CHAPTER 8

Could They Do It?: End User Experience Management

Web analytics is your best insight into your website, your brand, and your customers online. But if the site's not working properly, there simply won't be any activity to watch. Sites that aren't reachable can't sell things, and slow sites lose users. Google estimates that for every additional 500 milliseconds of delay, their site loses 20% of their traffic (*http://web2.sys-con.com/node/804850*).

Web applications break in strange and wonderful ways, many of which are beyond your control. Monitoring performance and uptime is usually the job of a web operations team, and it's usually an IT function. And that's the start of the problem.

In most companies, the skills needed to run a website are often broken up into distinct silos. One group is responsible for designing the application, another for testing it, another for managing the platforms and infrastructure, another for analyzing what happens on it, and another for understanding how web activity relates to community opinion.

This separation of roles might make sense on an organizational chart, but for web applications, it's dysfunctional. Web analytics and web performance are two sides of the same coin. They should be in the same business unit and should be judged by metrics like user satisfaction, conversion rates, and uptime. The teams responsible for the website's design, analytics, community, and support need to work alongside the web operations teams that are responsible for making sure the infrastructure, servers, and networks are healthy.

Web operators rely on two main types of tools to make sure their sites are functioning properly. The first, called synthetic testing, simulates visitor requests at regular intervals to make sure the website is working. The second, real user monitoring (RUM), analyzes actual user traffic to measure responsiveness and detect errors.

What's User Experience? What's Not?

A visitor's experience is the performance, availability, and correctness of the site they visit. For the purpose of this book, we'll define it as *a measure of how accurately a user's visit reflects the visit its designers intended*. We refer to the task of managing this experience as End User Experience Management, or EUEM.

In this book, we make the distinction between EUEM and usability. Usability, which we measure with WIA and usability testing, looks at how users tried to use the site. If visitors did something wrong—if they couldn't find a particular button, or if they entered their hat size in the price field—they weren't doing something the designers intended. While that issue is, of course, vitally important to web operators, it's a usability problem.

Here, we're concerned with how well the website delivers the experience the designers intended. Problems may include pages that took a long time to reach a visitor or rendered too slowly in the visitor's browser, websites that broke because third-party components or plug-ins didn't work, and content that wasn't correct or wasn't shown on the screen properly.

At its simplest, EUEM is about answering the question, "Is this site down, or is it just me?" Every site operator has experienced a moment of sudden dread when they try to visit their own site and find that it's not working. If you implement EUEM correctly, you'll know as soon as there's a problem, and you'll know who's affected.

ITIL and Apdex: IT Best Practices

Before we get further in this discussion, we need to talk about standards. The IT Information Library (ITIL) is a set of recommendations and best practices for operating IT services (*www.tso.co.uk*). It includes a dauntingly comprehensive set of processes and terms, but its fundamental message is simple and straightforward: *any IT service exists to perform a business function*.

ITIL is all about running IT as a service. If you're in web operations, your "customers" are really the rest of the business, and you're doing your job when the services the business needs are available as expected.

Terms like "performance" and "availability" have many meanings to many people. ITIL defines availability as "the ability of a service to perform its agreed function when required, as a function of reliability, maintainability, serviceability, performance, and security." In other words, availability is the percent of the time that you did what you said you would.

Availability decreases when the service you run—in this case, a website—isn't working properly. That may be because of an incident, which is "an unplanned interruption to an IT service" or "a reduction in the quality of an IT service." It may also be because

the site is slow. Slowness simply means that the response time—the time taken to complete an operation or transaction—exceeds an agreed-upon threshold.

Other factors, such as a security breach or an inability to update the site, all affect the availability of an IT service. In this book, we're going to focus primarily on what ITIL calls Response Time and Incident Management, both of which affect the availability of the web service. For web operators, the task of "day-to-day capacity management activities, including threshold detection, performance analysis and tuning, and implementing changes related to performance and capacity" is called *performance management*.

Though ITIL makes good sense, it's a bit formal for many startups that are focused on delivering products to markets before their funding runs out. If you're looking for something a bit less intimidating, consider Apdex (Application Performance Index), which is a formula for scoring application performance and availability that's surprisingly simple, and can be used for comparison purposes.

Originally conceived by Peter Sevcik, Apdex is now an industry initiative supported by several vendors. An Apdex score for an online application is a measurement of how often the application's performance and availability is acceptable.

Here's how it works: every transaction in an application (such as the delivery of a page to a browser) has a performance goal, such as, "This page should load in two seconds." Every page load is then scored against this goal. If the page is delivered within the goal, the visitor was "satisfied." If the page was delivered up to four times slower than the goal, the visitor was "tolerating." If it took more than four times longer—or simply wasn't delivered at all because of an error—the visitor was "frustrated."

To calculate an Apdex score, you use a simple formula, shown in Figure 8-1.

$$Score = \left\{ \frac{(Satisfied) + (Tolerating/2)}{All} \right\}$$

Figure 8-1. Calculating an Apdex score

In other words, you add up all the satisfied measurements, 50% of all the tolerating measurements, and 0% of all the frustrated ones, then divide by the total number of measurements.

http://www.apdex.org/overview.html has information on how to calculate an Apdex score, as well as vendors who support the metric. One of its key strengths is the ability to "roll up" scores across many different sites, pages, and applications to communicate performance and availability at different levels of an organization.

Apdex is useful for several reasons. You can take several Apdex scores for different pages or functions on a website, and roll them up into a total score, while retaining individual "satisfied" thresholds. It's also a consistent way to score a site's health, whether you have only a few measurements—a test every five minutes, for example—or many thousands of measurements every second.

A Note on Terminology

While ITIL's terms are useful, we're going to use terms that are more common in the web operations world. When we refer to *availability* here, we're talking about uptime—*the amount of time that the site can be reached, even if it's relatively unresponsive*. This is similar to ITIL's concept of incidents and problems—if you see an error, availability goes down. If your site's always up, its availability is 100%.

When we refer to *performance*, on the other hand, we mean web infrastructure performance, specifically *the responsiveness of an application as experienced by the end user*. While we often talk about average performance, averages themselves aren't very useful. We care more about how many of the measurements we've taken exceed a threshold or about what the worst-served visitors are experiencing.

Why Care About Performance and Availability?

You may have formal contracts with your site's users that dictate how much of the time your site will be working and how quickly it will handle certain requests. Even if you don't have a formal SLA in place, you still have an implied one. Your visitors have expectations. They'll get frustrated if your site is much slower than that of your competitors, or if your performance is inconsistent and varies wildly. On the other hand, they'll be more tolerant of delay if they're confident that you'll give them the information they're after or if you've been recommended by others.

Failing to meet those expectations hurts your business:

- A site that's unresponsive or plagued by incidents and unpredictable availability has lower conversion rates.
- Sites that deliver a consistently poor end user experience are less likely to attract a loyal following. Poor site performance may also affect perception of your company's brand or reputation.
- You may be liable for damages if you can't handle transactions promptly, particularly if you're in a heavily regulated industry such as finance or healthcare.
- Poor performance may cost you money. If you have a formal contract with users, you may be liable for refunds or service credits. Slow or unavailable sites also encourage customers to find other channels, such as phone support or retail outlets, that cost your organization far more than handling requests via the Web. Once

visitors try those channels, they may stick with them, costing you even more money.

There are six fundamental reasons companies measure the performance of their sites:

- To establish baselines
- To detect and repair errors
- To measure the effectiveness of a change
- To determine the impact of an outage
- To resolve disputes with users
- To estimate how much capacity will be needed in the future

Establish agreed-upon baselines

You need to determine what "normal" performance and uptime are, partly so others can tell whether you're doing your job and partly so you can set thresholds to warn you when something is wrong.

The developer who built your website had an idea of what performance should be like. She was using an idealized environment—her desktop—connecting to a server that was otherwise idle. To understand the composition and efficiency of a web page's design, developers rely on tools like Firebug and YSlow (Figure 8-2).

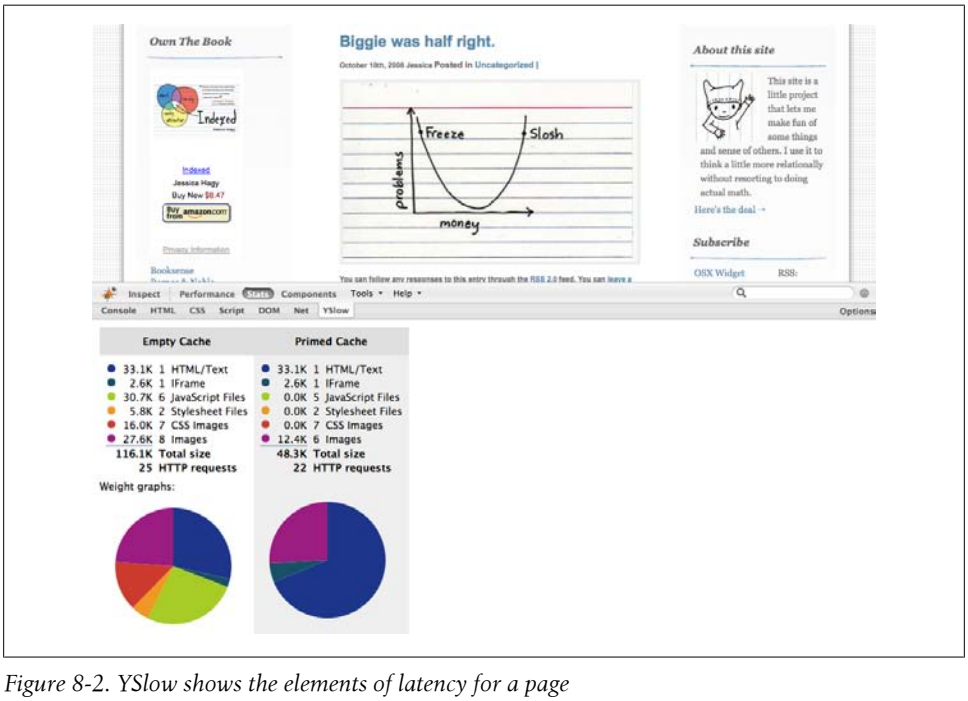


Figure 8-2. YSlow shows the elements of latency for a page

Once launched, websites seldom perform as expected. Changes in traffic, Internet conditions, and visitors' specific hardware and networks all affect how quickly a page loads. The performance you see during release testing won't match what users experience on a busy day, and to understand the impact of load you need to monitor the application at all times of the day, every day of the week. You also need to consider a variety of user environments. Knowing that your site is slow today is interesting, but knowing that it's slow every week at 3:00 A.M. is something you can act on.

Your analytics data can show you when peak traffic occurs, and can even identify times when particularly large or processor-intensive pages are being requested. Some sites, like online flower stores, university enrollment platforms, and tax filing portals see a hundredfold increase in traffic on just a few days of the year, and are comparatively idle the remainder of the year.

If you don't know what those expectations should be, a great place to start is to try your competitors' sites to see what performance they offer. Without agreement on what's "fast enough" and when maintenance is allowed, you're aiming at a target that the rest of the company can move whenever it wants to.

Detect and repair errors to reduce downtime

The most obvious day-to-day use of website monitoring technologies is to detect problems with the site before you hear about them from users. This improves total uptime and reduces user frustration.

Many companies that deploy monitoring tools are shocked by what they first see. They discover that users have been working around problems and suffering through slowdowns the site's operators knew nothing about. Because many of the commercial EUEM applications and services help you to visualize performance and availability, it's easy to communicate these problems with other departments and get the support and capital to fix them.

Measure the effectiveness of a change

Web applications are always changing (or at least, they should be if you're heeding our advice on web analytics and experimentation). Often, companies will find a version of a site that tests well with visitors but has an adverse long-term impact on EUEM or capacity. There's a trade-off to make, which is one of the reasons that user experience is now a concern for marketers as well as technologists.

Even if you're not altering content and software yourself, your server and hardware vendors may be sending you upgrades, or your service provider may be changing routing. Like it or not, you'll be overseeing hundreds of changes. Without monitoring performance before and after the changes, you'll have no idea whether things got better or worse.

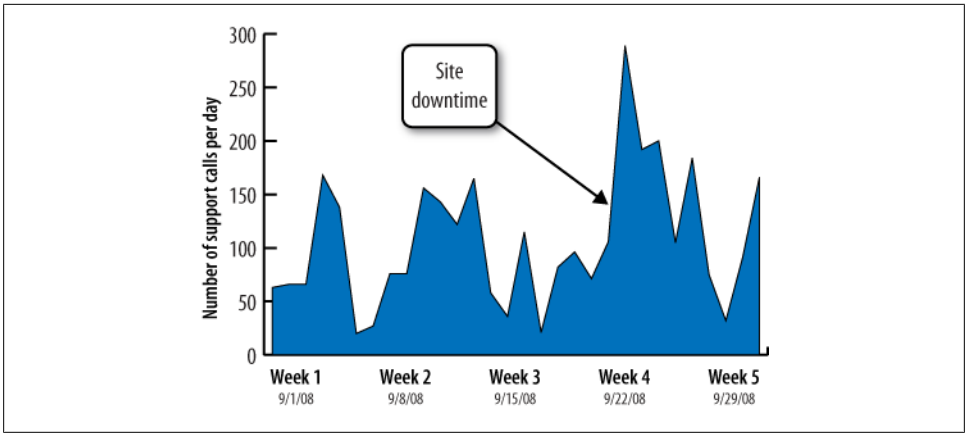


Figure 8-3. Site downtime caused support cases to dramatically increase Monday, September 22, 2008

EUEM tools answer a crucial question behind any change to content, code, equipment, or service providers: *did the change make performance better?* "Better" may mean faster, or less costly, or more reliable. Ultimately, however, you need to tie the change back to business outcomes like conversion, stickiness, and adoption. Dynamic web content updates need to be a part of your organization's change, configuration, and release management processes.

Know the impact of an outage

No matter how well you're doing your job, something will break. In the early days of web operations, sites went completely off the air. Today, we've developed technologies like load balancing and redundancy for websites to prevent most site-wide errors. Instead, errors are transient, affecting some users but not others.

When things break, you need to know the impact of the outage. How many users were affected? When the site returned, was there a flood of returning traffic that slowed things down? You need to track health so you can tie outages to changes in user behavior, such as a spike in phone banking when the web portal gets too slow (Figure 8-3).

In the IT world, the concept of Business Service Management (BSM) encourages us to treat every IT system as a business service. Hardware and network metrics need to be tied to business terms such as lost sales, abandoned shopping carts, and reduced contributions.

Resolve disputes with end users

There's no substitute for the truth. By monitoring your site through tests or user monitoring, you'll know what actually transpired. In disputes with customers, you're able to replace anecdotes with accountability.

If you have contractual obligations with your subscribers, you can prove something wasn't your fault and avoid having to issue a refund. If you're a SaaS provider, you have to deliver uptime and performance that's acceptable or your customers will cancel their contracts and you'll see increased churn, which will reduce revenues and mean you have to spend more money on sales.

Dispute resolution can go even further. By capturing a transcript of what happened, you have a record that may hold up in court. Some heavily regulated industries rely on these records. Consider, for example, an insurance portal. If a buyer purchases fire insurance, then suffers flooding damages and claims, "I actually bought flood insurance but the server got it wrong," having a copy of the user's session is extremely handy.

Estimate future capacity requirements

So you're monitoring the site, validating changes smoothly, optimizing campaigns, fixing problems as they occur, and resolving arguments with newfound finesse and aplomb. Great—your business is probably growing. And that means it's time to add more capacity.

How do you know when you'll need more servers? Because performance and availability can be related to traffic loads, as was the case for Twitter's Fail Whale (Figure 8-4), you can use performance data and web analytics to understand how soon you'll need new physical or virtual machines. EUEM tools let you profile your application to understand where bottlenecks occur and to see which tiers of your infrastructure need additional capacity.

Things That Affect End User Experience

These days, there are so many moving parts in a web application that analyzing end user experience can be tough. Despite all the mashups, plug-ins, mobile browsers, and rich content, however, if you understand the fundamental things that make up a web transaction, you're well on your way to measuring the experience of your website's end users.

At its core, EUEM can be broken down into two components: availability and performance.

Figure 8-4. Twitter.com's fail whale, a familiar sight during the service's growing pains

Availability

As visitors interact with your site, some of them will have problems. These fall into two categories: *hard* errors and *soft* errors. Hard errors are the obvious ones—the application has broken, and the visitor sees a message to that effect. This includes the ubiquitous 404 message, errors saying that the database isn't working, an abruptly terminated connection, or a sudden period of network congestion. They're easy to find if you know what you're looking for because there's a clear record of the error happening.

Soft errors are those in which the application doesn't do what you intended, for example, a site that forces users to log in twice, a navigational path that's broken, a form that won't let users submit the right data, or a page whose buttons don't work. Soft errors are more difficult to detect and more costly to diagnose because they confuse users without making it clear that the application is to blame.

In the end, errors can happen with the application, the network, or the infrastructure. With the many technical disciplines involved, resolving an outage can often mean getting everyone on a conference call to argue about why their part of the site isn't to blame. Better monitoring avoids these calls, or at least ends them quickly. This is why organizations need to start thinking of their websites as end-to-end systems, focusing on end user experience, and making web operations an integral part of a BSM strategy.

Performance problems

Your visitors will experience widely varying levels of performance. There are dozens of factors that affect the latency of a website:

- *Something specific to the user*, such as a slow connection or an old computer. This is a client-side issue best handled by customer service, site requirements, and user expectations.
- *Slowdowns on the Internet*, such as periods of congestion or connections from far away. This is a service management issue that can be addressed by choosing the right service providers, using content delivery networks (CDNs), and having a good geographic distribution strategy across your data centers.
- *Server software that's inherently slow* because it's doing something time-consuming, such as generating a report. This is a performance tuning issue that can be addressed with software engineering.
- *Infrastructure that's insufficient to handle the current traffic load* because many users are competing for resources. This is a capacity issue that can be addressed with additional servers or equipment.
- *Application issues* that are inherent to the kind of website you're running, the application server you're using, or the dependencies your application has on other systems.

The ultimate measurement of performance is how long it takes for the user to interact with the application. Slow performance isn't the only problem, however. Inconsistent performance can be particularly bad, because users learn to expect fast performance and assume your site is broken when delays occur, which amplifies abandonment.

Now that we know why performance and availability matter, let's look at how a web session works and where things go wrong. We're going to give you a working knowledge of web protocols, but we're going to keep it specific to HTTP and focus on things you need to know. If you're interested in learning more about networking, we strongly recommend Richard Stevens' book *TCP Illustrated* (Addison-Wesley Professional). For a better understanding of HTTP, check out *HTTP: The Definitive Guide* by David Gourley et al. (O'Reilly).

The Anatomy of a Web Session

Getting a single object, such as an image, from a web server to a visitor's browser is the work of many components that find, assemble, deliver, and present the content. They include:

- A *DNS server* that transforms the URL of a website (*www.example.com*) into an Internet Protocol (IP) address (10.2.3.4).
- *Internet service providers* (ISPs) that manage routes across the Internet.

- *Routers and switches* between the client and the data center that forward packets.
- A *load balancer* that insulates servers from the Web on many larger websites.
- *Servers and other web infrastructure* that respond to requests for content. These include web servers that assemble content, application servers that handle dynamic content, and database servers that store, forward, and delete content.

That's just for a single object. It gets more complicated for whole pages, and more so for visits.

Sessions, Pages, Objects, and Visits

You'll hear us talk a lot about sessions, pages, and objects as we describe web activity. Here's what we mean:

Session
: A connection between two computers established for the purpose of sending data, usually over Transmission Control Protocol (TCP).

Object
: A file retrieved from a web server. This may be a web page (*index.html*), a component of a page (*image.gif*), or a standalone object (*document.pdf* or *download.zip*). Log-based analytics tools often refer to this as a "hit."

Container object
: An object that includes references to other objects within it. A typical example is a file like *index.html* that has images, stylesheets, and JavaScript files within it.

Component object
: An object that's assigned to a container. An image embedded in a page is a component of that page. Note that some components (particularly stylesheets) can also be containers of other objects (such as background images).

Page
: A container that contains many objects (such as *index.html*, *search.php*, and so on) displayed to a user. This is what analytics tools call a "page view."

Transaction
: A series of pages across which a visitor performs a particular action, such as purchasing a book or inviting a friend.

Visit
: The extent of a visitor's interaction with a website, consisting of one or more pages (and possibly one or more transactions). Visits are normally considered terminated after 30 minutes, although this can vary tremendously depending on the type of site you operate. Analytics tools call this a "unique visit." Some tools call a visit a session, but in order to avoid confusion between visitor sessions and TCP sessions, we'll use "visit" to refer to a visitor's interaction with a sequence of pages.

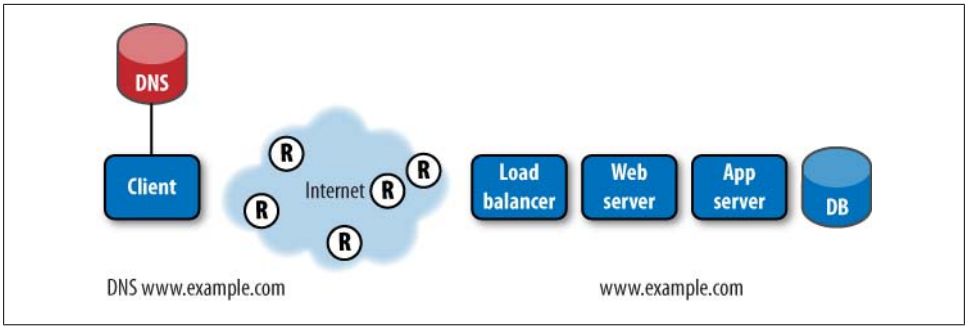


Figure 8-5. Requesting an IP address for www.example.com from a local DNS server

Finding the Destination

When you type a URL into a browser, the browser first needs to find out the IP address of the website. It does this with the Domain Name Service (DNS) protocol, which ultimately resolves the name into one or more addresses and sends the address back to your browser, as shown in Figure 8-5.

You can run your own DNS lookups by typing **`nslookup`** and the domain name you want to resolve at most command prompts. To launch a command prompt on a PC, click Start→Run, then type **`command.com`**. On a Mac running OS X, click the Spotlight icon in the top righthand corner of the screen (or press Command-Space) and type **`Terminal`**, then choose the Terminal application from the Applications list that appears.

In the following example, the IP address for yahoo.com is provided by a DNS server (cns01.eastlink.ca) as 68.180.206.184.

```
macbook:~ alistair$ nslookup yahoo.com
Server:    24.222.0.94 (cns01.eastlink.ca)
Address:   24.222.0.94#53

Non-authoritative answer:
Name:    yahoo.com
Address: 68.180.206.184
```

Here's what you need to know about DNS:

- The DNS lookup happens when the visitor first visits your site. The client's computer or his local ISP's DNS server may keep a copy of the address it receives to avoid having to look it up repeatedly.
- DNS can add to delay, particularly if your site is a mashup that has data from many places, since each new data source triggers another DNS query.
- If DNS doesn't work, users can't get to your site unless they type in the IP address—in fact, asking them to type in the IP address directly is one way to see whether the problem is with DNS.

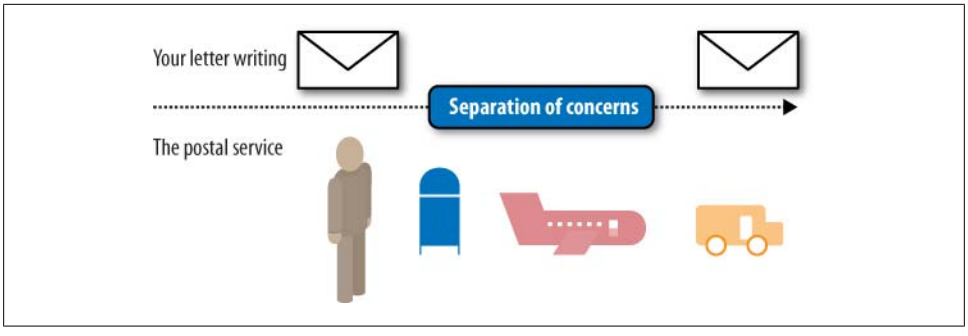


Figure 8-6. Separation of concerns between letter writing and postal services

Typing an IP address instead of a URL may not work either. If a web server is hosting multiple websites on one machine, the server needs to know which website users are looking for. Therefore, if a user requests a site using the server's IP without including the hostname in the request, he won't get the site he's looking for.

- There are many DNS servers involved in a lookup, so you may find that your site is available from some places and not others when a certain region or provider is having DNS issues.

Establishing a Connection

Armed with the IP address of the destination site, your browser establishes a connection across the Internet using TCP, which is a protocol—a set of rules—designed to send data between two machines. TCP runs atop IP (Internet Protocol), which is the main protocol for the Internet. TCP and IP are two layers of protocols.

The concept of protocol layers is central to how the Internet functions. When the Internet was designed, its architects didn't build a big set of rules for how everything should work. Instead, they built several simpler, smaller sets of rules. One set focuses on how to get traffic from one end of a wire to another. A second set focuses on how to address and deliver chunks of data, and another looks at how to stitch together those chunks into a "pipe" from one end of a network to another.

Here's an analogy to help you understand the Web's underlying protocols.

Think for a moment about the postal system—address an envelope properly, drop it in a mailbox, and it will come out the other end. You don't need to worry about the trucks, planes, and mail carriers in between (Figure 8-6). Just follow the rules, and it'll work as expected.

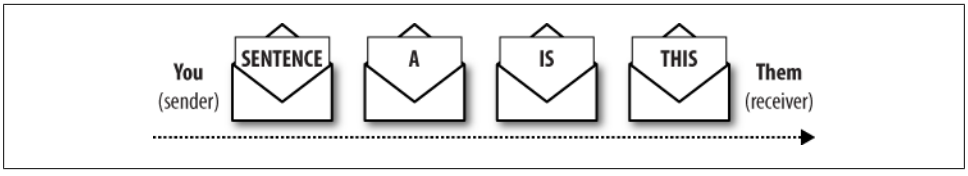


Figure 8-7. Sending a message with one word per envelope

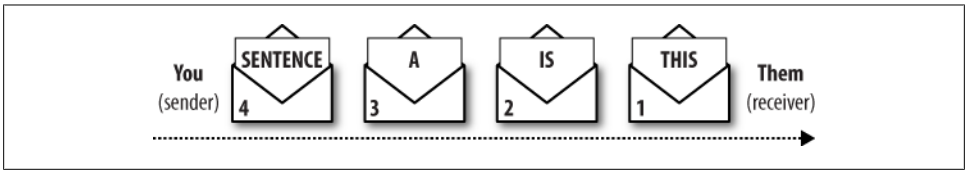


Figure 8-8. Adding a sequence number to the envelopes

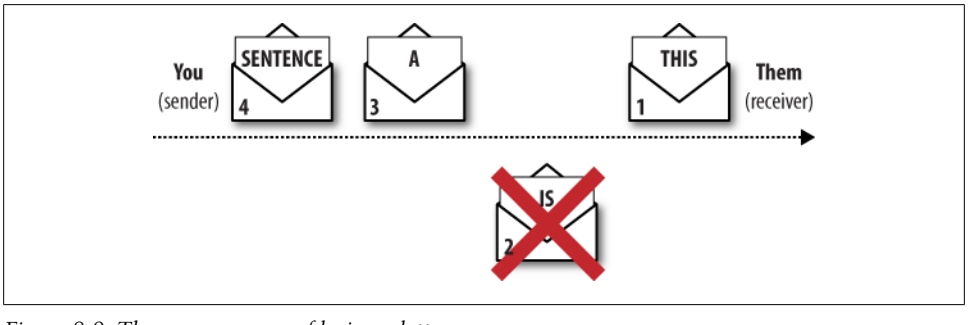


Figure 8-9. The consequences of losing a letter

Now imagine trying to write a message to someone, but having to use a new envelope for each word, as shown in Figure 8-7.

It wouldn't work. You'd have no control over the order in which the envelopes were delivered. If several envelopes arrived on the same day, or if they arrived out of order, the receiver wouldn't know in what order to open them.

To resolve this issue, you and your reader would need to agree on some rules. Perhaps you'd say, "I'll put a number on each envelope, and you can read them in the order they're numbered," (Figure 8-8).

We still have a problem: what if a letter is lost? As Figure 8-9 shows, your reader, stuck at letter 1, would be waiting for lost letter 2 to show up while the remaining letters continue to arrive.

So you'd also need a rule that said, "If you get a lot of letters after a missing one, tell me, and I'll resend the missing letter." The conversation between sender and receiver looks more complex, as Figure 8-10 illustrates, but it's also much more reliable.

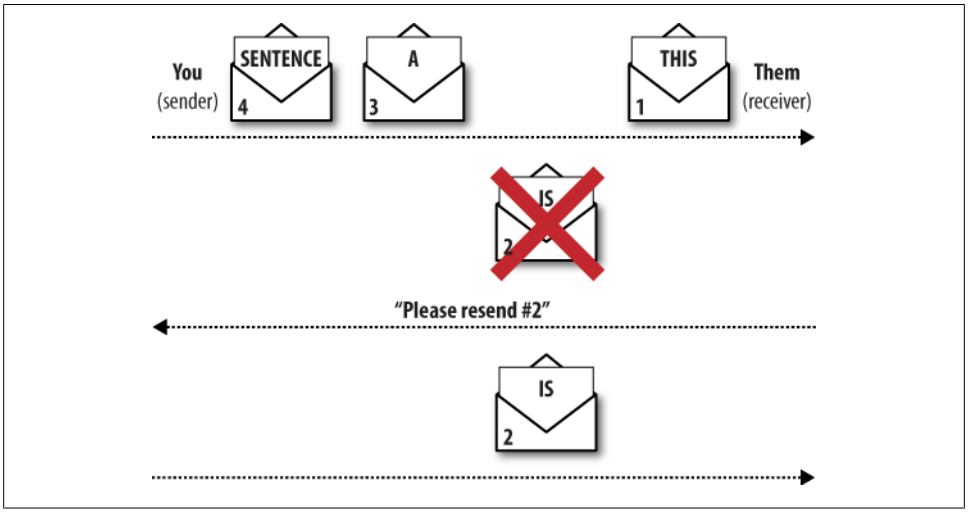


Figure 8-10. Receiver recovering from a lost letter

You might even say, "Let me know when you've received my letters," which would help you to understand how long they were taking to be delivered. If you didn't get an acknowledgment of delivery for a particular letter, you could assume it was lost, and resend it.

A set of rules like the ones we've just defined makes up a protocol. Notice that you don't need to concern yourself with how the postal service works in order to have your one-word-per-letter conversation. You don't care whether the letters are delivered by car, plane, carrier pigeon, or unladen swallow. Similarly, the postal service isn't concerned with the rules of your conversation. There is a *separation of concerns* between you and the postal service.

Put another way, your letter conversation is a "layer" of protocols, and the postal service is another "layer." You have a small amount of agreed-upon interaction with the postal layer: if you address the letter properly, attach postage, and get it in a mailbox, they'll deliver it.

In the same way, the Internet's layers have a separation of concerns. IP is analogous to the postal service, delivering envelopes of data. For your computer to set up a one-to-one conversation with a server across IP, it needs a set of rules—and that's TCP. It controls the transmission of data (that's why it's called the Transmission Control Protocol). The pattern of setting up a connection between two machines is so commonplace that we usually refer to it as the TCP/IP protocol stack. The connection between the two machines is the *TCP session*, shown in Figure 8-11.

The modern Internet is extremely reliable, but it still loses data from time to time. When it does, TCP detects the loss and resends what was lost. Just as your conversation with

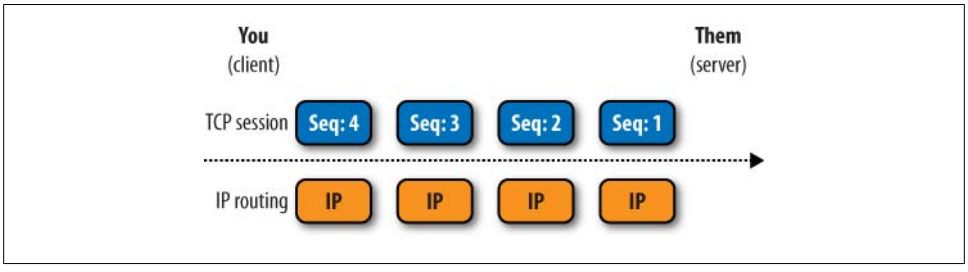


Figure 8-11. The combination of TCP and IP creates end-to-end sessions between a client and a server

a friend would slow down briefly while you resent a lost envelope, so packet loss on the Internet increases delay.

How TCP works

Because IP doesn't guarantee the order in which packets will be delivered—or even whether they'll arrive at all—TCP manages things like the sequence of delivery, retransmission, and the rate of transmission.

TCP also allows us to time the network connection, since we can measure how much time elapses between when we send a particular packet and when the receiver acknowledges receipt. This comes in handy when we're analyzing real user traffic and diagnosing problems.

You don't need to understand TCP in detail, but you should know the following:

- TCP creates an end-to-end connection between your browser and a server.
- If the network loses data, TCP will fix it, but the network will be slower.
- TCP uses a fraction of the available bandwidth to handle things like sequence numbers, sacrificing some efficiency for the sake of reliability.
- TCP makes it possible to measure network latency by analyzing the time between when you send a packet and when the recipient acknowledges it. This is what allows network monitoring equipment to report on end user experience.
- By hiding the network's complexity, TCP makes it easy for the builders of the Internet to create browsers, web servers, and other online applications we use today.

Deciding which port to use

A modern computer may have many TCP connections active at once. You may have an email client, a web browser, and a shared drive. All of them use TCP. In fact, when you surf several websites, you have TCP connections to each of them. And those sites have TCP connections to thousands of visitors' computers. To keep track of all these connections, TCP gives each of these sockets, or TCP ports, numbers.

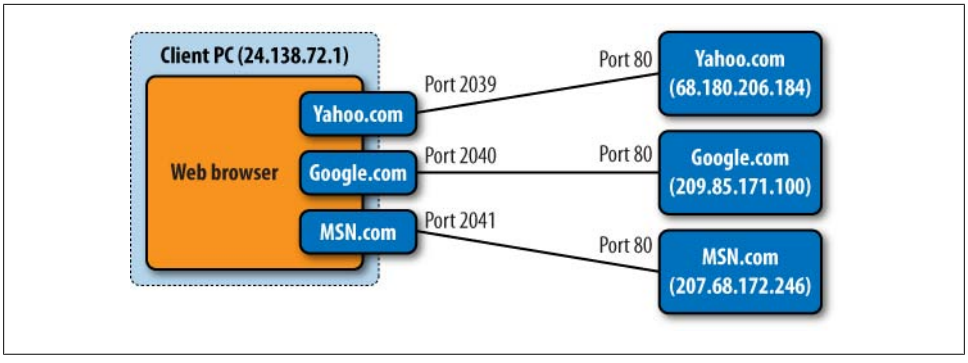


Figure 8-12. A single client connected to three web servers

For the really common applications, such as the Web, email, and file sharing, the Internet community has agreed on standard numbers. The Web is usually port 80, and encrypted web connections are usually port 443. However, you can connect to any port number on which a web server is running and ask it for content, so sometimes the web port is 8080, or 8000; it varies.

Four things uniquely identify a TCP session: the client and server IP addresses, and the TCP port numbers being used. In Figure 8-12, the TCP session to Yahoo.com is identified as 24.138.72.1:2039 to 68.180.206.184:80. No other pair of computers on the Internet has that same combination of addresses and ports at that time.

Setting up the connection

Your browser knows the IP address to which it wants to connect. And it knows it wants to connect to port 80 on the server because it wants to access that server's web content. The TCP layer establishes a session between the client running the browser and port 80 on the server whose IP it received from the initial DNS lookup. After a brief exchange of information known as a three-way handshake, there's a TCP session in place. Anything your browser puts into this session will come out at the other end, and vice versa.

You can try this out yourself. Open a command prompt and type the "telnet" command, followed by the domain name of a website and the port number, as follows:

```
macbook:~ alistair$ telnet www.bitcurrent.com 80
```

You'll see a message saying you're connected to the server, and the server will wait for your request.

```
Trying 67.205.65.12...
Connected to bitcurrent.com.
Escape character is '^]'.
```

You can request the home page of the site by typing GET, followed bya forward slash signifying the root of the site:

```
GET /
```

If everything's working properly, you'll see a flurry of HTML come back. This is the container object for the home page of the site, and it's what a browser starts with to display a web page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head profile="http://gmpg.org/xfn/11">
<script type="text/javascript" src="http://www.bitcurrent.com/
wp-content/themes/grid_focus_public/js/perftracker.js"></script>
<script>
```

After a while, you'll see the end of the HTML for the page, followed by a message showing that the connection has been closed.

```
</body>
</html>
Connection closed by foreign host.
```

In theory, you could surf the Web this way, but it wouldn't be much fun. Fortunately, browsers hide all of this complexity from end users.

Securing the Connection

There's one more thing to consider before your browser starts requesting web pages. If what you're about to send is confidential, you may want to encrypt it. If you request a page from a secure website (prefixed by https://), your browser and the server will use a protocol called the Secure Sockets Layer (SSL) to encrypt the link.

Again, you don't need to understand SSL. Here's what you do need to know:

- It makes the rest of your message impossible to read, which can make it harder for you to troubleshoot problems with a sniffer or to deploy inline collection for analytics.
- The server has to do some work setting up the connection and encrypting traffic, both of which consume server resources.
- SSL doesn't just secure the link, it also proves that the server is what it claims to be, because it holds a certificate that has been verified by a trusted third party.
- The browser may not cache some encrypted content, making pages load more slowly as the same object is retrieved with every page.
- It makes the little yellow padlock come on.

Retrieving an Object

Now you're finally ready to retrieve and interpret some web content using HTTP. Just as IP handles the routing of chunks of data and TCP simulates a connection between two computers, so HTTP focuses on requesting and retrieving objects from a server. The individual layers of communication are shown in Figure 8-13.

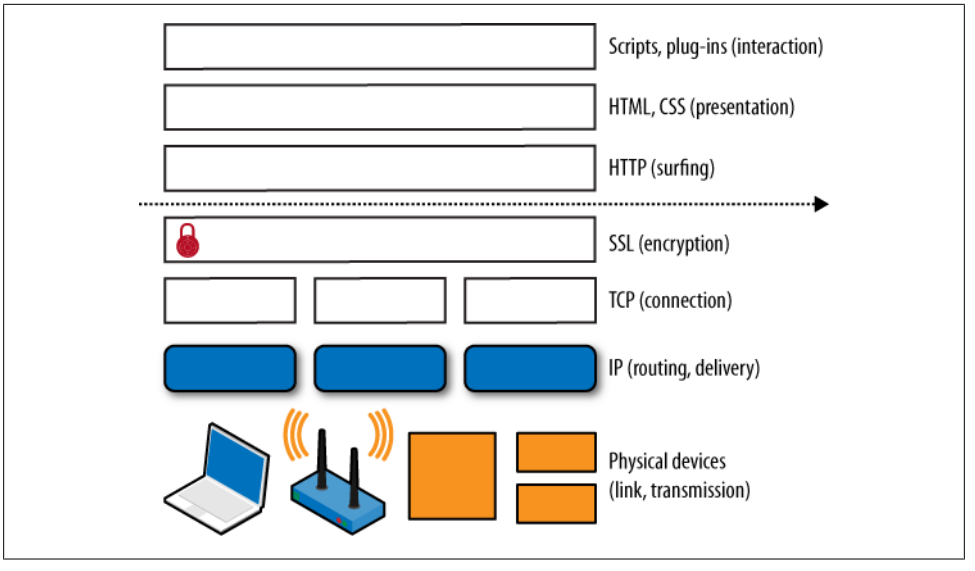


Figure 8-13. Layers of communication working together to deliver a website to a visitor

Your browser retrieves pages through a series of simple requests and responses. The browser asks for an object (*index.html*) by name. The server replies that yes, in fact, it has this object (200 OK) and sends it.

There's an important caveat here. In many cases, your browser may have already visited the website in question. Much of the content on a web page doesn't change that often. Menu bars, images, videos, and layout information are only modified occasionally. If your browser had to download every object every time it visited a site, not only would your bandwidth bill be higher, but the pages would also load more slowly. So before asking the server for an object, your browser checks to see if it has a copy in memory on your PC. If it does—and if the copy isn't stale—it won't bother asking for it. We'll look at caching in more detail shortly.

There are seven standard HTTP commands, called *methods*, that a browser can send to a server. Of these, only three are commonly used on most sites:

- GET asks for an object by name.
- POST sends data, such as the contents of a form, to the server from the client.
- HEAD asks the server to send only the descriptive information about the object, but not the object itself.

Other commands, such as PUT, TRACE, OPTIONS, and DELETE, were seldom used until recently, but PUT and DELETE have newfound popularity in Ajax-based websites.

Your browser also tells the server things about itself, such as what kind of browser it is, whether it can accept compressed documents, which kinds of objects it can display,

and which language the visitor speaks. The server can use some of this information to respond in the best way possible, for example, with a page specifically designed for Internet Explorer or one written in the visitor's native language.

The request to the server can be as simple as `GET page.html`, but it's common for a browser to send additional information to the server along with the request. This information can come in several formats. Common ones are:

Cookies
: This is a string of characters that the server gave to the browser on a previous visit. This allows the server to recognize you, so it can personalize content or welcome you back.

Information in the URL structure itself
: A request for *www.example.com/user=bob/index.html*, for example, will pass the username "bob" to the server.

A URI query parameter
: URI query parameters follow a "?" in the URL. For example, an HTTP request for *http://www.youtube.com/watch?v=Yu_moia-oVI&fmt=22* contains two parameters. The first is the video number (v=Yu_moia-oVI) and the second is the format (fmt=22, for high definition).

Remember that we said your browser only uses a cached copy of content if it's not stale? Your browser will check to see if it has a fresh copy of content, and then retrieve objects only if they're more recent than the ones stored locally. This is called a *conditional GET request*: the server only sends an object if the one it has is more recent than the one the browser has, which saves on bandwidth without forcing you to see old content.

This is one of the reasons why, when testing site performance, you need to specify whether the test should simulate a first-time or a returning user. If a browser already has much of the content, performance will be much better because the browser already has a copy of many parts of the page.

The initial response: HTTP status codes

If everything is running smoothly, the server will respond with an HTTP status code.

```
HTTP/1.0 200 OK
```

This is a quick message to acknowledge the request and tell the client what to expect. Status codes fall into four groups:

- The *200* group indicates everything is fine and the response will follow.
- The *300* group means "go look elsewhere." This is known as a *redirect*, and it can happen temporarily or permanently. It's used to distribute load across servers and data centers, or to send visitors to a location where the data they need is available. Instead of responding with a "200 OK" message, the server sends back a redirection to another web server.

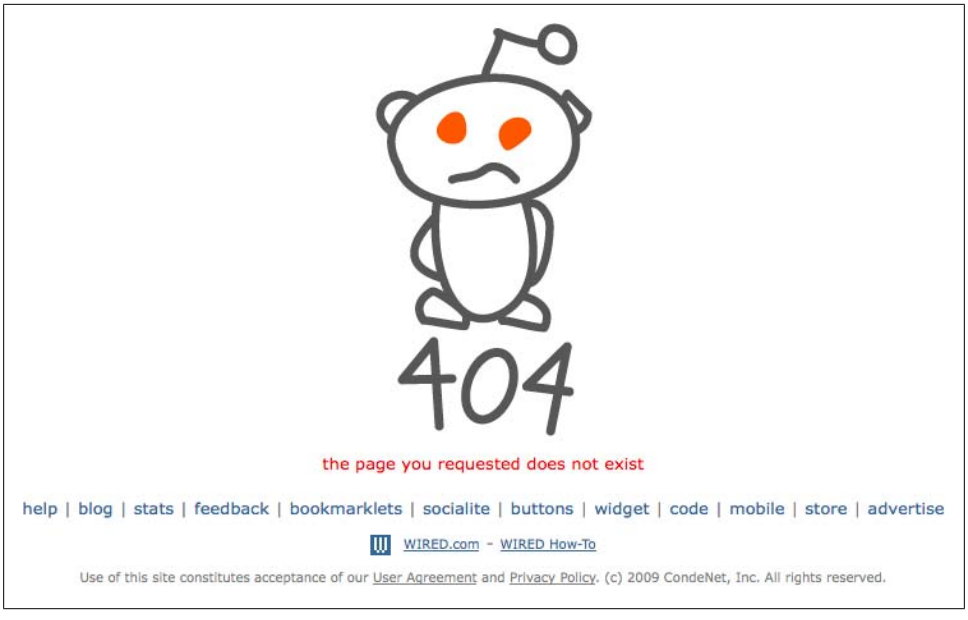


Figure 8-14. A 404 apology page offering suggested links to the visitor

- The *400* group indicates client errors. The client may have asked for something that doesn't exist, or the client may not have the correct permissions to see the object it has requested. While you might think that 400 errors are a visitor's problem, they can be the result of broken links you need to fix. Figure 8-14 shows an example of a 404 apology page.
- The *500* group indicates server errors. These are serious issues that can occur because of application logic, broken backend systems, and so on. Sometimes these errors can produce custom pages to inform visitors of the problem or to reassure them that something is being done to fix the issue, as shown in Figure 8-15.

418 I'm a Teapot

On April 1, 1998, Larry Masinter, currently principal scientist at Adobe, wrote an RFC for an HTTP extension. The Rationale and Scope reads as follows:

"There is coffee all over the world. Increasingly, in a world in which computing is ubiquitous, the computists want to make coffee. Coffee brewing is an art, but the distributed intelligence of the web-connected world transcends art. Thus, there is a strong, dark, rich requirement for a protocol designed espressoly [sic] for the brewing of coffee. Coffee is brewed using coffee pots. Networked coffee pots require a control protocol if they are to be controlled."

The RFC goes on to describe the 418 status code in detail at *http://tools.ietf.org/html/rfc2324*.

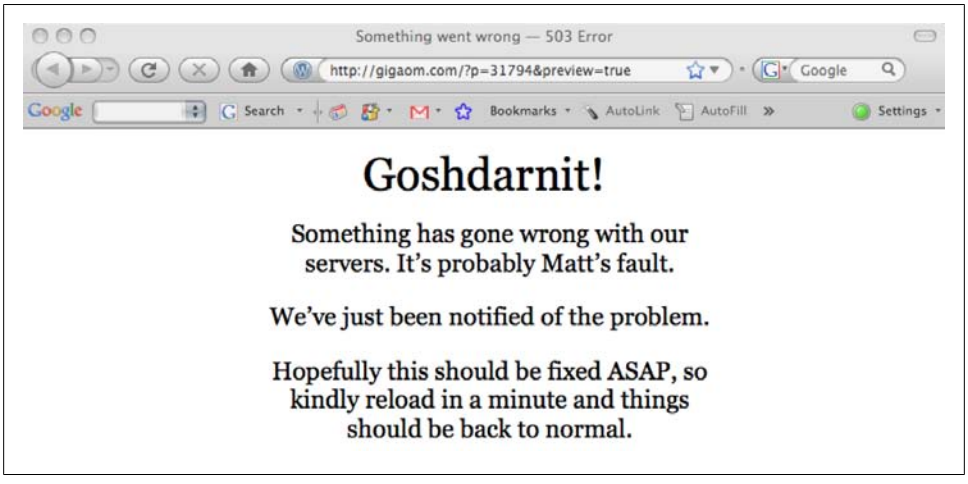


Figure 8-15. An application error apology page on a Wordpress server

Object metadata: Describing what's being sent

Just before it sends the object, the server gives us some details about the object the browser is about to receive. This can include:

- The size of the object
- When it was last modified (which is essential to know in order to cache objects efficiently)
- The type of content—a picture, a stylesheet, or a movie, and so on—so your browser knows how to display it
- The server that's sending the answer
- Whether it's OK to cache the object, and for how long to use it without checking for a fresh object from the server
- A cookie the client can use in subsequent requests to remind the server who it is
- Any compression that's used to reduce the size of the object
- Other data, such as the privacy policy (P3P) of the site

Here's an example of metadata following an HTTP 200 OK response:

```
HTTP/1.x 200 OK
Date: Mon, 13 Oct 2008 04:31:29 GMT
Server: Apache/2.2.4 (Unix) mod_ssl/2.2.4 OpenSSL/0.9.7e
Vary: Host
Last-Modified: Wed, 13 Jun 2007 19:15:36 GMT
Etag: "e36-70534600"
Accept-Ranges: bytes
Content-Length: 3638
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
```

```
Content-Type: image/x-icon
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR
 CUR ADM DEV TAI PSA PSD IVAi IVDi CONi TELo OTPi OUR DELi SAMi
 OTRi UNRi PUBi IND PHY ONL UNI PUR FIN COM NAV INT DEM CNT STA POL HEA PRE GOV"
```

Each line is a *response header*. Following the headers, the server provides the object the client requested. One of the reasons for HTTP's widespread success is the flexibility this affords: the server may provide many headers that your browser simply ignores. This has allowed browser developers and server vendors to innovate independently of one another, rather than having to release browsers and servers at the same time.

Preparing the response

Now the server has to send the object your browser requested. This may be something stored in the server's memory, in which case it's a relatively simple matter of retrieving it and stuffing it into the existing TCP session. Your browser will do the rest.

On most modern sites, the web page contains some amount of dynamic content. This means the server has to think about the answer a bit. This server-side processing—what we'll refer to here as *host time*—is a major source of poor performance. Assembling a single object may require the web server to talk to other servers and databases, and then combine everything into a single object for the user.

Once the object is prepared, the server sends it to your browser.

Sending the response

The other major source of poor performance comes from network latency (what we'll refer to as *network time*). From the moment the server puts the object into the TCP connection until the moment the last byte of that object comes out at the other end, the clock is ticking. Large objects take longer to send; if anything's lost along the way, or if the sender and receiver are far apart, they'll take longer still.

Your browser receives the object. If the request was for a standalone object, that would be the end of it: your browser would shut down the TCP connection through a sequence of special TCP messages, and you'd be able to access the received object (i.e., the web page).

Getting a Page

Pages contain many things, such as pictures, video, and Flash, as well as formatting information (stylesheets) and programming logic (JavaScript). So your browser now has to finish the job by retrieving the components of the container object it received.

Modern browsers are eager to display pages quickly. Your browser doesn't even wait for the container object to load completely—as it's flowing in, the browser is greedily analyzing it to find references to other objects it needs to retrieve. Once it sees additional objects listed in that page, it can go ahead and request them from the server, too. It can

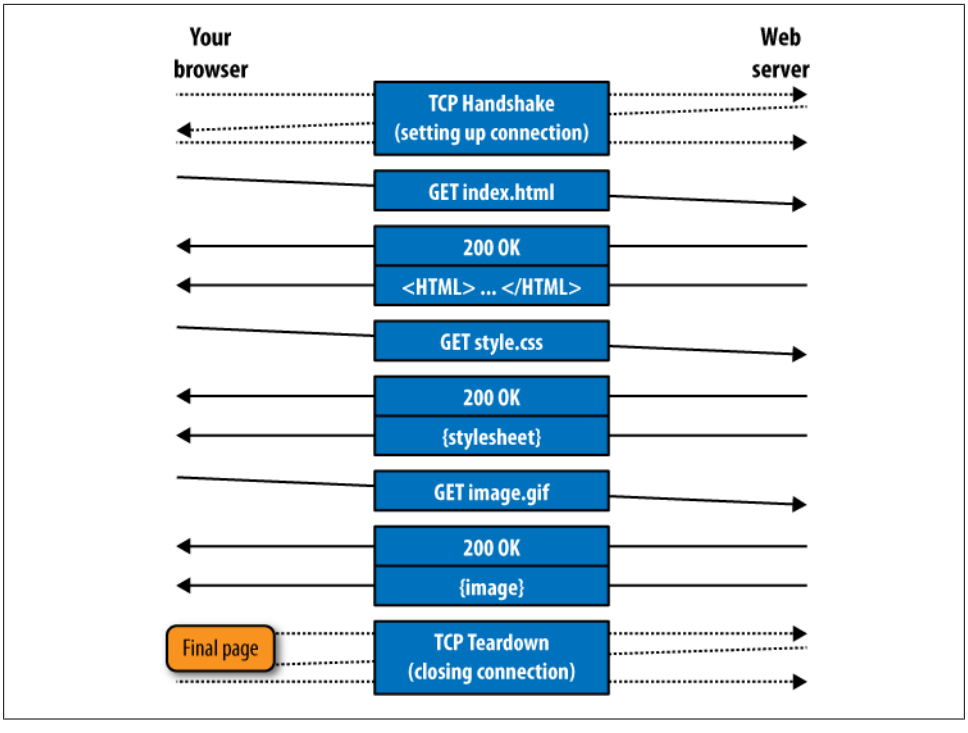


Figure 8-16. A bounce diagram showing the retrieval of a page and its component objects

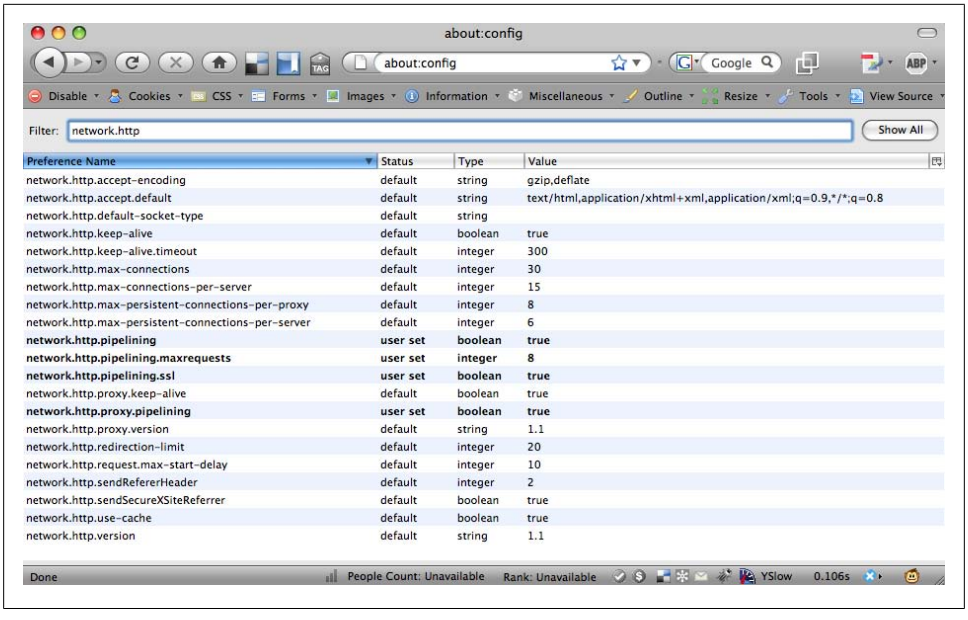
also start executing any JavaScript that's on the page and launching plug-ins, such as Java, ActiveX, or Flash, that it will need to display content.

With the client and server communicating so much, it would be a waste to tear down that TCP session just to reestablish it. Version 0.9 of HTTP initially behaved in this way. If your browser needed more objects (and it usually does), it had to reconnect to the server. Not only was this slow, it also kept the servers busy setting up and tearing down connections. The Internet's standards body moved quickly to correct this.

In version 1.1 of the HTTP protocol, the TCP connection is kept alive in anticipation of additional requests. Figure 8-16 shows this process.

Doing lots of things at once: Parallelism

The examples we've seen so far are serial—that is, the browser and the web server are requesting and sending one object at a time. In reality, web transactions run in parallel. Here's why: if you're on a one-lane road, it's hard to overtake someone. Add a second lane, and suddenly traffic flows freely. The same thing is true with network connections. A single TCP session means that a really big object can stop smaller objects from getting through quickly. This is known as *head-of-line blocking*, and is similar to a single slow truck delaying dozens of faster cars.



Preference Name	Status	Type	Value
network.http.accept-encoding	default	string	gzip,deflate
network.http.accept.default	default	string	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
network.http.default-socket-type	default	string	
network.http.keep-alive	default	boolean	true
network.http.keep-alive.timeout	default	integer	300
network.http.max-connections	default	integer	30
network.http.max-connections-per-server	default	integer	15
network.http.max-persistent-connections-per-proxy	default	integer	8
network.http.max-persistent-connections-per-server	default	integer	6
network.http.pipelining	**user set**	**boolean**	**true**
network.http.pipelining.maxrequests	**user set**	**integer**	**8**
network.http.pipelining.ssl	**user set**	**boolean**	**true**
network.http.proxy.keep-alive	default	boolean	true
network.http.proxy.pipelining	**user set**	**boolean**	**true**
network.http.proxy.version	default	string	1.1
network.http.redirection-limit	default	integer	20
network.http.request.max-start-delay	default	integer	10
network.http.sendRefererHeader	default	integer	2
network.http.sendSecureXSiteReferrer	default	boolean	true
network.http.use-cache	default	boolean	true
network.http.version	default	string	1.1

Figure 8-17. The About:config panel in Firefox, showing connection properties that users can adjust to change connection parallelism

Web browsers resolve this issue by establishing multiple TCP sessions to each server. By having two or more sessions open, the client can retrieve several objects concurrently. Some browsers, such as Firefox, let users configure many of these properties themselves, as shown in Figure 8-17. For more information about parallelization and pipelining, see Mozilla's FAQ entry at *www.mozilla.org/projects/netlib/http/pipelining-faq.html*.

TCP parallelism makes browsers work better, but it also makes your job of measuring things more difficult. Because many HTTP transactions are occurring simultaneously, it's harder to estimate performance. Many web operations tools use *cascade* diagrams like the one in Figure 8-18 to illustrate the performance of a test or a page request. Notice that several objects (*script.js* and *style.css*, for example) are being retrieved simultaneously.

Interpreting the page

Early browsers simply displayed what they received. With the introduction of JavaScript in 1993, developers started to embed simple programs within the container page. Initially, they focused on making pages more dynamic with functions such as rollover images, but these quickly grew into highly interactive web pages.

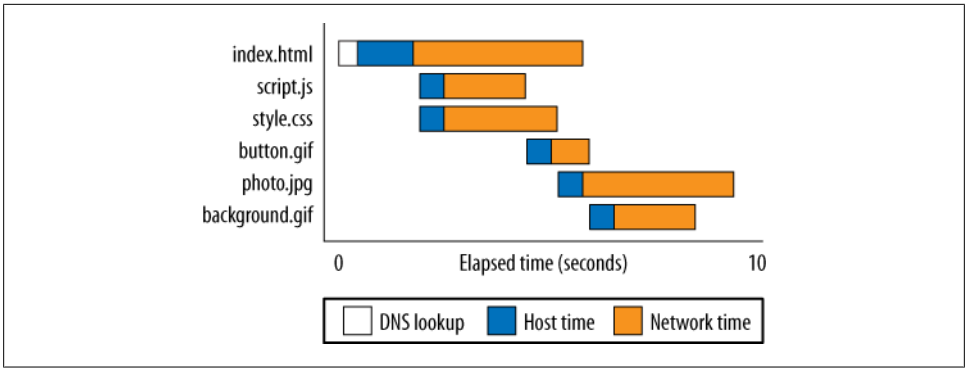


Figure 8-18. A simple cascade diagram for a page

Since then, JavaScript has become a powerful programming language in its own right. Interactive sites like Google Maps rely heavily on browser scripting to make their applications run quickly on the client.

At the time a page is loaded, the browser interprets the portions that are written in JavaScript. Because this may involve rewriting portions of the page that were just loaded, JavasScript needs to run first. That's right: a page can contain a program that actually modifies the page itself.

Why Should JavaScript Rewrite a Page?

There are times when it makes sense to have JavaScript actually rewrite much of the page that loaded it. If a page contains large quantities of repetitive data that can be generated programmatically, it may be faster and less taxing on network connections to have the browser manufacture the page.

Consider, for example, a page with a 100 × 100 cell table in it. Rather than sending the HTML for the table (which could contain 10,000 lines of `<td></td>` markup in it and would take a long time to deliver), JavaScript can create the HTML for that table once the page has arrived at the client.

Compared to the 10,000 cells of the original table, the code is relatively small. And by getting the browser to do some of the heavy lifting, the server frees itself up to do other things. If you use this approach, you must test it carefully and use it sparingly; in fact, with compression, the advantages of such an approach can be negligible at best.

Assembling the objects

Once your browser has the page and enough information about the objects it contains, it can display them to the user. Note that the browser doesn't need all the objects on the page to show you the page—if your browser knows how big an image will be, it can begin displaying the page, leaving a space where the image will go. Including image sizes within the container page means the browser can show you the page sooner.

A timeline of page milestones

There are several important milestones in the loading of a page, and these are the things you'll be monitoring with synthetic and RUM tools. They don't always happen in this order, and their order depends on how a particular site was built.

Initial action
: This is the action that started it all—either typing in a URL or clicking on a link. It's the basis for all page measurement.

Initial DNS response time
: If the browser is visiting a site for the first time, there must be a DNS lookup.

SSL negotiation
: If the page is encrypted, the browser and server need to set up the encrypted SSL channel before transmitting data.

Container first byte
: This is when the start of the container object reaches the browser. The time from the initial action to the first byte of the container can serve as a rough estimate of how much time the server took to process the request.

Component objects requested
: At this point, the browser knows enough to start asking for embedded elements, stylesheets, and JavaScript from the server.

Component first byte
: This is when the start of the component reaches the browser.

Third-party content DNS lookups
: If any of those component objects come from other sites or subdomains, the browser may have to look them up, too.

Start processing instructions
: This is when the browser starts interpreting JavaScript, possibly triggering other retrievals or page modification. This may block other page activity until the scripts are finished.

Page starts rendering
: This is the point at which the browser has rendered the page sufficiently for a user to start interacting with it. This is the critical milestone that governs end user experience.

All objects loaded
: Some objects, such as those below the viewable screen, may continue to load even after the user starts interacting with the page.

Entire page loaded, including backup resources
: To improve usability, scripts on the page may intentionally fetch some data after the page is completed.

In-page retrieval (type-ahead search, etc.)
: Many applications may retrieve additional components in response to user interactions. Microsoft's Live Maps, for example, retrieves new map tiles as users scroll.

Getting a Series of Pages

Few visits last for a single page. Each user action may trigger another page load, with a few differences from what happened when the initial page was loaded:

DNS lookups are unnecessary
: You shouldn't need to look up the IP address again through DNS.

Some content will be in the local cache
: There will hopefully be plenty of up-to-date content in the local cache, so there will be less to retrieve.

New cookies to send
: The browser may have new cookies to send this time around, particularly if this was the first time the user visited the site.

Wrinkles: Why It's Not Always That Easy

What we've just outlined is a relatively straightforward process. If everything goes as planned, the page will load properly. Unfortunately, things often go awry. Much of the work of EUEM is in smoothing out the many wrinkles that can happen along the way.

DNS Latency

DNS can take time to resolve, or fail to resolve entirely. If users can't get your IP address, they can't visit you. One of the primary tasks of a web operations team is to manage the propagation of IP addresses through DNS—not just making sure they're fast, but also that they're correct and that they can be changed relatively smoothly.

On some systems, you can use the `dig` command line utility to measure DNS latency (the `dig` utility is included in the Windows distribution of BIND, available at *www.isc.org/software/bind*).

```
macbook:~ alistair$ dig www.brainpark.com

; <<>> DiG 9.4.2-P2 <<>> www.brainpark.com
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 23382
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.brainpark.com.     IN    A

;; ANSWER SECTION:
```

```
www.brainpark.com. 1800 IN    CNAME  brainpark.com.
brainpark.com.     1800 IN    A      174.132.128.251

;; Query time: 98 msec
;; SERVER: 24.222.0.94#53(24.222.0.94)
;; WHEN: Sat Feb  7 16:25:00 2009
;; MSG SIZE  rcvd: 65
```

In this example, a DNS lookup for Brainpark.com returned a single IP address, and took 98 milliseconds.

DNS Latency for the Practical Man

If you do anything over the Internet, whether it's email or web browsing or pirating movies, DNS lookups are part of it. If you can't look up domain names, you can't surf the Web, because your browser doesn't know where to go. If your email server can't lookup domain names, it just holds on to your mail until it can. While it's supercritical to be able to do DNS lookups at all, there's a lot of redundancy built in, so it works almost all the time. So the next question is, "How fast does the DNS need to be?"

(The answer is going to be 20 milliseconds—that's 1/50 of a second, or less. But let's take the scenic route to explain how we get there.)

To keep a user happy, the answer depends on the activity.

Sometimes, it just doesn't matter. If you are going to spend hours downloading a multigigabit HD movie, any reasonable DNS lookup or lookups are just lost in that large transfer time. In general, whenever DNS is a small part of the transaction, it isn't an issue.

For web browsing, a trivial web page uses one DNS lookup, so if my DNS service has a 1/10 of a second latency (the time between sending a DNS query and getting the corresponding response) there's a 1/10 of a second, or 100 millisecond, DNS time tax. But that's a 1900s-style web page. Today, a really simple web page probably uses about 10 DNS lookups—a few for the content, and many more for the advertisements and other inserts. If each lookup takes 1/10 of a second, 10 might seem to add 1 second to the time it takes to view a page.

However, today's web browsers are clever and do the DNS lookups in parallel, usually up to some limit. So 10 lookups might be done as 2 convoys of 5 each, adding 2/10 of a second. That's barely noticeable compared to the time it takes to send the data, and probably not annoying. For a complicated Facebook page, there might be 200 DNS lookups, adding 4 seconds, and that's into the annoying range. Of course, once the DNS lookups get cached (i.e., remembered in a local copy) you stop paying for the lookups. So the good news is once your browser locates the ad servers, you only wait for the ads, rather than the DNS.

If I'm sending email, I may not care if the email server has slow DNS, since it rarely matters if there's a delay of minutes—of course, the systems administrator may care a lot if his email server gets clogged or he has to buy more servers. The typical email takes around a dozen DNS lookups for every hop it takes between sender and receiver, and three to fours hops is not unusual, so it adds up. Spammers generate a lot of DNS

lookups, even for mail that only goes a hop or two before being discarded. Not to be outdone, antiqspammers can use 5–10 DNS lookups to vet email by checking to see if the sender is on any of multiple "blacklists" of spammers.

So 20 ms is really a rule of thumb that seems to keep users happy and not cost service providers too much in terms of DNS server hardware and software.

But when ISPs want to sell their service as blindingly fast, even faster is better. An idle, reasonably hot PC these days answers queries in a very small number of milliseconds if it has good software (some is slow), so the time it takes to do a query is often dominated by waiting in line behind other users' queries at the server, or just how long it takes to move through the routers and cables between the user's machine and the DNS server. But the good news is that as the network's speed increases and processors get more powerful, it's possible to generate faster and cheaper DNS queries to keep up with the applications' growing appetites.

—Paul Mockapetris,
author of DNS and the chairman of Nominum

There are a number of DNS testing tools, and managed DNS services like UltraDNS, that can help you monitor DNS performance. Most synthetic testing tools will test the DNS component of page retrieval as part of their normal testing.

Multiple Possible Sources

Most big websites are hosted in several places. This is done for two main reasons. First, it spreads out the load and reduces the impact of a major outage. Second, it puts computers closer to users, which reduces the network delay when accessing websites.

Sometimes, the DNS response of your site will include several IP addresses. If the first address doesn't work, your browser can, in theory, try the others. Google, for instance, has four IP addresses that a browser can use:

```
macbook:~ alistair$ nslookup www.google.com
Server:  24.222.0.94
Address: 24.222.0.94#53

Non-authoritative answer:
www.google.com canonical name = www.l.google.com.
Name:    www.l.google.com
Address: 64.233.169.147
Name:    www.l.google.com
Address: 64.233.169.103
Name:    www.l.google.com
Address: 64.233.169.104
Name:    www.l.google.com
Address: 64.233.169.99
```

In practice, however, users won't wait long enough for the first site to time out—they'll have already gone elsewhere or tried to reload the page. DNS is the first line of defense against regional failure and global delay. This takes two forms: Global Server Load Balancing (GSLB) and CDNs.

Global Server Load Balancing

Instead of sending everyone on the planet to a single destination, large sites can send visitors to the servers closest to them. In GSLB, the DNS server decides which of the data centers is "best" for the visitor. This depends on several factors:

Whether the data center has the information the visitor needs
: Not every piece of content is available in every data center. There may also be legal constraints or pricing policies that require you to send certain users to certain locations.

The delay between the visitor and the data center
: This may be based on geography or on network round-trip time.

The responsiveness of the servers in each data center
: There's no point in sending a user to a data center that's nearby if the servers there are unacceptably slow.

GSLB doesn't always rely on DNS, however. Sometimes the web server needs to first determine what content a visitor wants, then redirect the visitor's browser to the optimal data center. The server can't rely on DNS-based GSLB, because it has to first receive the request from the browser, interpret it, then respond with an HTTP message sending the visitor elsewhere. This is a popular method because it overcomes the staleness and address caching issues that plague DNS-only GSLB. However, it introduces a redirect delay into the start of each session since the client must now request a page from another server (often with an additional DNS query).

Content delivery networks

You might want multiple locations for your data, both for resiliency and for better network performance, but it's expensive to deploy infrastructure in several locations around the world.

CDN companies are willing to help—for a price. Think of them as alternate networks that are tuned for fast delivery and circumvent many of the bottlenecks normal traffic faces. CDNs don't actually have parallel networks to the Internet. CDNs have two main strategies to speed up the Internet for their paying customers:

- *Get data close to users* to reduce the amount of information that must cross the Internet.
- *Speed Internet traffic up* as much as possible for the data that must cross the network.

Some CDN providers focus on making sure that all the content users need is at the edge; others focus on making sure that data travels quickly over the wide area network (WAN); and some blend the two functions.

To get data close to users, a CDN will:

- Handle the DNS lookup process on behalf of its subscribers, directing visitors to the closest place from which to serve data.
- Cache data at the edge, moving content that doesn't change often—such as images or MP3s—closer to users, where it can be served more quickly.
- Execute some processing on machines that the CDN owns, close to visitors. This is known as an *edge-side include*, and it's useful when a website needs to assemble data from several sources within a page.

To speed up Internet traffic, a CDN will:

- Choose better paths across the network, changing routes more rapidly based on better measurements than the Internet as a whole.
- Optimize for specific applications (such as video or web traffic) by tweaking certain parameters of protocols rather than offering one-size-fits-all handling (as the Internet does).
- Use the latest HTTP protocols to send multiple requests at once and compress the results.
- Send redundant packets along different paths to overcome sporadic packet loss.
- Bypass congestion points by connecting to multiple carriers and service providers rather than relying on the carriers to send users' traffic from one to another.
- Split single data streams at the edge to reduce the amount of traffic at the core, usually for streaming content.

A CDN takes over your DNS, and (hopefully) speeds it up along with the rest of your site. Many of the techniques just outlined mitigate or overcome the performance and availability bottlenecks that we're about to see—so if performance is important to your business, you need a CDN. While CDNs were once a daunting prospect requiring long-term contracts, recent developments by cloud computing firms like Amazon are turning them into a pay-as-you-go acceleration utility for web applications.

Slow Networks

Network latency is the result of two things: the time it takes to complete a trip across the network (round-trip time), and the number of times, or turns, the application needs to traverse it. Simply put, turns multiplied by round trips, equals network latency. All networked applications can be analyzed in this way. To improve your application's performance you need to either reduce the round-trip time or reduce the number of turns.

How Fast Can Networks Get?

There's an upper limit to web performance: the speed of light. That's as fast as data can travel, and it means a packet sent from New York will arrive in Las Vegas no sooner than 13 milliseconds later.

Or is it?

Quantum effects suggest that we may be able to send data instantly. We want to avoid being laughed at in the future, so we'll just point to *www.physorg.com/news137937526.html* and say we told you so.

Network connections seldom travel at the speed of light (see the sidebar "How Fast Can Networks Get?"), and there's lots of room for improvement in the effect networks have on end user experience.

An extremely simple object request, such as a GET for a tiny image, would take exactly one round trip to retrieve, assuming the server responded instantaneously. On a LAN, this is a matter of milliseconds. On an Internet connection, it's far greater. Let's compare a device on a LAN to one in China by running a ping test from both locations.

```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.250 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.995 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=1.974 ms
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.974/2.073/2.250/0.125 ms

$ ping yahoo.cn
PING yahoo.cn (202.165.102.247): 56 data bytes
64 bytes from 202.165.102.247: icmp_seq=0 ttl=44 time=367.263 ms
64 bytes from 202.165.102.247: icmp_seq=1 ttl=44 time=428.287 ms
64 bytes from 202.165.102.247: icmp_seq=2 ttl=44 time=399.339 ms
64 bytes from 202.165.102.247: icmp_seq=3 ttl=44 time=382.795 ms
--- yahoo.cn ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 367.263/394.421/428.287/22.604 ms
```

The round-trip time to China is 190 times greater than the round-trip time to a local device.

In addition to geographic distance, many other factors can affect round-trip time, such as low bandwidth connections, intermediate devices, and congestion.

Low bandwidth

A 14.4 Kbps modem would send a 5 MB file between two computers in about six minutes; the same object would take only five *milliseconds* over a dedicated Gigabit Ethernet network.

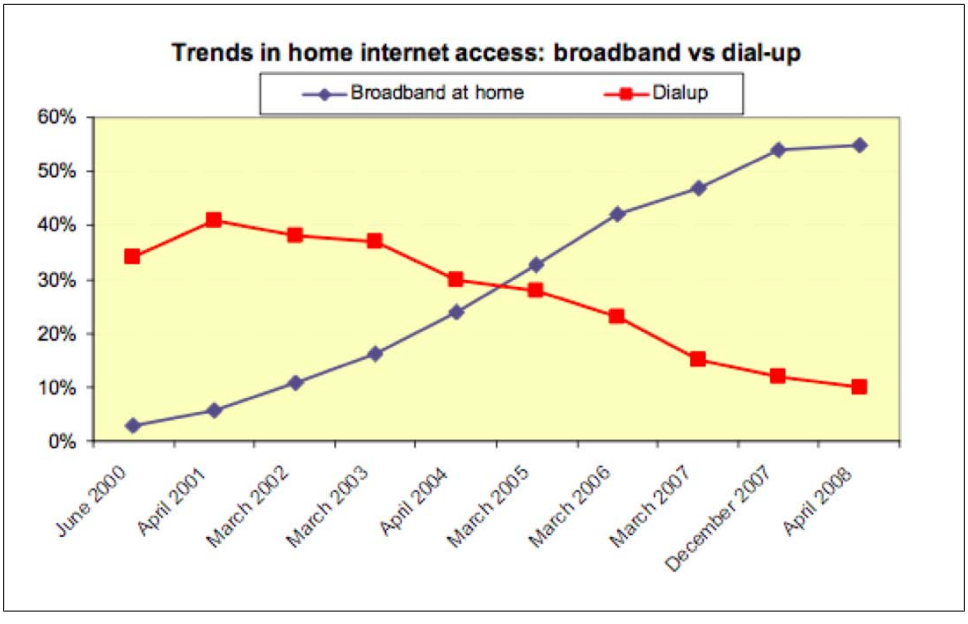


Figure 8-19. Consumer Internet connections are getting faster

Increasingly, we're surfing on faster connections using faster machines, as shown in Figure 8-19 (the Pew Internet and American Life Project, Home Broadband Adoption 2008 survey, *http://www.pewinternet.org/Reports/2008/Home-Broadband-2008.aspx*).

Despite our faster connections, performance issues plague the Internet at both the edge and the core. You can estimate how much of your population has which kind of bandwidth—and tweak your site design accordingly—using analytics or RUM tools, because the speed at which the web page loads will usually be subject to the bottleneck of the visitor's connection. This is known as the "last mile" of the connection, and it's generally the slowest.

Hops—devices between your browser and your destination

A broadband connection is no guarantee of a fast Internet. As a packet travels to the server, it traverses several routers, each of which knows where to send it next. Pairs of routers are connected by wires, sometimes in the same room, sometimes on opposite shores of an ocean. When a site slows down, it's often because one of these connections is slow. A command-line utility called *traceroute* can show you what's going on.

Traceroute Is a Hack

To prevent packets from circulating forever and filling up the tubes of the Internet, every packet on the Internet keeps track of how many routers have forwarded it. Each time a router forwards a packet, it decreases the hop count of that packet by one. When

the hop count reaches zero, the router deems the destination unreachable and sends a message (called using an ICMP TIME_EXCEED) back to the sender.

Traceroute exploits this feature by fiddling with the hop count of packets it sends. The first packet has a hop count of one, which the first router decreases. Realizing the hop count is now zero, the router sends back a message to the sender pronouncing the destination unreachable.

Traceroute now knows the IP address of the first router—after all, it just received a message from it. In addition to looking up the IP address of the router to find out its name (if it has one), traceroute can now increment the hop count by one and discover the second router. It continues this process until it gets a response from the destination, as shown in Figure 8-20.

Traceroute can use several different methods, depending on which protocol you want to test. Probes can be sent with UDP, TCP, or ping (ICMP) protocols. Depending on how intermediate devices treat different protocols, you may get different results. For example, a firewall may block UDP packets but allow pings to pass, meaning a UDP-based traceroute won't discover devices behind the firewall.

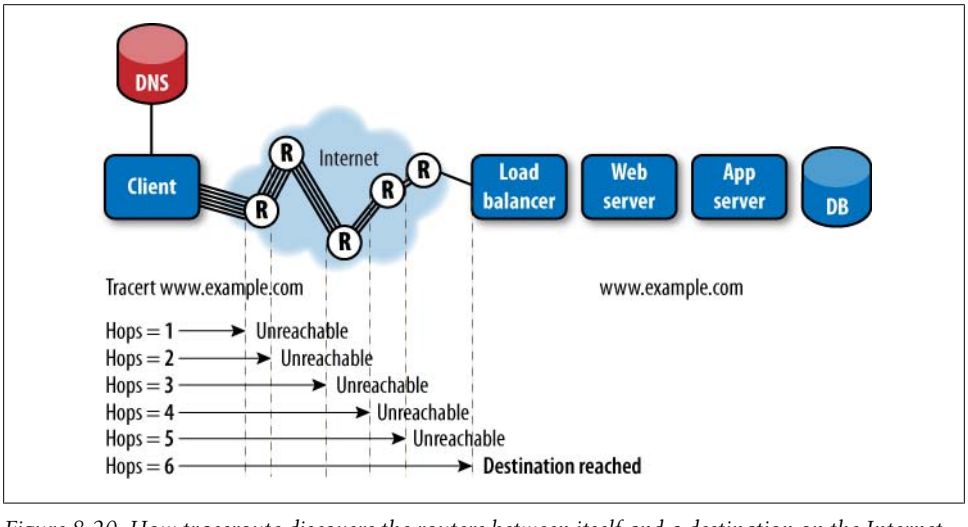


Figure 8-20. How traceroute discovers the routers between itself and a destination on the Internet

Let's look at an example of traceroute in action. First, we'll run a test from a home network located in Montreal to *www.brainpark.com*.

```
macbook:~ sean$ traceroute www.brainpark.com
traceroute to brainpark.com (208.71.139.130), 64 hops max, 40 byte packets
 1  192.168.0.1 (192.168.0.1)  1.102 ms  0.693 ms  0.566 ms
 2  10.183.128.1 (10.183.128.1)  5.655 ms  6.787 ms  8.134 ms
 3  24.200.227.77 (24.200.227.77)  5.510 ms  5.719 ms  9.674 ms
 4  24.200.250.82 (24.200.250.82)  8.187 ms  6.630 ms  5.733 ms
 5  ia-cnnu-bb04-ge13-1-0.vtl.net (216.113.122.14)  7.195 ms  7.696 ms  9.712 ms
```

```
 6  POS1-0.PEERB-MTRLPQ.IP.GROUPTELECOM.NET (66.59.191.157)  5.706
    ms  5.773 ms  6.117 ms
 7  GE4-0.WANA-MTRLPQ.IP.GROUPTELECOM.NET (66.59.191.137)  7.513 ms
    5.870 ms  5.863 ms
 8  POS4-0.WANA-TOROONXN.IP.GROUPTELECOM.NET (66.59.191.226)  14.216
    ms  16.317 ms  14.184 ms
 9  POS5-0.PEERA-CHCGIL.IP.GROUPTELECOM.NET (66.59.191.106)  26.245 ms
    28.390 ms  26.594 ms
10  GT-360NETWORKS.PEERA-CHCGIL.IP.GROUPTELECOM.NET (66.59.191.90)
    53.759 ms  52.112 ms  52.016 ms
11  lau1-core-01.360.net (66.62.7.1)  93.512 ms  93.913 ms  93.927 ms
12  pdx1-core-01.360.net (66.62.3.13)  214.342 ms  101.376 ms  199.833 ms
13  slc1-core-01.360.net (66.62.3.9)  88.250 ms  88.970 ms  87.604 ms
14  slc1-edge-01.360.net (66.62.5.3)  87.971 ms  87.835 ms  88.154 ms
15  66.62.56.26 (66.62.56.26)  101.184 ms  100.683 ms  99.282 ms
16  qwk.net (208.71.136.2)  95.589 ms  96.070 ms  95.583 ms
17  mail4.qwknetllc.com (208.71.139.130)  95.969 ms  94.018 ms  96.740 ms
```

In the resulting output, the first column shows the hop count—the number of routers between the client and the server. The second column shows the router at that hop, identified by an IP address and possibly a DNS address. This can sometimes provide useful clues (for example, POS4-0.WANA-TOROONXN.IP.GROUPTELECOM.NET suggests that this is a router in Toronto, Ontario, belonging to Group Telecom).

The third column shows the timing of three probes sent to that router. The last row in the output shows the final gateway to the destination address, which is 17 hops away.

If we try the same test from a server in Hong Kong, however, latency and hop count change significantly:

```
traceroute to brainpark.com (208.71.139.130), 30 hops max, 40 byte packets
 1  tmhs3506-v1.pacific.net.hk (202.14.67.252)  0.708 ms  0.611 ms
 2  v206.tmhc2.pacific.net.hk (202.64.154.65)  0.336 ms  0.310 ms
 3  v102.wtcc2.pacific.net.hk (202.64.5.6)  100.642 ms  0.972 ms
 4  202.64.3.142 (202.64.3.142)  0.579 ms  0.531 ms
 5  PNT-0039.GW1.HKG4.asianetcom.net (203.192.178.137)  0.710 ms  0.694 ms
 6  gi13-1.gw2.hkg3.asianetcom.net (122.152.184.42)  0.559 ms  0.608 ms
 7  po2-1-1.cr3.hkg3.asianetcom.net (202.147.16.73)  1.898 ms  1.917 ms
 8  po14-2-0.cr1.nrt1.asianetcom.net (202.147.0.169)  54.458 ms  54.375 ms
 9  po2-1-0.gw3.lax1.asianetcom.net (202.147.0.162)  180.820 ms  180.301 ms
10  gi1-31.mpd01.lax05.atlas.cogentco.com (154.54.11.133)  179.917 ms  257.576 ms
11  te4-2.ccr02.lax01.atlas.cogentco.com (154.54.6.189)  188.953 ms  189.052 ms
12  te4-1.ccr02.sjc01.atlas.cogentco.com (154.54.5.69)  191.762 ms  193.605 ms
13  te3-3.ccr02.sfo01.atlas.cogentco.com (154.54.2.125)  185.090 ms  185.235 ms
14  te4-3.mpd02.den01.atlas.cogentco.com (154.54.5.197)  217.569 ms  223.162 ms
15  te4-1.mpd01.den01.atlas.cogentco.com (154.54.0.109)  210.145 ms  209.567 ms
16  360-networks.demarc.cogentco.com (38.104.26.74)  213.263 ms  211.191 ms
17  66.62.4.65 (66.62.4.65)  223.749 ms  223.591 ms
18  66.62.3.21 (66.62.3.21)  223.494 ms  227.203 ms
19  66.62.5.3 (66.62.5.3)  226.369 ms  222.972 ms
20  66.62.56.26 (66.62.56.26)  239.029 ms  237.787 ms
21  208.71.136.2 (208.71.136.2)  233.600 ms  235.684 ms
22  mail4.qwknetllc.com (208.71.139.130)  223.938 ms  223.219 ms
```

As the output shows, not only does the latency of a single packet climb from 94 milliseconds to 220 milliseconds, but there are also five additional hops needed to reach the destination. Each of those links and each of those routers introduces additional delay into the network round-trip time.

Congestion and packet loss

Much of the Internet can't move as many packets as its users would like. Peer-to-peer traffic and spam further clog links. The routers have to do something with all of this excess traffic, and more often than not, they simply discard it, relying on TCP to recover from the loss.

Loss of data generally means that the packet must be resent, a process known as *retransmission*. We say "generally" because many UDP-based applications such as voice and video can handle a certain amount of loss. In certain situations (like real-time updates), delivery is more important than getting 100 percent of the data. For a voice call, it is better to have a brief moment of static than to introduce more and more delay as the call progresses.

As we've seen, the TCP stack handles retransmission of data. There's a bit more to it, though. Consider a highway on which it takes one minute to drive from point A to point B. If there's only one car driving back and forth on the highway, you can send at most one carload every two minutes across the highway. Double the number of cars, and you double the number of carloads per minute. But there's a limit to this: at some point, the highway becomes congested, and the introduction of additional cars slows down all cars on the highway. Maybe there's an accident or a stalled car; whatever the case, you need to reduce the number of cars in transit. In other words, there's an optimal number of cars in transit that maximizes the carloads per minute.

TCP tries to send as much data as possible without being greedy. It, too, is seeking the optimal number of things in transit. To do this, it first sends one packet and waits for a response. Then it sends a couple, and waits for the response. It continues to do this —putting more and more packets "in flight" between the sender and receiver—until something goes wrong. In practice, different computers and devices have different initial numbers of packets in flight and different ramp-up rates to squeeze more performance out of their Internet connections, but the basic functionality is the same.

Packet loss has three effects:

- TCP resends the packet, which consumes bandwidth.
- The sending TCP stack reduces the number of packets in flight, reducing the overall bandwidth of the connection.
- The receiving TCP stack, which is trying to deliver packets in the right order, has to hold on to all other packets until the right one arrives, which means it can't deliver the rest of the content until the arrival of the required packet.

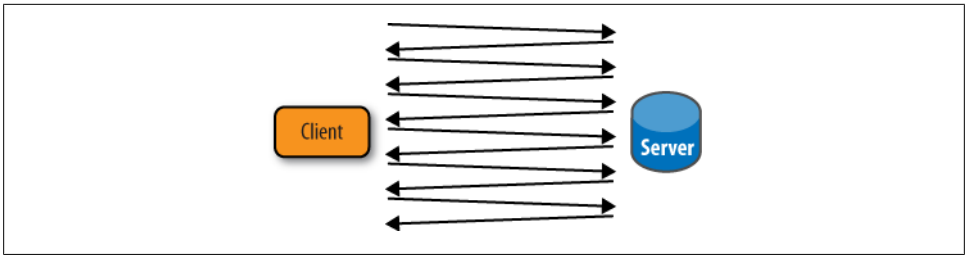


Figure 8-21. A high number of turns with relatively low LAN latency delivers performance that's fast enough

In other words, packet loss is bad. The bigger your pages, the more congested your networks, the greater the distance between sender and receiver, and the greater the number of objects, the higher the chances that loss will happen.

While you need to do everything you can to minimize the impact of round-trip time—most notably, getting your content close to your end users and making sure it's as small as possible—most of the delay in web applications comes from turns. It's the back and forth repetition that comes from retrieving many objects that really slows down websites.

Turns: Back and forth is the real problem

In early client/server computing, the client and the server lived on the same LAN, as shown in Figure 8-21. Performance was decent; the client queried the database, and the database responded.

Things were also predictable. If you knew the round-trip time, the number of turns, and the lossiness of your network, you had a fairly good idea of how significant the network delay would be.

When we moved these applications to WANs, however, they died. Most of them had been tested on fast connections to nearby servers, and the speed of those connections hid the constant chatter. On a slower WAN link, those hundreds of back and forth conversations slowed things down interminably, making otherwise decent applications ponderous and unwieldy, as Figure 8-22 shows.

The early web-centric design model replaced the client-server model. The browser was a dumb edge to a smart core, where the Web, application, and database talked amongst themselves and presented a ready-made page to the browser. This reduced the number of turns across the slow WAN, as shown in Figure 8-23, and gave visitors acceptable performance.

With the creation of larger pages, richer content, stylesheets, scripts, and third-party mashups, Internet applications are once again sliding toward a world filled with back and forth chatter (Figure 8-24).

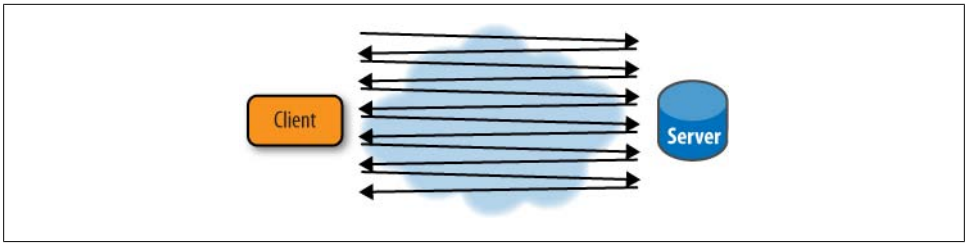


Figure 8-22. A high number of turns, coupled with high WAN latency, make the application unacceptably slow

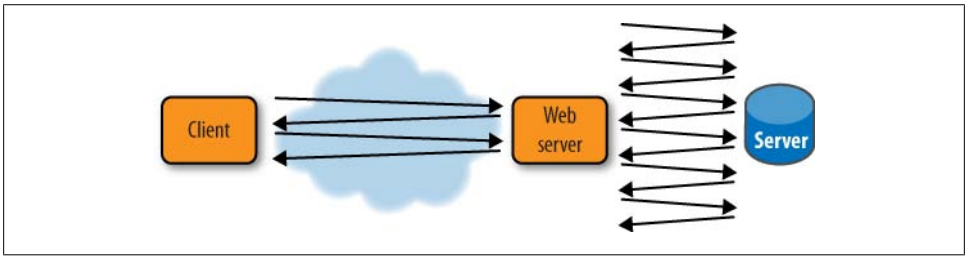


Figure 8-23. A smaller number of turns on a relatively slow WAN offered acceptable performance for early web applications

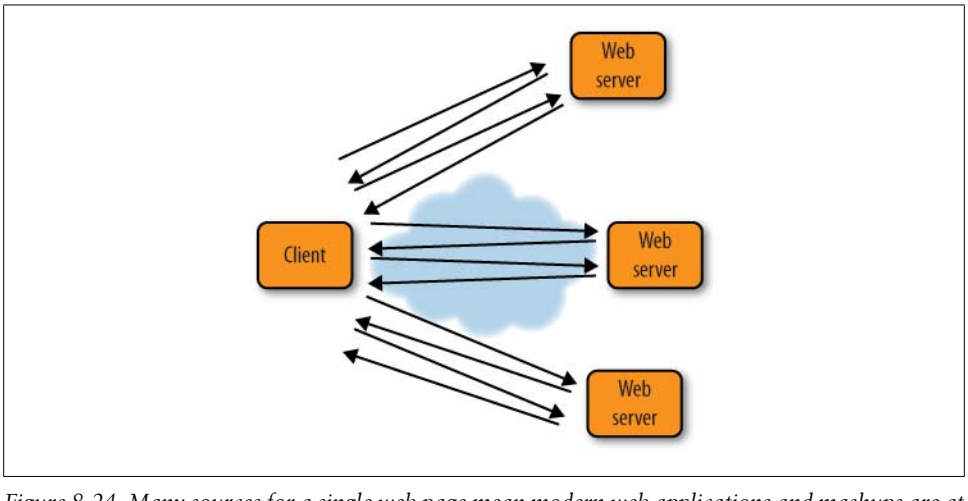


Figure 8-24. Many sources for a single web page mean modern web applications and mashups are at risk of poor performance

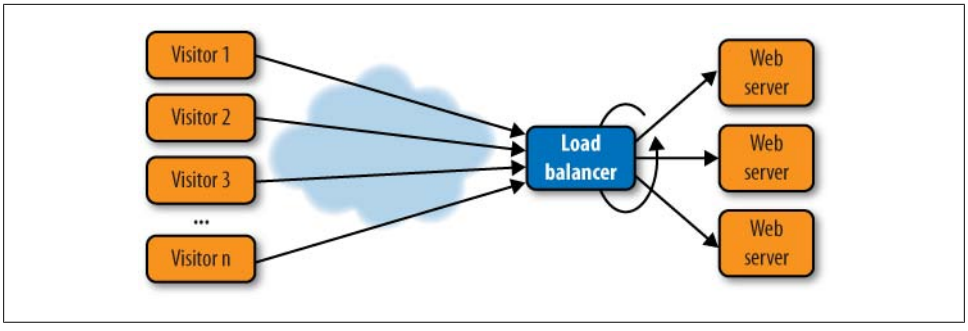


Figure 8-25. Load balancers distribute load across all available web servers in the data center on a per-visitor basis

So there are several factors behind network latency:

- The size of the content, which dictates how "big" the page is
- The number of objects and, by association, the number of turns you'll need to make across the Internet connection
- The available bandwidth
- The amount of packet loss and retransmisson that interferes with delivery
- The round-trip time between the browser and the web server

Fiddling with Things: The Load Balancer

Most websites of any significant size have a load balancer between them and the outside world. The load balancer shields the website from the Internet and attempts to make the site faster and more reliable.

Distributing load

One of the original reasons for using load balancers was to balance the load across many servers equitably. Rather than running one very big web server, companies could buy several smaller ones and "spray" traffic across them.

The simplest way to share load is to iterate incoming visits across web servers in a round-robin model, as shown in Figure 8-25. However the round-robin model doesn't take into account other factors, such as the work that each server is doing or the fact that some servers may be more powerful than others. Today, load balancers use more sophisticated algorithms, choosing the server with the fewest users on it or the one that responds most quickly.

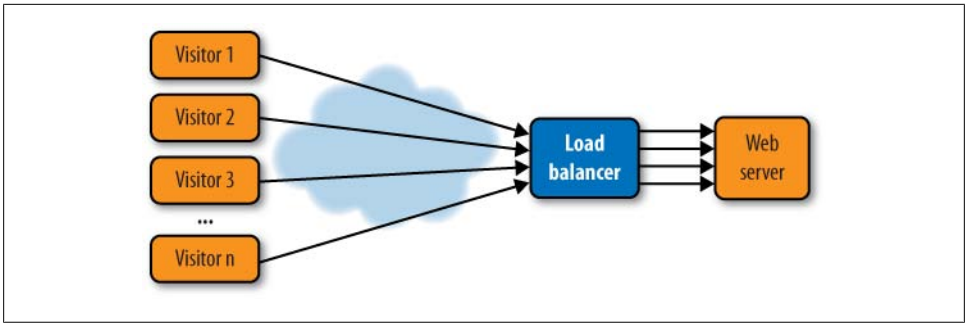


Figure 8-26. By consolidating incoming requests and sending them across a few, more efficient, connections to servers, load balancers offload network processing

Avoiding broken machines

The other basic reason for balancing load is reliability. By having several machines, your website has redundancy. If one server fails, the load balancer can remove it from rotation and insulate your visitors from the outage.

Address translation

The load balancer has a public IP address, while the servers behind it can have private ones. This makes a site harder to hack, since the attacker can't connect directly to a private IP address across the public Internet. It also means you don't need as many IP addresses overall.

Consolidating requests

A large part of the processing that web servers do involves network signaling—managing the many TCP sessions they have at any given moment. Both the client and the server have to keep track of packet sequencing, retransmission, and so on. For a client, this is easy—it's only handling a few connections. For a server, however, it can be deadly. Servers may be tracking thousands of TCP sessions, and this can consume more resources than the actual website itself.

Load balancers can help. They can offload these thousands of small network connections and instead maintain only a few connections to each server.

As Figure 8-26 shows, the web server has only a few connections to deal with. The load balancer can use a technique called *pipelining* to send several requests to the server at once, further increasing the efficiency of this connection. Note, however, that pipelining makes it much harder to monitor network connections, since the usual model of request and response isn't as easy to follow when several requests are sent simultaneously.

Compressing and encrypting responses

Encrypting traffic takes a lot of computation—something that's better left to the load balancer, which has special hardware to encrypt as efficiently as possible. This can include encrypting and decrypting traffic, as well as compressing data and even adjusting caching parameters to reduce the number of requests.

In the case of encryption, the load balancer needs to decrypt the traffic so it can look at the HTTP headers (which would otherwise be hidden) and decide which server should handle them, based on cookie and content. You don't need to understand the inner workings of a load balancer to monitor this, but it is important to know where the data is and isn't encrypted. If you sniff the wire between the load balancer and the web servers, you may see traffic in plain text that will be encrypted and compressed before it's sent across the Internet, so internal measurements of page size may not match those of the pages that reach the users.

Server Issues

There's only so much a load balancer can do to help with performance. While networking, protocol inefficiencies, and general-purpose tasks like redirection and encryption can often be tackled by hardware upgrades, bandwidth purchases, and the use of a CDN, application and server problems are more complex and may require rewriting or rearchitecting an application.

Server OS: Handling networking functions

The first part of a server to handle a browser's request is the operating system. It's a traffic cop for all TCP/IP connections, and may have other functions (such as logging or security) that consume processing power. Different operating systems will have different performance characteristics. There are many server monitoring tools that can provide insight into OS performance (Figure 8-27).

Web service: Presentation layer, file retrieval

Just as a desktop operating system runs applications (for example, a word processor), so a server operating system runs services (such as web, authentication, or database). The first part of any web application cluster is the actual HTTP service. Two of the dominant web servers on the Internet today are Apache and Microsoft's IIS web server, but hosted services like Google's web stack are behind a growing number of websites (shown in Figure 8-28).

If the TCP client on your desktop has its counterpart in the TCP stack of the server, the HTTP in your web browser (e.g., Firefox) has its counterpart in a web service (e.g., Apache). The web service handles the HTTP status codes and persistence needed to deliver objects properly. It gets those objects from memory, from disk, or from application processing.

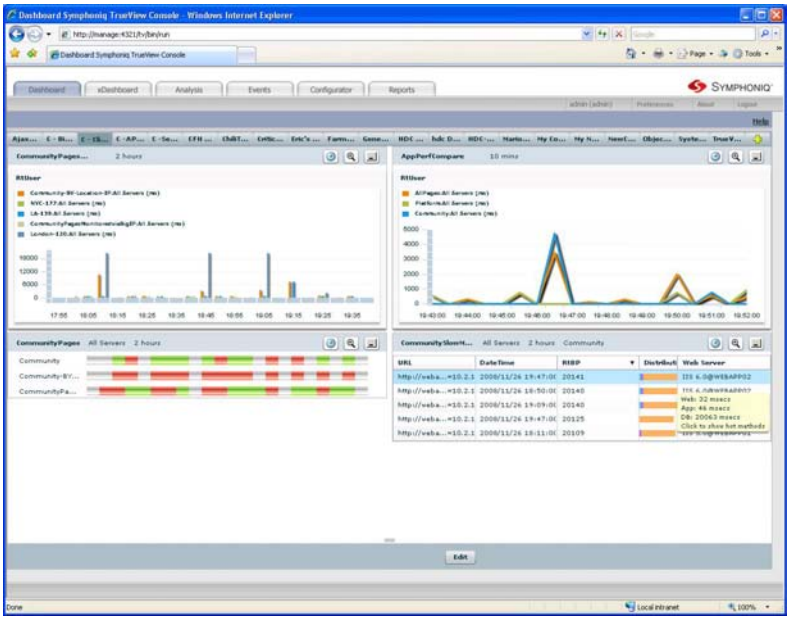


Figure 8-27. Symphoniq correlates OS, App Server, and database performance information with end user experience

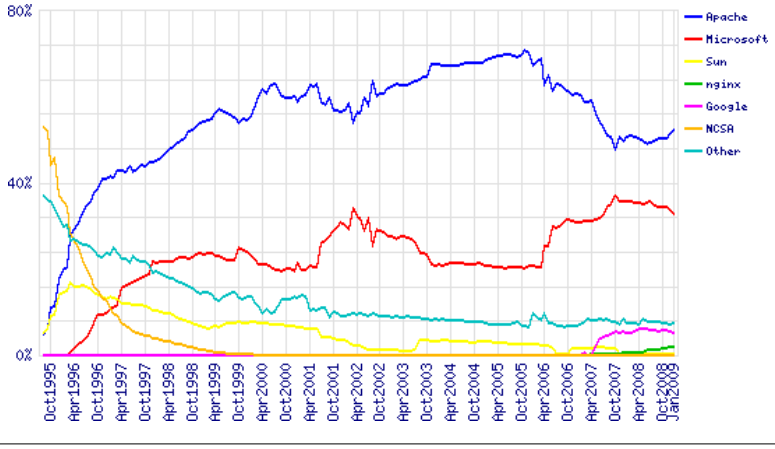


Figure 8-28. Netcraft's monthly "web server survey" results (http://news.netcraft.com/archives/web_server_survey.html)

When it comes to measuring just the web server's performance, the key metric is how long it takes to get back an HTTP response for a trivial object. If we request a tiny image from the web server, the server shouldn't have to think about the request much—it

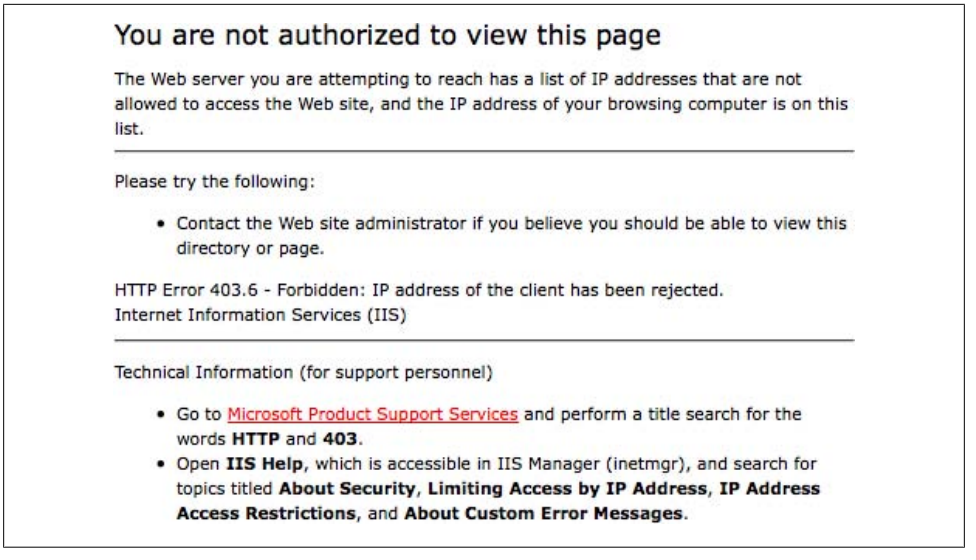


Figure 8-29. A 403 error

should just deliver the image. And because the image is tiny, we won't have several turns of the network to complicate things. The time it takes to retrieve a small object, less the network round-trip time, is roughly the delay that the web service has introduced.

We may also encounter problems with the client's request. While these are often malformed requests from browsers, they can also include blocked IP address ranges, as shown in Figure 8-29, or invalid visitor credentials. Problems like these are particularly hard to detect without looking at logfiles, leading to the all-too-common "it works for me" defense from IT teams.

Dynamic tier: Application logic

A web service that handles only static content isn't very common. Typically, the web service has an application service behind it, either on the same server or on a separate tier of servers. This service runs a program each time a request arrives and sends the output of that program back to the web tier for delivery.

There are a number of popular application servers online today—many more than there are web servers. In the Microsoft world, IIS is both a web server and an application server, and it focuses on generating Active Server Pages (the *.aspx* extension of many sites). Similarly, Java is a common server language that generates pages with a *.jsp* extension. A website can be written in any language, but some, like PHP, Ruby, and Python, are increasingly common.

You may be more familiar with turnkey applications such as WordPress, Drupal, Mediawiki, or PHPBB. All of these are application servers in their own right. Many of these

tools come with built-in logging and problem recording that's delivered within the web server logs, and they provide more detailed information about errors than web servers do on their own. Figure 8-30 shows a series of errors reported by the WordPress platform.

```
[root][/var/log/apache2]: grep "SELECT post_id" httpd.org-error.log
[Thu Jul 31 01:08:47 2008] [error] [client 24.201.75.95] WordPress database error Server shutdown in progress
 for query SELECT post_id, meta_key, meta_value FROM wp_postmeta WHERE post_id IN (8) ORDER BY post_id, meta_
key made by update_postmeta_cache
[Thu Jul 31 01:08:47 2008] [error] [client 24.201.75.95] WordPress database error Server shutdown in progress
 for query SELECT post_id, meta_key, meta_value FROM wp_postmeta WHERE post_id IN (8) [Thu Jul 31 10:46:16 20
08] [error] [client 63.251.186.252] File does not exist: /var/www/httpd.org/feed
```

Figure 8-30. WordPress errors in Apache server logs

Depending on the application server you're using and the verbosity of logging that you configure, you'll get varying degrees of visibility into its health. Remember, however, that logging consumes processing and storage resources—if you enable logging to better understand a problem, you may introduce even more delays than you had in the first place. Some large sites simply can't turn on logging because of the impact on web performance and the volume of data it creates.

From a performance standpoint, application servers cause a lot of delay. You can measure the application's delay by comparing the "trivial" web request you use to test the web service tier with the responsiveness of an application request.

Storage tier: Data, state persistence

Behind the application servers lie databases that the application tier can query to retrieve data. Databases store information such as product catalogs and social graphs; they may also store application state.

Databases are often the cause of host latency. The database may be busy doing something else (such as rebuilding an index or clearing out data). It may have many requests for the same data, it may be dealing with inefficient queries, or it may just be digging through many records. There are so many ways databases can go wrong that database tuning is a dark art that fills entire books (see IBM's list of known causes of slow database performance at *http://www-01.ibm.com/support/docview.wss?uid=swg21174563*).

Some sites have huge amounts of data, as in the case of search engines that use other storage technologies in which shards of data are spread out across many machines, each independent of the others. This shared-nothing approach, which is used in Google's filesystem and other cloud computing models, scales more smoothly because it avoids contention that can happen with traditional databases.

Sometimes a database will return an error to the application server. The application server should recognize the error and deal with it gracefully; if it does so, you need to collect the error as part of your monitoring even if the error wasn't shown to the user.

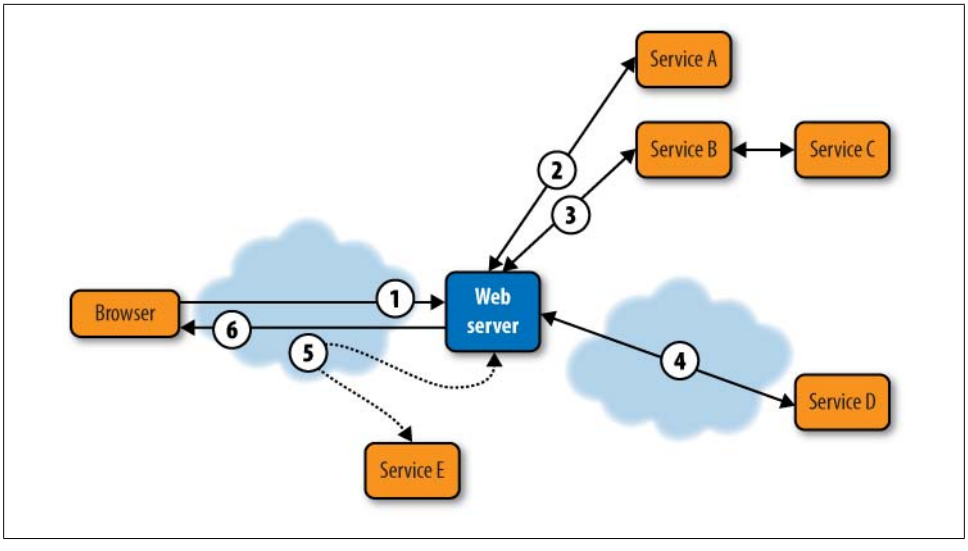


Figure 8-31. Communication model in an SOA-driven architecture

To determine the impact database latency has on overall site performance, compare the performance of pages that hit the database to those that are returned by the application server tier without consulting the database. Some web operators create custom pages that specifically test database functions such as writing data or retrieving information, and use this as part of their testing strategy.

Third-party components: Web services, transaction processing

With an increasingly distributed web, your application server may also talk to backend systems you don't control. For example, during credit card validation, your server may have to check with a merchant bank as well as fraud detection systems to validate the transaction before confirming a purchase.

Designing applications using a set of predefined services—known as a service-oriented architecture, or SOA—focuses on assembling building blocks that are interoperable through well-defined interfaces. This makes maintaining and upgrading the application easier. For example, a developer could replace one currency conversion service with another and not have to change the rest of the application.

However, SOA creates a great deal of machine-to-machine communication that's hard to troubleshoot. Figure 8-31 shows an SOA-driven backend for a website.

The browser sends a request (1) to the web server, which places backend calls (2) to other web services. Some of those web services (3) may themselves rely on other services; some may be located across dedicated wide-area connections (4) that increase overall latency; still others may be connected across the Internet (5) with all of the unpredictability that entails. Eventually, the server can send the response to the browser (6), but what appears to be host latency is in fact the result of third-party service delays.

Remote services can respond in unexpected ways, and it's up to your application servers to deal with these conditions gracefully. If the application relies on third-party services, you also need to consider the impact of the network between your data center and the third party.

Permissions and authentication

Even if everything is working smoothly, you may often encounter problems with authentication. Most content in an application has permissions associated with it. A surprising number of web problems stem from having the wrong permissions—a server that's not allowed to retrieve a certain object, for example, or a blog application that can't write to a particular directory.

The relationship between workload and performance

Sites that get suddenly busy often get suddenly slow. There is a correlation between traffic volume and web performance, but it's not a linear one, so it can be difficult to detect until it's too late. A server that can handle 1,000 hits per second may be fine at 900 hits per second, but at 1,001 hits a second, traffic is arriving faster than the server can deal with it, and a backlog quickly forms. If all else is equal, the greater the traffic to a site, the slower the responsiveness and the higher the number of incidents.

One of the most challenging aspects of web performance management is scaling. Because websites are connected to the entire world, their traffic levels can vary significantly from day to day. This is especially true for seasonal businesses, for example, a tax filing website at tax time or a costume store at Halloween.

The end result: Wide ranges of server responsiveness

All of these factors mean that responsiveness will vary widely depending on what users are doing and how many users there are on your site. *Many operational teams have no idea how many users are on a website or what they're doing*. We've found that fewer than half of the web operators we asked knew even basic statistics such as how many hits per second their site received, even though this is a strong predictor of website performance.

Instead, they looked only at the health of the underlying platforms. It's a myopia that can prove catastrophic.

Consider a blog, for example. Retrieving a blog page is relatively speedy: the content is ready, and it's a matter of a simple request to the database. Your blogging software may even have cached a copy of the requested page.

Compare that to the act of saving a blog. The blog software has to store the new post to the database, then it has to tell other web servers that the new blog is available, and update RSS feed information. It has to modify the list of tags and build a new cached copy of the website. That's a lot of work, and it can take several seconds.

If you're operating a blog and you see a sudden drop in performance, you need to determine whether it's because a few authors are submitting stories or because many visitors are suddenly looking for content. Without the context of what users are doing, you can't diagnose the slowdown.

Client Issues

As if network latency and server delays weren't enough, there's still the client to consider. As web applications become increasingly dependent on edge processing and rich browser frameworks like Flex, Silverlight, Google Gears, and Ajax, the client plays a more important role. More things can go wrong on the browser, where problems are hardest to see.

Desktop workload

Client computers have a finite amount of processing power. This is particularly true for subnotebooks designed to sip lightly from batteries—their power efficiency comes at the expense of CPU.

The desktop may cause delays because of slow network processing or because the user is doing other things that consume cycles the browser needs. The richer your pages, the more the desktop will slow down. This happens in two distinct ways:

- *Processing delays* prevent the browser from retrieving information from memory, launching plug-ins, or performing calculations.
- *Presentation delays* slow down visualization and screen display.

The browser

The modern browser is an operating system in its own right (*www.google.com/googlebooks/chrome/small_00.html*). It's handling multiple tasks (sites, each in their own tab) and managing local resources (Flash, Java, and Greasemonkey plug-ins and add-ons) even as it executes code.

Just like operating systems, browsers are extensible. They offer new ways of visualizing and navigating the Web, each of which poses its own performance and availability risks. New ways of accessing the Web, such as the Cooliris application shown in Figure 8-32, further push the boundaries of desktop processing capacity.

The more we ask the browser to do, the slower it becomes. More and more of a page's latency happens when it reaches the client. Fortunately, desktop performance has increased along with end user demands (thanks to the continued doubling of processor capacity that is Moore's Law)—a typical page from a modern website would have brought a PC from only a few years ago to a grinding halt.

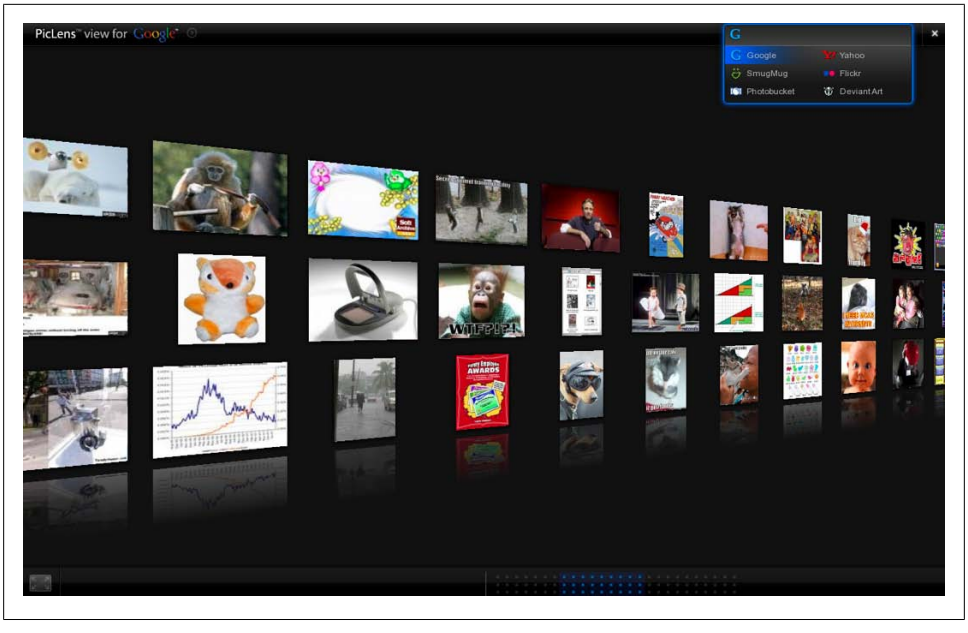


Figure 8-32. While not really a web browser, the Flash-based Cooliris application displays hundreds of images and uses the HTTP protocol

Browser compatibility

One of the first ways a browser can break things isn't actually the browser's fault at all. There are hundreds of plug-ins for the Web, each available in various versions. Adobe Acrobat, Sun Java, Macromedia Flash, and Microsoft Silverlight are just a few examples. Many sites try to run other plug-ins, such as ActiveX objects. If the browser can't correctly launch one of these plug-ins, either because the user won't allow it, because the browser or operating system doesn't support it, or because the plug-in won't load properly, then (as Figure 8-33 demonstrates), your site is broken for that segment of the market.

Even without plug-ins, many websites don't support certain browsers, as shown in Figure 8-34.

Client compatibility can only be addressed with good testing and an understanding of your audience's technical environment. You can learn a lot from web analytics by observing browser types, and you should always consider the impact of plug-ins—many of which won't be included in web monitoring tests—on performance.

Stylesheets and page layout

When HTML was first defined, it included both content and formatting. If you wanted to make a heading red and 22 pixels high, you said:

```
<H1 color=red size=22px>I upgraded your RAM</H1>
```

This was a problem. For a designer, changing the color of headings meant modifying the HTML on every page of the site without touching the content. It was also redundant—every heading needed the `color=red` and `size=22px` attached to it, which increased file size.

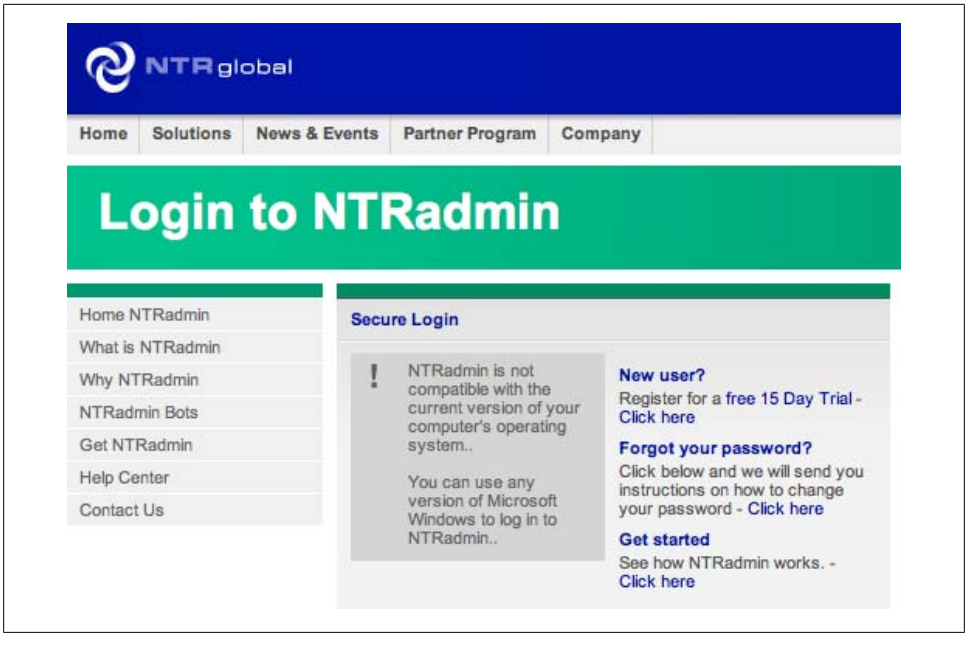


Figure 8-33. A web-based application that requires a specific desktop operating system to run

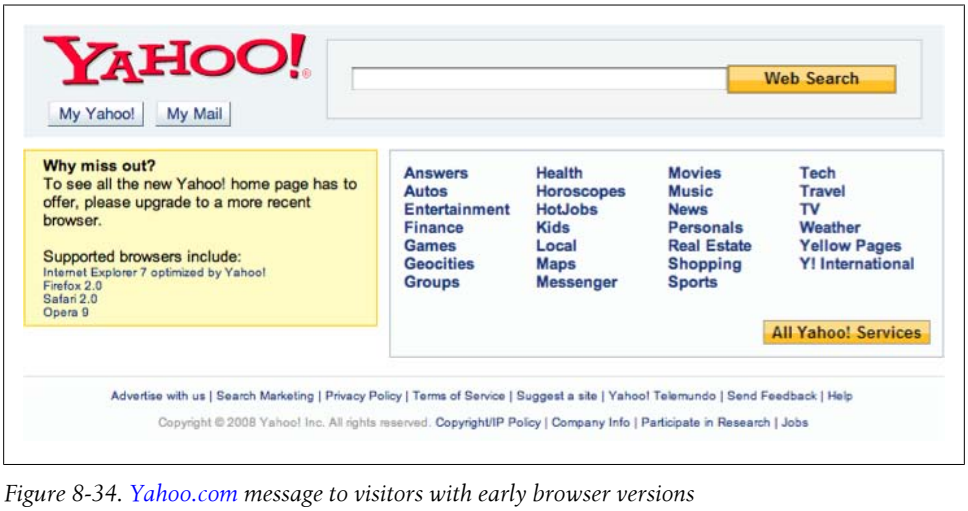


Figure 8-34. Yahoo.com message to visitors with early browser versions

To address this, we can separate formatting (`color=red`) from content and meaning (a heading called, `<H1>`) As a result, to define headings as red, the site will now have two files: one concerned with meaning and one concerned with formatting.

The HTML file includes the content and its semantic meaning:

```
<H1>I upgraded your RAM</H1>
```

A separate document, the Cascading Style Sheet (CSS), tells the browser how to format headings in general:

```
H1{font-size:22px;color:#333;}
```

This way, to change the color, the designer only has to change the formatting once, in one place—the stylesheet. Browsers then use the new formatting every time they render a page linked to the stylesheet.

This not only simplifies the process of changing styles and formatting, it also means that developers don't need to understand layout. It reduces the chances of errors in the editing process. And for large pages, a single note in a stylesheet is more efficient than formatting throughout the page.

Stylesheets aren't all good, however

- They're yet another object for browsers to retrieve, resulting in more turns.
- Stylesheets can contain references to other objects, such as background images, that complicate referrers and make it harder for monitoring tools to determine which components belong to which pages.
- There's additional browser compatibility to check, as not all browsers render stylesheets the same way.

Processing the page

Once a browser has loaded the page, it needs to assemble the contents and display them to the user. Depending on the amount of work involved, this can include executing JavaScript, launching plug-ins, and rendering images. The only way to capture this client-side delay is to be there when it happens, using JavaScript or a desktop agent to time the loading and rendering of the page.

Prefetching

Humans take time to read pages. During that time, the browser and the network are relatively idle. Why not put that idleness to good use?

One approach to improving performance is prefetching: the browser tries to guess what the user will want next, and gets to work loading it. On a page with only five hyperlinks, the browser can preload the five container objects that might come next and have them ready for the user.

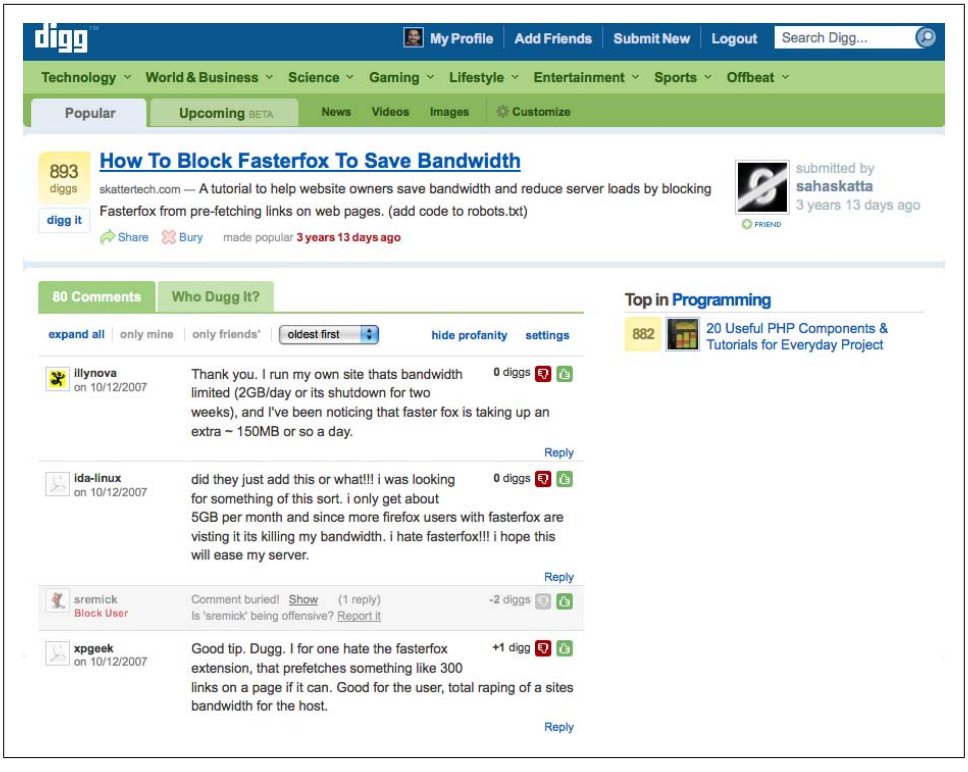


Figure 8-35. Digg criticism of the aggressive behavior of the Fasterfox plug-in's prefetching features

While this seems like a great idea for the user, it's a *tragedy of the commons*—a situation in which individuals acting in self-interest can ultimately destroy a shared resource, even though it's not in their communal interest to do so (*http://en.wikipedia.org/wiki/Tragedy_of_the_commons*). In our case, the prefetcher receives all the benefit of prefetching, but the damage to performance is shared by everyone. One user's experience is better, but the server is flooded with five times more requests than users will actually see. The site slows down for everyone, including the greedy prefetcher.

Prefetching is considered an aggressive practice when it's initiated by the client, and plug-ins like Fasterfox have been widely criticized for supporting it, as shown in Figure 8-35.

There have been other examples of prefetching built into browsers and plug-ins. Google's Web Accelerator prefetches content, but it does so from Google's regional caches rather than the root server in most cases. Because Google's servers handle the brunt of the load, it receives fewer complaints than client-based prefetching.

Prefetching is useful in the right conditions. It's common to preload the next slide in a slide deck or the next picture in a photo gallery website. There's a very high likelihood that the fetched content will be the next thing the user requests, and it's the site's

designer, not the browser, that initiates the prefetching. Good candidates for prefetching are situations in which end users are very likely to go to a single next item, and where that item will take time to load. This is a common Web 2.0 design pattern—when it's done by the site operator, not a browser plug-in.

From a performance and availability standpoint, prefetching makes page monitoring more difficult:

- As with CSS formatting, it's difficult to determine which objects belong to which containers because many "false" container downloads aren't followed by their component objects.
- It can skew page counts and per-visitor traffic estimates if you're collecting analytics through logfiles or passive capture rather than through JavaScript.
- Without detailed information about how prefetching is configured on the browser, it's hard to form an accurate estimate of page render time.
- Extra load on the site increases both network and host latency, and may exacerbate problems that only happen under load.

Other Factors

As the landscape of the Web changes, there are new factors that can undermine performance and availability or make user experience harder to monitor. Here are just a few.

Browser Add-ons Are the New Clients

Browsers can be extended in two main ways using what Mozilla refers to as add-ons. The first category of add-on—known as extensions—affects how a page is loaded and displayed. One extension, Greasemonkey (*https://addons.mozilla.org/en-US/firefox/search?q=greasemonkey&cat=all*), is a scripting platform that can modify pages as they're loaded, letting visitors customize how a site behaves within the browser.

Extensions may separate you from the user experience. Visitors may be playing a game such as WebWars or PMOG instead of interacting with your content. Browsers may be running the NoScript extension to avoid running scripts from untrusted sites. Extensions may also interfere with your JavaScript instrumentation of a page or break it entirely. However, extensions are still running within the browser, unlike the other form of add-on, known as a plug-in.

The Web has evolved in ways its creators couldn't foresee, and today it's the platform for broadcast media, gaming, human interaction, and more. Plug-ins are part of what makes this possible. They're standalone software applications like Java, Quicktime, Windows Media Player, Adobe Acrobat, and Flash. Your browser delegates work, such as playing an audio file, showing a video, or displaying a document to them.

Figure 8-36. Beatport.com is a B2C e-commerce application fully driven by Adobe Flash technology

Plug-ins can cause delay. For example, reading an Acrobat PDF means loading the Acrobat reader, and running a Java program means launching the browser's Java Virtual Machine. It takes time for the browser to initiate these components, and this may introduce additional delay that's hard to capture. Plug-ins take time to run their code or display their content, which further slows down the user experience in ways that are hard to measure.

For the web analyst, plug-ins represent a far bigger concern than just additional delay. Unlike extensions, which run within the browser and have limited access to the user's computer, plug-ins can do whatever they want. They may even communicate independently of the browser (for example, this is how plug-ins like Quicktime know that an upgrade is available).

Entire sites can be built in Flash using only a single container page, such as Beatport (shown in Figure 8-36). Flash is also the basis for much of the video on the Internet and many online games, some so processor-intensive that they only run on high-end machines.

Despite the broad adoption of Flash and JavaScript, there are relatively few standards for monitoring and measuring them. As a result, application developers who want to measure performance and availability must often handcode monitoring scripts and embed instructions in web pages to collect timings and send them back to the servers.

Timing User Experience with Browser Events

As all of these complications should make clear, with the advent of Web 2.0, we've been rethinking what makes pages slow. The focus has shifted from host and network time to client-side interactions and third-party mashups, from efficient page design to better scripting.

What's in a DOM?

The Document Object Model, or DOM, is one of the least understood and poorly explained concepts in the Web today. We'll try to fix that.

We can describe a house as a hierarchy of elements. We might have several categories of information about the house—rooms, occupants, and furniture, for example. Rooms would include house.rooms.bedroom, house.rooms.bathroom, and house.rooms.kitchen. Similarly, occupants might include house.occupants.mom and house.occupants.dad.

We could further extend these hierarchies: house.occupants.dad.pants.color might refer to the color of Dad's pants. Then, if we wanted to change the color of Dad's pants (in our megalomaniacal house DOM) we could set house.occupants.dad.pants.color=paisley.

Browsers have a similar hierarchy of elements that describe everything they're up to. The DOM starts with the browser itself, then the windows and tabs. When you load a page, the browser builds a hierarchy of all the things on that page, such as forms, titles, and headings.

Each of these elements has a value. For example, a form field may be referred to as window.document.form.element.text (or the specific names of those elements.) JavaScript running in that page can do several things with this field by referencing it with its DOM name:

- It can determine the value of an element by referencing it with its name. For example, it can determine what a user has typed into a field.
- It can change the element's value. For example, it can trim the spaces from a text entry in a field.
- It can tell the browser to let the script know when something happens to an element; e.g., it can do something special when the user presses Enter in a field.

These are relatively trivial examples, analogous to changing Dad's pants color in the house example. The real power of the DOM is that it lets JavaScript modify the entire web page—akin to rearchitecting the whole house, adding rooms, and moving furniture around.

On a browser, the DOM is also where details are stored, such as cookies associated with a site. By manipulating cookies with JavaScript, sites can store information on the browser until a visitor returns. The DOM is also where web analytics scripts get much

of their data about things like browser version and window size. For more information about DOM, see *www.w3.org/DOM/*.

JavaScript is an event-driven language. It relies on specific events, such as the click of a mouse, the scroll of a window, or the loading of an image to trigger actions. Dozens of events can occur for any element of a page. One of the most fundamental events for a page element is its arrival. When a browser loads a component object, it generates an event to tell JavaScript that the component is now loaded (for more information about DOM events, see *http://en.wikipedia.org/wiki/DOM_events*).

Traditionally, browsers pronounced a page finished when all its component objects were retrieved. You can see this in the status bar of most pages, which changes to "Ready" (or simply goes blank) to indicate that the page has been fully retrieved. This milestone, known as the OnLoad event, was used by many monitoring approaches to measure page load time from the client's perspective.

However, this event isn't as reliable an indicator of page load as it once was. Modern web designers thwart the OnLoad event. Some optimize pages carefully, placing JavaScript high on the page so it can run and so users can interact with the website before the page has finished loading. In these cases, the OnLoad time overstates delay, since performance measured from the OnLoad event is worse than it actually was for the user.

Other sites only start executing JavaScript once the OnLoad event occurs. In these cases, the OnLoad time is optimistic—the user's experience may be worse than what the OnLoad-based measurement says it is if the user needs the script in order to use the page.

There are several initiatives underway to correct this and come up with an accurate framework for client-side instrumentation. Most notably, Episodes, proposed by Steve Souders of Google, tries to standardize event timing and separate the task of instrumentation from the task of capture and reporting (*http://stevesouders.com/episodes/*). We'll return to the concepts behind Episodes when we look at RUM.

Nonstandard Web Traffic

The Web is a conversational, back-and-forth protocol. Web visits are initiated by your browser; servers don't connect to you indiscriminately. However, there are times when a server wants to push data out to a client—a stock ticker, for example, lets clients sign up for a particular stock, and pushes stock price changes out to those clients.

There are several ways to support this server push:

- The Real Time Media Protocol, supported by Adobe's Flash client, transmits a stream of data from server to client.

- A *perpetual GET* approach starts with a client that requests an object by HTTP and a server that returns a never-ending response. The client parses the response as it arrives.
- Web Sockets, a capability in the HTML 5 specification, creates a raw two-way connection between clients and servers. COMET is another similar approach to server push.
- In *long polling*, a browser requests an object, and the server intentionally takes a long time to respond. During this time, if the server has data to send, it can respond immediately; otherwise, at the end of the long response window, it can send back a null response and the browser can issue another long-poll request.

These asynchronous models, in which the client and the server can each act independently of one another, reflect a change in how we use the Web. With the growth of real-time chat and interactive media, we need to find new ways to measure such connections with metrics like message volume, number of streams, jitter, and bandwidth consumption.

RSS feeds and podcasts

Subscribers to web pages receive periodic updates about new content through either the RSS protocol or related web feeds like Atom. All of these feeds, collectively known as RSS (although this isn't entirely accurate), have one purpose: to update subscribers interested in a topic when new information is available.

RSS feeds are similar to traditional web traffic, but they have some important differences:

- They seldom carry a lot of content. The purest RSS feeds are simply structured text with a link to the content and a brief description. While some RSS readers load embedded images, this isn't a popular feature.
- They may offer only an excerpt of the content they're describing in order to encourage readers to visit the website in question to see more.
- They may include advertising; Google's FeedBurner service, shown in Figure 8-37, allows users to embed AdWords ads in content.
- It's hard to track abandonment. A user may receive hundreds of updates via RSS, but only read some of them. While this can be the first step in a conversion if the user loads the content and visits your site, disengagement is common: users simply don't read all the RSS feeds to which they subscribe.
- RSS feeds may be aggregated. A single web portal may request one RSS feed update on behalf of hundreds of subscribers. Some of these portals will tell you the number of subscribers in the user agent itself.

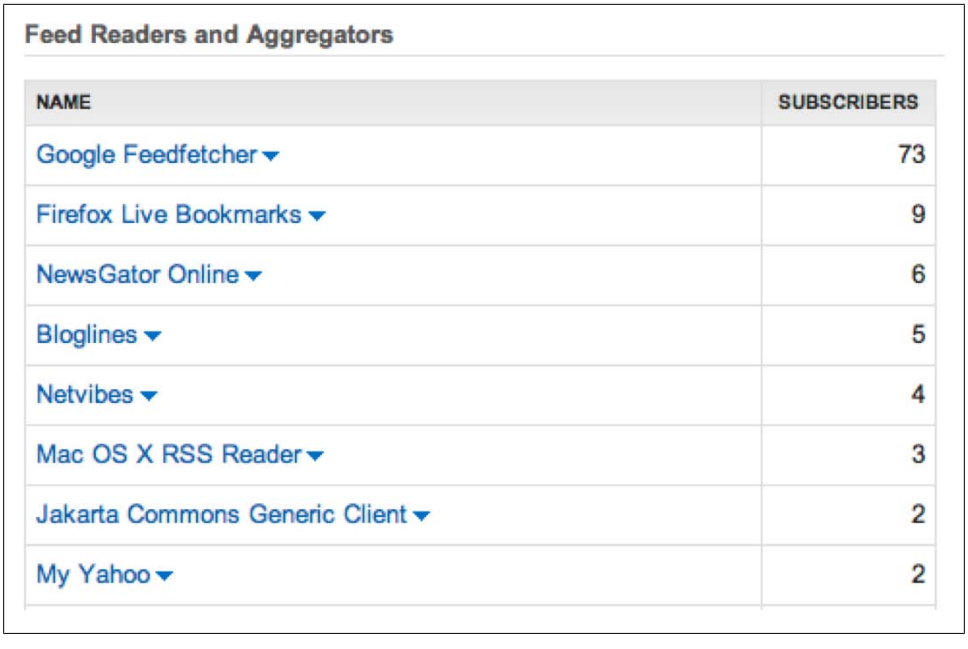


Figure 8-37. FeedBurner, showing feed aggregators consuming an RSS feed on behalf of their subscribers

For RSS feeds, we care more about availability and freshness. Since RSS feeds are updated in the background, the user's experience won't suffer if the feeds take a bit longer to get there. When we post new content, however, we want it to be available quickly so readers know about it.

Peer-to-peer clients

Video content consumes a tremendous amount of bandwidth. Today's approach of embedding video within a web page using a plug-in may work for viral videos, but it won't scale to Nielsen-level audiences and real-time distribution. We know of one large enterprise that saw its web traffic increase sixfold during the U.S. Open because of employees watching the game, which ultimately forced the company to block access to that site.

One way to address such scaling problems is to enlist the help of the clients. By turning a browser into a broadcaster that can share video content with those nearby, we can reduce the load on the servers and decrease the number of video streams running across the core of the network.

Peer-to-peer technologies, and the P4P working groups, are focusing on such approaches to scaling. We mention them here because they're going to require new ways of monitoring and new metrics to track end user experience. These will include streaming information such as the number of subscribers and the number of peers

participating in the stream, as well as more traditional video information like frame rate, jitter, and delay.

Pando Networks offers a good explanation of the P2P scaling challenges the video industry faces, as well as links to the P4P working group at *http://www.pandonetworks.com/p4p*.

A Table of EUEM Problems

Here's a summary of just some of the problems we've covered and how they affect successful page delivery. As Table 8-1 shows, there are so many things that can break in a web application that it's essential to monitor your website thoroughly.

Table 8-1. Some of the ways websites can fail

Error	Symptom	Possible Source
Server not found	Can't resolve IP address	DNS broken
No TCP response	Server fails to respond to the client's initial TCP request, leaving the client to time out	Network
TCP connection reset	Server establishes a TCP connection (SYN ACK) with the client, then resets it	Server overloaded
SSL negotiation failed	HTTP works, but HTTPS doesn't	SSL not enabled
No HTTP response	Despite a TCP session, HTTP GET receives no response from server	HTTP port wouldn't respond (service hung)
Bad redirect	Server sends client elsewhere, but new destination doesn't exist	Destination not working
Never-ending redirect	Server redirects client to itself	Misconfigured server or load balancer
400 error	HTTP 400 in the browser	Bad request from client (i.e., content doesn't exist)
500 error	HTTP 5xx in the browser	Server error
No content	Request gets a 200 OK, but no content follows	Server error
Response truncated	Server begins to respond, then stops	Server disconnected during response
Incompatible MIME type	Server sends image data, but browser displays it as text only	MIME type configurations on server incorrect
Broken content	User receives incorrect content	Corrupt data or application error
Client code error	Browser reports an error when executing code on page or launching plug-in	Bad JavaScript or old browser version
Components missing	Some elements of the page fail to load	Third-party site

Measuring by Hand: Developer Tools

Your job is to understand end user experience once a system is in production. Long before you deploy the application, your developers will have been using tools to estimate its performance. These include sniffers that can decode network traffic and find granular protocol issues, desktop software that measures responsiveness and suggests corrections, and hosted services that test a page from several locations to analyze its load time across the Internet.

If we've done our job explaining the elements of web performance, you'll be able to understand the results of these tools and pinpoint areas for optimization.

Network Problems: Sniffing the Wire

There's an old saying in networking: sniffers don't lie. Seeing the actual TCP/IP conversations that flow across a network connection and reassembling them into the HTTP transactions between a server and a browser is a time-consuming effort. It's also the best way to find out what really happened.

Wireshark (formerly known as Ethereal), shown in Figure 8-38, is the leading free packet sniffer. You can narrow the traffic it captures to a single TCP/IP session and reassemble the HTTP objects from the data stream. You can also use timing information to measure performance extremely precisely.

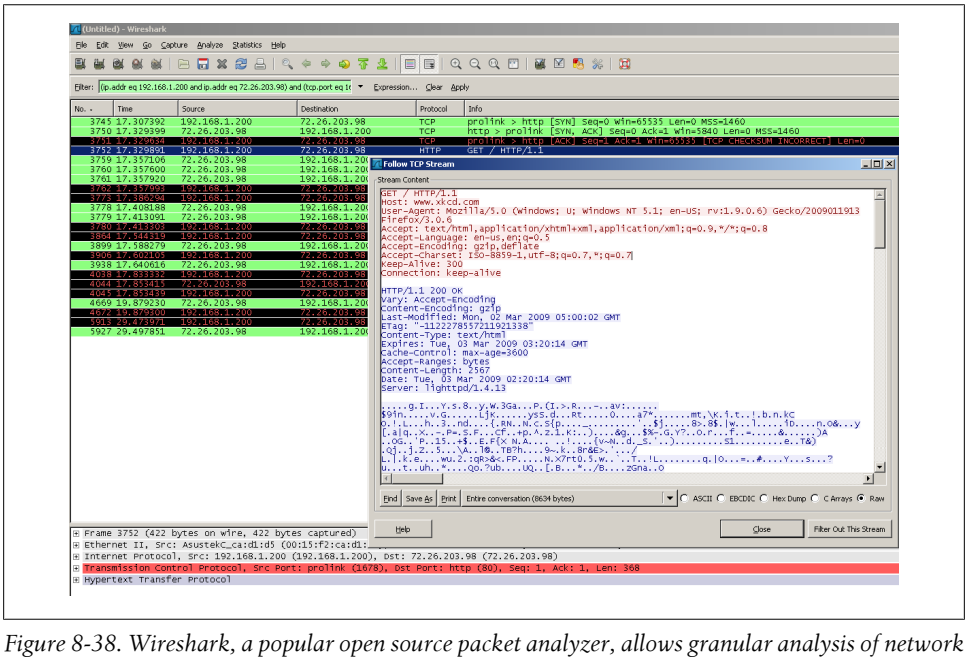


Figure 8-38. Wireshark, a popular open source packet analyzer, allows granular analysis of network traffic

Figure 8-39. An analysis of network latency in Firebug

If you want to understand how an application is working on the desktop itself, however, you need to use desktop tools. Packet sniffers can't show you latency that happens within the browser, and it can be difficult to reassemble modern web pages in an intelligible manner.

Application Problems: Looking at the Desktop

One of the most commonly used measurement tools is Firebug, a Firefox extension that allows developers to break down page load time, examine page components, and browse the DOM (*http://getfirebug.com/*). Figure 8-39 shows a cascade diagram of network latency for a simple page with 12 objects.

The YSlow extension to Firebug, shown in Figure 8-40, takes this analysis one step further, summarizing page performance and suggesting why slowness exists (*http://developer.yahoo.com/yslow/*). Steve Souders summarizes the best practices that are implemented in YSlow in his book, *High Performance Web Sites* (O'Reilly). Google Page Speed (*http://code.google.com/speed/page-speed/*) is another alternative.

Other desktop tools, such as HTTPWatch and Flowspeed for Internet Explorer, are also essential for web developers. All of them let your company fix problems before you go live and train your development team in how to think about performance. If you can report performance issues in ways they understand, using Firebug-style cascade diagrams or YSlow terminology, for example, they're more likely to follow your advice.

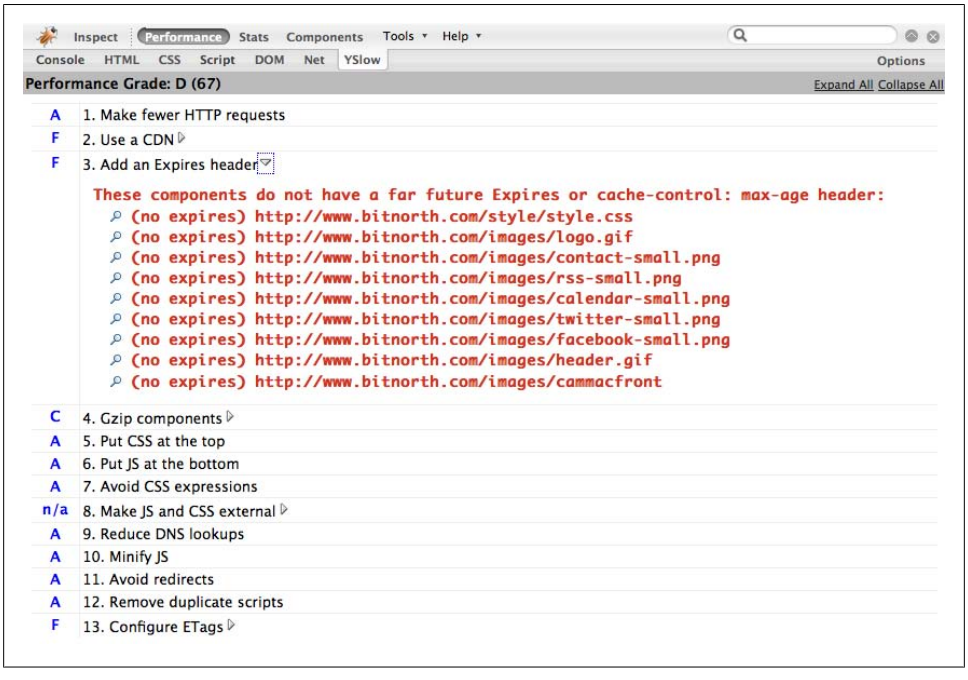


Figure 8-40. The YSlow extension suggests improvements to web page performance

Figure 8-41. Entering a page to test in www.webpagetest.org

Internet Problems: Testing from Elsewhere

Many sites will test web page performance from remote locations. Of these, AOL's PageTest is perhaps the most comprehensive, since it gives you control over many of the wrinkles we've outlined above—such as page caching—to produce more realistic results. WebPagetest.org (available at, strangely enough, *http://www.webpagetest.org*) is a live version of PageTest.

To use the system, first enter the URL you're measuring (Figure 8-41).

Choose the source of the test (Figure 8-42). Since the service is deployed as part of AOL, test servers are based in Virginia, but there is a new system in New Zealand to measure the impact of international traffic.

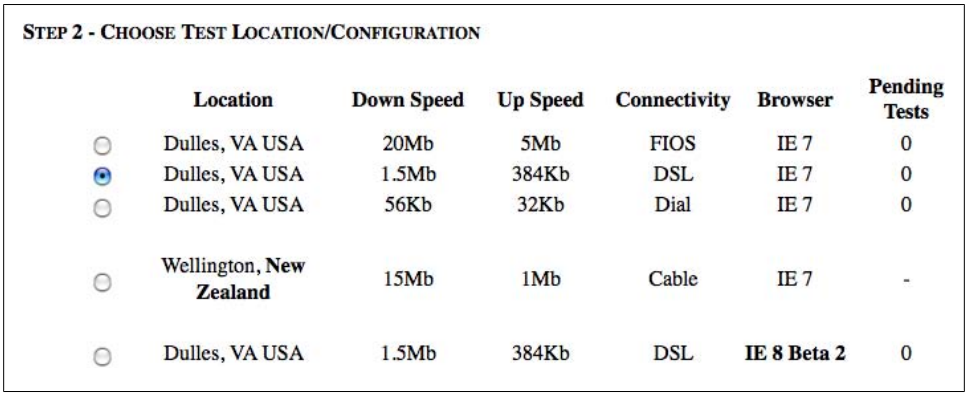


Figure 8-42. Selecting a testing location

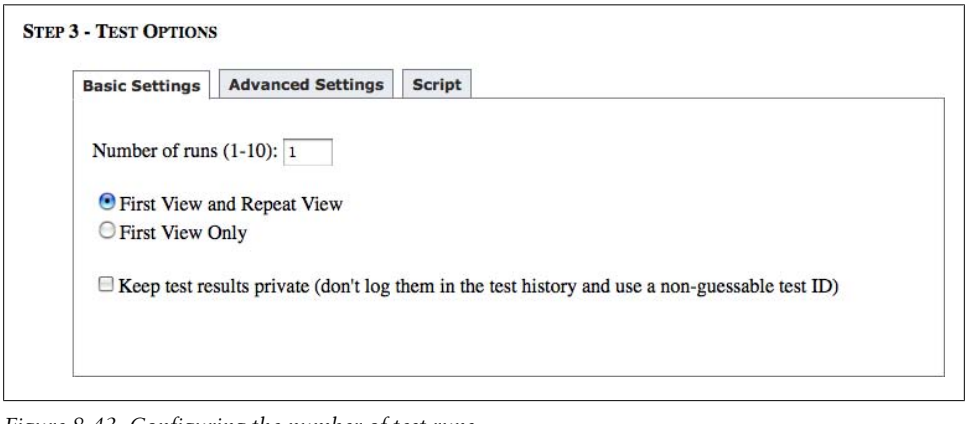


Figure 8-43. Configuring the number of test runs

Finally, customize how the test will run (Figure 8-43). You can specify the number of times the test will run, as well as whether to test a repeated view (where the browser cache is already filled).

More advanced options include where to stop measurement (once the entire page has loaded or when the DOM determines the document is complete) and how many parallel browser connections to simulate (Figure 8-44).

The result is an extremely comprehensive analysis of page performance across several runs. The system produces a performance summary like the one shown in Figure 8-45.

It also provides a cascade diagram of the various objects that were loaded within the page, along with many of the metrics and elements we've discussed (Figure 8-46).

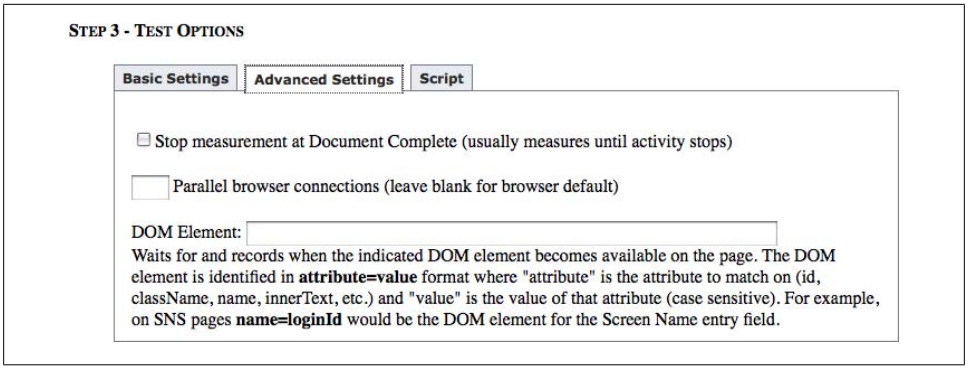
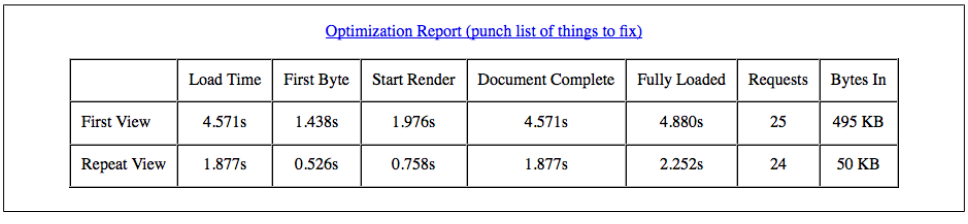


Figure 8-44. Adjusting parallelism and defining the end of page retrieval



	Load Time	First Byte	Start Render	Document Complete	Fully Loaded	Requests	Bytes In
First View	4.571s	1.438s	1.976s	4.571s	4.880s	25	495 KB
Repeat View	1.877s	0.526s	0.758s	1.877s	2.252s	24	50 KB

Figure 8-45. The high-level report of a PageTest run

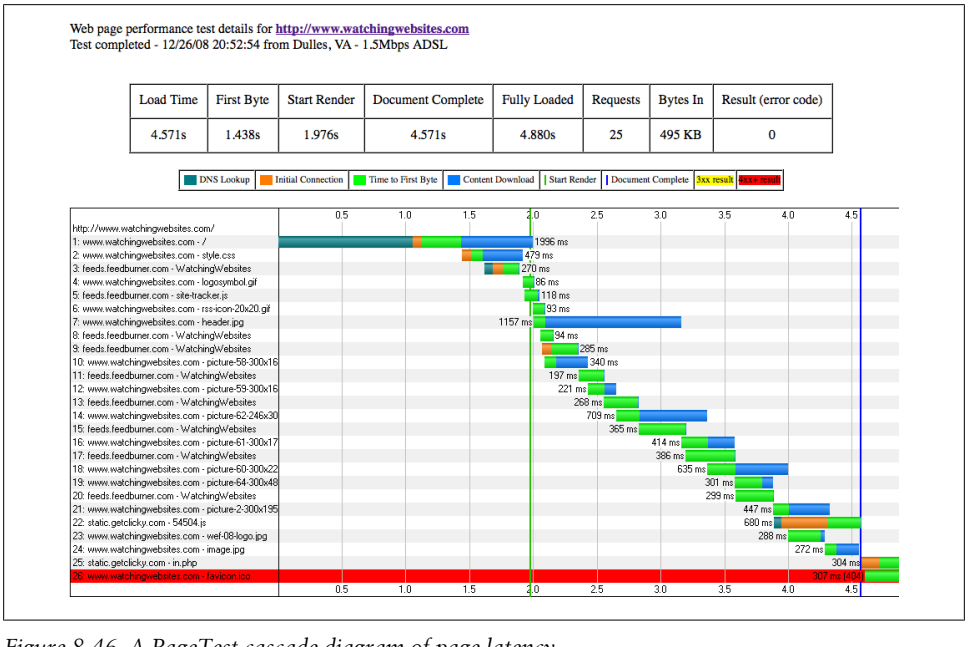


Figure 8-46. A PageTest cascade diagram of page latency

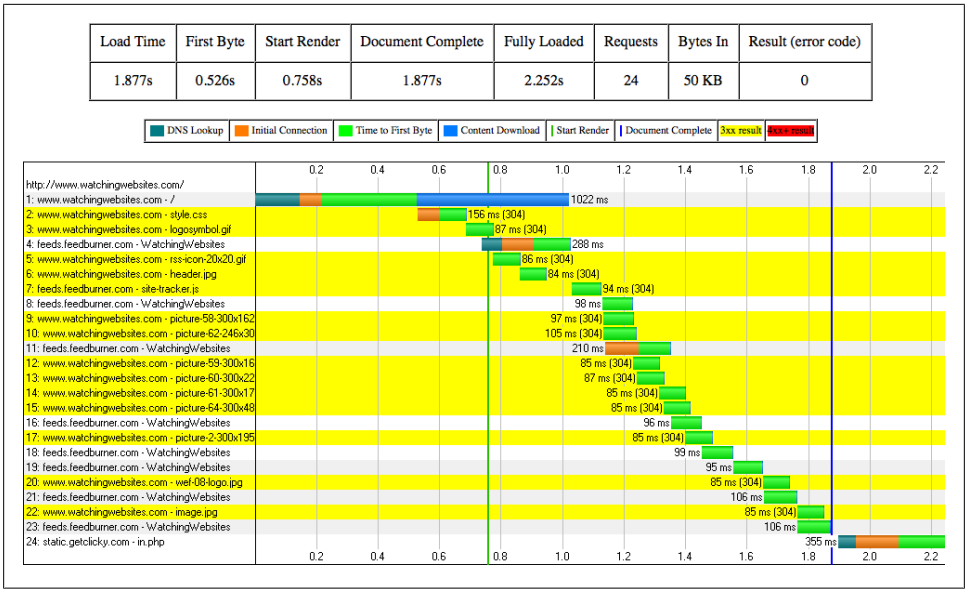


Figure 8-47. A cascade diagram for a PageTest simulation of a visitor whose browser has a full cache

Subsequent test runs reflect the user experience for visitors whose browsers have already cached the page. Get-if-modified requests (HTTP 304) are flagged, since many of these can be improved simply by setting caching parameters correctly so the browser doesn't check to see if it has the most recent version of content that changes infrequently (Figure 8-47).

Perhaps most importantly, the site makes performance recommendations similar to those of YSlow! that point to opportunities for performance improvement (Figure 8-48).

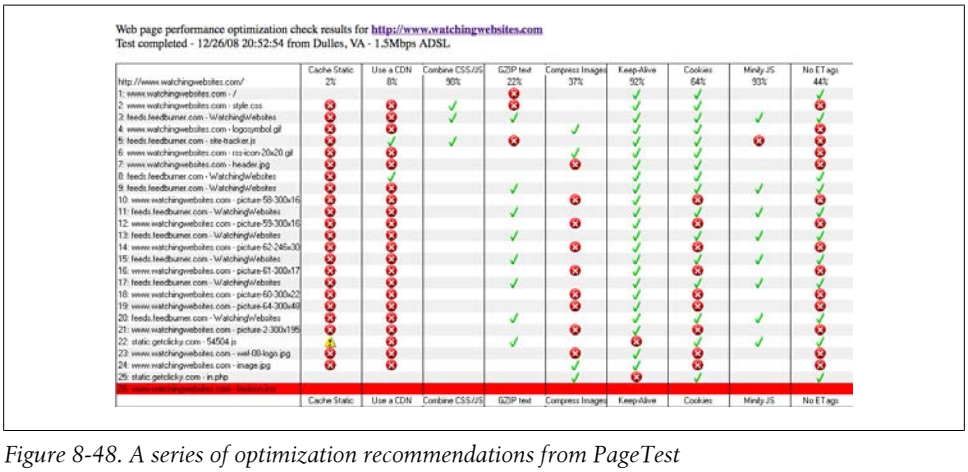


Figure 8-48. A series of optimization recommendations from PageTest

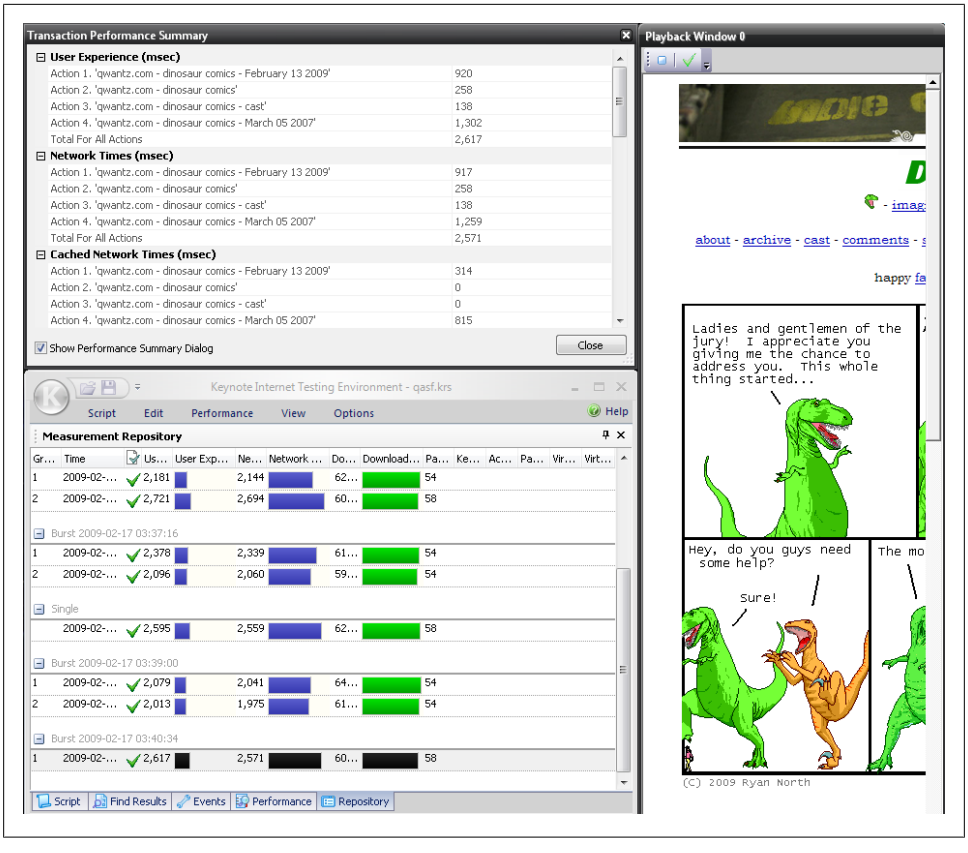


Figure 8-49. Keynote's KITE testing service

Another useful tool for Internet health testing, perhaps more suited to real-time operations than developer testing, is the Keynote Internet Testing Environment (KITE), a free service and Windows-based desktop application that can verify your site's health at various intervals, as shown in Figure 8-49.

KITE can drill down to individual objects on the tested pages to determine where performance and availability issues reside (Figure 8-50).

Getting developers and frontline operators using the same tools is a huge step toward breaking down organizational silos that exist in web monitoring. Operational teams responsible for real-time troubleshooting should train developers on Internet testing services. In return, developers should show web operators the desktop tools they use to examine web applications.

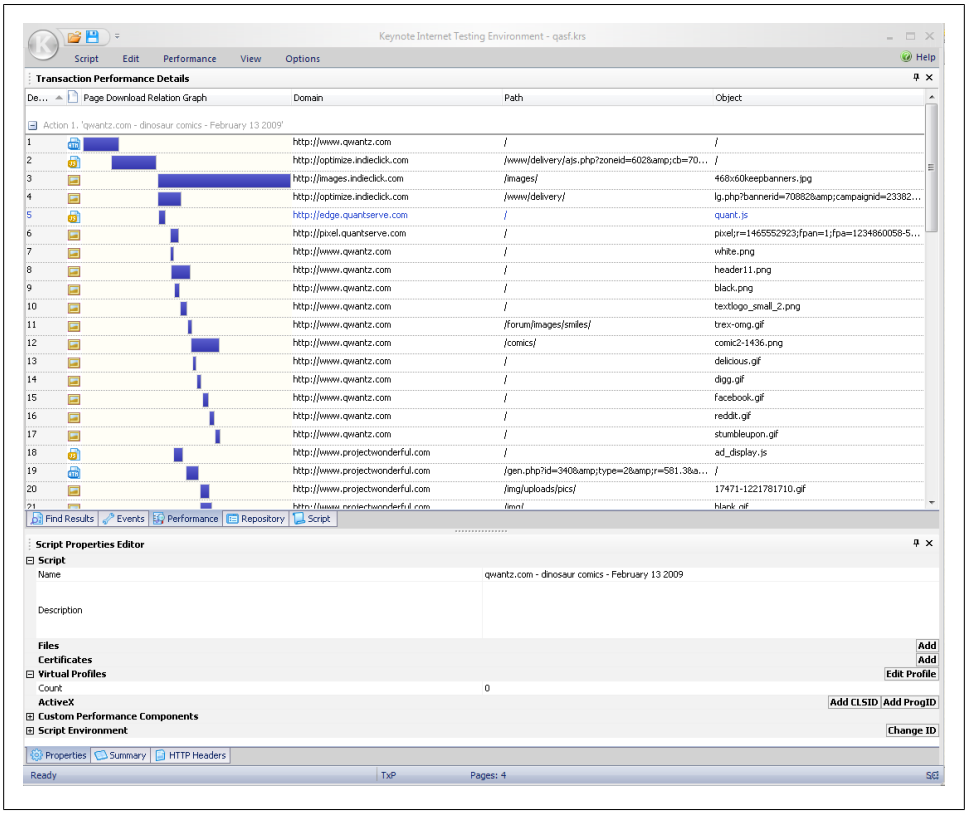


Figure 8-50. KITE drills down to object-level details, allowing you to determine your biggest sources of latency

Places and Tasks in User Experience

Recall the places-and-tasks model we outlined in Chapter 5. It's not just a good way to understand what users did—it's also useful for thinking about their experiences on your website.

Place Performance: Updating the Container

If the page is a place, we can measure the initial loading as outlined earlier. At some point, the page is loaded, and the user can interact with it. Something triggers a change to the place:

- Minor input, such as dragging an item into a shopping cart or upvoting a story.
- Client-side scripts that periodically refresh a display.

- Client-side prepushing (the opposite of prefetching), in which the browser sends content to the server ahead of time. This is a common pattern in blogging (saving a draft of a post) or webmail (uploading an attachment in the background).
- Server events that push new content out to the client across a nailed-up HTTP connection or a web socket.

These updates take time to load and may have errors, but they're only a small fraction of a full page retrieval. It's not correct to measure these updates as a sequence of events the way we do for pages. Instead, it's more accurate to describe them the way we would describe bandwidth: the delay between the server sending new content and the user seeing it. Place performance should therefore be described in terms of the frequency of updates and the latency with which the page is updated or the display refreshed.

Task Performance: Moving to the Next Step

Traditionally, the focus of monitoring is on tasks—and their abandonment—rather than on places. We expect users to move toward a particular outcome. This may still be within a single page (for example, a Flash shopping cart), but we have to measure the progress toward that outcome using a blend of analytics and performance metrics. These can include:

- Delay in loading the next step in the task
- Abandonment at each step
- The time the user spends thinking (considering an offer, filling out a form)

The start of a task is the moment the visitor clicks on the link to begin that task (for example, the Enroll button).

An organization with a mature monitoring strategy could map out places and tasks on its website, assign performance metrics to each of them (with a particular focus on latency in places and productivity in tasks) and tie this to business outcomes from analytics tools. This would provide a unified view of web health across the entire website that linked user experience to business goals.

Conclusions

Hopefully, you now have a good working knowledge of all the factors that affect web availability and performance. Your job is to understand the end user's experience as a result of these factors. To do this, you need to craft a monitoring strategy that can capture your site's performance as experienced by end users.

You might do this by testing critical pages or key functions at regular intervals—a synthetic testing model. Maybe you'll watch individual transactions and measure HTTP timings—RUM. As sites become more complex, you'll likely add your own application logging to round out the picture.

But as you design an EUEM strategy, remember that every web transaction begins with a set of requests and responses across multiple layers of communications protocols. However complicated a web application becomes, it's still all about turns, latency, and avoiding errors.

Now that we know all of the steps involved in web application delivery, we can look at how to measure performance.