The unprecedented popularity of mobile devices and their ubiquitous access to cellular data networks make surfing the World Wide Web (WWW) on-the-go a common sight. Mobile browsers have become one of the key entities in the smartphone ecosystem, with their generated mobile traffic volume exceeding that of any other application except for video streaming. Moreover, as the standard Web interface, HTTP is used by millions of smartphone apps, and many apps are simply customized programmable browsers.

The term mobile-friendly has been used in many contexts including, in particular, UI design of mobile apps and websites. Indeed, many websites do have their appearance tailored to mobile devices’ screens. A recent measurement study shows that 65 percent of the Alexa top 500 websites have mobile versions that are specifically designed for handheld devices. However, loading a webpage is a complex procedure involving many subsystems: object downloading, CSS/JavaScript parsing, content rendering, cache management, and so on. Only changing the appearance of a mobile website is therefore often superficial.

To achieve mobile-friendly Web browsing, three factors must be optimized: performance, energy usage, and bandwidth consumption. First, Internet users are sensitive to webpage load time (PLT). For example, with an extra delay of 500 milliseconds, Google will lose up to 20 percent traffic. With a 100 millisecond extra delay, Amazon will lose 1 percent in sales. In the mobile world, achieving fast page loading speed is more challenging due to unpredictable network conditions (for example, due to mobility) and the limited processing capability of handheld devices. Second, battery life has long been an issue for mobile devices. Over the past 15 years, the CPU performance has improved 250 times while the capacity of the li-ion battery has only doubled. In particular, the power-hungry cellular interface (3G Universal Mobile Telecommunications System/High-Speed Packet Access, or UMTS/HSPA, and 4G LTE) worsens the energy issue. Third, bandwidth is also a critical resource for cellular customers who are billed by their data plan usage. Therefore under the constraints of providing a satisfactory user experience, the bandwidth consumption of mobile Web needs to be minimized.

The remainder of this article discusses why today’s mobile Web is often not mobile-friendly, and proposes suggestions on improving the state-of-the-art. I will take a top-down approach by describing issues at each layer: website contents, the Web protocol (HTTP), the Secure Sockets Layer/Transport Layer Security (SSL/TLS) encryption, and the transport protocol. In many cases, the inefficiencies aren’t caused by a single layer but instead by unexpected cross-layer interactions.

Website Content

Despite its good looks, a professionally designed mobile website might consume an unexpectedly large amount of resources on a mobile device. Typical issues include using unnecessarily high-resolution images, embedding within a single page too much content that few users will read due to having to scroll down to the page’s bottom, employing complex CSS and JavaScript, and using excessive redirects that hurt the PLT. As a concrete example,
the height (that is, vertical dimension) of some popular mobile websites’ landing pages can reach up to 40 times of a smartphone’s screen height, leading to several megabytes of data being transferred during a page loading.1

These issues aren’t difficult to comprehend, detect, and fix. However, there are trickier problems that can be easily overlooked due to lack of awareness of how cellular radio works. We know that the power consumption characteristics of the cellular interface are quite different from those in Wi-Fi and wired networks. In cellular networks, it’s much more energy-efficient to transmit data in a single bundle, instead of sending them slowly and separately. This is because after a data transfer, the radio interface isn’t turned off until a fixed timer, called a tail timer, expires. Therefore, having multiple transfers taking place intermittently will significantly lengthen the radio-on time, leading to extra battery drainage, as Figure 1 shows.

The cellular tail effect has several implications on mobile Web browsing. As an example, copied from their desktop versions, many mobile sites perform infinite scrolling: when the user scrolls down to the bottom of a page, the browser will load and append more content to the page. This behavior is totally legitimate in wired networks. But in cellular networks, this bursty traffic pattern (see Figure 1) can potentially keep the radio interface always on as the user slowly scrolls the page, leading to energy inefficiencies. Another representative example is that many websites issue periodical pings for tracking users. These periodical pings are usually triggered by third-party JavaScript (for example, Chartbeat.com) that is embedded in the main HTML page. Again due to the tail effect, these periodical requests account for most of the radio energy consumption of loading a page although their sizes are small.

There are several fixes for these issues. Web designers should balance between a large initial loading and many small incremental loadings, which incur a key tradeoff between bandwidth and energy consumption. JavaScript-triggered delayed or periodical transfers should be minimized unless they are really necessary. For delay-tolerant transfers such as user tracking, there is usually some leeway in terms of when to schedule them. Therefore, their transmissions can be shifted to overlap with delay sensitive data to reduce the impact of the tails. Similarly, multiple instances of delay-tolerant transfers can also be batched together. Ideally, both optimizations (called piggybacking and batching, respectively) need to gain browser support.

Caching is another effective mechanism to reduce bandwidth consumption by eliminating redundant data transfers. The effectiveness of caching relies on two aspects: correct caching implementation (browsers must strictly conform to the protocol specification) and good caching semantics (content providers should properly set objects’ caching parameters, such as life time). Regarding caching implementation, prior measurement4 reveals that quite a few HTTP libraries don’t perform any caching, and even some popular mobile browsers don’t fully support HTTP/1.1 caching. For caching semantics, many professionally designed pages contain objects with a short lifetime (for example, 1 hour), and such objects often belong to images, fonts, and CSS files, that are not expected to change frequently. A similar situation happens with compression, which is often underserved for compressible textual objects such as HTML and JavaScript files.

The total page load time is 7.7 seconds.

For example, on a Samsung Galaxy S5 smartphone, I conducted an experiment over a commercial LTE network by loading CNN.com, whose 240 objects (1.4 Mbytes’ worth of data) from 70 domains were downloaded by 137 connections.

### HTTP and Its Interplay with TCP

Now we shift our focus from website contents to the Web protocol. As the key protocol that supports the WWW, HTTP has been stunningly successful. Based on recent measurement studies, HTTP accounts for at least 52 percent of Internet traffic,5 and 82 percent of the traffic delivered to mobile devices.6 The percentages are increasing because more and more non-Web applications are using HTTP.

HTTP functions as a request–response protocol. The client, such as a Web browser, sends an HTTP request message to the server asking for a particular resource object (for example, an HTML page or an image). The server then returns with an HTTP response containing the object data. HTTP runs above the Transmission Control Protocol (TCP), which ensures reliable and in-order delivery of the underlying byte stream over the network.

HTTP has been evolving during the past 25 years. The current HTTP version used by the vast majority of today’s Web servers is HTTP/1.1, which was standardized in 1999.7 However, HTTP/1.1 exhibits performance issues as webpages become rich and complex. A modern webpage might consist of hundreds of objects, which are loaded by a large number of short-lived TCP connections in today’s HTTP/1.1 scheme. For example, on a Samsung Galaxy S5 smartphone, I conducted an experiment over a commercial LTE network by loading CNN.com, whose 240 objects (1.4 Mbytes’ worth of data) from 70 domains were downloaded by 137 connections. The total page load time is 7.7 seconds.
Beyond Wires

From HTTP/1.1 to HTTP/2

HTTP/2 is the next planned version of HTTP. It aims at overcoming many limitations of HTTP/1.1 by redesigning the data transfer paradigm of HTTP. The Internet Engineering Task Force (IETF) HTTP working group began working on HTTP/2 in 2012. In 2015, the HTTP/2 specification was finally approved by IETF for standardization, and was published as RFC 7540. The design of HTTP/2 draws heavily from SPDY, a recently proposed protocol by Google for improving Web performance. A distinct feature of HTTP/2 is its support for multiple outstanding requests on one TCP connection, as Figure 2 shows. HTTP/2 encapsulates HTTP transactions into streams, such that a stream carries one or more HTTP transactions sequentially, and multiple streams are multiplexed over one TCP connection. Because the number of concurrent connections is reduced from many to one, bootstrapping overheads of short-lived TCP connections in HTTP/1.1 are significantly reduced, leading to more packed traffic on the multiplexed connection. HTTP/2 also supports request prioritization, header compression, and server push.

It’s worth mentioning that there are two ways to deploy SPDY: directly connecting to a SPDY server, or using a SPDY proxy. In the former scenario, the client usually establishes one connection for each domain. Many sites today employ a large number of domains with many domains pointing to the same IP address (for example, d1.cnn.com and d2.cnn.com). Known as hostname sharding, this makes SPDY behave similarly to HTTP, and is discouraged by the SPDY best practice. In contrast, when a SPDY proxy is used, the browser opens one TCP connection and reuses it across multiple domains. This is a popular and ideal way to use SPDY (for example, configurable in Chrome and used by the Amazon Silk browser). A similar disparity exists in HTTP/2.

SPDY (and HTTP/2) is not without limitations. For example, one known issue is that its performance degrades under conditions of noncongestion packet loss due to the use of a single TCP connection, which aggressively slows down the sending rate upon a loss. In contrast, in HTTP/1.1, a packet loss only affects one of the parallel connections, and the performance of other connections remains unaffected. Furthermore, the connection-level in-order delivery guarantee provided by TCP is too strict for HTTP/2 where only a stream-level in-order delivery guarantee is sufficient. This might cause head-of-line blocking when, for example, a packet loss in one stream prevents data belonging to another stream from being delivered to the upper layer, as Figure 3 shows.

Despite the previously discussed limitations, in wired networks, SPDY has been shown to exceed HTTP/1.1 in most cases with only a few exceptions such as in high packet loss environments. Because cellular networks usually have low noncongestion loss rates, in theory SPDY (and therefore HTTP/2) should also be a winner in the mobile world. However, a recent measurement revealed that in cellular networks, SPDY provides little performance boost, and sometimes might even underperform HTTP/1.1. This counterintuitive observation is attributed to the complex interaction between TCP and the cellular radio layer. More specifically, in cellular networks, when the radio interface state is changed from idle mode to active mode (as triggered by a packet to be transmitted to the base station), it incurs a latency that can last for up to 2 seconds. During this period, which is called the state promotion delay.
Beyond HTTP/2

It might be too early to anticipate what will happen beyond HTTP/2, but researchers have already started working on this issue. Among many proposals, it’s worth highlighting the Quick UDP Internet Connections (QUIC) protocol, which Google proposed recently. QUIC has already been deployed at some Google servers.

Similar to HTTP/2, QUIC also multiplexes objects into a single transport connection. However, the most notable feature of QUIC is that it works above UDP instead of TCP, thus eliminating the aforementioned head-of-line blocking issue that is a side effect of TCP’s connection-level ordering. Because UDP has no built-in congestion control (CC), QUIC implements a flexible CC framework into which various CC algorithms can be plugged. QUIC’s currently implemented CC employs a loss recovery mechanism that’s more aggressive than that of the default TCP, thus mitigating the impact of loss on multiplexing. Besides overcoming various limitations in SPDY and HTTP/2, QUIC also introduces several new features. For example, it supports zero-round-trip-time connection setup when the client revisits a server (in contrast, TCP’s conventional three-way handshake always takes one round trip); can optionally use forward error correction to better handle losses by adding redundancy to its data transmission; and provides better support for encryption and multipath, which is particularly attractive for mobile devices with multiple network interfaces (for example, Wi-Fi and cellular).

Early measurement using synthetic webpages shows that QUIC outperforms SPDY in many scenarios. However, because QUIC is still experimental, its performance for mobile Web is unclear, and some of its features are potentially not mobile friendly. For example, enabling forward error correction in QUIC consumes up to one-third of available bandwidth even when there is no loss.

HTTPS

TLS is the de facto protocol for securing a TCP connection. Using TLS to transfer data involves two phases: handshake and data transmission. In the handshake phase, TLS uses the Public Key Infrastructure to authenticate the server and to negotiate a symmetric session key, which is subsequently used in the data transmission phase for encryption and decryption. The use of HTTP over TLS (or its predecessor, SSL) is referred to as HTTPS (HTTP Secure). Historically, HTTPS was primarily used by Web services involving exchanging sensitive data (for example, a financial transaction). But it’s getting increasingly popular, exhibiting a potential trend of HTTPS everywhere. A recent measurement reports that as of 2014, more than 25 percent of server IPs accept HTTPS, which accounts for 50 percent of all HTTP connections. Today, even services such as YouTube use HTTPS.

HTTPS’ cryptographic operations incur little energy cost on mobile devices. However, the overheads introduced by the handshake phase are not negligible. First, a full TLS handshake takes at least two round trips. Assuming the average round-trip time in LTE is 70 milliseconds, which translates to 140 ms for a full handshake. Second, the bandwidth consumption of a TLS handshake is not trivial. A TLS handshake consumes on average 4.4 Kbytes of data. This might sound small for a single handshake, but when hundreds of connections are used to load a page in HTTP/1.1 (even SPDY might issue a large number of connections due to domain sharding as mentioned before), the overall penalty could be considerable. As a result, when loading mobile sites using a warm cache, the average bandwidth overhead of TLS is as high as 34 percent.

Two strategies can be leveraged to mitigate the negative impacts incurred by TLS. First, content providers should make fewer HTTPS sessions by, for example, upgrading to HTTP/2 and reducing the number of distinct domains when possible. Doing so facilitates TLS session reuse and mitigates the impact of domain sharding. Second, a Web server should be configured to support TLS Session Identifier or Session Ticket. These would allow lightweight TLS handshakes when the same client connects to the server within a certain time window since its last visit.

So far we have discussed how various aspects at different layers affect mobile-friendly Web browsing. At a high level, we see that achieving mobile-friendly Web browsing is much more than merely tailoring websites’ appearance for mobile device screens. It instead requires optimizations on webpage content, Web protocols, transport layers, and...
wireless technology. Fueled by joint efforts of all entities in the mobile ecosystem, including content providers, Web browser developers, operating system vendors, and mobile device manufacturers, mobile Web will achieve good performance, a small energy footprint, and low bandwidth consumption.

Acknowledgments
I thank Yih-Farn Robin Chen from AT&T Labs—Research for his valuable feedback.

References
2. Z. Wang et al., “How Far Can Client-Only Solutions Go for Mobile Browser Speed?”

Feng Qian is an assistant professor in the Computer Science Department, School of Informatics and Computing at the Indiana University Bloomington. His research interests include computer networking, mobile systems, network measurement, and energy efficiency. Qian has a PhD in computer science and engineering from the University of Michigan. Contact him at fengqian@indiana.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.