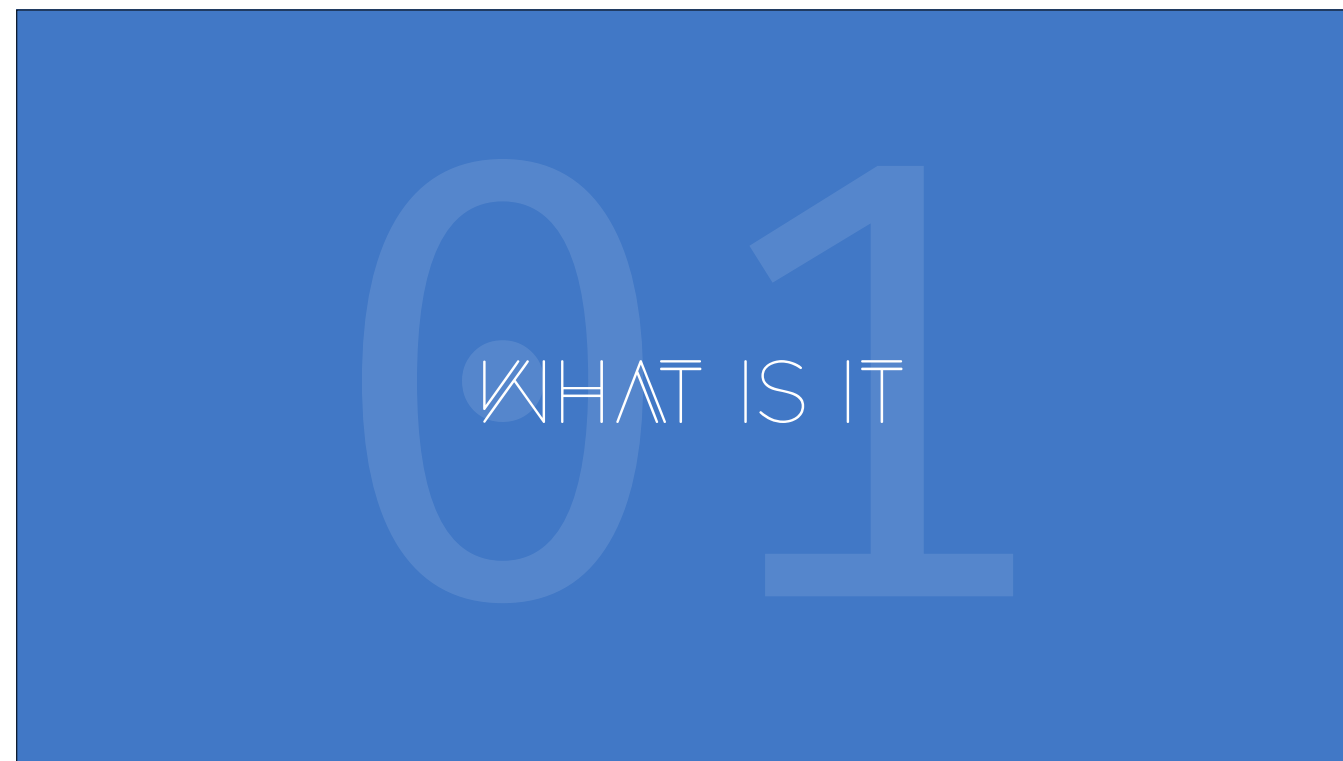


WEB SOCKETS VS SERVER SENT EVENTS



So what are WebSockets and Server-Sent Events

Realtime client-server communication

They are mechanisms for real-time communication between the client and the server

02

WEB SOCKETS

Ok, so, what exactly are WebSockets?

Introduced in 2011
RFC 6455

WebSockets were introduced in 2011 in RFC6455 by Google

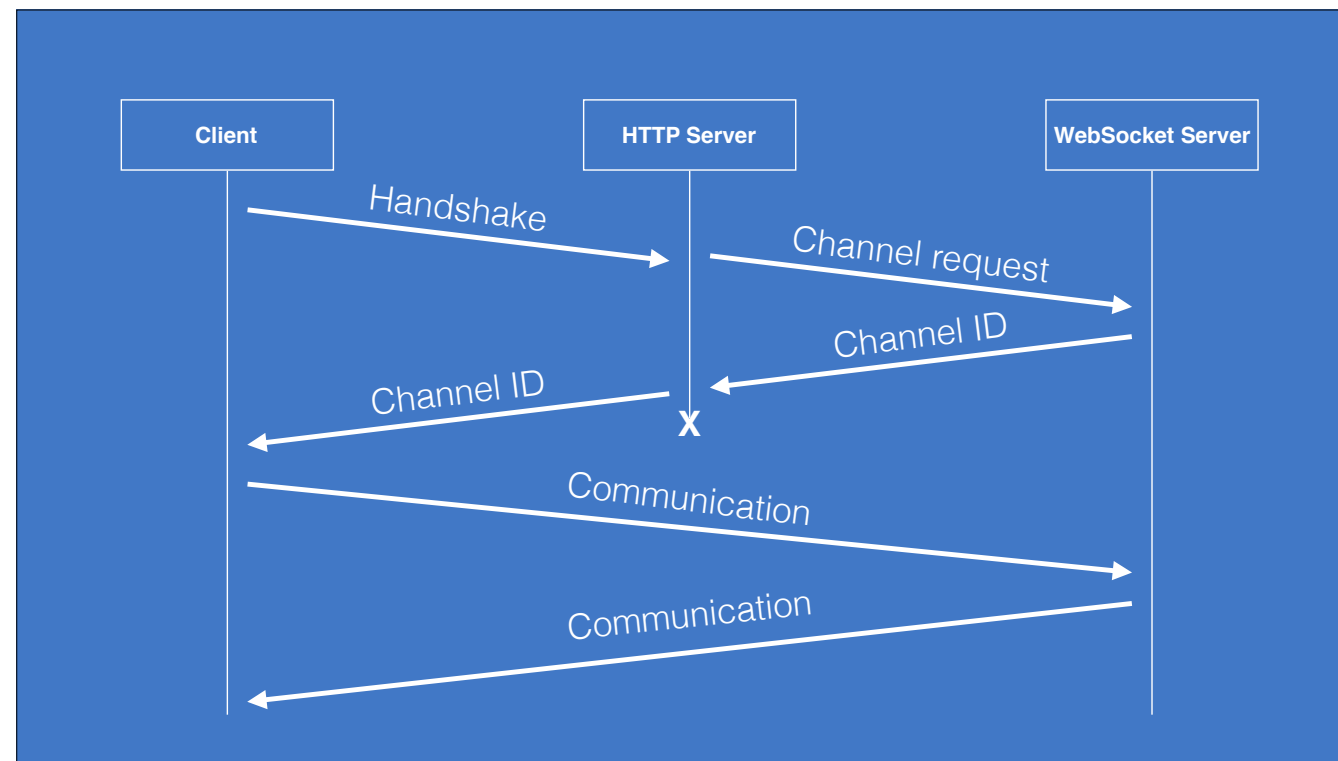
TCP based Full Duplex Connection

They provide a full-duplex connection. The word full-duplex comes from the telecommunications industry, it means that a communication channel supports simultaneous two-way communication. The telephone is a good example of a full-duplex connection, when you use it you can simultaneously talk to and hear the person you are talking with.

WebSockets are based on pure TCP. This has a few drawbacks.

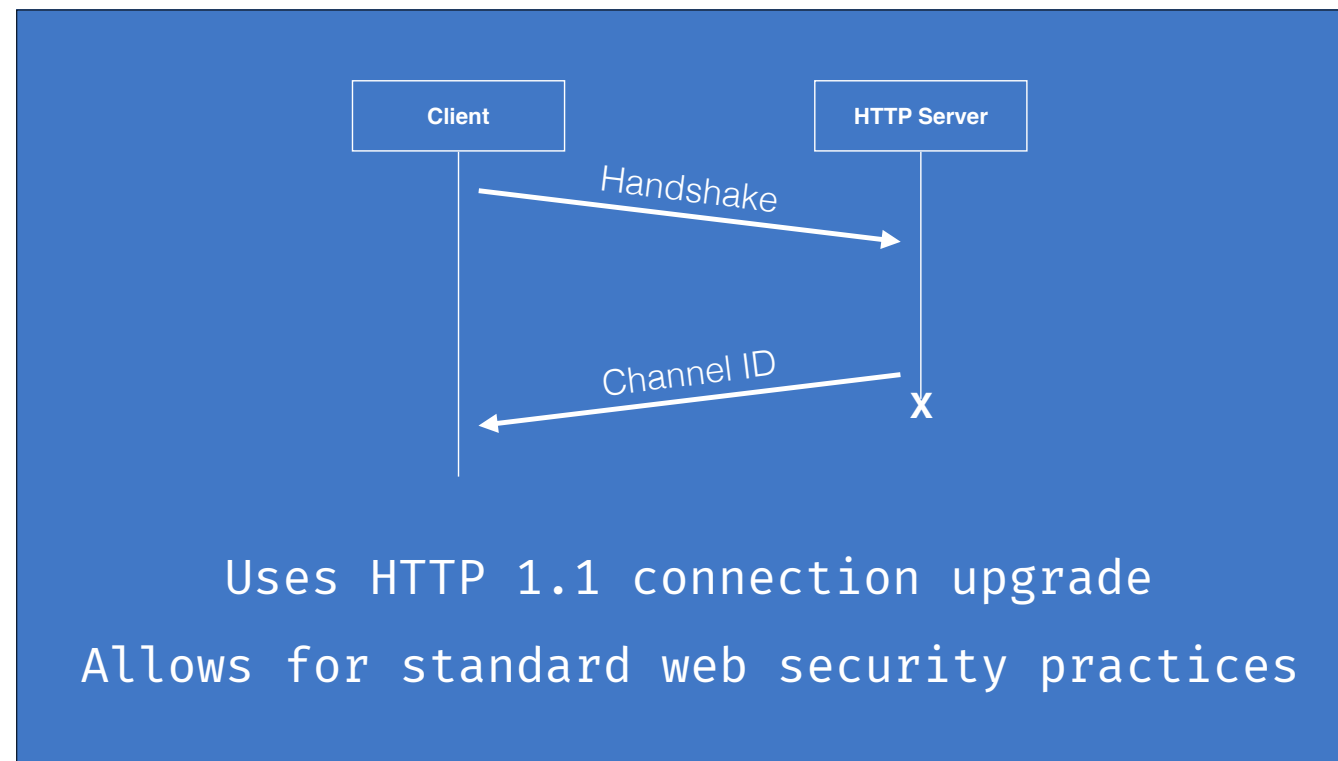
Each server implement its
own protocol

Namely, each WebSocket server implements their own protocol. WebSockets only define the initial connection, afterwards each server can do what, and how they want.

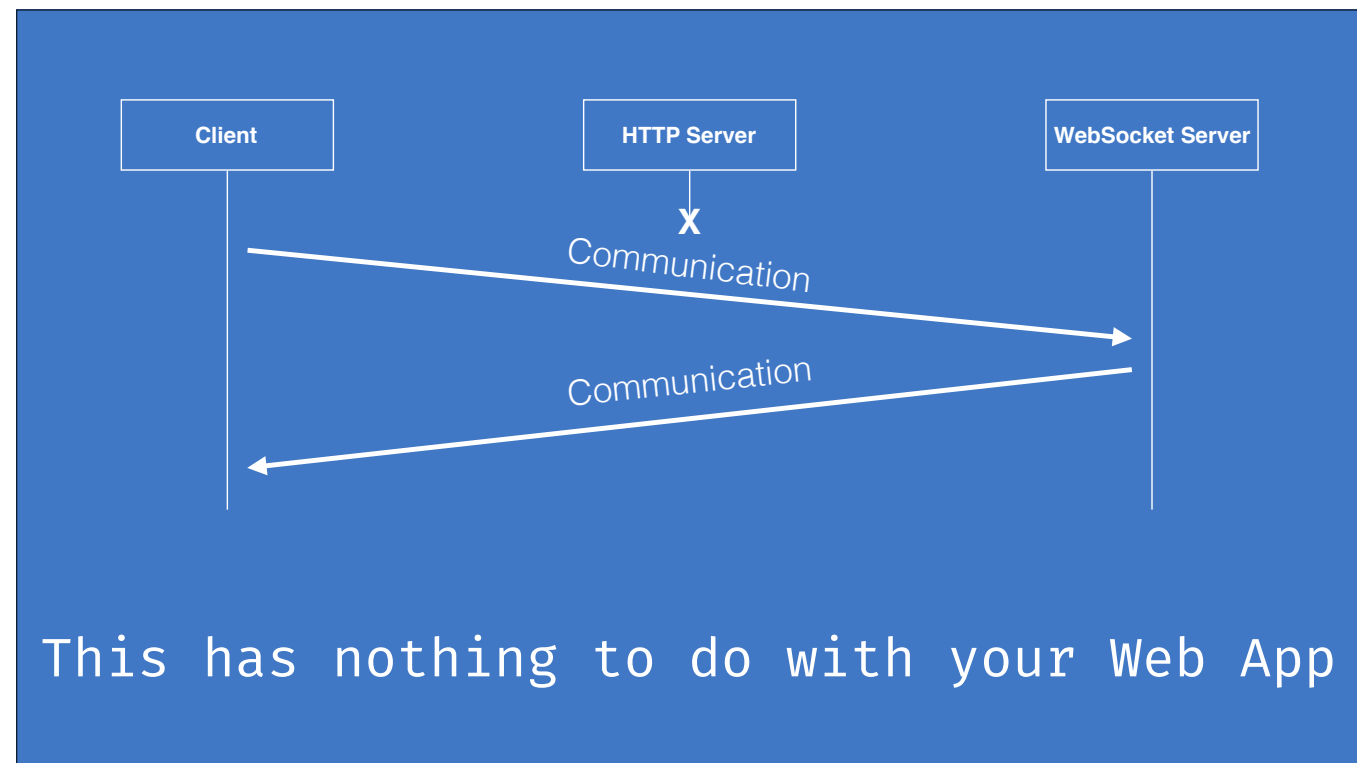


Here is a diagram of the process. After the Client-Server handshake is complete the client only talks to the WebSocket server.

Note, the handshake is used to upgrade the connection from HTTP to WebSockets, and for “security” since you can use default HTTP mechanisms like auth, headers, cookies and tokens.



So, HTTP 1.1 connection upgrade is used to tell the client on which IP and PORT it should connect to and give it the necessary information to establish the connection (such as tokens, channel ids, etc.). This allows the HTTP server to perform auth with the HTTP mechanisms we are used to.



After the connection is established, all further communication circumvents your HTTP server.

03 SERVER SENT EVENTS

Now for server sent events

Introduced in 2006
WHATWG Web Applications 1.0

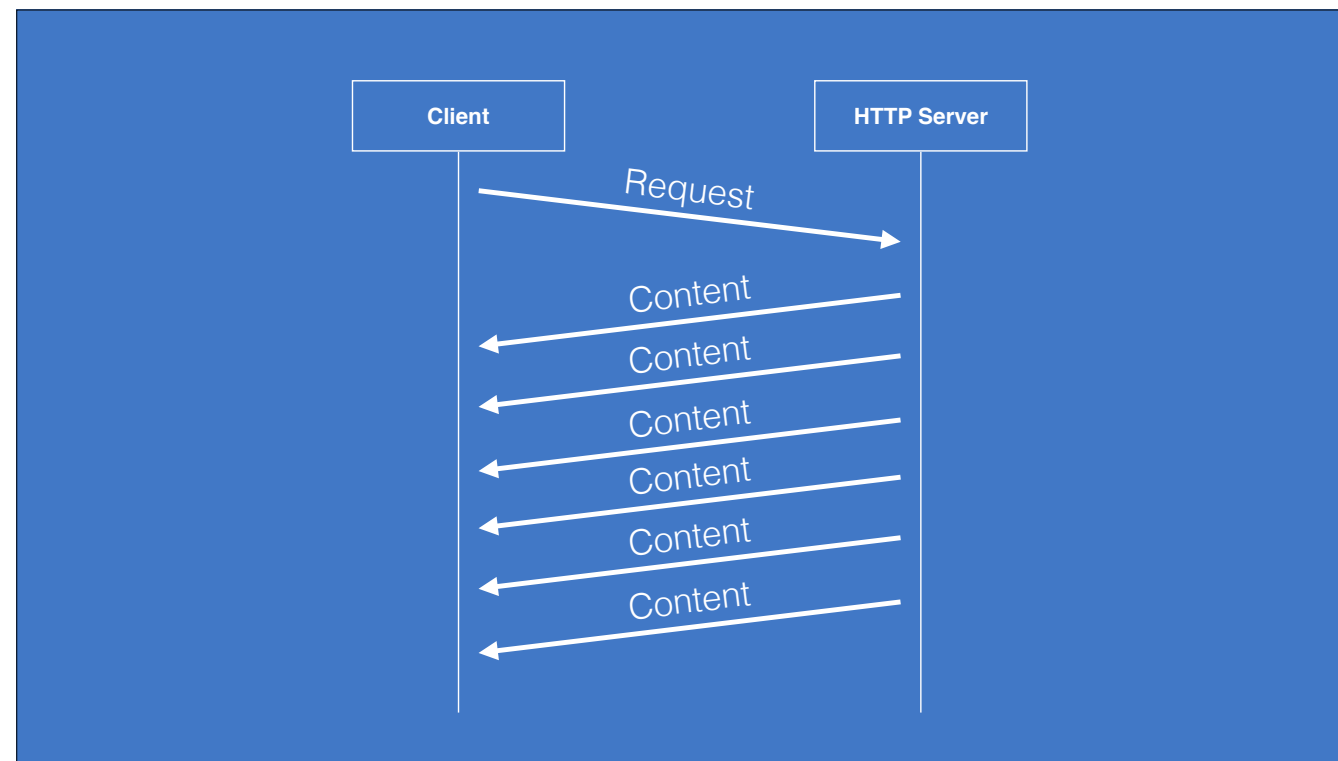
They are OOOOOOOOLD.

Only Server to Client communication

They only provide a way for the server to push data to the client in real-time.

HTTP based
Relies on HTTP streaming

It's completely HTTP based and is implemented with ordinary HTTP streams.

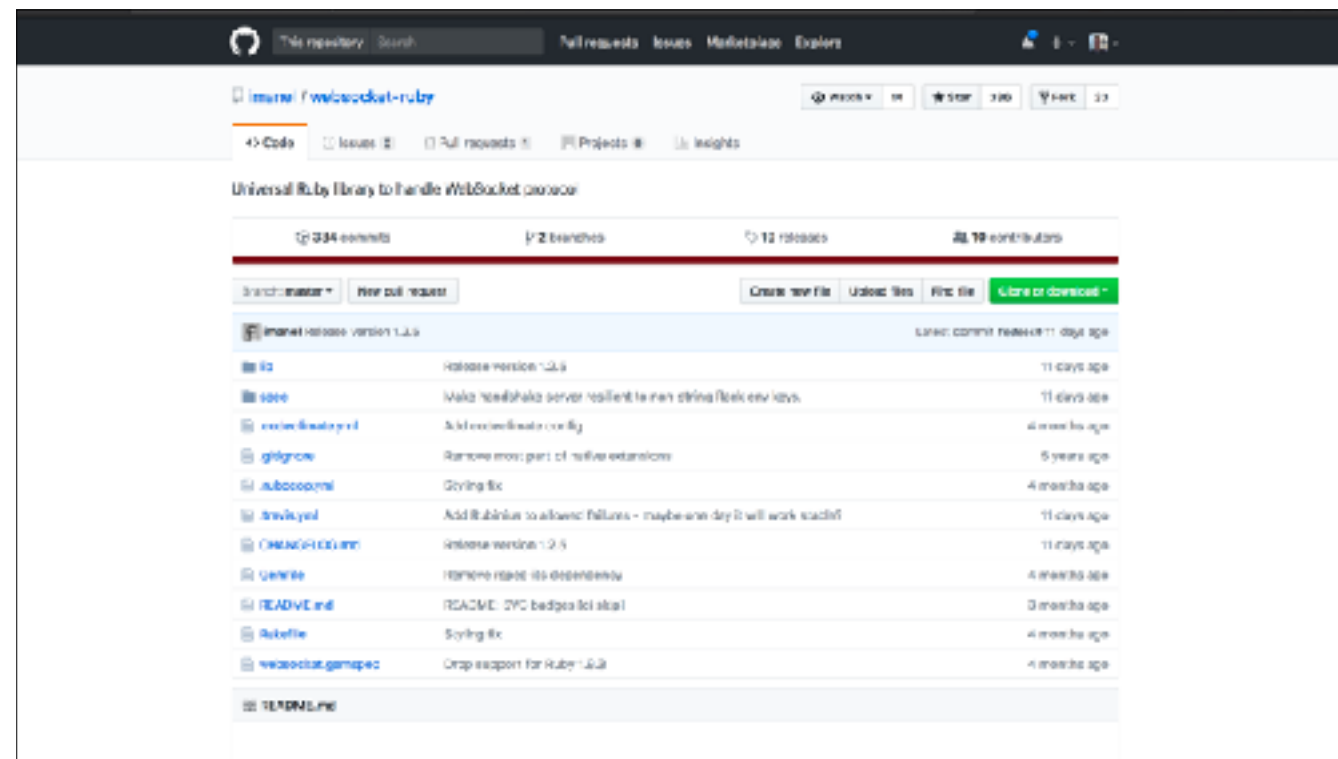


The client requests a stream, and the server pushes chunked responses back when changes occur.

04 IMPLEMENTATION

The nitty-gritty.

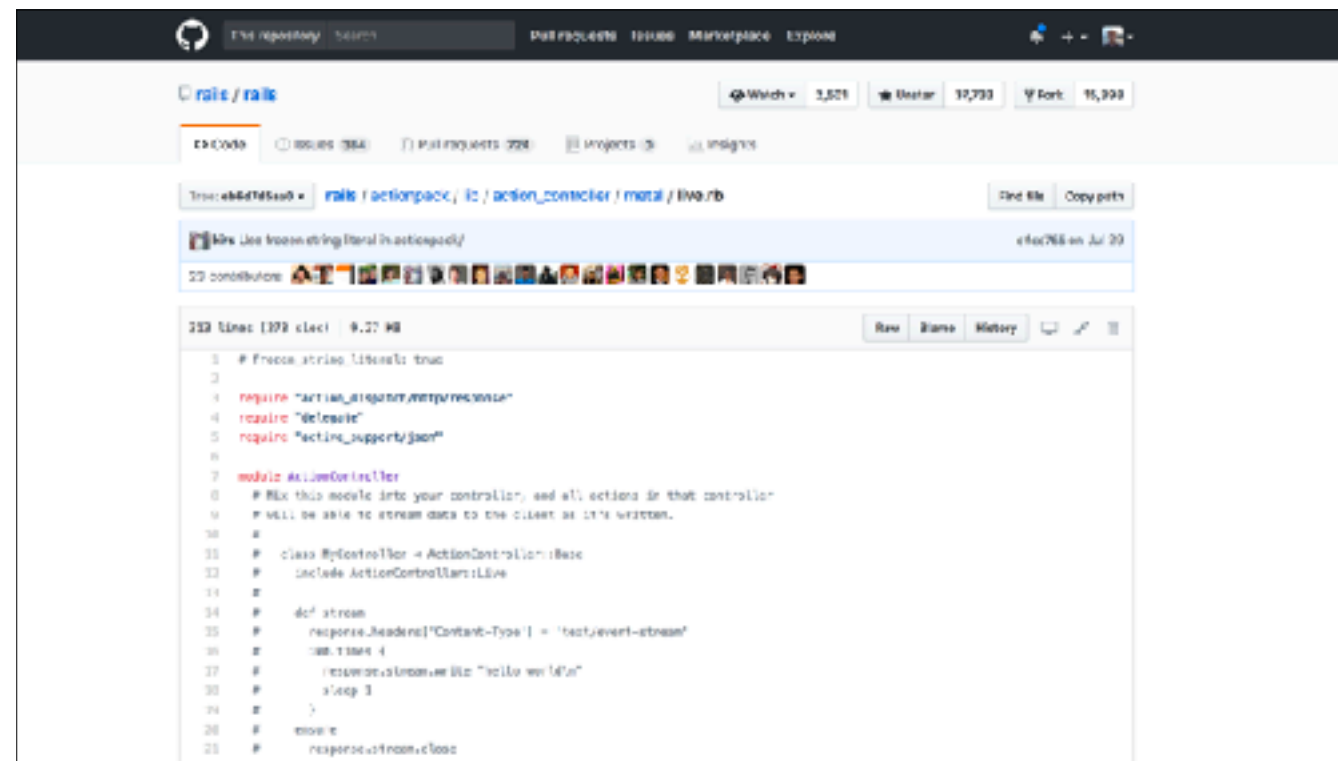
WebSockets



I used a standard implementation to check how complicated it is to implement WebSockets.

100k lines of code!
2 server implementations!
1 client implementation!

SSE



The screenshot shows the GitHub repository for rails/rails. The file rails/actionpack/lib/action_controller/metal/live.rb is open. The code is as follows:

```
1 # frozen_string_literal: true
2
3 require "action_dispatch/http/response"
4 require "delegate"
5 require "active_support/json"
6
7 module ActionController
8   # Mix this module into your controller, and all actions in that controller
9   # will be able to stream data to the client as it's written.
10
11   #
12   # class MyController < ActionController::Base
13   #   include ActionController::Live
14   #
15   #   def stream
16   #     response.headers["Content-Type"] = "text/event-stream"
17   #     sleep 4
18   #     response.stream.write "hello world!"
19   #     sleep 3
20   #   end
21   # end
22 end
```

I checked out Rails' implementation of SSE and Roda's and took the average of the two.

100 lines of code!
(with documentation)

It turns out to be really simple.

05 LIMITATIONS

What are the limitations of these mechanisms.

	WebSockets	Server Sent Events
Number of parallel connections from Browser	1024	6
Load Balancing and Proxing	Non-Standard / Complicated	Standard / Easy
Supported on all browsers	Yes (90%)	No (84% - not on IE and Edge)
Dropped Client Detection	Yes	No
Reconnection Handeling	No	Yes

You can have a large number of WebSocket connections while you can only have 6 SSE connections per browser tab.

Web sockets are complex to load balance since they rely on two factors, the HTTP server and the WebSocket server, each of which can become a bottleneck. And since they are generic TCP connections that implement custom protocols they can prove to be hard to proxy in any non-standard application. While SSE are easy to load balance since they rely on a single server and are easy to proxy since they are ordinary HTTP connections.

WebSockets are widely supported on all platforms, while SSE aren't supported on IE and Edge. This isn't a problem though, since SSE can easily be polyfilled.

WebSockets provide a mechanism to detect dropped connections while SSE do not, at least until they try to push data to a client and notice he's gone.

But, SSE handle reconnections, while in WS this needs to be implemented by the server and client respectively.

06

WHEN TO USE IT

Ok. When should you use one or the other?

WebSockets	Server Sent Events
Browser based <u>G</u> ames	<u>T</u> ext Chat
<u>V</u> oice Chat	Push <u>N</u> otifications
<u>V</u> ideo Chat	<u>F</u> eeds

WebSockets are the worst example for a text chat, that's like killing a fly with a bazooka. Text chat is much more suited for SSE.

WebSockets	Server Sent Events
Custom protocol required	Standard protocols apply
Simultaneous sending and receiving	Only receiving

In a nutshell. WebSockets are more suited for custom protocol implementations and applications that require full-duplex connections. So video and voice chat - something that isn't really suited to be transmitted over HTTP. While SSE are more suited for situations where data is pushed to the clients in real-time and can be represented via HTTP.