Lecture 10
Performance

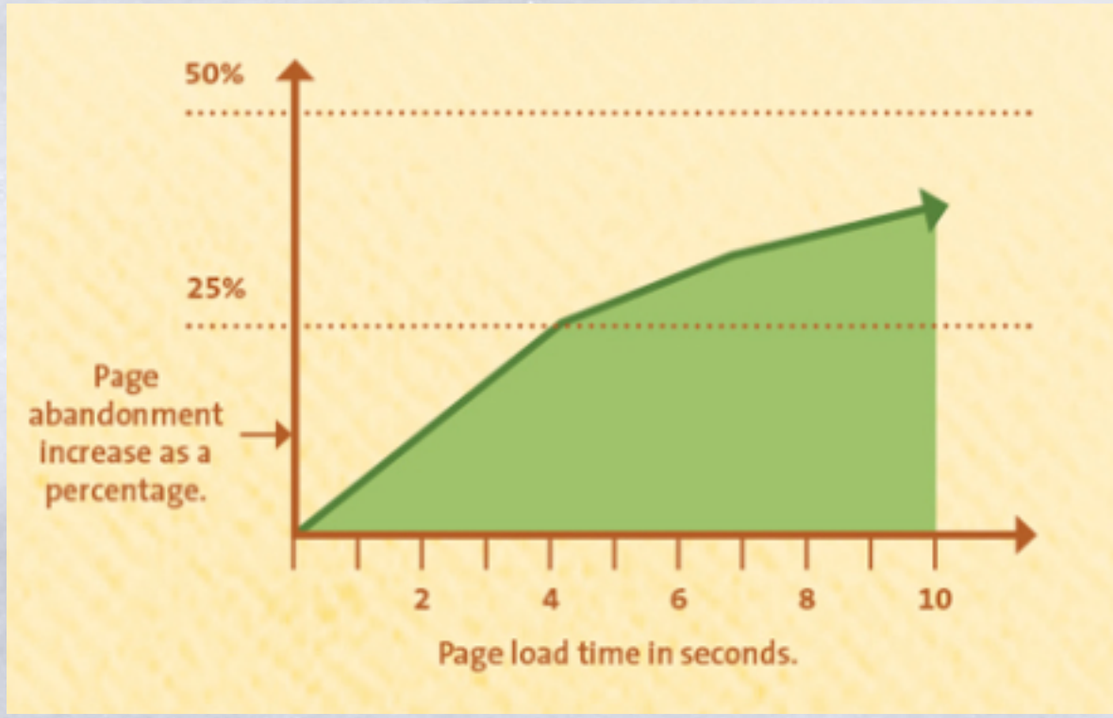
Play

What happened?

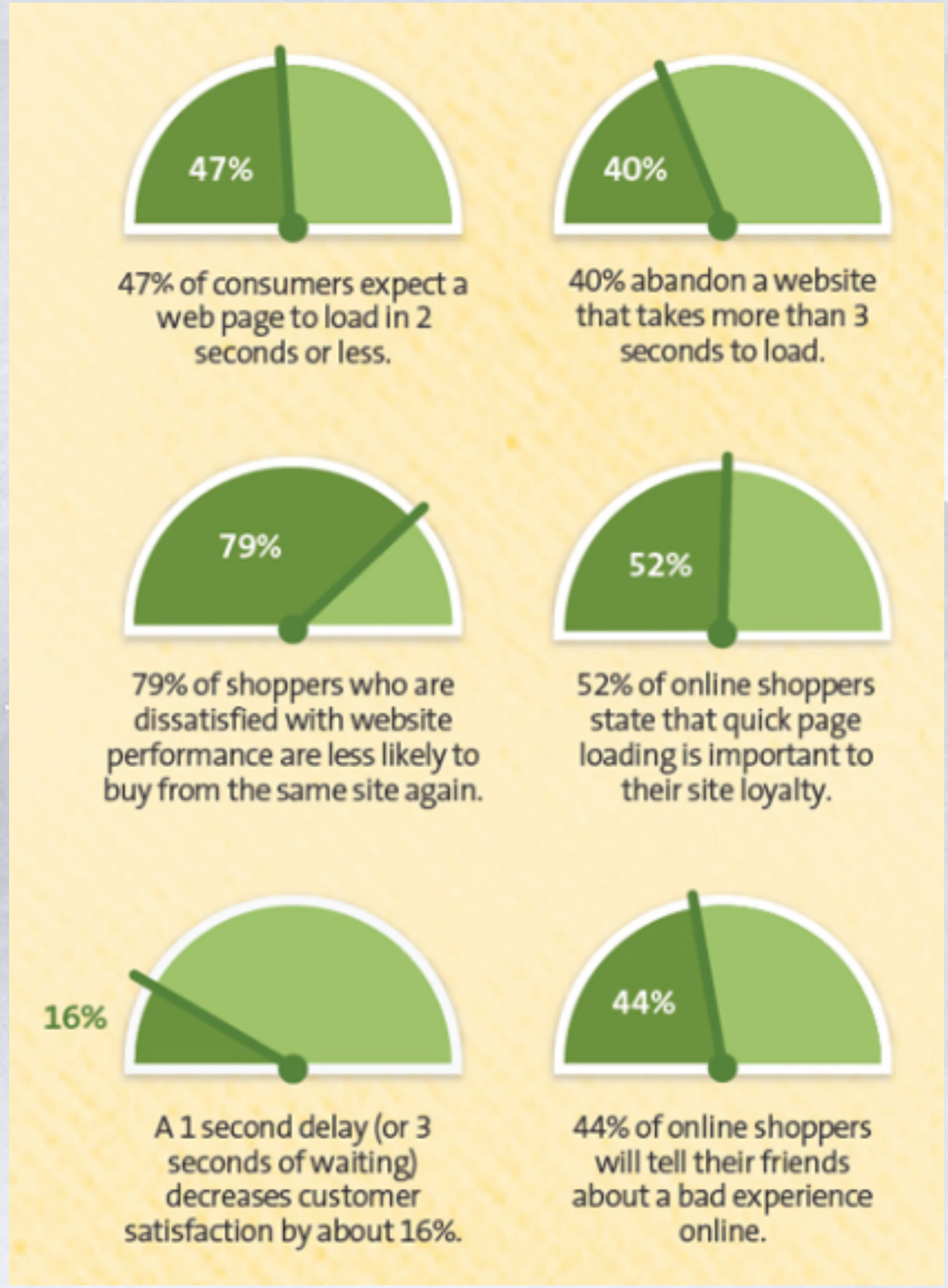
- Service is too busy

Why Website Performance Matters?

- 40% Abandon a site after waiting 3 seconds for a page to load.
- 80% of the people will NOT return
- Almost half will tell others about their negative experience.
- Amazon: 1% Revenue increase for every 100MS of improvement



How Website Performance Affects Shopping Behavior



Website Speed and SEO

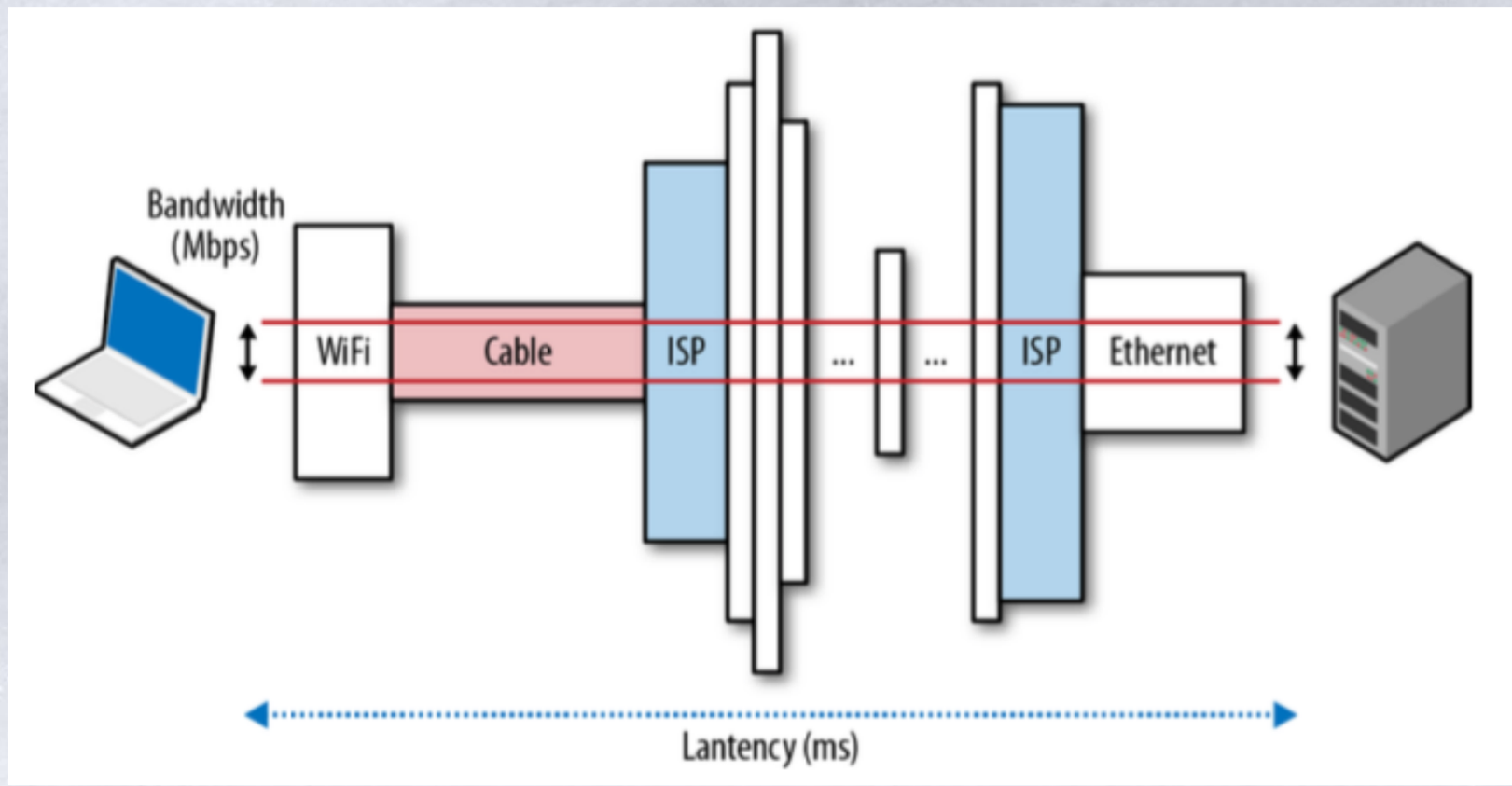
- Website speed is important to your SEO efforts because faster websites are:
  - Easier to crawl
  - Easier to access
  - More likely to rank (although this is marginal)
  - Most importantly, more likely to keep visitors around!

web performance optimization (WPO)

- Faster sites lead to better user engagement.
- Faster sites lead to better user retention.
- Faster sites lead to higher conversions.

Latency and Bandwidth

- Latency
  - The time from the source sending a packet to the destination receiving it
- Bandwidth
  - Maximum throughput of a logical or physical communication path



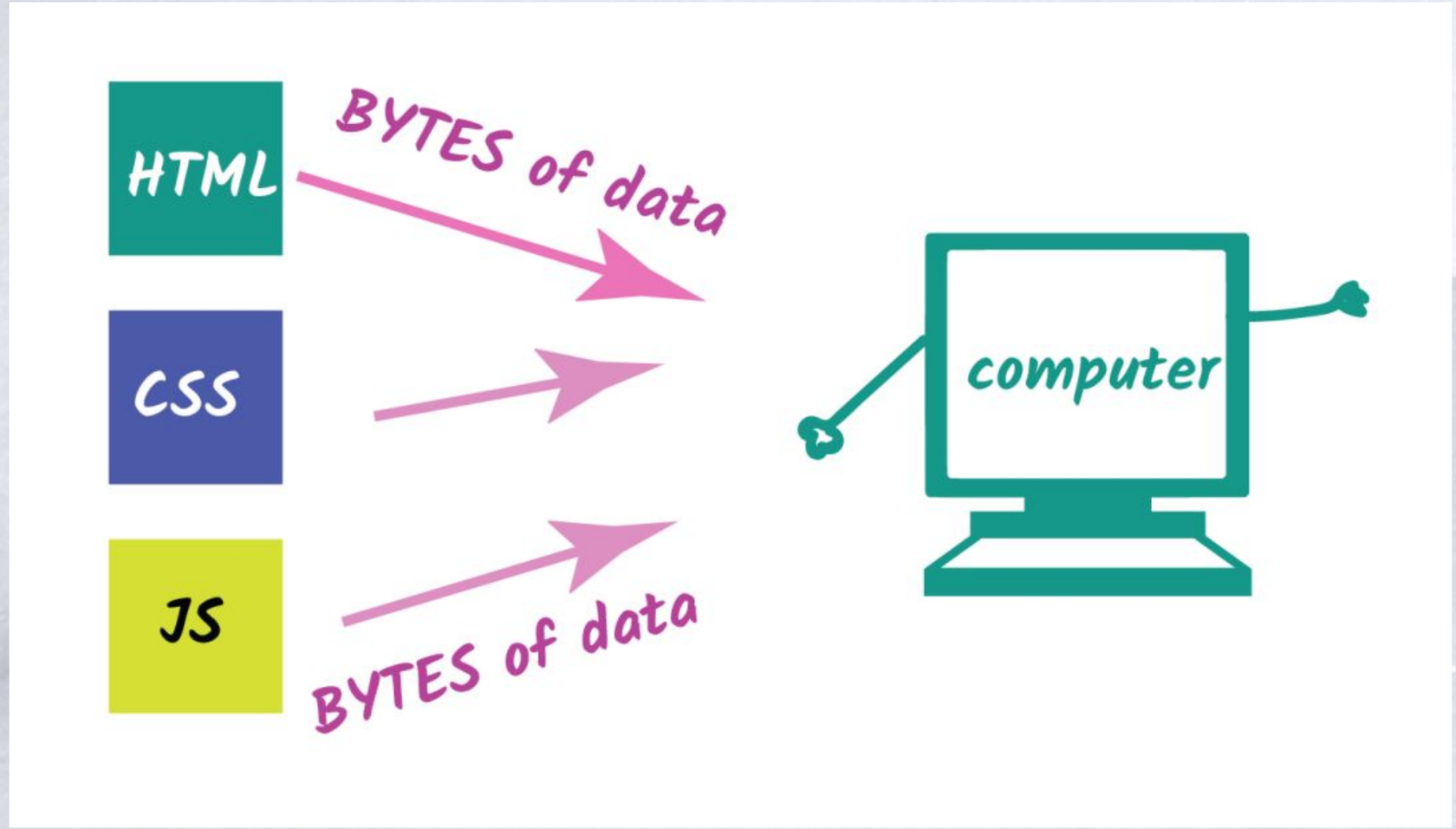
Components of Latency

- Propagation delay
  - Amount of time required for a message to travel from the sender to receiver, which is a function of distance over speed with which the signal propagates.
- Transmission delay
  - Amount of time required to push all the packet's bits into the link, which is a function of the packet's length and data rate of the link.
- Processing delay
  - Amount of time required to process the packet header, check for bit-level errors, and determine the packet's destination.
- Queuing delay
  - Amount of time the incoming packet is waiting in the queue until it can be processed.

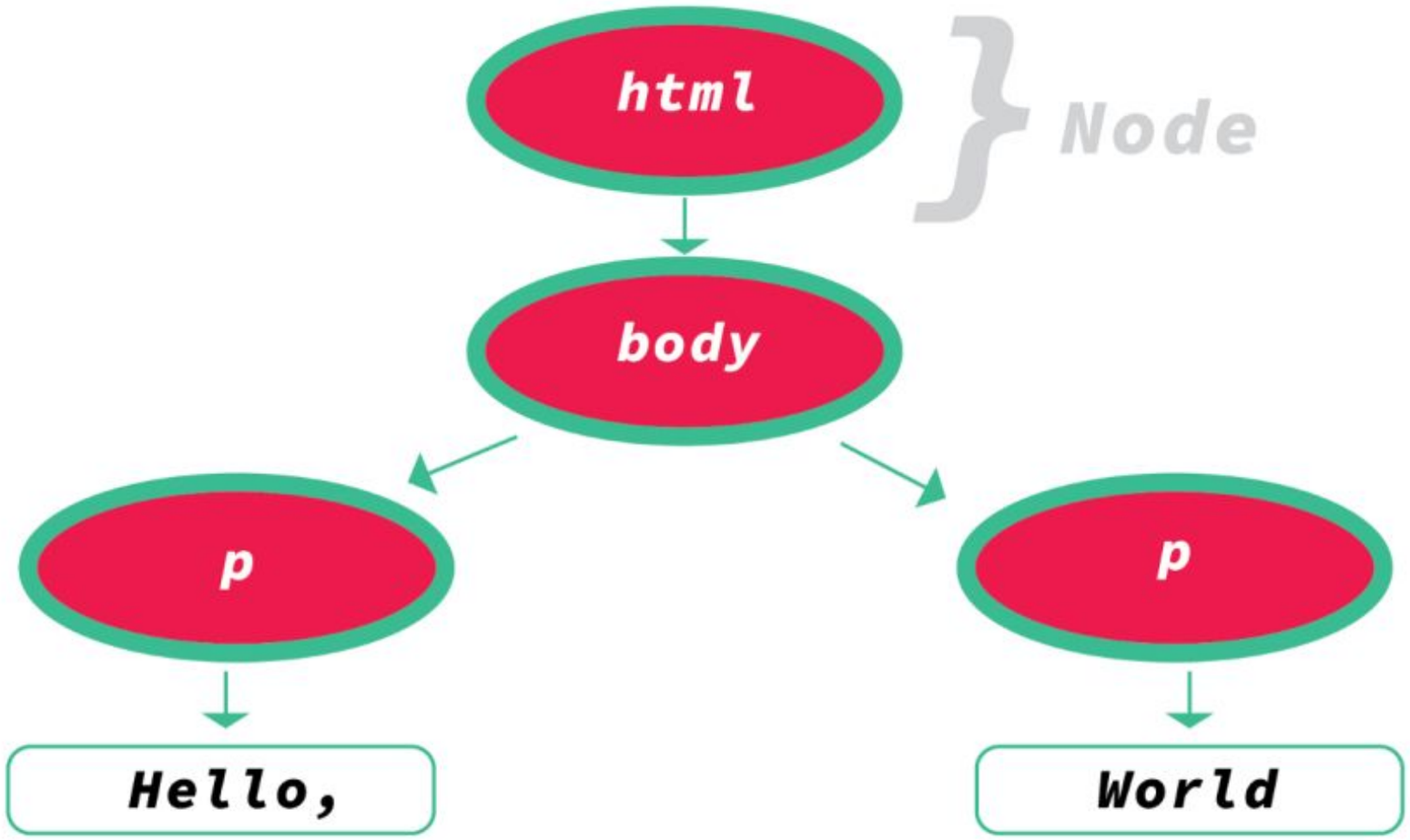
Primer on Web Performance

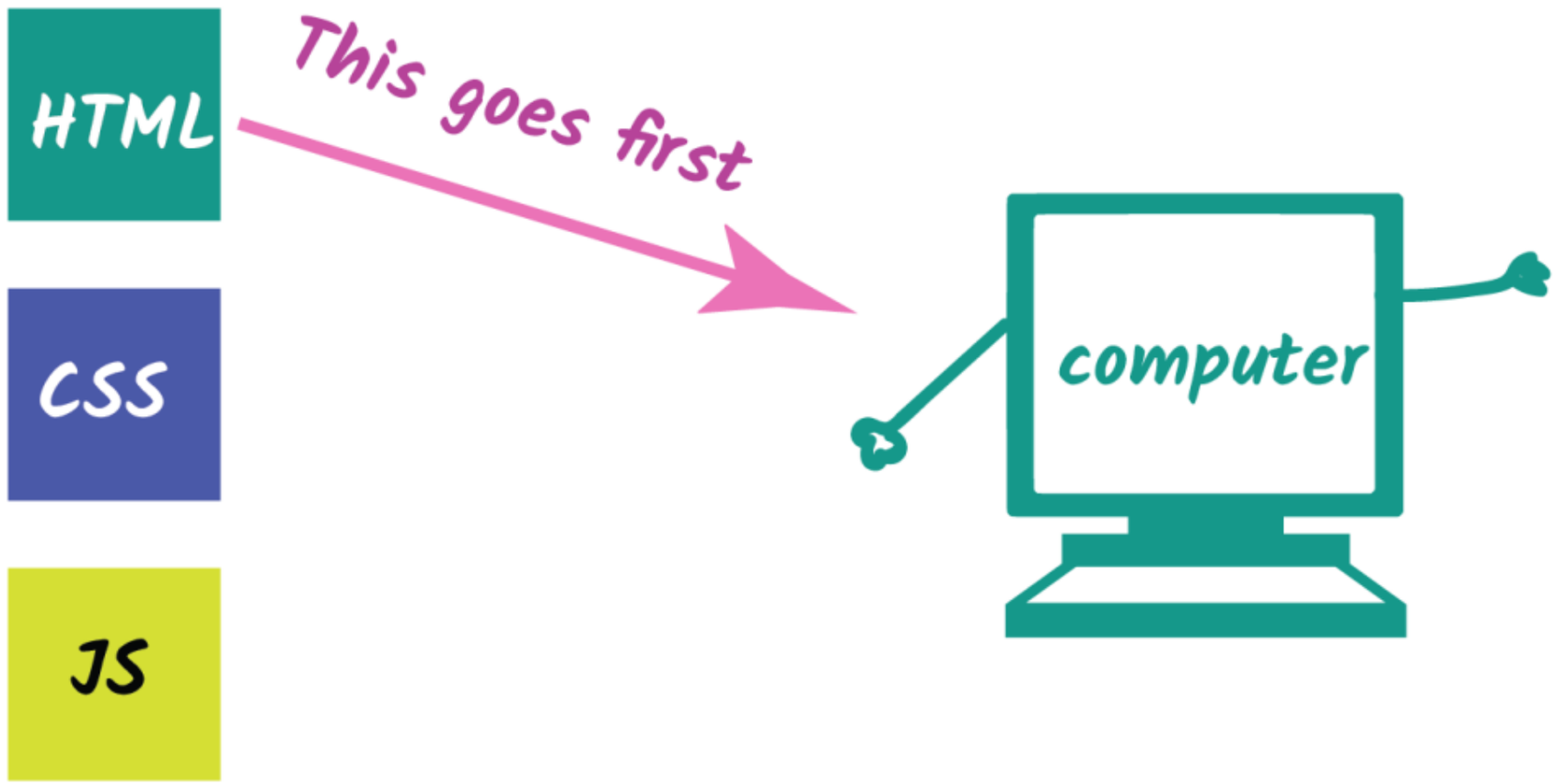
- Impact of latency and bandwidth on web performance
- Transport protocol (TCP) constraints imposed on HTTP
- Features and shortcomings of the HTTP protocol itself
- Web application trends and performance requirements
- Browser constraints and optimizations

浏览器页面渲染机制



https://blog.logrocket.com/how-browser-rendering-works-behind-the-scenes-6782b0e8fb10

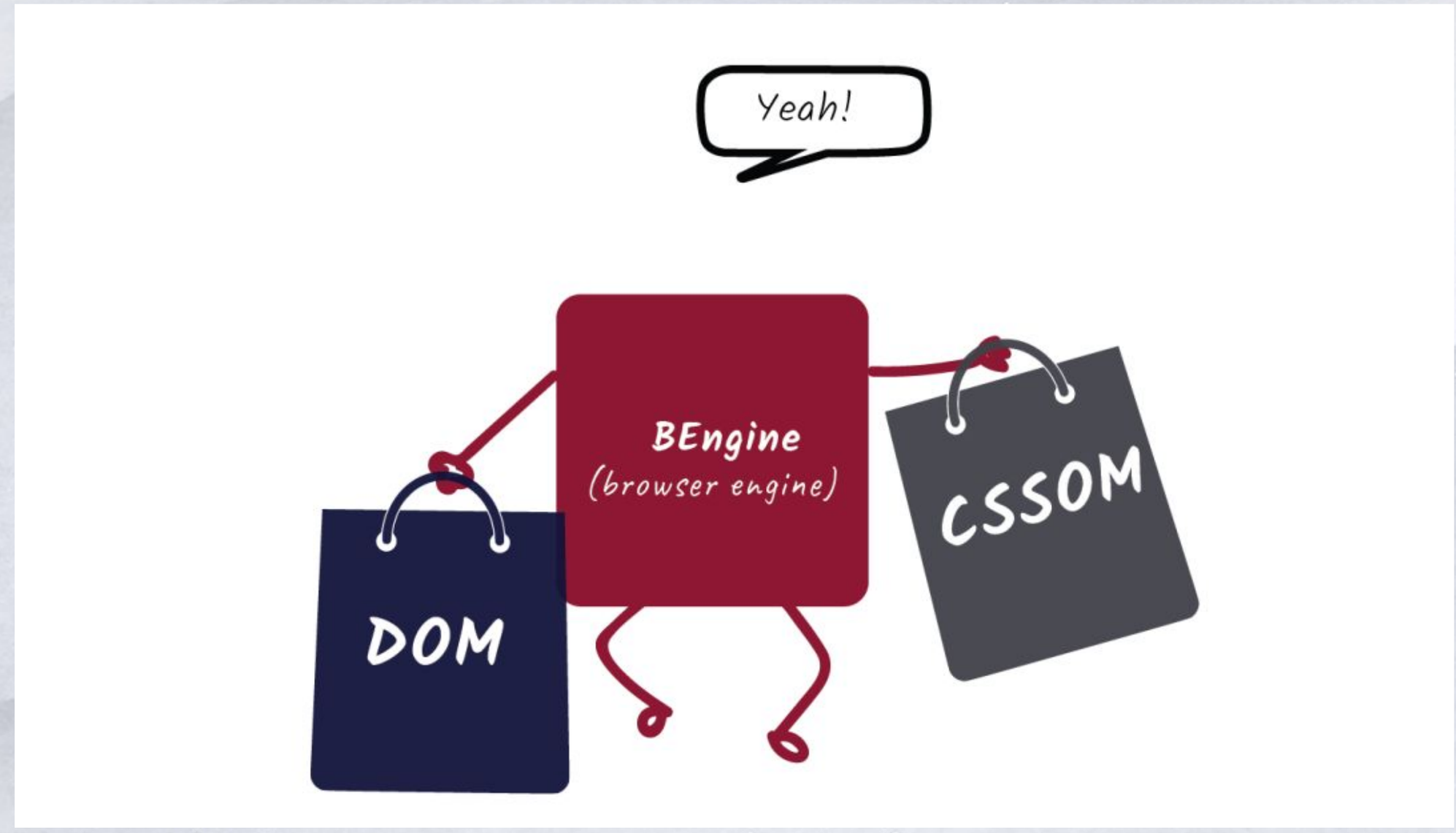

html
Node
body
p
p
Hello,
World



Bytes => Characters => Tokens => Node => DOM

从 CSS 的原始字节到 CSSOM

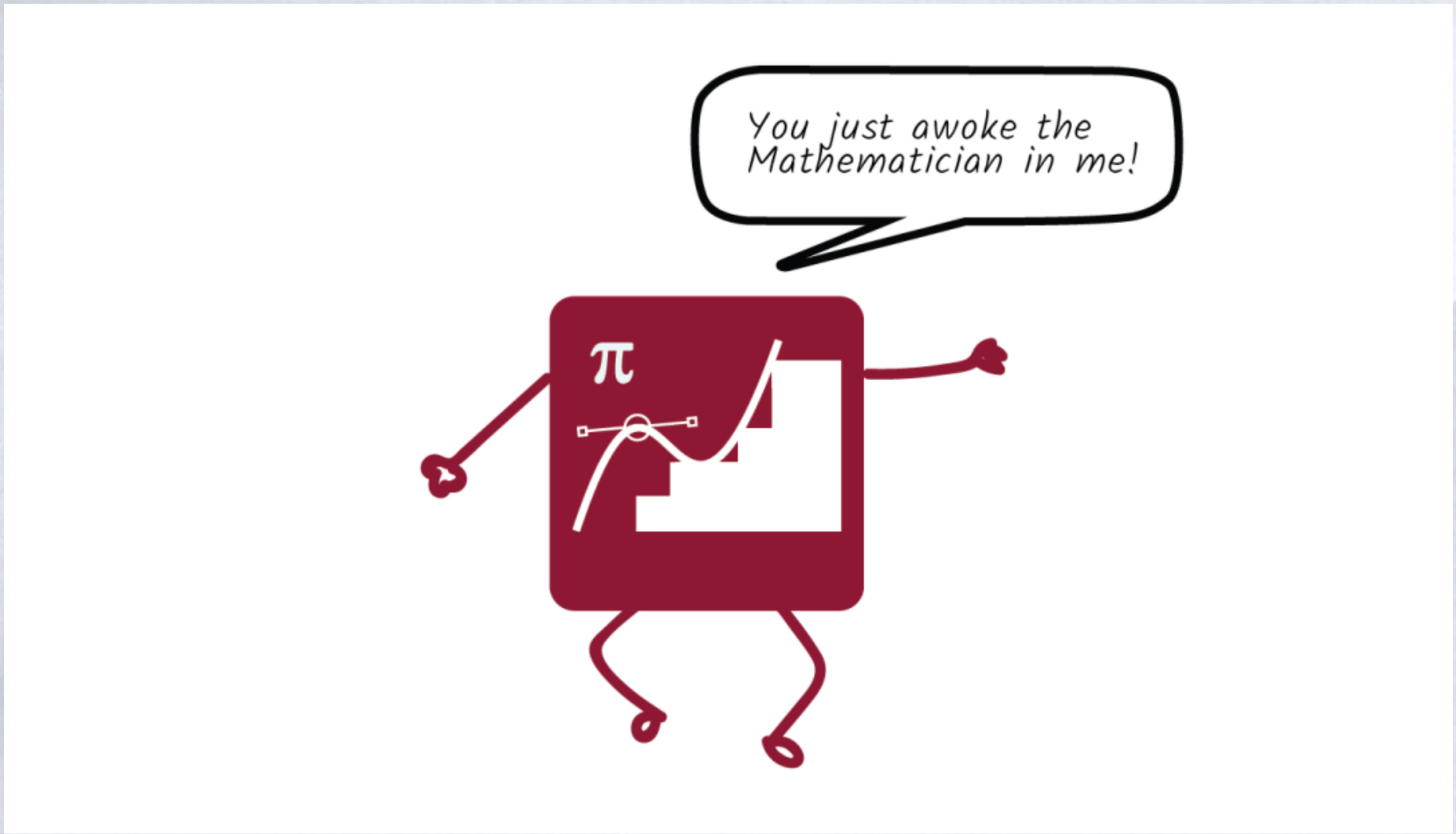
渲染树



渲染树

布局

❄ 回流（reflow）



渲染阻塞资源

- 渲染阻塞（render-blocking）
  - “有东西阻止了屏幕上节点的实际绘制”
- 应该尽快将 HTML 和 CSS 提供给客户端，以优化应用程序的首次渲染时间。

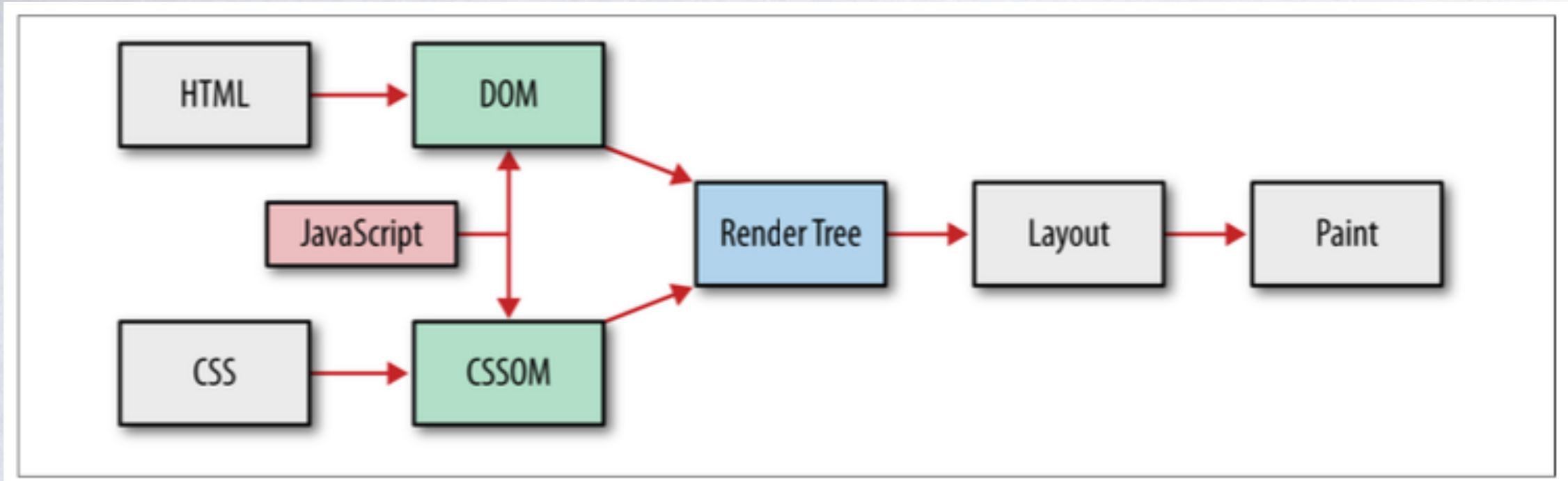
JavaScript 如何执行?

- 每当浏览器遇到脚本标签时，DOM 构造就会暂停！整个 DOM 构建过程都将停止，直到脚本执行完成。
  - JavaScript 可以同时修改 DOM 和 CSSOM

- async

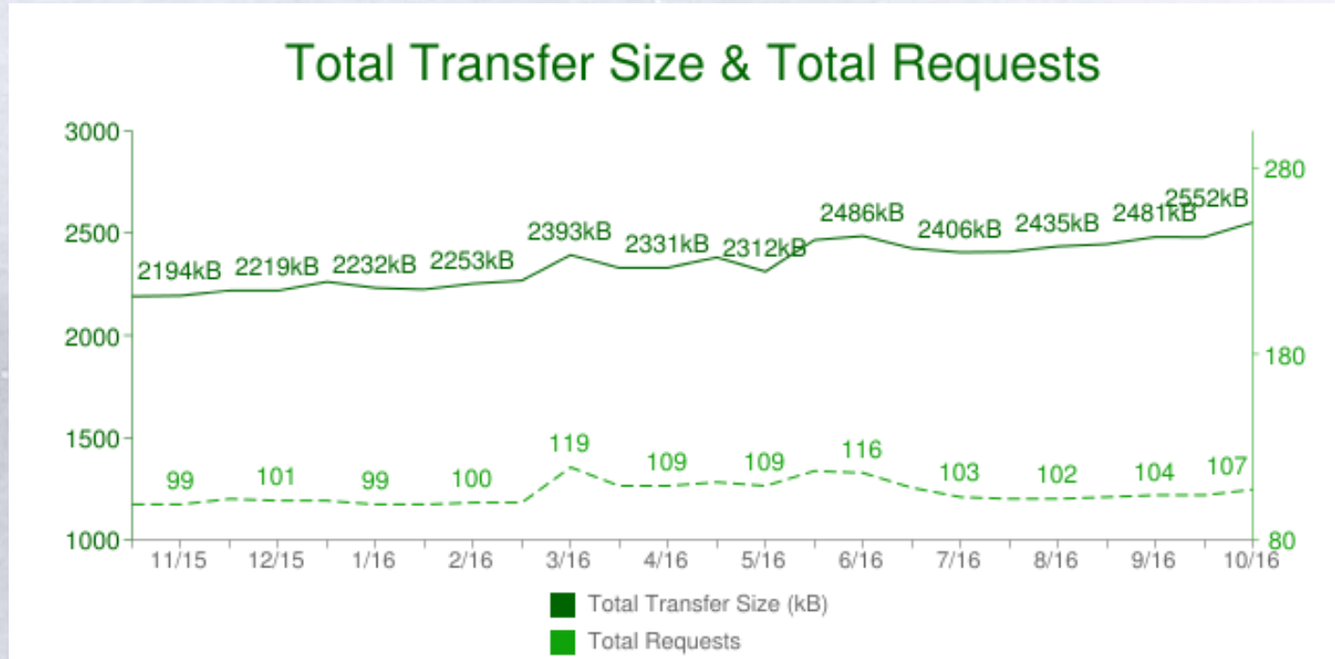
DOM, CSSOM and JavaScript

- Hypertext, Web Pages, and Web Applications



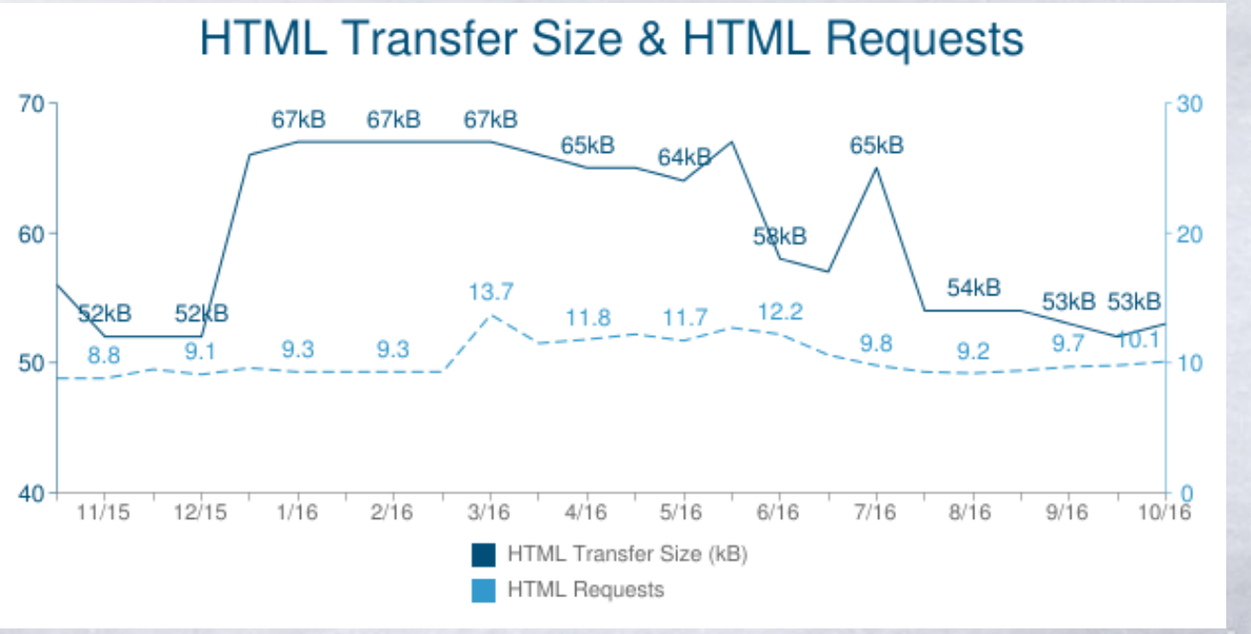
Anatomy of a Modern Web Application

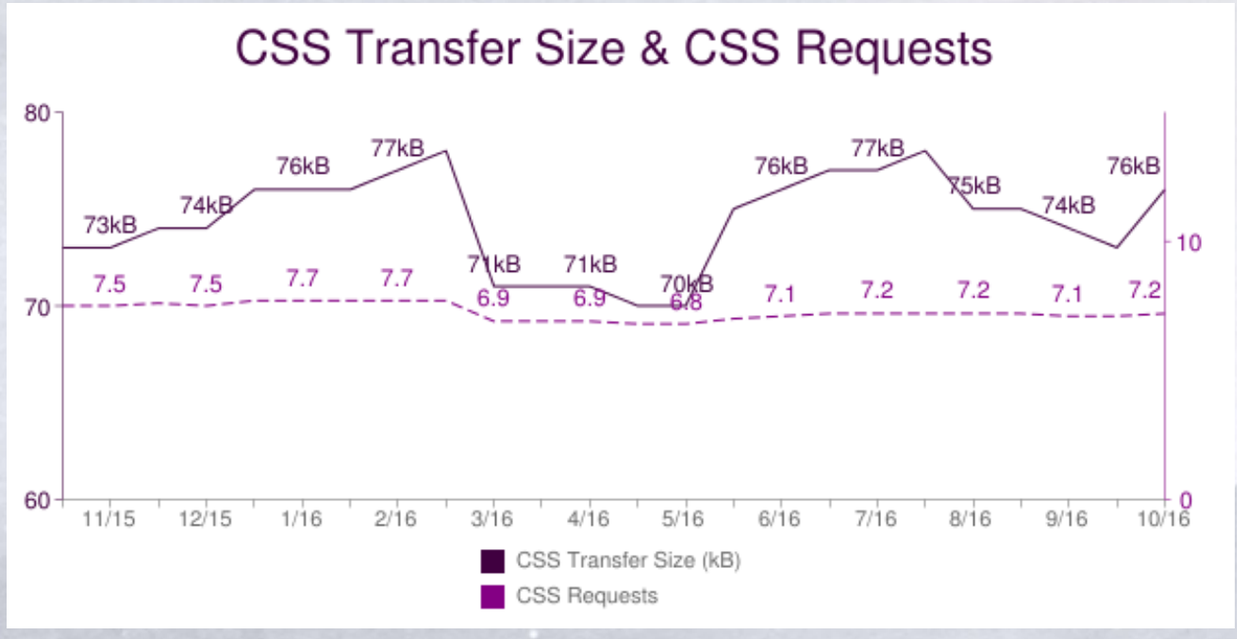
- 2013 年初，一个普通的 Web 应用由下列内容构成。
  - 90 个请求，发送到 15 个主机，总下载量 1311 KB
    - HTML:10 个请求，52 KB
    - 图片:55 个请求，812 KB
    - JavaScript:15 个请求，216 KB
    - CSS:5 个请求，36 KB
    - 其他资源:5 个请求，195 KB

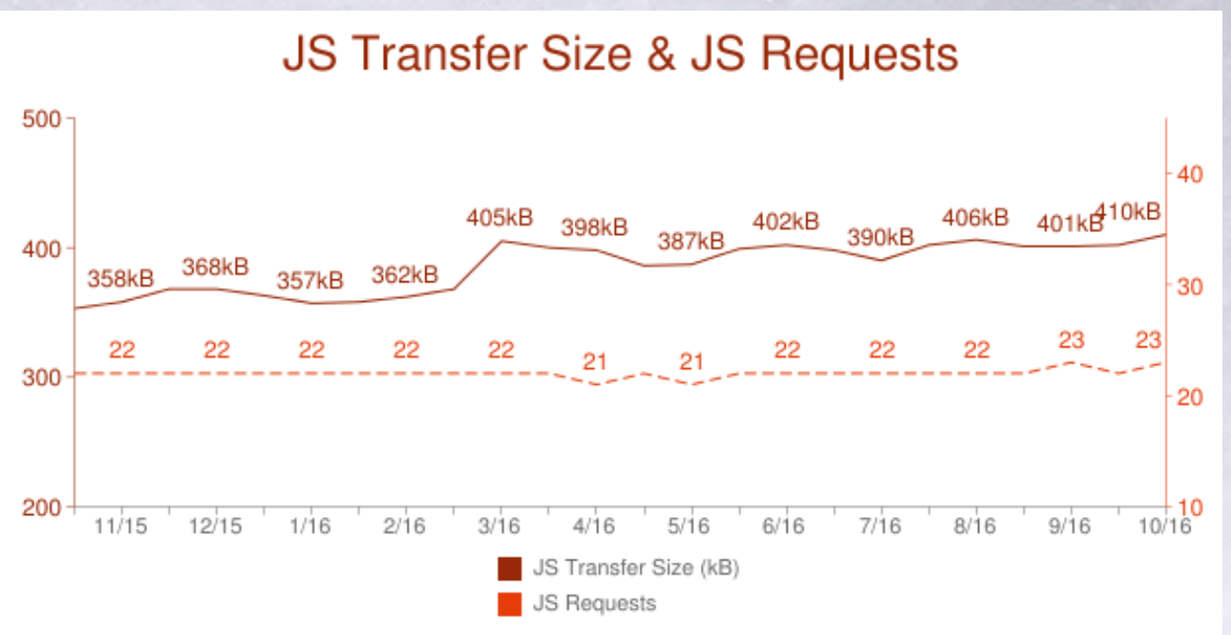


http://httparchive.org

Including


HTML Transfer Size & HTML Requests
70
60
50
40
30
20
10
0
67kB 67kB 67kB 65kB 64kB 58kB 65kB 54kB 53kB 53kB
52kB 52kB
8.8 9.1 9.3 9.3 13.7 11.8 11.7 12.2 9.8 9.2 9.7 10.1
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
HTML Transfer Size (kB)
HTML Requests


CSS Transfer Size & CSS Requests
80
70
60
10
0
73kB 74kB 76kB 77kB 71kB 71kB 70kB 76kB 77kB 75kB 74kB 76kB
7.5 7.5 7.7 7.7 6.9 6.9 6.8 7.1 7.2 7.2 7.1 7.2
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
CSS Transfer Size (kB)
CSS Requests


JS Transfer Size & JS Requests
500
400
300
200
40
30
20
10
358kB 368kB 357kB 362kB 405kB 398kB 387kB 402kB 390kB 406kB 401kB 410kB
22 22 22 22 22 21 21 22 22 22 23 23
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
JS Transfer Size (kB)
JS Requests

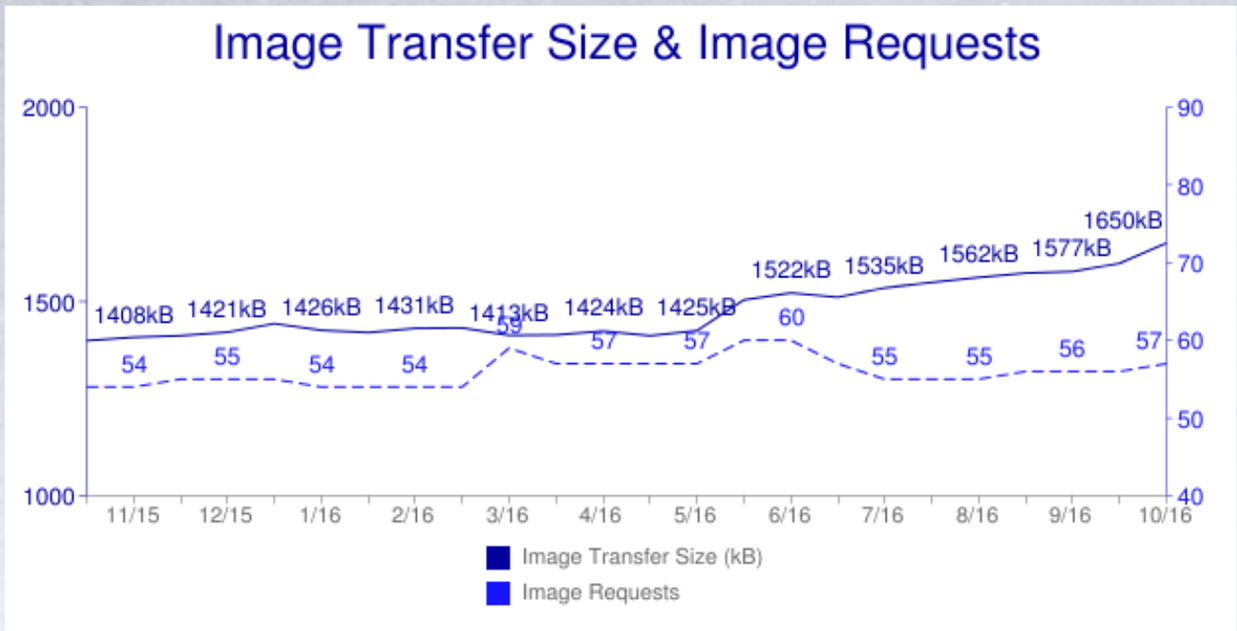
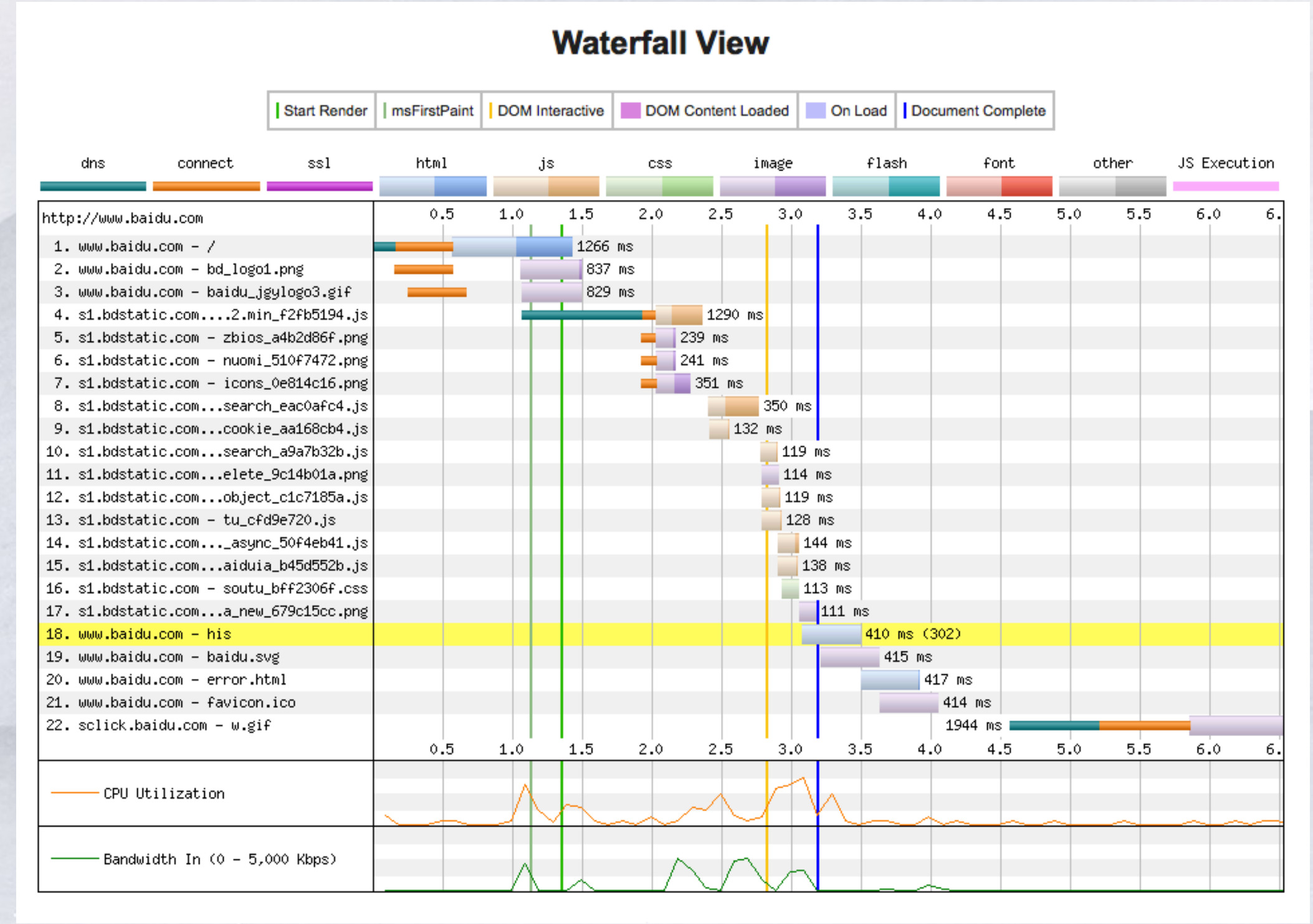


Image Transfer Size & Image Requests
2000
1500
1000
90
80
70
60
50
40
1408kB 1421kB 1426kB 1431kB 1413kB 1424kB 1425kB 1522kB 1535kB 1562kB 1577kB 1650kB
54 55 54 54 59 57 57 60 55 55 56 57
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
Image Transfer Size (kB)
Image Requests

Time and user perception

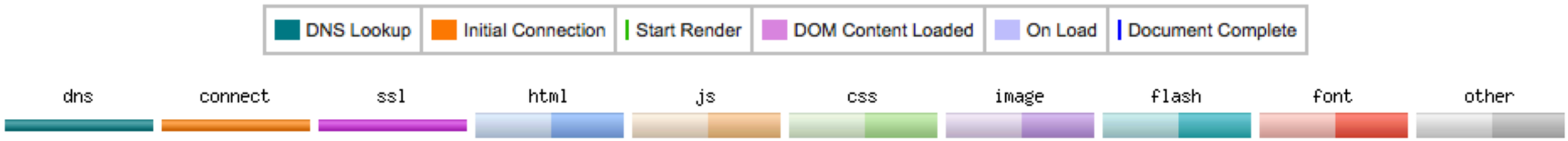
Delay	User perception
0–100 ms	Instant
100–300 ms	Small perceptible delay
300–1000 ms	Machine is working
1,000+ ms	Likely mental context switch
10,000+ ms	Task is abandoned

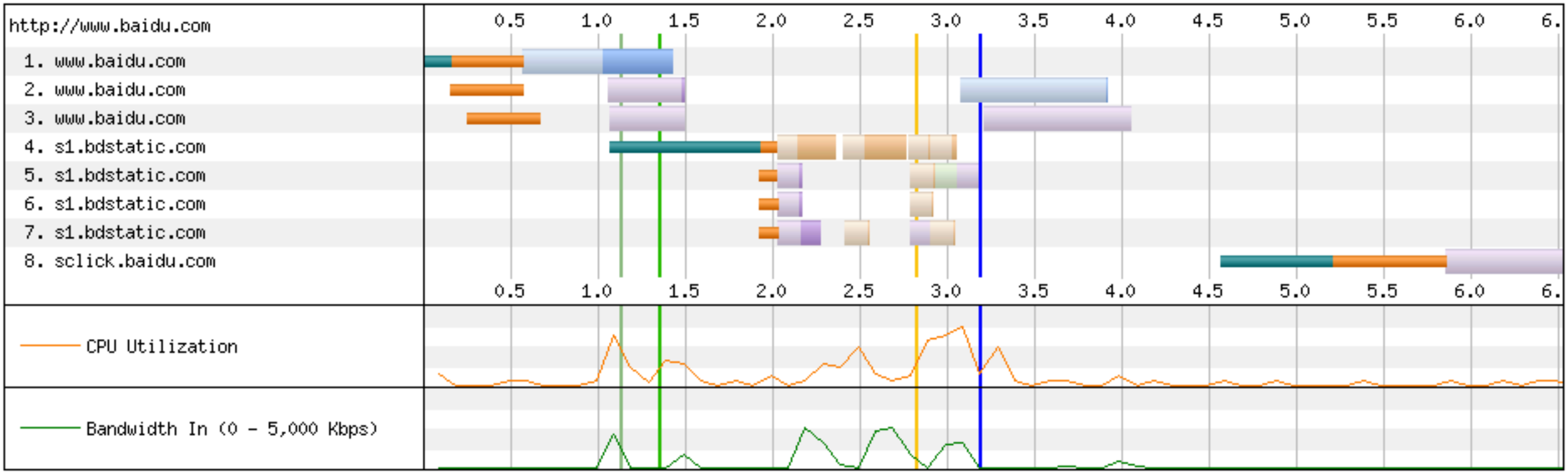
Analyzing the Resource Waterfall



http://www.webpagetest.org/

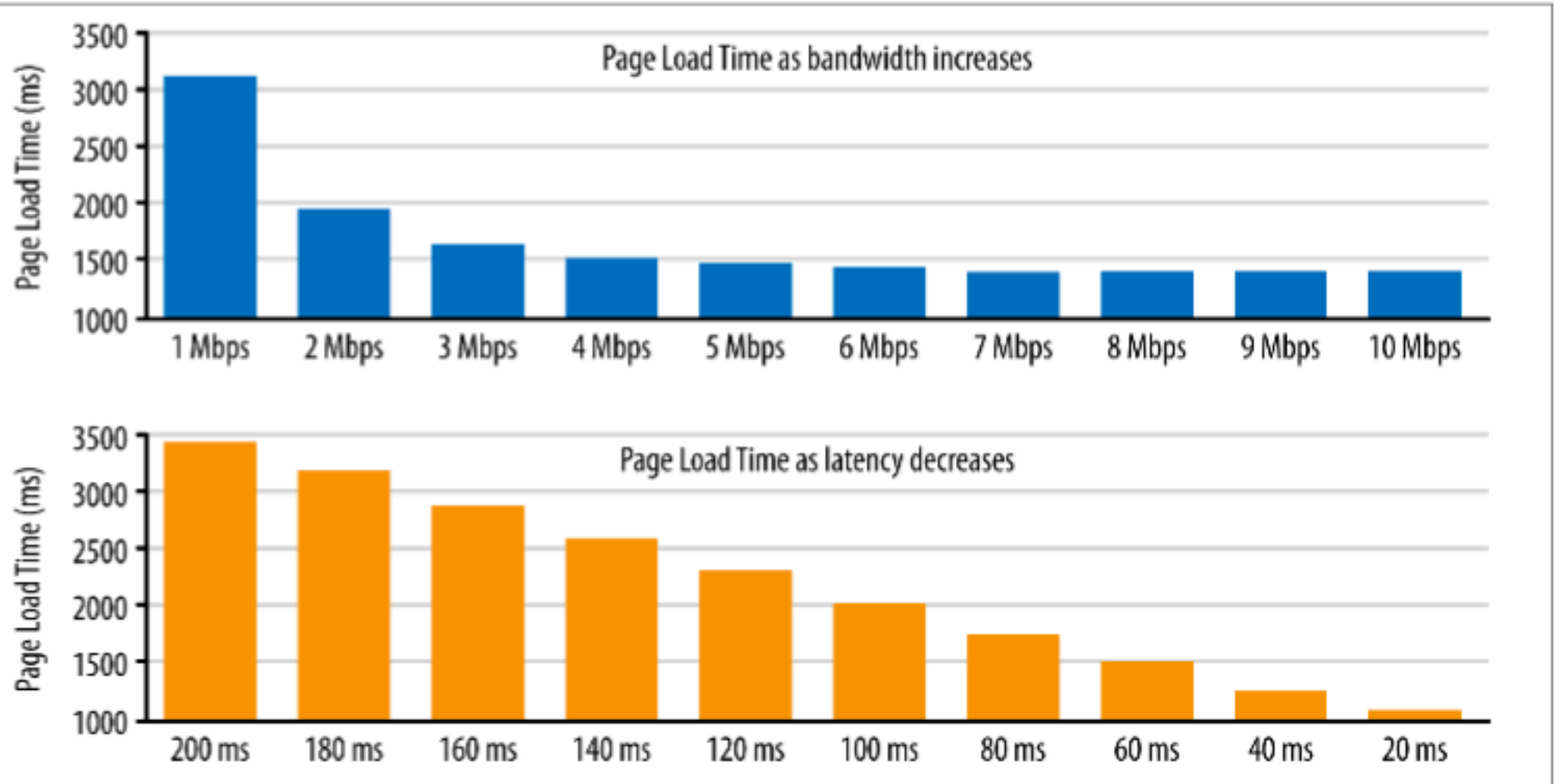
Connection View





Performance Pillars: Computing, Rendering, Networking

- The execution of a web program primarily involves three tasks: fetching resources, page layout and rendering, and JavaScript execution.
  - More Bandwidth Doesn't Matter (Much)
  - Latency as a Performance Bottleneck



Benchmarking Utilities

- Apache Bench
  - ab utility bundled with Apache
- Siege
  - http://www.joedog.org/JoeDog/Siege
- http_load (Excellent for latency tests)
  - http://www.acme.com/software/http_load/

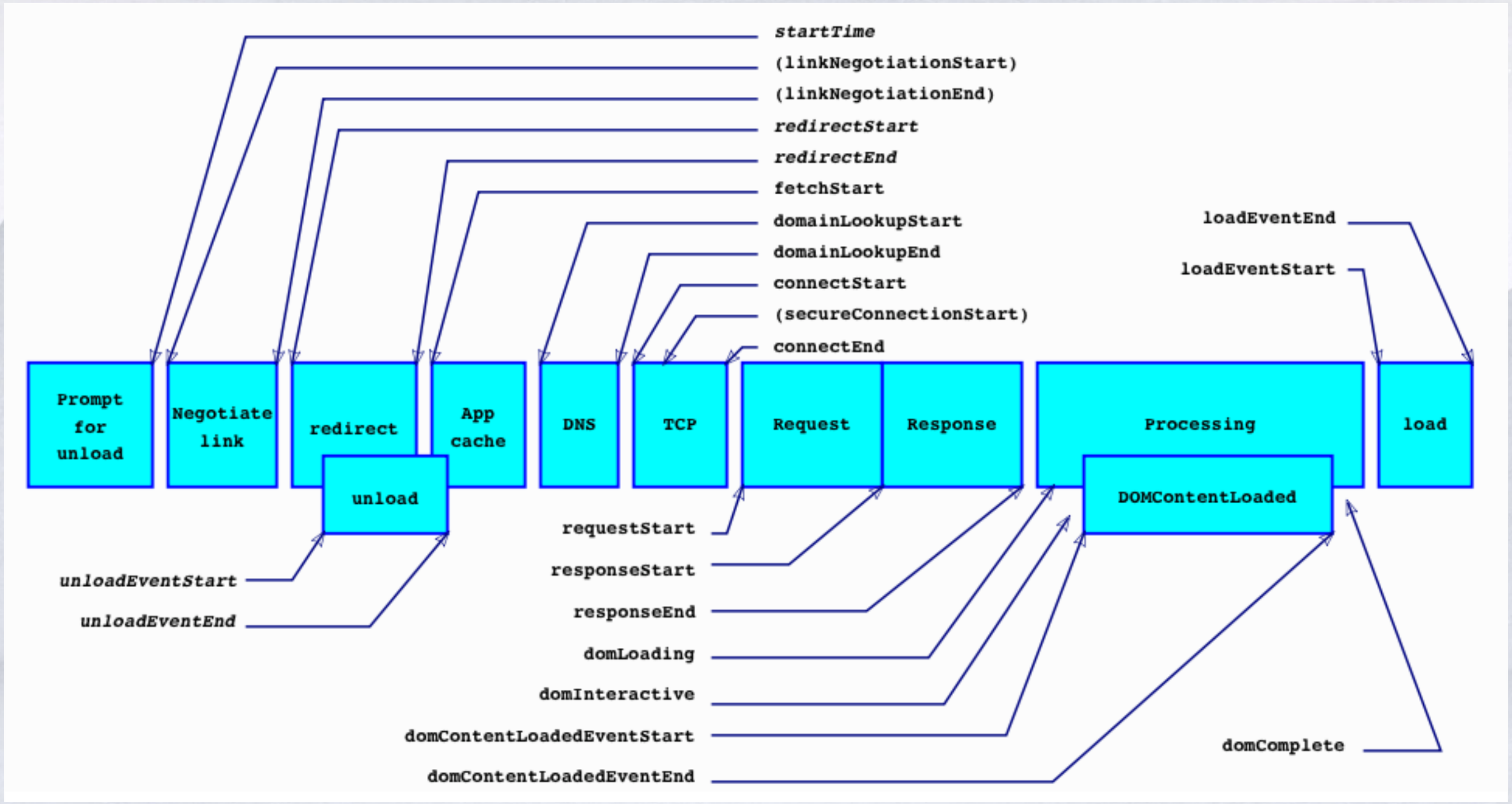
Affecting Your Benchmark Figures

- Geographical location
- Network issues
- Response size
- Code processing
- Browser behavior
- Web server configuration

问题

- 人造测试不能发现所有性能瓶颈。
  - 场景及页面选择:很难重复真实用户的导航模式;
  - 浏览器缓存:用户缓存不同，性能差别很大;
  - 中介设施:中间代理和缓存对性能影响很大;
  - 硬件多样化:不同的 CPU、GPU 和内存比比皆是;
  - 浏览器多样化:各种浏览器版本，有新有旧;
  - 上网方式:真实连接的带宽和延迟可能不断变化。

Navigation Timing 2



W3C Web Performance Working Group

Browser Optimization

- The exact list of performed optimizations will differ by browser vendor, but at their core the optimizations can be grouped into two broad classes:
  - Document-aware optimization : The networking stack is integrated with the document, CSS, and JavaScript parsing pipelines to help identify and prioritize critical network assets, dispatch them early, and get the page to an interactive state as soon as possible. This is often done via resource priority assignments, lookahead parsing, and similar techniques.
  - Speculative optimization: The browser may learn user navigation patterns over time and perform speculative optimizations in an attempt to predict the likely user actions by pre-resolving DNS names, pre-connecting to likely hostnames, and so on.

four techniques employed by most browsers

Resource pre-fetching and prioritization

- Document, CSS, and JavaScript parsers may communicate extra information to the network stack to indicate the relative priority of each resource: blocking resources required for first rendering are given high priority, while low-priority requests may be temporarily held back in a queue.

DNS pre-resolve

- Likely hostnames are pre-resolved ahead of time to avoid DNS latency on a future HTTP request. A pre-resolve may be triggered through learned navigation history, a user action such as hovering over a link, or other signals on the page.

TCP pre-connect

- Following a DNS resolution, the browser may speculatively open the TCP connec- tion in an anticipation of an HTTP request. If it guesses right, it can eliminate another full roundtrip (TCP handshake) of network latency.

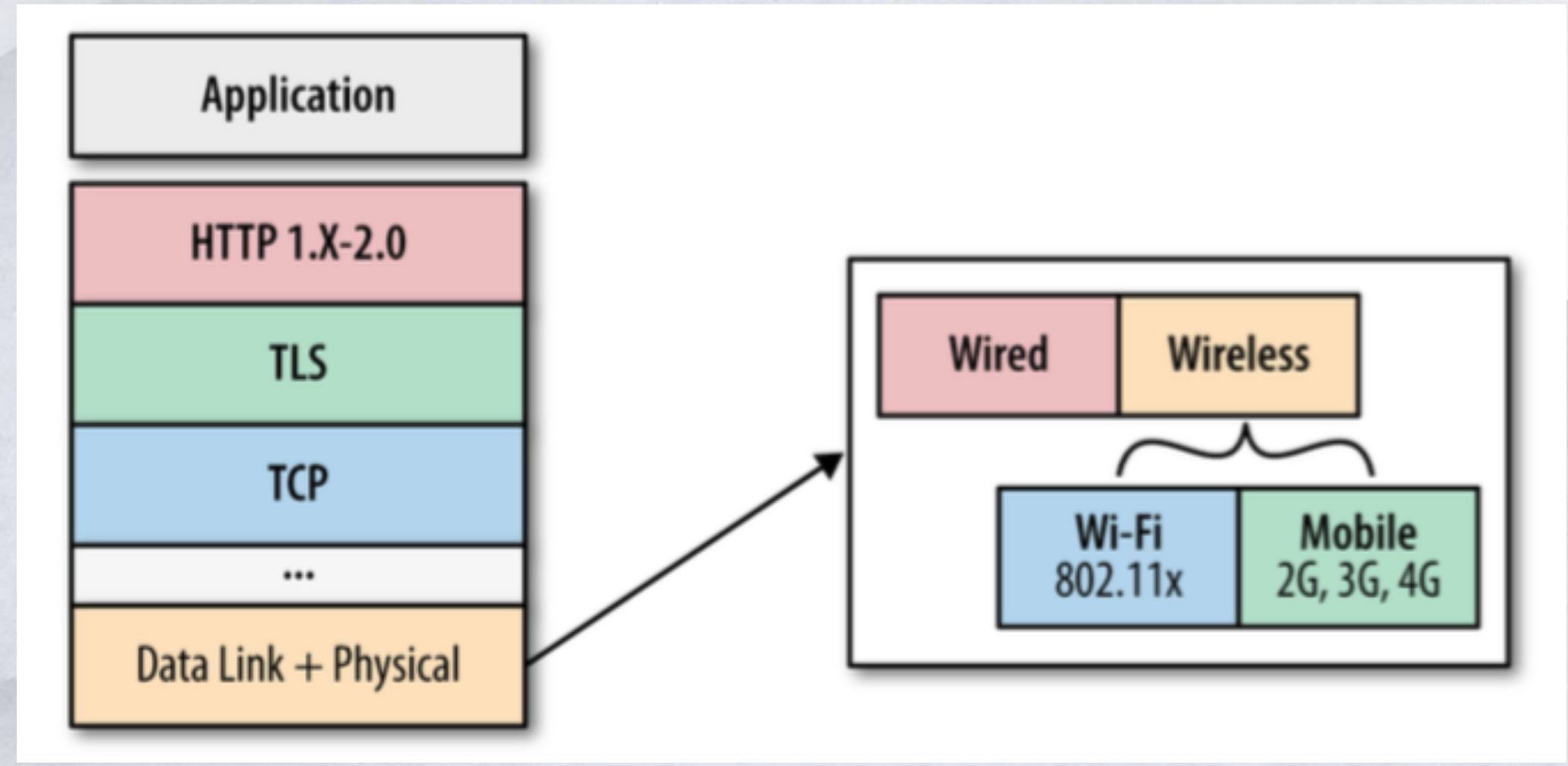
Page pre-rendering

- Some browsers allow you to hint the likely next destination and can pre-render the entire page in a hidden tab, such that it can be instantly swapped in when the user initiates the navigation.

speculative optimization

- <link rel="dns-prefetch" href="//hostname_to_resolve.com">
- <link rel=“subresource" href="/javascript/myapp.js">
- <link rel=“prefetch" href="/images/big.jpeg">
- <link rel="prerender"href="//example.org/next_page.html">
-

Optimizing Application Delivery



Evergreen Performance Best Practices

- two criteria
  - eliminate or reduce unnecessary network latency
  - minimize the amount of transferred bytes.

performance rules

- Reduce DNS lookups
  - Every hostname resolution requires a network roundtrip, imposing latency on the request and blocking the request while the lookup is in progress.
- Reuse TCP connections
  - Leverage connection keepalive whenever possible to eliminate the TCP handshake and slow-start latency overhead.
- Minimize number of HTTP redirects
  - HTTP redirects can be extremely costly, especially when they redirect the client to a different hostname, which results in additional DNS lookup, TCP handshake latency, and so on. The optimal number of redirects is zero.
- Use a Content Delivery Network (CDN)
  - Locating the data geographically closer to the client can significantly reduce the network latency of every TCP connection and improve throughput. This advice applies both to static and dynamic content.
- Eliminate unnecessary resources
  - No request is faster than a request not made.

More rules

- Cache resources on the client
  - Application resources should be cached to avoid re-requesting the same bytes each time the resources are required.
- Compress assets during transfer
  - Application resources should be transferred with the minimum number of bytes: always apply the best compression method for each transferred asset.
- Eliminate unnecessary request bytes
  - Reducing the transferred HTTP header data (i.e., HTTP cookies) can save entire roundtrips of network latency.
- Parallelize request and response processing
  - Request and response queuing latency, both on the client and server, often goes unnoticed, but contributes significant and unnecessary latency delays.
- Apply protocol-specific optimizations
  - HTTP 1.x offers limited parallelism, which requires that we bundle resources, split delivery across domains, and more. By contrast, HTTP 2.0 performs best when a single connection is used and HTTP 1.x specific optimizations are removed.

Cache Resources on the Client

- Cache-Control header can specify the cache lifetime (max-age) of the resource.
- Last-Modified and ETag headers provide validation mechanisms.

Compress Transferred Data

- The size of text-based assets, such as HTML, CSS, and JavaScript, can be reduced by 60%–80% on average when compressed with Gzip.
- Images, on the other hand, require a more nuanced consideration:
  - Images account for over half the transferred bytes of an average page.
  - Image files can be made smaller by eliminating unnecessary metadata.
  - Images should be resized on the server to avoid shipping unnecessary bytes.
  - An optimal image format should be chosen based on type of image.
  - Lossy compression should be used whenever possible.

Eliminate Unnecessary Request Bytes

- significant performance implications for your application:
  - Associated cookie data is automatically sent by the browser on each request.
  - In HTTP 1.x, all HTTP headers, including cookies, are transferred uncompressed.
  - In HTTP 2.0, compression is applied, but the potential overhead is still high.
  - In the worst case, large HTTP cookies can add entire roundtrips of network latency by exceeding the initial TCP congestion window.

Parallelize Request and Response Processing

- Here's how to get the best performance:
  - Use connection keepalive and upgrade from HTTP 1.0 to HTTP 1.1.
  - Leverage multiple HTTP 1.1 connections where necessary for parallel downloads.
  - Leverage HTTP 1.1 pipelining whenever possible.
  - Investigate upgrading to HTTP 2.0 to improve performance.
  - Ensure that the server has sufficient resources to process requests in parallel.

Optimizing for HTTP 1.x

- The order in which we optimize HTTP 1.x deployments is important: configure servers to deliver the best possible TCP and TLS performance, then carefully review and apply mobile and evergreen application best practices: measure, iterate.
  - Leverage HTTP pipelining
    - If your application controls both the client and the server, then pipelining can help eliminate significant amounts of network latency.
  - Apply domain sharding
    - If your application performance is limited by the default six connections per origin limit, consider splitting resources across multiple origins.
  - Bundle resources to reduce HTTP requests
    - Techniques such as concatenation and spriting can both help minimize the protocol overhead and deliver pipelining-like performance benefits.
  - Inline small resource
    - Consider embedding small resources directly into the parent document to minimize the number of requests.

Optimizing for HTTP 2.0

- The primary focus of HTTP 2.0 is on improving transport performance and enabling lower latency and higher throughput between the client and server.
  - Server should start with a TCP cwnd of 10 segments.
  - Server should support TLS with ALPN negotiation (NPN for SPDY).
  - Server should support TLS resumption to minimize handshake latency.
- Next up—surprise—apply the mobile and other evergreen application best practices: send fewer bytes, eliminate requests, and adapt resource scheduling for wireless networks.
- Finally, undo and unlearn the bad habits of domain sharding, concatenation, and image spriting; these workarounds are no longer required with HTTP 2.0.

Removing 1.x Optimizations

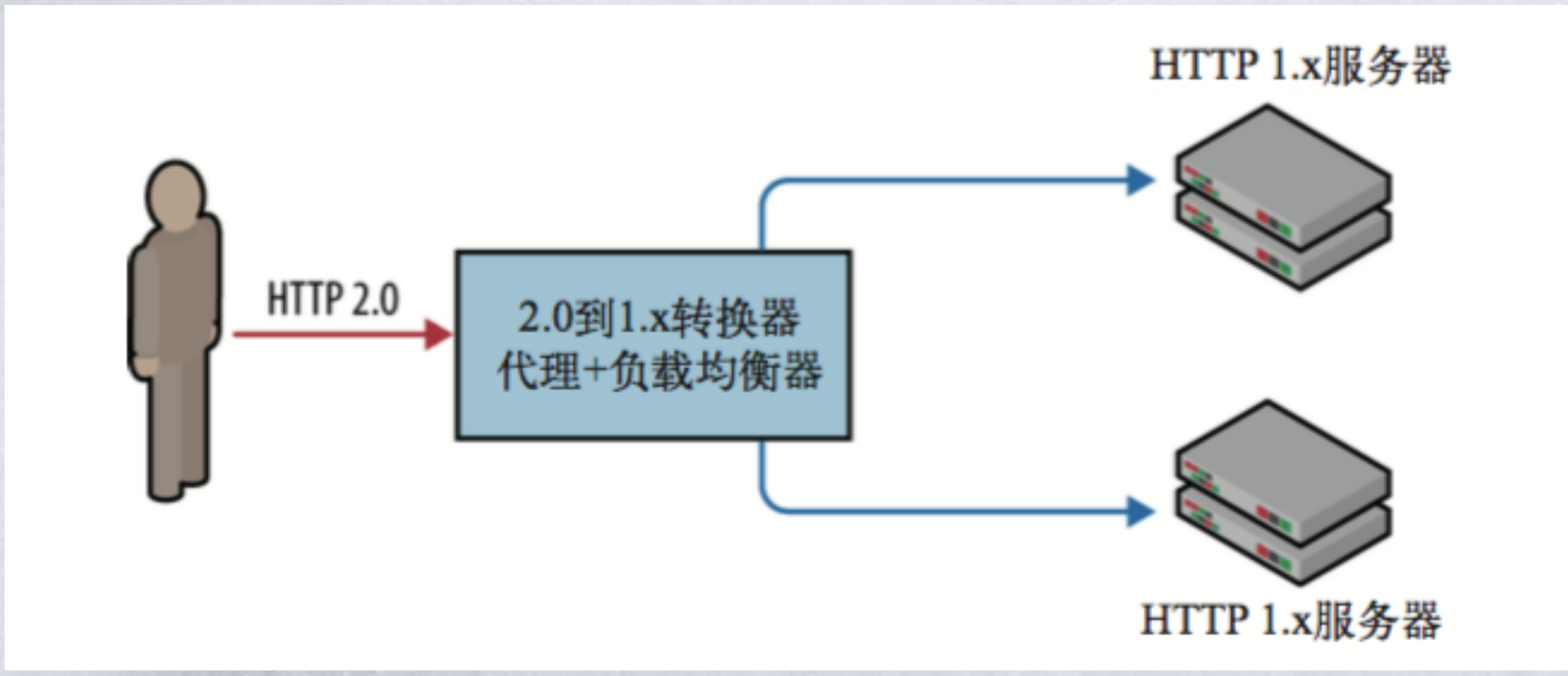
- Use a single connection per origin
  - HTTP 2.0 improves performance by maximizing throughput of a single TCP connection. In fact, use of multiple connections (e.g., domain sharding) is a performance anti-pattern for HTTP 2.0, as it reduces the effectiveness of header compression and request prioritization provided by the protocol.
- Remove unnecessary concatenation and image spriting
  - Resource bundling has many downsides, such as expensive cache invalidations, larger memory requirements, deferred execution, and increased application complexity. With HTTP 2.0, many small resources can be multiplexed in parallel, which means that the downsides of asset bundling will almost always outweigh the benefits of delivering more granular resources.
- Leverage server push
  - The majority of resources that were previously inlined with HTTP 1.x can and should be delivered via server push. By doing so, each resource can be cached individually by the client and reused across different pages, instead of being embedded in each and every page.

Dual-Protocol Application Strategies

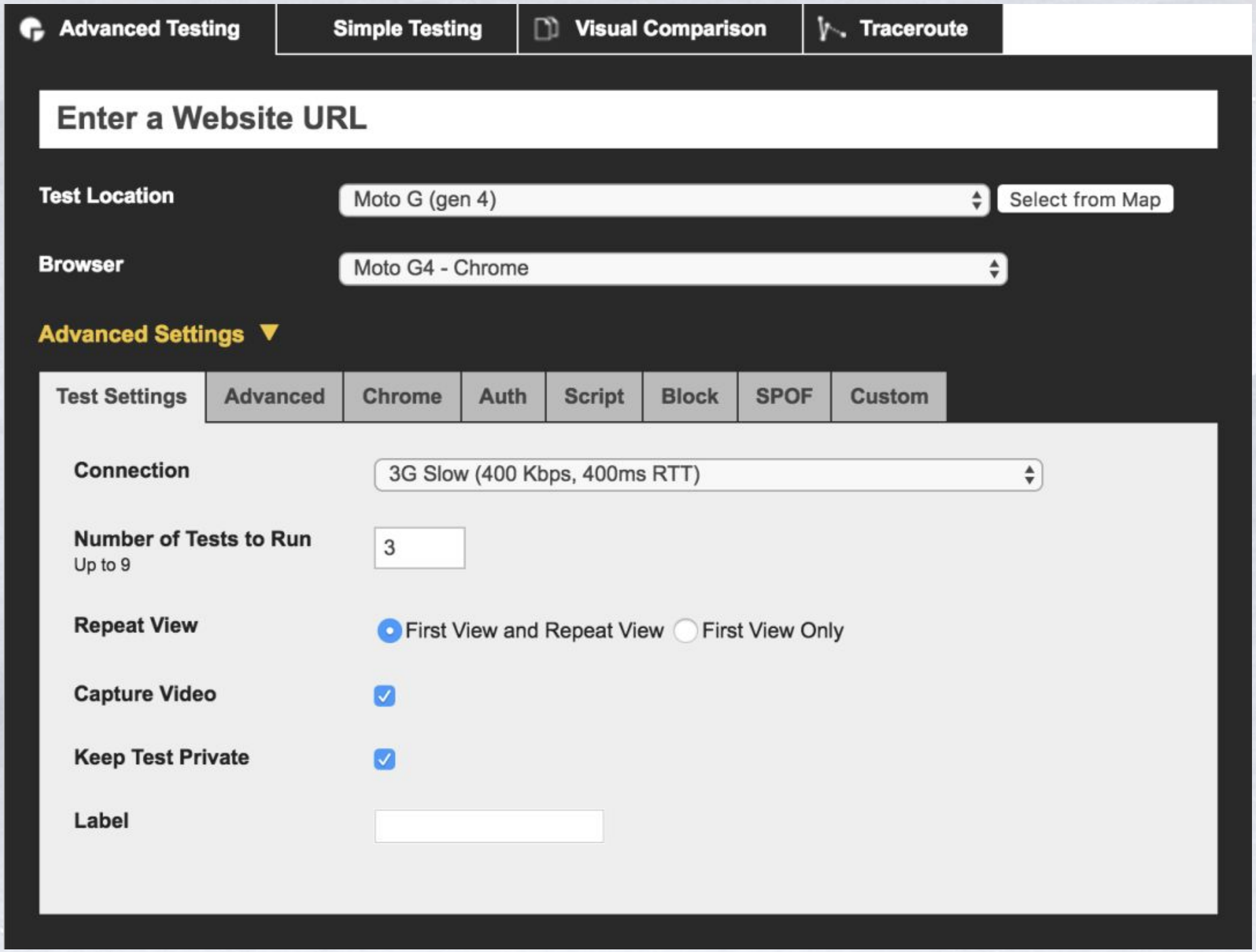
- Same application code, dual-protocol deployment
- Split application code, dual-protocol deployment
- Dynamic HTTP 1.x and HTTP 2.0 optimization
- HTTP 2.0, single-protocol deployment

- Dynamic Optimization with PageSpeed
  - Google's PageSpeed Optimization Libraries (PSOL) provide an open source implementation of over 40 various "web optimization filters," which can be integrated into any server runtime and applied dynamically to any application.

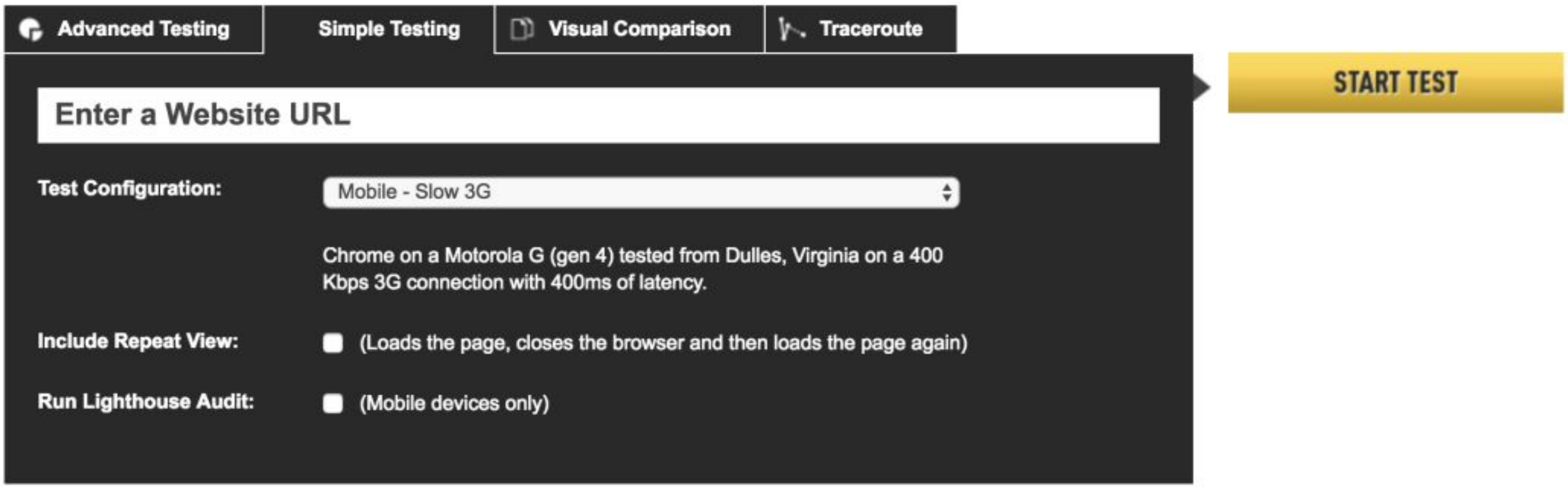
Translating 1.x to 2.0 and Back



WebpageTest



Simple Testing



结果

Need help improving?

Web Page Performance Test for

software.nju.edu.cn

From: Dulles, VA - **Chrome** - **Cable**
2018/11/14 上午10:44:50

A	A	F	C	F	X
First Byte Time	Keep-alive Enabled	Compress Transfer	Compress Images	Cache static content	Effective use of CDN

Summary | Details | Performance Review | Content Breakdown | Domains | Processing Breakdown | Screen Shot | Image Analysis | Request Map

Tester: VM1-07-192.168.11.94
Test runs: **3**
Re-run the test

Raw page data - Raw object data
Export HTTP Archive (.har)
View Test Log

Performance Results (Median Run)

						Document Complete			Fully Loaded			
	Load Time	First Byte	Start Render	Speed Index	First Interactive (beta)	Time	Requests	Bytes In	Time	Requests	Bytes In	Cost
First View (Run 3)	4.314s	0.694s	4.000s	4.848s	> 3.901s	4.314s	36	286 KB	5.870s	43	738 KB	$$---
Repeat View (Run 3)	1.327s	0.701s	1.400s	1.679s	> 1.317s	1.327s	2	25 KB	1.327s	2	25 KB	

Plot Full Results

指标

- 1. 网页级指标(Page-level Metrics)
  - 这些是为整个页面捕获并显示的顶级度量值。
- 2. 整页加载时间(Load Time)
  - 测量的时间是从初始化请求，到开始执行window.onload事件。
- 3. 页面所有元素加载时间(Fully Loaded)
  - 从初始化请求，到Document Complete后，2秒内（中间几百毫秒轮询）没有网络活动的时间，但这2秒是不包括在测量中的，所以会出现两个差值大于或小于2秒。
- 4. 第一个字节加载时间(First Byte)
  - 第一个字节时间（通常缩写为TTFB）被测量为从初始化请求，到服务器响应的第一个字节，被浏览器接收的时间（不包括DNS查询、TCP连接的时间）。
- 5. 页面渲染时间(Start Render)
  - 测量的时间是从初始化请求，到第一个内容被绘制到浏览器显示的时间。在瀑布图中有两个参数指标Start Render和msFirstPaint。
    - Start Render是通过捕获页面加载的视频，并在浏览器第一次显示除空白页之外的其他内容时查看每个帧来衡量的。它只能在实验室测量，通常是最准确的测量。
    - msFirstPaint（IE专用属性）是由浏览器本身报告的一个测量，它认为绘制的第一个内容。通常是相当准确，但有时它报告的时候，浏览器只画一个空白屏幕。
- 6. 首屏展现平均值(Speed Index)
  - 表示页面呈现用户可见内容的速度（越低越好）。有关如何计算的更多信息，请参见：Speed Index。
- 7. DOM元素数量(DOM Elements)
  - 在测试结束时测试页面上的DOM元素的计数。
- 8. 请求级度量标准(Request-level Metrics)
  - 这些是为每个请求捕获和显示的度量。

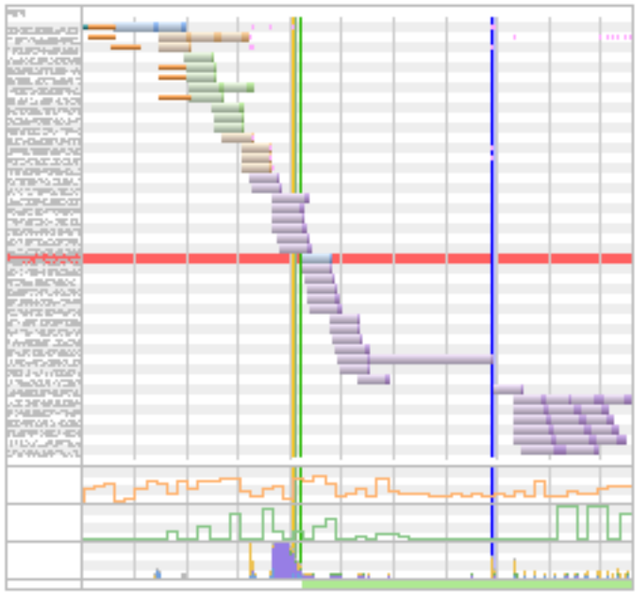
Performance Results (Median Run)

						Document Complete			Fully Loaded			
	Load Time	First Byte	Start Render	Speed Index	First Interactive (beta)	Time	Requests	Bytes In	Time	Requests	Bytes In	Cost
First View (Run 3)	4.281s	0.697s	4.100s	4.804s	> 4.051s	4.281s	36	286 KB	5.688s	43	738 KB	$$---

Plot Full Results

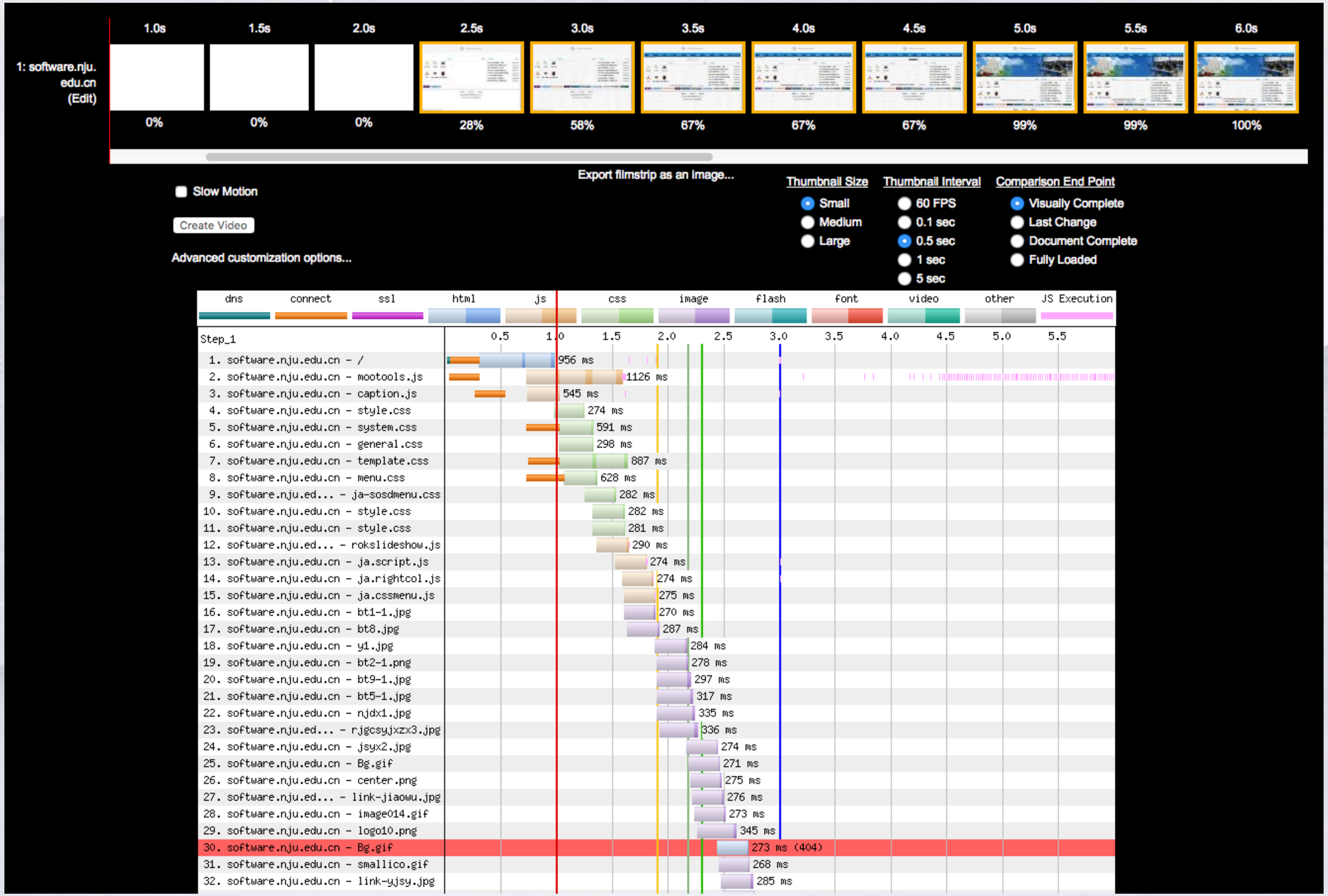
Test Results

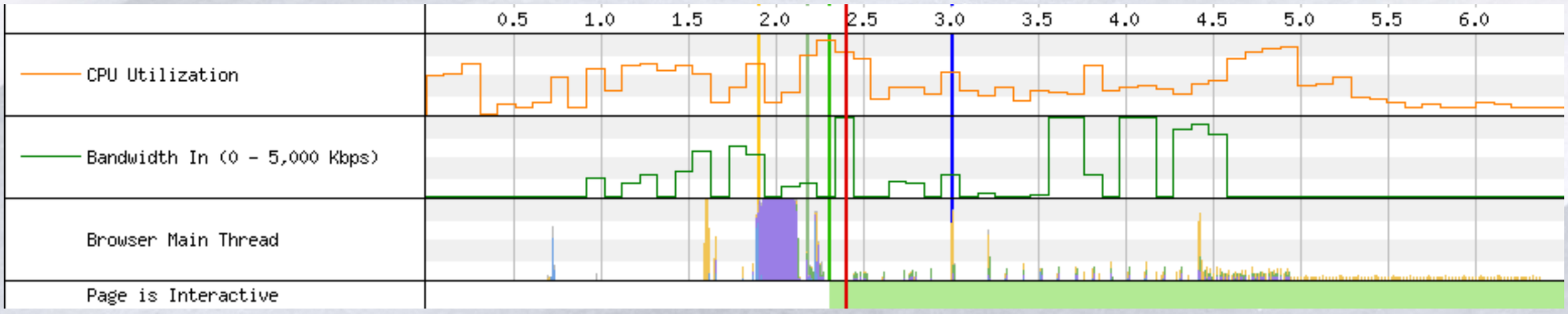
Run 1:

	Waterfall	Screen Shot	Video
First View (3.949s) Timeline (view) Processing Breakdown Trace (view)			Filmstrip View - Watch Video

Content Breakdown

Filmstrip View




0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
CPU Utilization
Bandwidth In (0 - 5,000 Kbps)
Browser Main Thread
Page is Interactive

A Netflix Web Performance Case Study

- Improving Time-To-Interactive for Netflix.com on Desktop
- 通过改进 Netflix.com 注册过程中所使用的 JavaScript 及预加载技术，开发团队能够为移动用户和桌面用户提供更好的用户体验，主要改进如下。
  - 加载和交互时间减少了 50%（Netflix.com 桌面登录主页）；
  - 在从 React 和其他客户端库切换到普通的 JavaScript 之后，JavaScript 包大小减少了 200KB。服务器端仍然使用 React。
  - HTML、CSS、JavaScript（React）预加载使后续页面的浏览交互时间减少了 30%。

https://medium.com/dev-channel/a-netflix-web-performance-case-study-c0bcde26a9d9

减少 JavaScript 传输，缩短交互时间

- Netflix 针对其登录主页的性能进行了优化
  - 这个页面最初包含 300KB 的 JavaScript，其中一些是 React 和其他客户端代码（比如像 Lodash 这样的实用程序库），还有一些是补充 React 状态所需的上下文数据。
- 使用 Chrome 的开发工具和 Lighthouse 模拟在 3G 连接上加载登录主页，结果显示，登录主页需要 7 秒的加载时间，对于一个简单的登录页面来说太长了，因此需要研究改进的可能。通过一些性能审计，Netflix 发现他们客户端的 JS 开销很高。
- 登录主页是否真得需要 React?
  - 用Vanilla JavaScript 替换

What is Vanilla JS?

- Vanilla JS is a fast, lightweight, cross-platform framework for building incredible, powerful JavaScript applications.

- Who's using Vanilla JS?
  - Facebook, Google, YouTube, Yahoo, Wikipedia, Windows Live, Twitter, Amazon, LinkedIn, MSN, eBay, Microsoft, Tumblr, Apple, Pinterest, PayPal, Reddit, Netflix, Stack Overflow

- In fact, Vanilla JS is already used on more websites than jQuery, Prototype JS, MooTools, YUI, and Google Web Toolkit - combined.

http://vanilla-js.com/

Getting Started

- To use Vanilla JS, just put the following code anywhere in your application's HTML:

```
<script src="path/to/vanilla.js"></script>
```

- When you're ready to move your application to a production deployment, switch to the much faster method:

```
```

JS框架的优点

- JS框架封装了复杂困难的代码;
- JS框架能加快开发速度,更快完成项目;
- JS框架让你更专注于产品内容的价值,而不是实现过程;
- JS框架让合作更简单,大家都对基础代码有共同的理解;
- JS框架还会强迫你练习,多实践不是坏事,顺能生巧嘛。

JS框架的问题

- 每个项目的开发都会遇到框架文档没有说明的问题,这时候就要深入框架查找原因,这时候就需要深厚的原生JavaScript功力了。
- 过去5年能被大家注意到的新框架产生了数十个,猜猜5年后会有多少新的框架发布,一旦确定了项目的技术栈,几年后项目怎么办?

Why Vanilla JavaScript

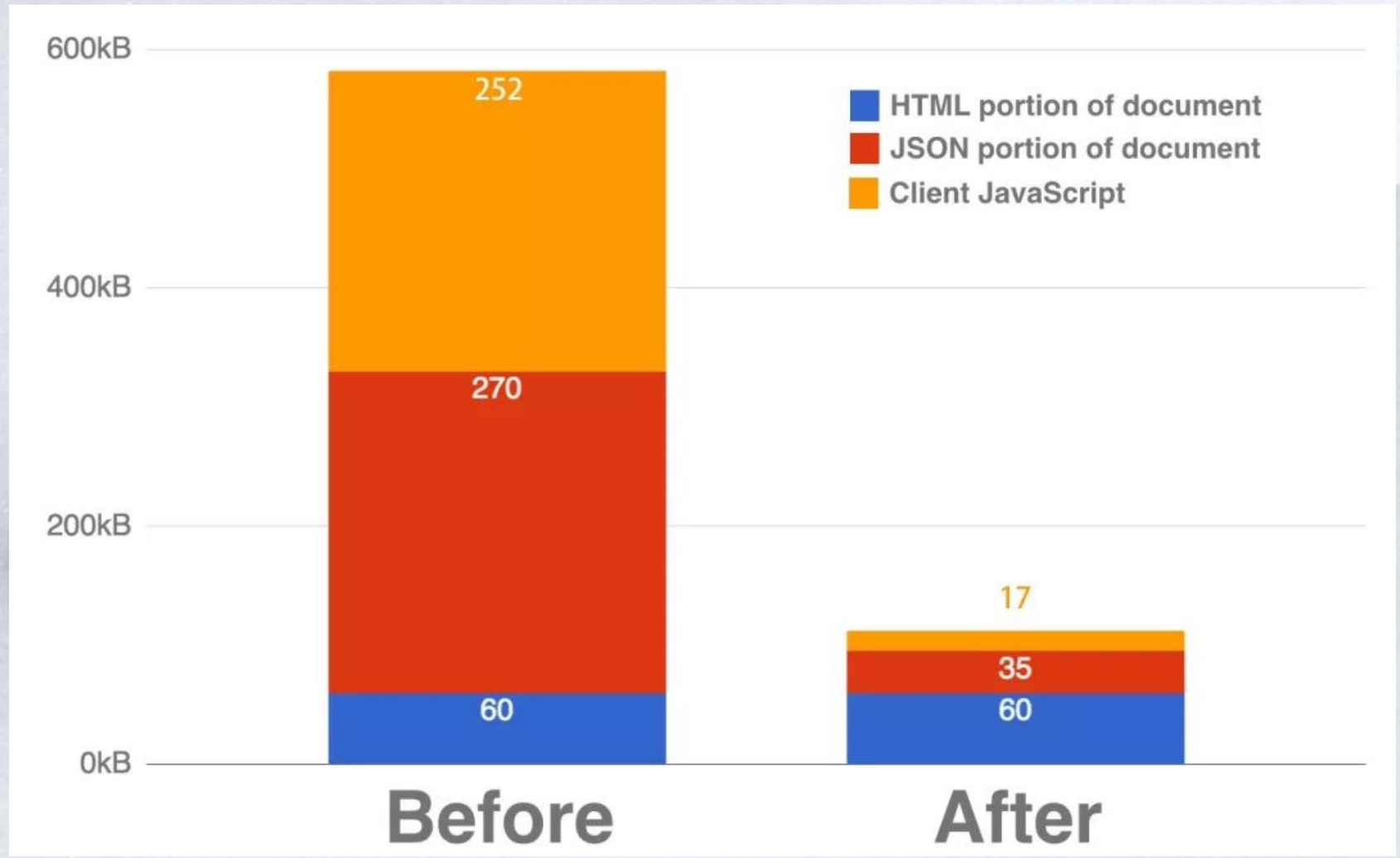
- 学会 Vanilla JavaScript能真正理解JS框架,甚至能为其贡献代码,还能帮助选择合适的框架。
- 如果不知道Web基本原则,语言本身的演变和新框架的不断到来。。。
- 知道纯JS将成为一个能够(不用疯狂搜索原因)解决复杂问题的关键工程师。
- 增加通用能力和生产力,不管在前端还是后端。
- 创新的工具,而不只是执行。
- 指导什么时候使用或者不使用框架。
- 更好地了解浏览器和计算机工作原理。

优化

- 移植到原生 JavaScript 的组件列表：
  - 基础交互（主页选项卡）
  - 语言切换器
  - “Cookie 横幅（Cookie banner）”（针对非美国用户）
  - 客户端日志分析
  - 性能度量和记录
  - 广告归属检测引导代码（出于安全考虑，放在沙箱式 iFrame 中）

结果

- 尽管 React 最初占用的空间仅为 45KB，但将 React、几个库和相应的应用程序代码从客户端移除后，JavaScript 的总量减少了 200KB 以上，这使得 Netflix 在登录主页的交互时间减少了 50% 以上。



后续页面的 React 预加载

- 预加载
  - 通过浏览器内置的 API 和 XHR 预加载
  - 交互时间减少了 30%

NETFLIX

COMPARISON OF PREFETCHING TECHNIQUES

	Browser API (<link />)	XHR
Browser Support	Partial	Full
Success Rate	30%-90%, depending on browser	> 95%
Ease of Implementation	1 line of code	3 lines of code
Can Prefetch HTML Document	Yes	No

总结

- 密切关注 JavaScript 的开销
- Netflix 的折中方案是，使用 React 在服务器端渲染登录页面，但同时也为注册过程的其他部分预取 React 代码。这不仅优化了首次加载性能，还优化了注册过程其余部分的加载时间，因为它是单页应用，所以有更大的 JS 包需要下载。
- 补充：
  - Netflix 考虑过 Preact，但是，对于一个交互性比较低的简单页面流，使用普通的 JavaScript 是一个更简单的选择。
  - Netflix 尝试使用 Service Workers 进行静态资源缓存。当时，Safari 不支持这个 API（现在支持了），但他们现在又在探索这个 API。Netflix 的注册过程需要比客户体验更多的遗留浏览器支持。许多用户会在旧的浏览器上注册，但会在他们本地的移动应用程序或电视设备上观看 Netflix。
  - Netflix 的登录页面极为动态。这是他们的注册过程中进行 A/B 测试最多的页面，机器学习模型用于根据位置、设备类型和许多其他因素定制消息和图像。支持近 200 个国家，每个派生页面都面对着不同的本地化、法律和价值信息挑战。

References

- https://developers.google.com/web/fundamentals/performance/why-performance-matters/
- https://www.webpagetest.org
- https://developers.google.com/web/fundamentals/performance/speed-tools/
- https://github.com/pwstrick/WebPagetest-Docs
- High performance browser networks

- 高性能网站建设指南
- 高性能网站建设进阶指南
- 大型网站技术架构－核心原理与案例分析
-

Thanks!!!