Evaluation of Server Push Technologies for Scalable Client-Server Communication

Elton F. de Souza Soares
Graduate Program in Informatics (UNIRIO)
Rio de Janeiro, RJ, 22290–240
IBM Research
Rio de Janeiro, RJ, 22290–040
Email: elton.soares@uniriotec.br,
eltons@br.ibm.com

Raphael Melo Thiago
IBM Research
Rio de Janeiro, RJ, 22290–040
Email: raphaelt@br.ibm.com

Leonardo Guerreiro Azevedo,
Maximilien de Bayser,
Viviane Torres da Silva,
Renato F. de G. Cerqueira
IBM Research
Rio de Janeiro, RJ, 22290–040
Email: {lga, mbayser, vivianet, rcerq}
@br.ibm.com

***Abstract*—When a consumer (or client) invokes a provider's service that has a long time processing, a synchronous call is not a good option. This kind of communication blocks the consumer until the response arrives, and, besides, the invocation timeout can be reached raising a timeout error. Hence, an asynchronous call is more appropriate where the consumer calls the provider's service and continues processing, and, when the provider's service finishes it pushes the response to the consumer. There are different ways for implementing asynchronous communication. A simple way is the consumer providing a callback function which the provider's service invokes when it finishes the processing, performing a server push operation. However, there are many cases where this solution cannot be applied requiring other alternatives, *e.g.*: the consumer keeps up with the service execution checking for its readiness or state, and when it finishes, it calls another provider's service to get the result; the consumer and provider keep an open connection for the asynchronous communication. This work analyzes the main server push technologies used for web development, presenting their weaknesses and strengths, and the existing main challenges. As a result, we provide a technologies comparison, and a classification based on multiple qualitative dimensions that helps one to choose the technology that fits its requirements and/or can be used to guide future researches in this field.**

I. Introduction

Client-Server communication on the Internet can be based on a wide variety of protocols and technologies which might apply to very different scenarios and problems that web applications might face. One of the main advantages of the layered architecture of the Internet is the possibility of exchanging the protocols and technologies used within each layer to better adapt towards specific performance requirements and constraints [1].

HTTP protocol is the main application layer protocol used to allow interaction between web clients and web services, specially RESTful ones. It's layered over TCP, since it requires reliable data delivery. Web applications that use HTTP protocol for Client-Server communication employ the request-response paradigm: (i) A client requests resources from the server through a HTTP request message; and, (ii) The server returns the requested resource through a HTTP response message [2]. This paradigm works well for web applications in which the requested resource can be returned by the server within the HTTP persistent connection timeout, which is the maximum time the HTTP connection between client and server can be idle [2].

In many applications, those resources are processed dynamically (*e.g.*, an HTML page that is generated by a PHP/MySQL back-end) and, in some of them, these resources might take a very long time to be processed making infeasible to use the traditional request-response paradigm to return its result(*e.g.*, the result of a simulation experiment or a machine learning model training might take hours or even days to be completed). Hence, this class of applications requires a different kind of communication workflow, since the server that provides the requested resource will not be able to return it within the HTTP persistent connection timeout.

In scenarios where both clients and servers have fixed IP addresses (*e.g.*, clients are also servers), message brokers, such as IBM MQ[1], RabbitMQ[2], and Redis[3], provide an adequate solution for the problem. Message Brokers allow asynchronous communication between clients and servers through message queues. Clients can request servers to process resources (*e.g.*, tasks); servers then read and update the resources' state, broadcasting changes through the message queue, which clients can monitor by subscribing to it. However, message brokers cannot be used when clients do not have fixed IPs. Desktop applications, website and mobile apps are examples of clients with non-fixed IP addresses.

Alternatively, desktop applications and mobile apps could communicate directly to the server over the transport layer using TCP sockets instead of HTTP [3], [4]. This solution is not applicable to websites, since in most cases web browsers do not allow raw socket connections to be used directly from website code[4], although this has been specified by W3C[5].

This challenge motivated the development of an assortment of alternative technologies, commonly called server push techniques [5], [6], [7], [8], [9]. These techniques either simulate or

[1]http://www-03.ibm.com/software/products/en/ibm-mq
[2]https://www.rabbitmq.com/
[3]https://redis.io/
[4]https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/API/TCPSocket
[5]https://www.w3.org/TR/raw-sockets/

0-7695-6394-5/18/$31.00 ©2018 IEEE
DOI 10.1109/SOSE.2018.00010

implement the proactive delivery of messages from the server to the client through an application layer protocol, such as HTTP. After reviewing the literature on this subject, we did not find any work that characterizes or analyses the most recent and relevant solutions available.

This work analyzes the main server push technologies, presenting their weaknesses and strengths, and comparing each one. Besides, we perform simulations to support our analysis claims. As a result, we provide a classification based on multiple qualitative and quantitative dimensions that helps one to choose the technology that fits its requirements and/or can be used to guide future researches in this field.

The remainder of this work is divided as follows. Section III presents the existing server push techniques. Section III presents the techniques characterization, pointing their weaknesses and strengths. Section IV presents evaluations of the technologies based on a set of defined criteria. Section V presents simulations of the techniques to support some of the claims resulting from the theoretical analysis. Section VI presents the conclusion, future work and open challenges.

II. Related Works

This section describes the main related works. They either present a survey or compair multiple server push technologies.

Although many works discuss server push solutions in different scenarios [6], [7], [10], [9], [11], [12], [13], [14], [15], [16] most of them focus on the proposal of a new techniques and architectures. From these works, only [10] and [15] present comprehensive surveys of state-of-the-art techniques and compare their performance using simulation/emulation experiments.

In [10] the authors discuss HTTP Pull (*i.e.* HTTP Polling), HTTP Streaming (*i.e.* a predecessor of Server-Sent Events) and the BAYEUX Protocol (*i.e.* Long-Pooling based solution). They setup an experiment using a sample Stock Ticker application and evaluate it's performance using a push based approach and a pull based approach. The main limitations of this work are the fact that the proposed experiment only compares two of the three techniques analyzed and that it does not cover the most recent server push technologies, such as Server-Sent Events and WebSockets.

In [15] the authors discuss HTTP Polling, HTTP Long Polling, HTTP Streaming and WebSockets. They setup an experiment using an instant message generator, web servers supporting each server push solution discussed and web applications running within a browser that will receive the messages. They evaluate the performance of each solution with respect to transmission delay and number of requests. The main limitation of this work is the fact that it neither discusses or evaluates the Server-Sent Events technique.

III. Server Push Technologies Analysis

This section describes existing server push technologies, presenting their weaknesses and strengths.

A. HTTP Polling

In HTTP Polling, the client sends a HTTP request to the server periodically to check its readiness or current state. The waiting time between two requests is called Polling Interval. If there are any updates during the polling interval, the response will carry the updated information [15][11][12].

Figure 1 represents the process of requesting a resource using HTTP Polling. The resource is requested, and it is unavailable until the processing is finished. This diagram illustrates how HTTP Polling works. In the case of the request of resource `A`, the server returns it in the first attempt. In the case of the request of resource `B`, the resource is unavailable in the first attempt, and it is returned in the second one. In a real scenarios, the number of exchanged messages may be much higher, depending on the processing time to generate the requested resource.

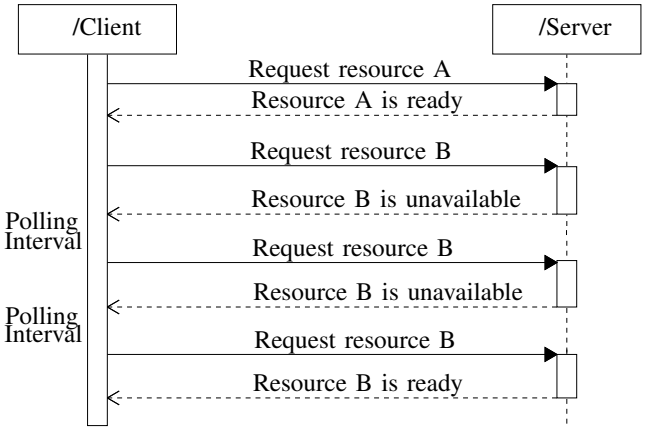

Fig. 1: Message exchange example using HTTP Polling.

The HTTP polling is the easiest technique to implement, since it requires a loop structure that contains four basic steps:

1) Send a HTTP request for processing the resource;
2) Wait/sleep for a predefined number of seconds;
3) Check the processing status;
4) Get the result iIf the process has finished, otherwise go to step (2).

However, this is the solution that implies the highest overhead on the client, the server and network infrastructure. The client may request several resources processing by using asynchronous HTTP APIs, and, if that HTTP Polling is use, the client would send checking processing status calls every polling interval, overloading the server and network with unnecessary requests. The client is overloaded too, since the more HTTP requests the client has to make the more hardware resources it consumes.

Strengths:

- There is no specific technology for polling.

Weaknesses:

- Generates processing overhead on client, server, and network due to polling (*i.e.*, it sends request to check

server processing status.), which requires high message exchanges when requesting a long time processing resource.

B. HTTP Long Polling

HTTP Long Polling attempts to minimize both the latency in server-client message exchange and the use of processing/network resources [5]. When a request is received, the server keeps the request open until the requested resource is available or connection timeout is reached. If the requested resource is available, the server sends a response containing it. If connection timeout is reached, the server sends an empty response [15][11].

Figure 2 illustrates the HTTP Long Polling request where the Client requests a resource `A` from the Server. The Server process the request and responds, in the first attempt. Afterwards, the Client requests resource `B` which takes more time to be processed than resource `A`. The connection timeout is reached during the process, and the Server sends a response message that does not contain the requested resource, but the current processing status. After while (Polling Interval), the client requests the resource B again. In a real scenario, a resource may take much more time to be processed and a high number of requests would be needed. However, that number of requests would be much higher if traditional HTTP Polling is used.

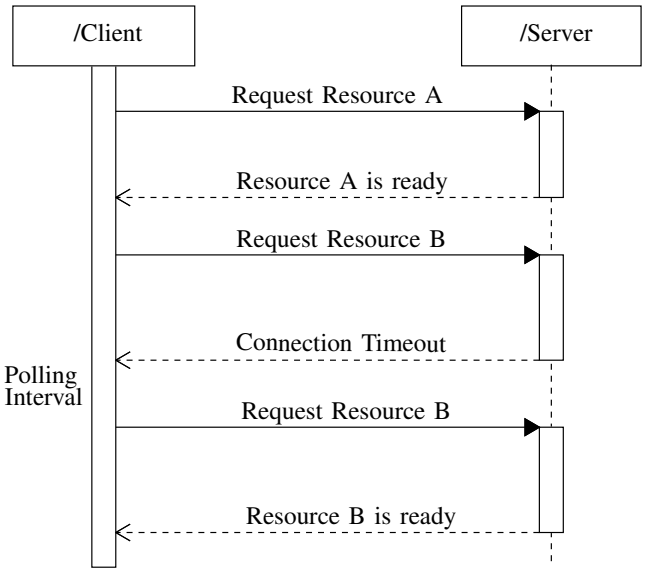

Fig. 2: Message exchange example using HTTP Long Polling.

Compared to polling technique, the use of long polling may reduce the number of requests. However, it does not provide the optimal solution, since the client would still be responsible to periodically request the server to inform the processing progress.

Strengths: Less messages exchanges compared to HTTP Polling, reducing client, server and network overhead related to simple polling.

Weaknesses:

- Requires specific server configurations and still demands client to send status checking requests (*i.e.*, polling).
- HTTP Long Polling will always occupy a TCP connection in a period of time, which is called HTTP Persistent Connection, trying to process the request during this time or timeout is reached. So, the number of HTTP connections between client and server are limited [17]. If the maximum number of HTTP connections is reached, incoming requests should be pulled in a queue.
- Long Polling may produce server side technical limitations if connections have to be kept alive for long period of time. In that case, the server has to be capable of handling such large number of simultaneous connections [15][11][5].

C. WebSockets

WebSocket was originally created by the Web Hypertext Application Technology Working Group and included within the original HTML5 specification. The specification was submitted to the Internet Engineering Task Force in December 2011 for further development [14].

The WebSocket Protocol enables two-way communication between a client and a server through a pure TCP connection. The protocol consists of an opening handshake through HTTP, followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for two-way communication with servers that does not rely on opening multiple HTTP connections [18].

Figure 3 illustrates how WebSocket protocol works. First, the Client sends an HTTP handshake request to establish a WebSocket connection with the Server. The Server responds with a HTTP 101 Status, and a full-duplex TCP connection (WebSocket connection) is established between the Client and the Server. Afterwards, the Client sends a processing request. This request is transmitted to the Server through a TCP datagram initiating the resource generation process. Due to the WebSocket connection, the Server is able to update the processing progress to the Client, and return the resource when it finishes without the need of extra requests.

Although in the diagram of Figure 3 we consider an initial request from the client to initiate the server processing, the server is able to send messages to the client without the need an initial request. In our scenario, however, the initial request from the client could contain the parameters for a machine learning experiment, for example, therefore the server would not be able to initiate the experiment execution without this initial information.

Provided that, the WebSocket technology provides very high scalability, since even if the resource takes a very long time to be processed, the Server is able to return result when it is ready, without having to respond polling requests while processing it. In some cases, this might even speed the resource processing since less server's hardware is consumed to respond

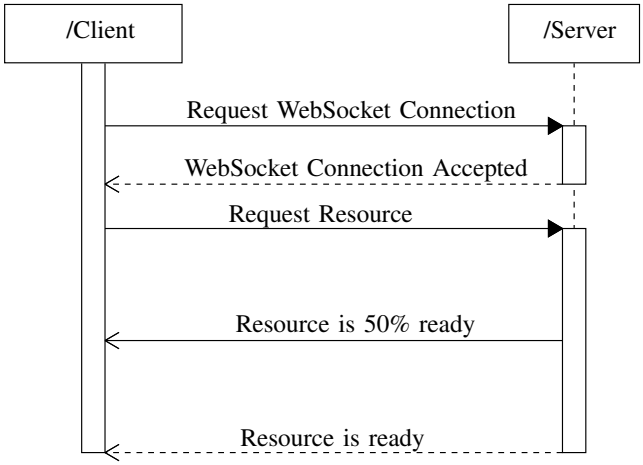

Fig. 3: Message exchange example using a WebSocket.

status requests, then more hardware can be allocated to process the requested resource.

Hence, the WebSocket protocol allows the Client to send one request for a resource, and the Server to send multiple responses informing the processing progress until the resource is available, when the resource is returned to the Client.

Strengths:

- By using WebSockets, the server is able to send data directly to he client without the constant polling from the client for status updates.
- Through its permanently open connection, it is possible to eliminate the client, server, and network overhead, allowing the connection to remain idle until the client or server sends a request. The server is able to send data continuously as the processing progresses, and remains idle when needed [13].

Weaknesses:

- For large-scale applications, it might be necessary to implement TCP loading balancing instead of HTTP load balancing [19]. So, the application server has to support this kind of load balancing, and it has to predict/monitor the overload of ports caused by the use of Websockets.
- It keeps a persistent connection for each request. It may be an open door for Denial of Service attacks [20] which is a big concern in the WebSockets technology because: (1) The client-server connection is exclusive; (2) The connection is closed explicitly by the client or by the server (*i.e.*, there is no timeout); (3) A client may open several private connection and keep them open for a long period of time.

D. Server-Sent Events

W3C defines Server-Sent Events (SSE) as an API for opening an HTTP connection for receiving push notifications from a server in the form of DOM event[6] [21]. In opposite to WebSockets bidirectional capabilities, SSEs are focused on delivering one-way real-time communication to the client [14]. Server-Sent Events use a persistent connection between a client and a server, which is initiated using HTTP. By operating over standard HTTP connections, a server support becomes widespread and issues such as port conflicts can be avoided.

SSEs include a new HTML element called EventSource, which is the client-side object used to receive events from the server [21]. Additionally, a new Multipurpose Internet Mail Extension (MIME) type called text/event-stream is used to define an event framing format. A publish-subscribe model is the used paradigm: clients subscribe to a channel and receive streaming updates from the server. Updates for this event-stream are pushed to the client at real-time as they occur, without the client having to send any requests. Since an event-stream may have many clients subscribed to it, the server is able to push messages to a large amount of clients and in a scalable fashion [21].

Figure 4 presents an example where a Client requests the Server to process a resource. The Client subscribes an EventSource to receive updates from an EventOutput from the Server. The Server is able to send processing progress updates and the requested resource when the processing is finished without the need of additional requests from the Client.

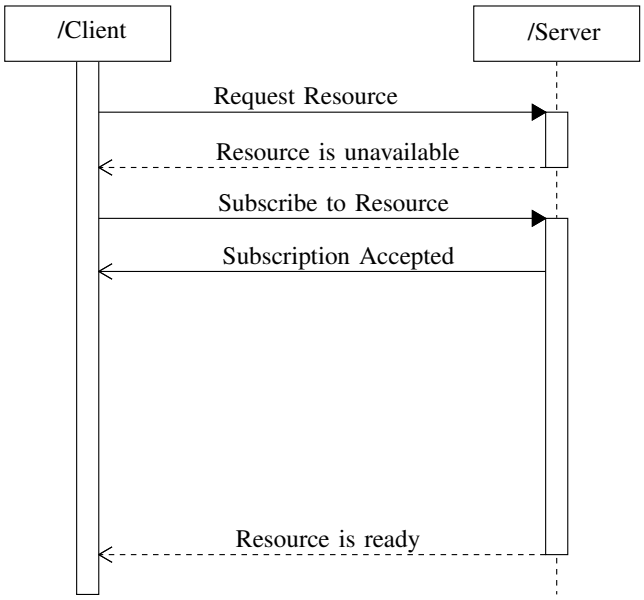

Fig. 4: Message exchange example using a Server Sent Event.

The use of Server-Sent Events requires the Client to send a request to start and register an EventSource to follow the processing progress. After the processing is finished, the requested resource could be returned through the same EventSource as a DOM element. This technology would require an extra request if compared to WebSockets, but it would allow better scaling if the resource being generated could be requested by multiple users at the same time. In that case, through SSEs, the resource would be returned in a broadcast fashion, while by using

[6]HTML DOM events allow JavaScript to register different event handlers on elements in an HTML document.

WebSockets the server would handve to return the resource for each user through their exclusive WebSocket connection.

Strengths:

- SSE is based on regular HTTP, and it is text-based, making it easy to create and parse events [12].
- SSE removes the overhead on the client, web server, and network infrastructure caused by the polling based techniques.

Weaknesses:

- Since SSE is based on HTTP Streaming, it inherits the good and bad aspects of this technique. One of them is the fact that HTTP Streaming does not allow intermediaries, such as proxies, transparent proxies, and gateways to forward and buffer HTTP responses to the client, in contrast with regular HTTP messaging [5].
- JavaScript and/or DOM elements that store SSE responses within the client might grow in size for every message received. Thus, in order to avoid unlimited growth of memory usage in the client, an HTTP streaming implementation occasionally needs to terminate the streaming response and send a request to initiate a new streaming response, what might increase the number of messages exchanged in comparison with a WebSocket solution [5].
- Using HTTP streaming, several application messages can be sent within a single HTTP response. The separation of the response stream into application messages needs to be performed at the application level and not at the HTTP level [5].

E. HTTP/2 Server Push

Hypertext Transfer Protocol Version 2 (HTTP/2) introduced several innovations to solve some of HTTP/1.1 performance problems. One of them is the server push mechanism that enables the server to preemptively send responses to a client in association with a previous client-initiated request. It is useful when the server knows the client needs those data available in order to fully process the response of the original request [22]. The main use case that motivated the inclusion of this mechanism in HTTP/2 was the overhead caused by the multiple GET requests sent during web page's initial load in order to obtain all the resources that were necessary for it to render properly, *e.g.*, stylesheets (CSS) and images. With HTTP/2 server push, the client browser would be able to send a single request for a web page, describing all resources related to it on its header. In response, the server sends all requested resources on multiple response messages.

Figure 5 illustrates the request of three resources through HTTP/2 server push technique. This technology was designed to improve web pages loading time by providing a mechanism for the Web Browser to request a resource `A` (*e.g.*, an HTML page) and required related resources `B` and `C` (*e.g.*, required Javascript and CSS files). The Server receives that single request and in response returns all three resources.

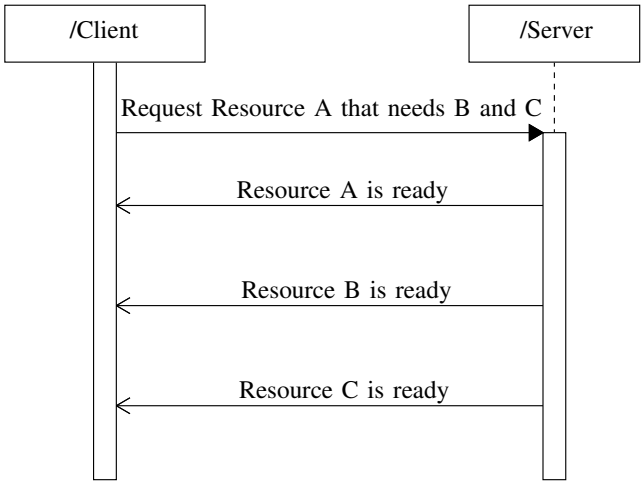

Fig. 5: Message exchange example using HTTP/2 Server Push.

Strengths:

- It removes client, server, and network overhead caused by polling techniques.
- It does not creates a persistent connection for each request.

Weaknesses:

- It requires server side support for HTTP/2.

F. WebPush

WebPush [16] is a protocol whose main objective is to extend the Push API [23] capabilities to the application layer protocol level, allowing user agents (*e.g.*, web browsers) to subscribe to a Push Service that receives messages from the application server and pushes them into the user agents. It is based on the HTTP/2 server push mechanism, since the Push Service subscription is made through a HTTP GET request that remains open until the user agent closes the connection. Messages sent by the application server are then retrieved as HTTP responses to that same request using a HTTP persistent connection.

Strengths:

- It allows the client to submit a single request and receive the requested resource when it is ready through the Push Service.
- It reduces the number of messages exchanged in comparison with HTTP Polling and works over HTTP.

Weaknesses:

- It requires server side support for HTTP/2 and demands effective management of the Publish Service TCP connections, since the number of open connections is directly proportional to the number of client subscriptions, which might be larger than the maximum number of ports available in the server [17].

TABLE I: Comparison of existing technologies.

Name	Hand-shake	Persistent connection	Polling-based	Event-based	Messages exchanged	Load balancing protocol
HTTP Polling	△ No	△No	▽Yes	▽No	▽High	△HTTP
HTTP Long Polling	△ No	▽Yes	▽Yes	▽No	▽Medium	△HTTP
WebSockets	▽ Yes	▽Yes	△No	△Yes	△Low	▽TCP
Server-Sent Events	▽ Yes	▽Yes	△No	△Yes	△Low	△HTTP
HTTP/2 Server Push	△ No	△No	△No	▽No	△Low	△HTTP
WebPush	▽ Yes	▽Yes	△No	△Yes	△Low	△HTTP

TABLE II: Web server support for each technology.
*Support for HTTP/2 available on a separate solution called Grizzly.
**Native support under development, third-parties are needed until it's released.

Name	Apache 2.4	Tomcat 9.0	Glassfish 5.0	Websphere 8.5	JBoss 3.0	Nginx 1.13	Tornado 4.5	Jetty 9.4	Node.js 9.2
HTTP Polling	Full	Full	Full	Full	Full	Full	Full	Full	Full
HTTP Long Polling	Full	Full	Full	Full	Full	Full	Full	Full	Full
WebSockets	Full	Full	Full	Full	Full	Full	Full	Full	Full
Server-Sent Events	Full	Full	Full	Full	Full	Full	Full	Full	Full
HTTP/2 Server Push	Full	Full	None*	None	Full	Full	None	Full	Full**
WebPush	Full	Full	None*	None	Full	Full	None	Full	Full**

TABLE III: Web browser support for each technology.

Name	Internet Explorer 11	Microsoft Edge 17	Mozilla Firefox 60	Google Chrome 65	Safari 11	Opera 50	iOS Safari 11	Opera Mini	Android Browser 62	Chrome for Android 62
HTTP Polling	Full	Full	Full	Full	Full	Full	Full	None	Full	Full
HTTP Long Polling	Full	Full	Full	Full	Full	Full	Full	None	Full	Full
WebSockets	Full	Full	Full	Full	Full	Full	Full	None	Full	Full
Server-Sent Events	None	None	Full	Full	Full	Full	Full	None	Full	Full
HTTP/2 Server Push	Partial	Full	Full	Full	Full	Partial	Full	None	Full	Full
WebPush	None	None	Full	Full	None	Full	None	None	None	Full

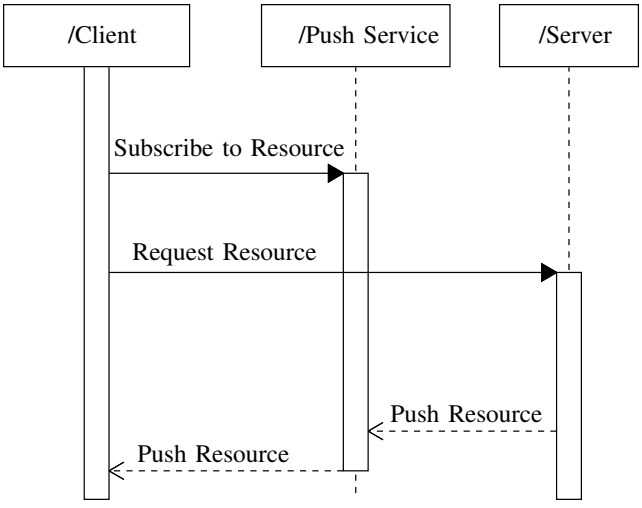

Fig. 6: Message exchange example using WebPush.

IV. Technologies evaluation

This section presents an analysis of the server-push techniques (Section III) based on a set of criteria which were defined from the techniques characteristics. These criteria were based on the main characteristics that differentiate each technology, such as existence of hand-shake requests, persistent connections, need of client polling and possibility of a client registering a callback to a request. Also, attributes that might influence deployment and performance management of applications that use each technology were included, such as the relative amount of messages exchanged and the protocol that should be considered by load balancing mechanisms. The selected criteria are listed bellow:

- **Hand-shake**: indicates if the client needs to establish a connection with the server through a hand-shake request;
- **Persistent connection**: indicates if the server keeps the connection open until the requested resource is available;
- **Polling-based:** indicates if the client is responsible to send requests to the server periodically to check processing status.

- **Event-based**: indicates if the client registers a callback to receive the processing response.
- **Messages exchanged**: indicates if the number of messages exchanged between client and server are low, medium or high.
- **Load balancing protocol**: indicates if the load balancing system employed by the server must be based on HTTP or TCP load balancing.

Table I presents the values of each criteria for each technology based on the references cited for each technology. $\triangle$ indicates strength of the technology, while $\triangledown$ indicates weakness.

Table II presents the servers and their support status for each technology. The table can be used as a guide for researchers and developers that want to use any of those technologies. We list the current support status for the most relevant web and application servers available for Java and Python according to [24] and [25], with the exception of NodeJS that has been included due to its growing usage and high support to most recent technologies [26].

The list of web browsers that support each technology is presented on Table III based on [27]. This is the main and most up to date browser support reference for web development.

V. Simulation

In this section, we present simulations to support some of the theoretical claims made throught this paper. Simulations were conducted using Simpy, a discrete-event simulation framework [28]. Simpy allows us to simulate a high number of long events (in simulated time) in a short period of (real) time. Events are anything that have an associated execution time and that can take an arbitrary ammount of time until conclusion. Long time running tasks can be directly modeled using the event paradigm.

Before simulations begins, each task receives a start time (scheduled start time). Tasks are scheduled following a Poission process [29] with varying task creation frequency λ. A Poisson process guarantees that the average number of occurences within an interval approaches λ. Figure 7 shows the distribution of tasks by time and λ; by varying λ we vary the number of tasks within the system. The goal of this simulation is to analyse how an increasing number of tasks impacts the system using each of the discussed strategies.

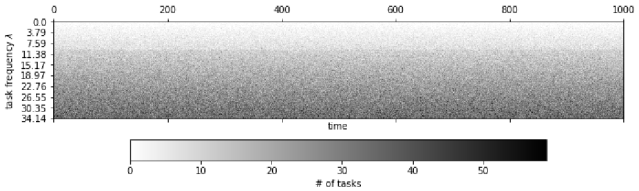

Fig. 7: Distribution of Long time running tasks.

The evaluate the strategies we use two different metrics: (i) increase in end time; and, (ii) number of client/server exchanged messages. For each task, its minimum end time is know (taks's scheduled start time plus task's duration). A task's (T_i) increase in end time is given by its simulated end time ($end(T_i)$) over its minimum end time ($expected\_end(T_i)$). Therefore:

Definition V.1. Average increase in tasks end time: Let T be a set of tasks spawend in instant t, then the increase in end time is given by:

$$\frac{\sum_{i=1}^{|T|} \frac{end(T_i)}{expected\_end(T_i)}}{|T|}$$

A. Simulation Configuration

For all simulations we stablished a fixed time interval of 1000 cycles for tasks' start times, i.e., only tasks spawned within this time interval were considered. Task creation frequency λ was varied 170 times in the interval $[0.0001, 34.15]$. All created tasks were identical, they took 600 cycles to conclude. We also modeled the server, identical in all simulation executions: 1) it accepted at most 1000 concurrent connections[7]; 2) opened HTTP connections timeout after 60 cycles.

To monitor tasks execution we modeled three different strategies: (i) **Polling**: Section III-A, a polling request takes 2 cycles to respond, the polling ends if the task finished, otherwise a new polling is scheduled 10 cycles ahead; (ii) **Long Polling**: Section III-B, the request takes 60 cycles – equal to server's connection timeout, long polling can return a response before 60 cycles if the task finishes, otherwise a new long polling is scheduled 10 cycles ahead; (iii) **Duplex connection**: a connection between server and client remains open during the whole task execution - sections III-C, III-D, III-E, and III-F.[8]

B. Simulation Results

For each monitoring strategy we followed the task distribution presented in Figure 7. These simulations are not concerned with where the long running tasks are processed. We are assuming unlimited computational power. Therefore, the only expected bottleneck is the maximum number of simultaneous connections the server can support, which is the main variable we want to analyze in our simulations.

1) Evaluation metrics: To evaluate the three methods we used two different metrics: (i) increase in end time (Definition V.1); and, (ii) number of messages exchanged between client and server.

Figure 8 presents the increase in end time of tasks executed through a Duplex Connection method. As expected, as the volume of tasks increases so does their end times (Figure 11), going as high as a 175% increase in end time. The distinct

[7]Duplex connections and HTTP requests have the same cost, i.e., the modeled server can hold 1000 concurrent duplex connections or serve 1000 simultaneous requests.

[8]This scenario was modeled by setting the Server's Connection Timeout to be as large as possible and using a Long Polling strategy, *e.g.*, given the collected metrics a Long Polling with infinite timeout is equivalent to a Duplex Connection.

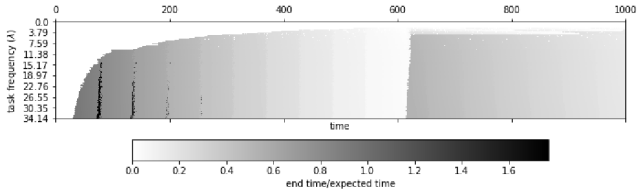

Fig. 8: Duplex increase in end time per time interval and generation frequency.

step, around time 600, represents the end of the first batch of tasks, which coincides with a release of connections.

End times increases for polling (Figure 9) and long polling (Figure 10) are only slightly affected by the increase in frequency, by at most 2.5% and 8%. This result is also expected, since server resources are occupied by only a short period of time. So, the server utilization rate is lower when doing polling and long polling. Another important point is that although both polling methods generate a large amount of unnecessary requests, if any of them receive a timeout the task end time is not affected, it can only be affected by timeouts in the first or last requests, which becomes unlikely because each request can be responded to in a fraction of the timeout.

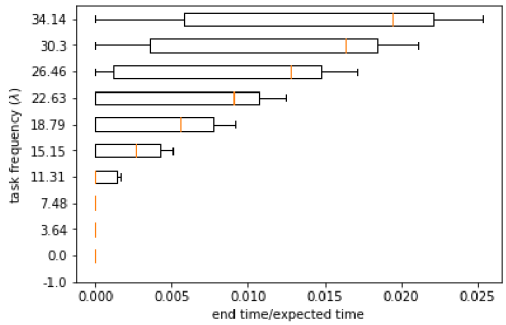

Fig. 9: Polling increase in task duration per time interval and generation frequency.

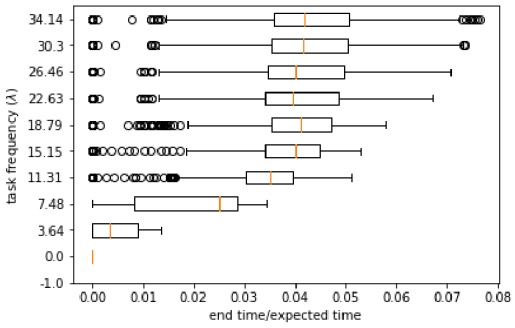

Fig. 10: Long Polling increase in task duration per time interval and generation frequency.

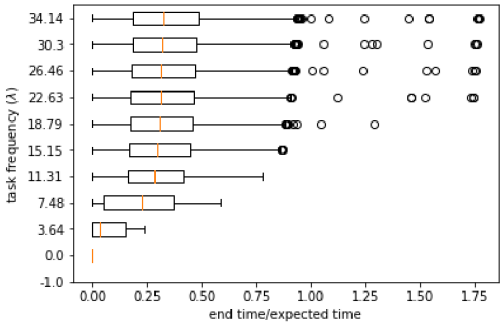

Fig. 11: Duplex increase in task duration per time interval and generation frequency.

In the context of this simulation, a message is any client's request to the server. This would include the beginning POST request for the task and all subsequent poll requests. Figures 12, 13, and 14 present the total number of messages exchanged by tasks started in a given instant t (X axis) by the increase in task creation frequency λ (Y axis). As is expected, there is a positive correlation between the frequency of task creation and the number of exchanged messages.

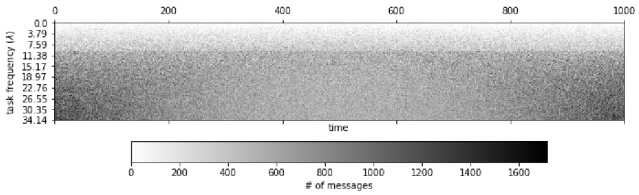

Fig. 12: Polling: Number of message per start task time

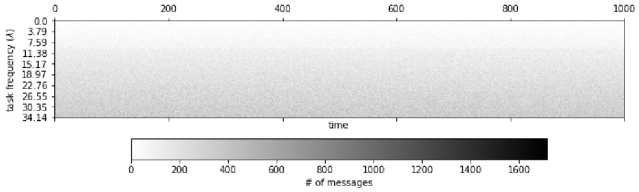

Fig. 13: Long Polling: Number of message per start task time

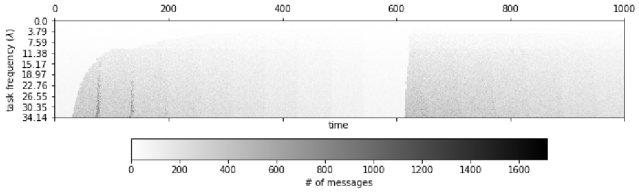

Fig. 14: Duplex: Number of message per start task time

Polling is by far the method that exchanges the higher number of messages (Figure 18), followed by Long Polling (Figure 19), and, finally, Duplex Connection (Figure 20).

Long Polling (Figure 16) and Duplex Connection (Figure 17) number of messages per task behave as expected. For Long Polling, the number of messages is stable throughout. For Duplex, the number of messages, in this case they represent retries, is positivly correlated with task frequency. However, Polling exhibits an interesting behavior (Figure 15), after a certain frequency the number of messages exchanged by task decreases and approaches a value. This occurs because the client is only free to send a new poll request after the previous one receives a response, which can take up to Server's timeout to occur, the decrease in messages indicates that server utilization is increasing, *i.e.*, that poll messages response times are approaching the timeout.

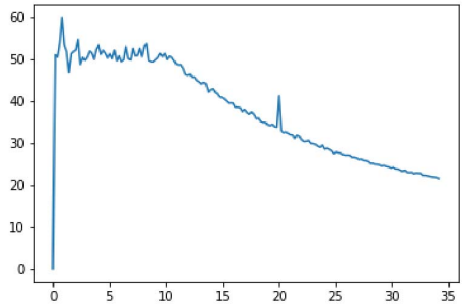

Fig. 15: Polling: Number of messages per task per frequency

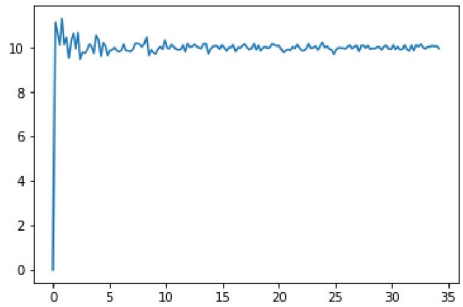

Fig. 16: Long Polling: Number of messages per task per frequency

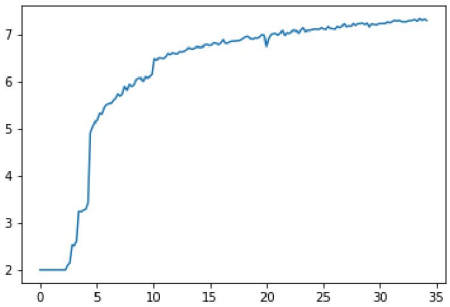

Fig. 17: Duplex: Number of messages per task per frequency

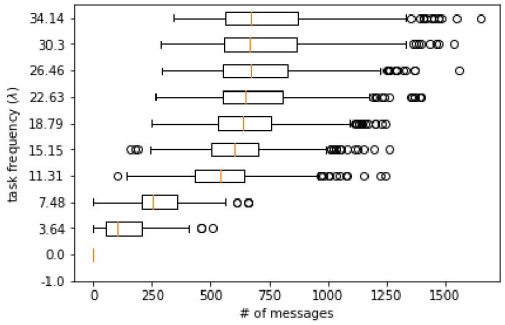

Fig. 18: Polling: Number of messages per task per frequency

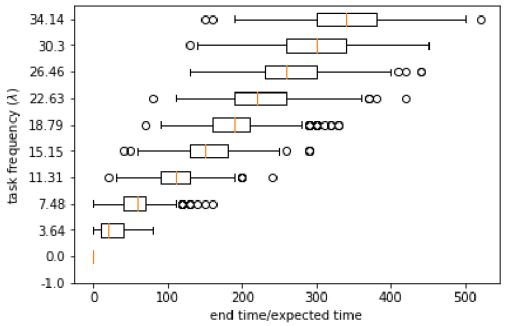

Fig. 19: Long Polling: Number of messages per task per frequency

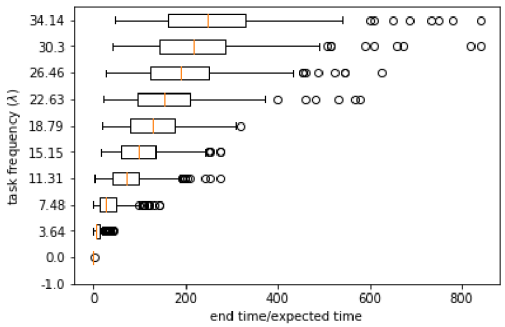

Fig. 20: Duplex: Number of messages per task per frequency

These simulation results indicates methods that release resources from the server are better when considering the time dimension but worse in network usage. Having fast transactions means the server is only occupied in serving a given task for a small fraction of its running time. As discussed previously, this is only true if the computational power available for the task execution exceeds the server's connection limits. In cases where such configuration is not possible, the increase in end time for all three simulated methods would be bounded by the computational power; making polling and long polling a worse option than duplex methods.

VI. Conclusion

In this work, we analyzed server push alternatives to a special case of client-server communication: a client requests a resource that takes a large amount of time to be processed by the server (*e.g.*, a simulation or data mining experiment result). This scenario requires that servers be able to push messages to the clients in order to keep them informed about the processing status and return the requested resource when it is ready. Technologies that provide such functionality are named *server push technologies*.

This work characterized, classified, and compared existing server push technologies and simulated their use. From our analysis we concluded that the most prominent technologies are *WebSockets*, *Server-Sent Events* and *WebPush*. They provide the highest efficiency in terms of client, network, and server overhead. HTTP/2 Server Push is very efficient in reducing client, network, and server overhead as well. However, it does not seem very useful for the long time client-server interaction problem, since it was designed to reduce the number of requests needed to get multiple resources that are dependent on each other, *e.g.*, CSS and Javascript files associated with HTML pages. HTTP Long Polling is a viable option if the expected processing time is not very high. However, it does not scale as well as WebSockets, Server-Sent Events, and WebPush.

Our results can be used by developers and researchers whose works can benefit from server push techonologies.

As future work, we plan to reproduce the simulated scenarios in a real environment, *i.e.*, use functionalities provided by a server that requires long time processing, and using each server push technology for the client-server communication. Hence, we can collect number of exchanged messages, I/O operations, CPU usage etc. We also plan to provide code samples that illustrate how to use these technologies, both in the client-side and, more specifically, in the server-side. From those studies, we will be able to provide guidelines for the usage of each technology.

References

[1] J. F. Kurose, *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[2] D. Gourley and B. Totty, *HTTP: the definitive guide*. OReilly Media, Inc., 2002.

[3] X. Qu, J. Xu Yu, R. P. Brent *et al.*, "A mobile tcp socket," 1997.

[4] B. P. Gerkey, R. T. Vaughan, K. Stoy, A. Howard, G. S. Sukhatme, and M. J. Mataric, "Most valuable player: A robot device server for distributed control," in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 3. IEEE, 2001, pp. 1226–1231.

[5] G. Wilkins, S. Salsano, S. Loreto, and P. Saint-Andre, "Known issues and best practices for the use of long polling and streaming in bidirectional http," *Request form Comments: 6202*, 2011.

[6] S. S. Rao, H. M. Vin, and A. Tarafdar, "Comparative evaluation of server-push and client-pull architectures for multimedia servers," *Proceedings of NOSSDAV96*, pp. 45–48, 1996.

[7] V. Kanitkar and A. Delis, "Real-time client-server push strategies: Specification and evaluation," in *Real-time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*. IEEE, 1998, pp. 179–188.

[8] M. Pohja, "Server push for web applications via instant messaging," *Journal of Web Engineering*, vol. 9, no. 3, pp. 227–242, 2010.

[9] ——, "Server push with instant messaging," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 653–658.

[10] E. Bozdag, A. Mesbah, and A. Van Deursen, "A comparison of push and pull techniques for ajax," in *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*. IEEE, 2007, pp. 15–22.

[11] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.

[12] S. Vinoski, "Server-sent events with yaws," *IEEE Internet Computing*, vol. 16, no. 5, pp. 98–102, 2012.

[13] A. Wessels, M. Purvis, J. Jackson, and S. Rahman, "Remote data visualization through websockets," in *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*. IEEE, 2011, pp. 1050–1051.

[14] E. Estep, "Mobile html5: Efficiency and performance of websockets and server-sent events," 2013.

[15] K. Shuang and F. Kai, "Research on server push methods in web browser based instant messaging applications." *JSW*, vol. 8, no. 10, pp. 2644–2651, 2013.

[16] M. Thomson, E. Damaggio, and B. Raymor, "Generic event delivery using http push," Tech. Rep., 2016.

[17] S. Sounders, "Roundup on parallel connections," *High Performance Web Sites blog, March*, 2008.

[18] I. Fette, "The websocket protocol," *Request for Comments: 6455*, 2011.

[19] K. Shamko, "Load balancing of websocket connections," https://www.oxagile.com/company/blog/load-balancing-of-websocket-connections/, 2016, accessed in 09/2017.

[20] S. Saffron, "Websockets, caution required!" https://samsaffron.com/archive/2015/12/29/websockets-caution-required, 2016, accessed in 09/2017.

[21] I. Hickson, "Server-sent events," *W3C Working Draft WD-eventsource-20091222, latest version available at¡ http://www.w3. org/TR/eventsource*, 2009.

[22] M. Belshe, R. Peon, and M. Thomson, "Rfc 7540: hypertext transfer protocol version 2 (http/2)," *Internet Engineering Task Force (IETF)//BitGo, Google Inc.//May*, 2015.

[23] A. Push, "W3c editors draft 19 jan 2015," *Available: w3c. github. io/push-api*.

[24] J. Koskelainen, "Web servers and python," https://wiki.python.org/moin/WebServers, 2016, accessed in 09/2017.

[25] Java-Source.net, "Open source web servers in java," https://java-source.net/open-source/web-servers, 2017, accessed in 09/2017.

[26] NodeJS, "About nodejs," https://nodejs.org/en/, 2017, accessed in 09/2017.

[27] A. Deveria, "Can i use... support tables for html5, css3, etc," http://caniuse.com/, 2017, accessed in 12/2017.

[28] N. Matloff, "Introduction to discrete-event simulation and the simpy language," *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, vol. 2, p. 2009, 2008.

[29] C. Gardiner, "Stochastic methods," *Springer Series in Synergetics (Springer-Verlag, Berlin, 2009)*, 1985.