

CHAPTER 6



Using the HTML5 Web Sockets API

In this chapter, we'll explore what you can do with the most powerful communication feature in the HTML5 specification: *HTML5 Web Sockets*, which defines a full-duplex communication channel that operates through a single socket over the Web, is not just another incremental enhancement to conventional HTTP communications; it represents a colossal advance, especially for real-time, event-driven web applications.

HTML5 Web Sockets provide such a dramatic improvement from the old, convoluted "hacks" that are used to simulate a full-duplex connection in a browser that it prompted Google's Ian Hickson—the HTML5 specification lead—to say:

"Reducing kilobytes of data to 2 bytes...and reducing latency from 150ms to 50ms is far more than marginal. In fact, these two factors alone are enough to make Web Sockets seriously interesting to Google."

<http://www.ietf.org/mail-archive/web/hybi/current/msg00784.html>

We'll show you just why HTML5 Web Sockets provide such a dramatic improvement in detail and you'll see how—in one fell swoop—HTML5 Web Sockets makes all the hacky old Comet and Ajax polling, long-polling, and streaming solutions (that were used to create an illusion of real-time full-duplex networking in a browser) obsolete.

Web Sockets provide a proxy server and firewall-friendly socket connection over the Web. It uses an HTTP handshake to upgrade to the Web Socket protocol and therefore it can use the standard HTTP and HTTPS ports (80 and 443). We'll take a detailed look at how Web Sockets work with proxy servers, firewalls, and load balancers and the remainder of the chapter will show you how the Web Sockets API works and how it can be used to create real-time web applications.

Overview of HTML5 Web Sockets

Let's take a look at how HTML5 Web Sockets can offer such an incredibly dramatic reduction of unnecessary network traffic and latency by comparing it to conventional solutions and how full-duplex "real-time" browser communication was achieved in the past.

History of the Real-Time Web

Normally when a browser visits a web page, an HTTP request is sent to the web server that hosts that page. The web server acknowledges this request and sends back the response. In many cases—for example, for stock prices, news reports, ticket sales, traffic patterns, medical device readings, and so on—the response

could be stale by the time the browser renders the page. If you want to get the most up-to-date real-time information, you can constantly refresh that page manually, but that's obviously not a great solution.

Current attempts to provide real-time web applications largely revolve around polling and other server-side push technologies, the most notable of which is Comet, which delays the completion of an HTTP response to deliver messages to the client. Comet-based push is generally implemented in JavaScript and uses connection strategies such as long-polling or streaming.

With polling, the browser sends HTTP requests at regular intervals and immediately receives a response. This technique was the first attempt for the browser to deliver real-time information. Obviously, this is a good solution if the exact interval of message delivery is known, because you can synchronize the client request to occur only when information is available on the server. However, real-time data is often not that predictable, making unnecessary requests inevitable and as a result, many connections are opened and closed needlessly in low-message-rate situations.

With long-polling, the browser sends a request to the server and the server keeps the request open for a set period. If a notification is received within that period, a response containing the message is sent to the client. If a notification is not received within the set time period, the server sends a response to terminate the open request. It is important to understand, however, that when you have a high message volume, long-polling does not provide any substantial performance improvements over traditional polling. In fact, it could be worse, because the long-polling might spin out of control into an unthrottled, continuous loop of immediate polls.

With streaming, the browser sends a complete request, but the server sends and maintains an open response that is continuously updated and kept open indefinitely (or for a set period of time). The response is then updated whenever a message is ready to be sent, but the server never signals to complete the response, thus keeping the connection open to deliver future messages. However, since streaming is still encapsulated in HTTP, intervening firewalls and proxy servers may choose to buffer the response, increasing the latency of the message delivery. Therefore, many streaming Comet solutions fall back to long-polling in case a buffering proxy server is detected. Alternatively, TLS (SSL) connections can be used to shield the response from being buffered, but in that case the setup and tear down of each connection taxes the available server resources more heavily.

Ultimately, all of these methods for providing real-time data involve HTTP request and response headers, which contain lots of additional, unnecessary header data and introduce latency. On top of that, full-duplex connectivity requires more than just the downstream connection from server to client. In an effort to simulate full-duplex communication over half-duplex HTTP, many of today's solutions use two connections: one for the downstream and one for the upstream. The maintenance and coordination of these two connections introduces significant overhead in terms of resource consumption and adds lots of complexity. Simply put, HTTP wasn't designed for real-time, full-duplex communication as you can see in the Figure 5.1, which shows the complexities associated with building a Comet-style web application that displays real-time data from a back-end data source using a publish/subscribe model over half-duplex HTTP.

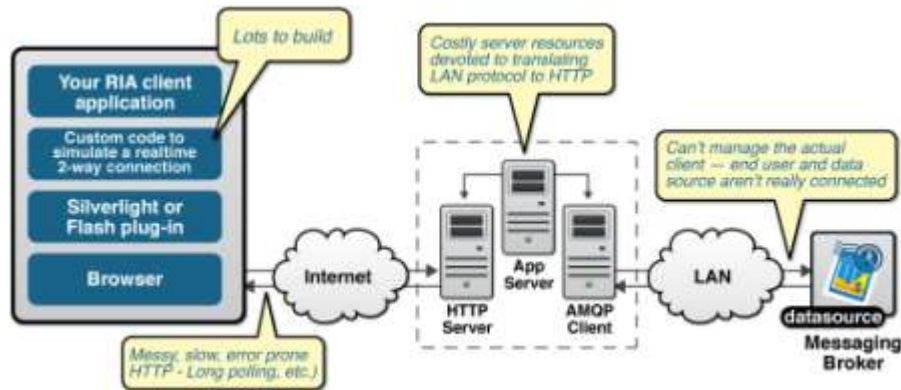


Figure 5.1. The complexity of Comet applications

It gets even worse when you try to scale out those Comet solutions to the masses. Simulating bi-directional browser communication over HTTP is error-prone and complex and all that complexity does not scale. Even though your end users might be enjoying something that looks like a real-time web application, this "real-time" experience has an outrageously high price tag. It's a price that you will pay in additional latency, unnecessary network traffic and a drag on CPU performance. HTML5 Web Sockets to the rescue!

Understanding HTML5 Web Sockets

HTML5 Web Sockets was first defined as "TCPConnection" in the Communications section of the HTML5 specification by Ian Hickson (lead writer of the HTML5 specification). The specification evolved and changed to Web Sockets, which is now an independent specification (just like Geolocation, Web Workers and so on, to keep the discussion focused). HTML5 Web Sockets represent the next evolution of web communications—a full-duplex, bidirectional communications channel that operates through a single socket over the Web. HTML5 Web Sockets provide a true standard that you can use to build scalable, real-time web applications. In addition, since it provides a socket that is native to the browser, it eliminates many of the problems Comet solutions are prone to. Web Sockets remove the overhead and dramatically reduces complexity.

What Do Web Sockets and Model Trains Have in Common?

Peter says: “Ian Hickson is quite the model train enthusiast; he has been planning ways to control trains from computers ever since 1984 when Marklin first came out with a digital controller, long before the Web even existed.

At the that time Ian added TCPConnection to the HTML5 specification, he was working on a program to control a model train set from a browser and he was using the prevalent pre-WebSocket “hanging GET” and XHR techniques to achieve browser to train communication. The train-controller program would have been a lot easier to build if there was a way to have socket communication in a browser—much like traditional asynchronous client/server communication model that is found in “fat” clients. So, inspired by what *could* be possible, the (train) wheels had been set in motion and the Web Sockets train had left the station. Next stop: the real-time Web”

The WebSocket Handshake

To establish a WebSocket connection, the client and server upgrade from the HTTP protocol to the Web Socket protocol during their initial handshake, as shown in Listing 5-1.

Listing 5-1. The WebSocket Upgrade handshake

From client to server:

```
GET /text HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: example.com
Origin: http://example.com
WebSocket-Protocol: sample
```

From server to client:

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://example.com
WebSocket-Location: ws://example.com/demo
WebSocket-Protocol: sample
```

Once established, WebSocket messages can be sent back and forth between the client and the server in full-duplex mode. This means that text-based messages can be sent full-duplex, in either direction at the same time. On the network each message starts with a 0x00 byte, ends with a 0xFF byte, and contains UTF-8 data in between.

The WebSocket Interface

The HTML5 initiative introduced the WebSocket interface, which defines a full-duplex communications channel that operates over a single socket and is exposed via a JavaScript interface in compliant browsers. Listing x-x shows the WebSocket interface.

Listing 5-2. The WebSocket interface

```
[Constructor(in DOMString url, in optional DOMString protocol)]
interface WebSocket {
  readonly attribute DOMString URL;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSED = 2;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  // networking
  attribute Function onopen;
  attribute Function onmessage;
  attribute Function onclose;
  boolean send(in DOMString data);
  void close();
};
WebSocket implements EventTarget;
```

Using the WebSocket interface couldn't be simpler. To connect to an end-point, just create a new WebSocket instance, providing the new object with a URL that represents the end-point to which you wish to connect. Note that a ws:// and wss:// prefix indicate a Web Sockets and a secure Web Sockets connection, respectively.

A WebSocket connection is established by upgrading from the HTTP protocol to the Web Socket protocol during the initial handshake between the client and the server, over the same underlying TCP/IP connection. Once established, WebSocket data frames can be sent back and forth between the client and the server in full-duplex mode. The connection itself is exposed via the onmessage and send methods defined by the WebSocket interface. In your code, you use asynchronous event listeners to handle each phase of the connection life cycle, much like you do in network socket programming.

```
myWebSocket.onopen = function(evt) { alert("Connection open ..."); };
myWebSocket.onmessage = function(evt) { alert("Received Message: " + evt.data); };
myWebSocket.onclose = function(evt) { alert("Connection closed."); };
```

A Dramatic Reduction in Unnecessary Network Traffic and Latency

So how dramatic is that reduction in unnecessary network traffic and latency? Let's compare a (pre-WebSocket) polling application and a WebSocket application side by side. As a polling example, I created a simple web application in which a web page requests real-time stock data from a RabbitMQ message broker

using a traditional publish/subscribe model. It does this by polling a Java Servlet that is hosted on a web server. The RabbitMQ message broker receives data from a fictitious stock price feed with continuously updating prices. The web page connects and subscribes to a specific stock channel (a topic on the message broker) and uses an XMLHttpRequest to poll for updates once per second. When updates are received, some calculations are performed and the stock data is shown in a table as shown in Figure 5-2.





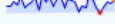


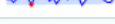

COMPANY	SYMBOL	PRICE	CHANGE	SPARKLINE	OPEN	LOW	HIGH
THE WALT DISNEY COMPANY	DIS	27.65	0.56		27.09	24.39	29.80
GARMIN LTD.	GRMN	35.14	0.35		34.79	31.31	38.27
SANBISK CORPORATION	SNOK	20.11	-0.13		20.24	18.22	22.26
GOODRICH CORPORATION	GR	49.99	-2.35		52.34	47.11	57.57
NVIDIA CORPORATION	NVDA	13.92	0.07		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

Figure 5-2. A JavaScript stock ticker application.

Note The back-end stock feed actually produces a lot of stock price updates per second, so using polling at one-second intervals is actually more prudent than using a Comet long-polling solution, which would result in a series of continuous polls. Polling effectively throttles the incoming updates here.

It all looks great, but a look under the hood reveals there are some serious issues with this application. For example, in Mozilla Firefox with Firebug, you can see that GET requests hammer the server at one-second intervals. Turning on the Live HTTP Headers add-on reveals the shocking amount of header overhead that is associated with each request. Listings x-x and x-x show the HTTP header data for just a single request and response.

Listing 5-3. HTTP request header

```
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

```

Listing 5-4. HTTP response header

```

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

```

Just for fun (ha!), I counted all the characters. The total HTTP request and response header information overhead contains 871 bytes and that does not even include any data. Of course, this is just an example and you can have less than 871 bytes of header data, but I have also seen cases where the header data exceeded 2,000 bytes. In this example application, the data for a typical stock topic message is only about 20 characters long. As you can see, it is effectively drowned out by the excessive header information, which was not even required in the first place.

So, what happens when you deploy this application to a large number of users? Let's take a look at the network throughput for just the HTTP request and response header data associated with this polling application in three different use cases.

- **Use case A:** 1,000 clients polling every second: Network throughput is $(871 \times 1,000) = 871,000$ bytes = 6,968,000 bits per second (6.6 Mbps)
- **Use case B:** 10,000 clients polling every second: Network throughput is $(871 \times 10,000) = 8,710,000$ bytes = 69,680,000 bits per second (66 Mbps)
- **Use case C:** 100,000 clients polling every 1 second: Network throughput is $(871 \times 100,000) = 87,100,000$ bytes = 696,800,000 bits per second (665 Mbps)

That's an enormous amount of unnecessary network throughput. If only we could just get the essential data over the wire. Well, guess what? You can with HTML5 Web Sockets. I rebuilt the application to use HTML5 Web Sockets, adding an event handler to the web page to asynchronously listen for stock update messages from the message broker (more on that in just a little bit). Each of these messages is a WebSocket frame that has just two bytes of overhead (instead of 871). Take a look at how that affects the network throughput overhead in our three use cases.

- **Use case A:** 1,000 clients receive 1 message per second: Network throughput is $(2 \times 1,000) = 2,000$ bytes = 16,000 bits per second (0.015 Mbps)

- **Use case B:** 10,000 clients receive 1 message per second: Network throughput is $(2 \times 10,000) = 20,000$ bytes = 160,000 bits per second (0.153 Kbps)
- Use case C: 100,000 clients receive 1 message per second: Network throughput is $(2 \times 100,000) = 200,000$ bytes = 1,600,000 bits per second (1.526 Kbps)

As you can see in Figure 5-3, HTML5 Web Sockets provide a dramatic reduction of unnecessary network traffic compared to the polling solution.

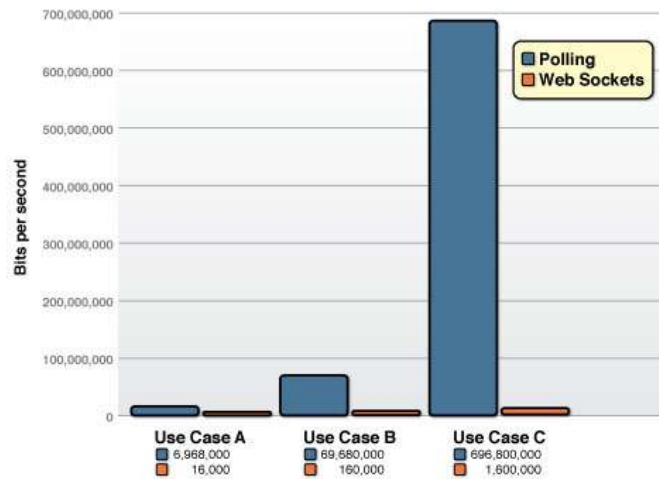


Figure 5-3. Comparison of the unnecessary network throughput overhead between the polling and the WebSocket applications.

And what about the reduction in latency? Take a look at Figure 5-4. In the top half, you can see the latency of the half-duplex polling solution. If we assume, for this example, that it takes 50 milliseconds for a message to travel from the server to the browser, then the polling application introduces a lot of extra latency, because a new request has to be sent to the server when the response is complete. This new request takes another 50ms and during this time the server cannot send any messages to the browser, resulting in additional server memory consumption.

In the bottom half of the figure, you see the reduction in latency provided by the WebSocket solution. Once the connection is upgraded to WebSocket, messages can flow from the server to the browser the moment they arrive. It still takes 50 ms for messages to travel from the server to the browser, but the WebSocket connection remains open so there is no need to send another request to the server.

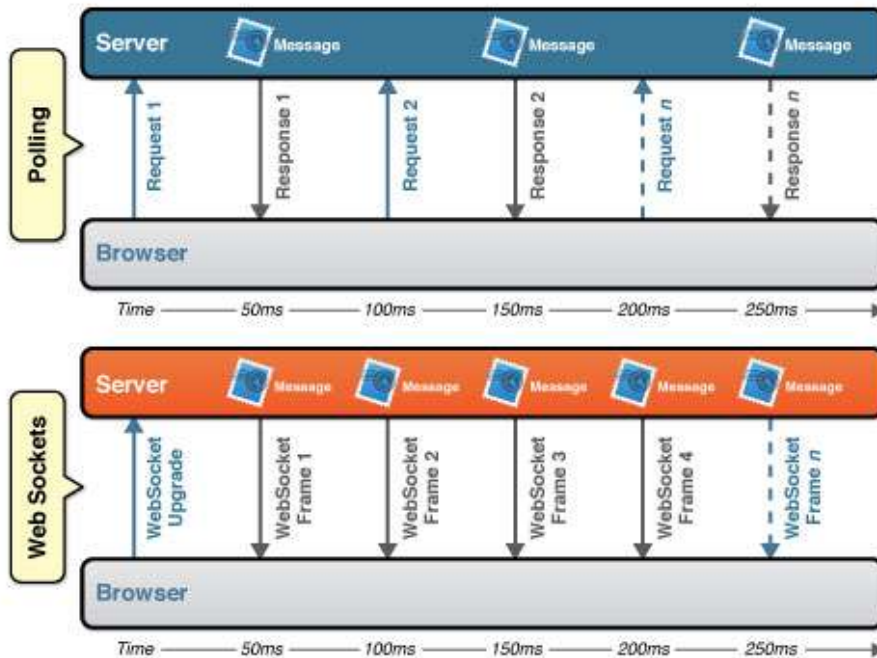


Figure 5-4. Latency comparison between the polling and WebSocket applications

HTML5 Web Sockets provides an enormous step forward in the scalability of the real-time web. As you have seen in this chapter, HTML5 Web Sockets can provide a 500:1 or—depending on the size of the HTTP headers—even a 1000:1 reduction in unnecessary HTTP header traffic and 3:1 reduction in latency. That is not just an incremental improvement; that’s revolutionary.

HTML5 Web Sockets and Proxy Servers

Will proxy servers automatically kill WebSocket connections? Do HTML5 Web Sockets deal with firewalls and proxy server issues better than Comet? Are Web Sockets the silver bullet in seamless proxy server traversal? In this section, we’ll look at how HTML5 Web Sockets work with proxy servers, load balancers, and firewalls.

One of the more unique features Web Socket provides is its ability to traverse firewalls and proxies, a problem area for many streaming Comet applications. This is why Comet-style applications typically employ long-polling—with all its inefficiencies and unnecessary overhead—as a rudimentary line of defense against firewalls and proxies. The technique is effective, but is not well suited for applications that have sub-500 millisecond latency or high throughput requirements. Plugin-based technologies such as Adobe Flash, also provide some level of socket support, but have long been burdened with the very proxy and firewall traversal problems that Web Sockets now resolve.

A proxy server is a server that acts as an intermediary between a client and another server (for example, a server on the Internet). Proxy servers are commonly used for content caching, Internet connectivity, security,

and enterprise content filtering. Typically, a proxy server is set up between a private network and the Internet. Proxy servers can monitor traffic and close a connection if it has been open for too long. The problem with proxy servers for web applications that have a long-lived connection (for example, Comet HTTP streaming or HTML5 Web Sockets) is clear: proxy servers can close connections at will. They may also buffer unencrypted HTTP responses, which introduces unpredictable latency during HTTP response streaming.

Let's see an example. A picture is worth a thousand words; in Figure 5-5 you see a simplified network topology in which people inside a corporate network are using a browser to access back-end TCP-based services using a full-duplex HTML5 WebSocket connection. The browsers are behind the corporate firewall and configured to access the Internet through a proxy server, which provides content caching, security, and perhaps even some corporate spying. Unlike regular HTTP traffic, which uses a request/response protocol, WebSocket connections can remain open for a long time. Proxy servers may allow this and handle it gracefully, but they can also throw a monkey wrench in the works, so let's take a look how HTML5 Web Sockets interact with proxy servers.

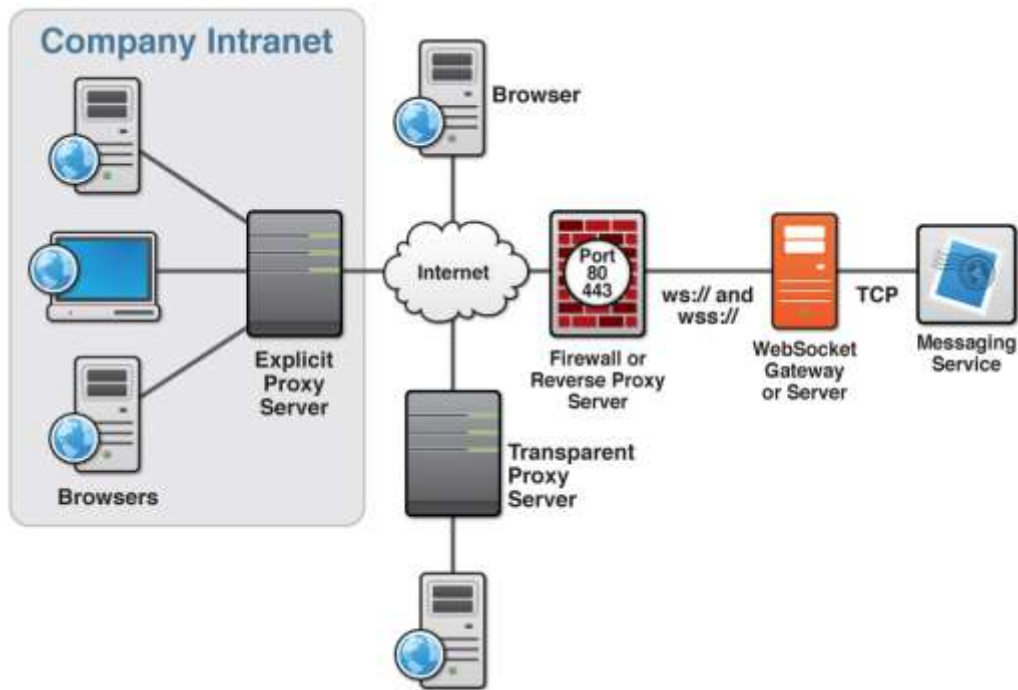


Figure 5-5. Standard WebSocket architecture with proxy servers

WebSocket connections use standard HTTP ports (80 and 443). Therefore, HTML5 Web Sockets do not require new hardware to be installed, or new ports to be opened on corporate networks—two things that would stop the adoption of the new protocol dead in its tracks. Without any intermediary servers (proxy or reverse proxy servers, firewalls, load-balancing routers and so on) between the browser and the WebSocket server, a WebSocket connection can be established smoothly, as long as both the server and the client understand the Web Socket protocol. However, in real environments, lots of network traffic is routed through intermediary servers. Let's take a look at how unencrypted WebSocket traffic differs from encrypted WebSocket traffic in the way it flows through proxy servers.

Unencrypted WebSocket Connections

The Web Socket protocol itself is unaware of proxy servers and firewalls; it just defines WebSocket upgrade handshake and the format for the WebSocket data frames. Let's take a look at unencrypted WebSocket traffic in two proxy server scenarios (explicit and transparent).

Unencrypted WebSocket Connections and Explicit Proxy Servers

If a browser is configured to use an explicit proxy server then it will first issue the HTTP CONNECT method to that proxy server while establishing the WebSocket connection. For example, to connect to the server example.com using the ws:// scheme (typically over port 80), the browser client sends the HTTP CONNECT method shown in Listing 5-x to the proxy server.

Listing 5-5. HTTP CONNECT method using port 80

```
CONNECT example.com:80 HTTP/1.1
Host: example.com
```

When the explicit proxy server allows the CONNECT method, the WebSocket connection upgrade handshake can be made and when that handshake succeeds, WebSocket traffic can start flowing unimpeded through the proxy server.

Unencrypted WebSocket Connections and Transparent Proxy Servers

In case the unencrypted WebSocket traffic flows through a transparent proxy on its way to the WebSocket server, the connection is likely to fail in practice, since the browser does not issue the CONNECT method in this case. When a proxy server forwards a request to the (WebSocket) server, it is expected to strip off certain headers, including the Connection header. Therefore, a well-behaved transparent proxy server will cause the WebSocket upgrade handshake to fail almost immediately.

Not all proxy servers conform to the HTTP standard regarding expected proxy behavior. For example, some proxy servers are configured not to remove the Connection: Upgrade header and pass it on to the WebSocket server, which in turn sends the 101 Web Socket Protocol Handshake response. Problems then arise when the client or the server starts sending the first WebSocket frame. Since the frame does not resemble anything the proxy server might expect (such as regular HTTP traffic), some exception will likely occur, unless the proxy server is specifically configured to handle WebSocket traffic.

Hop-By-Hop Upgrade

During the WebSocket handshake, the `Connection: Upgrade` header is sent to the WebSocket server. If the proxy server has to participate in the Upgrade mechanism, additional proxy server configuration would be required, because a hop-by-hop transport is used; the Upgrade sent from the proxy server to the browser is only good for that one hop, and the proxy server must send its own Upgrade header to handle the next hop from the proxy server to the WebSocket server (or to yet another intermediary server). In addition the proxy server must stop processing the request as HTTP.

Today, most transparent proxy servers will not yet be familiar with the Web Socket protocol and these proxy servers will be unable to support the Web Socket protocol. In the future, however, proxy servers will likely become Web Sockets-aware and able to properly handle and forward WebSocket traffic.

Encrypted WebSocket Connections

Now, let's take a look at encrypted WebSocket traffic in two proxy server scenarios (explicit and transparent).

Encrypted WebSocket Connections and Explicit Proxy Servers

If a browser is configured to use an explicit proxy server then it will first issue an HTTP `CONNECT` method to that proxy server while establishing the WebSocket connection. For example, to connect to the server `example.com` using the `wss://` scheme (typically over port 443), the browser client sends the HTTP `CONNECT` method shown in Listing 5-6 to the proxy server.

Listing 5-6. HTTP CONNECT method using port 443

```
CONNECT example.com:443 HTTP/1.1
Host: example.com
```

When the explicit proxy server allows the `CONNECT` method, the TLS handshake is sent, followed by the WebSocket connection upgrade handshake. After those handshakes succeed, WebSocket traffic can start flowing unimpeded through the proxy server. HTTPS work the same way; the use of encryption typically triggers the HTTP `CONNECT` method.

Encrypted WebSocket Connections and Transparent Proxy Servers

In the case of transparent proxy servers, the browser is unaware of the proxy server, so no HTTP `CONNECT` method is sent. However, since the wire traffic is encrypted, intermediate transparent proxy servers may simply allow the encrypted traffic through, so there is a much better chance that the WebSocket connection will succeed if Web Sockets Secure is used. Therefore, it is always best to use Web Sockets Secure with TLS encryption to connect to a WebSocket server, unless you're absolutely certain there are no intermediaries. While this has the added benefit of being more secure, TLS encryption does increase CPU consumption on both the client and the server, although this is usually not a dramatic increase and with hardware SSL/TLS acceleration, it can be reduced to near-zero.

Let's recap. Figure 5-6 shows the decisions that are made during the setup of a WebSocket connection between the browser and the WebSocket server. The figure shows the different connections scenarios in both the WebSocket (`ws://`) and WebSocket Secure (`wss://`) cases with explicit and transparent proxy servers.

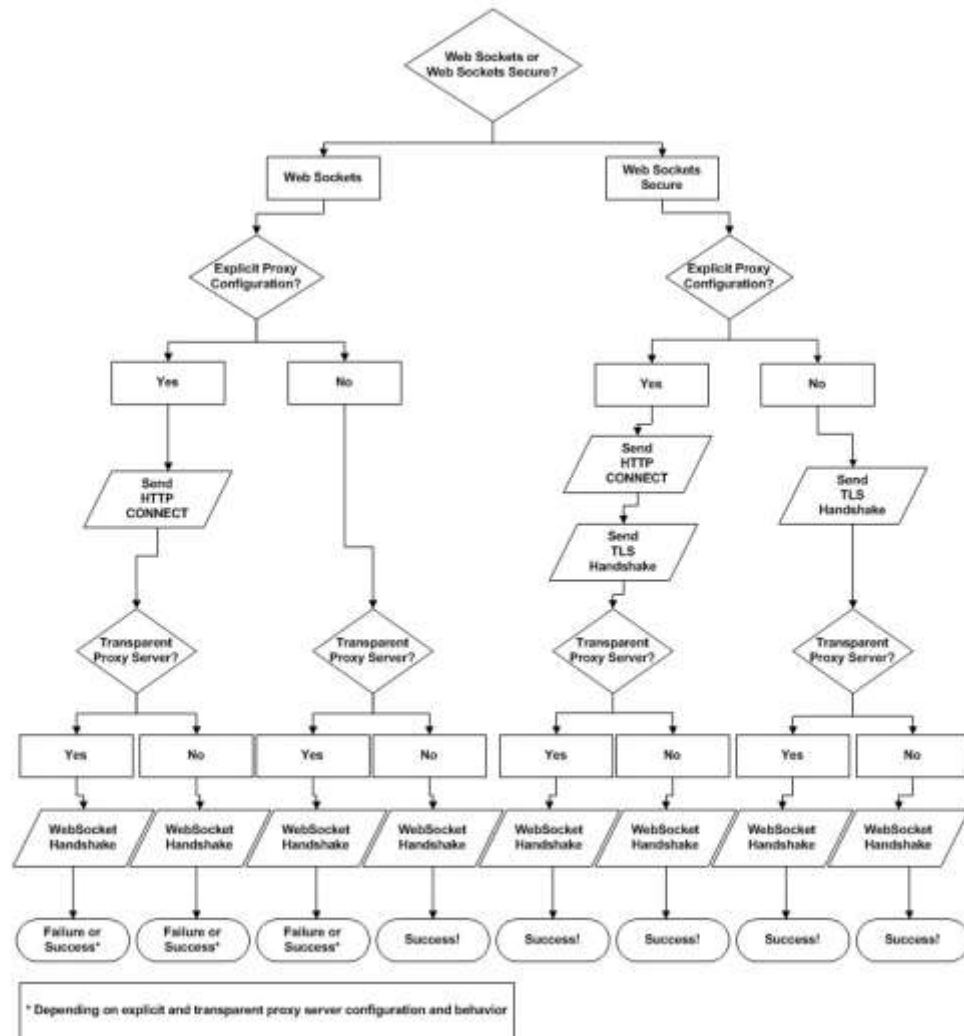


Figure 5-6. Proxy server traversal decision tree.

Figure 5-6 shows how using unencrypted WebSocket connections are more likely to fail in all but the simplest network topologies. It all comes down to understanding the end-to-end network topology you are deploying your WebSocket application in. Some HTTP proxy servers may restrict ports or allow access only to specific, authorized servers. A WebSocket server that is used in such a scenario must be added to the white list

of servers for connections to be successful. Typically, the decision about which scheme (`ws://` or `wss://`) to use to connect with has to be made up front by the client developer and is also based on the privacy characteristics of the WebSocket wire traffic. In the future, WebSocket servers may even be able to dynamically upgrade to Web Sockets Secure if an uncooperative proxy server is detected.

Browser Support for HTML5 Web Sockets

As shown in Table 5-1, HTML5 Web Sockets is supported and planned in various browsers at the time of this writing.

Table 5-1. Browser support for HTML5 Web Sockets

Browser	Details
Firefox	Not supported yet (planned for version 3.7.)
Safari	Not supported yet (supported in WebKit nightly builds)
Chrome	Supported in version 4+
Opera	Not supported yet
Internet Explorer	Not supported yet

Due to the varying levels of support, it is a good idea to first test if HTML5 Web Sockets are supported, before you use these elements. The section “*Checking for Browser Support*” later in this chapter will show you how you can programmatically check for browser support.

Writing a Simple Echo WebSocket Server

Before you can use the WebSocket API you need a server that supports Web Sockets. In this section we’ll take a look at how a simple WebSocket “echo” server is written. To run the examples for this chapter, we have included a simple WebSocket server written in Python. The sample code for the following examples is located in the `code/websockets` folder.

Existing WebSocket Servers

There are lots of Web Socket server implementations out there already and even more under development. The following are just a couple of the existing WebSocket servers:

1. **Kaazing WebSocket Gateway**—a Java-based WebSocket Gateway.
2. **mod_pywebsocket**—a Python-based extension for the Apache HTTP Server

Kaazing's WebSocket Gateway includes full client-side Web Sockets emulation support for browsers without native implementation of Web Sockets, which allows you to code against the Web Sockets API today and have your code work in all browsers.

To run the Python WebSocket echo server accepting connections at `ws://localhost:8000/echo`, open a command prompt, navigate to the folder that contains the file, and issue the following command:

```
python websocket.py
```

We have also included a *broadcast* server that accepts connections at `ws://localhost:8000/broadcast`. Contrary to the echo server, any WebSocket message sent to this particular server implementation will bounce back to *everyone* that is currently connected. It's a very simple way to broadcast messages to multiple listeners. To run the broadcast server, open a command prompt, navigate to the folder that contains the file, and issue the following command:

```
python broadcast.py
```

Both scripts make use of the Web Socket protocol library in `websocket.py`. You can add handlers for other paths that implement additional server- side behavior.

Note This is only a server for the Web Socket protocol and it cannot respond to HTTP requests. Because WebSocket connections begin with a subset of legal HTTP and rely on the Upgrade header, other servers can serve both WebSocket and HTTP on the same port.

Let's see what happens when a browser tries to communicate with this server. When the browser opens a connection, the server sends back the headers that finish the WebSocket handshake. A WebSocket handshake response must begin with *exactly* the line HTTP/1.1 101 Web Socket Protocol Handshake. In fact, the order and content of the handshake headers are more strictly defined than HTTP headers.

Note: If you are implementing a WebSocket server, you should refer to the protocol specification at the IETF at <http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol>.

```
def send_server_handshake(self, headers):
    self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
    self.send_bytes("Upgrade: WebSocket\r\n")
    self.send_bytes("Connection: Upgrade\r\n")
    self.send_bytes("WebSocket-Origin: %s\r\n" % headers["Origin"])
    self.send_bytes("WebSocket-Location: %s\r\n" % headers["Location"])

    if "Protocol" in headers:
```

```

self.send_bytes("WebSocket-Protocol: %s\r\n" % headers["Protocol"])

self.send_bytes("\r\n")

```

After the handshake, the client and server can send messages at any time. Each connection is represented on the server by a `WebSocketConnection` instance. The `WebSocketConnection`'s `send` function, shown below, transforms a string for the Web Socket protocol. The `0x00` and `0xFF` bytes surrounding the UTF-8 encoded string mark the frame boundary. In this server, each `WebSocket` connection is an `asyncore.dispatcher_with_send`, which is an asynchronous socket wrapper with support for buffered sends.

Note: There are many other asynchronous I/O frameworks for Python and other languages. `Asyncore` was chosen because it is included in the Python standard library.

`WebSocketConnection` inherits from `asyncore.dispatcher_with_send` and overrides the `send` method to UTF-8 encode strings and add `WebSocket` string framing.

```

def send(self, s):
    self.send_bytes("\x00")
    self.send_bytes(s.encode("UTF8"))
    self.send_bytes("\xFF")

```

Handlers for `WebSocketConnections` in `websocket.py` follow a simplified dispatcher interface. The handler's `dispatch()` method is called with the payload of each frame the connection receives. The `EchoHandler` sends back each message to the sender.

```

class EchoHandler(object):
    """
    The EchoHandler repeats each incoming string to the same WebSocket.
    """

    def __init__(self, conn):
        self.conn = conn

    def dispatch(self, data):
        self.conn.send("echo: " + data)

```

The basic broadcast server `broadcast.py` works in much the same way, but in this case, when the broadcast handler receives a string, it sends it back on all connected Web Sockets as shown in the following example.

```

class BroadcastHandler(object):
    """
    The BroadcastHandler repeats incoming strings to every connected
    WebSocket.
    """

```

```

def __init__(self, conn):
    self.conn = conn

def dispatch(self, data):
    for session in self.conn.server.sessions:
        session.send(data)

```

The handler in `broadcast.py` provides a lightweight message broadcaster that simply sends and receives strings. This is sufficient for the purposes of our example. Be aware that this broadcast service does not perform any input verification as would be desirable in a production message server. A production WebSocket server should, at the very least, verify the format of incoming data. As we see in the following chapter, another alternative is to use a robust messaging server and a protocol such as Stomp, XMPP, or AMQP to push data through the system in real-time.

For completeness, Listings 5-x and 6-x provide the complete code for `websocket.py` and `broadcast.py`.

Listing 5-7. complete code for websocket.py

```

#!/usr/bin/env python

import asyncio
import socket
import time

"""
Copyright (c) 2009-2010, Kaazing Corporation.
"""

class WebSocketConnection(asyncio.dispatcher_with_send):

    def __init__(self, conn, server):
        asyncio.dispatcher_with_send.__init__(self, conn)

        self.server = server
        self.server.sessions.append(self)
        self.readystate = "connecting"
        self.buffer = ""

    def handle_read(self):
        data = self.recv(1024)
        self.buffer += data
        if self.readystate == "connecting":
            self.parse_connecting()
        elif self.readystate == "open":
            self.parse_frame_type()

    def handle_close(self):
        self.server.sessions.remove(self)

```

```

self.close()

def parse_connecting(self):
    header_end = self.buffer.find("\r\n\r\n")
    if header_end == -1:
        return
    else:
        header = self.buffer[:header_end]
        # remove header and four bytes of line endings from buffer
        self.buffer = self.buffer[header_end+4:]
        header_lines = header.split("\r\n")
        headers = {}

        # validate HTTP request and construct location
        method, path, protocol = header_lines[0].split(" ")
        if method != "GET" or protocol != "HTTP/1.1" or path[0] != "/":
            self.terminate()
            return

        for line in header_lines[1:]:
            key, value = line.split(": ")
            headers[key] = value

        headers["Location"] = "ws://" + headers["Host"] + path

        self.readystate = "open"
        self.handler = self.server.handlers.get(path, None)(self)
        self.send_server_handshake(headers)

def terminate(self):
    self.ready_state = "closed"
    self.close()

def send_server_handshake(self, headers):
    self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
    self.send_bytes("Upgrade: WebSocket\r\n")
    self.send_bytes("Connection: Upgrade\r\n")
    self.send_bytes("WebSocket-Origin: %s\r\n" % headers["Origin"])
    self.send_bytes("WebSocket-Location: %s\r\n" % headers["Location"])

    if "Protocol" in headers:
        self.send_bytes("WebSocket-Protocol: %s\r\n" % headers["Protocol"])

    self.send_bytes("\r\n")

def parse_frametype(self):
    while len(self.buffer):
        type_byte = self.buffer[0]
        if type_byte == "\x00":

```

```

        if not self.parse_textframe():
            return

def parse_framelength(self):
    pass

def parse_textframe(self):
    terminator_index = self.buffer.find("\xFF")
    if terminator_index != -1:
        frame = self.buffer[1:terminator_index]
        self.buffer = self.buffer[terminator_index+1:]
        s = frame.decode("UTF8")
        self.handler.dispatch(s)
        return True
    else:
        # incomplete frame
        return false

def send(self, s):
    if self.readystate == "open":
        self.send_bytes("\x00")
        self.send_bytes(s.encode("UTF8"))
        self.send_bytes("\xFF")

def send_bytes(self, bytes):
    asyncio.dispatcher_with_send.send(self, bytes)

class EchoHandler(object):
    """
    The EchoHandler repeats each incoming string to the same WebSocket.
    """

    def __init__(self, conn):
        self.conn = conn

    def dispatch(self, data):
        self.conn.send("echo: " + data)

class WebSocketServer(asyncio.dispatcher):

    def __init__(self, port=80, handlers=None):
        asyncio.dispatcher.__init__(self)
        self.handlers = handlers
        self.sessions = []
        self.port = port
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()

```

```

        self.bind(("", port))
        self.listen(5)

    def handle_accept(self):
        conn, addr = self.accept()
        session = WebSocketConnection(conn, self)

if __name__ == "__main__":
    print "Starting WebSocket Server"
    WebSocketServer(port=8000, handlers={"/echo": EchoHandler})
    asyncore.loop()

```

Listing 5-8 complete code for broadcast.py

```

#!/usr/bin/env python

import asyncore
from websocket import WebSocketServer

class BroadcastHandler(object):
    """
    The BroadcastHandler repeats incoming strings to every connected
    WebSocket.
    """

    def __init__(self, conn):
        self.conn = conn

    def dispatch(self, data):
        for session in self.conn.server.sessions:
            session.send(data)

if __name__ == "__main__":
    print "Starting WebSocket broadcast server"
    WebSocketServer(port=8000, handlers={"/broadcast": BroadcastHandler})
    asyncore.loop()

```

Now that we've got a working echo server, we need to write the client side.

Using the HTML5 WebSocket API

In this section, we'll explore the use of HTML5 Web Sockets in more detail.

Checking for Browser Support

Before you use the HTML5 WebSocket API, you will want to make sure there is support in the browser for what you're about to do. This way, you can provide some alternate text, prompting the users of your application to upgrade to a more up-to-date browser. Listing 5-9 shows one way you can use to test for browser support.

Listing 5-9. Checking for browser support

```
function loadDemo() {
  if (window.WebSocket) {
    document.getElementById("support").innerHTML = "HTML5 Web Sockets is supported in your
      browser.";
  } else {
    document.getElementById("support").innerHTML = "HTML5 Web Sockets is not supported in
      your browser.";
  }
}
```

In this example, you test for browser support in the `loadDemo` function, which might be called when the application's page is loaded. A call to `window.WebSocket` will return the `WebSocket` object if it exists, or trigger the failure case if it does not. In this case, the page is updated to reflect whether there is browser support or not by updating a previously defined support element on the page with a suitable message. Another way to see if HTML5 Web Sockets is supported in your browser, you can also use the browser's console (Firebug or Chrome Developer Tools for example). Figure 5-7 shows how you can test whether Web Sockets is supported natively in Google Chrome (if it is not, the `window.WebSocket` command returns "undefined.")

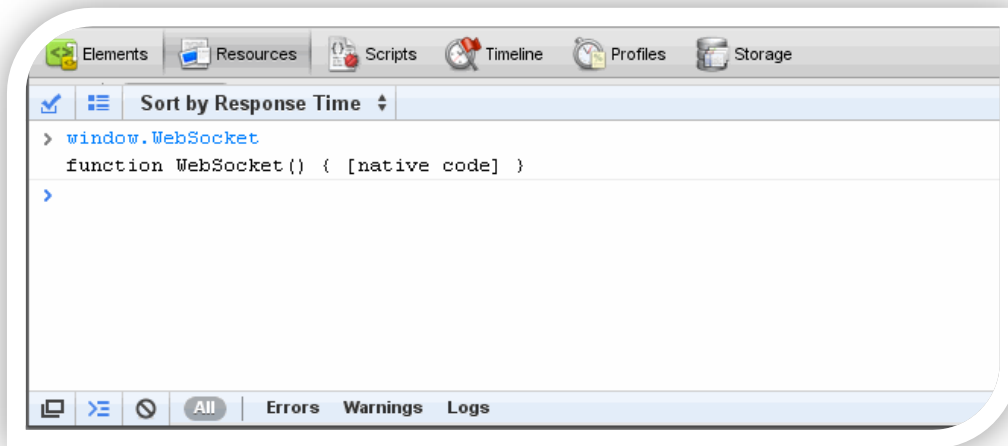


Figure 5-7. Testing Web Sockets support in Google Chrome Developer Tools

Basic API Usage

The sample code for the following examples is located in the `code/websockets` folder. This folder contains a `websocket.html` file and a `broadcast.html` file (and a `tracker.html` file used in the following section) as well as the `WebSocket` server code that can be run in Python.

Creating a WebSocket object and Connecting to a WebSocket Server

Using the WebSocket interface is straight-forward. To connect to an end-point, just create a new WebSocket instance, providing the new object with a URL that represents the end-point to which you wish to connect. You can use the `ws://` and `wss://` prefixes to indicate a Web Sockets and a Web Sockets Secure connection, respectively.

```
url = "ws://localhost:8000/echo";
w = new WebSocket(url);
```

Adding Event Listeners

WebSocket programming follows an asynchronous programming model; once you have an open socket, you simply wait for events. You don't have to actively poll the server anymore unnecessarily. To do this, you add callback functions to the WebSocket object to listen for events.

A WebSocket object dispatches three events: `open`, `close`, and `message`. The `open` event fires when a connection is established, the `message` event fires when messages are received, and the `close` event fires when the WebSocket connection is closed. As in most JavaScript APIs, there are corresponding callbacks (`onopen`, `onmessage`, and `onclose`) that are called when the events are dispatched.

```
w.onopen = function() {
  log("open");
  w.send("thank you for accepting this websocket request");
}
w.onmessage = function(e) {
  log(e.data);
}
w.onclose = function(e) {
  log("closed");
}
```

Sending Messages

While the socket is open (that is, after the `onopen` listener is called and before the `onclose` listener is called), you can use the `send` method to send messages. After sending the message, you can also call `close` to terminate the connection, but you can also leave the connection open.

```
document.getElementById("sendButton").onclick = function() {
  w.send(document.getElementById("inputMessage").value);
}
```

That's it. Bi-directional browser communication made simple. For completeness, Listing 5-10 shows the entire HTML page with the WebSocket code.

Listing 5-10. websocket.html code

```
<!DOCTYPE html>
<title>WebSocket Test Page</title>
```

```

<script>

var log;
log = function(s) {
  if (document.readyState !== "complete") {
    log.buffer.push(s);
  } else {
    document.getElementById("output").innerText += (s + "\n")
  }
}
log.buffer = [];

url = "ws://localhost:8000/echo";
w = new WebSocket(url);
w.onopen = function() {
  log("open");
  w.send("thank you for accepting this WebSocket request");
}
w.onmessage = function(e) {
  log(e.data);
}
w.onclose = function(e) {
  log("closed");
}

window.onload = function() {
  log(log.buffer.join("\n"))
  document.getElementById("sendButton").onclick = function() {
    w.send(document.getElementById("inputMessage").value);
  }
}
</script>

<input type="text" id="inputMessage" value="Hello, WebSocket!"><button id="sendButton">Send</button>
<pre id="output"></pre>

```

Running the WebSocket Page

To test the `websocket.html` page that contains the `WebSocket` code, open a command prompt, navigate to the folder that contains the `WebSocket` code, and issue the following command to host the HTML file:

```
python -m SimpleHTTPServer 9999
```

Next, open another command prompt, navigate to the folder that contains the `WebSocket` code, and issue the following command to run the Python `WebSocket` server:

```
python websocket.py
```

Finally, open a browser that supports Web Sockets natively and navigate to `http://localhost:9999/websocket.html`.

Figure 5-8 shows the web page in action.

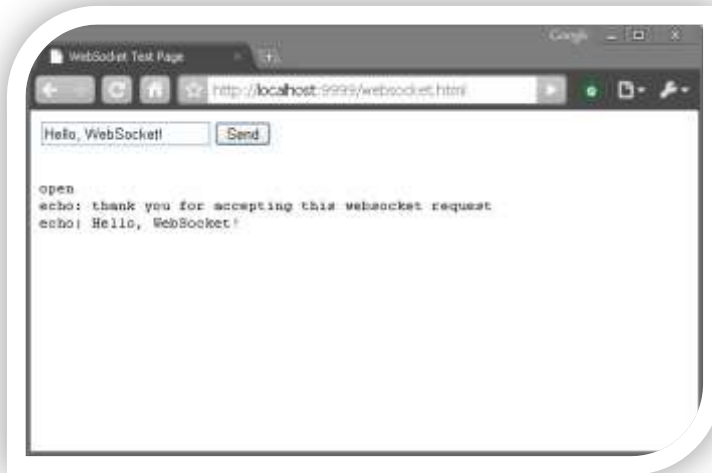


Figure 5-8. websocket.html in action

The example code folder also contains a web page that connects to the broadcast service that was created in the previous section. To see that action, close the command prompt that is running the WebSocket server and navigate to the folder that contains the WebSocket code, and issue the following command to run the python WebSocket server.

```
python broadcast.py
```

Open two separate browsers that supports Web Sockets natively and navigate (in each browser) to `http://localhost:9999/broadcast.html`.

Figure 5-9 shows the broadcast WebSocket server in action on two separate web pages.

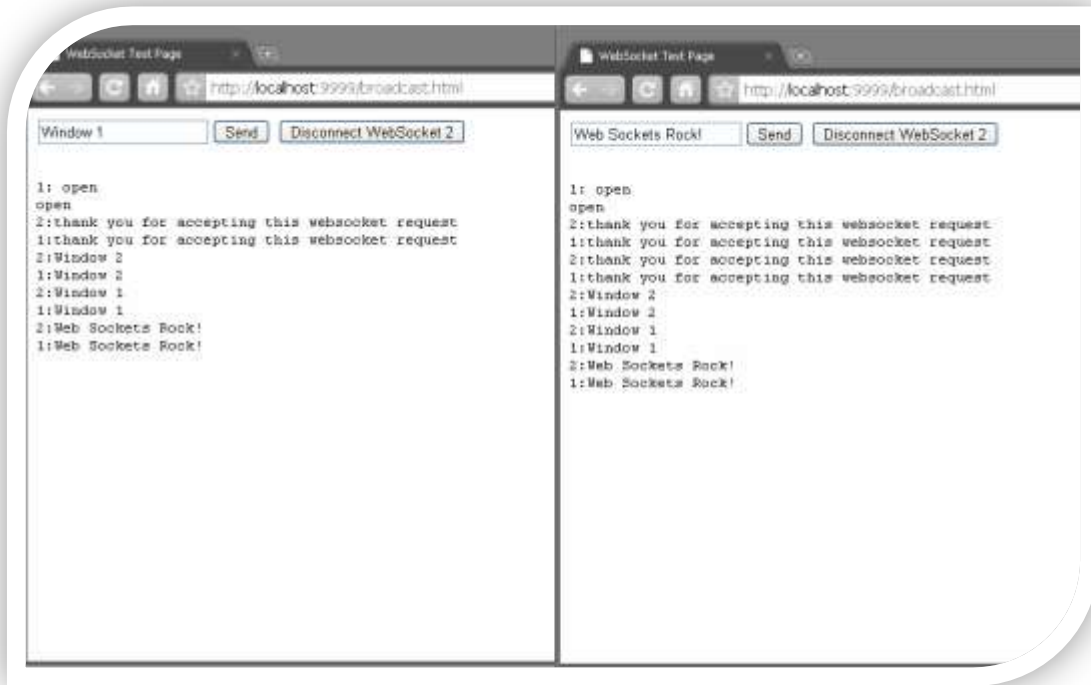


Figure 5-9. `broadcast.html` in action in two browsers

Building an Application with HTML5 Web Sockets

Now that we've seen the basics of Web Sockets, it's time to tackle something a little more substantial. Previously, we used the HTML5 Geolocation API to build an application that allowed us to calculate distance traveled directly inside our web page. We can utilize those same Geolocation techniques, mixed together with our new support for Web Sockets, and create a simple application that keeps multiple participants connected: a location tracker.

Note: We'll be using the broadcast WebSocket server described above, so if you aren't familiar with it you should consider taking some time to learn its basics.

In this application, we'll combine the Web Sockets and Geolocation support by determining our location and broadcasting it to all available listeners. Everyone who loads this application and connects to the same broadcast server will regularly send out their geographic location using the Web Socket. At the same time, the application will listen for any messages from the server and update in real-time display entries for everyone it hears about. In a race scenario, this sort of an application could keep runners informed of the location of all their competitors and prompt them to run faster (or slow down).

This tiny application does not include any personal information other than latitude and longitude location. Name, date of birth, and favorite ice cream flavor are kept strictly confidential.

You were warned!

Brian says: “This application is all about sharing your personal information. Granted, only a location is shared. However, if you (or your users) didn’t understand the browser warning that was offered when the Geolocation API was first accessed, this application should be a stark lesson in how easy it will be to transmit sensitive data to remote locations. Make sure your users understand the consequences of agreeing to submit location data.

When in doubt, go above and beyond in your application to let the user know how their sensitive data can be used. Make opting out the easiest path of action.”

But that’s enough warnings... Let’s dig into the code. As always, the entire code sample is located online for your perusal. We’ll focus on the most important parts here. The finished application will look like Figure 5-10. Although ideally, this would be enhance by overlaying it on a map.



Figure 5-10. The Location Tracker application

Coding the HTML File

The HTML markup for this application will be kept deliberately simple so that we can focus on the data at hand. How simple?

```
<body onload="loadDemo()">

<h1>HTML5 WebSocket / Geolocation Tracker</h1>

<div><strong>Geolocation</strong>: <p id="geoStatus">HTML5 Geolocation is <strong>not</strong>
supported in your browser.</p></div>
<div><strong>WebSocket</strong>: <p id="socketStatus">WebSockets are <strong>not</strong>
supported in your browser.</p></div>

</body>
```

Simple enough that we only include a title and a few status areas: one status area for Geolocation updates, and another to log any WebSocket activity. The actual visuals for location data will be inserted into the page as messages are received in real-time.

By default, our status messages indicate that a viewer's browser does not support either Geolocation or Web Sockets. Once we detect support for the two HTML5 technologies, we'll update the status with something a little friendlier.

```
<script type="text/javascript">

// reference to the Web Socket
var socket;

// a semi-unique random ID for this session
var myId = Math.floor(100000*Math.random());

// number of rows of data presently displayed
var rowCount = 0;
```

The meat of this application is once again accomplished via the script code. First we will establish a few variables:

- A global reference to our socket so that functions have easy access to it later.
- A random myId number between 0 and 100,000 to identify our location data online. This number is merely used to correlate changes in location over time back to the same source without using more personal information such as names. A sufficiently large pool of numbers makes it unlikely that more than one user will have the same identifier.
- A rowCount which holds how many unique users have transmitted their location data to us. This is largely used for visual formatting.

The next two functions should look familiar. As in other example applications, we've provided utilities to help us update our status message. This time, there are two status messages to update.

```
function updateSocketStatus(message) {
    document.getElementById("socketStatus").innerHTML = message;
}

function updateGeolocationStatus(message) {
    document.getElementById("geoStatus").innerHTML = message;
}
```

It is always helpful to include a user-friendly set of error messages whenever something goes wrong with location retrieval. If you need more information on the error codes, consult Chapter 4.

```
function handleLocationError(error) {
    switch(error.code)
    {
        case 0:
            updateGeolocationStatus("There was an error while retrieving your location: " +
                error.message);
            break;
        case 1:
            updateGeolocationStatus("The user prevented this page from retrieving a
                location.");
            break;
        case 2:
            updateGeolocationStatus("The browser was unable to determine your location: " +
                error.message);
            break;
        case 3:
            updateGeolocationStatus("The browser timed out before retrieving the location.");
            break;
    }
}
```

Adding the WebSocket Code

Now let's examine something more substantial. The `loadDemo` function is called on the initial load of our page, making it the starting point of the application.

```
function loadDemo() {
    // test to make sure that sockets are supported
    if (window.WebSocket) {

        // the location where our broadcast WebSocket server is located
        url = "ws://websockets.org:8787";
        socket = new WebSocket(url);
        socket.onopen = function() {
```

```

        updateSocketStatus("Connected to WebSocket tracker server");
    }
    socket.onmessage = function(e) {
        updateSocketStatus("Updated location from " + dataReturned(e.data));
    }
}

```

The first thing we do here is set up our Web Socket connection. As with any HTML5 technology, it is wise to check for support before jumping right in, so we test to make sure that `window.WebSocket` is a supported object in this browser.

Once that is verified, we make a connection to the remote broadcast server using the connect string format described above. This connect string can point to your own broadcast WebSocket server, but we've pointed it to our default broadcast server location for now. The connection is stored in our globally declared `socket` variable.

Finally, we declare two handlers to take action when our Web Socket receives updates. The `onopen` handler will merely update the status message to let the user know that we made a successful connection. The `onmessage` will similarly update the status to let the user know that a message has arrived. It will also call our upcoming `dataReturned` function to show the arriving data in the page, but we'll tackle that later.

Adding the Geolocation Code

The next section should be familiar to you from Chapter 4. Here, we verify support for the Geolocation service and update the status message appropriately.

```

var geolocation;
if(navigator.geolocation) {
    geolocation = navigator.geolocation;
    updateGeolocationStatus("HTML5 Geolocation is supported in your browser.");
}
else {
    geolocation = google.gears.factory.create('beta.geolocation');
    updateGeolocationStatus("Geolocation is supported via Google Gears");
}

// register for position updates using the Geolocation API
geolocation.watchPosition(updateLocation,
    handleLocationError,
    {maximumAge:20000});
}

```

Note: We've also included support for the Google Gears plugin's Geolocation API. Once loaded, it works identically to the built-in browser support, but it will eventually be replaced by native browser code.

As before, we watch our current location for changes and register that we want the `updateLocation` function called when they occur. Errors are sent to the `handleLocationError` function, and the location data is set to expire every twenty seconds.

The next section of code is the handler which is called by the browser whenever a new location is available.

```
function updateLocation(position) {
  var latitude = position.coords.latitude;
  var longitude = position.coords.longitude;
  var timestamp = position.timestamp;

  updateGeolocationStatus("Location updated at " + timestamp);

  // Schedule a message to send my location via WebSocket
  var toSend = ";" + myId + ";" + latitude + ";" + longitude;
  setTimeout("sendMyLocation('" + toSend + "')", 1000);
}
```

This section is similar to, but simpler than the same handler in Chapter 4. Here, we grab the latitude, longitude, and timestamp from the position provided by the browser. Then, we update the status message to indicate that a new value has arrived.

Putting It All Together

The final section calculates a message string to send to the remote broadcast WebSocket server. The string here will be of the form:

```
<id>;<latitude>;<longitude>
```

The ID will be the randomly calculated value already created to identify this user. The latitude and longitude are provided by the geolocation position object. Rather than sending the message string directly to the server, we will use the `setTimeout()` call to wait a second our position to the server.

Note: Waiting a short time via a timeout allows the geolocation code to finish without impacting the socket transmission. Plugins may be at play here, and they are fickle beasts.

The actual code to send the position to the server resides in the `sendMyLocation()` function.

```
function sendMyLocation(newLocation) {
  if (socket) {
    socket.send(newLocation);
  }
}
```

If a socket was successfully created—and stored for later access—then it is safe to send the message string passed into this function to the server. Once it arrives, the WebSocket message broadcast server will distribute the location string to every browser currently connected and listening for messages. Everyone will know where you are. Or, at least, a largely anonymous “you” identified only by a random number.

Now that we're sending messages, let's see how those same messages should be processed when they arrive at the browser. Recall that we registered an onmessage handler on the socket to pass any incoming data to a dataReturned() function. Next, we will look at that final function in more detail.

```
function dataReturned(locationData) {
  // break the data into ID, latitude, and longitude
  var allData = locationData.split(";");
  var incomingId = allData[1];
  var incomingLat = allData[2];
  var incomingLong = allData[3];
```

The dataReturned function serves two purposes. It will create (or update) a display element in the page showing the position reflected in the incoming message string, and it will return a text representation of the user this message originated from. The user name will be used in the status message at the top of the page by the calling function, the socket.onmessage handler.

The first step taken by this data handler function is to break the incoming message back down into its component parts using the string split() utility. Although a more robust application would need to check for unexpected formatting, we will assume that all messages to our server are of the form described above, and therefore our string separates cleanly into a random ID, a latitude, and a longitude, separated by semicolons.

```
// locate the HTML element for this ID
// if one doesn't exist, create it
var incomingRow = document.getElementById(incomingId);
if (!incomingRow) {
  incomingRow = document.createElement('div');
  incomingRow.setAttribute('id', incomingId);
```

Our demonstration user interface will create a visible <div> for every random ID for which it receives a message. This includes the user's ID itself; in other words, the user's own data will also be displayed only after it is sent and returned from the web socket broadcast server.

Accordingly, the first thing we do with the ID from our message string is use it to locate the display row element matching it. If one does not exist, we create one and set its id attribute to be the id returned from our socket server for future retrieval.

```
incomingRow.textContent = (incomingId == myId) ?
  'Me' :
  'User ' + rowCount;

incomingRow.setAttribute('class', (rowCount % 2 == 0) ? 'trackerBlue'
  : 'trackerOrange');

rowCount++;
```

The user text to be displayed in the data row is easy to calculate. If the ID matches the user's ID, it is simply 'Me'. Otherwise, the user name is a combination of a common string and a count of rows, which we will increment. For stylistic purposes, we will switch between two background colors based on whether or not the row is even or odd. Even simple demos should maintain a sense of style.

```
document.body.appendChild(incomingRow);
}
```

Once the new display element is ready, it is inserted into the end of the page. No matter if the display element is newly created or if it already existed—due to the fact that a location update was not the first for that particular user—the display row needs to be updated with the current text information.

```
// update the row text with the new values
incomingRow.innerHTML = incomingRow.userText + "\\ Lat: " +
    incomingLat + " \\ Lon: " +
    incomingLong;

return incomingRow.userText;
}
```

In our case, we will separate the user text name from the latitude and longitude values using a backslash (properly escaped, of course). Finally, the display name is returned to the calling function for updating the status row.

Our simple WebSocket and Geolocation mashup is now complete. Try it out, but keep in mind that unless there are multiple browsers accessing the application at the same time, you won't see many updates. As an exercise to the reader, consider updating this example to display the incoming locations on a global Google Map to get an idea of where HTML5 interest is flourishing at this very moment.

The Final Code

For completeness, the Listing 5-11 provides the entire tracker.html file.

Listing 5-11. The tracker.html code

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
<title>HTML5 WebSocket / Geolocation Tracker</title>
<style type="text/css">
@import url("styles.css");
</style>
<script src="gears_init.js"></script>
</head>

<body onload="loadDemo()">

<h1>HTML5 WebSocket / Geolocation Tracker</h1>

<div><strong>Geolocation</strong>: <p id="geoStatus">HTML5 Geolocation is <strong>not</strong>
supported in your browser.</p></div>
<div><strong>WebSocket</strong>: <p id="socketStatus">WebSockets are <strong>not</strong>
supported in your browser.</p></div>

<script type="text/javascript">
```

```

// reference to the Web Socket
var socket;

// a semi-unique random ID for this session
var myId = Math.floor(100000*Math.random());

// number of rows of data presently displayed
var rowCount = 0;

function updateSocketStatus(message) {
    document.getElementById("socketStatus").innerHTML = message;
}

function updateGeolocationStatus(message) {
    document.getElementById("geoStatus").innerHTML = message;
}

function handleLocationError(error) {
    switch(error.code)
    {
        case 0:
            updateGeolocationStatus("There was an error while retrieving your location: " +
                error.message);
            break;
        case 1:
            updateGeolocationStatus("The user prevented this page from retrieving a
                location.");
            break;
        case 2:
            updateGeolocationStatus("The browser was unable to determine your location: " +
                error.message);
            break;
        case 3:
            updateGeolocationStatus("The browser timed out before retrieving the location.");
            break;
    }
}

function loadDemo() {
    // test to make sure that sockets are supported
    if (window.WebSocket) {

        // the location where our broadcast WebSocket server is located
        url = "ws://websockets.org:8787";
        socket = new WebSocket(url);
        socket.onopen = function() {
            updateSocketStatus("Connected to WebSocket tracker server");
        }
    }
}

```

```

    socket.onmessage = function(e) {
        updateSocketStatus("Updated location from " + dataReturned(e.data));
    }
}

var geolocation;
if(navigator.geolocation) {
    geolocation = navigator.geolocation;
    updateGeolocationStatus("HTML5 Geolocation is supported in your browser.");
}
else {
    geolocation = google.gears.factory.create('beta.geolocation');
    updateGeolocationStatus("Geolocation is supported via Google Gears");
}

// register for position updates using the Geolocation API
geolocation.watchPosition(updateLocation,
    handleLocationError,
    {maximumAge:20000});
}

function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var timestamp = position.timestamp;

    updateGeolocationStatus("Location updated at " + timestamp);

    // Schedule a message to send my location via WebSocket
    var toSend = ";" + myId + ";" + latitude + ";" + longitude;
    setTimeout("sendMyLocation('" + toSend + "')", 1000);
}

function sendMyLocation(newLocation) {
    if (socket) {
        socket.send(newLocation);
    }
}

function dataReturned(locationData) {
    // break the data into ID, latitude, and longitude
    var allData = locationData.split(";");
    var incomingId = allData[1];
    var incomingLat = allData[2];
    var incomingLong = allData[3];

    // locate the HTML element for this ID
    // if one doesn't exist, create it
    var incomingRow = document.getElementById(incomingId);

```

```

if (!incomingRow) {
    incomingRow = document.createElement('div');
    incomingRow.setAttribute('id', incomingId);

    incomingRow.userText = (incomingId == myId) ?
        'Me' :
        'User ' + rowCount;

    incomingRow.setAttribute('class', (rowCount % 2 == 0) ? 'trackerBlue' :
        'trackerOrange');
    rowCount++;

    document.body.appendChild(incomingRow);
}

// update the row text with the new values
incomingRow.innerHTML = incomingRow.userText + "\\ Lat: " +
    incomingLat + "\\ Lon: " +
    incomingLong;

return incomingRow.userText;
}
</script>
</body>
</html>

```

Summary

In this chapter, you have seen how HTML5 Web Sockets can be used to create compelling, real-time applications. First we showed that HTML5 Web Sockets defines a revolutionary, full-duplex communication channel that operates through a single socket over the Web.

We showed you how HTML5 Web Sockets provide such a dramatic improvement by comparing it to Comet and Ajax polling, long-polling, and streaming solutions that were previously used to create an illusion of real-time full-duplex networking in a browser and may soon be obsolete.

We went on to describe how Web Sockets work with proxy servers and showed you how you can use the WebSocket API to create real-time web applications.

In the next chapter, we'll demonstrate more ways that HTML5 lets you...<<**TODO: Add information about next chapter**>>