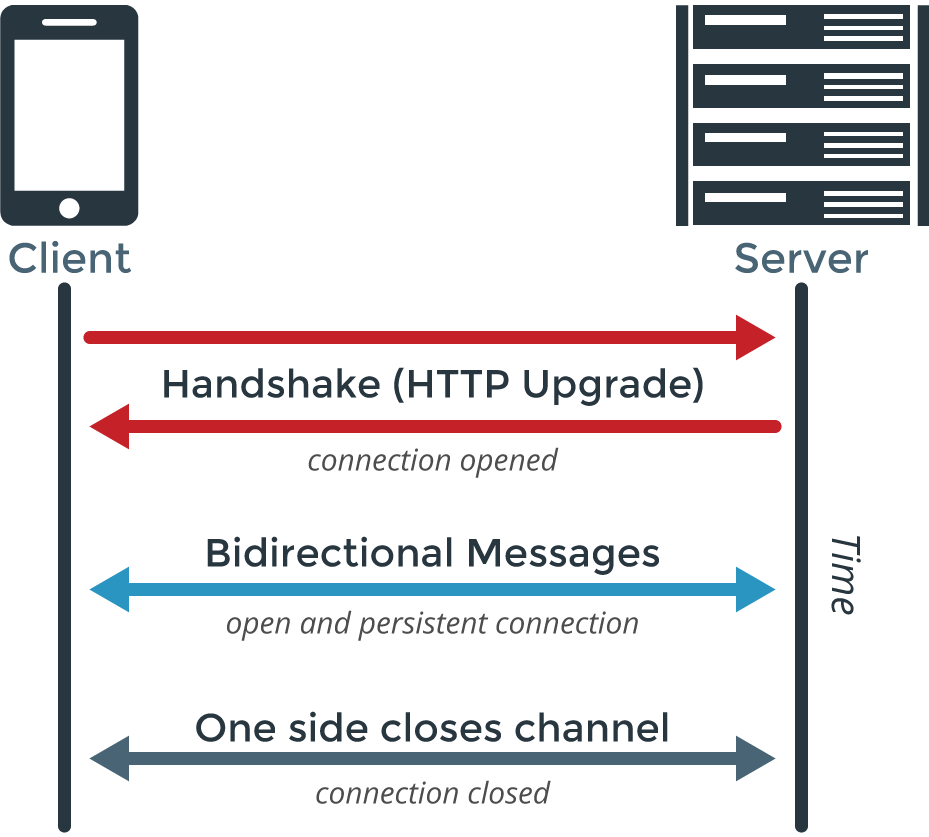
I don't think a clear comparison of Websockets and HTTP is possible, since they're no rivals nor solve the same problems. Websockets are useful for chunks of binary data, custom low level protocols and server pushs (e.g. status updates in the message stream of facebook). Using HTTP you have to rely on a set of problematic techniques called Comet [1] to implement sever pushs. HTTP on the other hand is the medium of choice for the typical request response messaging style of 99% of all web applications.  
  
**Performance:**Websockets are essentially raw TCP-Sockets and therefore don't have the overhead HTTP has (which is also transferred over TCP). On the other hand HTTP can be very fast due to local and network caching, avoiding a connection in the first place.  
  
**Memory/CPU/ResponseTime:** will hence be rougly the same, since the parsing overhead for HTTP is rather small. Only content-hashes, authentication tokens and  compression will have some (probably imperceptible) impact on the speed of HTTP. The response time depends on the round trip time to your server and will be eqal, unless some cache intercepts the connection which would give an advantage to HTTP.

***HTTP*** is based around the concept of 'requesting' a file and then receiving a 'response'. This works really well when you are following links and sending forms, but is very bad at things like delivering notifications and updates to a page. We've tried and sort of lived by with using AJAX for these problems, but it isn't going to be good enough.  
   
***WebSockets*** is a new protocol that allows the server to send things to the browser without the browser having to request it. The browser can also send data to the server. This is quite useful for sending notifications and updates because the server can send them when it gets them, instead of waiting for the browser to ask for them.



What is REST?

In [REST](https://github.com/pubnub/pubnub-evangelist-blog-posts/blob/master/Jasdeep-RESTvsWebsockets.md#rest), or REpresentational State Transfer, is another abstraction for creating API’s for applications in a standardized way. With typical, and now traditional, web applications, creating REST endpoints using HTTP is how the vast majority of applications are architected. Whether it’s Ruby, Java, Go, NodesJS or any of the multitude of technologies available, they are fundamentally similar in that they receive to Requests for information, and then Responding to the request.

REST organizes these Requests in predictable ways, using HTTP operation types, or verbs, to construct appropriate responses. Requests originate from the client, and the common HTTP verbs include GET, POST, PUT, DELETE but there are several others. They correspond to expected operations, retrieving data, submitting data, updating data, and deleting data.

REST is by far the most standardized way of structuring the API for Requests. But since it involves using HTTP is also has the overhead associated with that protocol. For most applications, information only needs to be transferred when a user takes an action. For instance, when browsing a news site, once the browser has requested the article, the user is busy reading it, and not taking actions. Having the port-socket close during this time is actually saving resources. With less frequent interaction, HTTP works very well, and it is why it is used.

For more real time interaction, or real time transfer or streaming of data, HTTP and REST aren’t the best suited protocol and abstraction combination. This is where Sockets and WebSockets shine.

WebSockets vs REST: A Comparison of Performance

The overhead of opening and closing connections is very real. The [performance](https://github.com/pubnub/pubnub-evangelist-blog-posts/blob/master/Jasdeep-RESTvsWebsockets.md#websockets-rest-performance-a-comparison) of being able to send and receive data and the number of concurrent devices that can do so is a significant consideration. The use of polling versus pushing is also a very real burden on servers.

REST Performance

Metaphorically, we could think of this as an army with a chain of command. Let’s make the server the General, and all the browsers the soldiers on the ground waiting for orders. Using HTTP/REST, if every soldier has to ask the General if there are any new orders, it burdens the General tremendously, particularly when there isn’t anything new. That means the General is saying “no, nothing new” most of the time.

If your servers are the General, their resources are being wasted mostly saying that there is nothing new. Furthermore, because the General can only answer so many soldiers at a time, it takes a long time to say “nothing new” to every soldier even once, there is a time delay where the last soldiers in line hear the “nothing new” far later than the first soldier who asked.

