


≡

High Performance Browser Networking | O'Reilly

# Primer on Browser Networking

BROWSER APIS AND PROTOCOLS, CHAPTER 14



A modern browser is a platform specifically designed for fast, efficient, and secure delivery of web applications. In fact, under the hood, a modern browser is an entire operating system with hundreds of components: process management, security sandboxes, layers of optimization caches, JavaScript VMs, graphics rendering and GPU pipelines, storage, sensors, audio and video, networking, and much more.

Not surprisingly, the overall performance of the browser, and any application that it runs, is determined by a number of components: parsing, layout, style calculation of HTML and CSS, JavaScript execution speed, rendering pipelines, and of course the networking stack. Each component plays a critical role, but networking often doubly so, since if the browser is blocked on the network, waiting for the resources to arrive, then all other steps are blocked!

As a result, it is not surprising to discover that the networking stack of a modern browser is much more than a simple socket manager. From the outside, it may present itself as a simple resource-fetching mechanism, but from the inside it is its own platform (Figure 14-1), with its own optimization criteria, APIs, and services.

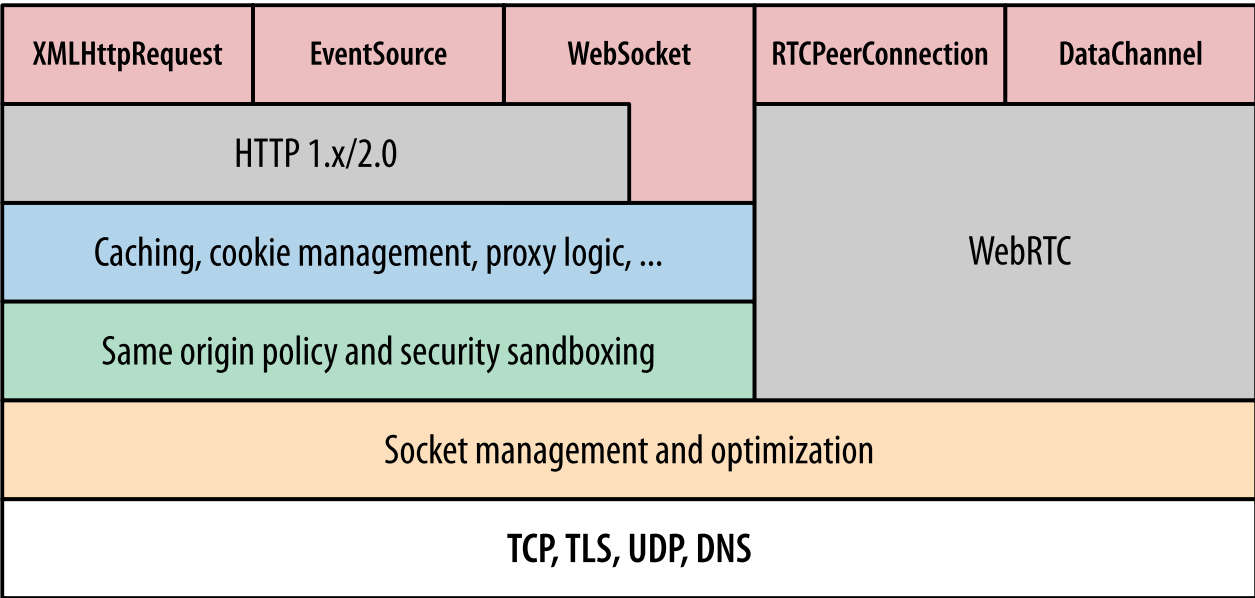


Figure 14-1. High-level browser networking APIs, protocols, and services

When designing a web application, we don't have to worry about the individual TCP or UDP sockets; the browser manages that for us. Further, the network stack takes care of imposing the right connection limits, formatting our requests, sandboxing individual applications from one



removed, our applications can focus on the application logic.

However, out of sight does not mean out of mind! As we saw, understanding the performance characteristics of TCP, HTTP, and mobile networks can help us build faster applications. Similarly, understanding how to optimize for the various browser networking APIs, protocols, and services can make a dramatic difference in performance of any application.

## Connection Management and Optimization



Web applications running in the browser do not manage the lifecycle of individual network sockets, and that's a good thing. By deferring this work to the browser, we allow it to automate a number of critical performance optimizations, such as socket reuse, request prioritization and late binding, protocol negotiation, enforcing connection limits, and much more. In fact, the browser intentionally separates the *request management* lifecycle from *socket management*. This is a subtle but critical distinction.

Sockets are organized in pools (Figure 14-2), which are grouped by origin, and each pool enforces its own connection limits and security constraints. Pending requests are queued, prioritized, and then bound to individual sockets in the pool. Consequently, unless the server intentionally closes the connection, the same socket can be automatically reused across multiple requests!

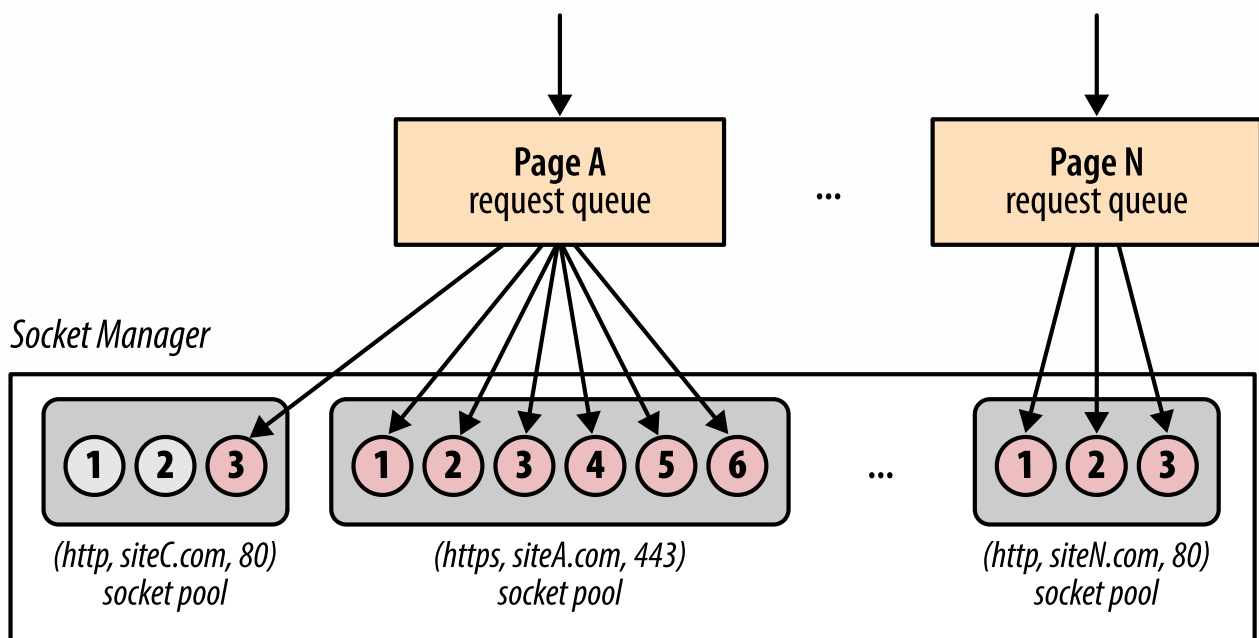


Figure 14-2. Auto-managed socket pools are shared among all browser processes

### Origin

A triple of application protocol, domain name, and port number—e.g., (http, www.example.com, 80) and (https, www.example.com, 443) are considered as different origins.

### Socket pool



Automatic socket pooling automates TCP connection reuse, which offers significant performance benefits; see [Benefits of Keepalive Connections](#). However, that's not all. This architecture also enables a number of additional optimization opportunities:

- The browser can service queued requests in priority order.
- The browser can reuse sockets to minimize latency and improve throughput.
- The browser can be proactive in opening sockets in anticipation of request.
- The browser can optimize when idle sockets are closed.
- The browser can optimize bandwidth allocation across all sockets.

In short, the browser networking stack is our strategic ally on our quest to deliver high-performance applications. None of the functionality we have covered requires any work on our behalf! However, that's not to say that we can't help the browser. Design decisions in our applications that determine the network communication patterns, type and frequency of transfers, choice of protocols, and tuning and optimization of our server stacks play critical roles in the end performance of every application.

### Speculative Networking Optimization in Google Chrome



We have already established that the network stack of a modern browser is much more than a simple socket manager. However, even that does not do full justice to some of the optimization techniques performed by modern browsers.

For example, Google Chrome browser gets faster as you use it. Chrome learns the topology of visited sites and typical browsing patterns and then leverages this information to perform a variety of "speculative optimizations" designed to anticipate likely user actions and eliminate unnecessary network latency: DNS pre-resolving, TCP pre-connect, page pre-rendering, and more. Simple actions, such as a mouse hover over a link can trigger a signal from the browser to the network stack "predictor," which may then select the best optimization based on past performance data!

For more details, see our earlier discussion on [Browser Optimization](#). And if you're curious to learn more about Chrome's networking optimization, check out "[High Performance Networking in Google Chrome](#)" .

## Network Security and Sandboxing



Deferring management of individual sockets serves another important purpose: it allows the browser to sandbox and enforce a consistent set of security and policy constraints on untrusted application code. For example, the browser does not permit direct API access to raw network sockets, as that would allow a malicious application to initiate arbitrary connections to any host



### Connection limits

Browser manages all open socket pools and enforces connection limits to protect both the client and server from resource exhaustion.

### Request formatting and response processing

Browser formats all outgoing requests to enforce consistent and well-formed protocol semantics to protect the server. Similarly, response decoding is done automatically to protect the user from malicious servers.

### TLS negotiation

Browser performs the TLS handshake and performs the necessary verification checks of the certificates. The user is warned when and if any of the verification fails—e.g., server is using a self-signed certificate.

### Same-origin policy

Browser enforces constraints on which requests can be initiated by the application and to which origin.

The previous list is not complete, but it highlights the principle of "least privilege" at work. The browser exposes only the APIs and resources that are necessary for the application code: the application supplies the data and URL, and the browser formats the request and handles the full lifecycle of each connection.

#### Note

*It is worth noting that there is no single "same-origin policy." Instead, there is a set of related mechanisms that enforce restrictions on DOM access, cookie and session state management, networking, and other components of the browser.*

*A full discussion on browser security requires its own separate book. If you are curious, Michal Zalewski's *The Tangled Web: A Guide to Securing Modern Web Applications* is a fantastic resource.*

## Resource and Client State Caching



The best and fastest request is a request not made. Prior to dispatching a request, the browser automatically checks its resource cache, performs the necessary validation checks, and returns a local copy of the resources if the specified constraints are satisfied. Similarly, if a local resource is not available in cache, then a network request is made and the response is automatically placed in cache for subsequent access if permitted.

- The browser automatically evaluates caching directives on each resource.

- The browser automatically manages the size of cache and resource eviction.

Managing an efficient and optimized resource cache is hard. Thankfully, the browser takes care of all of the complexity on our behalf, and all we need to do is ensure that our servers are returning the appropriate cache directives; see [Cache Resources on the Client](#). You are providing Cache-Control, ETag, and Last-Modified response headers for all the resources on your pages, right?

Finally, an often-overlooked but critical function of the browser is to provide authentication, session, and cookie management. The browser maintains separate "cookie jars" for each origin, provides necessary application and server APIs to read and write new cookie, session, and authentication data and automatically appends and processes appropriate HTTP headers to automate the entire process on our behalf.

Note

*A simple but illustrative example of the convenience of deferring session state management to the browser: an authenticated session can be shared across multiple tabs or browser windows, and vice versa; a sign-out action in a single tab will invalidate open sessions in all other open windows.*

Application APIs and Protocols

§

Walking up the ladder of provided network services we finally arrive at the application APIs and protocols. As we saw, the lower layers provide a wide array of critical services: socket and connection management, request and response processing, enforcement of various security policies, caching, and much more. Every time we initiate an HTTP or an XMLHttpRequest, a long-lived Server-Sent Events or WebSocket session, or open a WebRTC connection, we are interacting with some or all of these underlying services.

There is no one best protocol or API. Every nontrivial application will require a mix of different transports based on a variety of requirements: interaction with the browser cache, protocol overhead, message latency, reliability, type of data transfer, and more. Some protocols may offer low-latency delivery (e.g., Server-Sent Events, WebSocket), but may not meet other critical criteria, such as the ability to leverage the browser cache or support efficient binary transfers in all cases.

	XMLHttpRequest	Server-Sent Events	WebSocket
Request streaming	no	no	yes
Response streaming	limited	yes	yes
Framing mechanism	HTTP	event stream	binary framing

Binary data transfers	yes	no (base64)	yes
Compression	yes	yes	limited
Application transport protocol	HTTP	HTTP	WebSocket
Network transport protocol	TCP	TCP	TCP

Table 14-1. High-level features of XHR, SSE, and WebSocket

Note

*We are intentionally omitting WebRTC from this comparison, as its peer-to-peer delivery model offers a significant departure from XHR, SSE, and WebSocket protocols.*

This comparison of high-level features is incomplete—that’s the subject of the following chapters—but serves as a good illustration of the many differences among each protocol. Understanding the pros, cons, and trade-offs of each, and matching them to the requirements of our applications, can make all the difference between a high-performance application and a consistently poor experience for the user.

[« Back to the Table of Contents](#)

Copyright © 2013 [Ilya Grigorik](#). Published by O'Reilly Media, Inc. Licensed under [CC BY-NC-ND 4.0](#).