WebSockets: A Guide

Apr 2nd, 2012 | by *[Jack Lawson](http://www.twitter.com/ajacksified)*

Efficient, realtime data transfer for modern web games

WebSockets provide two-way realtime communication between a client and server, and thus are exceedingly useful in building modern web games. Browser-based games can profit from an always-on, low-latency connection by enabling the rapid transmission of information about player and global game state previously emulated by methods such as Ajax polling and Comet. It is useful to first look at the history of WebSockets and gain an understanding of how WebSockets work at a technical level before we examine how we may use WebSockets most effectively. Armed with this knowledge, we can simplify the network layer and build amazingly responsive games that provide a high level of multiplayer interactions.

History

The Internet was developed (to grossly oversimplify) as a way to allow organizations to share information efficiently and with little delay. Information on the Internet is transported using a suite of connection protocols named *[TCP/IP](http://tools.ietf.org/html/rfc1122" \o "RFC 1122 - Requirements for Internet Hosts - Communication Layers (HTCPCP/1.0))*, defining the method through which computers would share information on a decentralized network with reasonable certainty that the information arrived correctly. TCP/IP functions by providing information about the message, such as source and destination. The message contains a checksum (a value calculated from the data in the message) that can be used by a receiver to verify if all of the information was received correctly. The spread of the “web” as we know it was through the *[HTTP protocol](http://tools.ietf.org/html/rfc2616" \o "RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1)*, providing a layer of abstraction that packages TCP/IP connections in an envelope containing information about the request as well as the data for the request itself, such as form fields and cookie values. This HTTP interface provides a simple request-response interface that works well for actions such as fetching a web page, loading an image, or submitting data to a server for persistence.

During the growth of web application development around 2004 and onwards, Ajax became an immensely useful method through which to retrieve data from a server. Ajax provided an interface through which javascript could create a HTTP request and handle the response asynchronously with a callback to a function on success or failure. Where users previously had to refresh or change pages to view updated content, a small amount of javascript could call a server, get updated data, and render this data on the page for a more seamless application. Gmail, Facebook, and Twitter are all notable examples of “single-page apps”, pulling data through behind-the-scenes server calls, and allowing the user a smoother workflow. HTTP remained unchanged as it had been since 1999, helping move along these packages of data through one-time-use connections.

As web applications evolved, so did the need for real-time communication. Chat applications, online games, and notification systems relied on abusing the HTTP protocol through systems such as Ajax polling, Comet persistent HTTP connections, opening iframes to poll for fresh data from the server, or using Flash either for the network layer or to build entire applications. While clever, each method had downsides, whether inefficiencies or complexities; and yet all along, the answer was right under the nose of the HTTP protocol: the same TCP connection that powered HTTP could be used for two-way, persistent, efficient connections directly between a client browser and a web server.

The *[WebSocket specification](http://tools.ietf.org/html/rfc6455" \o "RFC 6455 - The WebSocket Protocol)* has finalized at a fantastic time in the era of web application development: the advent of HTML5 and a plethora of related open web technologies. It’s now a stable spec supported in modern browsers like Chrome, Firefox, and Internet Explorer 10. Its persistent TCP connection means that developers can build responsive, connected games in ways far more efficient- both for server and client resource usage and development time- by using a natural pipe instead of a polling system. With WebSockets, one user in a game can move, and within milliseconds, their character may move on another user’s screen; a player can chat, and their message appear instantly in-game; a tank can shoot a shell, and the vector can be traced in real-time on two screens at once; and all of this can be built with open web technologies.

A WebSocket Dissection

*We’ll be talking about the RFC6455 (hybi-17) model, which has some differences from earlier implementations. As this is the official standard, we won’t discuss the differences.*

Technically speaking, a WebSocket is a bi-directional full duplex persistent TCP connection secured by a client-key handshake and an origin-based security model. It also masks its data transmissions to prevent plain text packet sniffing.

Breaking this statement down:

* Bi-directional: the client is connected to the server, and the server is connected to the client. Both receive events such as connected and disconnected, as well as the ability to send data to the other.
* Full duplex: the server and the client can send data to each other at the same time without collision.
* TCP: the underlying protocol of all Internet communication, which provides a mechanism for reliably transporting a stream of bytes from one application to another.
* Client-key handshake: the client sends a base64-encoded, 16-byte secret key to the server, which then appends a key (specified in the protocol as “258EAFA5-E914-47DA-95CA-C5AB0DC85B11”) and sends the sha1 hashed value back to the client. In this way, the client can be sure that the same server that it had sent its key to is the same server that is opening the connection.
* Origin-based security: the origin of the WebSocket request is verified by the server to determine if the origin of the request is from an authorized domain. The server can reject any socket connections from an untrusted domain.
* Masked data transmissions: The client sends a four-byte masking key along with the beginning frame of each message, which is used by performing a bitwise XOR against this key and the data. This helps to prevent data sniffing, as an attacker must be able to determine the starting byte of the message in order to decrypt the entire message.

The RFC documentation defines a WebSocket frame as the following:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-------+-+-------------+-------------------------------+

|F|R|R|R| opcode|M| Payload len | Extended payload length |

|I|S|S|S| (4) |A| (7) | (16/64) |

|N|V|V|V| |S| | (if payload len==126/127) |

| |1|2|3| |K| | |

+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +

| Extended payload length continued, if payload len == 127 |

+ - - - - - - - - - - - - - - - +-------------------------------+

| |Masking-key, if MASK set to 1 |

+-------------------------------+-------------------------------+

| Masking-key (continued) | Payload Data |

+-------------------------------- - - - - - - - - - - - - - - - +

: Payload Data continued ... :

+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +

| Payload Data continued ... |

+---------------------------------------------------------------+

The data frame is fairly simple; it contains information on the state of this particular frame, its payload length, a masking key, and the data for the frame.

Beginning from the top left, we have:

1. Fin (bit 0): determines if this is the last frame in the message. This would be set to 1 on the end of a series of frames, or in a single-frame message, it would be set to 1 as it is both the first and last frame.
2. RSV1, RSV2, RSV3 (bits 1-3): these three bits are reserved for websocket extensions, and should be 0 unless a specific extension requires the use of any of these bytes.
3. Opcode (bits 4-7): these four bits deterimine the type of the frame. Control frames communicate WebSocket state, while non-control frames communicate data. The various types of codes include:
   1. x0: continuation frame; this frame contains data that should be appended to the previous frame
   2. x1: text frame; this frame (and any following) contains text
   3. x2: binary frame; this frame (and any following) contains binary data
   4. x3 - x7: non-control reserved frames; these are reserved for possible websocket extensions
   5. x8: close frame; this frame should end the connection
   6. x9: ping frame
   7. xA: pong frame
   8. xB - xF: control reserved frames
4. Mask (bit 8): this bit determines whether this specific frame uses a mask or not.
5. Payload Length (bits 9-15, or 16-31, or 16-79): these seven bytes determine the payload length. If the length is 126, the length is actually determined by bits 16 through 31 (that is, the following two bytes). If the length is 127, the length is actually determined by bits 16 through 79 (that is, the following eight bytes).
6. Masking Key (the following four bytes): this represents the mask, if the Mask bit is set to 1.
7. Payload Data (the following data): finally, the data. The payload data may be sent over multiple frames; we know the size of the entire message by the payload length that was sent, and can append data together to form a single message until we receive the message with the Fin flag. Each consecutive payload, if it exists, will contain the 0 “continuation frame” opcode.

The data frame in RFC6455 allows us to understand how large the entire message is, its encoding type, and its masking (if any). We can send messages of any length whose length can be put into a base-64 number (that is, 9,223,372,036,854,775,808 digits in length).

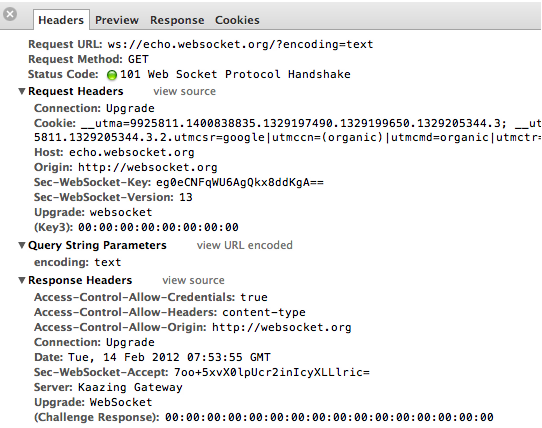
WebSocket API

*[The WebSocket API](http://dev.w3.org/html5/websockets/" \o "The WebSocket API)* that we will be interfacing with in our javascript is elegant and simple. The API defines an object that contains:

1. Information about the state of the connection (connecting, open, closing, and closed)
2. Methods for interacting with the websocket connection (closing a connection, and sending data)
3. Events that are fired when a websocket event occurs (when a socket is opened, closed, or receives an error response)

A simple implementation of a WebSocket connection may look as follows:

Once the WebSocket constructor is called, the browser and server initiate a handshake. The initial connection is made using a HTTP “upgrade” command; the WebSocket *[echo test tool](http://websocket.org/echo.html" \o "WebSocket Echo Test)* displays the following information in Chrome 19’s developer tools networking tab:



There is plenty of interesting information contained within this header:

1. Request Headers
   1. Connection: the “Upgrade” handler sent tells the server that we’re attempting to upgrade to a WebSocket connection if available.
   2. Origin: this origin is verified by the server to determine if this origin is allowed by its security protocols;.
   3. Sec-WebSocket-Key: this key forms the first part of the handshake. It is a randomly-generated and base64-encoded 16-byte string. Sec-WebSocket-Version: this allows the server to respond to the protocol properly to the version of the protocol used, if available.
2. Response Headers
   1. Sec-WebSocket-Accept: the server will append the protocol-specified string “258EAFA5-E914-47DA-95CA-C5AB0DC85B11” [6] to the client’s Sec-WebSocket-Key and sha-1 hash the result. The client will verify the result of this hash to determine if the key, and therefore the server, is the same.

Assuming all goes well with the server’s response, our socket object now has a readyState of 1, which corresponds to open [4]. We can send messages from this state and use our onMessage function to handle the reception of data from the server. Possible readyStates are 0 (connecting), 1 (open), 2 (closing), and 3 (closed).

Implementation Practices

When working with Ajax, the prevailing amount of examples use callbacks to handle responses from the server. Ajax requests (as Ajax is simply a method of making a HTTP call) make a request and receive a response, which in turn calls a specified callback method. These requests were often built as a polling system that pinged the server every few seconds for new data. An example may be a game with a chat.

1. An Ajax request is sent to the server every 5 seconds, to check the server for new chat messages.
2. If there is a new message, then show it.
3. If not, then do nothing. Check again in five seconds.

However, this architecture does not scale well as we begin to have to deal with our calls truly asynchronously. We may receive any arbitrary data at any moment that we have to respond to in some way in our logic; the key difference is that we cannot rely on this data as a response to a call we made previously. A callback-based architecture tightly couples our network layer with our client layer, and we end up with assumptions about race conditions and server availability. In doing this, we can fall into a trap of allowing our network layer to determine the control flow of our application.

Let us examine a simple example of a chat system of a game where the logic is driven through callbacks. A player on one end of the system submits a message; the message is routed through the server, and another player receives formatted JSON data with that chat message. This message may take one of several shapes; it could be a normal message, a private message, or a system announcement. A traditional approach may include code such as the following:

This kind of system can quickly grow unmaintainable; as we add new chat message types, for example, the list grows longer and longer. Adding features such as alerting you if your name is mentioned adds even more complexity onto the system. Once we start adding more actions on the same socket connection, we’ll have to start adding conditionals to check if the data type is “shot fired”, or “player moved”, or “player joined the game”, and so on. This list can grow into an ever-growing list of functions and corresponding types.

In contrast, a more robust structure using the *[AmplifyJS](http://amplifyjs.com/api/pubsub/" \o "PUB/SUB AMPLIFYJS API DOCUMENTATION)* *[publish-subscribe](http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe/_pattern" \o "Wikipedia: Publish-Subscribe pattern)* (henceforth denoted as pub/sub) management system may look like the following:

While the implementation of this simple functionality seems somewhat inflated at first glance, we can demonstrate a huge increase in maintainability and flexibility. By splitting the WebSocket handling away from callbacks, we can include any number of external scripts and modules which can send or receive data without needing to know any details of the connection and without injecting additional logic and conditionals into the WebSocket management code. The WebSocket connection itself is another module (a module for handling network traffic) which listens to pub/sub calls as well, and can perform actions against a server. With a bit of imagination, we can even extend the network layer with localstorage and an AI module for offline singleplayer gameplay! One can abstract the network communication layer away and provide a method to update data no matter what the network layer looks like; the layer could be comprised of WebSockets, Ajax, or even an entirely client-side update system with no connection at all. Additionally, by writing the code as modules, the code gains testability and maintainability by limiting the concerns of each module.

We still require some kind of mapping that we can simplify by using a dictionary of type definitions that will determine the channel of the call. One example may be ["chat", "message", "normal"], sequentially mapped. Our robust pub-sub type WebSocket layer might look something such as this example:

This update reduces our entire channeling system to a single line of code, based on the type of the data we’re receiving; data.Type may look something like “message” or “player” and data.SubType may look like “normal” or “move”. For even further robustness, we may instead pass Type as an array, allowing us to subscribe to any specificity of a data type that we’re interested in. Chat may listen into a base “message” channel, while private messenger alerts may listen to a more strict “message:private” channel.

function(data){ pubsub.publish(data.Type.join(":"), data); };

On the subscription side, we can now start to see the interesting effects. For example, if we use a pubsub system that allows us to use predicates such as *[Mediator.js](http://thejacklawson.com/Mediator.js/" \o "Mediator.js)*, we can use even more advanced subscription methods that will only run that method if a specific function acting upon the data returns true. In the following example, we set up a subscription that alerts a player that their name was mentioned, if the predicate function passed in returns true.

Through this architecture, we can have a chat module, a game module, a lobby module, a player module, a friendslist module, and modules for any other game functionality. All portions of the game can be built seperately, with the common ability to communicate through an abstract interface. The code has full flexibility to add and remove features at will, and the developer has confidence in modifying and adding code that’s completely specific to the feature that we’re working on.

The developer also gains the ability to easily write test-driven code through a framework such as *[Jasmine](http://pivotal.github.com/jasmine/" \o "Jasmine: BDD for your Javascript)* or*[Sinon](http://sinonjs.org/" \o "Sinon.JS)*, without needing to mock a WebSocket connection and without crossing too many concerns with our tests (test-driven application development is another subject for another article, however).

Limitations

WebSockets aren’t a cure-all to everything. HTTP still commands a key role in communication between client and server as a way to send and close connections for one-time data transfer, such as initial asset loading. HTTP requests can perform more efficiently than WebSockets by closing connections once complete rather than maintaining connection state.

Additionally, WebSockets are only available where users are using modern browsers with javascript enabled. While [*Flash shims*](https://github.com/gimite/web-socket-js/) are available and useful to provide functionality to non-compliant browsers, some users will not be able to access javascript-only or Flash-only content. If applicable, an alternate method of accessing the application should be provided through [*progressive enhancement*](http://en.wikipedia.org/wiki/Progressive/_enhancement).

An impact on network architecture should also be considered. WebSockets, as a persistent connection, can potentially use far more resources and tie up servers as compared to a standard web server. Impact on load balancers and firewalls can be mitigated; the WebSocket specification allows for transferring connections. A client may connect to a loadbalancer, which then passes the connection off to an application server to handle the actual data frame processing.

Links and Resources

WebSocket Servers

Most popular languages have WebSocket implementations. The following is a list of some of the more recently-active libraries for a small selection of languages.

Disclaimer: I am a principal developer on Alchemy Websockets.

1. Node
   1. [*Socket.io*](http://socket.io/) (also supports fallbacks for non-WebSocket clients, and has alternate-language implementations)
   2. [*WebSocket-node*](https://github.com/Worlize/WebSocket-Node) (hybi-8, 13)
2. C#
   1. [*Alchemy Websockets*](http://www.alchemywebsockets.net/) (hybi-00, 10, 13, 17, RFC 6455)
   2. [*Fleck*](https://github.com/statianzo/Fleck) (hybi-00, 10, 13, 17, RFC 6455)
3. Java
   1. [*Java-WebSocket*](https://github.com/TooTallNate/Java-WebSocket.git) (hixie-75, hybi-00, 10, 17, RFC 6455)
   2. [*jWebSocket*](http://jwebsocket.org/)
4. Python
   1. [*ws4py*](https://github.com/Lawouach/WebSocket-for-Python)
5. Erlang
   1. [*Misultin*](https://github.com/ostinelli/misultin)
6. Ruby
   1. [*em-websocket*](https://github.com/igrigorik/em-websocket) (hixie-75, hybi-00, 5, 7, 13, 17)

Browser Support

WebSockets are supported in most major browsers, although some browsers have support only in recent alpha or beta versions. Wikipedia [*maintains a list of browsers*](http://en.wikipedia.org/wiki/WebSocket#Browser_support). Browser support includes:

1. Desktop
   1. Internet Explorer: version 10 has hybi-10 support
   2. FireFox: version 11 supports RFC6455
   3. Chrome: version 16 supports RFC6455
   4. Safari: version 5.0.1 supports hybi-00
   5. Opera: version 11.00 supports hybi-00 (must be enabled by user)
2. Mobile
   1. Android browser: none; however, Chrome for Android has RFC6455
   2. iOS 5 browser: supports hixie-75
   3. Opera Mobile: hybi-00 enabled through system settings

Additional WebSocket support can be given to non-supportive browsers through use of a Flash shim such as[*web-socket-js*](https://github.com/gimite/web-socket-js/). You can test your own browser’s support at [*http://websocket.org/echo.html*](http://websocket.org/echo.html), or [*learn more about how it evolved here*](http://blogs.msdn.com/b/ie/archive/2011/09/15/site-ready-websockets.aspx).

Client-side Event Management

Callbacks can lead to a confusing data flow; using a system of custom events or a publish-subscribe system with WebSocket messages can keep code clean and modular.

Libraries that can help with implementing the pub/sub pattern include:

1. [*AmplifyJS*](http://amplifyjs.com/api/pubsub/): pub/sub management
2. [*Backbone*](http://documentcloud.github.com/backbone/): application architecture framework that includes custom events (check out [*this blog post*](http://andyet.net/blog/2011/feb/15/re-using-backbonejs-models-on-the-server-with-node/)about using Backbone with WebSockets)
3. [*YUI*](http://www.fleegix.org/articles/2008-07-29-painless-publish-subscribe-with-yui-custom-events): custom events
4. [*Dojo*](http://dojotoolkit.org/reference-guide/dojo/publish.html#introduction): pub/sub management
5. [*Mediator.js*](http://thejacklawson.com/Mediator.js/): Pub/sub management class built for Ajax and WebSocket handling; allows predicates and namespaced channels

Addy Osmani wrote [*an excellent post on pub/sub*](http://msdn.microsoft.com/en-us/scriptjunkie/hh201955) that explains where, why, and how to implement the pattern.

Posted by Jack Lawson Apr 2nd, 2012 in [*Multiplayer*](http://buildnewgames.com/blog/categories/multiplayer/) .