Introducing WebSockets: Bringing Sockets to the Web

By Malte Ubl and Eiji Kitamura

Published: October 20th, 2010
Comments: 48

The Problem: Low Latency Client-Server and Server-Client Connections

The web has been largely built around the so-called request/response paradigm of HTTP. A client loads up a web page and then nothing happens until the user clicks onto the next page. Around 2005, AJAX started to make the web feel more dynamic. Still, all HTTP communication was steered by the client, which required user interaction or periodic polling to load new data from the server.

Technologies that enable the server to send data to the client in the very moment when it knows that new data is available have been around for quite some time. They go by names such as "Push" or "Comet". One of the most common hacks to create the illusion of a server initiated connection is called long polling. With long polling, the client opens an HTTP connection to the server which keeps it open until sending response. Whenever the server actually has new data it sends the response (other techniques involve Flash, XHR multipart requests and so called htmlfiles). Long polling and the other techniques work quite well. You use them every day in applications such as GMail chat.

However, all of these work-arounds share one problem: They carry the overhead of HTTP, which doesn't make them well suited for low latency applications. Think multiplayer first person shooter games in the browser or any other online game with a realtime component.

Introducing WebSocket: Bringing Sockets to the Web

The WebSocket specification defines an API establishing "socket" connections between a web browser and a server. In plain words: There is an persistent connection between the client and the server and both parties can start sending data at any time.

Getting Started

You open up a WebSocket connection simply by calling the WebSocket constructor:

```
var connection = new
WebSocket('ws://html5rocks.websocket.org/echo', ['soap',
'xmpp']);
```

Notice the `ws:`. This is the new URL schema for WebSocket connections. There is also `wss:` for secure WebSocket connection the same way `https:` is used for secure HTTP connections.

Attaching some event handlers immediately to the connection allows you to know when the connection is opened, received incoming messages, or there is an error.

The second argument accepts optional subprotocols. It can be a string or an array of strings. Each string should represent a subprotocol name and server accepts only one of passed subprotocols in the array. Accepted subprotocol can be determined by accessing `protocol` property of WebSocket object.

The subprotocol names must be one of registered subprotocol names in IANA registry. There is currently only one subprotocol name (soap) registered as of February 2012.

```
// When the connection is open, send some data to the
server
```

```
connection.onopen = function () {
  connection.send('Ping'); // Send the message 'Ping' to
the server
};

// Log errors
connection.onerror = function (error) {
  console.log('WebSocket Error ' + error);
};

// Log messages from the server
connection.onmessage = function (e) {
  console.log('Server: ' + e.data);
};
```

Communicating with the Server

As soon as we have a connection to the server (when the `open` event is fired) we can start sending data to the server using the `send('your message')` method on the connection object. It used to support only strings, but in the latest spec it now can send binary messages too. To send binary data, you can use either `Blob` or `ArrayBuffer` object.

```
// Sending String
connection.send('your message');

// Sending canvas ImageData as ArrayBuffer
var img = canvas_context.getImageData(0, 0, 400, 320);
var binary = new Uint8Array(img.data.length);
for (var i = 0; i < img.data.length; i++) {
  binary[i] = img.data[i];
}
connection.send(binary.buffer);

// Sending file as Blob
var file =
document.querySelector('input[type="file"]').files[0];
connection.send(file);
```

Equally the server might send us messages at any time. Whenever this happens the `onmessage` callback fires. The callback receives an event object and the actual message is accessible via the `data` property.

WebSocket can also receive binary messages in the latest spec. Binary frames can be received in `Blob` or `ArrayBuffer` format. To specify the format of the received binary, set the binaryType property of WebSocket object to either 'blob' or 'arraybuffer'. The default format is 'blob'. (You don't have to align binaryType param on sending.)

```
// Setting binaryType to accept received binary as either
'blob' or 'arraybuffer'
connection.binaryType = 'arraybuffer';
connection.onmessage = function(e) {
  console.log(e.data.byteLength); // ArrayBuffer object if
binary
};
```

Another newly added feature of WebSocket is extensions. Using extensions, it will be possible to send frames compressed, multiplexed, etc. You can find server accepted extensions by examining the extensions property of the WebSocket object after the open event. There is no officially published extensions spec just yet as of February 2012.

```
// Determining accepted extensions
console.log(connection.extensions);
```

Cross Origin Communication

Being a modern protocol, cross origin communication is baked right into WebSocket. While you should still make sure only to communicate with clients and servers that you trust, WebSocket enables communication between parties on any domain. The server decides whether to make its service available to all clients or only those that reside on a set of well defined domains.

Proxy Servers

Every new technology comes with a new set of problems. In the case of WebSocket it is the compatibility with proxy servers which mediate HTTP connections in most company networks. The WebSocket protocol uses the

HTTP upgrade system (which is normally used for HTTP/SSL) to "upgrade" an HTTP connection to a WebSocket connection. Some proxy servers do not like this and will drop the connection. Thus, even if a given client uses the WebSocket protocol, it may not be possible to establish a connection. This makes the next section even more important :)

Use WebSockets Today

WebSocket is still a young technology and not fully implemented in all browsers. However, you can use WebSocket today with libraries that use one of the fallbacks mentioned above whenever WebSocket is not available. A library that has become very popular in this domain is socket.io which comes with a client and a server implementation of the protocol and includes fallbacks (socket.io doesn't support binary messaging yet as of Februrary 2012). There are also commercial solutions such as PusherApp which can be easily integrated into any web environment by providing a HTTP API to send WebSocket messages to clients. Due to the extra HTTP request there will always be extra overhead compared to pure WebSocket.

The Server Side

Using WebSocket creates a whole new usage pattern for server side applications. While traditional server stacks such as LAMP are designed around the HTTP request/response cycle they often do not deal well with a large number of open WebSocket connections. Keeping a large number of connections open at the same time requires an architecture that receives high concurrency at a low performance cost. Such architectures are usually designed around either threading or so called non-blocking IO.

Server Side Implementations

- Node.js
    - Socket.IO
    - WebSocket-Node
    - ws

- Java
  - Jetty
- Ruby
  - EventMachine
- Python
  - pywebsocket
  - Tornado
- Erlang
  - Shirasu
- C++
  - libwebsockets
- .NET
  - SuperWebSocket

Protocol Versions

The wire protocol (a handshake and the data transfer between client and server) for WebSocket is now RFC6455. The latest Chrome and Chrome for Android are fully compatible with RFC6455 including binary messaging. Also, Firefox will be compatible on version 11, Internet Explorer on version 10. You can still use older protocol versions but it is not recommended since they are known to be vulnerable. If you have server implementations for older versions of WebSocket protocol, we recommend you to upgrade it to the latest version.

Use Cases

Use WebSocket whenever you need a truly low latency, near realtime connection between the client and the server. Keep in mind that this might involve rethinking how you build your server side applications with a new focus on technologies such as event queues. Some example use cases are:

- Multiplayer online games
- Chat applications
- Live sports ticker
- Realtime updating social streams

Demos

- Plink
- Paint With Me
- Pixelatr
- Dashed
- Massively multiplayer online crossword
- Ping server (used in examples above)
- HTML5demos sample

References

- The WebSocket API
- The WebSocket Protocol
- WebSockets - MDN