

DO YOU REALLY NEED WEBSOCKETS

Hi! Spoilers! This question is actually the conclusion of this topic, but we'll get to that. At least once a year I get into an argument with somebody about websockets, this lead me to make this talk. Before we start, please feel free to interrupt me to ask a question or add something. I think it's more interesting for everybody if we have a discussion. So, lets start!



Stanko Krtalic Rusendic

 github.com/stankek

 [@monorkin](https://twitter.com/monorkin)

 hey@stanko.io

 stanko.io

A quick introduction is in order. My name is Stanko, I'm a software engineer at FloatingPoint where I mostly work with Elixir. Though Rust and Ruby are both my fortes. You can reach me via any media listed here, and if you use Mastodon my twitter and mastodon are the same, I'm on mastodon.social

01 THE REAL TIME WEB

How many of you have ever worked on a project that had any real-time component? Show of hands! Note that notifications count as real-time for this purpose. [taunt crowd response] This is usually how this dreaded discussion starts. A project, at some point, requires some real time component. Somebody on the engineering team instantly shouts “websockets” and it’s settled! Right?

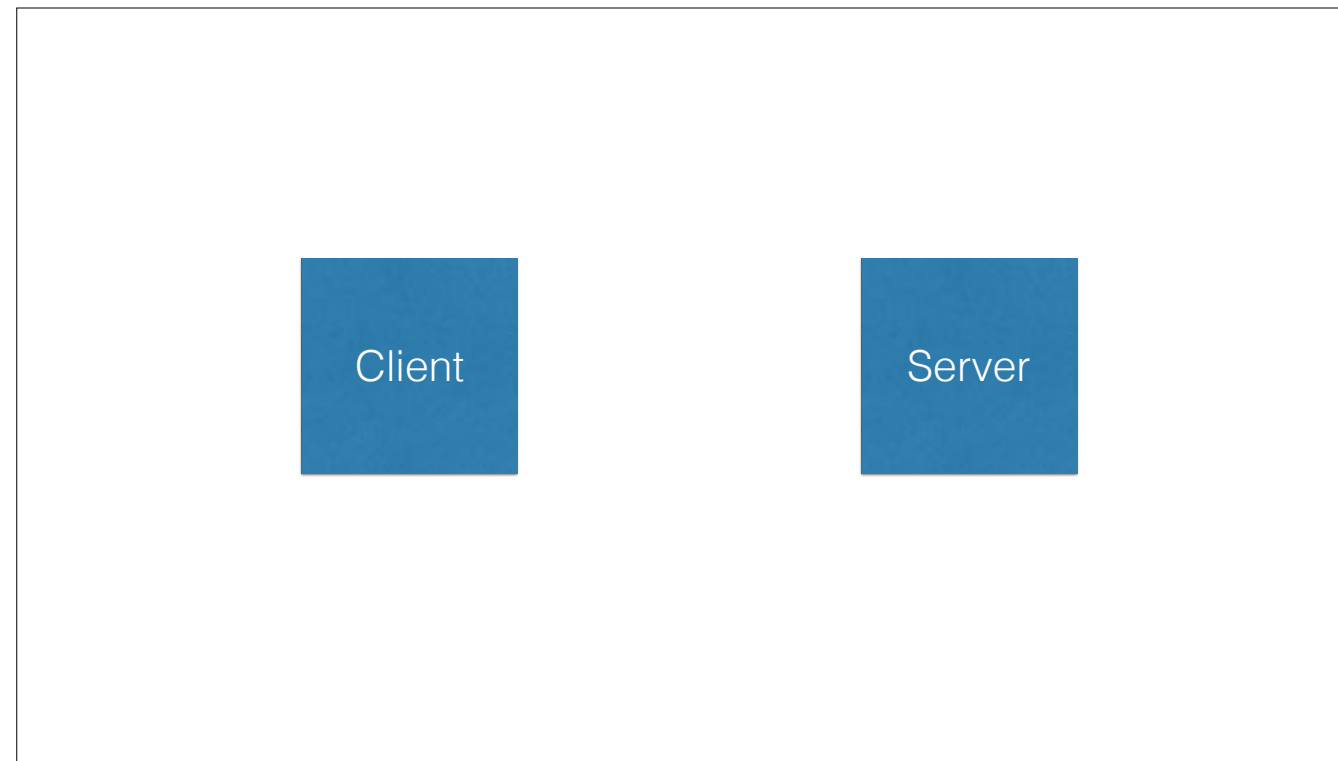


WebSocket are cool

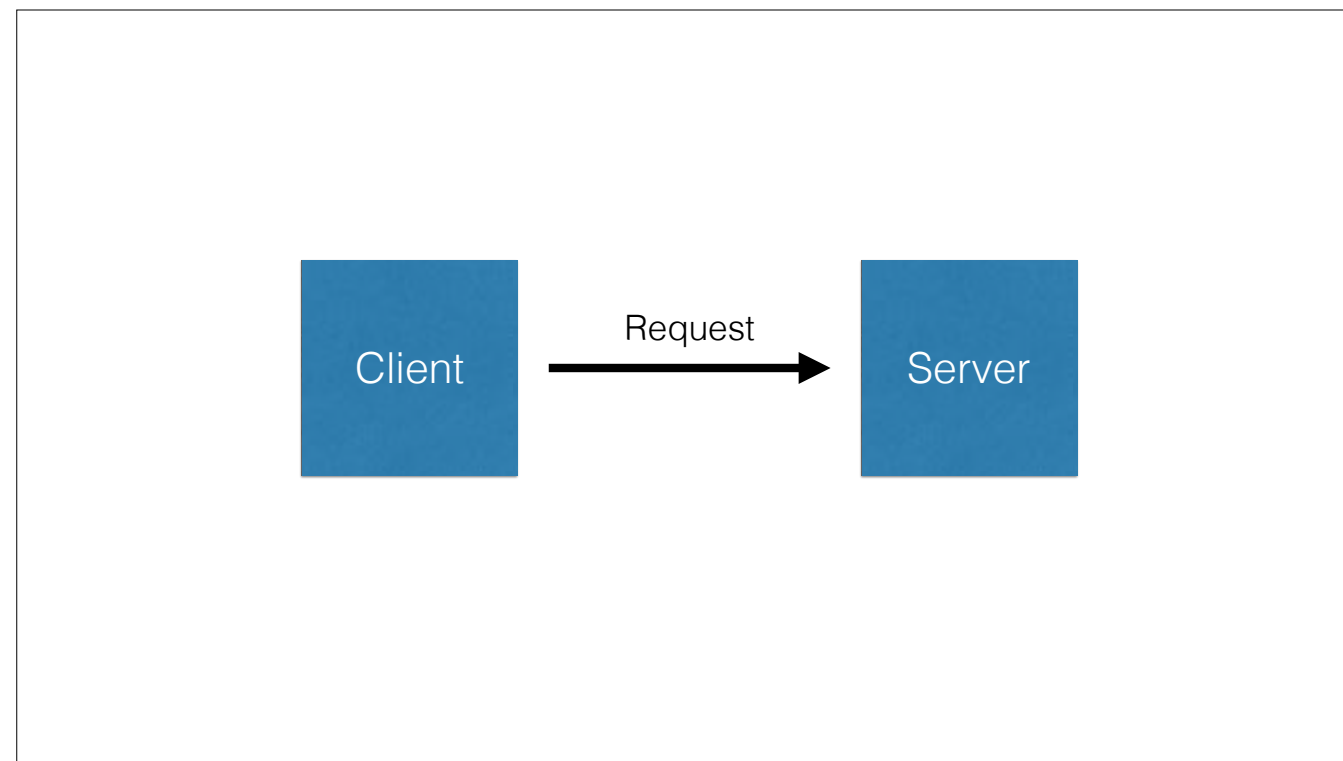
I mean... WebSockets are really cool. They enable us to break free from the standard web request-response cycle and do something really cool.

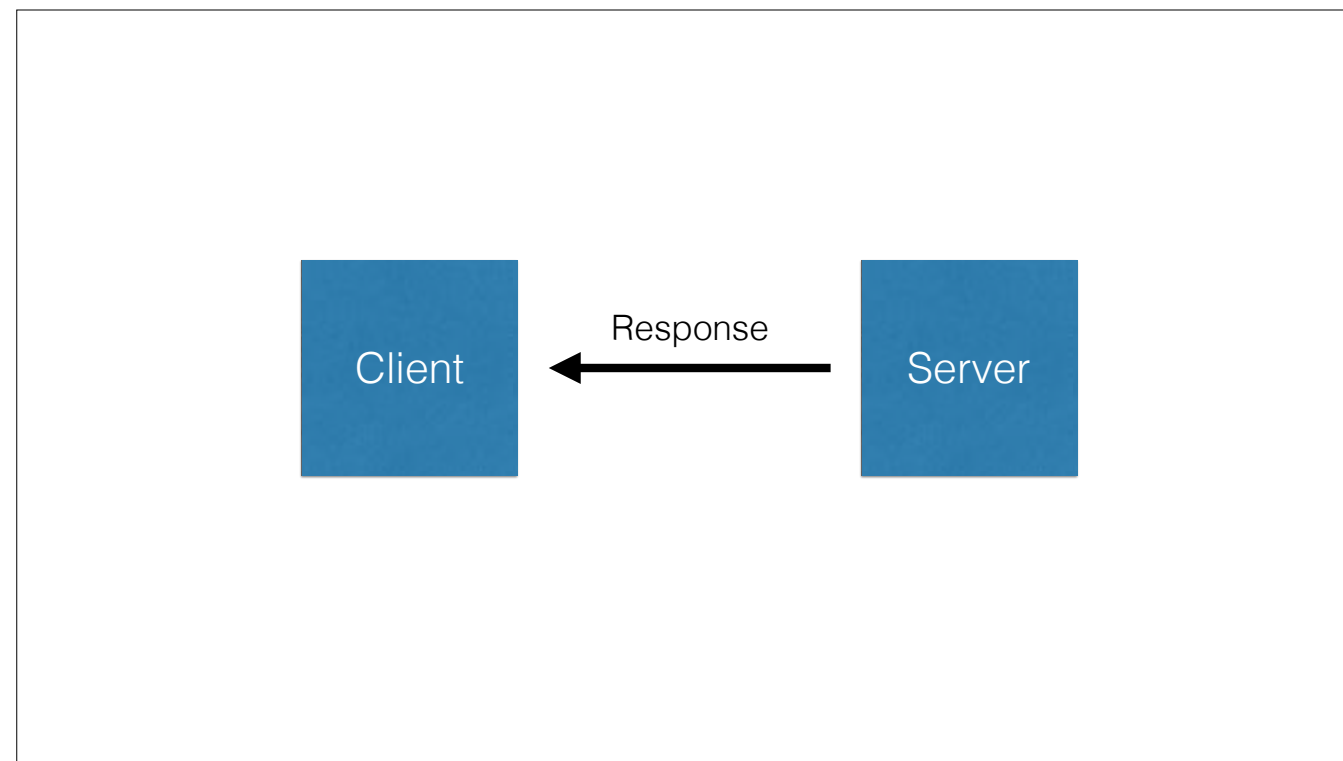
Asynchronous communication

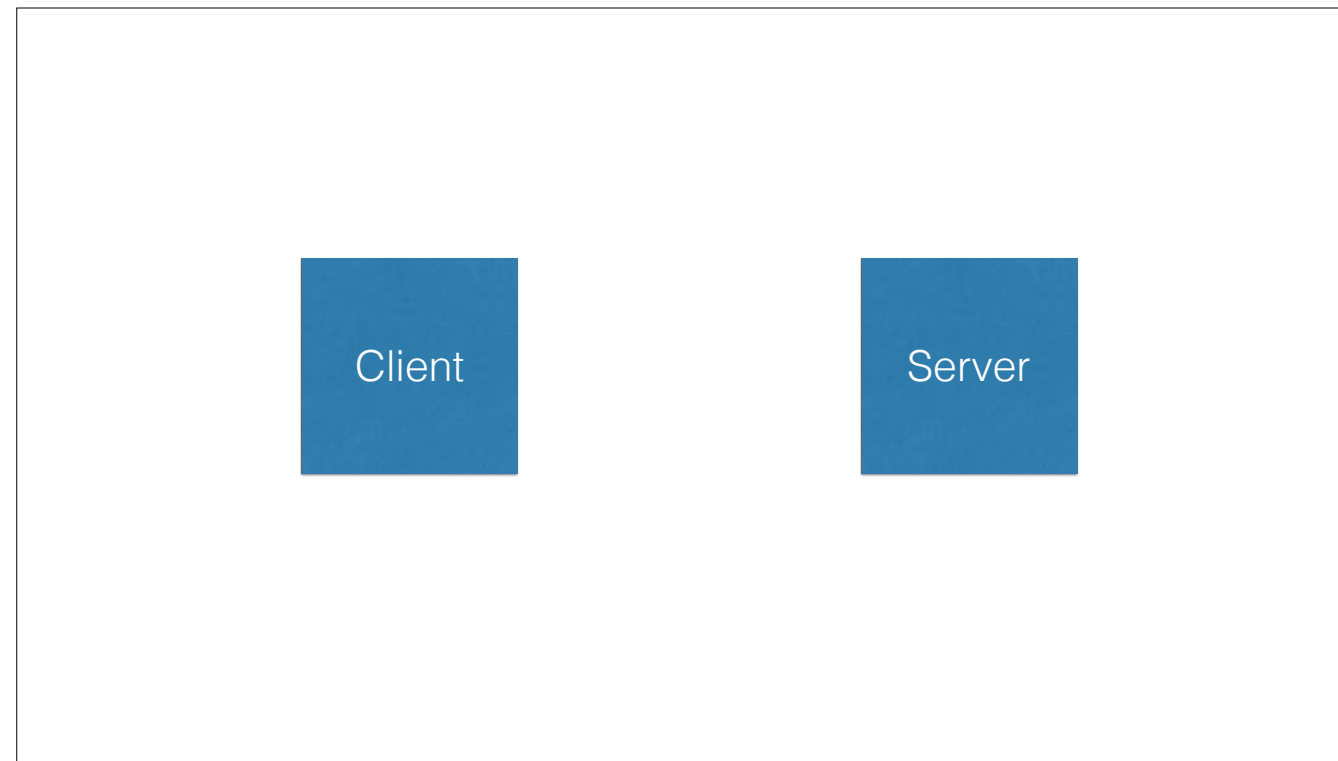
With them the browser and the server can communicate asynchronously.



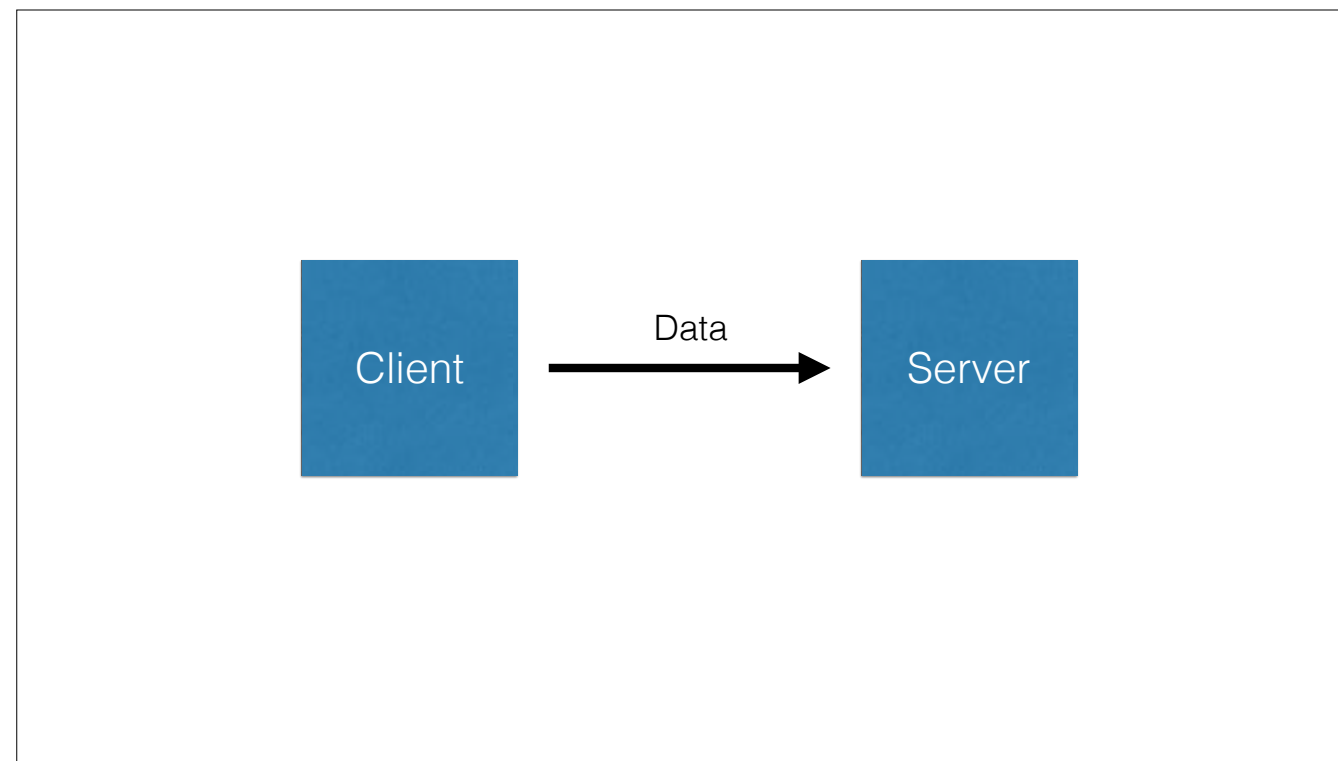
What I mean by asynchronously is that instead of the client initiating a request and waiting for a response

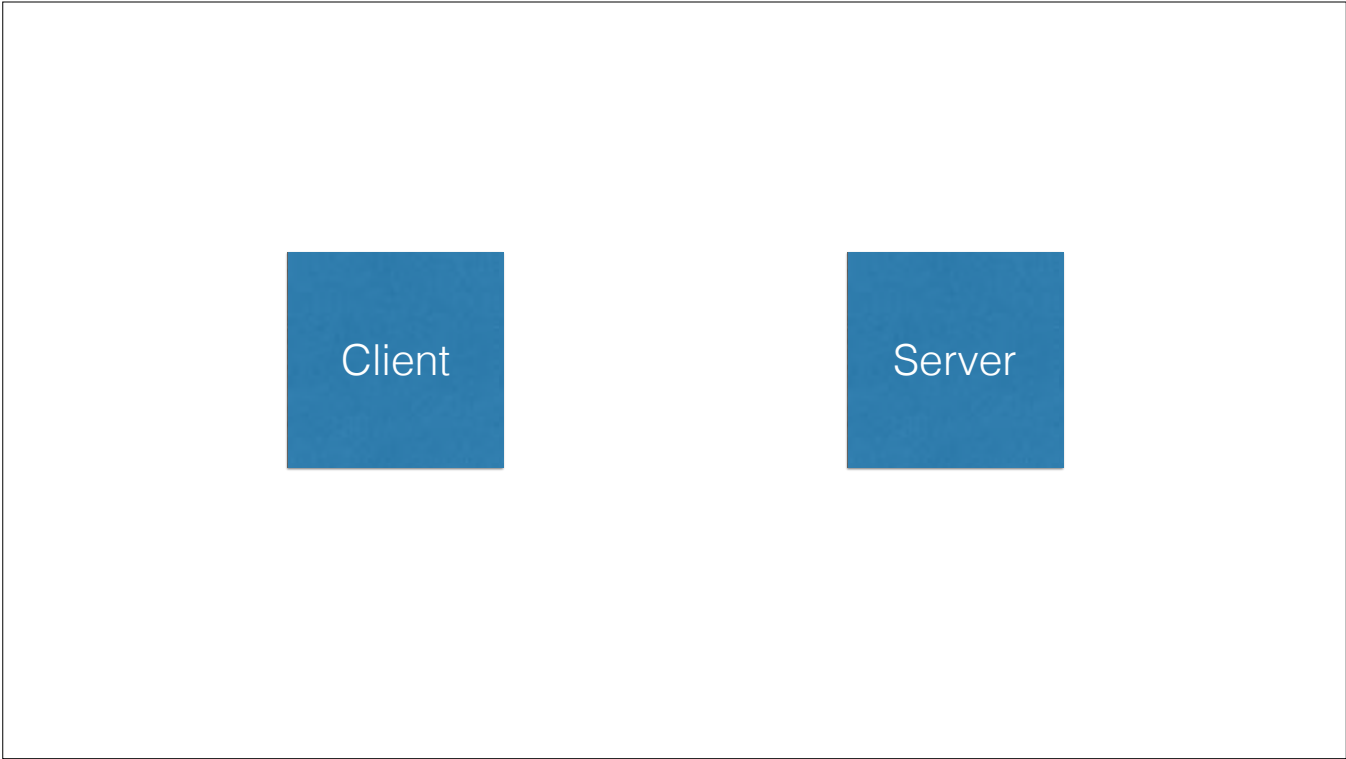


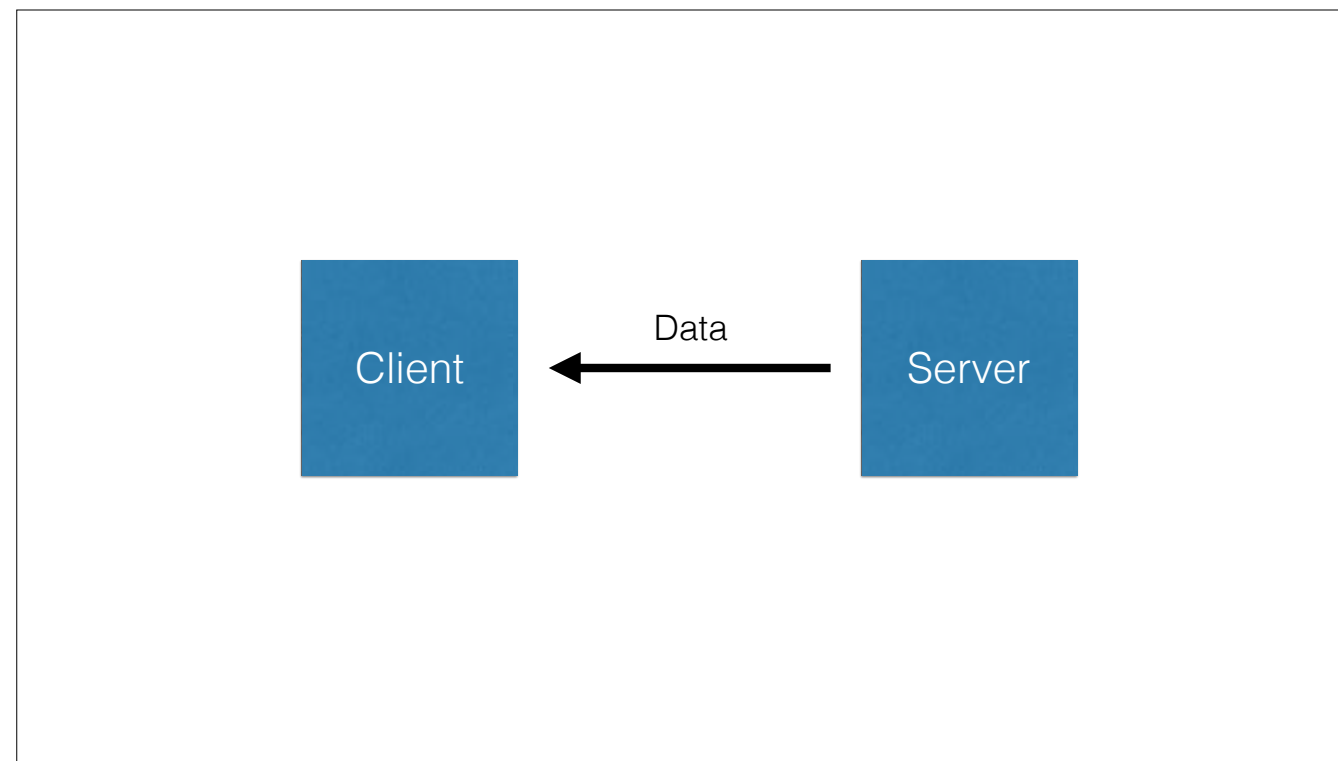




The client can fire-and-forget. And the server can send data at any time.







Full-duplex

This model of communication is classified as a full-duplex connection. This means that all nodes (in this case the client and the server) can communicate with one another at any time and simultaneously. This is analogous to a telephone call. After the call is made, all participants can talk to each other at their leisure and over one another.

Realtime client-server communication

WebSockets enable the server and client to send messages to each other at any time, after a connection is established, without an explicit request by one or the other.

Introduced in 2011
RFC 6455

WebSockets were introduced in 2011 in RFC6455 by Google. Google used them originally for it's Wave project which combined elements of email and IM. It was neat, the idea was lifted straight from Firefly. You could even see messages as they were being written by the other person.

The protocol is...

[read]

...interesting

[read]

Each WebSocket starts
as a HTTP request

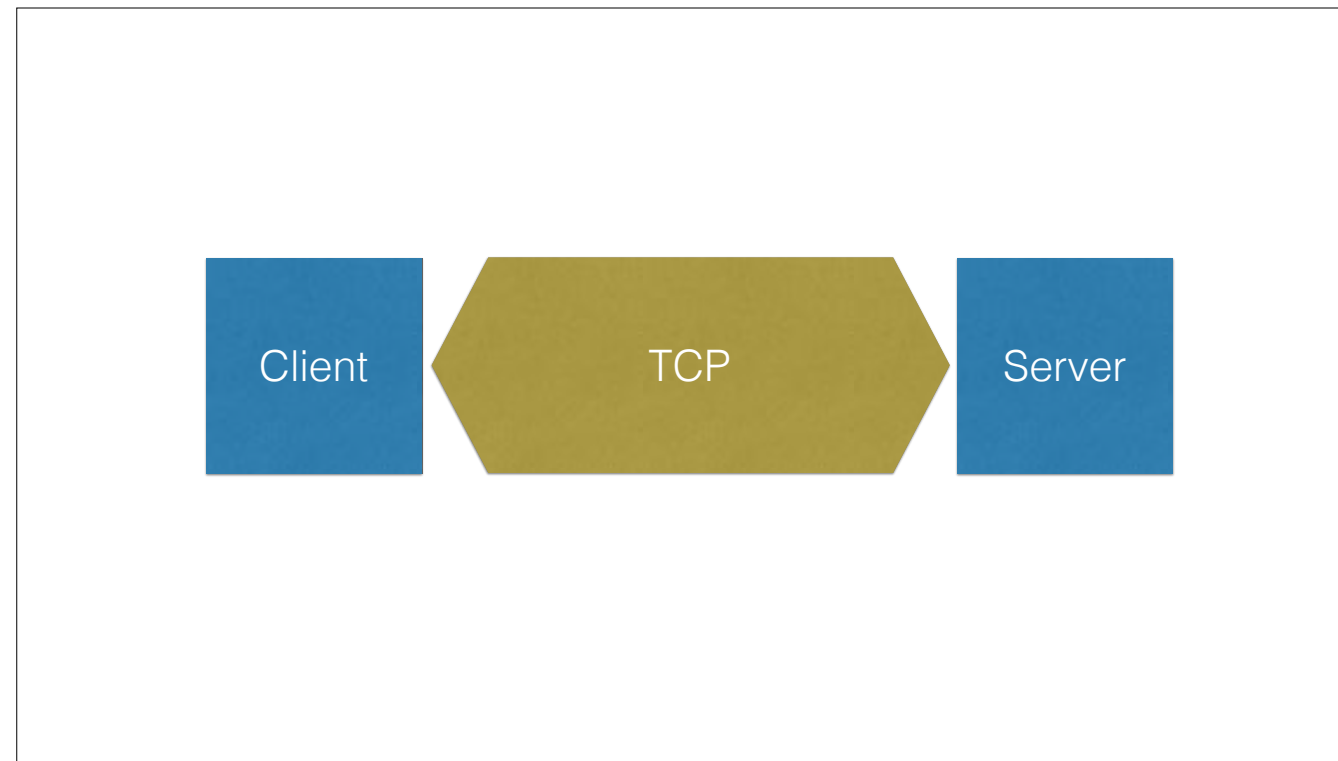
[read]

But ends up as a WebSocket?

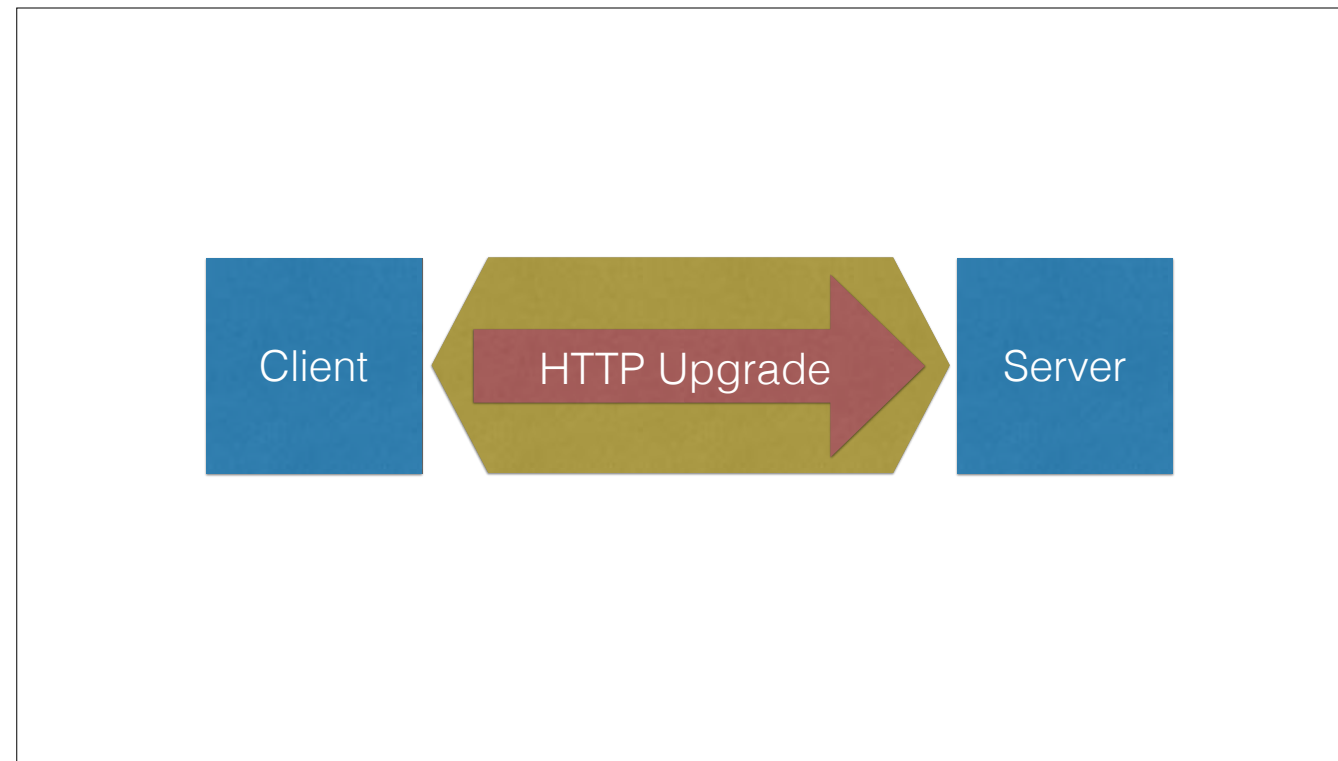
[read]



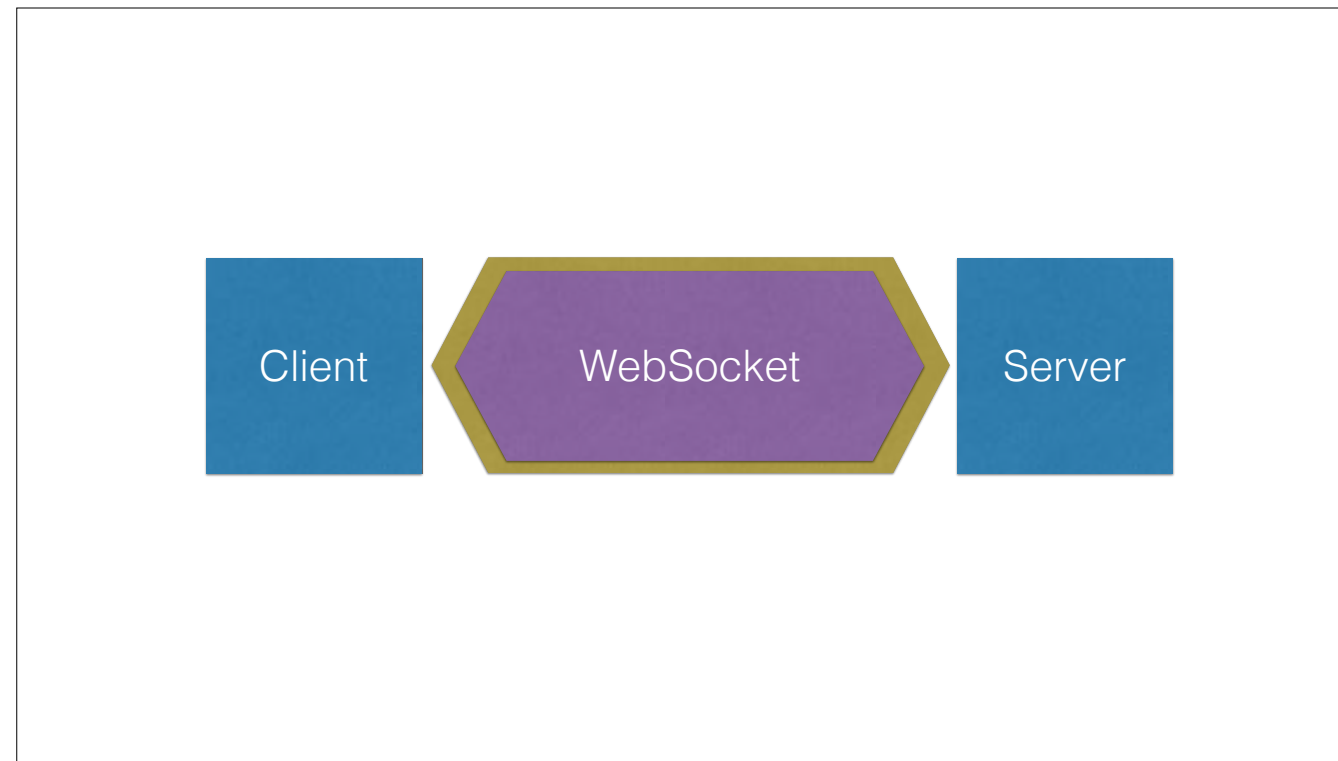
The question arises, why does that happen. So here is the protocol



First a TCP connection is established between the client and the server.



Through that TCP connection a HTTP request is made. It's a special kind of HTTP request called upgrade which was introduced with HTTP 1.1. It's intended to be used to switch protocols, in the case that the HTTP protocol gets updated in the future, but it allows upgrades to other protocols as well. In this case we want to switch from HTTP to the WebSocket protocol. But, since the initial request is HTTP this means that we can utilise our existing authentication and authorisation mechanisms and provide the websocket session with extra information about the connecting user. This also allows servers that don't support the protocol to handle those connections, and error gracefully.



After the server upgrades the connection



Part of HTML5 spec

WebSockets are part of the HTML5 spec.


```
1 // Create WebSocket connection.
2 const socket = new WebSocket('ws://localhost:8080');
3
4 // Connection opened
5 socket.addEventListener('open', function (event) {
6   socket.send('Hello Server!');
7 });
8
9 // Listen for messages
10 socket.addEventListener('message', function (event) {
11   console.log('Message from server ', event.data);
12 });
```

websocket-demo is hosted with ❤ by GitHub [view raw](#)

WebSocket JS API docs from <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>

That means that most browsers have a native API to handle websocket connections.



Up to 1024 connections

A single tab can have up to 1024 websocket connections open. I'm not sure what for you would need that many, but they are there.

Non standard proxying/LB

Though, since websockets aren't HTTP connection, most proxies and load balancers won't work with them unless configured to do so.

Dropped client detection

WebSockets can detect when a client joins or leaves. This is useful for presence tracking.

No reconnection handling

Though if a client drops no native websocket implementation will automatically reconnect them.

What are they useful for?

[read]

Voice chat
Video chat
Games

[read] Anything that requires a full-duplex connection

Real-time **client to server** communication

They are best suited for situations where the CLIENT has to push data to the server in real-time.

02 OVERKILL SOLUTION

WebSockets present an overkill solution in all other cases

Separate controls flow from the HTTP server

Assuming you have a well structured application. To add websockets to your project you will have to create a separate controllflow to handle user connections and room membership, as well as session management and broadcasting rules.

More complexity

It introduces more complexity into your app. Since now you have two separate presentations of your app, and your business logic has to adapt and become more abstract. Basically you are introducing a sub-app in your web-app.

Alternatives?

There are a couple alternatives to websockets that may be better suited for your problem.

03 SERVER SENT EVENTS

Server Sent Events

Introduced in 2006
WHATWG Web Applications 1.0

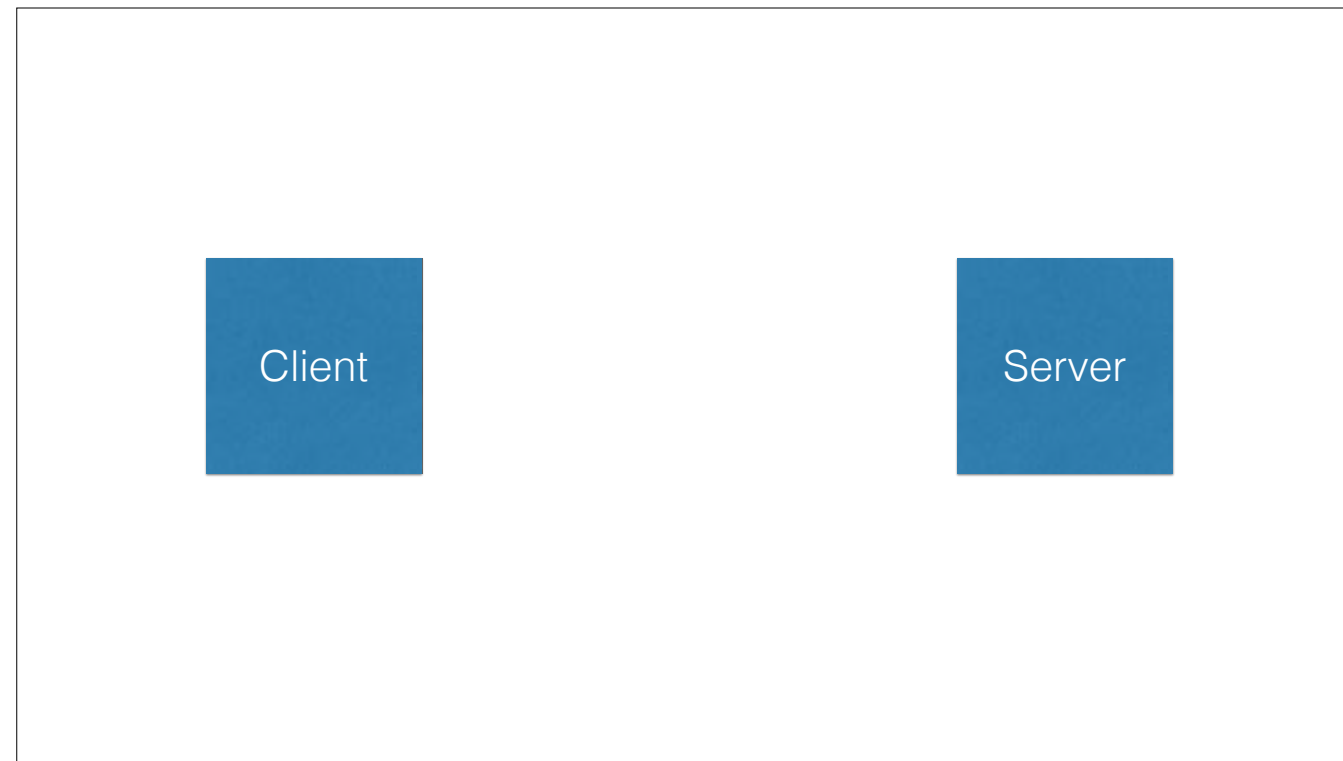
They were introduced to a long time ago. [read]

Only Server to Client communication

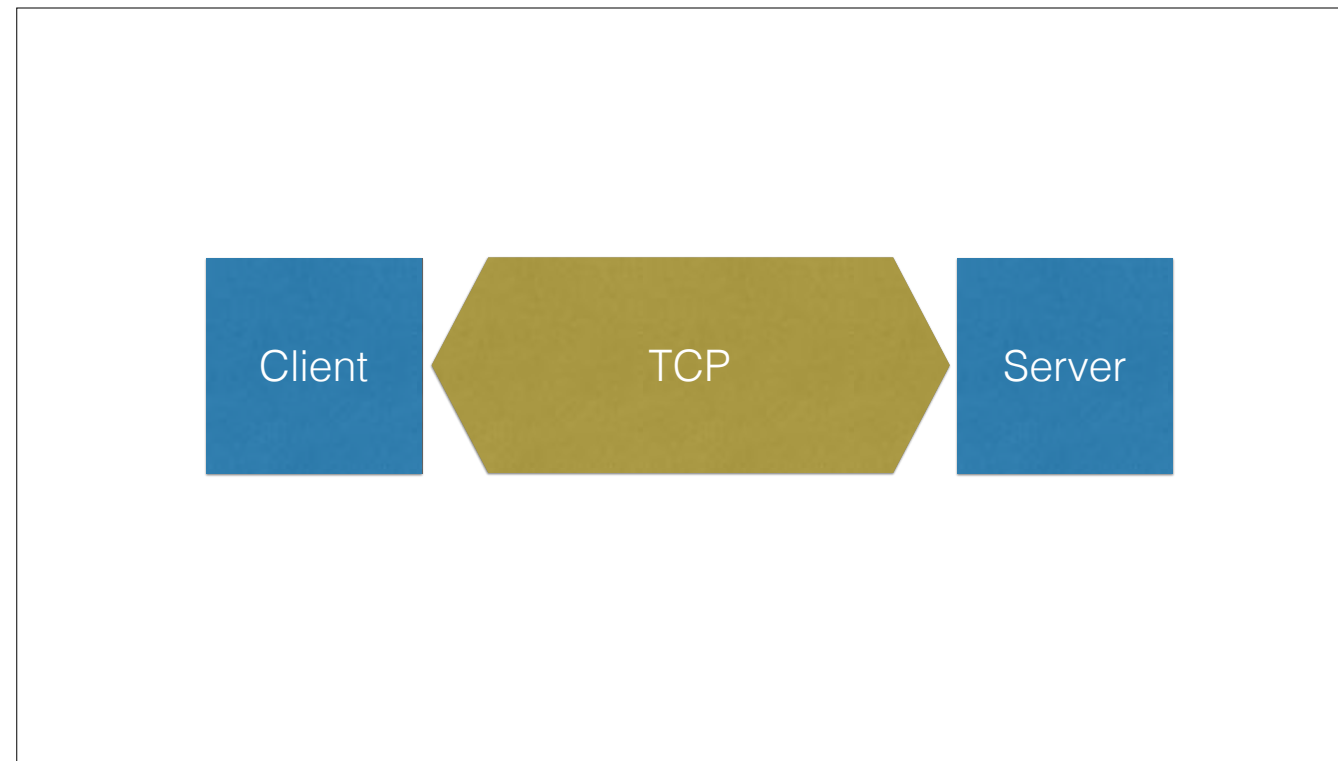
They only provide a way for the server to push data to the client in real-time.

HTTP based
Relies on HTTP streaming

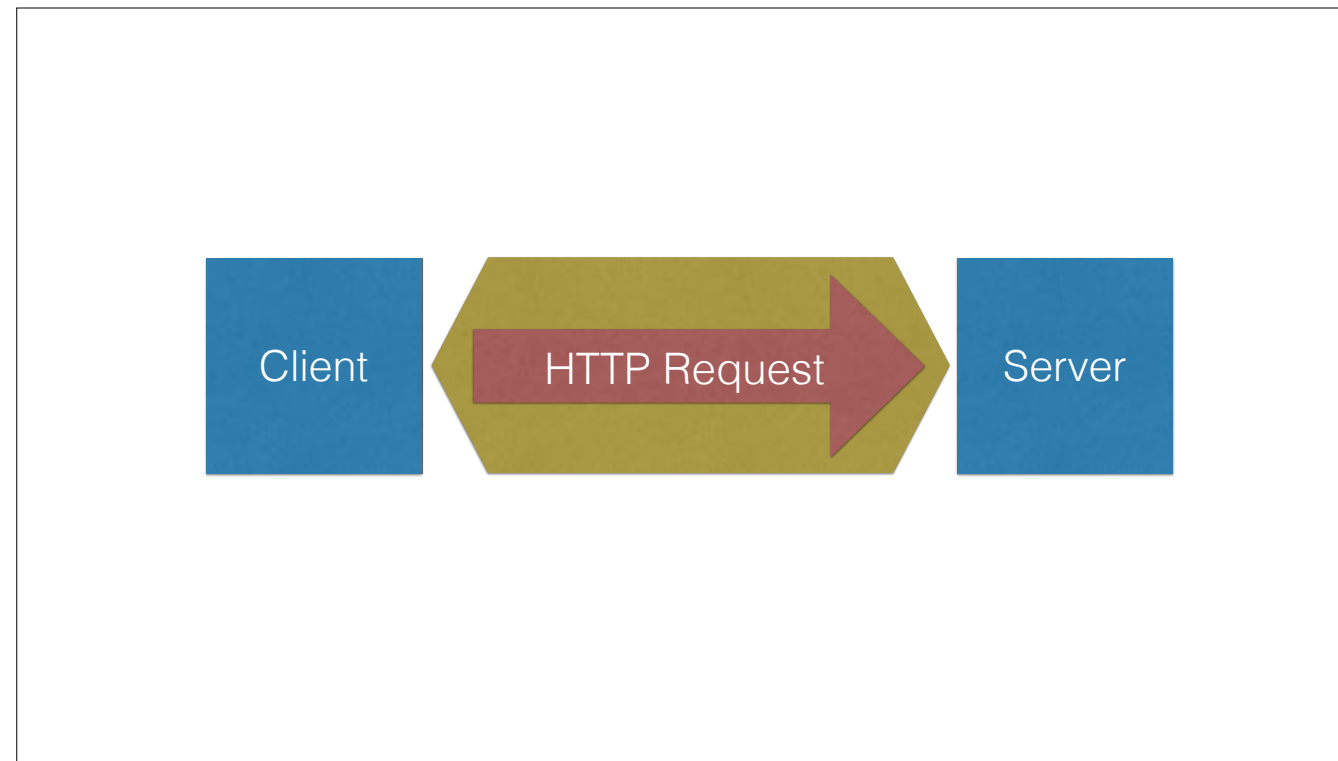
It's completely HTTP based and is implemented with ordinary HTTP streams.



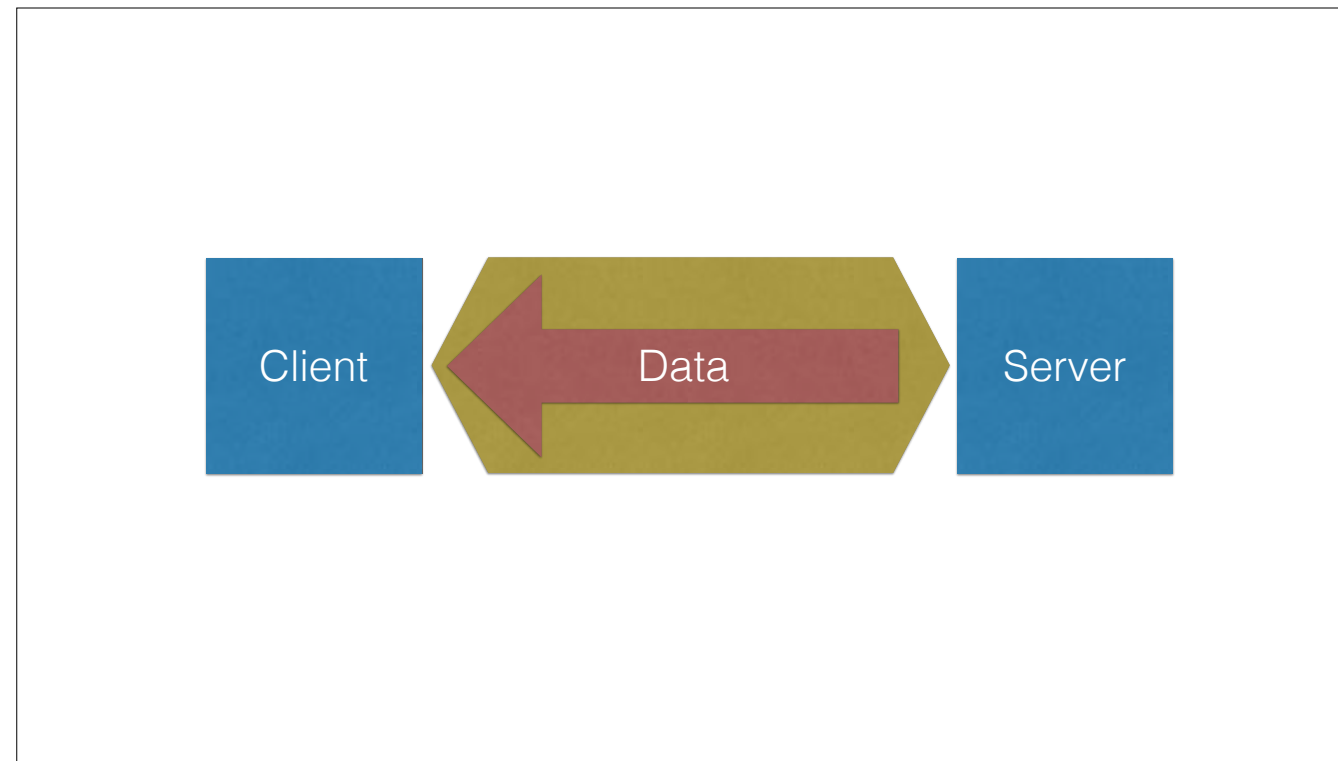
They work exactly like regular old HTTP



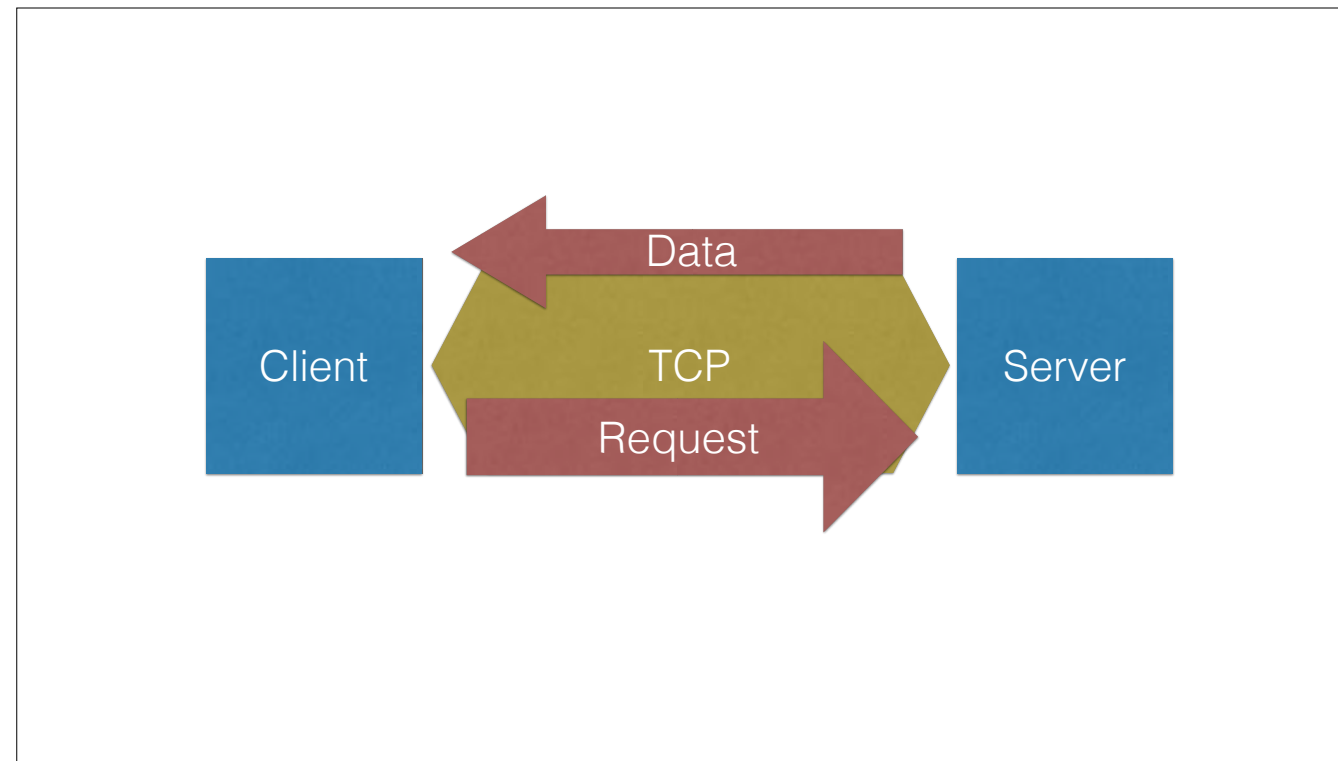
First a TCP connection is established between the client and the server.



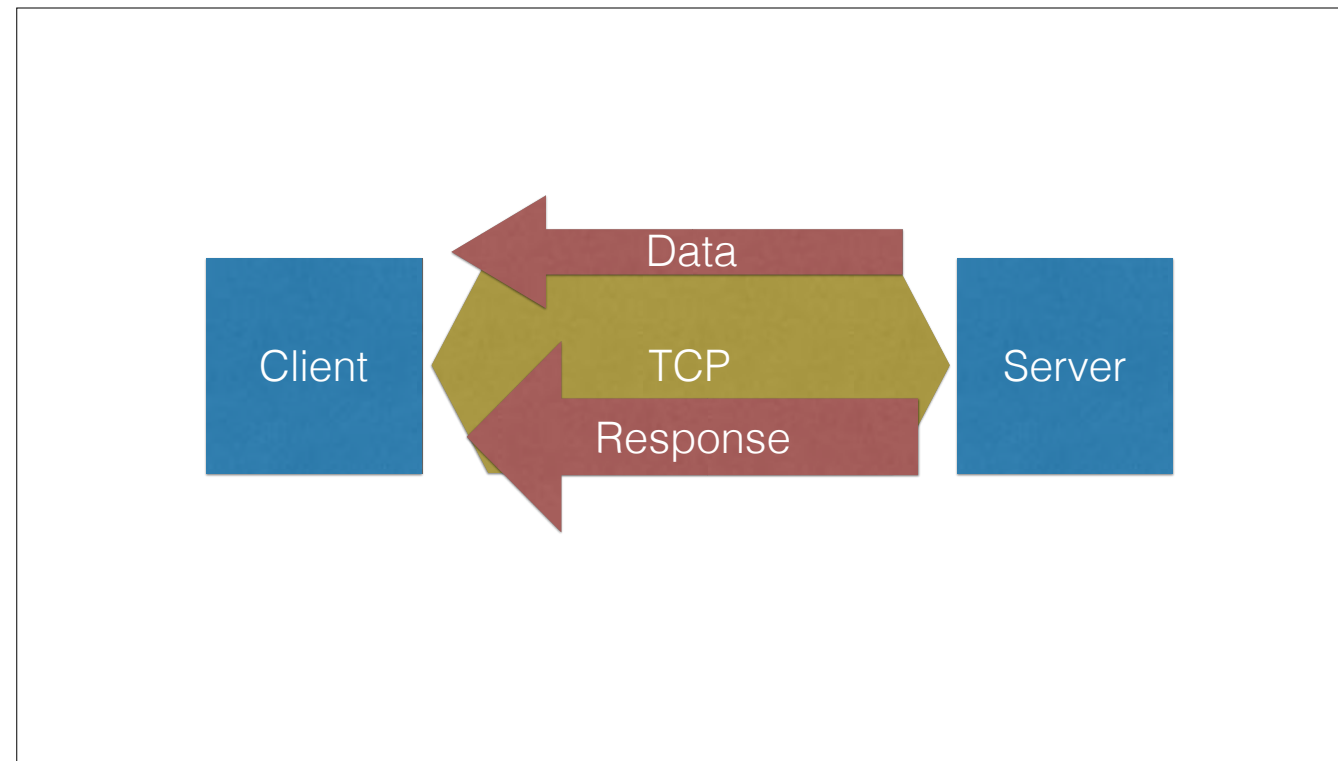
Then a regular HTTP request is made to the server. Again, as with websockets we have access to all standard session information, authentication and authorization mechanisms we are used to.



Then a regular HTTP request is made to the server. Again, as with websockets we have access to all standard session information, authentication and authorization mechanisms we are used to. Since the connection is a stream, the server can now send data at any time.



If the client wants to send that to the server. It's possible to do so by making a regular POST request



To which the server responds as usual.

A solid blue rectangular box with a thin black border, centered on the page. Inside the box, the text "Not supported on IE / Edge" is written in a white, monospaced font.

Not supported on IE / Edge

Sadly, SSE is not supported on MS platforms. But is available through a polyfill.

Subject to the ~6 connection
limit

If you didn't know. Most modern browsers limit the number of connections a page can make to a domain to about ~6. For performance reasons. SSE counts into that limit.

Automatic reconnection handling

SSE will handle client reconnection for you. This is a feature that most Web Socket libraries must implement.

A solid blue rectangular box with a thin black border, centered on the page.

No dropped client detection

But SSE can't detect if a client has stopped listening to the server. This can be solved by keeping a heartbeat.



Benefits?

Though an objectively inferior technology SSE provide a few benefits.



Same controls flow

You don't need a separate control flow for SSE, you can just add another controller to your web app and you are set.

Simple protocol

Since SSE is just HTTP streaming, most web developers will already be familiar with them.

What are they useful for?

[read]

Text chat Notifications

[read] Anything that requires a half-duplex connection.

Real-time **server to client** communication

Half-duplex meaning anything that requires the server to push data to the client in real time.

04 LONG POLLING

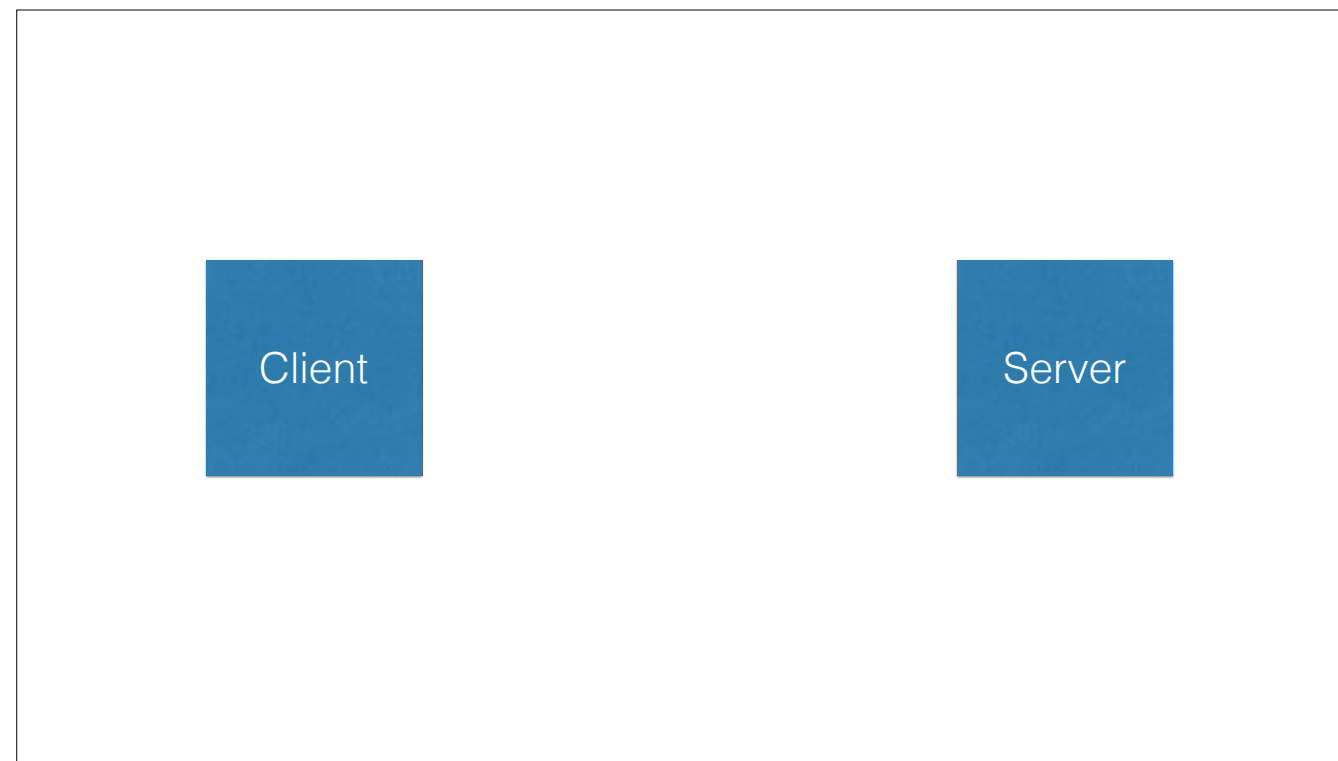
There is another alternative to websockets, and that is long polling.

Pure HTTP

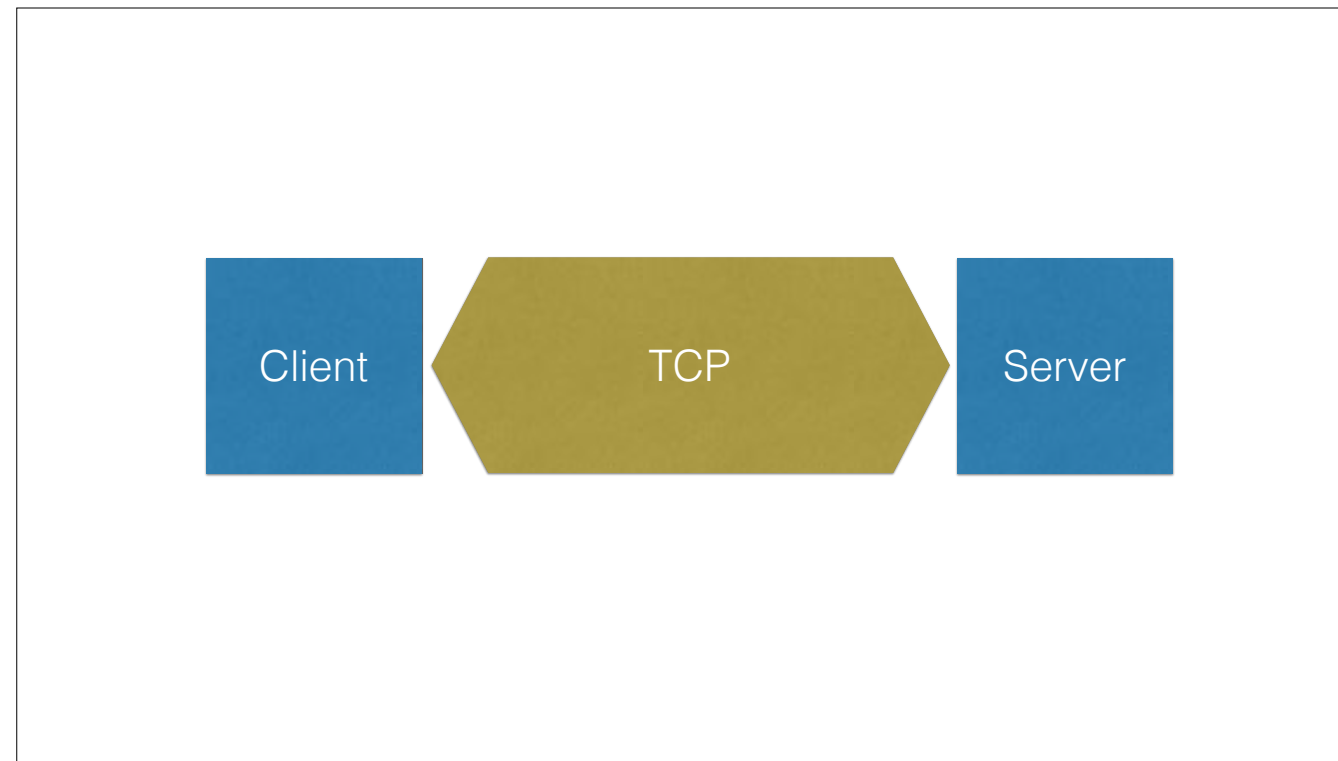
Long polling is pure HTTP

The server delays the
response

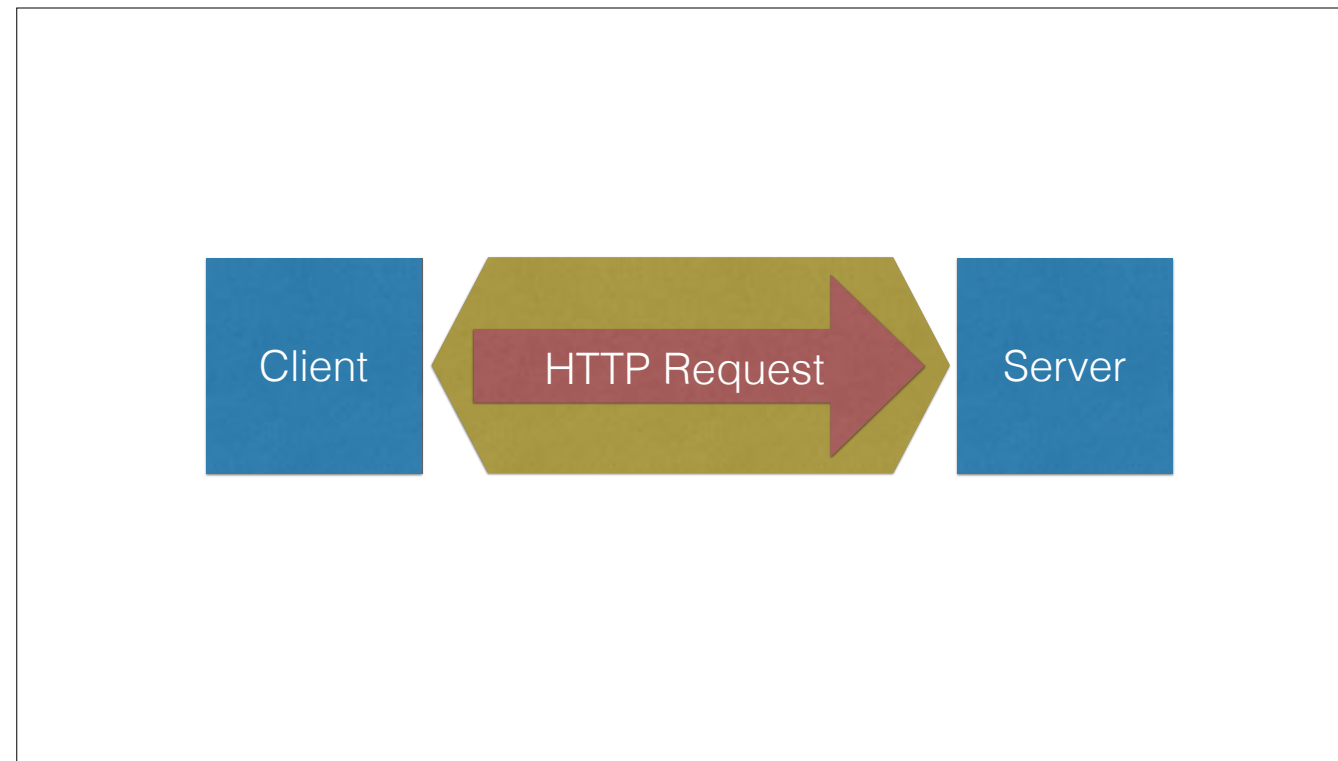
Long polling is pure HTTP



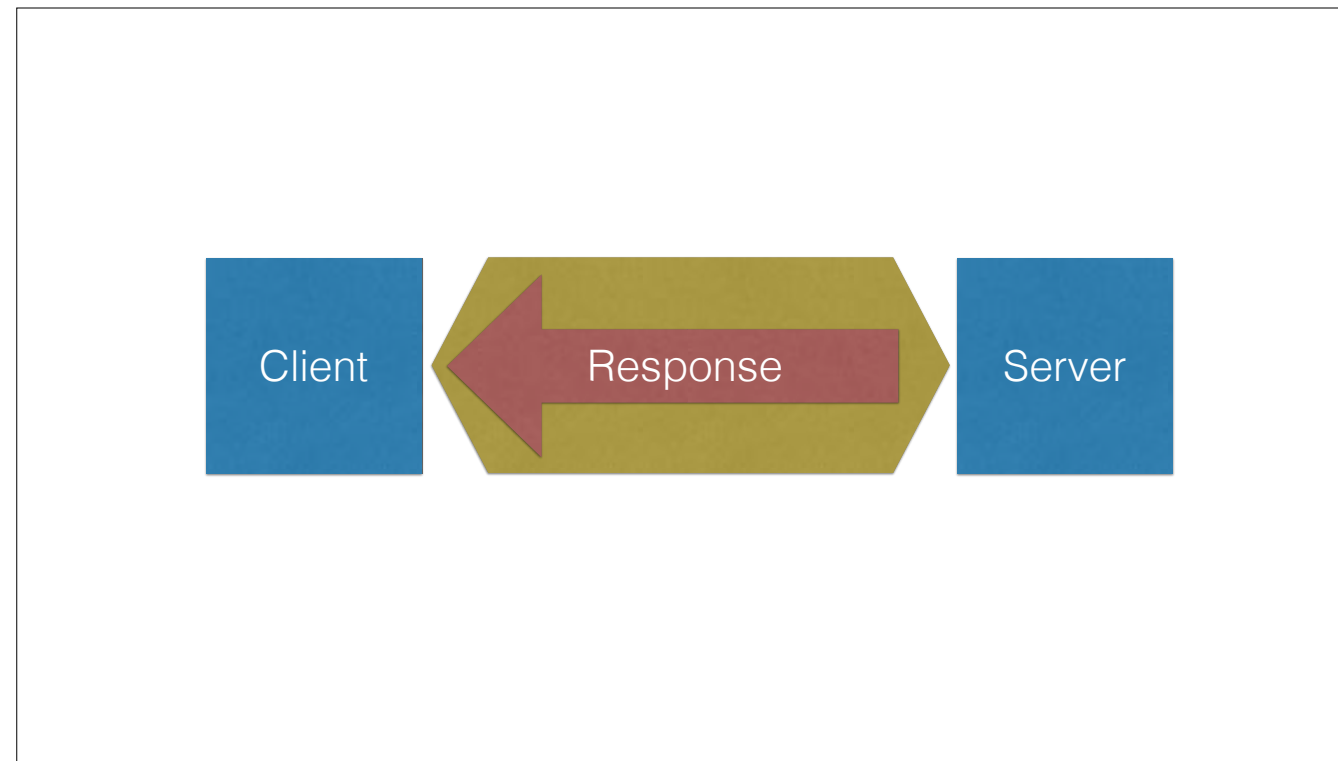
Again, we have a client and a server.



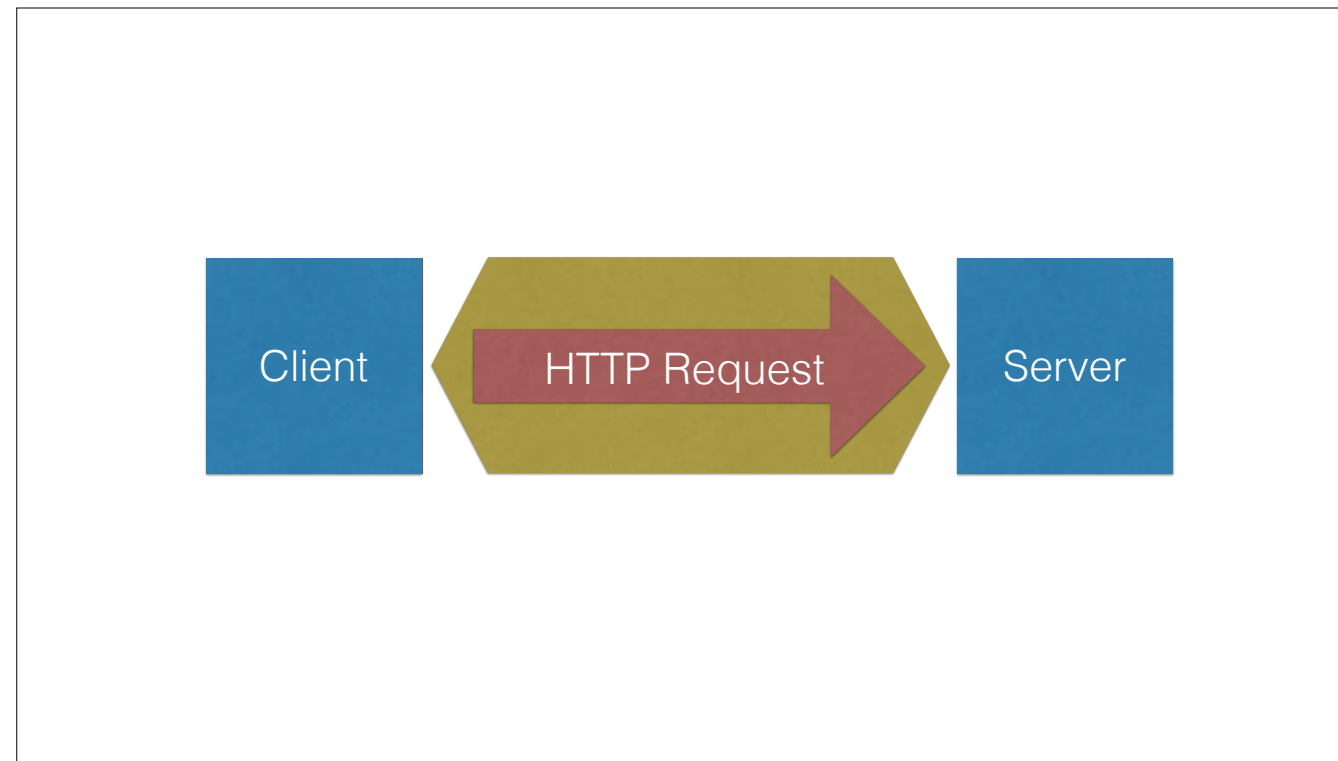
A TCP connection is established between them.



The client makes a request.



Now. If the server has data for the client it responds immediately. If it doesn't have any data it keeps the connection open and sleeps. If new data arrives while the server is sleeping, it gets sent to the client as a response.



As soon as the client gets a response it creates a new request, and the cycle starts again. This also happens if the server waits for too long and the connection timeout.

Subject to the ~6 connection
limit

Same as with SSE

Blackout handling

Because the client and server don't have a constant connection open to each other, there are blackouts. The server needs to accumulate data that it hasn't yet sent to the client, and push it when the client reconnects.



Benefits?

So why would someone pick long polling over SSE?



Supported everywhere

Since they are just plain HTTP they are supported everywhere. Though that doesn't mean that there are browser APIs for them.



Same controls flow

You don't need a separate control flow for SSE, you can just add another controller to your web app and you are set.

Better for large numbers of
users

They are arguable better for large numbers of users. I don't have experience this, but the reasoning is that since LP periodically releases connections to the server it's harder to starve the server of resources. This is partially supported by the fact that Facebook's Messenger uses long polling.

What are they useful for?

[read]

Same as SSE

[read]

05 COMPARISON

Those are all available methods for client-server real-time communication. So let's compare them.




	WebSockets	Server Sent Events	Long Polling
Number of parallel connections from Browser	1024	~6 per domain	~6 per domain
Load Balancing and Proxying	Non-Standard / Complicated	Standard / Easy	Standard / Easy
Supported on all browsers	Yes (90%)	No (84% - not on IE and Edge)	Yes (100%)
Dropped Client Detection	Yes	No	No
Reconnection Handling	No	Yes	No

[read]

06 CONCLUSION

There is no clear winner here. The take away is that most of the time you only do real-time server to client pushes, and therefore websockets are an overkill solution. If that is your case, and you are thinking about which technology to choose, go with SSE. Its simple and quick to implement, you can deliver the feature quickly and test the market. If your user base grows, go for LP. If you suddenly get the need to send data back to the server in real-time, go for websockets. Note! This doesn't mean you should run off and convert existing applications to SSE. When it comes to pushing data from the server, all technologies are more-or-less the same. Just remember that websockets aren't the go-to-solution for every web-related real-time problem.

QUESTIONS

 @monorkin
 hi@stanko.io
 github.com/stankec

Thank you for your patience. If anybody has questions I'll accept them now.
If you are shy, you can also contact me via Twitter, Mastodon or Email.
The slides are available on my GitHub

blog.stanko.io/343aed40aa9b

If you want to read more on this subject, I wrote a fairly popular blog post. It has animations detailing how the protocols work and links to everything I referenced.