# Why you don’t need Socket.IO

Ivan Vanderbyl Follow
Mar 15, 2016 · 6 min read

Asynchronous communication between browsers and servers has come a very long way in the last 6 or 7 years.

Back then, if you wanted to send a message asynchronously, you had to employ a bunch of hacks. These typically included some or all of the following:

- Long polling
- Flash Sockets (yes, Adobe Flash)
- *Possible* upgrade to WebSockets

These hacks were commonly employed as part of the very popular abstraction library, *Socket.IO*. Which was designed to initiate a connection with the lowest common denominator, then ‘upgrade’ the connection if the client supported a better feature.

This approach worked in nearly all cases, right back to browsers which nobody cares about anymore. However it came at the cost of expensive connection times, excess CPU/Memory/Garbage collection, extra resources to load, and a limited ability to support the developing standard features of the WebSockets protocol, which most notably until very recently, that meant *binary streams*.

## Socket.IO

Over the years, Socket.IO became insanely popular, with over 25,000 stars on Github, and 5,000+ forks.

But the project always felt like it was trying to do too much. Is it a chat library? Is it a presence service? Is it a Socket? Is it a Sledgehammer? (I would have named it Sledgehammer.IO)

Nowadays Socket IO is actually two libraries. Engine.io which powers the socket abstractions, and connection management. And Socket.IO, which from what I can figure handles reconnection, event emitting, and message namespacing (kind of like chat rooms).

This is all excess when all you really need or want is a Socket.

## Socket programming

Sockets are the basis for all connections over the internet. They are the end pieces which both a client and server terminate a connection with.

Low level sockets are essentially a stream of bytes. You write to one socket, and it appears byte for byte on the other socket’s read buffer.

When you make an HTTP request over the internet, it typically opens a socket connection to the server, and sends a request header, followed by an optional request body. This is just a stream of bytes, and it’s up the the receiving socket to know when the message is completely received. For HTTP this is handled by specifying the Length of the payload. The client can then just receive until it has Length bytes, then take those bytes and parse them as a request. To reply, it would then send back a response using the same semantics. Before closing the connection (unless instructed not to by a Keep-Alive header).

This is essentially what we call Framing messages. More on this later.

Some streams can be essentially endless and continue to write data, such as a video stream or audio stream. The underlying socket functions the same way, except messages are delineated using a special byte sequence, such as a null byte.

Up until relatively recently, the Web, as in browsers, didn’t have an equivalent to sockets. You could only make asynchronous requests for resources from the client, but the server never had a way to reply without the client asking it something. Around the time Node.js came along, the ability for the server to send out of band messages to the client was seen

as incredibly useful, especially for an async server, which is why Socket.IO became so popular.

WebSockets were still very young, and there wasn’t yet a protocol standard which all browsers agreed to implement. It took a number of years, and many completing standards before in 2011 RFC 6455 became the offical WebSocket protocol.

## WebSockets

WebSockets aren’t *real* sockets. But they’re pretty damn close, and they work remarkably well.

The WebSocket protocol introduced a couple of extra attributes over the humble TCP socket. Firstly, in order to establish a connection, the client must request an “upgrade” from HTTP to WebSocket during the handshake process. This is initiated using a standard HTTP request to a given WebSocket endpoint. If the server supports WebSockets it will respond with a *HTTP/1.1 101 Switching Protocols* response, which instructs the client to keep the connection open, and use it as a stream.

After upgrading the connection, you’re free to send data down the socket in both directions, but here comes the second main difference: You’re not receiving a stream of bytes. All messages are framed automatically by the socket. So once one end finishes writing, the message will be available on the other end, and the socket will automatically decide how many bytes belong to this message. This makes WebSockets considerably easier to program with compared with regular sockets.

WebSockets also implement their own Ping/Pong semantics to keep the connection alive in the absence of any data being sent. This is something you typically have to account for with TCP sockets to ensure the other end hasn’t gone away.

Another useful feature of being able to establish a connection using HTTP is the ability to use HTTP authentication semantics for the socket, such as the Authorization header for Basic Auth or Bearer Token Auth.

## Authentication strategies

I mentioned earlier that you can use standard HTTP authentication semantics, however sometimes you want to handle authentication at a later stage, or asynchronously.

A common pattern for this is to require that the first message over the socket is an authentication message, commonly called an “*authentication handshake*”. This is where the two parties exchange enough information about each other that they agree to keep the connection open.

Typically the client exchange with the server something like this:

```
CLIENT > AUTH: secretpasscodeletmein
SERVER < OK
```

The server — which is waiting for a message of type AUTH — would take the value and check its authenticity. If the value turns out to be correct, the connection stays open and server replies with some sort of *OK* message. Otherwise it simply closes the connection.

Because you can send whatever you like down the socket, it’s up to you to design a protocol which makes sense to both your server and client’s needs. This commonly would include a protocol negotiation message so you can support different features as your app develops without breaking older clients.

Or using an existing protocol, like STOMP or similar.

## Use case: Chat

Socket IO’s initial attraction was that you could build a chat service which didn’t require continually polling the server for new messages, and anything you types could appear on multiple clients seemingly immediately.

To implement something like this with WebSockets is actually quite easy, and a lot more flexible than you might expect.

On the server side you need to handle clients connecting. These will represented as client objects, one for each connection which is open. You might have a flag to indicate a few required attributes about the client:

```
let client = {
 id: 1,
 isAuthenticated: true,
```

```
 protocolVersion: 1,
 connection: <WebSocket Connection instance>
}
```

Once authenticated you would set the isAuthenticated flag to true, as there will be some delay between the connection opening and the client sending the initial authentication sequence, and you’ll probably want to exclude this client from certain broadcasts.

To handle broadcasts to each client, either one-to-one, or one-to-many, you could create a Map, keyed on channel name, with a Set of values representing the IDs of each subscribed client.

How you establish a subscription is up to you, but it could be as simple as the client exchanging:

```
CLIENT > SUB:general
SERVER < OK

CLIENT > UNSUB:general
SERVER < OK
```

Then when you have a message to deliver — which could come from another client, internally, or an API request — you just need to look up all the clients associated with that channel, and send the message to each one. Example:

```
const subscriptions = new Map();
subscriptions.set(“general”, new Set())

// Subscribe a client to “general” channel.
subscriptions.get(“general”).add(client.id);

// Send message to a channel
subscriptions.get(“general”).forEach((clientId) => {
  clients.get(clientId).connection.send(message);
});

// Unsubscribe
subscriptions.get(“general”).delete(client.id);
```

See https://www.npmjs.com/package/websocket for a good starting point for a NodeJS server implementation, of if you’r using Go, https://github.com/gorilla/websocket is a good choice.

## Compatibility with other clients

A major advantage of adopting WebSockets over Socket IO is interoperability with many different clients. There are standards compliant WebSocket client implementations in ObjectiveC, Javascript, Go, Haskell, Erlang, Ruby, Swift, and of course all major browsers. And this is the way it should be. You shouldn’t need to adapt your code to many different transports that do the same thing.

## Thanks to Socket IO

Without Socket IO a lot of very cool and useful apps would never have been built, and without the hard work of hundreds of contributors the library wouldn’t be were it is today. But I think its time to upgrade to pure WebSockets, and remove the extra cruft from your app.

## Conclusion

If you’re a browser and you need a socket, use WebSockets. All major browsers have supported WebSockets for years, so go forth and be async.