At the Forge

Real-Time Messaging

Reuven M. Lerner

Abstract

Want to send messages to all the browsers connected to your site? The pub-sub paradigm, run through Web sockets, might be just the solution.

Back in the 1980s, BSD UNIX introduced the idea of a "socket", a data structure that functioned similarly to a file handle, in that you could read from it or write to it. But, whereas a file handle allows a program to work with a file, a socket is connected to another process—perhaps on the same computer, but quite possibly running on another one, somewhere else on the Internet. Sockets brought about a communications revolution, in no small part because they made it easy to write programs that communicated across the network.

Today, we take that for granted. Dozens or hundreds of sockets are open on my computer at any given moment, and I don't know if they're communicating with local or remote programs. But, that's just the point—it's so easy to work with sockets, we no longer think of networked programs as anything special or unusual. The people who created sockets couldn't possibly have imagined the wide variety of protocols, applications and businesses that were built using their invention.

My point is not to praise sockets, but to point out that the inventors of a technology, particularly one that provides infrastructural support and a new abstraction layer, cannot know in advance how it'll be used.

In that light, consider a new network communication protocol called Web sockets, part of the standards known collectively as HTML5. To me, at least, Web sockets are the most undersold, least discussed parts of the HTML5 suite, with the potential to transform Web browsers into a fully fledged application platform.

Web sockets don't replace HTTP. Rather, much like BSD sockets, they provide bidirectional, long-term communication between two computers. The "bidirectional" and "long-term" aspects distinguish Web sockets from HTTP, in which the client sends a request, the server sends a response, and then the connection is terminated. Setting up a Web socket has very little overhead—and once communication is established, it can continue indefinitely.

Now that Web sockets exist, and are even supported by a growing number of browsers, what can you do with them? That question is still hard to answer, in no small part because Web sockets are so new. After all, if you had asked someone in the 1980s what you could do with BSD sockets, it's unlikely that streaming video would have come to mind.

That said, there are some applications for which Web sockets are already showing their advantage. In particular, applications that benefit from real-time data updates, such as stock-market tickers, now can receive a steady stream of data, rather than perform repeated Ajax calls to a server. Real-time chat systems are another example of where Web sockets can shine, and where HTTP has not performed adequately. Indeed, any Web application that handles or displays a constant flow of data can benefit from Web sockets.

But you can go even farther than that. Remember, Web sockets provide communication between a single server and a single client. There are, however, numerous applications in which the server might want to "broadcast" information to a large number of clients simultaneously. You can imagine how this could work with Web sockets, creating a Web socket connection between a server and each of the clients, and then sending messages to each of the clients, perhaps by iterating over the array of Web sockets and invoking send() on each one.

This is certainly possible, but implementing such a system yourself would be time-consuming and difficult, and might not scale easily. Fortunately, now there are third-party services that (for a fee) will handle such connections for you. Such publish-subscribe ("pub-sub") systems make it possible for a server to send to any number of clients almost simultaneously, opening the door to all sorts of Web applications.

In this article, I review the basics behind Web sockets and then move forward to demonstrate a simple application that uses the pub-sub paradigm. Even if you don't currently need this sort of functionality in your Web application, I have no doubt you'll eventually encounter a situation that can benefit from it. When the time comes, you'll hopefully realize that it's not too difficult to put it into place.

Working with Web Sockets

Web sockets, as with the rest of the HTML5 standard, have to do with programming within the browser—which, of course, happens in JavaScript or a language that compiles into JavaScript. To create a new Web socket, you simply say:

```
var ws = new WebSocket("ws://lerner.co.il/socket");
```

The beauty of this API is its simplicity. I don't know about you, but I'm tired of protocols that expect me to remember which parameter represents the hostname, which the protocol and which the port (if any). In the case of Web sockets, as you would expect from a Web standard, you pass all of that along in a URL whose protocol is defined as "ws" or "wss" (for SSL-encrypted Web sockets). Also notice that you don't have to define the Web socket as being read-only, write-only or read/write; Web sockets are all bidirectional.

You can send data to the other side of your Web socket by invoking the "send" method:

```
ws.send("Hello");
```

Or, if you want to send something a bit more complicated, it's typical to use JSON:

```
var stuff_to_send = {a:1, b:2}; ws.send(JSON.stringify(stuff_to_send));
```

What happens when your Web socket receives some data? Nothing just yet. You have to tell the browser that each time it receives data, it should do something with that data, such as display it on the screen. You tell it what to do with a callback, as you might expect in a functional language, such as JavaScript. That is, you tell the Web socket that when it receives data, it should execute a function, passing the received data as a parameter to that function. A simple example might be: *Garrick, shrink below.*

```
ws.onmessage = function(message) {  alert("Received ws message: '" + message.data + "'"); };
```

You also can do something more interesting and exciting with the data:

```
ws.onmessage = function(message) {  $("#wsdata").html(message.data); };
```

Of course, the incoming data isn't necessarily a string. Instead, it might be a bunch of JSON, in which case, it might contain a JavaScript object with fields. In such a case, you could say:

```
ws.onmessage = function(message) {  parsed_message = JSON.parse(message)  $("#one").html(parsed_message.one);  $("#on
```

Now, it's important to remember that the Web sockets protocol is a distinct protocol from HTTP. This means that when I say I want to connect to ws://lerner.co.il/socket, I need to be sure I'm running a Web socket server on lerner.co.il that responds to items at that URL. This is not the same thing as Apache, nginx or whatever your favorite HTTP server is.

So, when I say here that your browser connects to a server, you need to provide such a server. The Resources section of this article describes a number of systems that make it possible and fairly straightforward to create a server for Web sockets.

Pub-Sub

As you can see, working with Web sockets is fairly straightforward. But, what happens if you want to send messages to multiple clients? For example, let's say your company deals with stocks, and you want the home page of your company's Web site to show the latest value of certain stocks and stock indexes, updated continuously.

The simplest and seemingly most straightforward way is to use the strategy I described above—namely, that the server can store its Web sockets in an array (or similar data structure). At a set interval, the server then can execute **`ws.send()`** to each of the clients, either sending a simple text string or a JSON data structure with one or more pieces of information. The client, upon receiving this data, then executes the **`onmessage`** callback function, which then updates the user's browser accordingly.

This approach has a number of problems, but the main one that bothers me is the lack of a real abstraction layer. As application developers, you want to send the message, rather than consider how the message is being sent, or even who is receiving it. This is one way of looking at the publish-subscribe (pub-sub) design pattern. The publisher and subscriber aren't connected to each other directly, but rather through a middleman object or server that takes care of the connections. When the publisher wants to send a message, it does so through the broker, which then uses the existing Web socket connection to send a message to each client.

Now, this might sound something like a message queue, which I described about a year ago in this space. But message queues and pub-sub systems work quite differently from each other, and they are used for different purposes.

You can think of a message queue as working something like e-mail, with a single sender and a single recipient. Until the recipient retrieves the message, it waits in the message queue. The order and timing in which messages appear isn't necessarily guaranteed, but the delivery of the message is.

By contrast, a pub-sub system is something like a group IM chat. Everyone who is connected to the pub-sub system and is subscribing to a particular channel, receives the messages sent to that channel. If you happen not to be subscribed at the time a message is sent, you won't receive it; there is no mechanism in pub-sub for storing or replaying messages.

If you were interested in giving people real-time stock updates from your home page, the pub-sub way of doing that would be to have each client register itself as a subscriber with the pub-sub server. The server then would send new messages at regular intervals to the pub-sub server, which would pass them along to the appropriate clients. Remember, in a pub-sub system, the publisher doesn't know how many subscribers there are or when they enter/leave the system. All the publisher needs to know is the API for sending data to the pub-sub server, which passes things along to the appropriate clients.

Implementing Pub-Sub

Pub-sub has long existed outside the Web and is a fairly standard architecture for “broadcasting” information to a variety of clients. *And, you can create a pub-sub system on your own, but there are at least two commercial services—Pusher and PubNub are the best known—that make it trivially easy to implement real-time messaging within your Web application.* Pusher uses Web sockets, substituting a Flash-based solution when a browser doesn't support them. PubNub uses a different system, known as “HTTP long polling”, which avoids the problem of browser support for Web sockets. Both are worth consideration if you're looking for a commercial pub-sub service, but I use Pusher here (as well as in my own consulting work), partly because I prefer to use Web sockets, and partly because Pusher lets you tag each message with an event type, giving you a richer mechanism for sending data.

Because Pusher is a commercial service, you need to register with it before you can use it. It has a free “sandbox” system that is more than viable for systems in development. Once you go beyond its limits of 20 connections and 100,000 messages per day, you need to pay a monthly fee. After signing up with Pusher, you need three pieces of information to implement the application:

- A key: this is the equivalent of your user name. It's used by the publisher to send messages and by the client to retrieve messages. This cannot be a secret, because it'll be in the HTML files that your users display in their browsers.
- A secret: the equivalent of a password. This will not be used on the client, or subscriber, side of things. But, it will be used on the server (publisher) side, which is sending data to the client. This ensures that only you are sending data.

- Finally, each application that you use with Pusher has its own application ID, a unique numeric code. If you have different applications running with Pusher, you need to register for additional application IDs.

Once you have those three pieces of information, you can begin to create your Web application. The simple application that you'll create here is a Web page that displays the latest information about a particular stock. Which stock? Whichever one the publisher has decided to show you. Such updates, of stock names and values, will be sent to your browser via pub-sub, letting you update the page without needing a refresh or any input from the user.

To get this all to work, you'll need an application in three parts. To start off with, let's create a trivially simple Web application using the Ruby-based Sinatra framework. Here is the entire application, which I put in a file named stock.rb:

```
#!/usr/bin/env ruby require "sinatra" require 'erb' get "/" do erb :index end
```

This program says that the Web application responds to GET requests on the / URL, and nothing more. Requests for any other URLs, or with any other methods, will be met with errors. If you are asked to display the / URL, you'll display the ERb (embedded Ruby) file, named views/index.erb. You can start your Web application by typing:

```
./stock.rb
```

On my system, I get the following response:

```
== Sinatra/1.3.3 has taken the stage on 4567 ↪for development with backup from Thin >> Thin web server (v1.5.0 codena
```

In other words, if I now make a request for http://localhost:4567/, I'll get an error, because the template is not in place. Creating a subdirectory named “views”, I then can create the file index.erb within it, which is shown in Listing 1. *Garrick, shrink listing 1.*

Listing 1. index.erb*11415l1.qrk*

```
<!DOCTYPE html> <!-- -*-html-*- --> <head> <title>Stock Market</title> <script src="http://ajax.googleapis.com/ajax/l
```

As you can see, index.erb is a simple HTML file. Its body consists of a headline and a single paragraph: *Garrick, shrink below.*

```
<p>Current value of <span id="name">NAME</span> is ↪<span id="price">PRICE</span>.</p>
```

The above line is the primitive stock ticker. When your publishing system will send a new stock name and price, you will update this line to reflect that message.

Just as you used a callback to handle incoming messages on your Web socket, you also will need to define a callback to handle messages sent by the publisher to your Pusher “channel”, as it is known. (Each application can have any number of channels, and each channel can have any number of events. This allows you to distinguish between different types of messages, even within the same application.)

In order to do this, you need to load the JavaScript library (from pusher.com), and then create a new Pusher object with the key of the account you have created:

```
var pusher = new Pusher('cc06430d9bb986ef7054');
```

You then indicate that you want to subscribe to a particular channel, the name of which does not need to be set in advance:

```
var channel = pusher.subscribe('stock_ticker');
```

Finally, you define a callback function, indicating that when you receive a message of type “update_event” on the stock_ticker channel, you want to replace the HTML in the body of this document:

```
channel.bind('update_event', function(data) {  $("#name").html(data['name']);  $("#price").html(data['price']); });
```

Notice that I'm using jQuery here in order to replace the HTML on the page. In order for that to work, I've also brought in the jQuery library, downloading it from Google's servers.

With this HTML page in place, and my Sinatra application running, I'm now ready to receive messages. I run the Sinatra application and point my browser to localhost:4567. I should see the static version of the page, with NAME and PRICE in the paragraph.

Publishing a message is almost as easy as receiving one. Different applications will have different use cases. Sometimes, you will want to send a message from the Web application itself, indicating that a new message has been posted to a forum or that the number of signed-in users has changed. In other cases, you'll want these updates to come from an external process—perhaps one that is running via cron or is monitoring the database separately from the Web application.

For this particular example, I wrote a small Ruby program, update-stocks.rb, which is shown in Listing 2. This program uses the “pusher” gem, provided free of charge by the Pusher people. You then choose one of the companies in your list (the constant array COMPANIES), then choose a random number up to 100. Next, you send the message to all of the subscribers on the “stock_ticker” channel, indicating that you've sent an “update_event”. Because of the decoupled nature of communication between publisher and subscriber, you won't get an error message if you misspell the channel or event name. Rather, the message will be delivered to no one. Thus, you will want to be particularly careful when writing these and ensure that the same names are used in your client and your server.

Listing 2. update-stocks.rb*11415l2.qrk*

```
#!/usr/bin/env ruby COMPANIES = %w(ABC DEF GHI JKL MNO) require 'pusher' Pusher.app_id = APP_ID_FROM_PUSHER Pusher.ke
```

Conclusion

Web sockets are going to change the Web dramatically, but it's not yet clear how or when. Being able to update a large number of client displays almost simultaneously using pub-sub is already changing the way people see Web apps—and as you can see from this small example application, it isn't very difficult to do. Pub-sub isn't appropriate for all applications, but if you are sending the same data to many people, and if they might want to receive updates automatically into their browsers, this is an easy and straightforward way to do it.

Resources*11415s1.qrk*

You can learn about Web sockets from a variety of sources. The W3C's API and definition are at http://www.w3.org/TR/2009/WD-websockets-20090423, in a document that is surprisingly readable. Another good source of information is the book *Programming HTML5 Applications* written by my colleague Zach Kessin and published by O'Reilly.

Web socket servers have been written in nearly every language you can imagine. I found a relatively up-to-date list, with links, on Wikipedia, under the “Web socket” entry, and thus, I'll not try to reproduce it here.

You can learn more about Pusher at http://pusher.com or a popular competitor, PubNub, at http://pubnub.com.