as before, you can actually amend them to the URI of the request and then request something like this:

somepage.html?**one**=avalue&**two**=asdf

Note the question mark. Everything after the question mark is called the **query string**. You can generally call "one" and "two" your "GET variables", and people will know what you're talking about. And you can also do something like this:

somepage.html?**island**&one=avalue..

This is possible, although it's not really any more useful than the standard notation. It's like **island** is a variable with an empty value.

GET requests with a query string are supposed to use URIs that don't necessarily have a permanent meaning. Therefore, browsers probably won't cache anything with a query string - even an image or whatever else.

POST requests

With a POST request *in a typical website setting*, the name/value string is actually the body of the request[7] The body of a request occurs in the same place as the body of a response. The POST method is semantically designated for requests in which you want to affect the state of the server (non-idempotently, to be exact). Therefore, the browser may handle them differently, like barking at you whenever you try to refresh a page which is the response of a POST request. This semantic difference is actually one of the most important differences between GET and POST.

POST can be nice, since the data doesn't show up in your browsing history as part of the URI and thus it is generally hidden from programs that only scan the headers, such as programs that keep a log of the Web requests.

[7]In atypical settings, the body may actually be any format, such as a JPEG file; this format is not dictated by the standard. We will see this with Web services.

GET wins for convenience

Sometimes, you're working in the confines of homebrew program - or some quick shell script - and you can't exactly write the request yourself, including custom headers and a body. You can just ask for a URI, and some application framework or some function will provide you with the data of the response. So, if you need to retrieve some custom data from the Internet and you can only give an URI, you can effectively build a GET request and inject some variables into the URI. This has some benign and malicious applications.

Miscellaneous: HEAD requests

If you send a HEAD request, then the server will only reply with the header of the response it *would have given* for a GET request. This can be used to poll and see if a resource gives a 404 or a redirect, for example, or perhaps check the most recent modification date.

Cookies!

As you may begin to see, there is no ongoing dialogue with the server once the page has finished loading; each request is encapsulated with its own headers and everything, even if you are requesting 50 JPEGs from the same site[8]. You can't exactly communicate with the server in a duplex manner; you can't exactly stay connected between page loads. This is because the WWW is "stateless". Or at least it uses stateless connections for practically all Web pages. We would like to be able to keep data persistently, like to maintain a record that the user logged in five minutes ago and doesn't need to log in for every page. How can we fake this kind of persistence?

A cookie is a text file stored by the browser on the user's hard drive. It keeps persistent data that is attributed to each site which writes a cookie. The text file merely contains these name/value strings, plus perhaps some parameters for cookie characteristics, so it is pretty simple and pretty familiar. But it is very useful; it opens up a world of possibilities. Cookies are not necessarily evil, but they can be annoying when they

[8]HTTP 1.1 allows persistent connections so the browser can make several requests with one connection, but this does not change the traditional model of information exchange.

fall into the wrong hands.

Cookie data is read & written within the HTTP headers. The response contains a header like this:

```
Set-Cookie: invasive=yes
```

And the next request will look like this:

```
Cookie: invasive=whynot
```

Cookies can be specificially limited to particular subdomains and folders of a site, and they may have expiration times (number of seconds to live). Setting a time-to-live of zero means that the cookie expires when the browser closes.

Server-side programs, like PHP programs, can set cookies. In addition to that, on the client side, JavaScript can set cookies; see **document**.`cookie` here.