PART I
Front End

- **CHAPTER 1:** A Refresher on Web Browsers
- **CHAPTER 2:** Utilizing Client-Side Caching
- **CHAPTER 3:** Content Compression
- **CHAPTER 4:** Keeping the Size Down with Minification
- **CHAPTER 5:** Optimizing Web Graphics and CSS
- **CHAPTER 6:** JavaScript, the Document Object Model, and Ajax

1

A Refresher on Web Browsers

WHAT'S IN THIS CHAPTER?

- Reviewing web browsers and the HTTP protocol
- Understanding the steps involved in loading a web page
- Getting to know Keep Alive and parallel downloading

To access a website, you need a web browser — the piece of client-side software that requests resources from a web server and then displays them. Web browsers are one of the most important pieces of software in the modern Internet, and competition between vendors is fierce — so much so that many vendors have chosen to give their browsers away for free, knowing that an increased share of the browser market can indirectly reap profits in other areas.

Although such competition is good news for consumers, it can be a different story for web developers, who must strive to make their sites display correctly in the myriad of browsers, each of which has its own idiosyncrasies and nonstandard behavior. To understand how this situation has evolved, let's begin by returning to the early days of the World Wide Web.

A BRIEF HISTORY OF WEB BROWSERS

Although Mosaic is often thought of as the first web browser to hit the market, this isn't actually true — that honor falls on WorldWideWeb, a browser developed by Tim Berners-Lee in 1990 at the same time as he developed the HTTP 0.9 protocol. Other browsers soon followed, including Erwise, ViolaWWW, MidasWWW, and Cello — with Cello being, at this point, the only browser for Microsoft Windows. The year 1992 also saw the release of Lynx, the first text-based browser — the others all utilized graphical user interfaces (GUIs).

In 1993, Marc Andreessen and Eric Bina created Mosaic. Although Mosaic was not as sophisticated as its competitors, a lot of effort had gone into making it easy to install. And it had one other big advantage. Previous browsers had mostly been student projects, and as such, they often floundered after the students graduated. On the other hand, Mosaic had a team of full-time programmers developing it and offering technical support. Thanks to some clever marketing, Mosaic and the web were starting to become linked in the minds of the public.

In 1994, a dispute over the naming of Mosaic forced a rebranding, and Netscape Navigator was born. Unfortunately, regular changes to the licensing terms meant that, for the next few years, there was ongoing confusion over how free it actually was.

Microsoft entered the market in 1995 with Internet Explorer (IE) 1.0, which was also based on Mosaic, from whom Microsoft had licensed the code. IE 2.0 followed later that year, with IE 3.0 following in 1996. IE 3.0 was notable for introducing support for cascading style sheets (CSS), Java, and ActiveX, but Netscape continued to dominate the market, with IE making up only approximately 10 percent of the market.

Netscape Loses Its Dominance

Over the following years, the market swiftly turned in Microsoft's favor. By IE 4.0 (released in 1997), Microsoft's share of the market had increased to 20 percent, and, by the release of IE 5 in 1999, this had risen to 50 percent. Microsoft's dominance peaked in the first few years of the twenty-first century, with IE 6.0 (released in 2001) claiming more than 80 percent of the market.

Microsoft's aggressive marketing included a decision to bundle IE with Windows. But there's no denying that, at this point in the late 1990s, IE was simply the better browser. Netscape was prone to crashing, it was not as fast as IE, and it was beginning to look distinctly old-fashioned.

In an attempt to revive its fortunes, Netscape decided to release the source code for Navigator, and branded it as Mozilla (also known as Netscape 5), entrusting it to the newly formed Mozilla Foundation. Although this was an important turning point in the history of the web, it did little to help in the immediate future. AOL purchased Netscape, and released Netscape 6 in 2000 and Netscape 7 in 2002. This failed to halt the downturn, though, and AOL eventually announced the end of Netscape in 2008, a year after the release of both Netscape 8 and 9 (which, ironically, were now based on Firefox).

The Growth of Firefox

By 2000, it was clear that Microsoft had won the browser wars, and for the next few years, it enjoyed unchallenged dominance of the market. However, the Mozilla Foundation was still hard at work. Mozilla 1.0 was released in 2002 but failed to make much of an impact in the Windows environment.

Some Mozilla developers were becoming increasingly unhappy with the direction Mozilla was taking, feeling it was becoming increasingly bloated, and branched off their own port of the Mozilla code. After several changes to the name, this ultimately became Mozilla Firefox — usually referred to simply as Firefox.

Firefox 1.0 was released in 2004, but it wasn't until version 2.0, released 2 years later that things began to take off. Mozilla marketed Firefox heavily to the everyday user as a faster, more secure alternative to IE; while bloggers and techies were quick to praise the more advanced features. Finally, it was felt, there was a worthy rival to IE, and by the end of 2006, Firefox's share of the market had risen to 10 percent.

Firefox 3.0 was released in 2008, and by the end of 2010, had a market share of approximately 25 to 30 percent. It's ironic that just as IE's early growth was boosted by dissatisfaction among Netscape users, Firefox's growth was aided enormously by growing dissatisfaction among IE users. Indeed, it was felt that, having won the browser war, Microsoft had become somewhat complacent, with IE 6 and 7 being somewhat insipid.

The Present

Microsoft managed to get back on track with the release of IE 8 in 2008. As well as being compliant with CSS 2.1 and Acid 2, IE 8 finally included tabbed browsing — a feature that had been present in Opera and Firefox for some time.

In 2011, IE 9 was released, boasting CSS 3 support; improved graphics rendering; and a new JavaScript engine, Chakra, which was capable of better utilizing multicore CPUs. Also in 2011, Firefox 4 was released with its own new JavaScript engine (JagerMonkey) and hardware graphics accelerator.

INSIDE HTTP

Before beginning an examination of optimization techniques, it would be beneficial to understand how the web works. The remainder of this chapter recaps the basics of the HyperText Transfer Protocol (HTTP), discusses the differences between HTTP 1.0 and 1.1 (in particular, those relating to performance), and then follows the steps taken when a browser requests a page — from the initial domain name service (DNS) look-up through to the rendering. Later chapters revisit these steps in more detail, and you will learn ways to improve performance.

The HyperText Transfer Protocol

HTTP is the means by which web browsers (clients) communicate with web servers and vice versa. It's a text-based protocol operating at the application layer, and, as such, HTTP doesn't concern itself with issues such as routing or error checking: This is the job of the lower layers such as transmission control protocol (TCP) and Internet Protocol (IP).

THE OSI MODEL

The Open Systems Interconnection (OSI) model is a commonly used means of representing the various parts of network traffic in terms of layers, reflecting the way in which encapsulation of data works. The OSI model defines seven layers (the older TCP/IP model defines just four). The seven layers include the following:

- **Physical layer (layer one) —** This is the underlying means of transmitting the electrical signal across the network (for example, Ethernet, USB, or Bluetooth).
- **Data Link layer (layer two) —** This sits above the physical layer and provides transport across it. In the case of Ethernet, the data link layer handles the construction of Ethernet frames, and communicates with devices via their Media Access Control (MAC) addresses.
- **Network Layer (layer three) —** This deals with packet routing across more complicated networks. Internet Protocol (IP) is the most commonly used protocol at this level, and is capable of traveling across multiple networks, and through intermediate devices such as routers.
- **Transport Layer (layer four) —** This sits on top of the network layer and provides higher-level features such as flow control and the concept of connections. The most commonly seen protocols at this level are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).
- **Session Layer (layer five) —** This handles the management of sessions between the applications on either side of the network connection. Protocols used at this layer include NetBios, H.245, and SOCKS.
- **Presentation Layer (layer six) —** This handles the formatting of data. One of the most common examples is seen in telnet, where differences in the capabilities of terminals must be accounted for. Here, the presentation layer ensures that you see the same thing in your telnet session no matter what your terminal capabilities or character encodings are.
- **Application Layer (layer seven) —** At the top of the OSI model is the application layer, which contains some of the most well-known protocols, including Simple Message Transport Protocol (SMTP), HTTP, File Transfer Protocol (FTP), and Secure Shell (SSH). In many cases, these protocols are plain text (rather than binary), and are, by their nature, high level.

Instead, HTTP deals with the higher-level requests involved in navigating the web, such as, Fetch the Index Page from `http://www.google.com` or Post This Form Data to the CGI Script at Such-and-Such.

Navigating to a web page in your browser typically results in a series of HTTP requests being issued by the client to fetch the resources contained on the page. For each request, the server issues a response. Usually, the response contains the resource requested, but sometimes it indicates an error

(such as the infamous `404 Not Found` error) or some other message. Let's take a look at the HTTP protocol in action.

Using the Live HTTP Headers extension for Firefox, you can watch the HTTP headers that flow as you browse the web. This is an incredibly useful extension, and one that you will frequently use. If your knowledge of HTTP is a bit rusty, now would be a good time to install Live HTTP Headers and spend a bit of time watching traffic flowing.

Here is the traffic generated when you view a simple test page. (For brevity, some lines have been removed.)

```
GET /test.html HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.1.8)
     Gecko/20100308 Iceweasel/3.5.8 (like Firefox/3.5.8) GTB7.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Keep-Alive: 300
Connection: keep-alive


HTTP/1.1 200 OK
Server: Apache/2.2.15 (Debian) PHP/5.3.2-1 with Suhosin-Patch mod_ssl/2.2.15
     OpenSSL/0.9.8m mod_perl/2.0.4 Perl/v5.10.1
Last-Modified: Thu, 29 Jul 2010 15:02:49 GMT
Etag: "31b8560-3e-48c8807137840"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 68
Keep-Alive: timeout=3, max=10000
Connection: Keep-Alive
Content-Type: text/html
-----------------------------------------------------------
GET /logo.gif HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.1.8)
     Gecko/20100308 Iceweasel/3.5.8 (like Firefox/3.5.8) GTB7.1
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive


HTTP/1.1 200 OK
  Server: Apache/2.2.15 (Debian) PHP/5.3.2-1 with Suhosin-Patch mod_ssl/2.2.15
       OpenSSL/0.9.8m mod_perl/2.0.4 Perl/v5.10.1
Last-Modified: Wed, 15 Apr 2009 21:54:25 GMT
Etag: "31bd982-224c-4679efda84640"
Accept-Ranges: bytes
Content-Length: 8780
```

```
Keep-Alive: timeout=3, max=9999
Connection: Keep-Alive
Content-Type: image/gif
-----------------------------------------------------------
GET /take_tour.gif HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.1.8)
     Gecko/20100308 Iceweasel/3.5.8 (like Firefox/3.5.8) GTB7.1
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive


HTTP/1.1 200 OK
  Server: Apache/2.2.15 (Debian) PHP/5.3.2-1 with Suhosin-Patch mod_ssl/2.2.15
       OpenSSL/0.9.8m   mod_perl/2.0.4 Perl/v5.10.1
Last-Modified: Wed, 15 Apr 2009 21:54:16 GMT
Etag: "31bd9bc-c9e-4679efd1ef200"
Accept-Ranges: bytes
Content-Length: 3230
Keep-Alive: timeout=3, max=10000
Connection: Keep-Alive
Content-Type: image/gif
-----------------------------------------------------------
```

In this example, you first see the browser send a `GET` request for `/test.html` from the web server on `127.0.0.1`. Notice that the wanted HTTP protocol version is also stated. The remaining lines of the request include some additional information: the browser user agent; the Multipurpose Internet Mail Extension (MIME) type, languages, and compression methods that the browser accepts; and `Keep Alive/Connection` options.

The server responds with `HTTP/1.1 200 OK` (indicating success) and returns `test.html` in the body of the response. (Remember, this discussion is about headers here, not bodies.) The server response indicates the MIME type of the resource returned (text/HTML), the size, and the compression type (`gzip` here). The last modified time is given, which will be important when you learn about caching.

After the browser has fetched `test.html`, it can parse the HTML and request any resources contained in it. The test page contains two images, and the browser now requests these. The responses are similar to the first, but there are a few subtle differences: The `Content-Type` in the response is now `image/gif`, and no `Content-Encoding` header is set. (This web server isn't configured to use compression when delivering images.)

A full discussion of HTTP could take up a whole book. Assuming you're broadly happy with HTTP, let's continue the discussion by looking at the areas that relate to performance.

HTTP Versions

The history of the HTTP protocol can be traced back to the first version, 0.9, defined in 1991. The web was a different place then, and although this crude protocol served the job, it wasn't long

before refinements were made. These resulted in the creation of HTTP 1.0, defined in RFC 1945 in 1996.

Whereas HTTP 0.9 was limited to making simple `GET` requests with no additional headers, version 1.0 added most of the features now associated with the protocol: authentication, support for proxies and caches, the `POST` and `HEAD` methods, and an array of headers. HTTP 0.9 is pretty much obsolete now, but HTTP 1.0 is still occasionally seen in use and is still a usable protocol for navigating the modern web.

Although the move from HTTP 0.9 to 1.0 marked a major improvement to the protocol, the current version — HTTP 1.1 (laid out in RFC 2616 in 1999) — is essentially just a fine-tuning of HTTP 1.0, with particular improvements made to caching, reuse of connections, and support for virtual hosting. Let's look in more detail at the major improvements.

Support for Virtual Hosting

In the early days of the web, each domain had its own IP address, and there was never any need for an HTTP request to specify from which domain it was requesting a resource. It simply connected to the appropriate IP (obtained via DNS), and the web server knew which domain name this mapped to. As the Internet boomed, and concerns grew about the limited IPv4 address space, web servers began to support *virtual hosting* — a method to host multiple domain names on the same IP address. One of the changes in HTTP 1.1 was the introduction of the `Host` header, which enabled the client to specify for which domain it was requesting the resource.

A typical HTTP 1.0 request would have looked like this:

```
GET /index.html HTTP/1.0
```

In HTTP 1.1, this now appears as follows:

```
GET /index.html HTTP/1.1
Host: mydomain.com
```

Although this feature has little to do with performance, it had a big impact on the growth of the web and is one of the most compelling reasons to use HTTP 1.1 over 1.0.

Caching

Caching is an important topic that will be discussed numerous times in this book. In general, it consists of storing resources in a temporary location for faster retrieval in the future. In the case of client-side caching, this temporary location is an area of the user's hard disk, set aside by the web browser. In many situations, the browser can retrieve previously requested resources directly from the cache, without needing to query the web server.

Although the caching mechanisms of HTTP 1.0 provide fairly good caching support (albeit somewhat vaguely defined), HTTP 1.1 extends these and adds a host of new options, including the `Cache-Control` and `Vary` headers. These offer you much greater control over how browsers and intermediate proxies can cache your content.

> **NOTE** *You'll learn more about intermediate caches and proxies in Chapter 2, "Utilizing Client-Side Caching."*

HOW BROWSERS DOWNLOAD AND RENDER CONTENT

One of the most important building blocks for the budding web optimizer is an understanding of how browsers render websites. They don't always behave as you might expect, and there can be subtle differences from browser to browser. Only when you fully understand these can you be in a position to perform effective optimization.

"Waterfall" graphs are invaluable when trying to understand this. Let's dive in with an example, albeit for a fairly simple page — `http://kernel.org`, the home of the Linux kernel — shown in Figure 1-1.

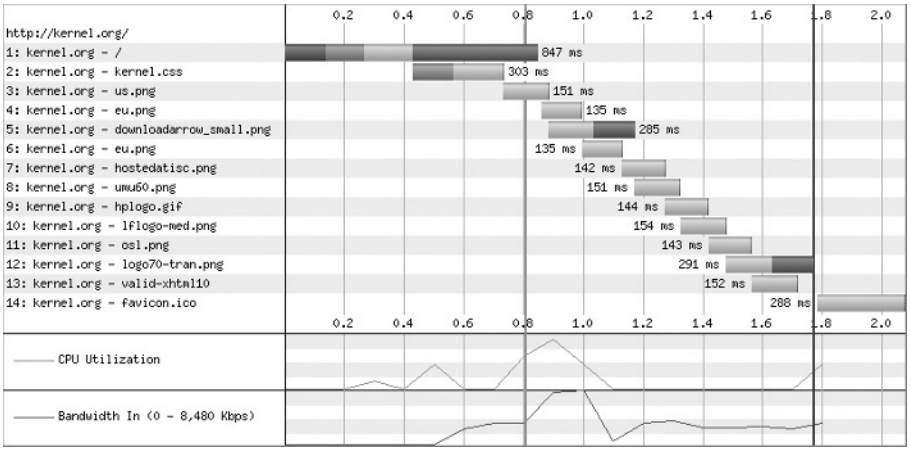

FIGURE 1-1

The first thing the browser does is resolve `kernel.org` to an IP address using DNS, as indicated by the first segment of the first request line. It then attempts to open an HTTP connection to `kernel.org`. The second segment shows the time taken to do this.

At the beginning of the third segment, the TCP connection has been created, and, at this point, the browser issues its request. However, it isn't until the start of the fourth segment that the web server starts to send back content. (This can be attributed to latency on the web server.) Finally, some 847 milliseconds (ms) after the start, the HTML document has been fully retrieved.

Of course, most web pages don't consist of simply an HTML document. (If they did, the lives of web masters would be less complicated.) Invariably, there are also links to style sheets, images, JavaScript, and so on, embedded in the page, which the browser must also retrieve.

The browser doesn't wait until it finishes retrieving the HTML document before it starts fetching these additional resources. Naturally, it can't start fetching them immediately, but as soon the HTML document starts to arrive, the browser begins parsing it and looks for links. The first of these is the style sheet (`kernel.css`) contained near the top of the page in the head, and it duly requests this. You now have two connections running in parallel — this is enormously faster than the requests made in a linear fashion, one by one.

This time, you don't have the delay of a DNS lookup (the response from the previous lookup has been cached by the browser), but you once again have a delay while the browser initiates a TCP connection to the server. The size of this particular CSS file is a mere 1.7 KB, and the download segment of the request is hence barely visible.

Given what you now know about the browser parsing the document as it comes in, why is there such a delay until `us.png` is requested? Perhaps this image isn't referenced until approximately three-quarters of the way through the document (because the download appears to begin approximately three quarters of the way through downloading the HTML). Actually, this image is first referenced on line 35 of the document (which is more than 600 lines in total).

The reason for the delay is that, historically, most browsers only download two resources in parallel from the same host. So, the request for `us.png` doesn't begin until the request for `kernel.css` finishes. Look carefully at the rest of the waterfall to see that at no point are there ever more than two requests running in parallel. (You'll learn more about this shortly.)

There's something else different about `us.png` (and the resources that follow it) — there is no TCP connection segment. The browser is reusing the existing TCP connection it has with the server, cutting out the time required to set up a new connection. This is an example of the persistent connections mentioned earlier in this chapter. In this example, the saving is approximately 0.1 second on each request — and more than 12 requests, which mounts up to sizeable savings.

It's also worth noting that, in this example, the overhead involved in making the request makes up a significant proportion of the overall time. With the first, fifth, and twelfth resources, the actual downloading of data accounts for approximately one-half the time needed to fetch the resource. With the other resources, the download time is virtually insignificant, dwarfed by the latency of issuing the request and waiting for a response from the server. Although this is only one example, the pattern of many small resources is common and illustrates that performance is not all about keeping the size of resources small.

Rendering

After the browser retrieves the HTML document, it can begin to parse the document and render it on the screen. Referring to the waterfall view in Figure 1-1, the first vertical line shows the point at which the browser begins rendering, whereas the second vertical line shows the point at which rendering is complete.

If the image dimensions are not known, the browser does not allocate any screen space to them during the initial rendering. As a result, the page flow must be recalculated after they have been downloaded, and the dimensions become known. This can lead to the rather ugly effect of text jumping around the page as the page loads.

Although the `kernel.org` example implies that the page takes approximately 1 second to render, this is a little misleading. Actually, the majority of the page renders in the blink of an eye. Then you must wait for the images to download. If no images were involved (or they were already in the browser's cache), how long would it take the browser to simply parse and render the HTML document? This is an area discussed again in Chapter 6, "JavaScript, the Document Object Model, and Ajax," when you learn about ways to reduce rendering times.

Persistent Connections and Keep-Alive

In HTTP 1.0, the default behavior is to close the connection after a resource has been retrieved. Thus, the following is the flow of events when the client needs to fetch multiple resources:

1. The client opens a TCP connection to the server.
2. The client requests the resource.
3. The server responds and closes the connections.
4. The client opens a new TCP connection.
5. The client requests another resource.

This is a rather wasteful approach, and the process to build up and tea down the TCP connections adds a significant amount of latency to requests (not to mention extra CPU and RAM usage on both client and server). This overhead is even more significant when requesting many small resources, which tends to be the nature of most websites.

Figure 1-2 shows this problem with a waterfall view showing a page containing 22 small images loading in IE 7. The effect has been exaggerated a bit by conducting the test from a dial-up connection to a web server located on the other side of the Atlantic. (So the latency is quite high.) The problem still exists for broadband users, just not to the same degree.

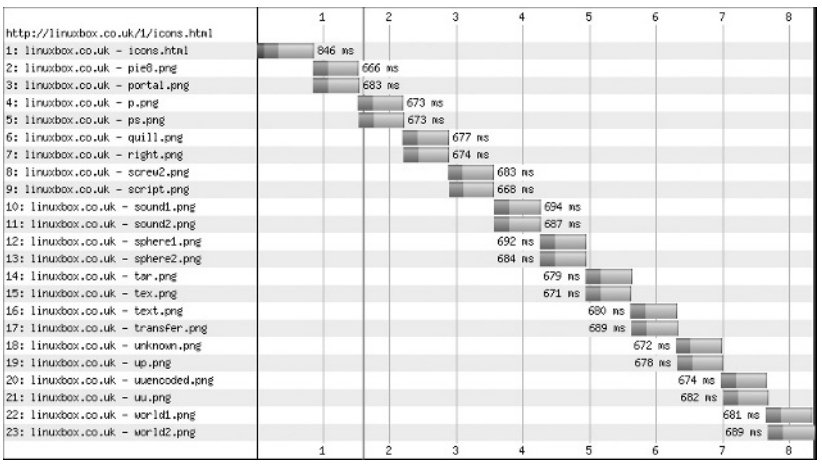

FIGURE 1-2

Clearly, this is not an ideal situation, which is one of the reasons that (as you have seen) browsers typically issue more requests in parallel when talking in HTTP 1.0.

Keep-Alive

This shortcoming was partially addressed by the introduction of `Keep-Alive`. Although it was never an official part of the HTTP 1.0 specifications, it is well supported by clients and servers.

With `Keep-Alive` enabled, a server will not automatically close the connection after sending a resource but will instead keep the socket open, enabling the client to issue additional requests. This greatly improves responsiveness and keeps network traffic down.

A client indicates it wants to use `Keep-Alive` by including the header `Connection: Keep-Alive` in its request. If the server supports `Keep-Alive`, it signals its acceptance by sending an identical header back in the response. The connection now remains open until either party decides to close it.

Unfortunately, browsers can't always be relied upon to behave themselves and close the connection when they finish issuing requests, a situation that could lead to server processes sitting idle and consuming memory. For this reason, most web servers implement a `Keep-Alive` timeout — if the client does not issue any further requests within this time period (usually approximately 5 to 10 seconds), the server closes the connection.

In addition, servers may also limit the number of resources that may be requested during the connection. The server communicates both these settings with a `Keep-Alive` header like this:

```
Keep-Alive: timeout=5, max=100
```

However, `Keep-Alive` is not an officially recognized header name and may not be supported by all clients.

Persistent Connections

HTTP 1.1 formalized the `Keep-Alive` extension and improved on it, the result being known as persistent. Persistent connections are the default in HTTP 1.1, so there's no need for the client or server to specifically request them. Rather, the client and server must advertise their *un*willingness to use them by sending a `Connection: close` header.

> **NOTE** *Just to clarify,* `Keep-Alive` *is the name of the unofficial extension to HTTP 1.0, and persistent connections are the name for the revised version in HTTP 1.1. It's not uncommon for these terms to be used interchangeably, despite the (admittedly small) differences between them. For example, the Apache* `KeepAlive` *directives (which you'll learn about in Chapter 7, "Working with Web Servers") also relate to persistent connections.*

Keep-Alive and Persistent Connections

Let's revisit an earlier example, again conducted from a dial-up connection but this time to a server that has `Keep-Alive` enabled. Figure 1-3 shows the waterfall view.

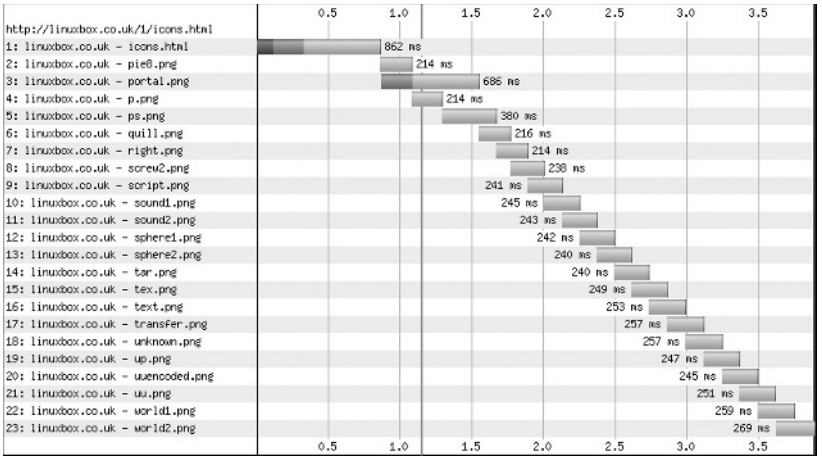

FIGURE 1-3

This time, the results are significantly better, with the page loading in less than half the time. Although the effect has been intentionally exaggerated, `Keep-Alive` is still a big asset in the majority of situations.

When Not to Use Keep-Alive

So, if `Keep-Alive` is only an asset in the *majority* of situations, under which circumstances might `Keep-Alive` not be beneficial?

Well, if your website mostly consists of HTML pages with no embedded resources (CSS, JavaScript, images, and so on), clients will need to request only one resource when they load a page, so there will be nothing to gain by enabling `Keep-Alive`. By turning it off, you allow the server to close the connection immediately after sending the resource, freeing up memory and server processes. Such pages are becoming rare, though, and even if you have some pages like that, it's unlikely that every page served up will be.

Parallel Downloading

Earlier in this chapter, you learned that most browsers fetch only a maximum of two resources in parallel from the same hostname and saw a waterfall view of IE 7 loading `http://kernel.org` to illustrate this point. Given that parallel downloading is so beneficial to performance, why stop at two?

The reason that browsers have (historically) set this limit probably stems from RFC 2616 (which details version 1.1 of HTTP; you can find it at `http://www.ietf.org/rfc/rfc2616.txt`). According to that RFC, "Clients that use persistent connections *should* limit the number of simultaneous connections that they maintain to a given server. A single-user client *should not* maintain more than 2 connections with any server or proxy.... These guidelines are intended to improve HTTP response times and avoid congestion."

However, the Internet has come a long way since 1999 when that RFC was written, and for some time, web developers have been more than happy to use tricks to get around this limit. You'll learn about some of these shortly.

More recently, browser vendors have started to flout this guideline, too, usually justifying it by pointing out that the RFC says "should not," not "must not." Of course, their main concern is with providing a faster browsing experience to users (especially if other vendors have already increased their maximum connections per domain) — any increase in network congestion or overloading of the web server isn't their problem.

The issue has caused fierce debate, with apocalyptic predictions of web servers being brought to their knees by this flood of parallel connections. This hasn't happened yet, though, and for good reason. While a browser making eight parallel requests rather than two can certainly increase server load, it stands to reason that these eight connections will be open for a shorter period of time. So, although it may cause a slight spike in load, it's only short-lived. It would be hypocritical for web developers to complain about this aspect of browser behavior because they often use tricks to increase parallel downloading.

The first major browser to break from this "rule" of two maximum connections per hostname was Safari, starting with version 3 in 2007. Firefox 3 and Opera followed in 2008, and IE 8 in 2009. Table 1-1 shows the current state of play.

TABLE 1-1 Maximum Parallel Connections Per Host

BROWSER	MAX PARALLEL CONNECTIONS PER HOST
IE 6 and 7	2
IE 8	6
IE 9	6
IE 10	8
Firefox 2	2
Firefox 3	6
Firefox 4 to 17	6
Opera 9.63	4
Opera 10	8
Opera 11 and 12	6
Chrome 1 and 2	6
Chrome 3	4
Chrome 4 to 23	6
Safari 3 and 4	4

It's worth noting that IE 8 automatically reduces this figure to two for users on dial-up connections and enables web developers to detect the current value in JavaScript via `window.maxConnectionsPerServer`.

Increasing Parallel Downloads

Earlier, you learned that web developers have traditionally often used tricks to increase the amount of parallelization. With the current generation of browsers all enabling more than two parallel connections, the need for this has greatly diminished. As of this writing, IE 7 still makes up a significant share of the market, though, so it's still a topic worth discussing.

Eagle-eyed readers will have noticed that this discussion has been quite pedantic in saying that browsers establish only *x* number of parallel connections *per hostname*. It's not globally (there's another limit for that), per server, per IP address, or even per domain. You can easily exploit this by forcing the browser to fetch more than two resources in parallel.

Consider the following HTML snippet:

```
<img src="images2/tar.png"><img src="images2/sound2.png">
<img src="images2/sound1.png"><img src="images2/screw1.gif">
<img src="images2/portal.gif"><img src="images2/world1.gif">
<img src="images2/pie1.gif"><img src="images2/pie2.gif">
<img src="images2/pie3.gif"><img src="images2/pie4.gif">
<img src="images2/pie5.gif"><img src="images2/pie6.gif">
<img src="images2/pie7.gif"><img src="images2/generic.png">
<img src="images2/folder.png"><img src="images2/link.png">
<img src="images2/layout.png"><img src="images2/dvi.png">
<img src="images2/broken.png"><img src="images2/bomb.png">
```

Let's look at how this loads in IE 7, a browser that sticks to the RFC guidelines, as shown in Figure 1-4.

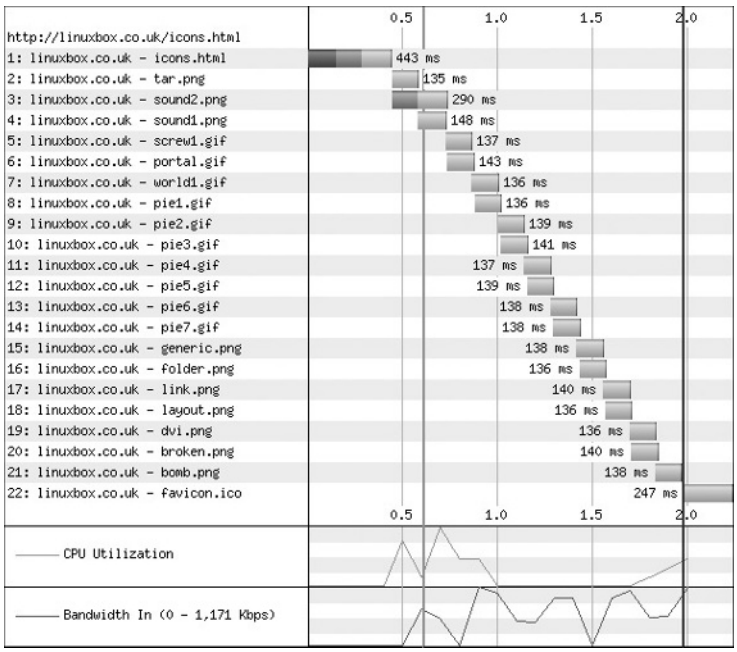

FIGURE 1-4

Increase the level of parallelization by splitting the images across two subdomains: `static.linuxbox.co.uk` and images.`linuxbox.co.uk`, as shown here:

```
 <img src="/images2/tar.png">
      <img src="http://images.linuxbox.co.uk/images2/sound2.png">
<img src="http://images.linuxbox.co.uk/images2/sound1.png">
      <img src="http://static.linuxbox.co.uk/images2/screw1.gif">
<img src="http://static.linuxbox.co.uk/images2/portal.gif">
      <img src="http://images.linuxbox.co.uk/images2/world1.gif">
<img src="http://images.linuxbox.co.uk/images2/pie1.gif">
      <img src="http://static.linuxbox.co.uk/images2/pie2.gif">
<img src="http://static.linuxbox.co.uk/images2/pie3.gif">
      <img src="http://images.linuxbox.co.uk/images2/pie4.gif">
<img src="http://images.linuxbox.co.uk/images2/pie5.gif">
      <img src="http://static.linuxbox.co.uk/images2/pie6.gif">
<img src="http://static.linuxbox.co.uk/images2/pie7.gif">
      <img src="http://images.linuxbox.co.uk/images2/generic.png">
<img src="http://images.linuxbox.co.uk/images2/folder.png">
      <img src="http://static.linuxbox.co.uk/images2/link.png">
<img src="http://static.linuxbox.co.uk/images2/layout.png">
      <img src="http://images.linuxbox.co.uk/images2/dvi.png">
<img src="http://images.linuxbox.co.uk/images2/broken.png">
      <img src="http://static.linuxbox.co.uk/images2/bomb.png">
```

As shown in Figure 1-5, using this simple technique has cut approximately 0.6 seconds off the page loading time. (Keep in mind that the nature of the Internet makes it difficult to exactly replicate the conditions between runs. The second run may have found the network a little quieter, or the web server less busy.)

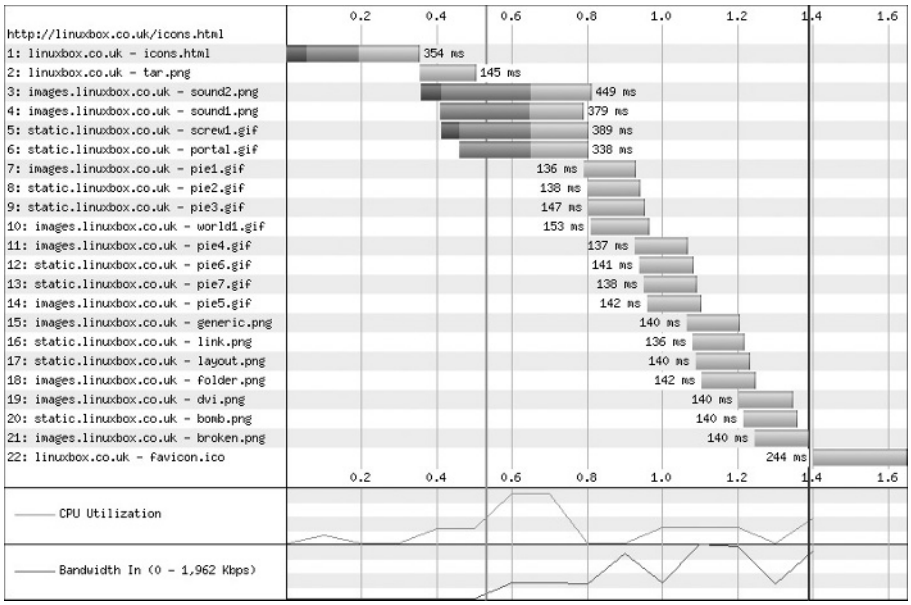

FIGURE 1-5

What if you take this to extremes and use enough hostnames to make all the images download in parallel?

```
<img src="/images2/tar.png">
     <img src="http://images.linuxbox.co.uk/images2/sound2.png">
<img src="http://images.linuxbox.co.uk/images2/sound1.png">
     <img src="http://static.linuxbox.co.uk/images2/screw1.gif">
<img src="http://static.linuxbox.co.uk/images2/portal.gif">
     <img src="http://content.linuxbox.co.uk/images2/world1.gif">
<img src="http://content.linuxbox.co.uk/images2/pie1.gif">
     <img src="http://resources.linuxbox.co.uk/images2/pie2.gif">
<img src="http://resources.linuxbox.co.uk/images2/pie3.gif">
     <img src="http://media.linuxbox.co.uk/images2/pie4.gif">
<img src="http://media.linuxbox.co.uk/images2/pie5.gif">
     <img src="http://bart.linuxbox.co.uk/images2/pie6.gif">
<img src="http://bart.linuxbox.co.uk/images2/pie7.gif">
     <img src="http://homer.linuxbox.co.uk/images2/generic.png">
<img src="http://homer.linuxbox.co.uk/images2/folder.png">
     <img src="http://marge.linuxbox.co.uk/images2/link.png">
<img src="http://marge.linuxbox.co.uk/images2/layout.png">
     <img src="http://lisa.linuxbox.co.uk/images2/dvi.png">
<img src="http://lisa.linuxbox.co.uk/images2/broken.png">
     <img src="http://maggie.linuxbox.co.uk/images2/bomb.png">
```

Figure 1-6 shows the result. It's not quite what you had hoped for — the page loading time has just doubled! Although the additional DNS lookups required haven't helped (but this could be eliminated by using IP addresses rather than hostnames), the main problem seems to be the web server (Apache, in this case), which was somewhat slow to answer the barrage of requests.

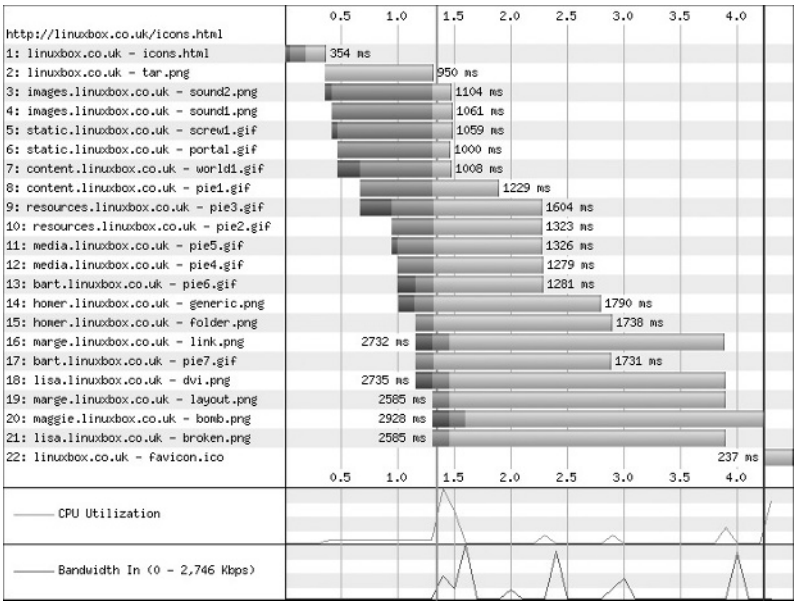

FIGURE 1-6

After a little Apache tuning (you'll learn exactly what was done in Chapter 7), if you rerun the test, the results are more promising, as shown in Figure 1-7.

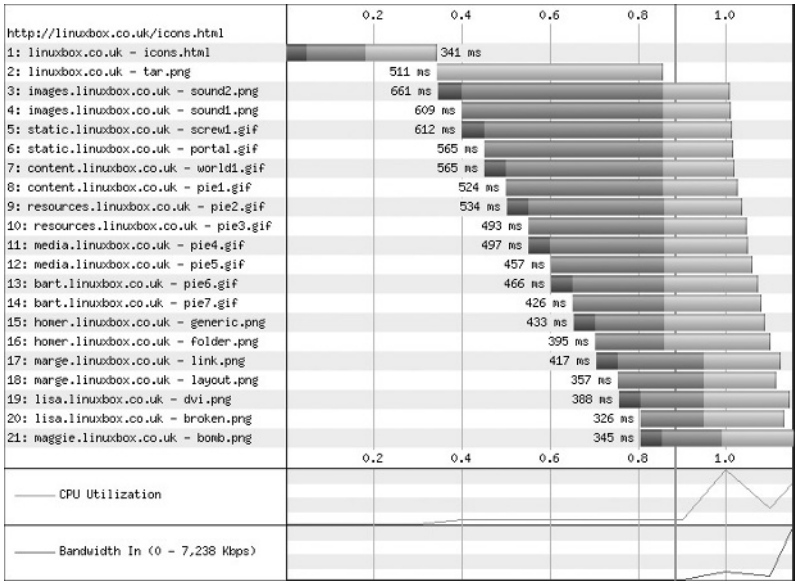

FIGURE 1-7

This time, the load time was 1.15 seconds, an improvement on the test using two subdomains. This illustrates that, unless the web server is geared toward such a flood of traffic, too much parallelization can dramatically reduce loading times.

So, where is the sweet spot? Research suggests that splitting content across two to four hostnames offers the best performance. With anything more than four hostnames, you risk flooding the server with requests, and (in the case of dial-up users) saturating the network connections.

It's a mistake to think that parallelization can solve the problem of pages being too heavy, too. It's a great way to reduce the latency involved with issuing many requests for small resources (which is how most sites are structured). However, if you have, for example, two large images, arranging for them to download in parallel will simply result in each downloading more slowly. Bandwidth is a finite resource.

Coping with Modern Browsers

Although splitting resources across hostnames used to be a great technique to reduce loading times, its days are numbered because the most recent generation of browsers moves away from the RFC guidelines. Catering for both older and newer browsers poses somewhat of a problem, though. It's quite easy to envision a scenario in which a site, carefully tuned to accommodate the two-connection limit of IE 7, results in a surge of connections in IE 8, driving up CPU usage on both the client and server, and killing performance.

One option is to compromise. Splitting resources across just two domains will still speed things up a little for older browsers but will lessen the risk of the more recent browsers creating a packet storm.

Another solution would be to use server-side scripting to determine the browser type and version, and rewrite the HTML document accordingly. Thus, a Firefox 2 user may see resources split across four domains, whereas a visitor on Safari would see them split across just two. Naturally, this causes some additional CPU usage on the server, but it need not be too intense if the server were simply returning different versions of a static document, rather than rewriting the page on-the-fly.

Coping with Proxies

This isn't quite the full story, though. When multiple users behind a proxy visit a site simultaneously, the web server sees a flood of connections all coming from the same IP address. With older browsers, the two-connection limit helped to lessen this effect. With newer browsers issuing perhaps four or six requests in parallel, the potential for a flood is much greater, and an Intrusion Detection System (IDS) could easily mistake this for a SYN flood or other form of denial-of-service (DoS) attack (perhaps leading to the IP address being blocked). For this reason, most newer browsers lower the number of parallel connections when the browser connects via a proxy, and, consequently, their behavior is more like that of older browsers.

Incidentally, most proxies don't support HTTP 1.1, so a client connecting through a proxy usually must downgrade to HTTP 1.0, which leads neatly into the next subject.

Parallel Downloads in HTTP 1.0

Although persistent connections (also known as `Keep-Alive`) are supported in many HTTP 1.0 implementations, they are not an official part of the specification. As such, clients connecting via a proxy — or to a server that does not support HTTP 1.1 — lose out on performance. Increasing the number of parallel connections is a way to mitigate some of this performance loss. As a result, many browsers use different levels of parallelization for HTTP 1.0 and 1.1. Thus, Table 1-1 can be refined to show the data in Table 1-2.

TABLE 1-2: Different Levels of Parallelization for HTTP 1.0 and HTTP 1.1

BROWSER	MAX PARALLEL CONNECTIONS	
	HTTP 1.1	HTTP 1.0
IE 6 and 7	2	4
IE 8	6	6
IE 9	6	6
IE 10	8	
Firefox 2	2	8

Firefox 3	6	6
Firefox 4 - 17	6	6
Opera 9.63	4	4
Opera 10	8	8
Opera 11 and 12	6	6
Chrome 1 and 2	6	4
Chrome 3	4	4
Chrome 4 - 23	6	6
Safari 3 and 4	4	4

The effect is most noticeable in the older browsers, with newer browsers generally having abandoned this strategy — perhaps because of the conflict with the proxy connection limitations previously discussed.

In the past, both `aol.com` and Wikipedia have intentionally downgraded to HTTP 1.0, presumably with the intention to speed up their loading times by tricking the browser into increasing parallelization. This seems to be a somewhat muddled logic. If you must increase parallelization, the techniques described in this chapter are a better path to follow.

SUMMARY

After a refresher on the history of the web, this chapter provided an inside look at the HTTP protocol, and some of the key concepts for performance — persistent connections, parallel downloading, and rendering. You also met caching, which is an essential part of any website — both for speeding up loading times for the user, and for reducing load on the server.

In Chapter 2, you'll learn about the various types of caching, how they work, and how to implement them to achieve blistering performance.