The realtime web: HTTP/1.1 to WebSocket, SPDY and beyond

Guillermo Rauch @ *QCon*. November 2012.

Guillermo.

CTO and co-founder at LearnBoost**.**

Creator of **socket.io** and **engine.io**.

@**rauchg** on twitter

http://devthought.com

Author of Wiley's **Smashing Node**

http://smashingnode.com

The **realtime** web

HTTP over TCP/IP

No specific **techniques** or **protocols**

Realtime = fast data exchange.

Let's analyze the timeline of a typical web request.

https://mywebsite.com

1. We first need to look up the **DNS** record for mywebsite.com

DNS. Low **bandwidth**. Potentially high **latency**.

Browsers have different approaches and optimizations.

Our best shot at optimizing this is `<link rel="dns-prefetch">`.

2. *http* vs *https*.

“In January this year (2010), Gmail switched to using **HTTPS for everything by default** *[…] In our production frontend machines, SSL/TLS accounts for less than* **1% of the CPU** *load, less than* **10KB of memory** *per connection and less than* **2% of network overhead**.*”*

http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html

(emphasis mine)

SSL allows us to **upgrade** the protocol without breaking outdated proxies (eg: WebSocket).

3. Request headers

1. **Accept:**`text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`
2. **Accept-Charset:**`ISO-8859-1,utf-8;q=0.7,*;q=0.3`
3. **Accept-Encoding:**`gzip,deflate,sdch`
4. **Accept-Language:**`en-US,en;q=0.8`
5. **Cache-Control:**`max-age=0`
6. **Connection:**`keep-alive`
7. **Host:**`www.imperialviolet.org`
8. **If-Modified-Since:**`Tue, 06 Nov 2012 21:22:53 GMT`
9. **User-Agent:**`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.5 Safari/537.17`

Consider a character-by-character realtime editor (GDocs).

In the following example we send the character “**a**”.

Headers

1. **Accept:**`text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`
2. **Accept-Charset:**`ISO-8859-1,utf-8;q=0.7,*;q=0.3`
3. **Accept-Encoding:**`gzip,deflate,sdch`
4. **Accept-Language:**`en-US,en;q=0.8`
5. **Cache-Control:**`max-age=0`
6. **Connection:**`keep-alive`
7. **Host:**`www.myhost.com`
8. **User-Agent:**`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.5 Safari/537.17`

Body (JSON obj)

1. { “c”: “a” }

HTTP/1.1 overhead: **370~** bytes per character. Boo.

This only considers **sending** data.

How ‘bout **getting** data?

We **poll** the server through a **GET** request

When the server responds, we send **another** one.

And **another** one.

…and **another** one. **Boo**.

If the server has no data to send us, we would be generating a **LOT** of traffic.

Instead, we make the server **wait** a bit if it has no data.

This is what we call **long-polling**.

Pseudo-code time.

```
function send(data){
  var xhr = new XMLHttpRequest();
  xhr.open(‘POST’, ‘/’, false);
  xhr.send(data);
}
function get(fn) {
  var xhr = new XMLHttpRequest();
  xhr.open(‘GET’, ‘/’, false);
  xhr.send(data);
  xhr.onreadystatechange = function(){
    if (200 == xhr.status) {
      fn();   // send data to user
      get();  // restart poll
    }
  }
}
```

Most applications need to send data in a particular **order**.

For our example, the order that the user types.

Therefore, to replicate **socket** semantics we need to implement **buffering**. Boo.

For example, the user types **a** then **b c d** every 100ms.

If an existing **send()** is in-flight…

We accumulate the other data packets in an array.

Then we send that over when the server replies.

But instead of just sending one request per character.

We try to create a **data packet** with “b” “c” and “d” to minimize the **request headers traffic**.

Headers

1. **Accept:**`text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`
2. **Accept-Charset:**`ISO-8859-1,utf-8;q=0.7,*;q=0.3`
3. **Accept-Encoding:**`gzip,deflate,sdch`
4. **Accept-Language:**`en-US,en;q=0.8`
5. **Cache-Control:**`max-age=0`
6. **Connection:**`keep-alive`
7. **Host:**`www.myhost.com`
8. **User-Agent:**`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.5 Safari/537.17`

Body (JSON Array)

1. **[{ “c”: “b” }, { “c”: “c” }, { “c”: “d” }]**

This means, however, that we need a **protocol**.

A protocol on top of a protocol (HTTP).

Server and client need to agree that the data is not simply the **request body**, but **within** it.

This is a lot of work before you can focus on your app!

This has been a common trend in web development.

Need rounded corners?

Circa **2001**

```
<div class=”wrap-outer”>
  <div class=”wrap”>
    <div class=”wrap-inner”>
      <div class=”content”>
      </div>
    </div>
  </div>
</div>
```

```
.wrap-outer {
  background: …;
  padding: …;
}
.wrap {
  background: …;
  padding: …;
}
```

```
.wrap-inner {
  background: …;
  padding: …;
}
```

Now

```
<div class=”content”>
</div>
```

```
.content {
  border-radius: 3px;
}
```

For the realtime web…

Circa **2001**

- Data buffering on server within GET polls.
- Data buffering on client.
- JSON protocol or equivalent.
- XMLHttpRequest fallbacks
- Cross-domain tricks
- …

Now

- WebSocket!

Simpler client side API.

```
var ws = new WebSocket();
ws.onopen = function(){};
ws.onclose = function(){};
ws.onmessage = function(){};
```

Light-weight **protocol**.

Instead of sending many HTTP requests.

Browser sends one that contains a header:

Upgrade: WebSocket

The connection is then taken over.

Like TCP but unlike HTTP, it’s **bi-directional**.

We can send and receive over the same socket.

In terms of our realtime app example…

WebSocket overhead: **2-5~*** bytes per character.

* depending on spec

But WS is buggy on mobile, unavailable in older browsers, not understood by all middleware, breaks often without SSL.

Meet **engine.io**, the data transport engine of socket.io

```
var ws = new eio.Socket();
ws.onopen = function(){};
ws.onclose = function(){};
ws.onmessage = function(){};
```

Same API, long-polling first, **upgrades** to WS if possible.

The main advantage of WS is lightweight framing!

Most of the benefits that WS offers, like lower-bandwidth, would **still** benefit XMLHttpRequest.

Meet **SPDY**.

Headers overhead?

“SPDY compresses request and response HTTP headers, resulting in fewer packets and fewer bytes transmitted.”

http://www.chromium.org/spdy/spdy-whitepaper

TCP overhead?

“SPDY allows for unlimited concurrent streams over a single TCP connection.”

http://www.chromium.org/spdy/spdy-whitepaper

SPDY works **transparently** to your app.

SPDY assumes SSL.

Just upgrade your SSL terminator or load balancer and reap the benefits!

In other words, if you host your app on Heroku…

Keep doing what you’re doing, and wait for the upgrade.

But wait, many apps still need socket-like **semantics**.

I want to write “`socket.send(‘data’);`”
instead of “`new XMLHttpRequest`”

I’m not even sending **XML**!@!!@

Solution: **WebSocket** layering over SPDY/3.

“*With this protocol, WebSocket API get both benefits of usable bi-directional communication API, and a fast, secure, and scalable multiplexing mechanism on the top of SPDY*”

https://docs.google.com/document/d/1zUEFzz7NCls3Yms8hXxY4wGXJ3EEvoZc3GihrqPQcM0/edit#

Currently a draft.

SPDY/3 is currently experimental and available through *chrome://flags*.

Solution until the ideal world occurs: **engine.io**

Moving forward… Most applications are **not** data-packet oriented.

socket.io builds on top of engine.io to offer easy-to-use realtime **events** for everyone.

```
var socket = io.connect();
socket.on(‘character’, function(){
  // render!
});
socket.on(‘chat message’, function(){
  // render!
});
socket.emit(‘ready!’);
```

The goal is to not just focus on the speed of data **transfer** but also speed of **development**.

While ensuring the reliability that the data can be exchanged in the **fastest** way, in **every network**, on **every device**.

What’s next?

- **Raw TCP/UDP** sockets?

```
chrome.experimental.socket.create('tcp', '127.0.0.1', 8080, function(socketInfo) {
  chrome.experimental.socket.connect(socketInfo.socketId, function (result) {
        chrome.experimental.socket.write(socketInfo.socketId, "Hello, world!");
    });
});
```

For now, only for extensions.

- Removing the server out of the equation.

WebRTC **PeerConnection**.

Currently called “`webkitDeprecatedPeerConnection`”

Let’s be patient!

The **end**.