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

HTML5 WebSockets provide such an 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 WebSockets seriously interesting to Google.”

—www.ietf.org/mail-archive/web/hybi/current/msg00784.html

We'll show you in detail just why HTML5 WebSockets provide such a dramatic improvement and you’ll see how—in one fell swoop—HTML5 WebSockets makes all the old Comet and Ajax polling, long-polling, and streaming solutions obsolete.

**Overview of HTML5 WebSockets**

Let's take a look at how HTML5 WebSockets can offer a reduction of unnecessary network traffic and latency by comparing HTTP solutions to full duplex "real time" browser communication with WebSockets.

**Real-Time and HTTP**

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.

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 of time. 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.

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 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 web application that displays real-time data from a back-end data source using a publish/subscribe model over half-duplex HTTP.