# HTTP and Websockets: Understanding the capabilities of today’s web communication technologies

Deciding what to choose for your next web API design

There are so many classifications for APIs. But when it comes to web communication, we can identify two significant API types — **Web Service APIs** (e.g. SOAP, JSON-RPC, XML-RPC, REST) and **Websocket APIs**. But, what do these really mean? Let’s dive into the world of web communication protocols and discuss how to choose the best API mechanisms at the end.

**1) HTTP**

HTTP is the *underlying communication protocol* of World Wide Web. **HTTP functions as a request–response protocol in the client–server computing model.**HTTP/1.1 is the most common version of HTTP used in modern web browsers and servers. In comparison to early versions of HTTP, this version could implement critical performance optimizations and feature enhancements such as persistent and pipelined connections, chunked transfers, new header fields in request/response body etc. Among them, the following two headers are very notable, because most of the modern improvements to HTTP rely on these two headers.

* Keep-Alive header to set policies for long-lived communications between hosts (timeout period and maximum request count to handle per connection)
* Upgrade header to switch the connection to an enhanced protocol mode such as HTTP/2.0 (h2,h2c) or Websockets (websocket)

If you are interested in knowing what these really do, I have documented all important information for you in the below article.

**2) REST**

The architectural style, REST (REpresentational State Transfer) is by far the most standardized way of structuring the web APIs for requests. REST is purely an architectural style based on several principles. The APIs adhering to REST principles are called *RESTful APIs*. REST APIs use a request/response model where every message from the server is the response to a message from the client. In general, RESTful APIs uses HTTP as its transport protocol. For such cases, **lookups** should use GET requests. PUT, POST, and DELETE requests should be used for **mutation, creation**, and**deletion** respectively(avoid using GET requests for updating information).

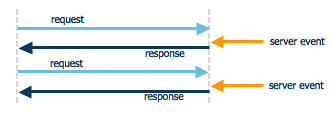
**3) HTTP Polling**

In HTTP Polling, the client polls the server requesting new information by adhering to one of the below mechanism. Polling is used by the vast majority of applications today and most of the times goes with RESTful practices. In practice, HTTP Short Polling is very rarely used and HTTP Long Polling or Periodic Polling is always the choice.

* **HTTP Short Polling**: Simpler approach. A lot of requests are processed as they come to server, creating a lot of traffic (uses resources, but frees them as soon as response is sent back). Since each connection is only open for a short period of time, many connections can be time-multiplexed.

00:00:00 C-> Is the cake ready?   
00:00:01 S-> No, wait.  
00:00:01 C-> Is the cake ready?  
00:00:02 S-> No, wait.  
00:00:02 C-> Is the cake ready?   
00:00:03 S-> Yeah. Have some lad.  
00:00:03 C-> Is the other cake ready?

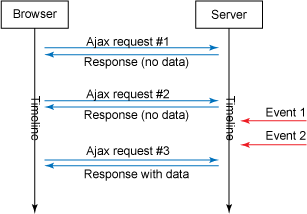
* **HTTP Long Polling**: One request goes to server and client is waiting for the response to come. The server holds the request open until new data is available (it’s unresolved and resources are blocked). You are notified with no delay when the server event happens. More complex and more server resources used.



HTTP Long Polling — Response is held until server process data (Image from [shyamapadabatabyal.wordpress.com](https://shyamapadabatabyal.wordpress.com/))

12:00 00:00:00 C-> Is the cake ready?   
12:00 00:00:03 S-> Yeah. Have some lad.  
12:00 00:00:03 C-> Is the other cake ready?

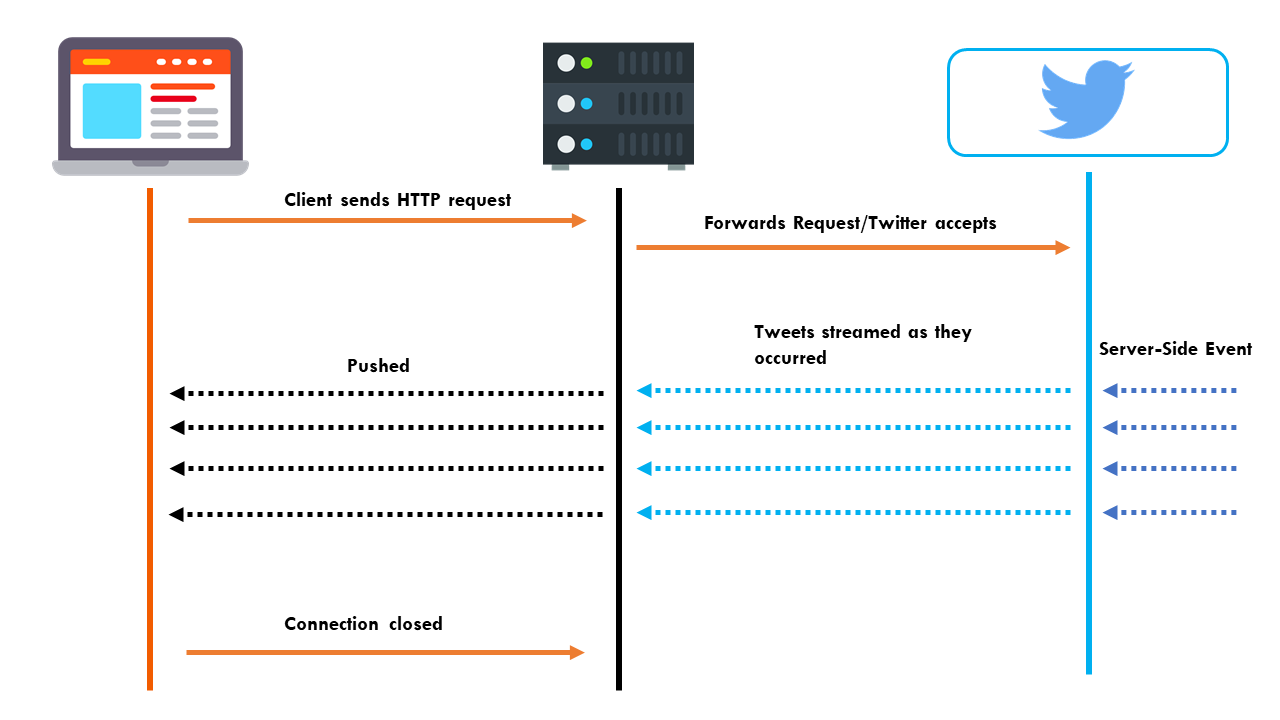
* **HTTP Periodic Polling**: There’s a predefined time gap between two requests. This is an improved/managed version of polling. You can reduce server consumption by increasing time gap between two requests. But if you need to be notified with no delay when the server event happens, this is not a good option.



HTTP Periodic Polling — Request is sent every n seconds (Image from ibm.com)

00:00:00 C-> Is the cake ready?   
00:00:01 S-> No, wait.  
00:00:03 C-> Is the cake ready?  
00:00:04 S-> Yeah. Have some lad.  
00:00:06 C-> Is the other cake ready?

**4) HTTP Streaming**



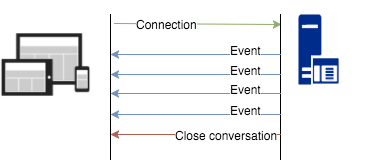
HTTP Streaming — provides a long-lived connection for instant and continuous data push (Image from [realtimeapi.io](https://realtimeapi.io/hub/getting-started-realtime/))

Client makes an HTTP request, and the server trickles out a response of indefinite length (it’s like polling infinitely).HTTP streaming is performant, easy to consume and can be an alternative to websockets.

* Issue: Intermediaries can interrupt the connection (e.g. timeout, intermediaries serving other requests in round-robin manner). In such cases, it cannot guarantee the complete realtimeness.

00:00:00 CLIENT-> I need cakes   
00:00:01 SERVER-> Wait for a moment.  
00:00:01 SERVER-> Cake-1 is in process.  
00:00:02 SERVER-> Have cake-1.  
00:00:02 SERVER-> Wait for cake-2.  
00:00:03 SERVER-> Cake-2 is in process.  
00:00:03 SERVER-> You must be enjoying cake-1.  
00:00:04 SERVER-> Have cake-2.  
00:00:04 SERVER-> Wait for cake-3.  
00:00:05 CLIENT-> Enough, I'm full.

**5) SSE (Server Sent Events / EventSource)**



SSE — events can be broadcast to multiple clients (Image from[javaee.ch](http://javaee.ch/2016/04/17/java-server-sent-event-automatically-update-web-pages/))

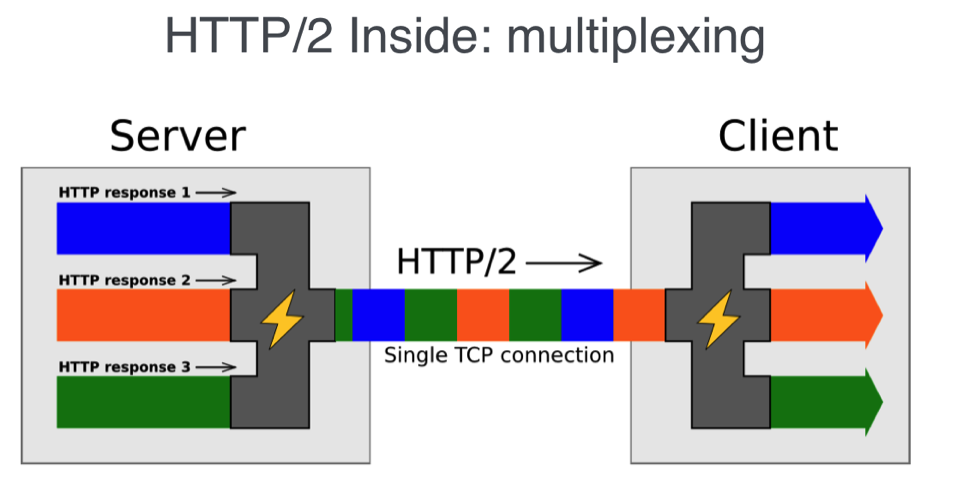
* SSE connections can only push data to the browser. (communication is carried out from server to browser only, browsers can only subscribe to data updates originated by server, but cannot send any data to the server)

00:00:00 CLIENT-> I need cakes   
00:00:02 SERVER-> Have cake-1.  
00:00:04 SERVER-> Have cake-2.  
00:00:05 CLIENT-> Enough, I'm full.

* Sample applications: Twitter updates, stock quotes, cricket scores, notifications to browser
* Issue #1: [Some browsers don’t support SSE](https://caniuse.com/#feat=eventsource).
* Issue #2: Maximum number of open connections is limited to 6 or 8 over HTTP/1.1 (based on browser version). If you use HTTP/2, there won’t be an issue because one single TCP connection is enough for all requests (thanks to multiplexed support in HTTP/2).

**6) HTTP/2 Server Push**

* A mechanism for a server to proactively push assets (stylesheets, scripts, media) to the client cache in advance
* Sample applications: Social media feeds, single page apps



HTTP/2 is an efficient transport layer based on multiplexed streams (Image from [SessionStack.com](https://blog.sessionstack.com/how-javascript-works-deep-dive-into-websockets-and-http-2-with-sse-how-to-pick-the-right-path-584e6b8e3bf7)) — According to IETF, a “stream” is an independent, bidirectional sequence of frames exchanged between the client and server within an HTTP/2 connection. One of its main characteristics is that a single HTTP/2 connection can contain multiple concurrently open streams, with either endpoint interleaving frames from multiple streams.

* Issue #1: Intermediaries (proxies, routers, hosts) can choose not to properly push information to client as intended by the origin server.
* Issue #2: Connections aren’t kept open indefinitely. A connection can be closed anytime even when the content pushing process happens. Once closed and opened again, these connection cannot continue from where it left.
* Issue #3: Some browsers/intermediaries don’t support Server Push.

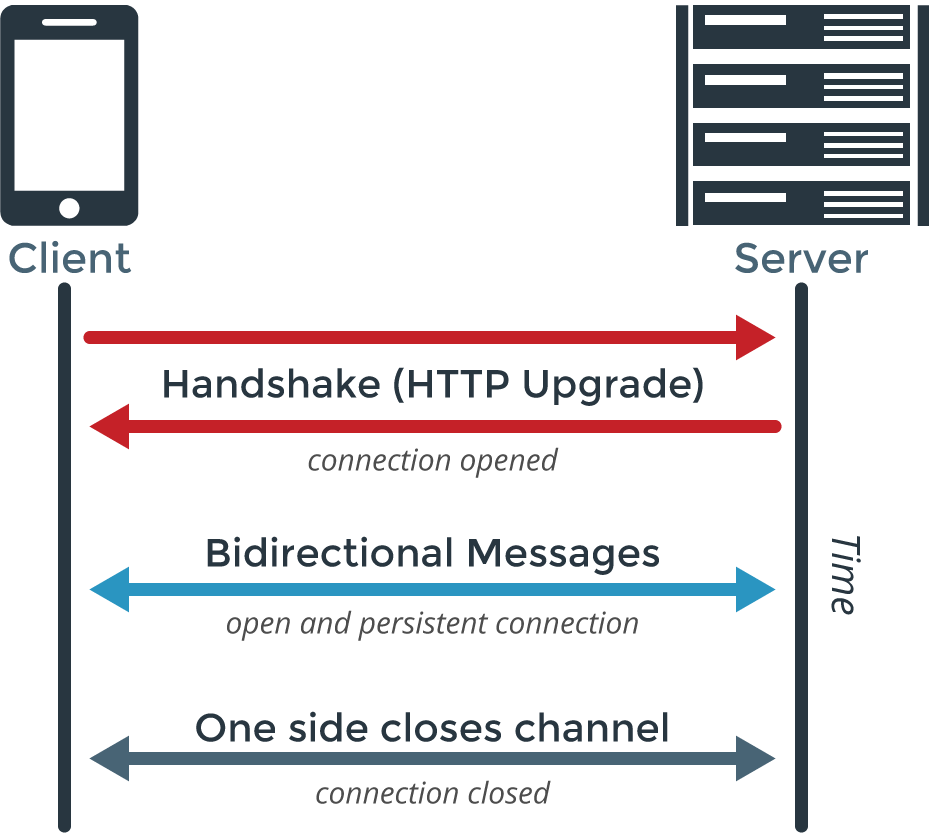
**7) Websockets**

WebSockets allow both the server and the client to push messages at any time without any relation to a previous request. One notable advantage in using websockets is, [almost every browser support websockets](https://caniuse.com/#feat=websockets).

WebSocket solves a few issues with HTTP:

* Bi-directional protocol — either client/server can send a message to the other party (In HTTP, the request is always initiated by client and the response is processed by server — making HTTP a uni-directional protocol)
* Full-duplex communication — client and server can talk to each other independently at the same time.
* Single TCP connection — After upgrading the HTTP connection in the beginning, client and server communicate over that same TCP connection throughout the lifecycle of Websocket connection.

00:00:00 CLIENT-> I need cakes   
00:00:01 SERVER-> Wait for a moment.  
00:00:01 CLIENT-> Okay, cool.  
00:00:02 SERVER-> Have cake-1.  
00:00:02 SERVER-> Wait for cake-2.  
00:00:03 CLIENT-> What is this flavor?  
00:00:03 SERVER-> Don't you like it?  
00:00:04 SERVER-> Have cake-2.  
00:00:04 CLIENT-> I like it.  
00:00:05 CLIENT-> But this is enough.



Websocket connection (Image from [PubNub.com](https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/))

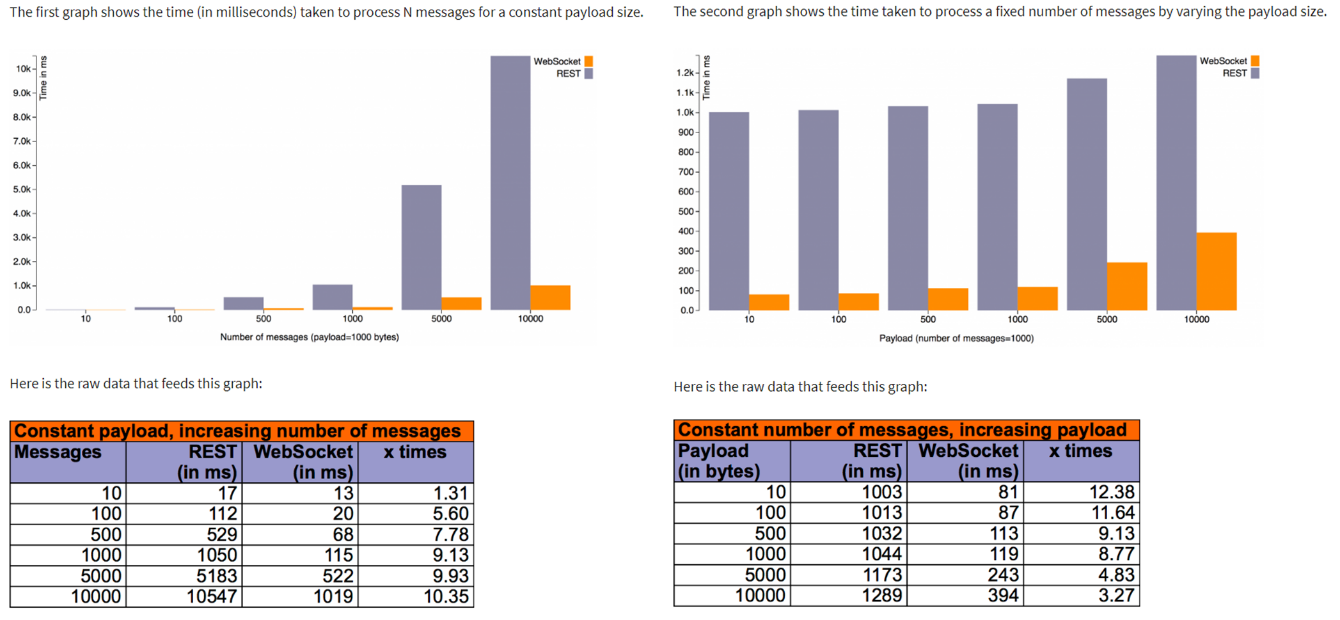
Sample applications: IM/Chat apps, Games, Admin frontends

Although Websockets are said to be supported every browser, there can be exceptions in intermediaries too:

* Unexpected behaviors in intermediaries: If your websocket connections go through proxies/firewalls, you may have noticed that such connections fail all the times. Always use Secured Websockets (WSS) to drastically reduce such failures. This case is nicely explained here: [How HTML5 Web Sockets Interact With Proxy Servers](https://www.infoq.com/articles/Web-Sockets-Proxy-Servers) and here too: [WebSockets, caution required!](https://samsaffron.com/archive/2015/12/29/websockets-caution-required" \t "_blank). So take the caution and get ready to handle them by using WSS and falling back to a supportive protocol.
* Intermediaries that don’t support websockets: If for some reason the WebSocket protocol is unavailable, make sure your connection automatically fallback to a suitable long-polling option.

**8) REST vs Websockets — Perf Test**

If you do a performance test for REST and Websockets, you may find that Websockets do better when high loads are present. This does not necessarily mean that REST is inefficient. My personal opinion is, comparing REST with Websockets is like comparing apples to oranges. These two features solves two different problems and cannot be compared with a simple perf test like this:

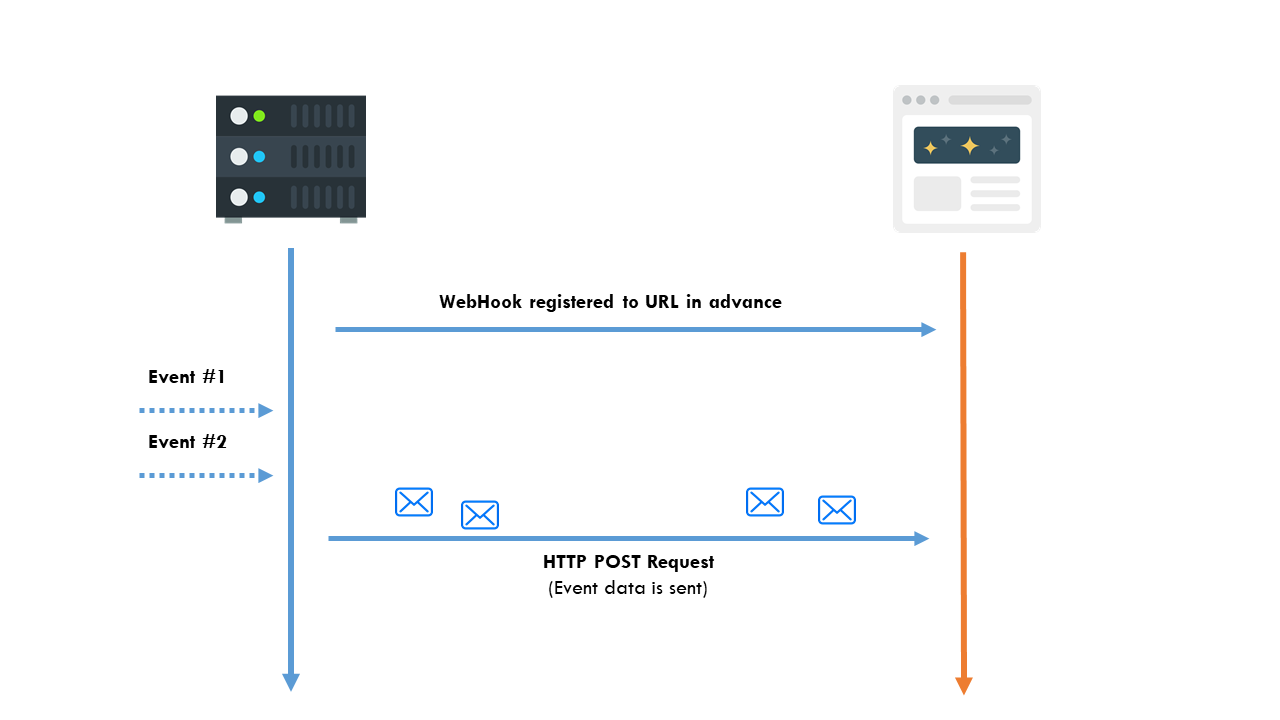


In first graph, REST overhead increases against number of messages because that many TCP connections need to be initiated and terminated and that many HTTP headers need to be sent and received. In the second graph, incremental cost of processing request/response for a REST endpoint is minimal and most of the time is spent in connection initiation/termination and honoring HTTP semantics. (Perf test and analysis from [arungupta.me](http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/))

However, you should now understand that websockets are a great choice for handling long-lived bidirectional data streaming in near real-time manner, whereas REST is great for occasional communications. Using websockets is a considerable investment, hence it is an overkill for occasional connections.

**9) Webhooks (for communication between servers)**

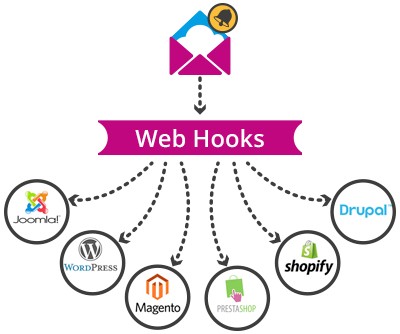
If you want to obtain data from your API on change of data, polling must be the first option that comes to your mind. But when it comes to communication between servers, inefficiencies in polling cost us a lot (on average, 98.5% of polls get wasted).



Webhooks — simple way for sending data between servers with no long-lived polling connections (Image from [realtimeapi.io](https://realtimeapi.io/hub/getting-started-realtime/))

Webhooks are the savior for this problem. Here remember that the communication generally happens between servers. First, the sender node registers a callback URL in the receiver node/s in advance. When an event occurs in the sender side, the webhook gets triggered and sends an event object with new data as a HTTP POST request to the receiver node/s using callback URLs registered in each of them.

The cool thing is, server load for both the sender and receiver nodes can be reduced drastically with webhooks. It ensures the better user experience while your developers can utilize your service endpoints for meaningful things without wasting for polling.



Webhooks are generally used for send notifications and state changes among servers when an event occurs. For an example, when a user unsubscribes via a button click in email, it gets to a server and the user unsubscribe event occurs, this event triggers the corresponding webhooks and they notify all the servers/services that the user has now unsubscribed from their service (Image from [kloudymail.com](http://www.kloudymail.com/wp-content/uploads/2015/06/kloudyman_integrazione_WebHooks3.jpg))

* Sample applications: Notifications when a new user registers or a current user updates an existing profile setting
* Issue: Developers find it difficult to setup webhooks and scale HTTP services.

**What to use for your next API?**

Which technique to use depends on what makes more sense in the context of your application. Sure, you can use some tricks to simulate the behavior of one technology with the other, but it is usually preferable to use the one which fits your communication model better when used by-the-book.

✅ HTTP/1.1 vs HTTP/2: These are transport protocols. Mutiplexing capability in HTTP/2 is great, but not supported everywhere yet. In such cases, make sure falling back from HTTP/2 and HTTP/1.1 won’t create any mess in your application. For transferring data, HTTP/1.1 is still a great choice.

✅ RESTful APIs: So far, RESTful APIs are okay for web applications. But there are discussions happening on exploring better ways. An example is this concept — “[Replace RESTful APIs with JSON-Pure](https://mmikowski.github.io/json-pure/)”. I like the idea, in fact, JSON is developer-friendly and you can do wonders with it. But you know, these are developing concepts.

✅ JSON vs XML: Use JSON. It’s the trend, and so convenient to deal with too.

✅ HTTP Polling: This is old, but still a great choice for dealing with APIs. If your data is changing frequently or in real time, don’t use Short Polling, just use websockets, the better technology for real time. Always use Long and Periodic Polling appropriately (with REST principles).

✅ HTTP Streaming: This is good for applications like news/social media feeds, stock/score boards, tweets etc. But in practice, people use websockets than HTTP Streaming.

⚠️ HTTP/2 Server Push is great for sending resources to client in more managed way. But all intermediaries and browsers will not support this. Make sure you gracefully handle such fallbacks.

⚠️ Server-sent events are a rather new technology which isn’t yet supported by all major browsers, so it is not yet an option for a serious enterprise level web application (*Yes, I agree that you can trick this technology to work*).

✅ WebSockets provide a richer protocol to perform bi-directional, full-duplex communication. Having a two-way channel is more attractive for things like games, messaging apps, collaboration tools, interactive experiences (inc. micro-interactions), and for cases where you need real-time updates in both directions.

✅ Webhooks are different from all above technologies because it solves a very specific problem. If your servers need to communicate frequently and/or bidirectionally, go for websockets. If your servers communicate occasionally, use REST calls. If your servers need to communicate unidirectionally on an event trigger, then go for webhooks. Don’t use polling to check changes in data or state, it’s a waste.

**Any tips?**

☝️ Proxies and intermediaries are crazy. They will eat up your packets or timeout unexpectedly. Be aware of that, and handle it gracefully.

☝️ Use secured transport protocols (HTTPS or WSS) to handle your communication. Then, intermediaries will have less impact on your connections. And it’s secure, you know.

☝️ Developers love webhooks. But make sure you apply it right.

☝️ You really need not to embrace the latest technology always, especially when you create enterprise level applications. Make sure all the infrastructure is supporting the technology you use. And if the infrastructure does not support your technology/architecture, make sure it falls back to a technology that just works everywhere and everything works gracefully with reduced capabilities.z

**References:**

* [REST vs WebSocket Comparison and Benchmarks](http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/)
* [Short-polling vs Long-polling for real time web applications?](https://stackoverflow.com/questions/4642598/short-polling-vs-long-polling-for-real-time-web-applications)
* [Getting Started with Realtime](https://realtimeapi.io/hub/getting-started-realtime/)

# Evolution of HTTP — HTTP/0.9, HTTP/1.0, HTTP/1.1, Keep-Alive, Upgrade, and HTTPS

Understanding how HTTP works in the real world

Disclaimer: This article focuses on explaining some underlying implementation details of HTTP, which will be helpful for readers to better understand my blog article — “[**Web API Design with HTTP and Websockets**](https://medium.com/platform-engineer/web-api-design-35df8167460)”

Invented by Tim Berners-Lee at CERN in the years 1989–1991, HTTP (Hypertext Transfer Protocol) is the underlying communication protocol of World Wide Web. **HTTP functions as a request–response protocol in the client–server computing model.** HTTP standards are developed by the [Internet Engineering Task Force](https://en.wikipedia.org/wiki/Internet_Engineering_Task_Force) (IETF) and the [World Wide Web Consortium](https://en.wikipedia.org/wiki/World_Wide_Web_Consortium) (W3C), culminating in the publication of a series of [Requests for Comments](https://en.wikipedia.org/wiki/Requests_for_Comments) (RFCs). HTTP has four versions — HTTP/0.9, HTTP/1.0, HTTP/1.1, and HTTP/2.0. Today the version in common use is HTTP/1.1 and the future will be HTTP/2.0.

## 1) HTTP/0.9 — The One-line Protocol

* Initial version of HTTP — a simple client-server, request-response, telnet-friendly protocol
* Request nature: single-line (method + path for requested document)
* Methods supported: GET only
* Response type: hypertext only
* Connection nature: terminated immediately after the response
* No HTTP headers (cannot transfer other content type files), No status/error codes, No URLs, No versioning

$> telnet ashenlive.com 80**(Connection 1 Establishment - TCP Three-Way Handshake)**Connected to xxx.xxx.xxx.xxx**(Request)**  
GET /my-page.html**(Response in hypertext)**  
<HTML>  
A very simple HTML page  
</HTML>**(Connection 1 Closed - TCP Teardown)**

Popular web servers (Apache, Nginx) still supports HTTP/0/9. Try opening up a Telnet session and accessing google.com

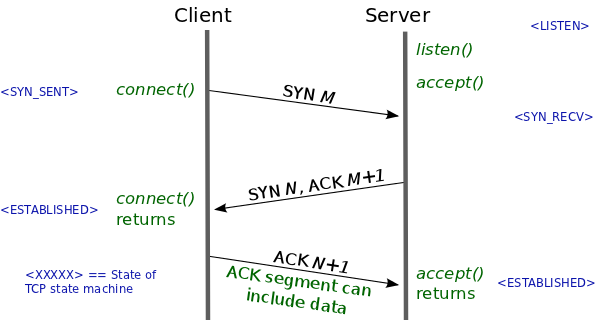
## 2) HTTP/1.0 — Building extensibility

* Browser-friendly protocol
* Provided header fields including rich metadata about both request and response (HTTP version number, status code, content type)
* Response: not limited to hypertext (Content-Type header provided ability to transmit files other than plain HTML files — e.g. scripts, stylesheets, media)
* Methods supported: GET , HEAD , POST
* Connection nature: terminated immediately after the response

**(Connection 1 Establishment - TCP Three-Way Handshake)**  
Connected to xxx.xxx.xxx.xxx**(Request)**  
GET /my-page.html HTTP/1.0   
User-Agent: NCSA\_Mosaic/2.0 (Windows 3.1)**(Response)**HTTP/1.0 200 OK   
Content-Type: text/html   
Content-Length: 137582  
Expires: Thu, 01 Dec 1997 16:00:00 GMT  
Last-Modified: Wed, 1 May 1996 12:45:26 GMT  
Server: Apache 0.84  
  
<HTML>   
A page with an image  
 <IMG SRC="/myimage.gif">  
</HTML>**(Connection 1 Closed - TCP Teardown)------------------------------------------(Connection 2 Establishment - TCP Three-Way Handshake)**  
Connected to xxx.xxx.xxx.xxx**(Request)**GET /myimage.gif HTTP/1.0  
User-Agent: NCSA\_Mosaic/2.0 (Windows 3.1)  
  
**(Response)**  
HTTP/1.0 200 OK   
Content-Type: text/gif   
Content-Length: 137582  
Expires: Thu, 01 Dec 1997 16:00:00 GMT  
Last-Modified: Wed, 1 May 1996 12:45:26 GMT  
Server: Apache 0.84[image content]**(Connection 2 Closed - TCP Teardown)**

## 3) Establishing a new connection for each request — major problem in both HTTP/0.9 and HTTP/1.0

Both HTTP/0.9 and HTTP/1.0 required to open up a new connection for each request (and close it immediately after the response was sent). Each time a new connection establishes, a TCP three-way handshake should also occur. For better performance, it was crucial to reduce these round-trips between client and server. HTTP/1.1 solved this with *persistent connections*.

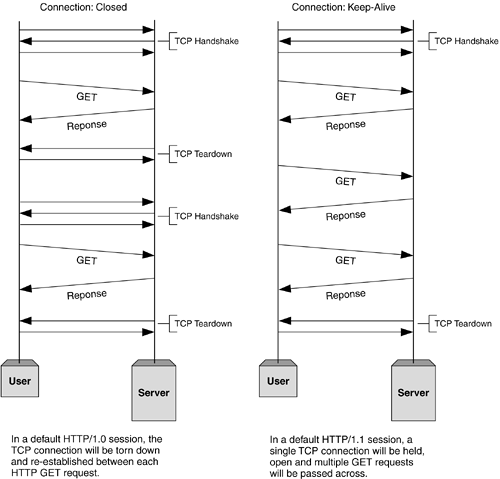


A typical TCP Three-way Handshake (see how the TCP state machine is changing its state) from [lwn.net](https://lwn.net/Articles/508865/)

# HTTP/1.1 — The standardized protocol

* This is the HTTP version currently in common use.
* Introduced critical performance optimizations and feature enhancements — persistent and pipelined connections, chunked transfers, compression/decompression, content negotiations, virtual hosting (a server with a single IP Address hosting multiple domains), faster response and great bandwidth savings by adding cache support.
* Methods supported: GET , HEAD , POST , PUT , DELETE , TRACE , OPTIONS
* Connection nature: long-lived

**(Connection 1 Establishment - TCP Three-Way Handshake)**  
Connected to xxx.xxx.xxx.xxx**(Request 1)**  
GET /en-US/docs/Glossary/Simple\_header HTTP/1.1  
Host: developer.mozilla.org  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate, br  
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple\_header  
  
**(Response 1)**  
HTTP/1.1 200 OK  
Connection: Keep-Alive  
Content-Encoding: gzip  
Content-Type: text/html; charset=utf-8  
Date: Wed, 20 Jul 2016 10:55:30 GMT  
Etag: "547fa7e369ef56031dd3bff2ace9fc0832eb251a"  
Keep-Alive: timeout=5, max=1000  
Last-Modified: Tue, 19 Jul 2016 00:59:33 GMT  
Server: Apache  
Transfer-Encoding: chunked  
Vary: Cookie, Accept-Encoding  
  
[content]  
  
**(Request 2)**  
GET /static/img/header-background.png HTTP/1.1  
Host: developer.cdn.mozilla.net  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0  
Accept: \*/\*  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate, br  
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple\_header  
  
**(Response 2)**  
HTTP/1.1 200 OK  
Age: 9578461  
Cache-Control: public, max-age=315360000  
Connection: keep-alive  
Content-Length: 3077  
Content-Type: image/png  
Date: Thu, 31 Mar 2016 13:34:46 GMT  
Last-Modified: Wed, 21 Oct 2015 18:27:50 GMT  
Server: Apache  
  
[image content of 3077 bytes]**(Connection 1 Closed - TCP Teardown)**



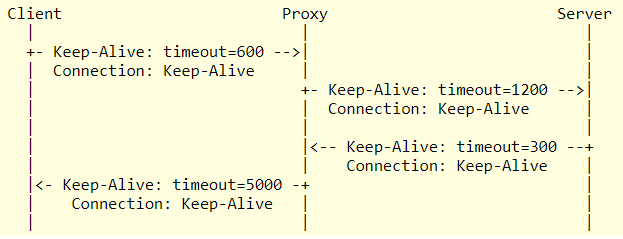
Before establishing any connection, a TCP three-way handshake happens. At the end, after sending all data to Client, Server sends a message saying there’s no more data to send. Then the client closes the connection (TCP teardown). The problem in HTTP/1.0 is, for each request-response cycle, a connection needs to be opened and closed. And the advantage of using HTTP/1.1 is, we can reuse the same open connection for multiple request-response cycles. (Image from [informit.com](http://www.informit.com/articles/article.aspx?p=169578))

# Keep-Alive and Upgrade headers

## Keep-Alive header

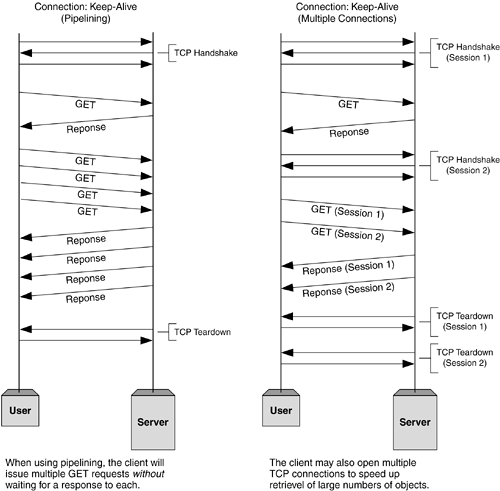
* The Keep-Alive header was used prior to HTTP/1.1 and was obsoleted by HTTP/1.1 making persistent connections the default behavior. Keep-Alive header can be used to define policies for long-lived communication between hosts (i.e. allows a connection to stay active until an event occurs). This laid foundation for persistence, reusable connections, pipelining, and many more enhanced capabilities in modern web communication protocols.
* Client, server, or any intermediary can provide information for Keep-Alive header independently. Also, a host can add timeout and max parameters in order to set a timeout or limit maximum request count per connection.

HTTP/1.1 200 OK  
*Connection: Keep-Alive*  
Content-Encoding: gzip  
Content-Type: text/html; charset=utf-8  
Date: Thu, 11 Aug 2016 15:23:13 GMT  
*Keep-Alive: timeout=5, max=1000*  
Last-Modified: Mon, 25 Jul 2016 04:32:39 GMT  
Server: Apache  
  
[body]



This example from [ietf.org](https://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html) shows how a Keep-Alive header could be used. **All connections are independently negotiated**. The client indicates a timeout of 600 seconds (10 minutes), but the proxy is only prepared to retain the connection for at least 120 seconds (2 minutes). On the link between proxy and server, the proxy requests a timeout of 1200 seconds and the server reduces this to 300 seconds. As this example shows, the timeout policies maintained by the proxy are different for each connection. **Each connection hop is independent**.

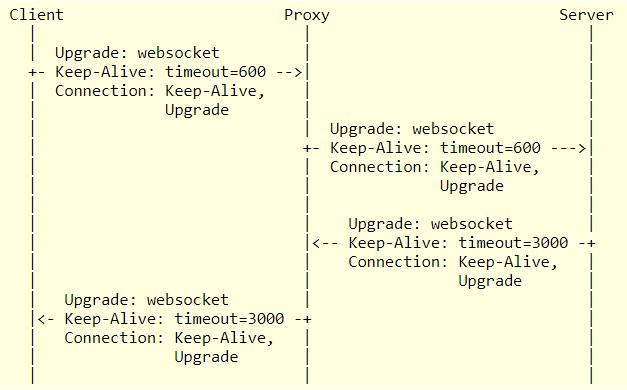
* HTTP pipelining, multiple connections, and many more improvements have been implemented, thanks to the Keep-Alive header’s behavior.



HTTP Pipelining and Multiple Parallel Connections (Image from [informit.com](http://www.informit.com/articles/article.aspx?p=169578))

## Upgrade header

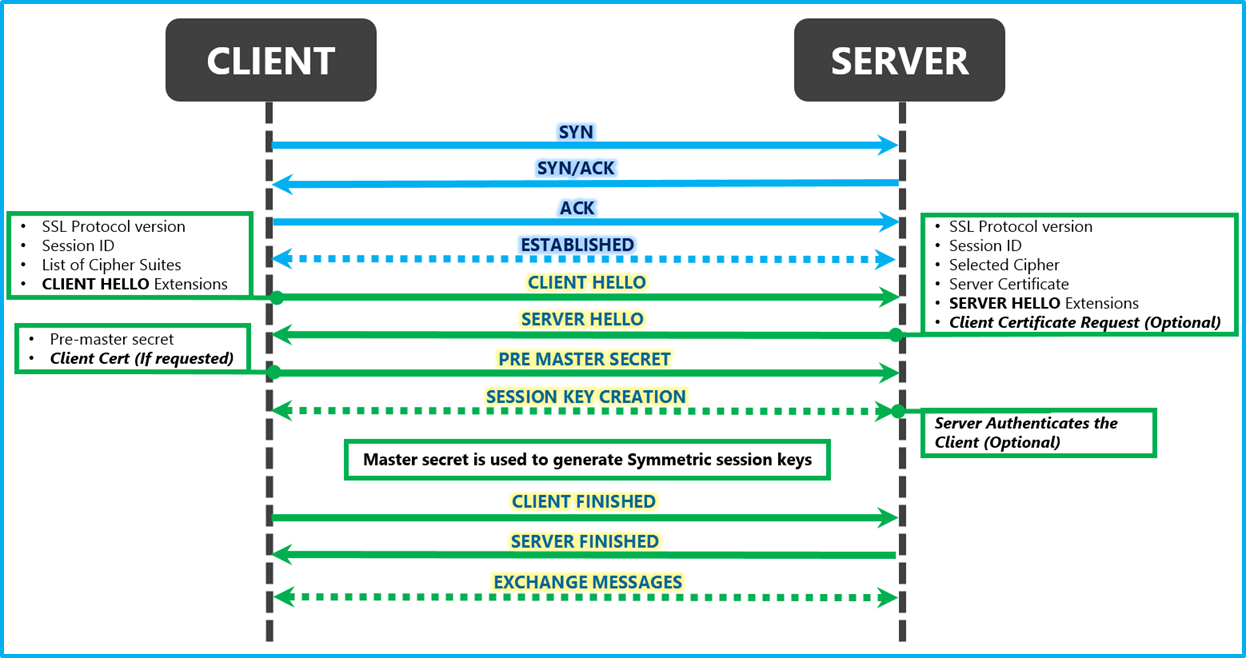
* With Upgrade header introduced in HTTP/1.1, it is possible to start a connection using a commonly-used protocol, such as HTTP/1.1, then request that the connection switch to an enhanced protocol type like HTTP/2.0 or WebSockets.
* In an upgraded protocol connection, max parameter (maximum request count) is not present. The upgraded protocol can provide new policies for timeout parameter (if not specifically defined, it uses default timeout value in underlying protocol).



This example from [ietf.org](https://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html) shows the headers included in an upgrade from HTTP/1.1 to [WebSocket](https://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html" \l "RFC6455" \t "_blank)[RFC6455]. With a websocket upgrade, the connections on each hop cannot have independent lifecycles on either side of an intermediary. After the upgrade, timeout policies cannot be independent for each connection. The proxy adjusts the timeout value to reflect the lower of the values set by client and the proxy policies so that the server is aware of the connection characteristics; similarly, the value from the server is provided to the client. Upgrade is said to be a hop-by-hop header.

# HTTPS

* Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP. It uses SSL/TLS for secure encrypted communications.
* Originally developed by Netscape in mid-1990s, SSL (Secure Socket Layer) is a cryptographic protocol enhancement to HTTP, which defines how client and server should communicate with each other securely. TLS (Transport Layer Security) is the successor of SSL.
* An HTTPS connection can protect the data transfer from the man-in-the-middle attacks and common security threats by providing bidirectional encryption for communications between a client and server.



Diagrammatic representation of the SSL Handshake from [msdn.microsoft.com](https://blogs.msdn.microsoft.com/kaushal/2013/08/02/ssl-handshake-and-https-bindings-on-iis/) — TCP Connection > SSL/TLS Client Hello > SSL/TLS Server Hello > SSL/TLS Certificate > SSL/TLS Client Key Exchange > SSL/TLS New Session Ticket > HTTPS Encrypted Data Exchange

## SSL/TLS Handshake — major problem in HTTPS

* Although HTTPS is secure by its design, the SSL/TLS handshake process consumes a significant time before establishing an HTTPS connection. It normally costs 1–2 seconds and drastically slows down the startup performance of a website.

# HTTP/2.0 and the future

All above features are being used by major web servers and browsers today. But modern enhancements like HTTP/2.0, Server Side Events (SSE), and Websockets have changed the way that the traditional HTTP works. In my next article on [Web API Design with HTTP and Websockets](https://medium.com/platform-engineer/web-api-design-35df8167460), we will discuss how we should choose them in real-world projects.

## References:

1. [Evolution of HTTP — MDN Web Docs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP)
2. [Hypertext Transfer Protocol (HTTP) Keep-Alive Header — ietf.org](https://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html)
3. [Protocol upgrade mechanism — MDN Web Docs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Protocol_upgrade_mechanism)
4. [Brief History of HTTP — High Performance Browser Networking (hpbn.co)](https://hpbn.co/brief-history-of-http/)