acmqueue

Rules for Mobile Performance Optimization

An overview of techniques to speed page loading

Tammy Everts, Radware

Performance has always been crucial to the success of Web sites. A growing body of research has proven that even small improvements in page-load times lead to more sales, more ad revenue, more stickiness, and more customer satisfaction for enterprises ranging from small e-commerce shops to megachains such as Walmart.

For years Web developers could count on steady improvements in hardware and bandwidth to help deliver an optimal user experience. In recent years, however, the explosion of mobile Web browsing has reversed this. The lower bandwidth, higher latency, smaller memories, and lower processing power of mobile devices have imposed an even more urgent need to optimize performance at the front end in order to meet user expectations.

This article summarizes the case for front-end optimization and provides an overview of strategies and tactics to speed up your pages, with an emphasis on addressing mobile performance issues.

WHY PERFORMANCE MATTERS

No matter how interesting, beautiful, or cleverly interactive your Web pages are, if they take more than two or three seconds to render, whether on a desktop or a mobile device, users quickly become impatient. They are measurably less likely to convert from browsing to buying and may even hit the back button or close the browser before the page ever loads.

Even delays of less than one second significantly affect revenues. In 2006 Marissa Mayer, with Google at the time, recounted that, after users indicated they wanted to see more than 10 search results per page, Google experimented with showing 30 instead. To Google's surprise, traffic and revenue dropped by 20 percent in this experiment, apparently because the pages with more results took just an extra half-second to load.[5]

User expectations have only escalated since then. A 2009 study by Forrester Research on behalf of Akamai identified two seconds as the threshold for acceptable Web-page response times and found that 40 percent of consumers abandon a page that takes longer than three seconds to load. Just one year later, another study done for Akamai found that the number of users who abandon a page after three seconds had risen to 57 percent.[1,7]

Furthermore, users on mobile devices expect performance to be at least as good as—if not better than—what they experience on their desktops. The Harris Interactive 2011 Mobile Transactions Survey, commissioned by Tealeaf Technology (now part of IBM), reported that 85 percent of adults who had conducted a mobile transaction in the previous year expected the mobile experience to be equal to or better than shopping online using a laptop or desktop computer, and 63 percent said they would be less likely to buy from the same company via other channels if they experienced a problem conducting a transaction on their mobile phones.[10] In other words, poor mobile performance hurts companies on all other platforms, including brick-and-mortar.

Mobile traffic is expanding rapidly. For many consumers, their phone or tablet has become their primary portal to the Internet, but performance is falling short of expectations. A study published by Equation Research on behalf of Compuware in February 2011 found that almost half (46 percent) of mobile users said Web sites load more slowly than expected on their phones. Nearly 60 percent expect pages to load in three seconds or less, and 74 percent report they would leave a site if a single page took five seconds or more to load. A 2012 study of 200 leading e-commerce sites by Strangeloop Networks (now part of Radware) found that the median load time was 11.8 seconds over 3G (figure 1); performance over LTE fared only slightly better, at 8.5 seconds.[8]

THREE LIMITING FACTORS FOR MOBILE PERFORMANCE

As already mentioned, mobile devices have inherent performance limitations: lower bandwidth, smaller memories, and lower processing power. These challenges are compounded by external issues, notably:

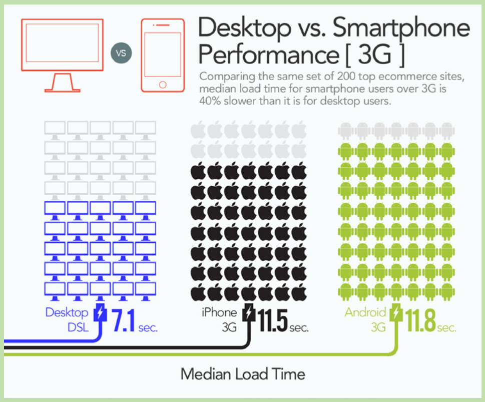
Web pages are bigger than ever. According to the HTTP Archive, the average Web page carries a payload of more than 1 MB and contains at least 80 resources such as images, JavaScript, CSS (Cascading Style Sheets) files, etc. This has a significant impact on desktop performance. Its impact on mobile performance—and particularly on 3G performance—is much more dramatic. This impact will be felt even more keenly over the next three years. At the current rate of growth, average page size could surpass 2 MB by 2015.

Latency can vary widely. It can range from as little as 34 ms for LTE to 350 ms or more for 3G. Mobile latency is consistent only in its inconsistency, even when measured at the same location. This is due to a number of variables beyond the amount of data passing through the tower. Factors such as the weather, and even the direction the user is facing, can have a significant impact.

Download speeds can also vary greatly. The speeds can range from a mere 1 Mbps over 3G to as much as 31 Mbps over LTE. It's interesting to compare this to the average U.S. broadband speed of 15 Mbps, and to note that 3G can be up to 15 times slower than broadband, while LTE can be up to twice as fast.

FIGURE 1

Median Load Times for Desktop and Mobile Devices



Source: Strangeloop

M.SITES ARE NOT A CURE-ALL FOR MOBILE PERFORMANCE PAINS

Many site owners attempt to respond to the combination of high user demands, large Web pages, and poor connection speeds by developing smaller, faster, stripped-down *m.sites*; however, these attempts are not completely effective, as up to 35 percent of mobile users will choose to view the full site when given the option.

These full-site visitors are significantly more likely to spend than m.site visitors. One study found that for every $7.00 of mobile-generated revenue, $5.50 was generated via full sites. Only $1.00 came via m.sites, and $0.50 via apps.[9]

ADDRESSING THE PROBLEM

The chief strategies for improving site performance have not changed as usage has migrated from the desktop to mobile phones and tablets, although a few new tactics have emerged.

Only 20 percent of the time required to display a typical Web page, whether in a desktop or mobile browser, is consumed by loading the page's HTML. The remaining 80 percent is spent loading the additional resources needed to render the page—including style sheets, script files, and images—and performing client-side processing.

The three main strategies for improving performance are:

- Reducing the number of HTTP requests required to fetch the resources for each page.
- Reducing the size of the payload needed to fulfill each request.
- Optimizing client-side processing priorities and script execution efficiency.

Because mobile networks are usually slower than those available to desktop machines, reducing requests and payloads takes on huge importance. Mobile browsers are slower to parse HTML and execute JavaScript, so optimizing client-side processing is crucial. In addition, mobile browser caches are much smaller than those of desktop browsers, requiring new approaches to leveraging local storage of reusable resources.

The remainder of this article summarizes tactics you can use to address these challenges. While automated tools are available for most of these practices, many can also be implemented manually (by an experienced front-end developer). It is crucial to note that an overarching challenge with the manual implementation of many of these techniques is control of resources. Often in CMS (content management system) or other Web applications, pages can include HTML snippets, CSS, and JavaScript files that are either generated or hosted off-site, meaning developers do not have access to optimize them.

REDUCE REQUESTS

The biggest drain on performance is usually the need to complete dozens of network round-trips to retrieve resources such as style sheets, scripts, and images. This is especially true with the relatively low bandwidth and high latency of mobile connections. CDNs (content delivery networks) can help a bit by bringing content geographically closer to users, but the number of requests has a much greater impact on page-load times than the distances those requests travel. In addition, recent findings suggest that CDNs have limited effectiveness for mobile users.[3]

The sections that follow discuss several approaches to minimizing HTTP requests.

CONSOLIDATE RESOURCES

By now it's standard practice for developers to consolidate JavaScript code and CSS styles into common files that can be shared across multiple pages. This technique simplifies code maintenance and improves the efficiency of client-side caching.

In JavaScript files, be sure that the same script isn't downloaded multiple times for one page. Redundant script downloads are especially likely when large teams or multiple teams collaborate on page development. It might surprise you how often this occurs.

Spriting is a CSS technique for consolidating images. Sprites are simply multiple images combined into a rectilinear grid in one large image. The page fetches the large image all at once as a single CSS background image and then uses CSS background positioning to display the individual component images as needed on the page. This reduces multiple requests to only one, significantly improving performance.

Ease of implementation: Moderate, but requires having access to resources. Depending on the level of control that the developer has over the Web site, some resources may not be able to be consolidated (e.g., if they are generated by a CMS). Also, some resources may be located on external domains, which can cause problems for consolidation. It's also important to note that resource consolidation can be a double-edged sword for mobile browsers. Reducing requests improves performance the first time, but larger consolidated resources may not be cached efficiently, so be careful to combine consolidation techniques with other techniques to optimize `localStorage`.

USE BROWSER CACHING AND LOCALSTORAGE

All modern browsers use local memory to cache resources that have been tagged with Cache-Control or Expires headers that indicate how long the item can be cached. In addition, ETag (entity tag) and Last-Modified headers indicate how resources should be repopulated in the cache after they have expired. The browser fetches cached items locally whenever possible, avoiding unnecessary server requests, and it flushes items that have expired or haven't been used recently when cache space runs short. The resources stored in browser object caches commonly include images, CSS, and JavaScript code, and caching is essential to acceptable site performance. (A separate cache holds entire rendered pages to support use of the Back and Forward buttons.)

Mobile browser caches, however, are usually much smaller than those on desktop machines, causing items to be flushed quickly. The HTML5 Web storage specification provides a great alternative to relying only on browser caching. The HTML5 `localStorage` JavaScript object has been implemented in all the major desktop and mobile browsers. Script code can easily check for support of HTML5 `localStorage` and then use it, if available, to save and read key/value data, usually 5 MB per domain. This capability makes `localStorage` very well suited for client-side caching, although read/write speeds do vary for different mobile browsers. It is usually significantly faster to retrieve a resource from `localStorage` than to request it from a server, and it is more flexible and reliable than relying only on cache headers and the limited browser cache storage available on most mobile devices. In addition, this is one area where mobile browsers are currently ahead of the desktop in efficiency—`localStorage` performance has lagged in desktop implementations, where using the standard browser cache may still be the best option.

Ease of implementation: Advanced. While the `localStorage` mechanism may be simple to use, building a cache around it does create some complexities. You will need to take into account all of the issues that a cache handles for you, such as cache expiry (when do you remove items?), cache misses (what if you expected something to be in `localStorage` and it is not?), and what to do when the cache is full.

EMBED RESOURCES IN HTML FOR FIRST-TIME USE

The standard pattern in HTML is to include links to external resources. This makes it easier to maintain these resources as files on the server (or in a CDN) and to update them at the source rather than in each of many pages. This pattern also supports browser caching by allowing cached resources to be fetched automatically from the cache rather than from the server, as previously discussed.

For resources that aren't already cached in the browser or in `localStorage`, however, this pattern of linking to external resources has a negative impact on performance. A typical page can require dozens of separate requests in order to gather the resources needed to render the page. So, from a performance standpoint, if a resource doesn't have a high likelihood of already being cached, it is often best to embed that resource in the page's HTML (called *inlining*) rather than storing it externally and linking to it. Script and style tags are supported in HTML for inlining those resources, but images and other binary resources can also be inlined by using data URIs that contain base64-encoded text versions of the resources.

The disadvantage of inlining is that page size can become very large, so it is crucial for a Web application that uses this strategy to be able to track when the resource is needed and when it is already cached on the client. In addition, the application must generate code to store the resource on the client after sending it inline the first time. For this reason, using HTML5 `localStorage` on mobile devices is a great companion to inlining.

Ease of implementation: Moderate. This technique requires the site to have a mechanism to generate a different version of the page based on whether the user has visited that page before.

USE HTML5 SERVER-SENT EVENTS

Web applications have used various polling techniques to update pages continually by requesting new data from a server. The HTML5 `EventSource` object and `Server-Sent` events enable JavaScript code in the browser to open a much more efficient unidirectional channel from the server to the browser. The server can then use this channel to send data as it becomes available, eliminating the HTTP overhead of creating multiple polling requests. This is also more efficient than HTML5 WebSockets, which is a richer protocol for creating a two-way channel over a full-duplex connection when a lot of client-server interaction is called for, such as in messaging or gaming.

Ease of implementation: Advanced. This technique is very implementation-specific. If your site is currently using other AJAX or Comet techniques for polling, converting to use Server-Sent events may take quite a bit of recoding the site's JavaScript.

ELIMINATE REDIRECTS

When users attempt to navigate to a standard desktop site from a mobile device, the Web application often will read the user-agent HTTP header to detect that the request is from a mobile device. The application then can send an HTTP 301 (or 302) response with an empty body and a Location header, redirecting the user to the mobile version of the site as required. However, the extra round-trip to the client and back to the mobile site often consumes several hundred milliseconds over mobile networks. Instead, it is faster to deliver the mobile Web page directly in response to the original request, rather than delivering a redirect message that then requests the mobile page.

As a courtesy to users who prefer to view the desktop site even on their mobile devices, you can provide a link on the mobile site that signals your application to suppress this behavior.

Ease of implementation: While this technique is easy in theory, it may not always be possible to put into practice. Many sites redirect to a different server for their m.sites, since those may be hosted elsewhere. Other sites send cookies with the redirect to tell the Web application that they are mobile once they redirect. This may be tougher to control, depending on the Web application.

REDUCE PAYLOAD

Size matters. Smaller pages render faster, and smaller resources are fetched faster. Reducing the size of each server response doesn't usually help performance as much as reducing the number of responses needed for each page. Several techniques, however, do yield a net benefit for performance, especially on mobile devices where bandwidth and processing power must be managed wisely.

COMPRESS TEXT AND IMAGES

Compression technologies such as gzip reduce payloads at the slight cost of adding processing steps to compress on the server and decompress in the browser. These operations are highly optimized, however, and tests show that the overall effect is a net improvement in performance. Text-based responses, including HTML, XML, JSON (JavaScript Object Notation), JavaScript, and CSS, can all be reduced in size by as much as 70 percent.

Browsers announce their decompression capabilities in the Accept-Encoding request header, and they perform decompression automatically when servers signal that a response is compressed in the Content-Encoding response header.

Ease of implementation: Easy. All modern Web servers will support compressing responses if correctly set up. However, there are still desktop security tools that will remove the Accept-Encoding headers from requests, which will prevent users from getting compressed responses even though their browsers support it.

MINIFY CODE

Minification, which is usually applied only to scripts and style sheets, eliminates inessential characters such as spaces, newline characters, and comments. Names that are not publicly exposed, such as variable names, can be shortened to just one or two characters. A correctly minified resource is used on the client without any special processing, and file-size reductions average about 20 percent. Script and style blocks within HTML pages can also be minified. There are many good libraries available to perform minification, often along with services to combine multiple files into one, which additionally reduces requests.

Minification not only reduces bandwidth consumption and latency, but also may mean the difference between a cacheable object and one that is too big for the cache on a particular mobile device. Gzip compression is no help in this regard, because objects are cached by the browser after they have been decompressed.

Ease of implementation: Easy. The Closure Compiler from Google does an incredible job of understanding and minifying JavaScript. CSS minification is a little more troublesome as there are so many CSS hacks for different browsers that can easily confuse minifiers or no longer work correctly after minification. Also note that there have been published reports of minification breaking pages, even though the removed characters should not be essential. So be sure to perform functional tests on any pages where you apply this technique.

RESIZE IMAGES

Images often consume the majority of the network resources required to load Web pages, and the majority of the space required to cache page resources. Small-screen mobile devices present opportunities for speeding transmission and rendering by resizing images. High-resolution images waste bandwidth, processing time, and cache space if the user will be viewing the images only in a small mobile browser window.

To speed up page rendering and reduce bandwidth and memory consumption, dynamically resize images in your application or replace images with smaller versions for mobile sites. Don't waste bandwidth by relying on the browser to scale a high-resolution image into a smaller width and height.

Another option is to load a very low-resolution version of an image initially to get the page up as quickly as possible and then replace that with a higher-resolution version on the `onload` or ready event after the user has begun interacting with the page.

Ease of implementation: Advanced, especially for highly dynamic sites.

SIMPLIFY PAGES WITH HTML5 AND CSS 3.0

The HTML5 specification includes new structural elements, such as `header`, `nav`, `article`, and `footer`. Using these semantic elements yields a simpler and more efficiently parsed page than using generic nested div and span tags. A simpler page is smaller and loads faster, and a simpler DOM (Document Object Model) means faster JavaScript execution. The new tags are quickly being adopted in new browser versions, including mobile browsers, and HTML5 was designed to degrade gracefully in browsers that don't yet support it.

HTML5 input elements in forms support lots of new attributes that enable declarative HTML code to implement features that previously required JavaScript. For example, the new placeholder attribute can specify instructional text that appears until a user makes an entry, and the new autofocus attribute can specify which input should automatically get the initial focus.

There are also several new types of input elements, which automatically implement commonly needed features without JavaScript. The new types include e-mail, URL, number, range, date, and time, which are efficiently rendered as complex controls with friendly user interfaces and validation. In mobile browsers, the pop-up keyboards often automatically provide keystroke choices appropriate to the specified input type when text input is required. Browsers that don't support the specified input type will simply display a text box.

In addition, new CSS 3.0 features can help create lightweight pages by providing built-in support for gradients, rounded borders, shadows, animations, transitions, and other graphical effects that previously required images to be loaded. These new features can speed up page rendering.

A number of Web sites offer regularly updated lists showing which features are supported by which desktop and mobile browsers (for example, http://caniuse.com/ and http://mobilehtml5.org/).

Ease of implementation: Advanced. Making these changes manually is extremely complex and time consuming, if not impossible. If you use a CMS, it may generate a great deal of HTML and CSS that you have no control over.

OPTIMIZE CLIENT-SIDE PROCESSING

The order in which a browser executes the various steps needed to construct a page can have a major impact on performance, as do the complexity of the page and the choice of JavaScript techniques. This is especially true on mobile devices where client-side processing is constrained by slower CPUs and less memory. The following sections provide some tactics for increasing the efficiency of page processing.

DEFER RENDERING BELOW-THE-FOLD CONTENT

You can assure that the user sees the page quicker by delaying the loading and rendering of any content that is below the initially visible area, sometimes called "below the fold." To eliminate the need to reflow content after the remainder of the page is loaded, replace images initially with placeholder `<img>` tags that specify the correct height and width.

Ease of implementation: Moderate. Some good JavaScript libraries are available that can be used for below-the-fold lazy image loading.[12]

DEFER LOADING AND EXECUTING SCRIPTS

Parsing JavaScript can take up to 100 milliseconds per kilobyte of code on some mobile devices. Many script libraries aren't needed until after a page has finished rendering. Downloading and parsing these scripts can safely be deferred until after the `onload` event. For example, scripts that support interactive user behavior, such as drag and drop, can't possibly be called before the user has even seen the page. The same logic applies to script execution. Defer as much as possible until after `onload` instead of needlessly holding up the initial rendering of the important visible content on the page.

The script to defer could be your own or, often more importantly, script from third parties. Poorly optimized scripts for advertisements, social media widgets, or analytics support can block a page from rendering, sometimes adding precious seconds to load times. Also, carefully evaluate the use of large script frameworks such as jQuery for mobile sites, especially if you are using only a couple of objects in the framework.

Ease of implementation: Moderate. Many third-party frameworks now provide deferred or async versions of their APIs. The developer just has to switch to these new versions. Some JavaScript may be more complex to defer as there are many caveats to running scripts after `onload` (e.g., what do you do if you have a script that wants to attach to the `onload` event? If you defer it until after `onload`, it has missed its chance).

USE AJAX FOR PROGRESSIVE ENHANCEMENT

AJAX (Asynchronous JavaScript and XML) is a technique for using the XHR (`XMLHttpRequest`) object to fetch data from a Web server without refreshing the page where the code is running. AJAX enables a page to display updated data in a section of a page without reconstructing the entire page. This is often used to respond to user interaction, but it can also enable your application to load a bare-bones version of a page quickly, and then to fill in more detailed content while the user is already viewing the page.

Despite the name, `XMLHttpRequest` doesn't tie you to using only XML. You can call its `overrideMimeType` method to specify "application/json" and work with JSON instead of XML. Using `JSON.parse` is up to twice as fast and more secure than using the generic `eval()` function.

Also, remember that AJAX responses will benefit from many of the same optimization techniques recommended for standard responses. Be sure to apply cache headers, minification, gzip compression, resource consolidation, etc. to your AJAX responses.

Ease of implementation: Difficult to quantify, as this technique is very application-specific. Because of cross-domain issues, you would need to use XHR2, as well as control the external domain to make cross-domain XHR requests.

ADAPT TO THE NETWORK CONNECTION

Especially with mobile networks that may charge extra for using more bandwidth, certain techniques should be used only when combined with code to detect the type of connection. For example, preloading resources in anticipation of future requests is usually smart, but it may not be a responsible strategy if the user's bandwidth is metered and some of those resources may never be needed.

On Android 2.2+, the *navigator.connection.type* property returns values that allow you to differentiate Wi-Fi from 2G/3G/4G connections. On Blackberry, *blackberry.network* provides similar information. In addition, server-side detection of User-Agent header data or other information embedded in requests can alert your application to the quality of the connection in use.

Ease of implementation: Advanced. The Network Information API has changed recently.[11] Rather than defining the network as Wi-Fi, 3G, etc., it now gives information about the bandwidth, with suggested values such as "very-slow, slow, fast and very-fast." There is a property that tries to tell the estimated MB/s, and a Boolean "metered" measurement that does its best to be correct, but this is very difficult for a browser to determine. Measuring somewhere and adapting is probably still the best idea but is quite challenging.

USE THE HTML5 WEB WORKER SPEC FOR MULTITHREADING

The Web Worker specification in HTML5 introduces multithreaded concurrent execution to the world of JavaScript programming. In addition, this particular implementation of multithreading eliminates problems that have plagued developers working with multiple threads on other platforms—specifically, problems that occur when one thread modifies a resource that is also being used by another thread. In Web Worker code, spawned threads cannot access the resources of the main user-interface (UI) thread.

For improving the performance of mobile sites, Web Worker code can be valuable for preloading resources that a user is likely to need to complete future actions, especially when the user's bandwidth isn't metered. With the limited processor capabilities of mobile devices, extensive preloading can interfere with UI responsiveness in the current page. Using multithreaded code that employs Web Worker objects (and possibly `localStorage` to cache the data), operations that preload resources can execute on a separate thread without impacting current UI performance.

Note that the Web Worker spec, while implemented in Android since 2.0, was not supported on the iPhone until iOS 5. On the desktop, Internet Explorer was the laggard, adding support for Web Worker only in IE 10.

Ease of implementation: Moderate. While this technique is not incredibly difficult to implement, there are some restrictions to Web Workers that make them tough to find places for. They do not have access to the page's DOM and cannot modify anything on the page. Making this practice work requires a very specific type of background calculation or process that fits well as a background Web Worker.

REPLACE CLICK EVENTS WITH TOUCH EVENTS

On touch-screen devices, the `onclick` event does not fire immediately when a user taps the screen. Instead, the device waits up to half a second (300 milliseconds on most devices), giving the user a chance to initiate some other gesture rather than a click. This delay, however, can significantly impede the responsive performance that users expect. To fix this, use the `touchend` event instead. That event fires immediately when the user taps the screen.

To ensure that the user doesn't experience unexpected behavior, you may also want to use the `touchstart` and `touchmove` events. For example, don't assume that `touchend` on a button means click unless there was also a `touchstart` event on the button—not if the user touched somewhere else and dragged to the button before ending the touch. You could use a `touchmove` event after `touchstart` to prevent treating the following `touchend` as a click, assuming that the moving gesture wasn't intended to be a click.

In addition, you may still want to handle the `onclick` event to ensure that the browser changes the appearance of the button to show a clicked state, and to support browsers that don't handle touch events. To avoid duplicate code execution when both `touchend` and `onclick` code fire, add a click event handler that calls `preventDefault` and `stopPropagation` if the click was the result of a user tap that was already handled by `touchend`.[4]

Ease of implementation: Advanced. This technique requires much more work to add and maintain links on a page. The code testing for touch events must be resilient against gestures that may be happening instead of a click, such as a zoom or swipe.

SUPPORT THE SPDY PROTOCOL

Some of the performance bottlenecks that afflict Web sites, whether desktop or mobile, result from inefficiencies in the application-layer HTTP and HTTPS protocols. In 2009, Google began work on an alternative protocol named SPDY (pronounced "speedy") that addresses some of these limitations. The goal is to make this an open source project that will be implemented by multiple browsers and Web servers, but initially it was supported only in Google's Chrome browser (in version 10 or later) and on Google sites. As Web servers are released that implement SPDY, sites will be able to use this protocol for any user with a browser that supports it. In a test implementing SPDY on a representative group of 25 of the top 100 Internet sites, Google observed speed improvements ranging from 27 percent to 60 percent.[2]

SPDY automatically uses gzip compression on all content, and unlike HTTP, it also uses gzip on header data. SPDY employs multiplexing technology to enable multiple streams of requests or responses to be sent over a single TCP connection. In addition, SPDY allows requests to be prioritized, so, for example, a video that is central to a page's content can be given a higher priority than an advertisement in the margin.

Perhaps the most revolutionary innovation in SPDY is that streams can be bidirectional and can be initiated by either the client or the server, allowing content to be pushed to clients without first being requested. For example, when a user first visits a site, and therefore doesn't yet have any of the site content cached, the server can push all required resources in response to the first page request instead of waiting for each resource to be separately requested. As an alternative, the server can send hints to the client, suggesting resources that will be needed, but still allowing the client to initiate the requests. This is still faster than waiting for the client to parse the site pages and discover the resource requirements on its own.

Although SPDY is not specific to mobile platforms, the limited bandwidth available over mobile networks means that SPDY optimizations will be especially useful in reducing latency for mobile sites when supported.

Ease of implementation: Moderate to advanced, depending on the site and server environment. Google has a SPDY module for Apache 2.2—`mod_spdy`—that is available for free; however, `mod_spdy` has threading model issues and doesn't play well with `mod_php` by default, so this requires additional attention in order to ensure that it is running correctly on your site.[6]

DON'T FORGET TO TEST!

No discussion of performance optimization would be complete without a reminder that continuous and careful testing is essential. Every change to your system is just a theory until it is tested against a baseline. Guessing where performance bottlenecks occur is meaningless unless based on real test data.

Great open source and commercial tools are available to provide synthetic tests, complete with geographical distribution and bandwidth/latency throttling. In addition, RUM (real user monitoring) tools take testing out of the lab and into the field of unpredictable user behavior.

Look for testing options that support mobile, as well as desktop scenarios. If you choose an automated solution, be sure to choose one that continually tests and refines the optimizations it applies.

Performance optimization cannot be effective if it is merely a single step in a linear development process. Rather, it must become part of an ongoing cycle of continuous improvement.

REFERENCES

1. Bustos, L. 2009. Every second counts; how website performance impacts shopper behavior. GetElastic; http://www.getelastic.com/performance/.
2. Chromium Projects. SPDY: an experimental protocol for a faster Web; https://sites.google.com/a/chromium.org/dev/spdy/spdy-whitepaper.
3. Everts, T. 2013. Case study: how effective are CDNs for mobile visitors. Web Performance Today; http://www.Webperformancetoday.com/2013/05/09/case-study-cdn-content-delivery-network-mobile-3g/.
4. Fioravanti, R. 2011. Creating fast buttons for mobile Web applications. Google Developers; http://code.google.com/mobile/articles/fast_buttons.html.
5. Linden, G. 2006. Marissa Mayer at Web 2.0. Geeking with Greg; http://glinden.blogspot.com/2006/11/marissa-mayer-at-Web-20.html.
6. mod-spdy; http://code.google.com/p/mod-spdy/.
7. PhoCusWright. 2010. PhoCusWright/Akamai study on travel site performance; http://connect.phocuswright.com/2010/06/phocuswrightakamai-study-on-travel-site-performance/; http://www.akamai.com/dl/whitepapers/Akamai_PCW_Travel_Perf_Whitepaper.pdf.

8. Radware. 2011. Case studies from the mobile frontier: the relationship between faster mobile sites and business KPIS; http://www.strangeloopnetworks.com/resources/research/state-of-mobile-ecommerce-performance/.
9. Bixby, J. 2012. 2012 state of mobile e-commerce performance; http://www.strangeloopnetworks.com/resources/videos/case-studies-from-the-mobile-frontier-the-relationship-between-faster-mobile-sites-and-business-kpis-video/.
10. Tealeaf. 2011. Report on the Mobile Customer Experience. Based on the Harris Interactive 2011 Mobile Transactions Survey.
11. W3C. 2012. Network Information API; http://www.w3.org/TR/netinfo-api/.
12. YUI. ImageLoader. Yahoo! User Interface Library; http://yuilibrary.com/yui/docs/imageloader/.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

TAMMY EVERTS has worked in user experience and Web performance since 1997. She currently works at Radware, evangelizing performance both in-house and out in the world. Previously, she held roles as research director at Strangeloop Networks and director of user experience at Habanero Consulting. She blogs about performance issues, research, and trends at http://webperformancetoday.com.