

HTML5 的 WebSocket: Web 领域一次巨大的飞跃

信息与软件技术, 2011

彼得·吕贝尔斯, 弗兰克·格列柯

最近出现了许许多多的议论, 围绕 HTML5 的 WebSocket, 它定义了通过运行在 Web 上的单一插槽全双工通信通道。HTML5 的 WebSocket 不只是对传统 HTTP 通信一个增量升级; 它更代表着一个巨大的进步, 尤其是对实时的, 事件驱动的 Web 应用程序。HTML5 的 WebSocket 提供了这样一个从旧的戏剧性的改善, 复杂的“黑客”用于模拟全双工连接在一个浏览器的通信, 它促使谷歌的 Ian Hickson, HTML5 规范的领导者这样说说: “减少的数据为 2 千字节.....但降低了延迟远远超过从 150 毫秒到 50 毫秒这个范围, 事实上, 对谷歌来说仅这两个因素都已经足以让 WebSocket 认真而且有趣。”让我们来看看 HTML5 的 WebSocket 是如何做到可以通过比传统的解决方案提供了这样一个令人难以置信的而且大幅度减少不必要的网络流量和延迟的方案。

轮询, 长轮询和流媒体, 头部请求 2.0

通常, 当浏览器访问一个网页, 一个 HTTP 请求发送到托管该网页的 Web 服务器。该网站的服务器确认该请求并发送回响应。在许多情况下, 例如, 股票价格, 新闻报道, 门票销售, 流量模式, 医疗设备的读数, 等等的响应可能是过时的由浏览器呈现页面的时间。如果你想获得最先进最新的实时信息, 你可以不断地手动刷新页面, 但是这显然不是一个完美的解决方案。

其中最显著的提供实时 web 应用程序的解决方案主要是围绕轮询和其它服务器端推送技术, 很不幸的是, 这将延迟完成 HTTP 响应从服务器传递到客户端。基于 Comet 推送一般是用 JavaScript 实现, 并使用连接策略, 例如长轮询或流。

轮询, 浏览器定期发送 HTTP 请求并立即收到响应。这种技术是第一次尝试为浏览器提供的实时信息。显然, 这是一个很好的解决方案, 如果消息传递的确切时间间隔是已知的, 因为你可以同步发生在客户端的请求, 只有当信息在服务器上可用。然而, 实时数据往往不是预测, 发送不必要的请求不可避免的, 因此, 很多连接被打开, 并在低信息速率的情况下关闭不必要的。

长轮询, 浏览器发送一个请求到服务器, 服务器将保持开放连接在设定时间内。如果通知是在该期间内接收到的, 包含该消息的响应发送到客户端。如果通知没有在规定时间内收到, 服务器发送一个响应终止打开的请求。重要的是要了解是很重要的, 但是, 当你有一个很高的消息量, 长轮询与传统的轮询相比是没有任何显著的性能改善。事实上, 它还有可能会更糟, 因为长轮询可能会失控进入流连续循环。

随着流媒体, 浏览器发送一个完整的请求, 但服务器发送并维护不断更新, 并保持无限期开放 (或设定的时间段) 公开回应。响应然后每当更新一个消息是准备好发送, 但服务器从接收到信号完成响应, 从而保持连接打开, 以接收未来的消息。然而, 由于流仍封装在 HTTP, 中间防火墙和代理服务器可

以选择来缓冲反应，增加了消息传递的延迟。因此，许多流媒体解决方案，还是长轮询中被检测到的缓冲代理服务器的情况。或者，TLS（SSL）连接可以用来从被缓冲屏蔽的响应，但在这种情况下，每一个连接的建立和关闭会占用更多的可用的服务器资源。

最终，所有这些方法提供的实时数据涉及到的 HTTP 请求和响应头，其中包含许多额外的，不必要的头数据，并引入延迟。最重要的是，全双工连接需要的不仅仅是从服务器到客户端的下行连接。在努力模拟从半双工到 HTTP 全双工通信，许多今天的解决方案使用两个连接：一个用于下行，另一个用于上行。这两个连接的维护和协调引入巨大的开销在资源消耗方面，并增加了许多不可控的复杂性。简单地讲，HTTP 不被设计用于实时，全双工通信，你可以在下面的图中，它显示了与建设，显示从后端数据源的实时数据彗星 Web 应用程序的复杂性看使用发布/订阅模式在半双工 HTTP。

当您试图向外扩展那些复杂的解决方案，很显然它会变得更糟。模拟基于 HTTP 的双向通信的浏览器是容易出错的，复杂的和所有的复杂性不能扩展。即使您的最终用户可能会享受的东西，看起来像一个实时 Web 应用程序，这种“实时”体验是建立在极高的代价之上的。这是你会在额外的延迟，不必要的网络流量和 CPU 性能的拖累所必须付出的代价。

HTML5 的 WebSocket 来救援

定义在 HTML5 规范的通信部分，HTML5 的 WebSocket 代表网络通信，一个通过运行在 Web 上的单一插槽全双工，双向通信信道的一个推进。HTML5 的 WebSocket 提供了一个真正的标准，你可以用它来构建可扩展，实时 web 应用程序。此外，由于它提供了一个接口，是基于浏览器底层的，它消除了许多在 Comet 解决方案上容易发生的问题。WebSocket 降低了开销，减轻了程序的复杂性。

要建立一个 WebSocket 的连接，从他们的初次握手期间 HTTP 协议 WebSocket 协议的客户端和服务器的升级，如下面的例子：

例 1 - WebSocket 的握手（浏览器请求和服务器响应）

```
GET /text HTTP/1.1\r\n Upgrade: WebSocket\r\n Connection: Upgrade\r\n Host:
www.websocket.org\r\n ... \r\n
HTTP/1.1 101 WebSocket Protocol Handshake\r\n Upgrade: WebSocket\r\n
Connection: Upgrade\r\n ... \r\n
```

一旦建立，WebSocket 的数据帧可以在全双工模式下被来回传送客户端和服务器之间。文本和二进制帧可以全双工的在同一时间被发送，在任一方向被发送。这些数据被压缩最小只有两个字节。在文本框的情况下，每个帧开始于一个字节为 0x00，以 0xFF 字节结束，并且包含在之间的 UTF-8 数据。

WebSocket 的文本框使用一个终结者，而二进制帧使用一个长度前缀。

注：虽然网络套接字协议已准备好支持客户端的多样化，它不能提供原始二进

制数据到 JavaScript，因为 JavaScript 不支持字节类型。因此，如果客户端是 JavaScript 的，但它可以被传递到支持它的其他客户端的二进制数据将被忽略。

蜕变与 HTML5 的 WebSocket

因此，是如何戏剧性的使不必要的网络流量和延迟减少的？让我们比较一个轮询应用程序，通过和 WebSocket 应用做对比。

对于轮询的例子，我创建了一个 Web 页面请求从的 RabbitMQ 消息代理实时股票数据使用传统的发布/订阅模型一个简单的 Web 应用程序。它通过轮询承载一个 Web 服务器上的 Java Servlet 的做到这一点。该 RabbitMQ 的消息代理接收数据从一个虚构的股票价格与实时不断更新的价格。网页连接并订阅一个特定的股票频道（在消息代理的主题），并使用一个 XMLHttpRequest 轮询每秒一次的更新。当收到更新时，一些执行计算和库存数据显示在一个表中，如在下面的图像所示。








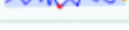

COMPANY	SYMBOL	PRICE	CHANGE	SPARKLINE	OPEN	LOW	HIGH
THE WALT DISNEY COMPANY	DIS	27.65	0.56		27.09	24.39	29.80
GARMIN LTD.	GRMN	35.14	0.35		34.79	31.31	38.27
SANDISK CORPORATION	SNDK	20.11	-0.13		20.24	18.22	22.26
GOODRICH CORPORATION	GR	49.99	-2.35		52.34	47.11	57.57
NVIDIA CORPORATION	NVDA	13.92	0.07		13.85	12.47	15.23
CHEVRON CORPORATION	CVX	67.77	-0.53		68.30	61.49	75.11
THE ALLSTATE CORPORATION	ALL	30.88	-0.14		31.02	27.92	34.12
EXXON MOBIL CORPORATION	XOM	65.66	-0.86		66.52	59.87	73.17
METLIFE INC.	MET	35.58	-0.15		35.73	32.16	39.30

图 1 一个 JavaScript 股票行情应用程序

注：后端股票数据实际上产生了大量每秒股票价格更新，因此使用轮询在一秒钟的间隔，其实比使用长轮询的解决方案，这将导致一系列连续的请求更谨慎。轮询有效节流这里传入的更新。

这一切都看起来不错，但引擎盖下观察，可以发现有一些与此应用程序的严重问题。例如，在 Mozilla Firefox 用 Firebug（一个 Firefox 插件，允许你调试网页和监控需要加载的网页和执行脚本的时候），你可以看到，GET 请求服务器在一秒钟的间隔。开启实时 HTTP 头（另一个 Firefox 插件，显示实时的 HTTP 头流量）揭示了包头开销与每个请求关联的惊人数量。下面的两个例子显示了 HTTP 头数据只是一个单一的请求和响应。

例2 - HTTP 请求头

```
GET /PollingStock//PollingStock HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/PollingStock/
Cookie: showInheritedConstant=false; showInheritedProtectedConstant=false;
showInheritedProperty=false; showInheritedProtectedProperty=false;
showInheritedMethod=false; showInheritedProtectedMethod=false;
showInheritedEvent=false; showInheritedStyle=false; showInheritedEffect=false
```

例2 – HTTP 响应头

```
HTTP/1.x 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: text/html;charset=UTF-8
Content-Length: 21
Date: Sat, 07 Nov 2009 00:32:46 GMT
```

这里所有的字符只是为了做一个测试。总的 **HTTP** 请求和响应头信息开销包含 871 个字节，而且还不包括任何数据！当然，这只是一个例子，你可以有不超 871 个字节的报头数据，但我也看到了其中的头数据超过 2000 字节的情况。在这个示例应用程序，数据为典型的股票主题消息只有约 20 个字符长。正如你所看到的，它有效地淹没在过多的报头信息，它甚至没有被要求摆在首位！

那么，是什么，当你部署这个应用程序的大量用户的情况发生？让我们来看看网络吞吐量刚刚与此轮询应用程序在三个不同的用例相关的 **HTTP** 请求和响应头数据。

用例 A：1000 客户端，每秒轮询一次网络吞吐量（ 871×1000 ）= 871000 字节=每秒 6,968,000 位（6.6 Mbps）的

用例 B：10000 客户端，每秒轮询一次网络吞吐量（ $871 \times 10,000$ ）= 8,710,000 字节=每秒 6968 万比特（66 Mbps）的

使用案例 C：100,000 客户查询每 1 秒一次网络吞吐量（ $871 \times 100,000$ ）
= 87100000 字节=每秒 696800000 位（665 Mbps）的

这是不必要的网络吞吐量数据！如果我们能只得到过线的基本数据。嗯，会怎么样？可以使用 HTML5 的 WebSocket！我重建了使用 HTML5 的 WebSocket 的应用程序，添加一个事件处理程序到 web 页面异步监听来自消息代理股票更新消息。这些消息是，只有两个字节的开销的 WebSocket 的信息（而不是前面的 871 字节）！来看看它如何影响我们三个用例的网络吞吐量开销。

用例 A：1000 客户端，每秒轮询一次网络吞吐量（ 2×1000 ）= 2000 字节=每秒 16,000 位（0.015 Mbps）的

用例 B：10000 客户端，每秒轮询一次网络吞吐量（ $2 \times 10,000$ ）= 20,000 字节=每秒 160,000 位（0.153 Mbps）的

使用案例 C：100,000 客户端，每秒轮询一次网络吞吐量（ $2 \times 100,000$ ）= 200,000 字节=每秒 160 万比特（1.526 Mbps）的

正如你可以在下面的图中看到，HTML5 的 WebSocket 提供了一个戏剧性的减少了不必要的网络流量较轮询解决方案。

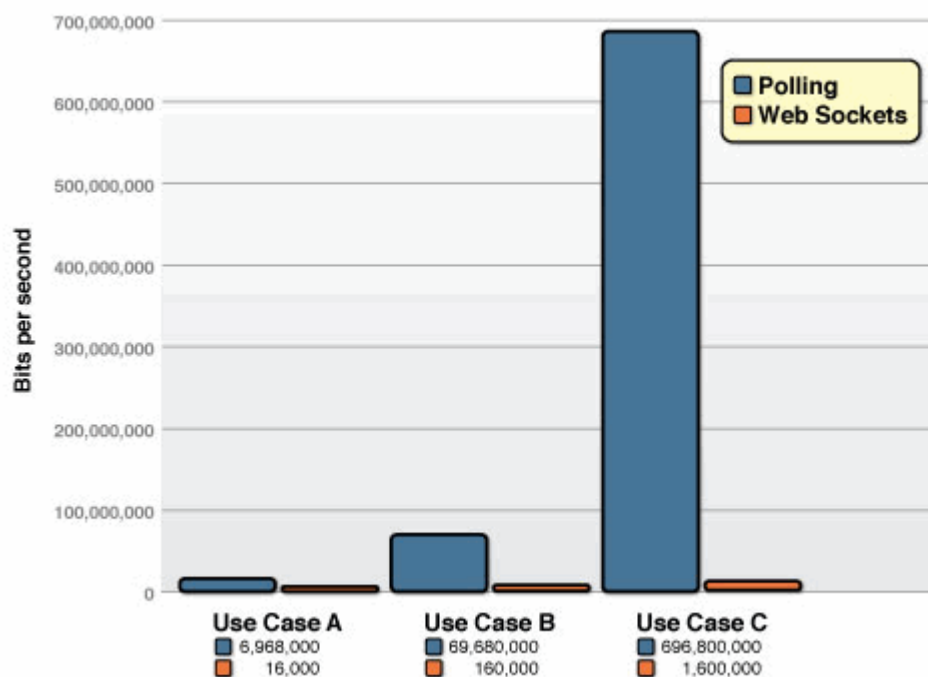


图 2 轮询和 WebSocket 的应用程序之间的不必要的网络吞吐量开销对比

和什么有关延迟的减少？来看看下图。在上半部分，你可以看到半双工轮询方案的延迟。如果我们假定，在本例中，它需要 50 毫秒的消息从服务器传送到浏览器，那么轮询应用介绍了许多额外的延迟，因为新的请求已被发送到服务器时的响应已完成。这个新的请求另需 50 毫秒，在此期间服务器不能发送任何消息给浏览器，导致额外的服务器内存消耗。

在底部的数字的一半，您将看到由 WebSocket 的解决方案提供了延迟的减少。一旦连接升级到 WebSocket 通信，消息可以从服务器流到浏览器，他们到达的那一刻。它仍然需要 50 毫秒的消息从服务器传播到浏览器，但是 WebSocket 的连接保持打开状态所以没有必要发送另一个请求到服务器。

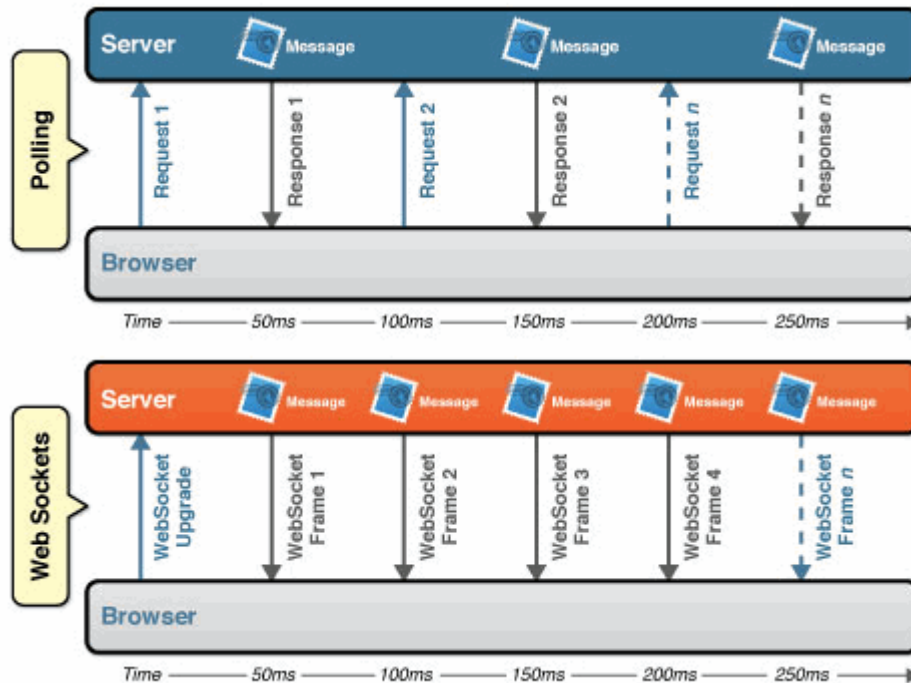


图 3 轮询和 WebSocket 的应用程序之间的延迟比较

HTML5 的 Web Socket 和 Kaazing WebSocket 的网关

今天，只有谷歌的 Chrome 浏览器支持 HTML5 的 WebSocket 本身，但其他浏览器将很快跟进。若要解决这个限制，Kaazing 的 WebSocket 网关提供完整的 WebSocket 仿真的所有旧的浏览器（IE 5.5 +，Firefox 1.5 以上时，Safari 5.0 + 和 Opera 9.5 +），所以你可以从今天开始使用 HTML5 的 WebSocket API。WebSocket 的是伟大的，但你可以做什么，一旦你有一个全双工套接字连接在你的浏览器中作用就更大了。要充分利用 HTML5 的 WebSocket 的全部力量，Kaazing 为二进制通信和更高级别的库文件如 AMQP，XMPP，IRC 和更多的，建立在 WebSocket 的顶部协议的 ByteSocket 库。

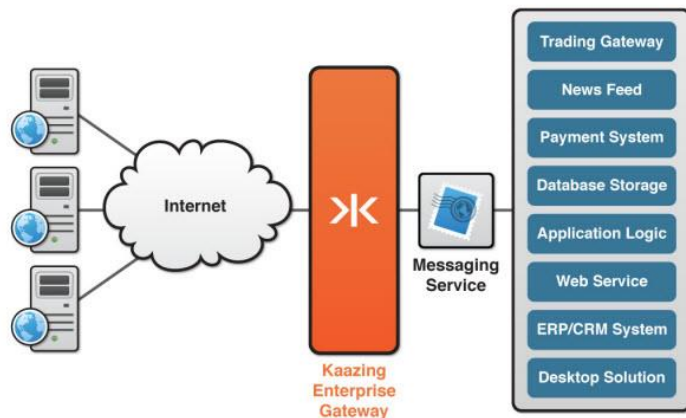


图 4 Kaazing Socket 网关扩展基于 TCP 的消息到浏览器的超高性能

总结

HTML5 的 Web Socket 提供实时网络的可扩展性里面前进了一大步。正如你所看到的这篇文章中，HTML5 的 WebSocket 可以提供 500:1 或差不多大小的 HTTP 头，甚至 1000:1 减少不必要的 HTTP 头流量和 3:1 减少等待时间。这不仅是一个渐进式的改进，这是一个革命性的跳跃，更是一个质的飞跃！

Kaazing WebSocket 的网关，使 HTML5 的 WebSocket 的代码工作在所有浏览器的今天，同时提供额外的协议库，让您充分利用全双工套接字连接的 HTML5 的 Web Socket 提供的全部功能，并直接传送到后端服务。