

# DO YOU REALLY NEED WEBSOCKETS

Hi! I'd like to start my talk with a question - do you really need websockets? The goal with this talk is to give you everything you need to know to answer this question for your self.



## Stanko Krtalic Rusendic

🐙 [github.com/monorkin](https://github.com/monorkin)

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

🐙 [monorkin@mastodon.social](mailto:monorkin@mastodon.social)

✉ [hey@stanko.io](mailto:hey@stanko.io)

🌐 [stanko.io](https://stanko.io)

Before we continue, let me introduce my self. My name is Stanko, I'm a software engineer, I mostly work with Ruby, Rust and Elixir. I'm also one of the organisers of the Ruby meet up group in Zagreb.



I'm currently employed at Very Big Things, which is a digital products agency.

stanko.krtalic.rusendic@verybigthings.com

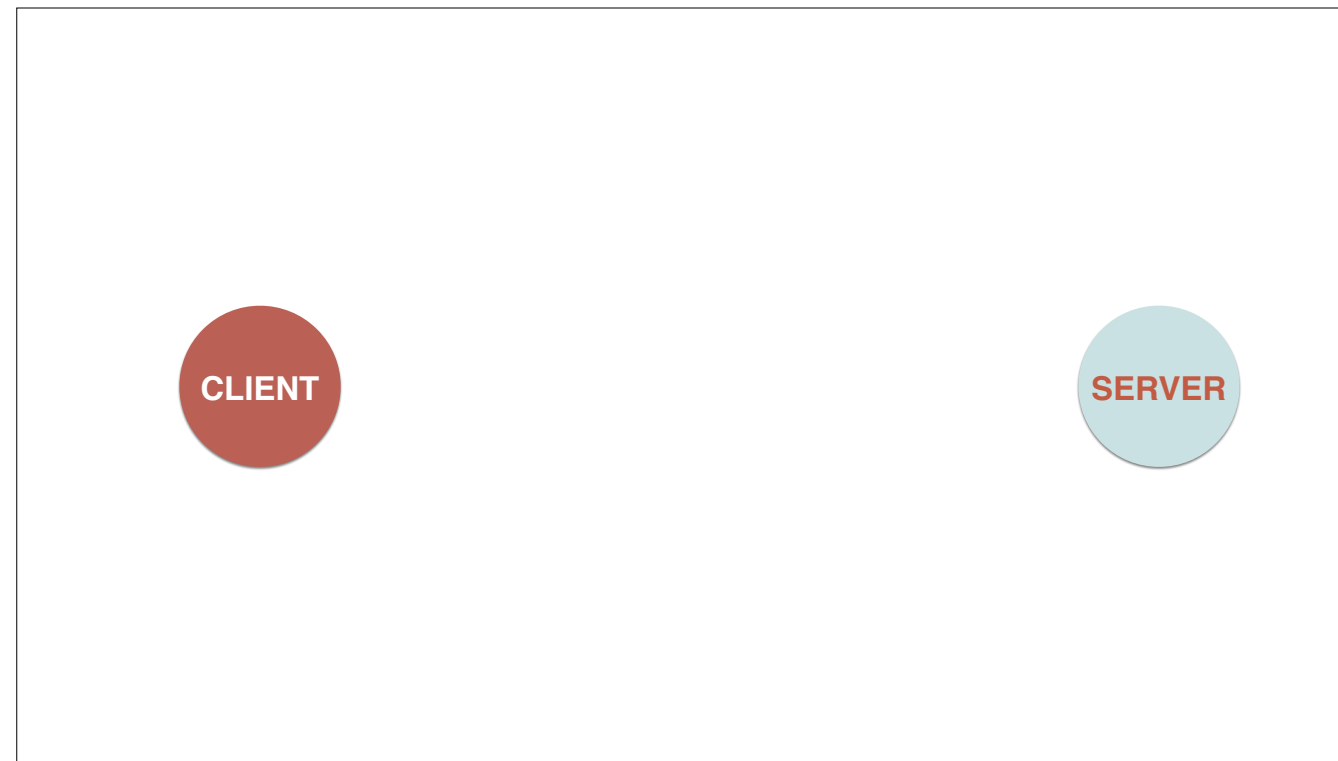
When I got employed there I thought that VBT was just marketing friendly name, but I figured they really mean it when they gave me this email address and no alias for it

# 01 WHY WEBSOCKETS

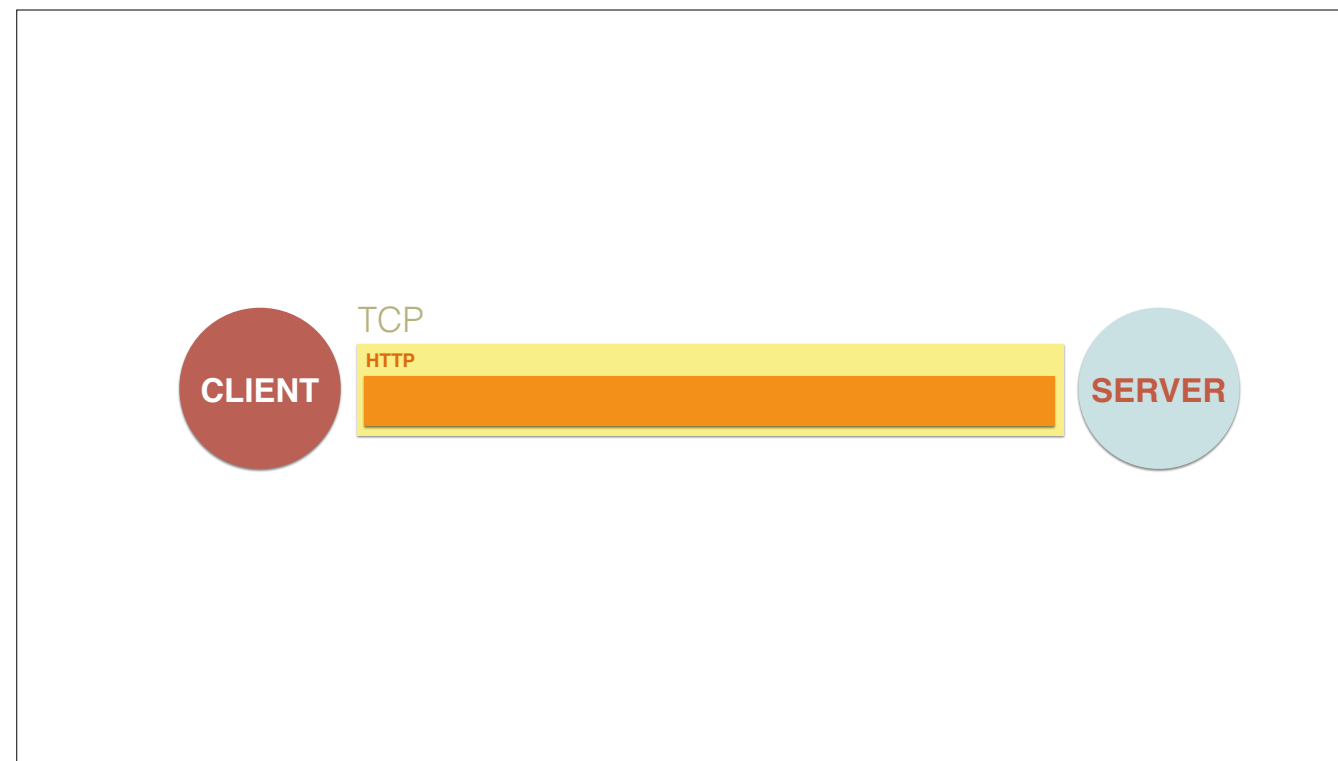
Why do we use WebSockets?

# Challenge - Response

The whole web as we know it is built around the challenge - response principle. This is perhaps better known to backend and frontend engineers as the request - response cycle.

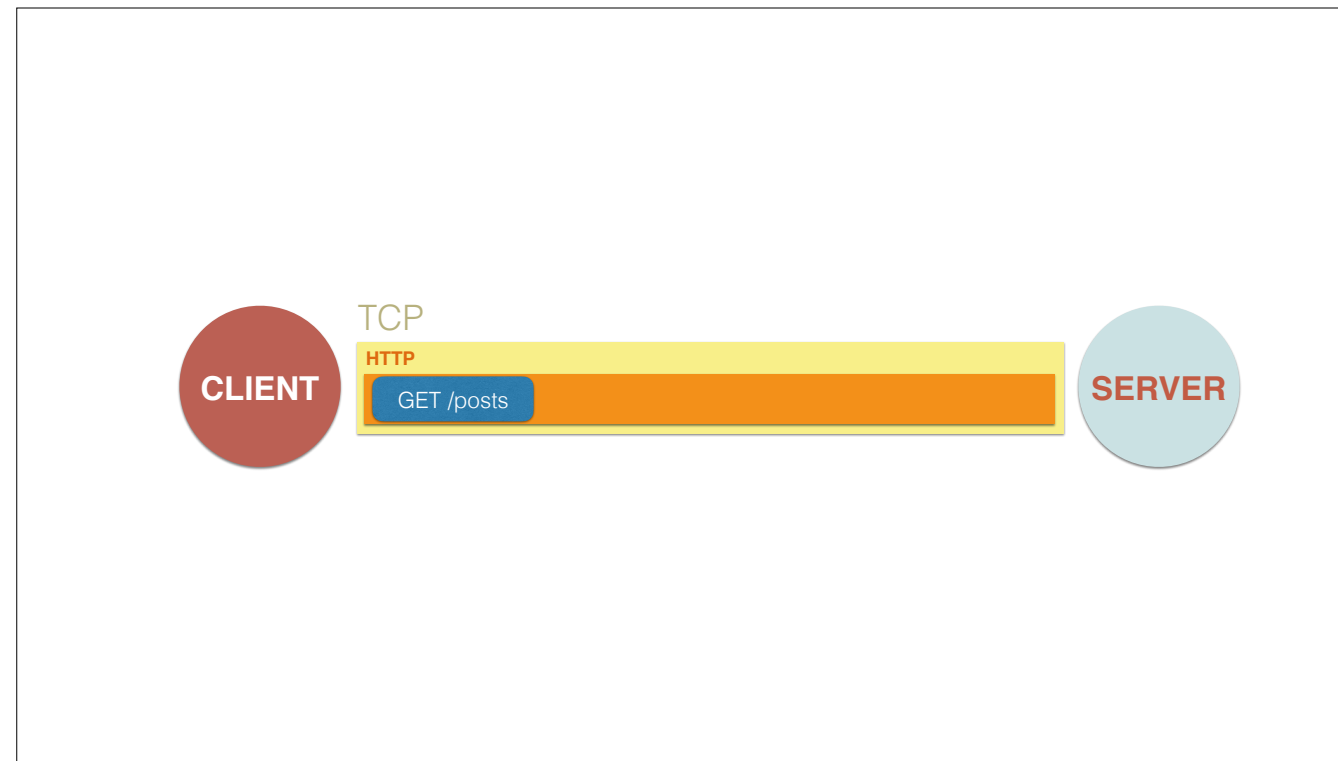


The basic principle of the cycle is that the client asks the server for information it wants to get

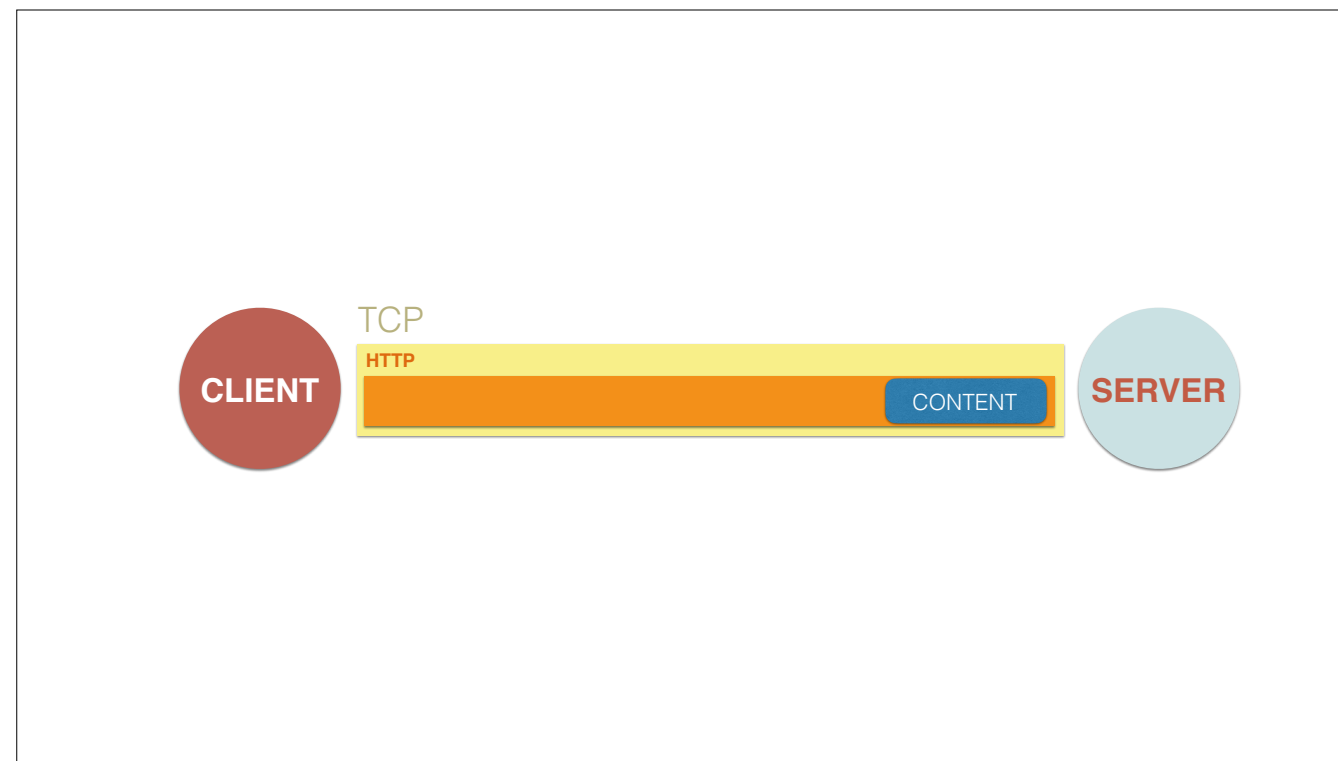


To do so it first establishes a connection to the server





Then it sends it's challenge or request for the server



And the server responds with the requested information

# HTTP is Half-duplex

This mode of communication is also known as a half-duplex connection. Meaning that only one party can send data at any one time. Perhaps the most well-known example of a real-life half-duplex connection are walkie-talkies. HTTP solely uses this mode of communication.

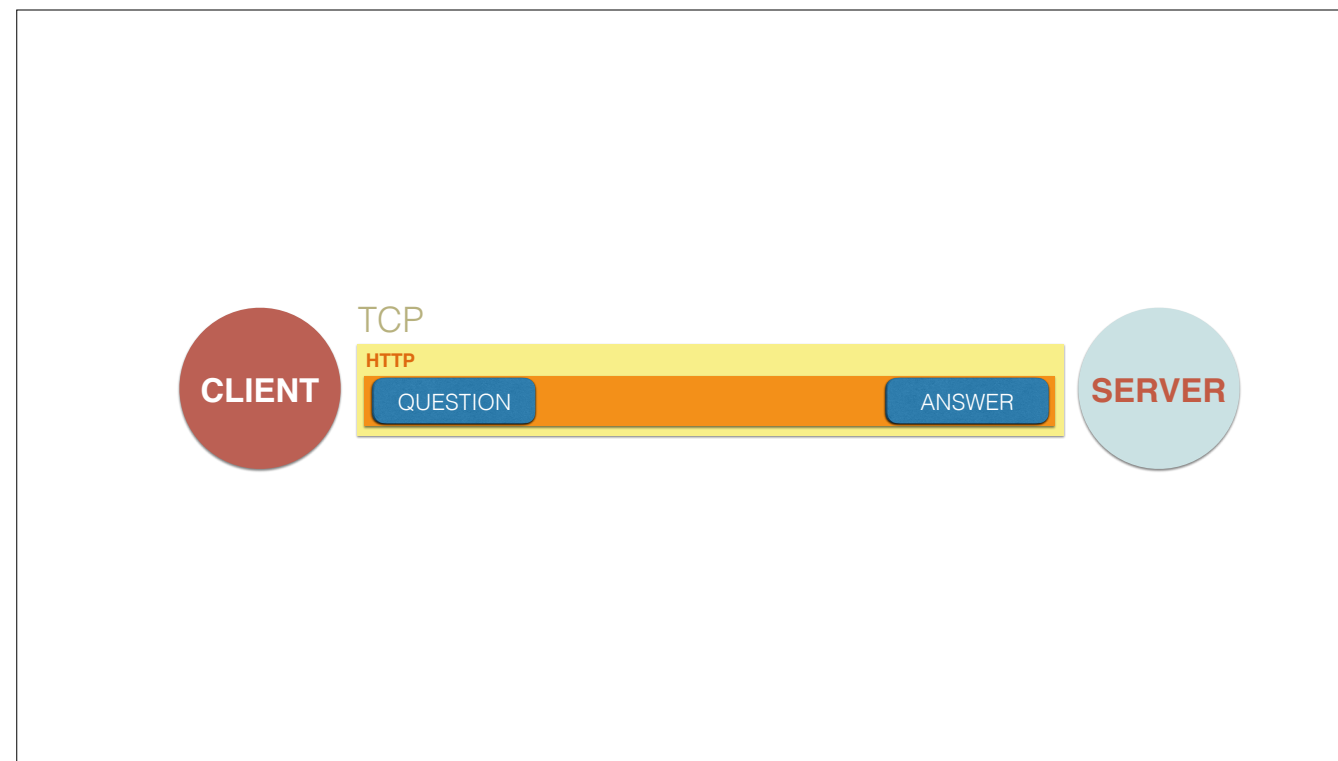


The problem?

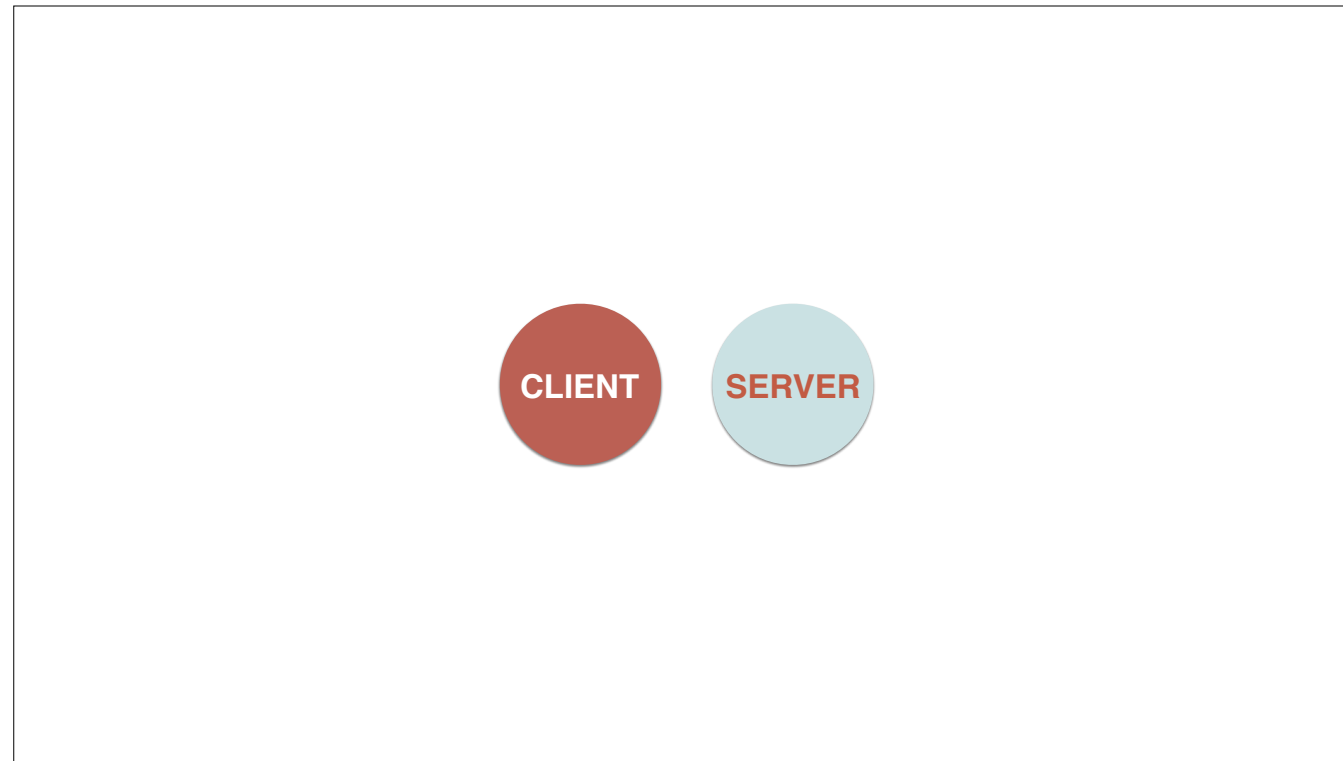
So what's the problem? Let me explain it on an example.



Say you are building a chat. It wouldn't be very useful if the user had to constantly refresh the site to check if new messages have come in.



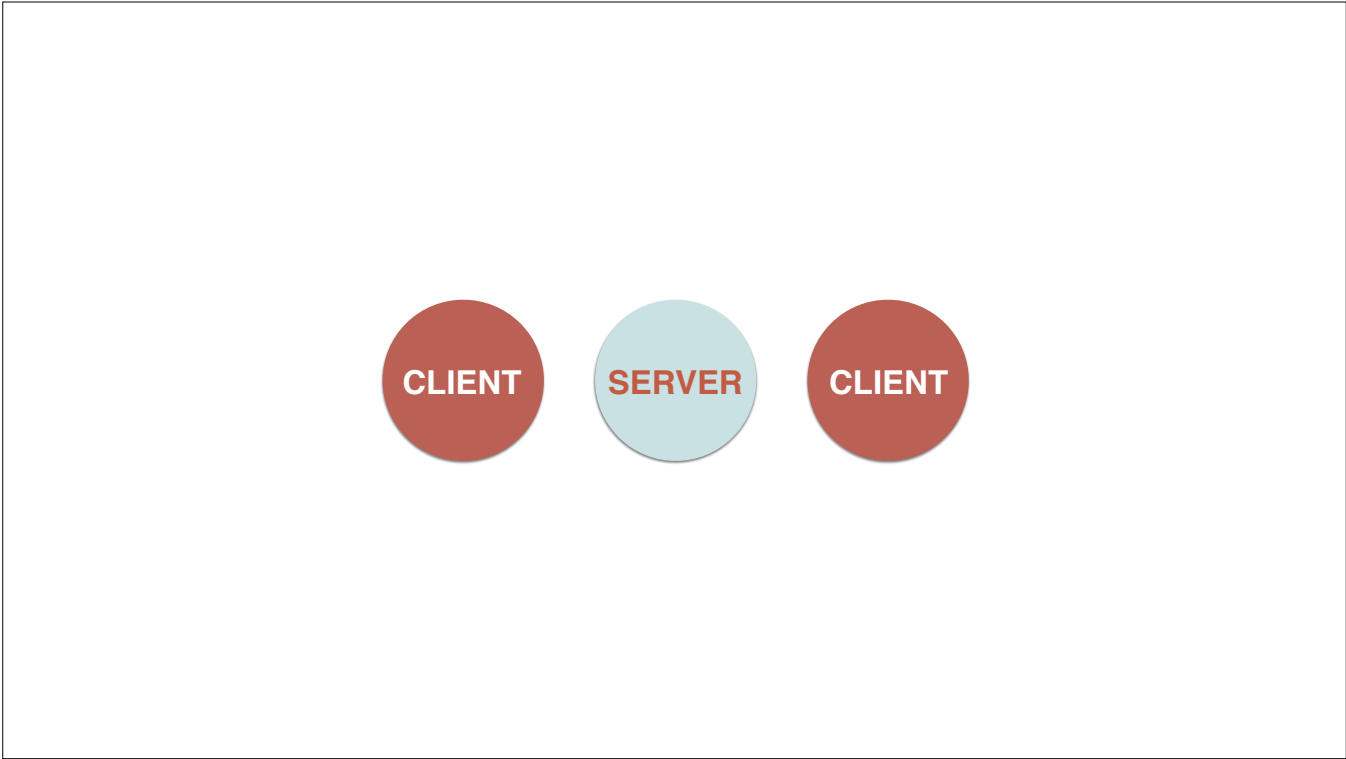
With the request-response cycle there is no other way of solving this problem except to ask the server if there is any new information every second. It is the internet equivalent of “Are we there yet?”.



Don't get me wrong. This solution will work... for a while.

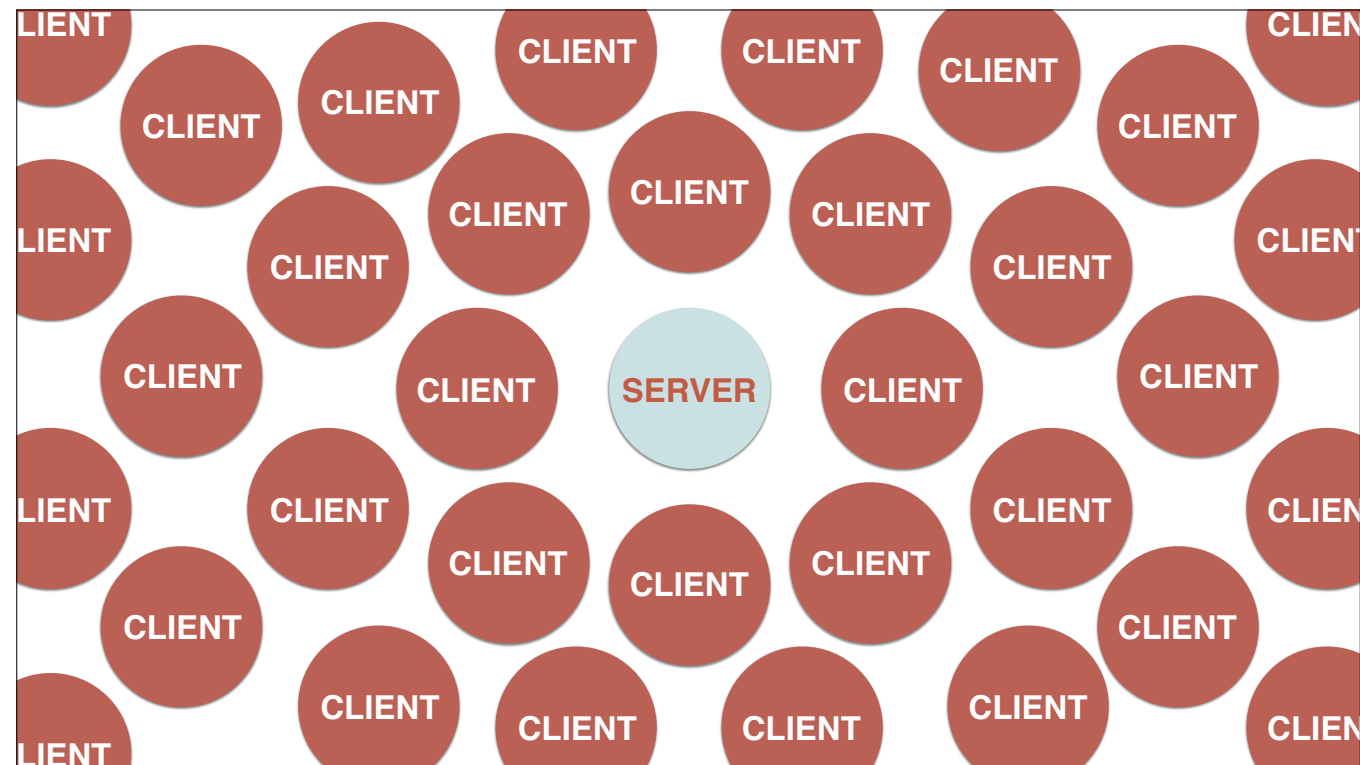
The shortcomings of this approach don't become apparent when there are only a few clients connected to the server.

Say... 1



Or 2...

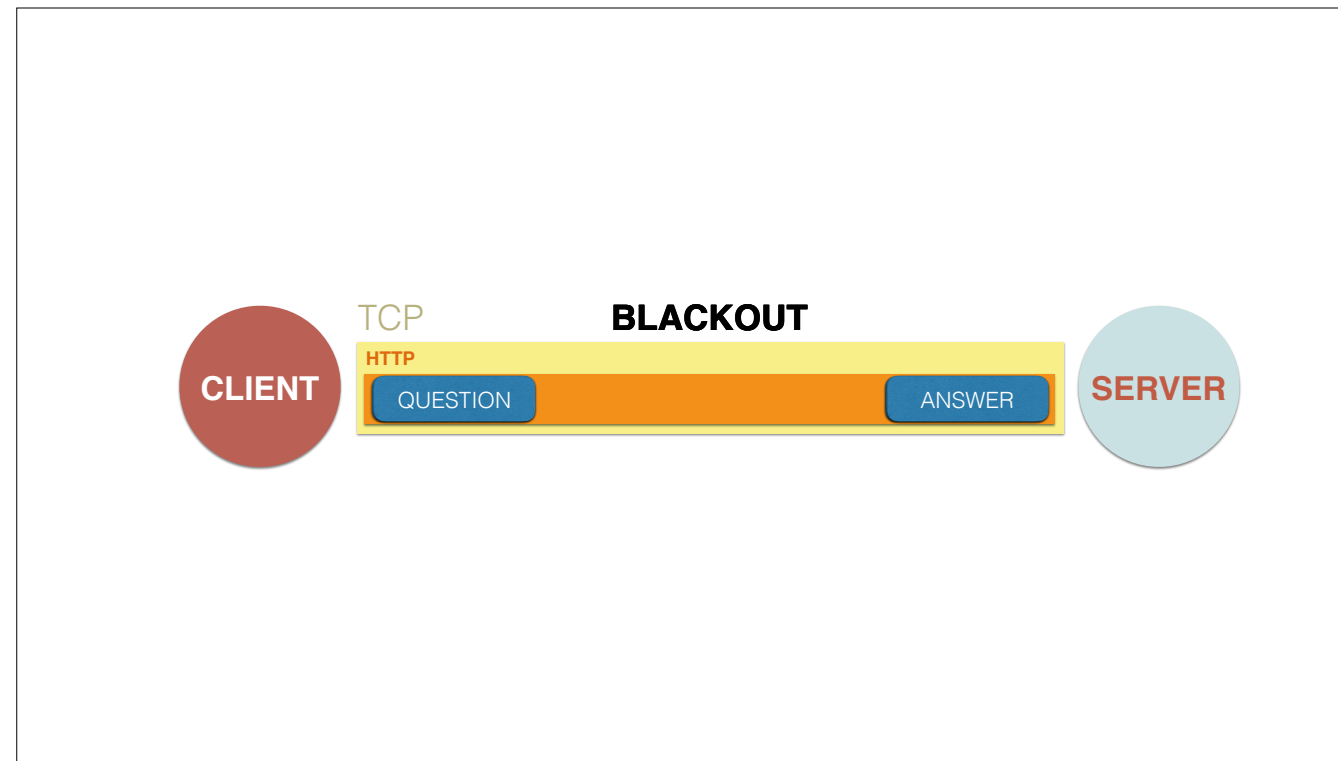




But it becomes apparent when there are many clients. The server will spend more time answering to requests for new data than processing actual user requests.

# Blackouts

Then there is the problem of blackouts.



A blackout is the time period between two server responses, when the client doesn't know if there is any new data.

We can make the problem of blackouts less apparent by shortening the interval between requests, but that increases the load on the server even more.

# Full-duplex

What we need is a full duplex connection. We need something like a telephone line between the server and the client to send data to each other, after a connection was established, without one explicitly asking for it.

# 02 WEBSOCKETS

WebSockets enable us to create full-duplex connections on the web.

# Realtime client-server communication

They enable real-time communication, without blackouts.

# Introduced in 2011

## RFC 6455

They were introduced 7 years ago (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.

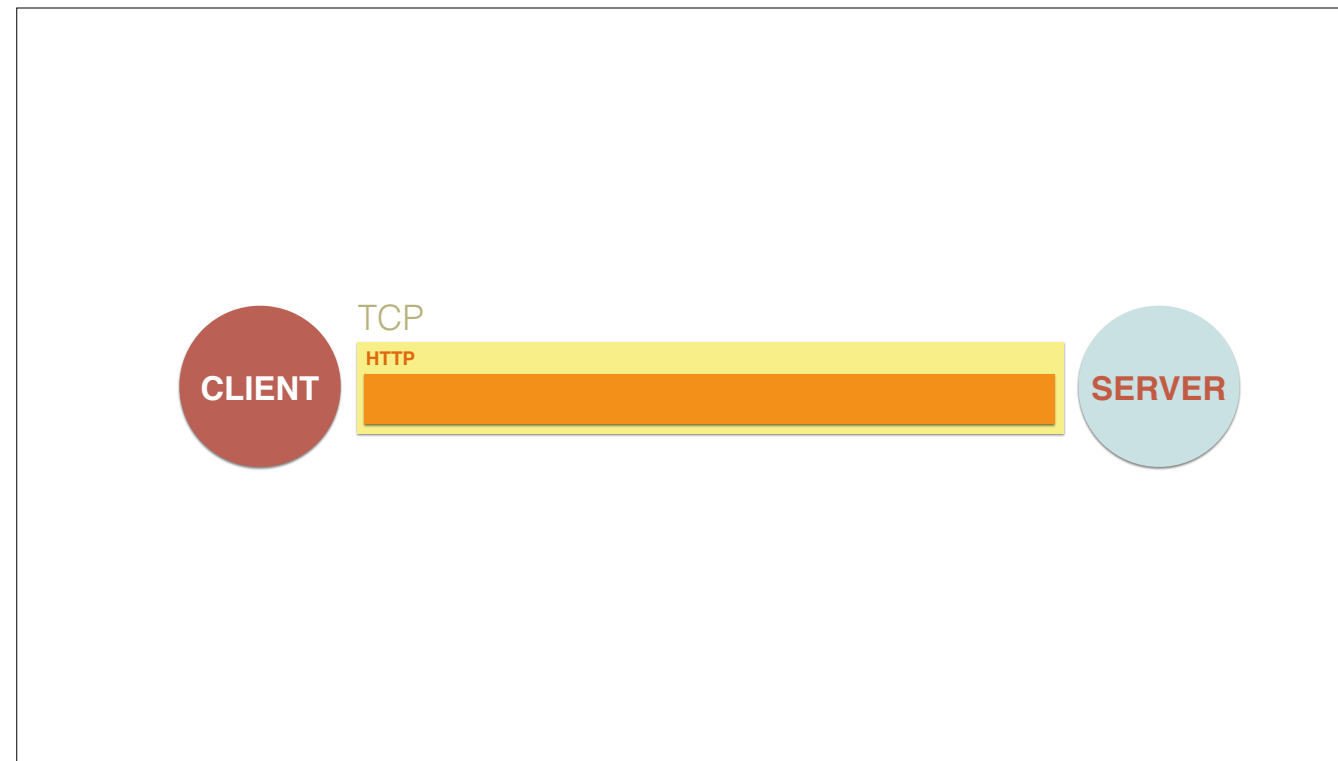
The protocol is...

The protocol itself...

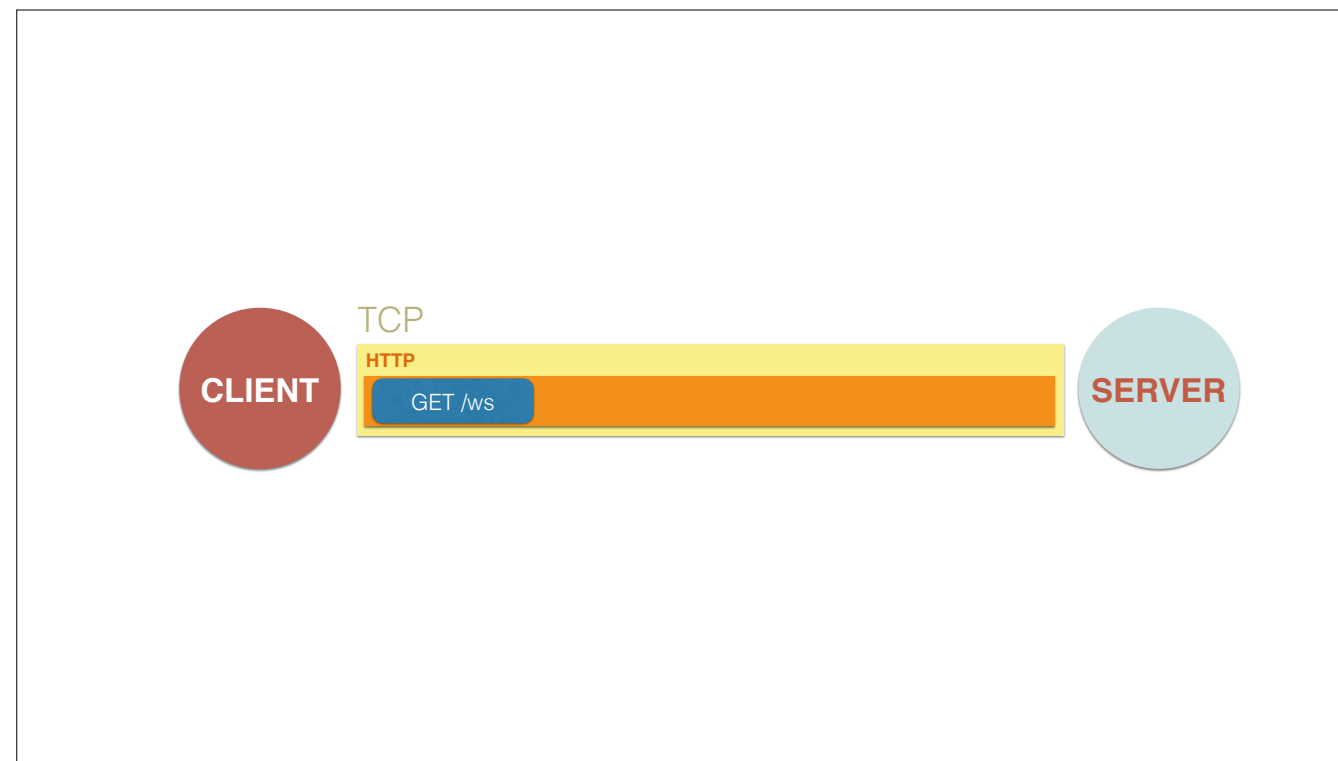


...interesting

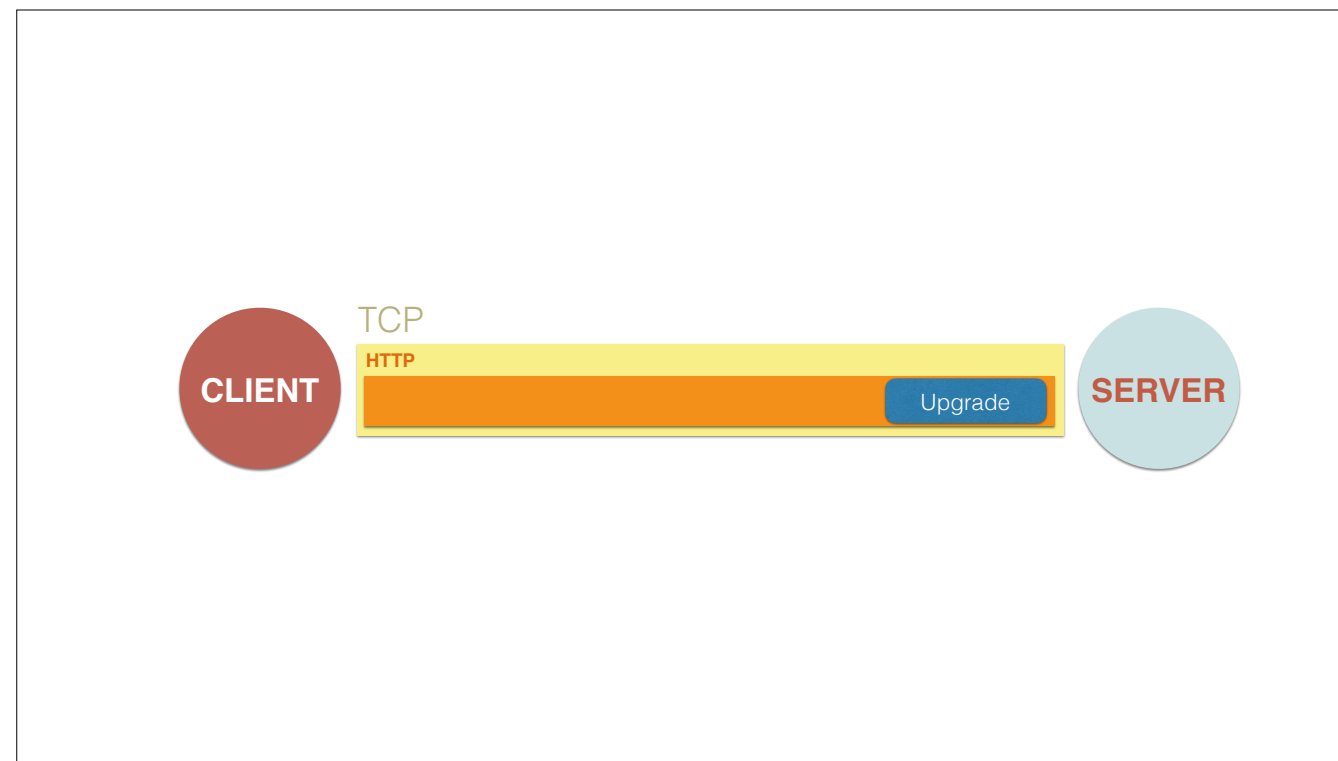
Has a few interesting quirks



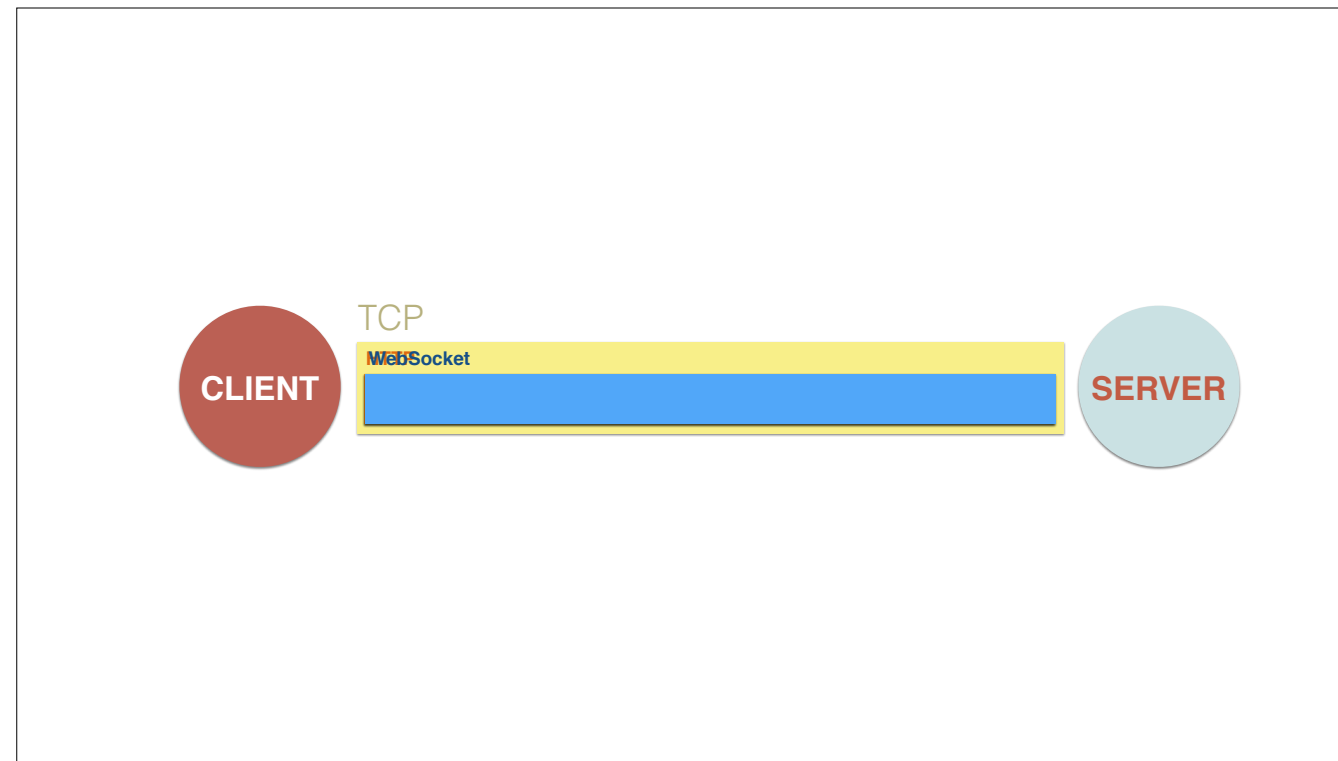
First a regular old HTTP/1.1 connection is established.



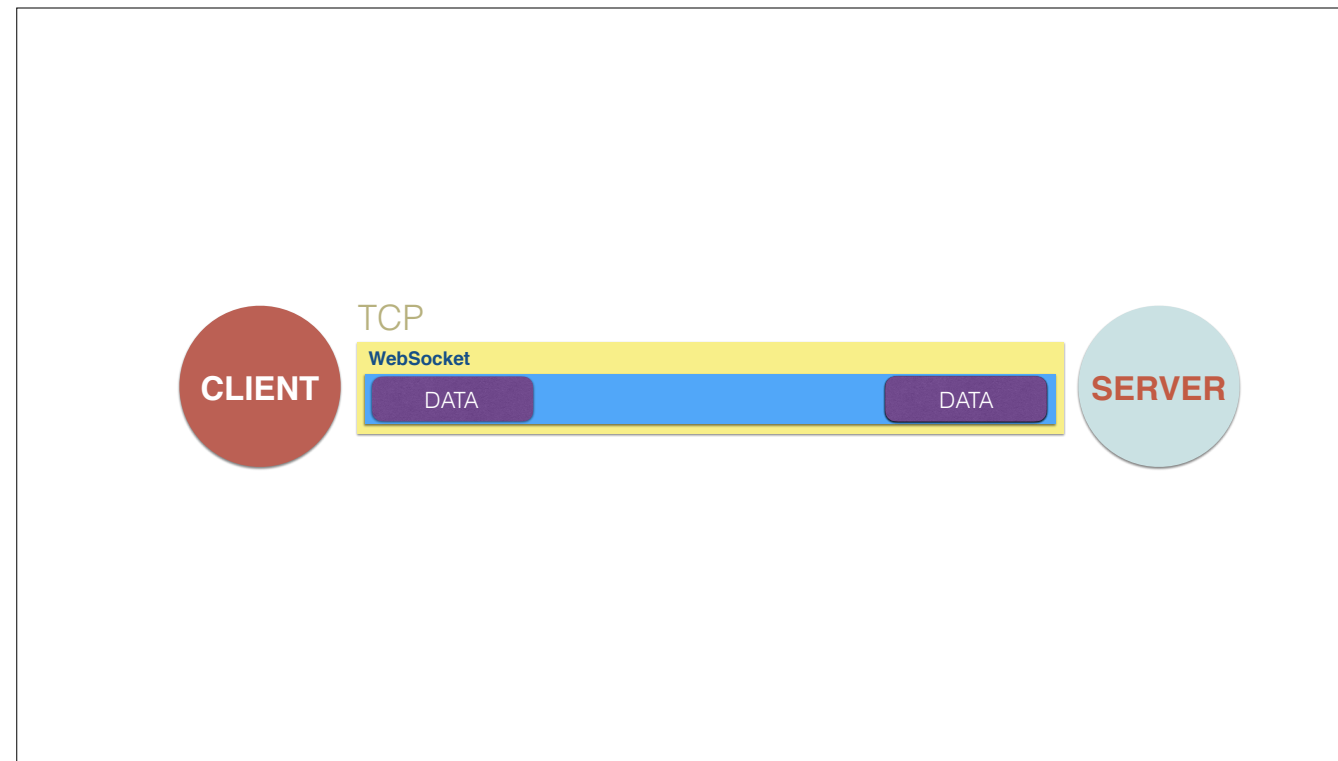
Then a GET request is made to the WebSocket endpoint of your application.



The server responds with an “upgrade request” which is part of HTTP/1.1



As part of the upgrade HTTP is abandoned and a new WebSocket protocol connection is established over the existing TCP connection.



After the upgrade is done, the server as well as the client can send data to each other at any time.

Every WebSocket starts out  
as a HTTP request

Every WebSocket starts out as a HTTP request, this small tidbit solves backwards compatibility issues with browsers that don't support WebSockets, as they can just fail with an unknown response. The other upside is authentication - you can authenticate the user in HTTP and pass the connection to WebSockets afterwards.



Part of HTML5 spec

When it comes to browser support. WebSockets are part of the HTML 5 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.js hosted with ❤ by GitHub

[view raw](#)

WebSocket JS API demo 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

You can have up to a 1024 simultaneous WebSocket connections in most browsers.

# Joined/Dropped client detection

They can immediately detect when a client connects or disconnects.

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

No reconnection handling

Though, there is no out-of-the-box reconnection handling.

# Non standard proxying

And you will need to adjust your firewall, proxy and load balancer as WebSockets aren't HTTP and most proxies will require additional configuration for them to work.

Can we solve our chat  
problem with them?

Now with this full-duplex connection that are WebSockets can we solve our chat problem?

YES!

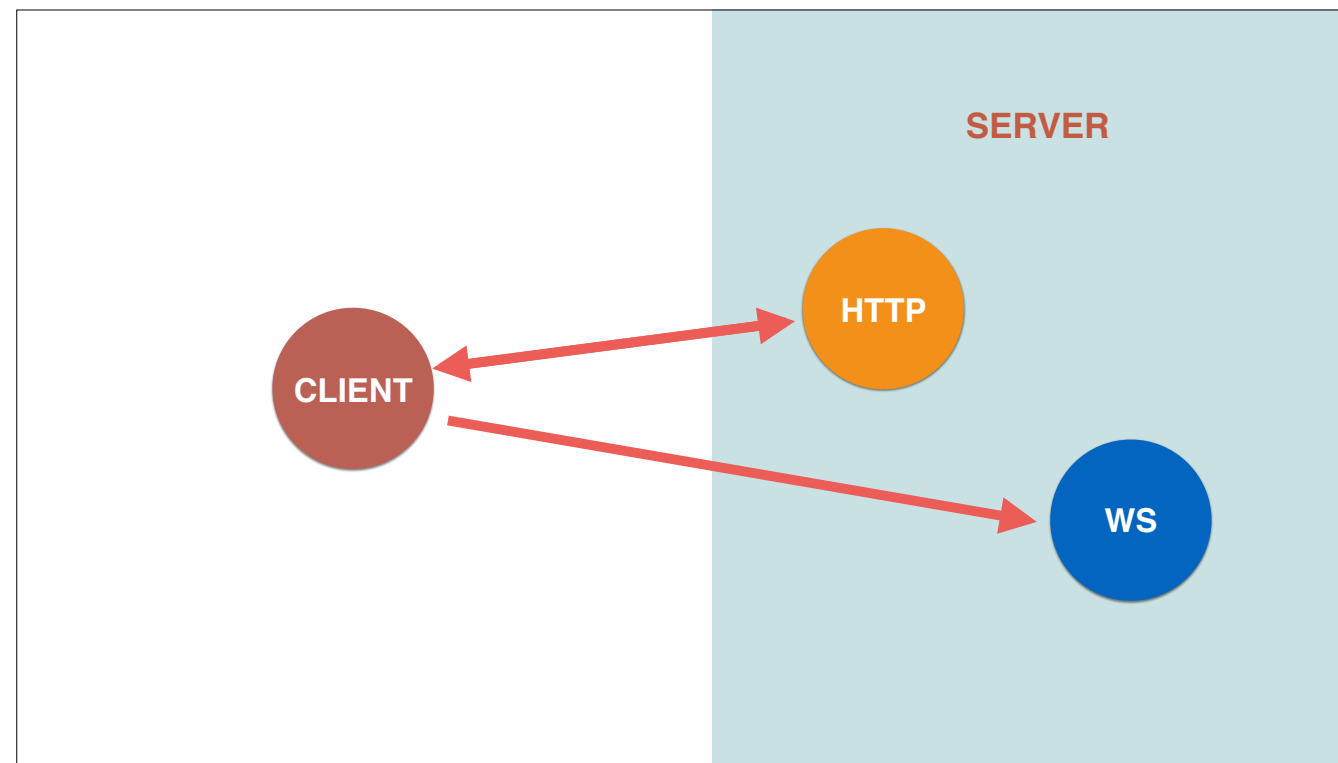
We can!



But...

But what new problems did we cause?





We separated control flow from our web server and added another WebSocket control flow along side it.

This introduces more complexity into the system, as you now have two presentation layers to your application.

# Overkill

All this just to push data from the server to the client...

WebSockets are an overkill solution to this problem.

We created a whole different protocol which enables us to push data from the server to the client and vice versa.

We added an alternate control flow to our application.

We introduced a whole new sub-system to our application!

Voice chat  
Video chat  
Games

The perfect application for WebSockets are voice and video chat applications, or games

# Real-time **client to server** communication

Or, to generalise it even further.

They are best suited for situations where the CLIENT has to push data to the SERVER.

Not the opposite.



# Alternatives?

Ok. So what alternatives do we have?

# 03 SERVER SENT EVENTS

The first alternative I'm going to talk about are Server Sent Events.  
More widely known as EventSource to frontend developers.

# Introduced in 2006

## WHATWG Web Applications 1.0

They were introduced 5 years before WebSockets.

And are part of the HTTP living standard, meaning they have been supported in browsers for a long time.

# HTTP based

## Rely on HTTP/1.1 streaming

It's completely HTTP based.

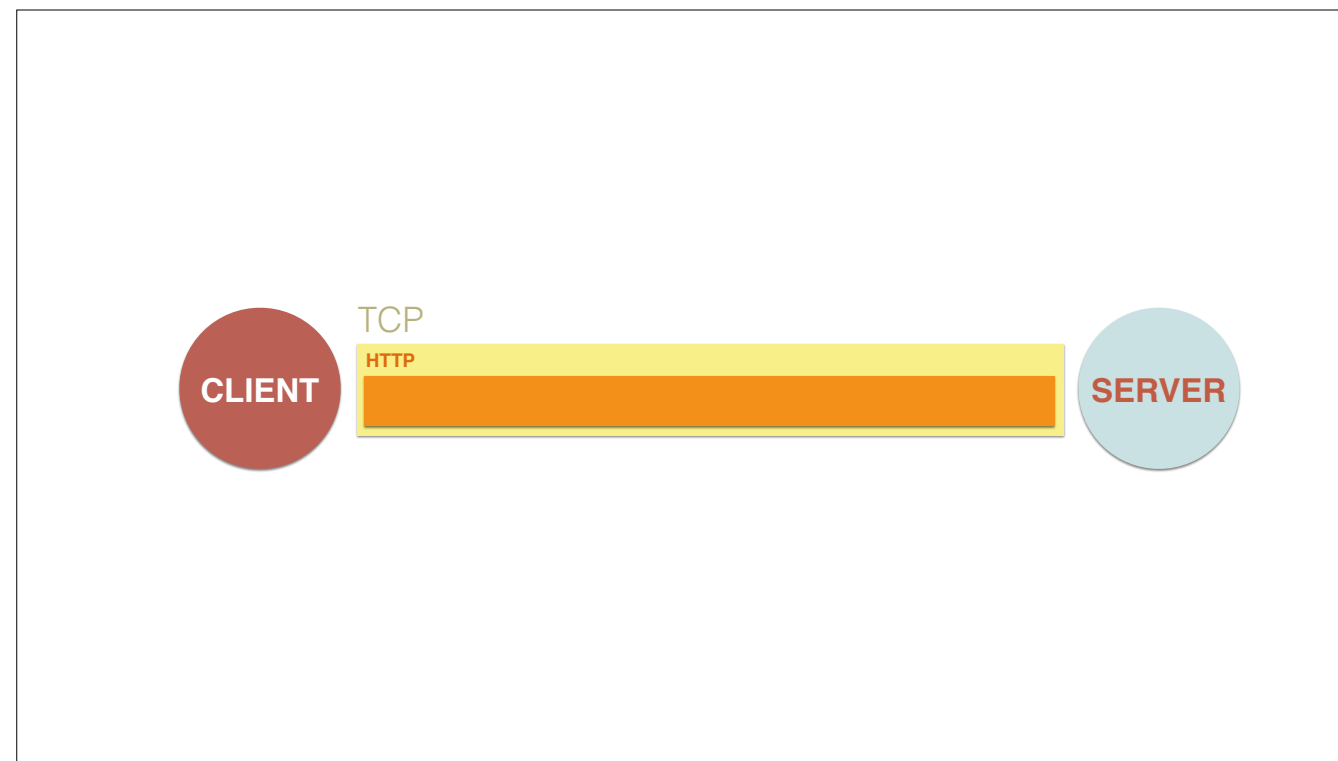
This is good for all the same reasons as WebSockets, but they don't switch protocols at any time.

This has the added benefit that they can be poly filled in all browsers that support HTTP/1.1

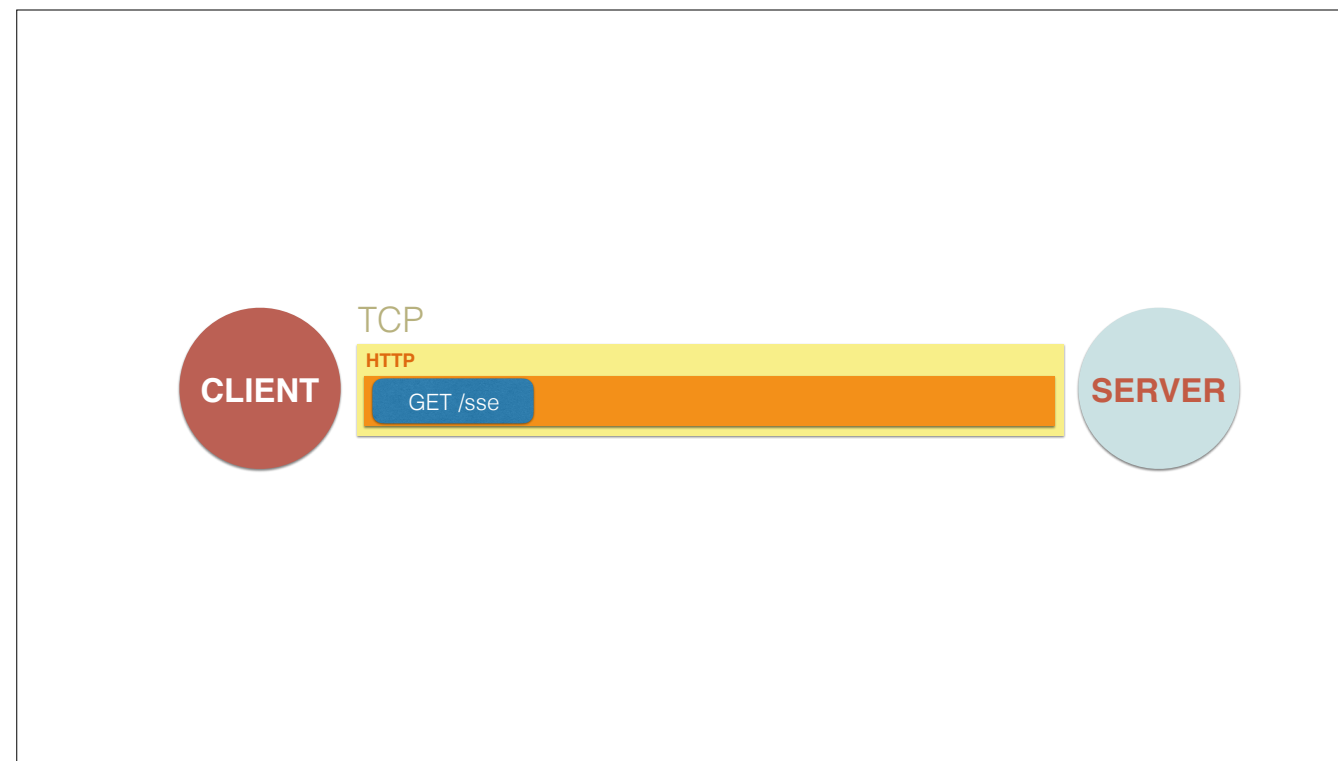


Only Server to Client  
communication

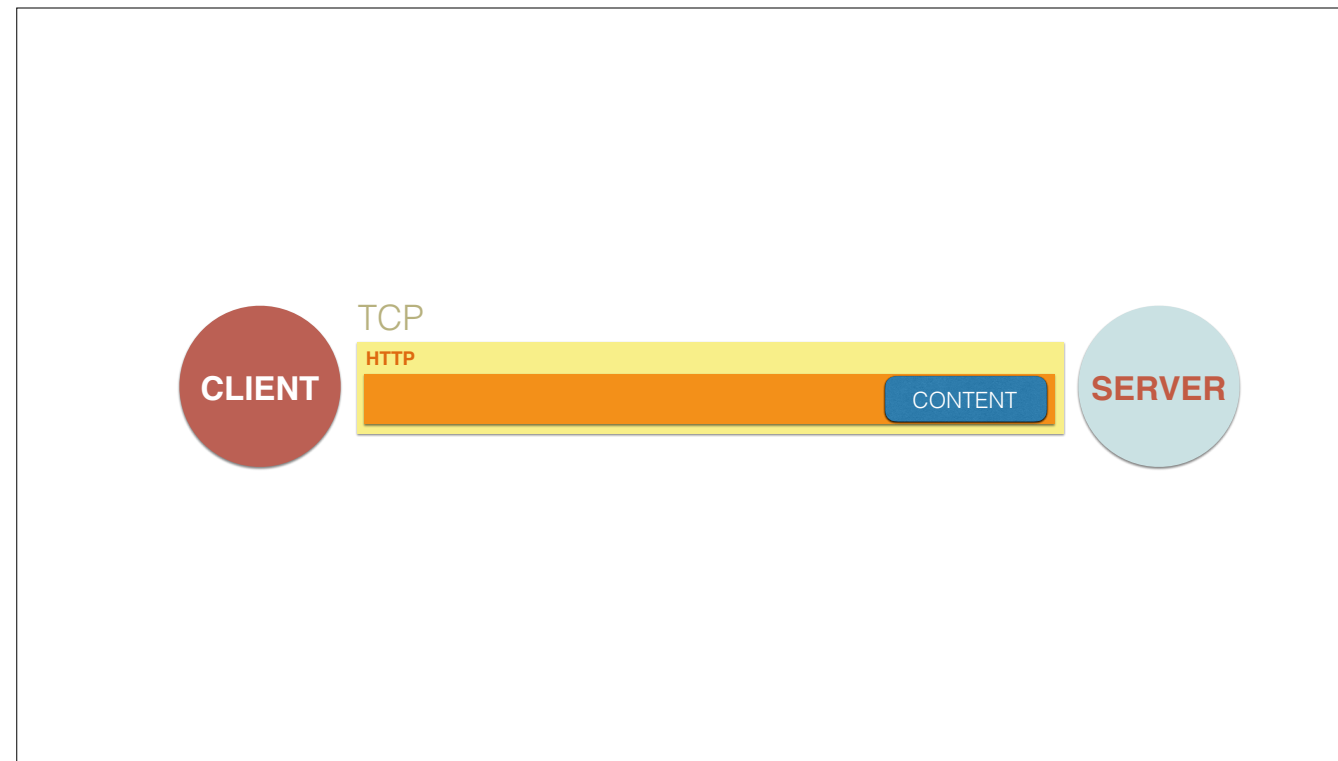
But they only provide a half-duplex connection.  
Only the server can push data to the client.



First a regular old HTTP/1.1 connection is established.



Then a GET request is made to the server sent events endpoint of your application.



The server responds with a regular response with the type octet-stream which is part of HTTP/1.1

The connection is kept alive, again a part of HTTP/1.1 and the server just sends data to the client whenever data is ready.

A solid blue rectangular box with a thin black border, centered on the page. It contains the text "Not supported on IE / Edge" in white, sans-serif font.

Not supported on IE / Edge

Sadly, SSE is not supported on Microsoft platforms.  
But, again, a polyfill for them is available.

```
1  var evtSource = new EventSource('/stream');
2
3  evtSource.onmessage = function(e) {
4    console.log('DATA', e.data);
5  }
```

event-source.js hosted with ❤ by GitHub

[view raw](#)

But in all other browsers there is this API available

Subject to the ~6  
connection limit

If you didn't know. Most modern browsers limit the number of connections 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.



# No dropped client detection

But SSE can't detect if a client has stopped listening to the server.

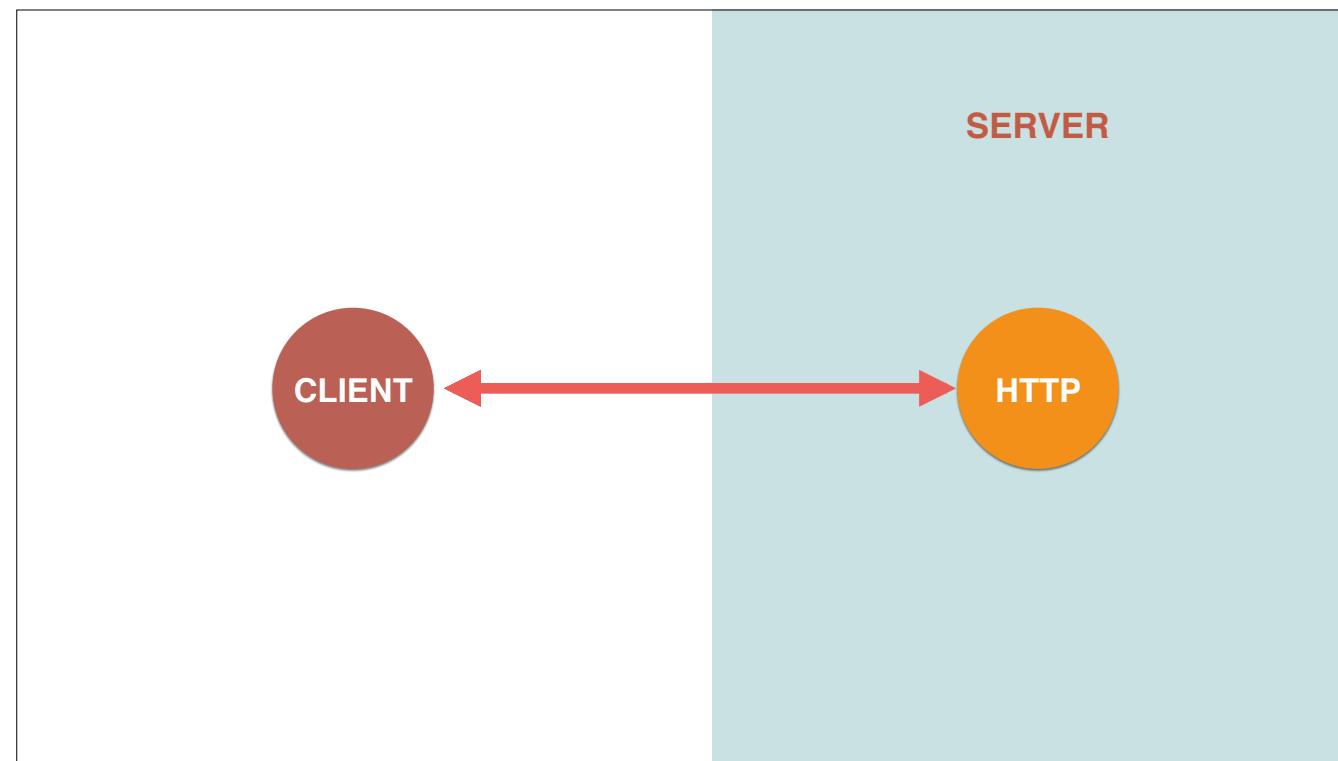
This can be solved by keeping a heartbeat.

Please note that all WebSocket libraries implement a heart beat, for practical reasons.



# Benefits?

Though an objectively inferior technology SSE provide a few benefits.



We now have just one control flow in our application.

```

31     r.get 'stream' do
32       response['Content-Type'] = 'text/event-stream;charset=UTF-8'
33       q = Queue.new
34       QUEUES << q
35       q << { heartbeat: true }
36       stream(loop: true, callback: proc { QUEUES.delete(q) }) do |out|
37         loop do
38           out << "data: #{q.pop.to_json}\n\n"
39         end
40       end
41     end
42   end
43 end

```

sse\_chat.rb hosted with ❤ by GitHub

[view raw](#)

Chat app example from — [https://github.com/stankek/lectures/tree/master/19-rabbitmq\\_is\\_more\\_than\\_a\\_sidekiq\\_replacement/demo/01-simple\\_job\\_queue](https://github.com/stankek/lectures/tree/master/19-rabbitmq_is_more_than_a_sidekiq_replacement/demo/01-simple_job_queue)

In fact the SSE endpoint can be just another controller in your application.

This example is in Ruby, mostly because it looks like pseudo-code but the principle is the same in all languages and implementations.

```
31     r.get 'stream' do
32       response['Content-Type'] = 'text/event-stream;charset=UTF-8'
33       q = Queue.new
34       QUEUES << q
35       q << { heartbeat: true }
36       stream(loop: true, callback: proc { QUEUES.delete(q) }) do |out|
37         loop do
38           out << "data: #{q.pop.to_json}\n\n"
39         end
40       end
41     end
42   end
43 end
```

sse\_chat.rb hosted with ❤ by GitHub

[view raw](#)

Chat app example from — [https://github.com/stankek/lectures/tree/master/19-rabbitmq\\_is\\_more\\_than\\_a\\_sidekiq\\_replacement/demo/01-simple\\_job\\_queue](https://github.com/stankek/lectures/tree/master/19-rabbitmq_is_more_than_a_sidekiq_replacement/demo/01-simple_job_queue)

Here we create a “stream” endpoint that response to GET

```

31     r.get 'stream' do
32       response['Content-Type'] = 'text/event-stream;charset=UTF-8'
33       q = Queue.new
34       QUEUES << q
35       q << { heartbeat: true }
36       stream(loop: true, callback: proc { QUEUES.delete(q) }) do |out|
37         loop do
38           out << "data: #{q.pop.to_json}\n\n"
39         end
40       end
41     end
42   end
43 end

```

sse\_chat.rb hosted with ❤ by GitHub

[view raw](#)

Chat app example from — [https://github.com/stankek/lectures/tree/master/19-rabbitmq\\_is\\_more\\_than\\_a\\_sidekiq\\_replacement/demo/01-simple\\_job\\_queue](https://github.com/stankek/lectures/tree/master/19-rabbitmq_is_more_than_a_sidekiq_replacement/demo/01-simple_job_queue)

We set the response type to event-stream or octet-stream, both will work.

```

31     r.get 'stream' do
32       response['Content-Type'] = 'text/event-stream;charset=UTF-8'
33       q = Queue.new
34       QUEUES << q
35       q << { heartbeat: true }
36       stream(loop: true, callback: proc { QUEUES.delete(q) }) do |out|
37         loop do
38           out << "data: #{q.pop.to_json}\n\n"
39         end
40       end
41     end
42   end
43 end

```

sse\_chat.rb hosted with ❤ by GitHub

[view raw](#)

Chat app example from — [https://github.com/stankek/lectures/tree/master/19-rabbitmq\\_is\\_more\\_than\\_a\\_sidekiq\\_replacement/demo/01-simple\\_job\\_queue](https://github.com/stankek/lectures/tree/master/19-rabbitmq_is_more_than_a_sidekiq_replacement/demo/01-simple_job_queue)

Finally we just loop and send data as it comes in.  
In the above example “pop” is a blocking action.

```

31     r.get 'stream' do
32       response['Content-Type'] = 'text/event-stream;charset=UTF-8'
33       q = Queue.new
34       QUEUES << q
35       q << { heartbeat: true }
36       stream(loop: true, callback: proc { QUEUES.delete(q) }) do |out|
37         loop do
38           out << "data: #{q.pop.to_json}\n\n"
39         end
40       end
41     end
42   end
43 end

```

sse\_chat.rb hosted with ❤ by GitHub

[view raw](#)

Chat app example from — [https://github.com/stankek/lectures/tree/master/19-rabbitmq\\_is\\_more\\_than\\_a\\_sidekiq\\_replacement/demo/01-simple\\_job\\_queue](https://github.com/stankek/lectures/tree/master/19-rabbitmq_is_more_than_a_sidekiq_replacement/demo/01-simple_job_queue)

Another apparent benefit is that the protocol is simple.

There are a total of 4 keywords followed by a colon “:” and the data.

And the whole system works with only one keyword - “data” which is used to send data.

The remaining 3 keywords are just metadata: “id”, “event” and “retry”

Which set the id and the tag of the data, and the reconnection interval.



# Text chat Notifications

The best use case for Server Sent Events are notifications and chat applications.

# Real-time **server to client** communication

Or in a broader sense, anything that requires the server to push data to the client.

# 04 LONG POLLING

There is another alternative even to server sent events, and that is long polling.

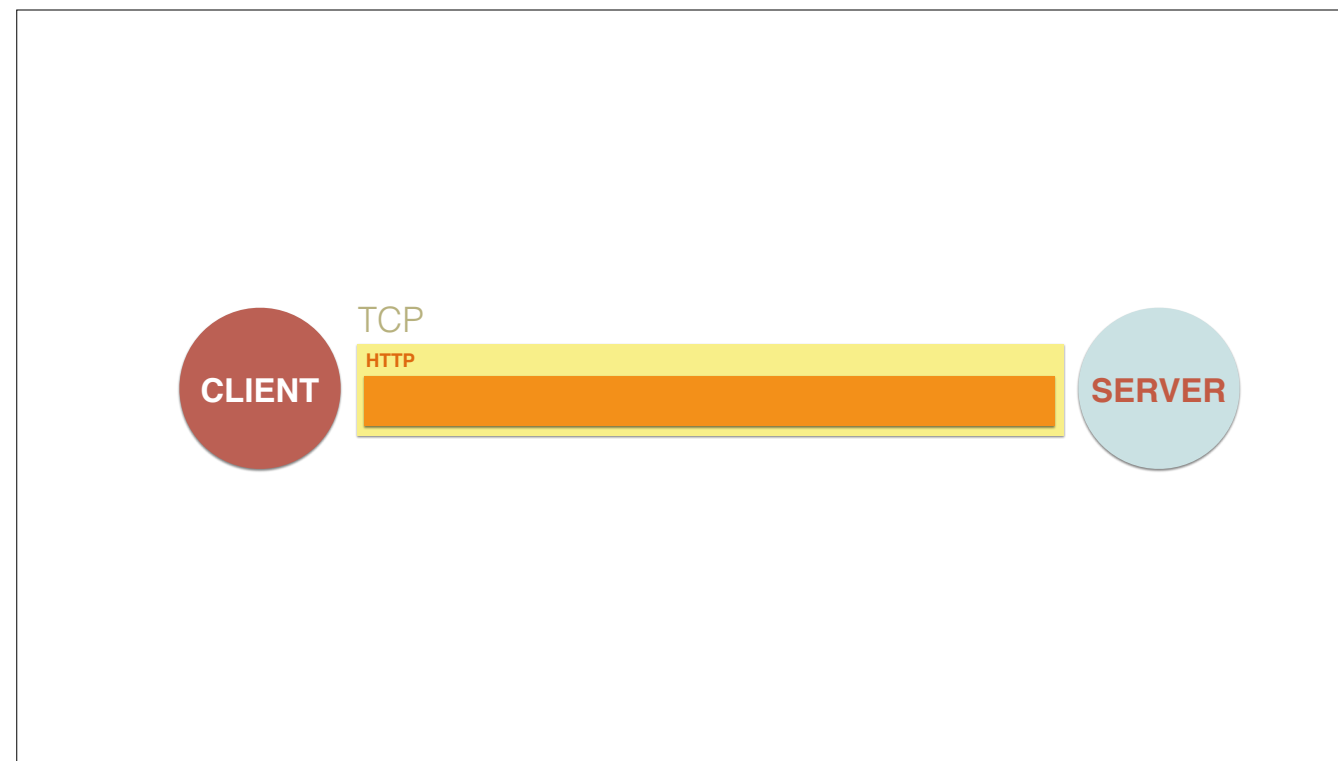
# Pure HTTP

Long polling is pure HTTP.

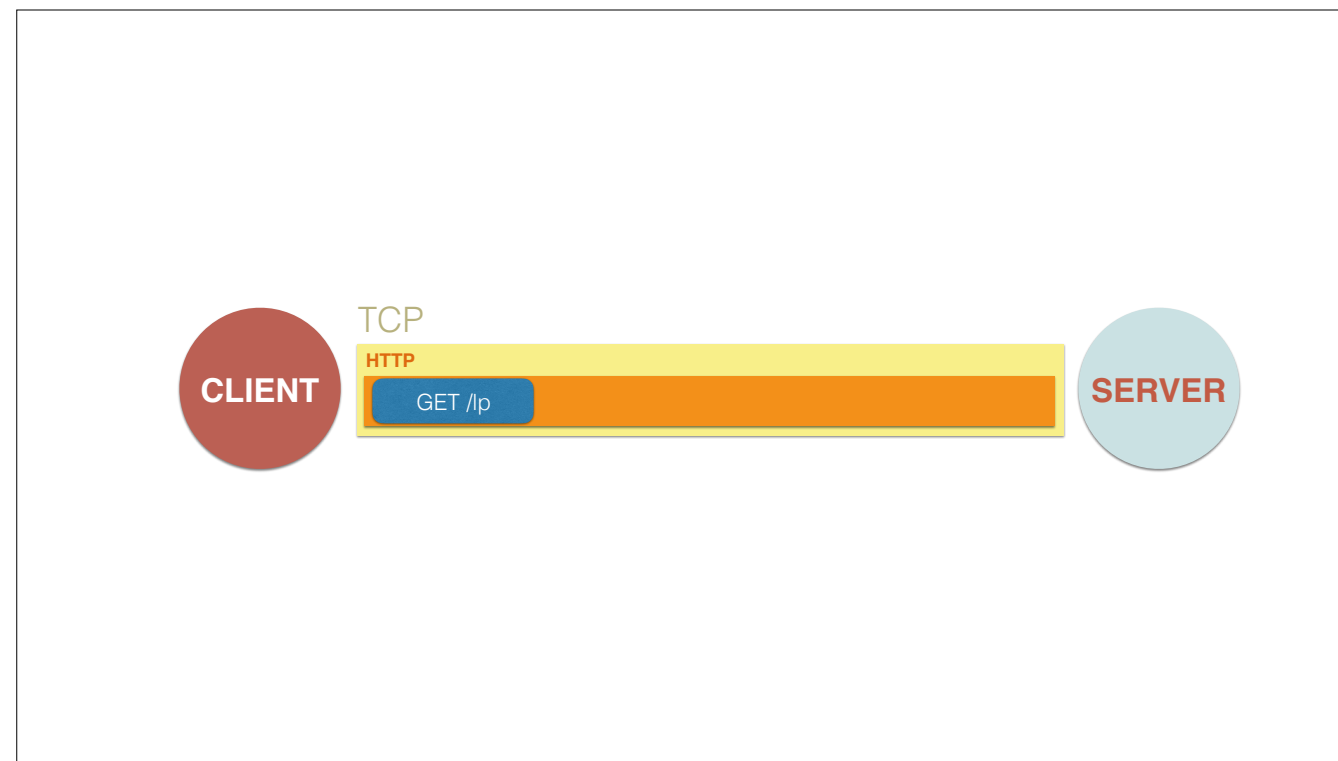
No fancy streaming or upgrades or keep alives.

The server delays the  
response

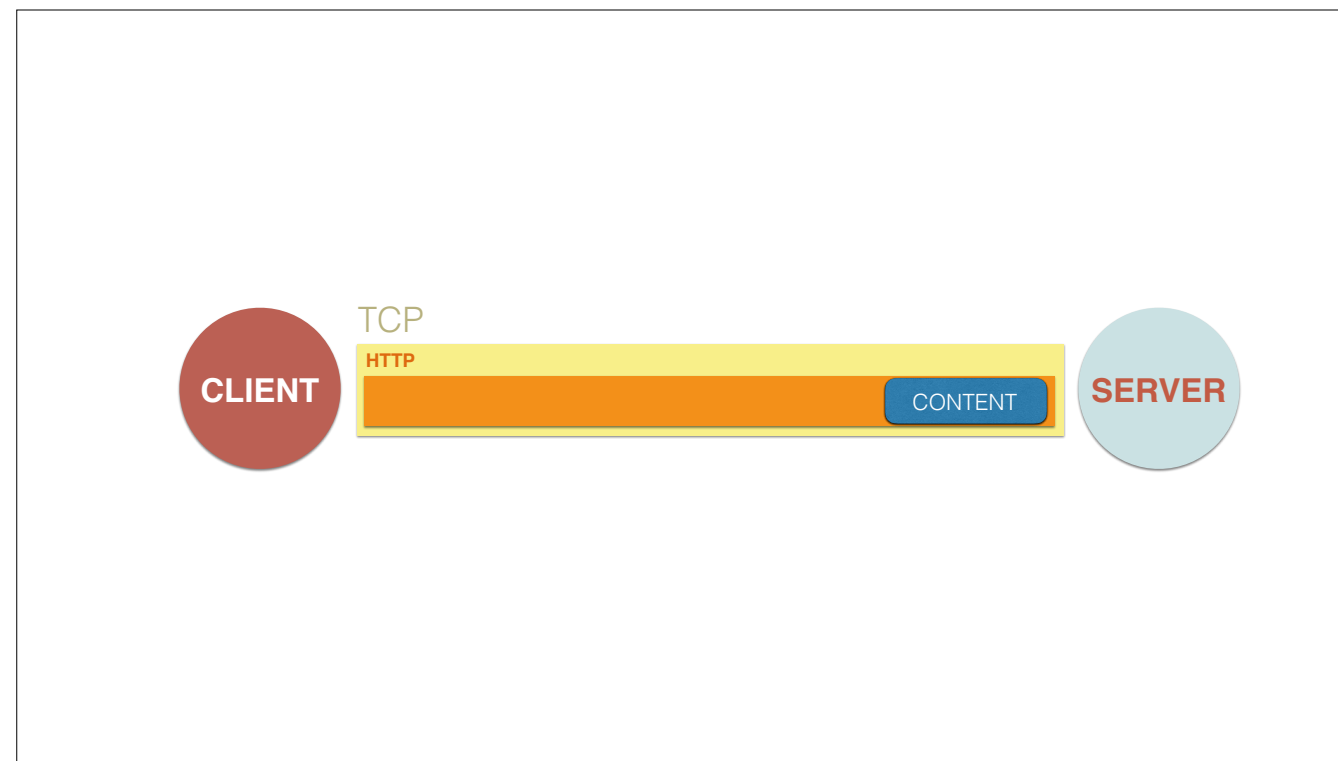
The whole idea behind long polling is that the server just delays it's response



First a regular old HTTP connection is established.



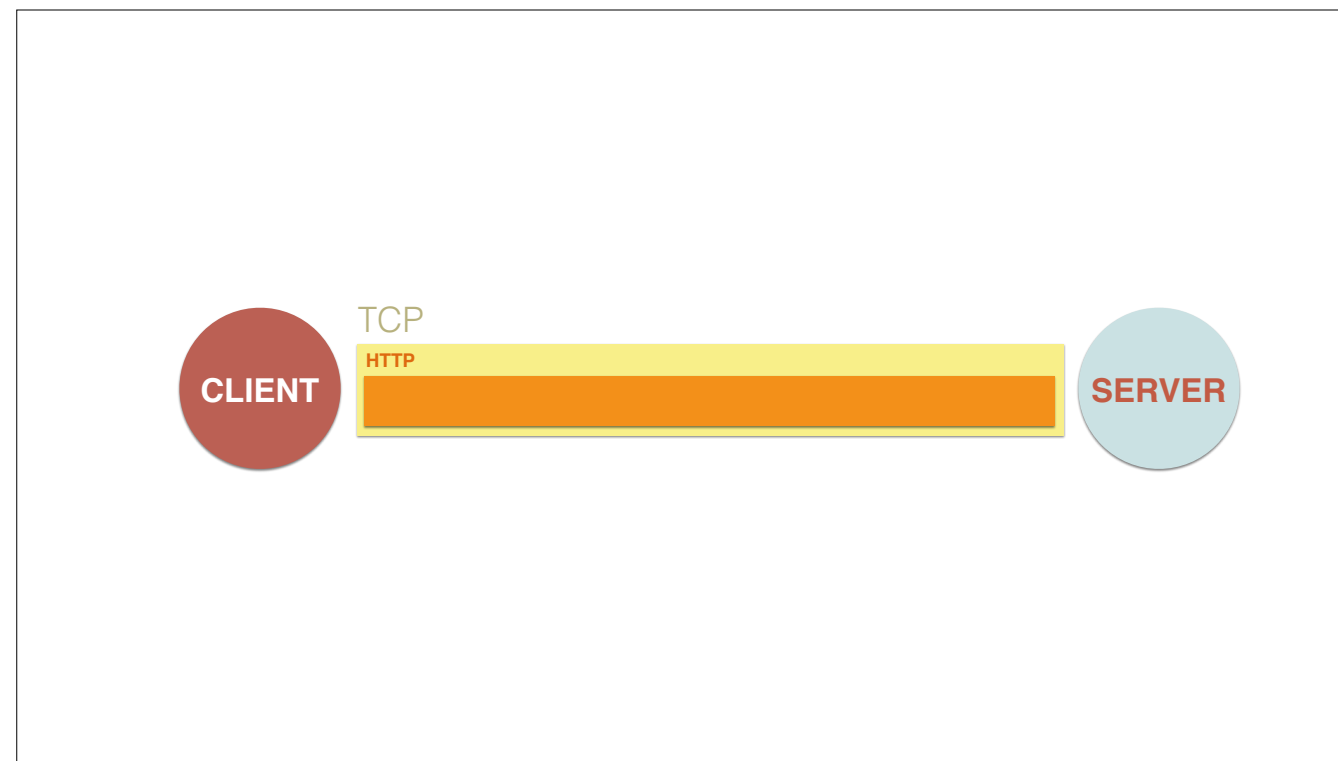
Then a GET request is made to the long polling endpoint of your application.



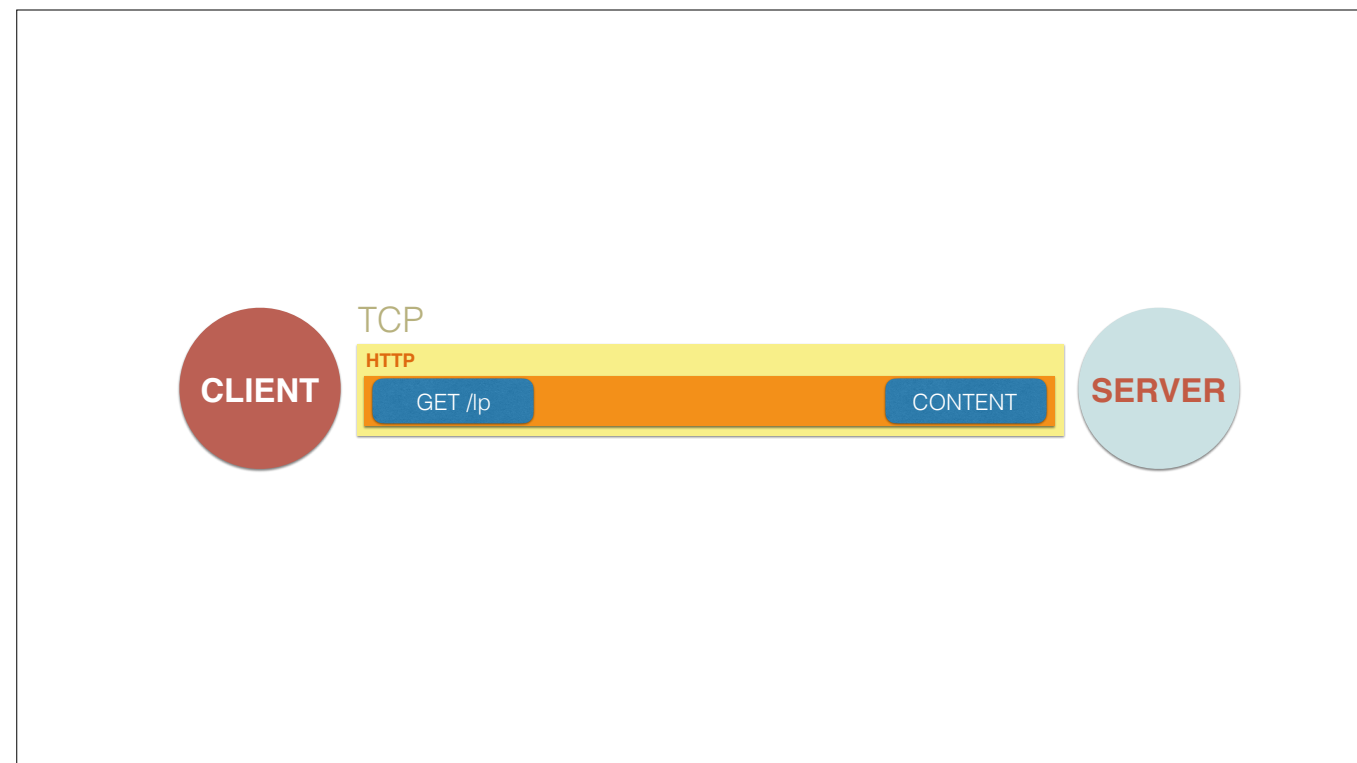
The server responds with a regular response.

The server just sends data to the client whenever data is ready.

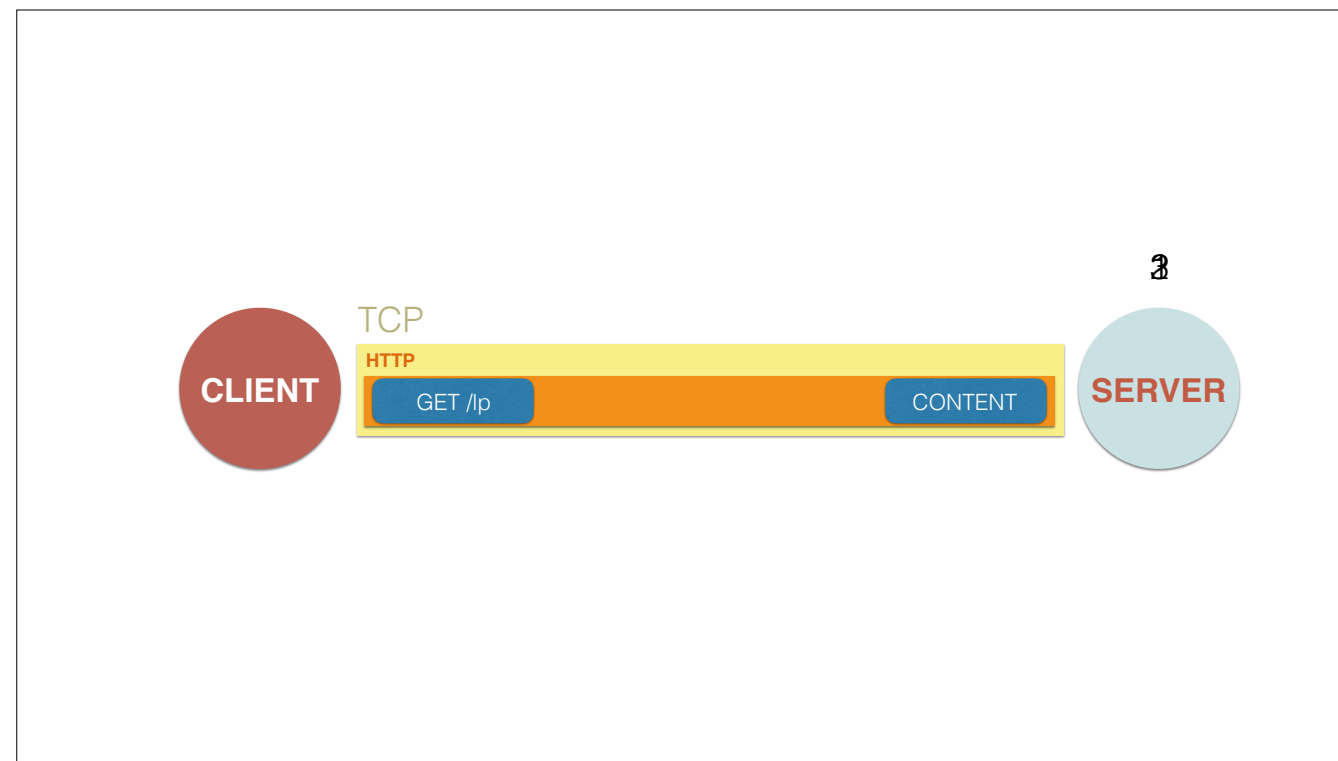




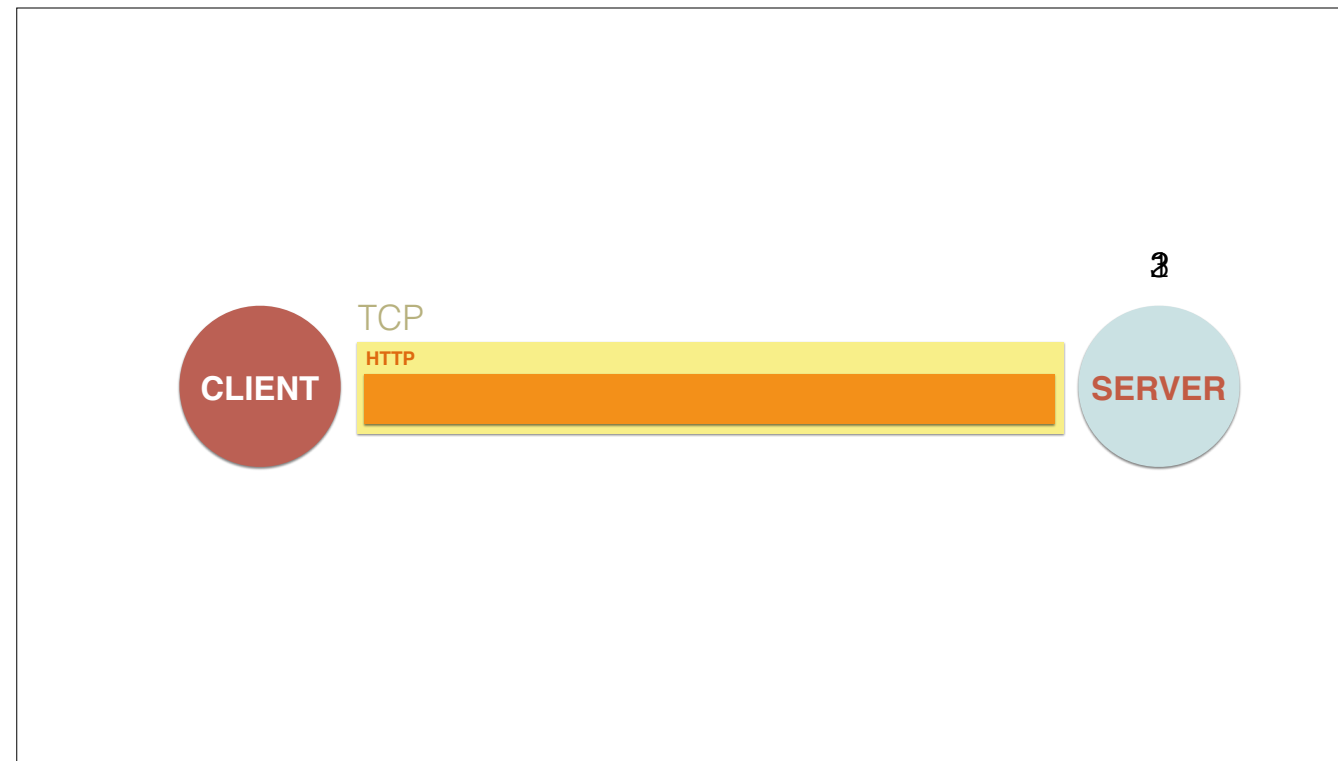
And the connection is torn down



And then the cycle begins again.



If the server doesn't have any new data. It delays the response until new data is available.



It's also possible that the server delays for too long and some intermediary, be it the proxy, the load balancer, or the browser itself kills the connection. That's completely fine, in fact it's expected. The client will start the cycle again.



No native APIs

You can notice that this is just an extension of our original idea.

Therefore there are no native APIs for aout-of-the-box long polling, you just ask the server for data as normal.

# Blackouts

The point of long polling is to make blackouts much shorter and lessen the load on the server.

Subject to the ~6  
connection limit

They are subject to the 6 connection limit.



# Supported everywhere

But they are supported literally everywhere.

There is no device or browser that can't do long polling.





Same controls flow

Again, you don't need a separate control flow.

You can just use your regular controllers to implement long polling.



Better for large numbers  
of users

They are arguable better for large numbers of users.

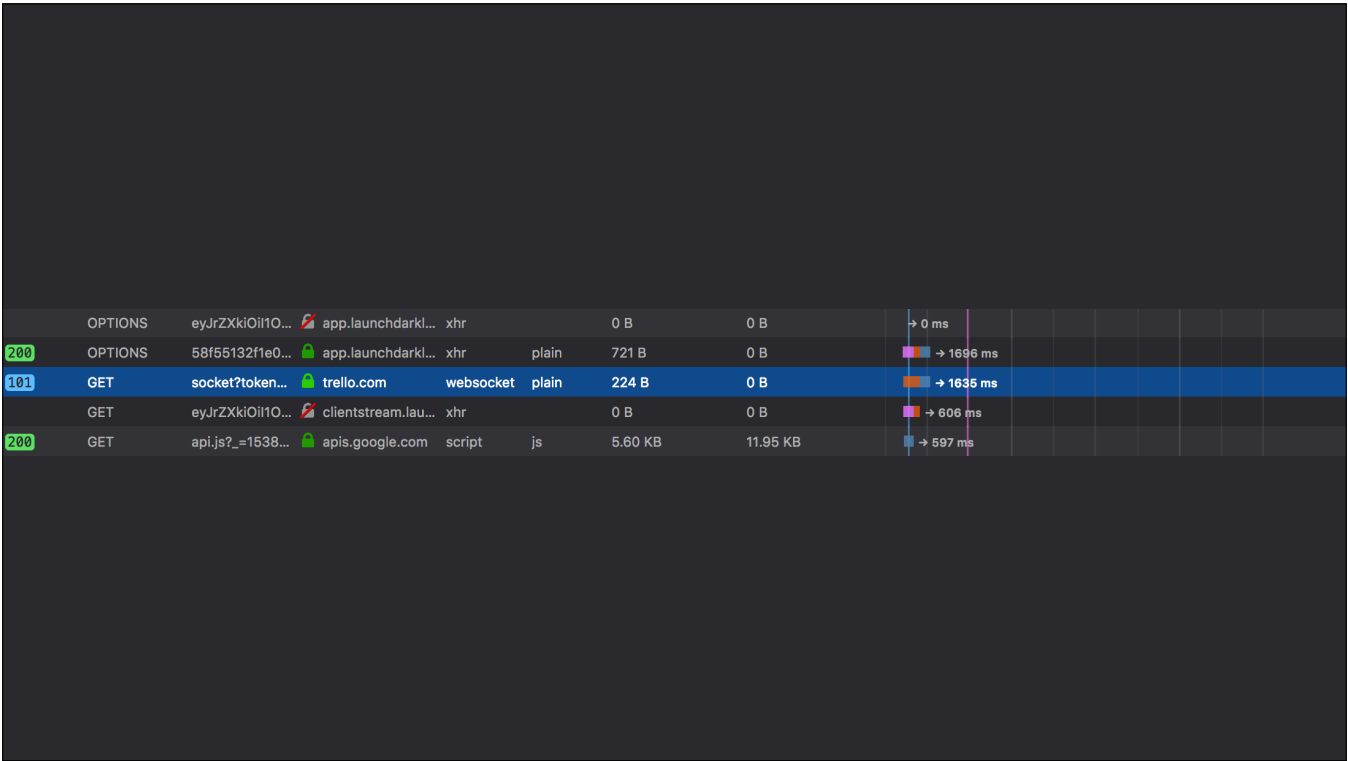
I don't have experience this, but the reasoning is that since LP periodically releases connections to the server it's harder to starve the server of resources.

This is partially supported by the fact that Facebook's Messenger and FreeNode uses long polling for their chat applications.

# 05

## CONCLUSION

So to conclude.



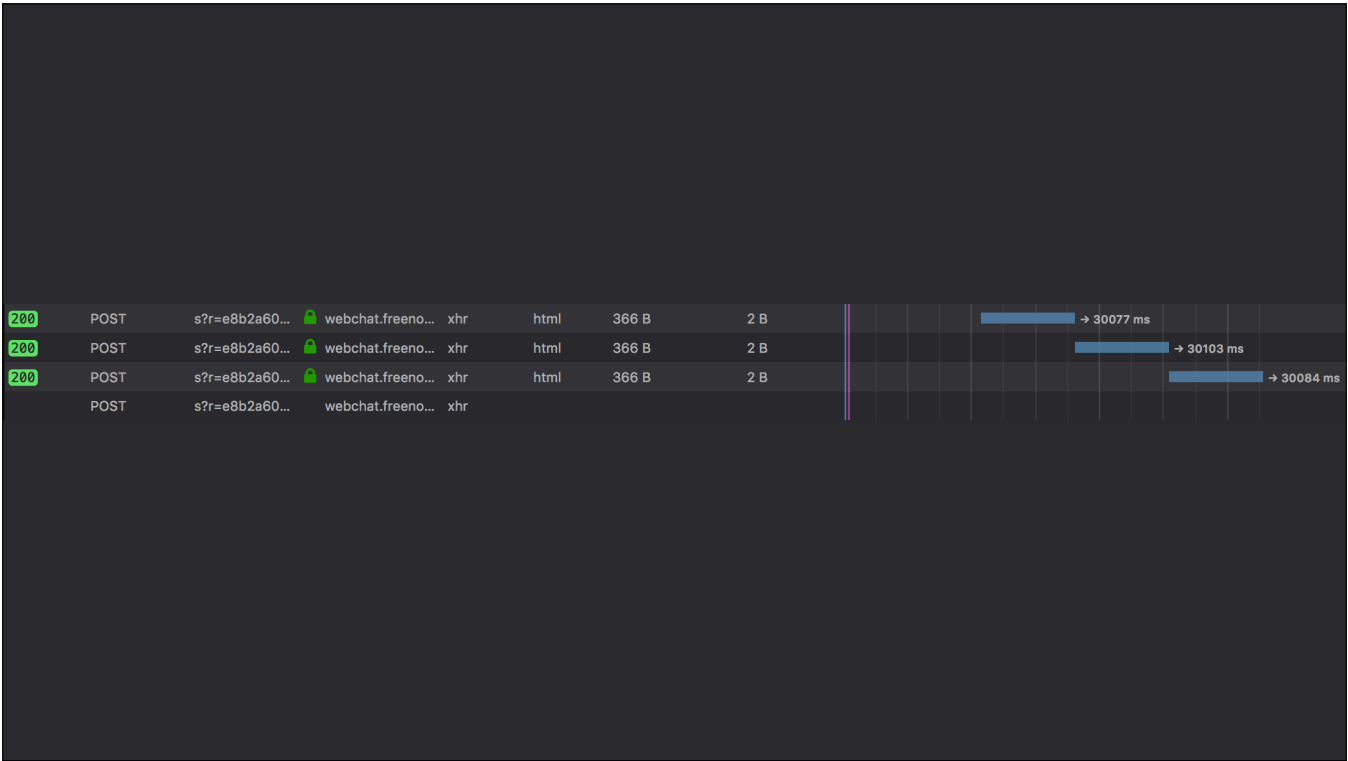
|     |         |                  |                     |           |       |         |          |           |  |  |  |  |  |  |  |  |  |
|-----|---------|------------------|---------------------|-----------|-------|---------|----------|-----------|--|--|--|--|--|--|--|--|--|
|     | OPTIONS | eyJrZXkiOiI1O... | app.launchdarkl...  | xhr       | 0 B   | 0 B     | → 0 ms   |           |  |  |  |  |  |  |  |  |  |
| 200 | OPTIONS | 58f55132f1e0...  | app.launchdarkl...  | xhr       | plain | 721 B   | 0 B      | → 1696 ms |  |  |  |  |  |  |  |  |  |
| 101 | GET     | socket?token...  | trello.com          | websocket | plain | 224 B   | 0 B      | → 1635 ms |  |  |  |  |  |  |  |  |  |
|     | GET     | eyJrZXkiOiI1O... | clientstream.lau... | xhr       | 0 B   | 0 B     | → 606 ms |           |  |  |  |  |  |  |  |  |  |
| 200 | GET     | api.js?_=1538... | apis.google.com     | script    | js    | 5.60 KB | 11.95 KB | → 597 ms  |  |  |  |  |  |  |  |  |  |

If you want to know how somebody implemented a real-time component just fire up your network inspector.

If you see a 101 it's most likely a web socket connection. For instance, Trello uses WebSockets.

|     |      |                 |                 |          |      |         |          |         |  |  |  |  |  |  |  |  |        |
|-----|------|-----------------|-----------------|----------|------|---------|----------|---------|--|--|--|--|--|--|--|--|--------|
| 200 | GET  | /               | localhost:9292  | document | html | 2.18 KB | 2.12 KB  | → 9 ms  |  |  |  |  |  |  |  |  |        |
| 304 | GET  | jquery-3.2.1... | code.jquery.com | script   | js   | cached  | 84.63 KB | → 58 ms |  |  |  |  |  |  |  |  |        |
|     | GET  | stream          | localhost:9292  | xhr      |      |         |          |         |  |  |  |  |  |  |  |  |        |
| 404 | GET  | favicon.ico     | localhost:9292  | img      | html | 70 B    | 0 B      | → 6 ms  |  |  |  |  |  |  |  |  |        |
| 200 | POST | messages        | localhost:9292  | xhr      | html | 97 B    | 33 B     |         |  |  |  |  |  |  |  |  | → 2 ms |

If you see a never ending GET request it's probably SSE.



If you see many long requests one after another it's most likely Long Polling.  
Eg. This is from FreeNode, but you would see the same on Messenger.

|   | 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              |

Dropped client detection and reconnection handling is something that most libraries implement for those technologies. There is no clear winner here.

The take away is that, in my experience, 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 or LP.  
 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.



Thank you for your patience.  
If anybody has questions I'll accept them now.





[goo.gl/wZKhkB](https://goo.gl/wZKhkB)

github.com/monorkin  
@monorkin  
monorkin@mastodon.social  
hey@stanko.io  
stanko.io

Also, if you would be so kind as to rate my talk on [joinind.in](https://joinind.in), the link and QR code are on-screen.  
I would really appreciate it.