**Before WebSockets: The birth of XMLHttpRequest and AJAX**

Two of the most significant capabilities introduced at that time were the ability to embed Java applets into a page, and Microsoft’s own offering – ActiveX controls.

These were essentially precompiled components that could optionally present an embedded user interface of their own within a web page. More than that though, they allowed a whole host of additional possibilities beyond JavaScript’s own (at the time) meager suite of scripting capabilities.

While there were a few comparable networking capabilities available via Java, the most significant background communication feature first appeared in 1999, with the Microsoft XMLHTTP ActiveXObject interface. It was available natively in Internet Explorer 5.0 without installing plugins, could be instantiated with a single line of JavaScript, and didn’t bring any of the usual friction when dealing with Java applets.

The XMLHTTP object made it possible to silently issue a request to a server and receive a response – all without reloading the page or otherwise interrupting the user’s experience. JavaScript code could then decipher the response and construct modifications to the page, enabling a whole host of rich experiences to be integrated into a website.

Common early use cases included things like allowing a drop-down box to be populated with options based on a user’s prior input, and ‘instant’ validation of username availability while filling out a user registration form.

XMLHTTP later became the XMLHttpRequest de facto standard due to its adoption by other browsers. This was about the time that the term “AJAX” was coined, standing for "Asynchronous JavaScript and XML”.

The JSON standard later came along and made everything better, but the ‘X’ in AJAX (not to mention the ‘XML’ in XMLHttpRequest) never really went away, despite actual XML-formatted data having largely disappeared from standard messaging payloads.

**Ajax Polling vs Long-Polling vs WebSockets vs Server-Sent Events vs HTTP/2 Server Push**

In realtime applications it goes without saying that we need information from our servers as soon as it’s available – and, fundamentally, the classic HTTP request/response paradigm isn’t up to the job. That’s because the server will be silent, new data or not, unless or until a consumer requests an update.

All these technologies and approaches – from Comet to HTTP long polling – have one thing in common: Essentially, they set out to create the illusion of truly realtime (event-driven) data exchange/communication, so when the server has some new data, it sends a response.

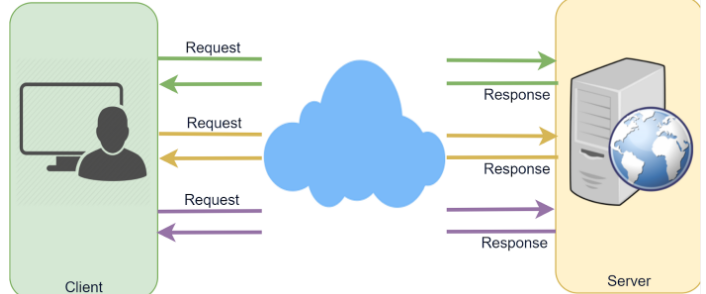
Even though HTTP is not an event-driven protocol, so is not truly realtime, these approaches actually work quite well in specific use cases, Gmail chat for instance. Problems emerge, however, in low-latency applications or at scale, mainly because of the processing demands associated with HTTP.

**Ajax Polling**

Polling is a standard technique used by the vast majority of AJAX applications. The basic idea is that the client repeatedly polls (or requests) a server for data. The client makes a request and waits for the server to respond with data. If no data is available, an empty response is returned.

1. The client opens a connection and requests data from the server using regular HTTP.
2. The requested webpage sends requests to the server at regular intervals (e.g., 0.5 seconds).
3. The server calculates the response and sends it back, just like regular HTTP traffic.
4. The client repeats the above three steps periodically to get updates from the server

The problem with Polling is that the client has to keep asking the server for any new data. As a result, a lot of responses are empty, creating HTTP overhead.



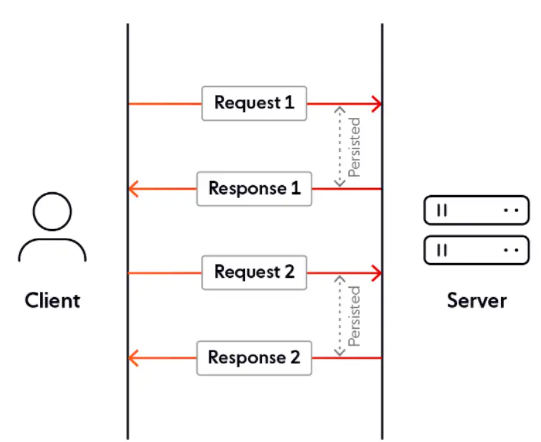
**HTTP Long-Polling**

This variation of the traditional polling technique allows the server to push information to a client whenever the data is available. With Long-Polling, the client requests information from the server exactly as in regular polling, but with the expectation that the server may not respond immediately. That’s why this technique is sometimes referred to as a “Hanging GET.”

* If the server does not have any data available for the client, instead of sending an empty response, the server holds the request and waits until some data becomes available.
* Once the data becomes available, a complete response is sent to the client. The client then immediately re-requests information from the server so that the server will almost always have an available waiting request that it can use to deliver data in response to an event.

The basic life cycle of an application using HTTP Long-Polling is as follows:

1. The client makes an initial request using regular HTTP and then waits for a response.
2. The server delays its response until an update is available or a timeout has occurred.
3. When an update is available, the server sends a complete response to the client.
4. The client typically sends a new long-poll request, either immediately upon receiving a response or after a pause to allow an acceptable latency period.
5. Each Long-Poll request has a timeout. Therefore, the client has to reconnect periodically after the connection is closed due to timeouts.



**WebSockets**

WebSocket provides Full duplex communication channels over a single TCP connection. It provides a persistent connection between a client and a server that both parties can use to start sending data at any time. The client establishes a WebSocket connection through a process known as the WebSocket handshake. If the process succeeds, then the server and client can exchange data in both directions at any time. The WebSocket protocol enables the communication between a client and a server with lower overheads, facilitating real-time data transfer from and to the server.

WebSocket is a standard protocol that provides a persistent connection between a server and a client. WebSockets are bidirectional. This means a server and client and send and receive data are in the same channel. It is a full-duplex communication protocol implemented on TCP/IP socket.

WebSockets help counter the limits of the HTTP protocol.

First, the HTTP protocol is not bidirectional. The client requests a specific resource on the server. Once the server finds and sends the resource to the client, the connection closes. That means on a very active data flow such as streaming service, there will be too many server requests.

Unlike HTTP, WebSockets can maintain a connection until either the client or the server terminates it. It works by first creating a handshake between the client and the server, followed by an UPGRADE header. Once established, a flow of data between the server and the client is established.



**Server-Sent Events (SSEs)**

Under SSEs, the client establishes a persistent and long-term connection with the server. The server uses this connection to send data to a client.

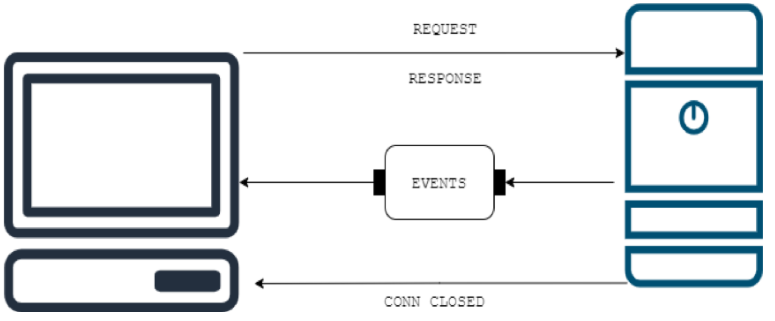
Server-Sent Event (SSEs) is a technology that allows the client to receive updates from an HTTP server. Although it has always been possible to push updates from the server to the client, the client would have to request if any updates exist on the server. Using SSEs, updates are automatic.

SSEs are implemented using regular HTTP data streams. Therefore, SSEs are limited to the client’s (browser) connection pool of 6 concurrent HTTP connections to one server. However, they do not provide the functionality to detect a dropped client.

If the client wants to send data to the server, it would require another technology/protocol to do so.

1. Client requests data from a server using regular HTTP.
2. The requested webpage opens a connection to the server.
3. The server sends the data to the client whenever there’s new information available.

SSEs are best when we need real-time traffic from the server to the client or if the server is generating data in a loop and send multiple events to the client.



**HTTP/2**

HTTP/2 or HTTP2 is the second implementation of the HTTP network protocol used to define the format and transmission of data. The purpose of HTTP/2 is to enhance performance over HTTP by reducing latency, applied by enabling features such as full request and response, and minimizing protocol overhead via compression of header files.

HTTP/2 is supported in major browsers and used all around the web.

The following are some of the advantages offered by HTTP/2:

* Backward compatible with HTTP/1, including status codes, headers, and URIs are reserved.
* Multiple data stream in a single connection via Request multiplexing.
* Header compression, which improves performance significantly.
* Task execution via binary protocol instead of text commands which simplifies command application.
* **Server push** allowing the server to send additional data to the requesting client even if the data is not initially requested.

It removes features such as domain sharding

**Where to use what**

**WebSockets** should be used when we need to constantly update to data (chatting application, биржа, …)

**AJAX polling** is used when we just update something on the page, without refreshing the browser page

**Long Polling** when we need to update data, but time is not vital

