MUST : Mutable State Transfer

Kévin Rauscher, Sylvain Cherrier, Thomas Pape, Yacine Ghamri-Doudane

MUST : Mutable State Transfer

Kévin Rauscher*, Sylvain Cherrier*, Thomas Pape*, Yacine M. Ghamri-Doudane†
* *Université Paris-Est , Laboratoire d'Informatique Gaspard Monge (CNRS : UMR8049)*
† *L3i Lab, University of La Rochelle, La Rochelle, France.*

***Abstract*—In new internet applications, up-to-date information is vital. Users ask for a constant flow of information. New technologies offer a myriad of data sources but these information rapidly become obsolete. Social networks and Internet of Thing, whose importance never stopped growing in the past decade, are excellent examples of applications where up-to-date data is both key and volatile. Sensors can change values in a mater of seconds. With the WebSocket protocol, W3C is offering a solution for real-time applications for developers. However the protocol itself relies on the use of sub-protocols to function, most of which are message oriented. MUST is a full-duplex and object or state oriented sub-protocol. Its similarities with HTTP makes it familiar and easy to use for web development. And it is aimed to be object-oriented architecture friendly.**

***Keywords*-Resource Oriented Protocols; Asynchronous Messaging; Full Duplex Communication**

I. Introduction

With its spread, the Internet has evolved to be more and more dynamic and social thus creating an increasing amount of content alteration sources, be it user inputs, sensors values evolutions or any other connected media. Any response being sent over the internet representing an instant result of a time and situation related computation, its validity is ephemeral. This tends to make information more and more volatile. Data might be rendered obsolete only a few instants after being sent. In a system where instantaneous and up-to-date are keys, it begs the question of how to conciliate the ever-moving nature of information and user's or system's eagerness for up-to-date content.

Around 2007, what is now called the Web 2.0 by some, whether true revolution or pure marketing, have been created to tackle some of these new problems. These solutions aimed to handle some form of user interactivity, elevating interfaces from pure snapshot to more dynamic systems. However, these technologies mostly focus on a more local scope, allowing data to evolve between interactions with servers. Thus the question of data obsolescence remains so long as you consider it as coming from a more global perspective.

Data alterations considered in this article are not local, thus knowledge of these modifications is only available on the server side. HTTP based solutions meet difficulty when trying to solve this problem due to its client-server nature. As HTTP is unable to initiate a communication, between user requests, an information can be considered as being in an uncertain state. It may or may not be obsolete. A few solutions have been found to tackle with this issue such as Long-Polling or Server-Sent-Events. While they can solve the problem at hand, they also encounter other difficulties, like browser or framework support.

In this context, WebSocket [17] is a protocol created to bring a solution to the need of full-duplex standardized communication channels. However WebSocket is merely an applicative transport layer and, as other transport protocols, rely on the use of another protocol, standardized or not, to format the data transfer. On this day 17 SubProtocols have been registered like existing protocols adapted to WebSockets such as XMPP [13], MQTT [9] or SOAP [1].

While not adapted "as is" in full-duplex communications, HTTP is a foundation of the modern Web. We believe that its massive adoption is due to its relative simplicity and its object or resource oriented paradigm. Thus, in our research for a full-duplex protocol adapted to classic and simple web interfaces usage, we fixed ourselves three constraints :

- Ease of use and implementation
- Resource oriented and RESTful [6]
- Full-Duplex

As our research for a protocol meeting these prerequisites failed, we decided to propose a protocol which would follow these guidelines as much as possible.

Our objective is to provide a protocol aimed to access, modify and listen for changes in representational states. MUST is a protocol heavily inspired by HTTP structure. In fact, MUST can be considered as an asynchronous and extended HTTP with JSON formatted messages. MUST has been designed to be used as a tool for pretty usual web application with relatively volatile data. Thus, large data transfer is out of our scope, so are constrained network and platforms. With this in mind, we designed it to be easy to implement an familiar to most developers thanks to it's HTTP lineage.

This paper is organised as follow: Section II presents related works and some background for our solution. Section III describes the constraints and advantages an asynchronous architecture provides. Section IV explains how the server-side notifications are handled. Finally, concluding remarks end this paper in Section V.

II. Related Work and Background

WebSocket (RFC 6455[5]) is a solution that provides bi-directional communications for WebSites. As the inter-

actions increase between the client and the server, the requesting approach used by HTTP suffers the lack of reactivity. AJAX (Asynchronous Javascript and XML) [7] was introduced in order to solve that issue. But the bidirectional communications provided by Websockets offer a better solution. Herwing et al. [8] show that Websockets leverage the network usage, providing reactivity and a smaller traffic. For example, this decrease can benefit WSAN (*Wireless Sensors and Actuators Networks*) because of their energy constraint.

The problem is that Websockets are only a technical solution for implementing bidirectional communications, but does not fulfill the application needs. Doukas et al. propose COMPOSE [4], a Mobile Software Development Kit that aims to link all kinds of technical solutions such as MQTT [9] and Websocket. Betz et al. have described a gateway [3] that translates REST to Websockets.

In the field of the Internet of Things, according to Jung et al. [10], CoAP [15] and MQTT are privileged in order to fit with the constraints of WSAN. The main issue with Websocket in this area is the lack of a complete implementation.

As said above, Websocket offers a bidirectional communication, but is not an application protocol. To embed them, Websocket needs the definition of a sub-protocol [17] The content of each application message has to be described in a sub-protocol of Websockets to be supported. Some of the well-known application protocols are already or nearly standardized: XMPP [16], or (*draft*) CoAP [14].

Karagiannis et al. have shown in their survey [11] the interest for the CoAP protocol in the IoT because of its mimic of the well spread REST protocol [6]. They also recommend MQTT because of its lightness.

But MQTT is a protocol that aims to exchange messages. Thus, each client has to implement the logic corresponding to the semantic associated with the received message. In the message approach, there is an imbalance of processing between both participants. The transmitter is in charge of sending the data. The receiver receives them, and has to analyze them. In a REST approach, the server sends a representation of the resource (for the web) or object (in the case of IoT) in its current state. It hides the data processing, which can be the result of multiple interactions, with multiple stakeholders. This introduces a unique point of processing in the global interaction, and a common point of view for each client.

Kovatsch et al. present a tool that represents each object as their states, using for that purpose both REST and CoAP [12]. By following the reasoning described by Herwing et al. [8], to offer one sub-protocol allying representation of a resource and assets of websockets (*bidirectional, keep alive*) allows to relieve the exchanges while respecting the well installed protocols.

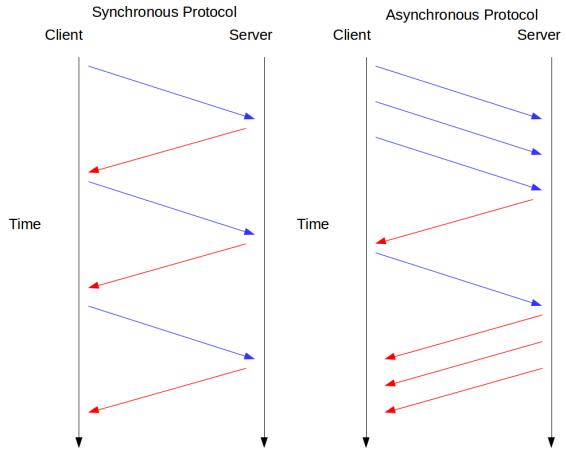

Figure 1. While synchronous protocols can associate request and response by using order as a reference, asynchronous protocols are inherently unable to do so this way.

III. Asynchronism in MUST

A. Synchronous Architectures

A vast majority of communications in Web Applications are handled by synchronous protocols like HTTP or SOAP. They are efficient ways of transferring data in simple request-response architecture and also adapt to many software architectures. However, troubles emerge when data's lifespan shortens. In many fields, up-to-date information plays a major role.

With the emergence of new technologies, sources of data modifications have increased significantly. Users play a far more central role in web contents. With the internet of things, smart objects start to do too. A concrete example of highly volatile data could be sensors, where values can vary significantly in only a few seconds and thus make previously sent values obsolete very rapidly.

Synchronous protocols relies on the simple assumption that request and response are alternated. In the application layer, this often means that methods sending request are either blocking or using some sort of threaded tool such as Promises or Futures structures. The main advantage of this type of architecture is its simplicity. However, it also strongly enforces the client-request paradigm, typical in synchronous protocols.

Troubles arise when you load data at an either fixed or user controlled rate. In our Internet of Thing example, a sensor state could be modified only a few instants after a response. In a synchronous protocol, there is no way for the server to update it.

B. Asynchronous Architectures

While pretty efficient, synchronous structures have two major downsides. First the time consumed waiting for the response is unused network time which can become a serious

problem for either high latency networks or when requesting slow computations. However tools and work around exist to mitigate these effects. Second is the inability, by design, of the server to initiate a communication.

On the other hand, asynchronous protocols are able to send requests at any rate. However, this also requires the protocol to be able to associate response back to the original request. Thus, similarly to XMPP message structure or HTTP/2 [2] stream id, MUST requests include a unique numeric id. MUST server then simply rewrites the request ID in the response message to allow clients to regroup requests and responses.

While the overhead in network transfers created by our architecture is a downside, it is, firstly, a relatively small overhead compared to many applicative solutions like headers or cookies. Secondly MUST target non-constrained networks, which tend to even more mitigate the importance of this factor. Moreover, being able to send requests at any given rate grant the application layer with an increased flexibility on the use of it's network resources.

IV. Resource Oriented Architecture

A. Message oriented protocols

While related to the HTTP client-server architecture, WebSockets offer a full duplex communications. This kind of infrastructure is often associated with messaging protocols where there is no hierarchy between participants. Thus, this is no surprise if WebSockets SubProtocols list includes XMPP, MQTT and STOMP which are all message oriented. Message oriented protocols, as their name might indicate, are mostly focused on messages rather than on the meaning of their content. By focusing on messages, they offer a great flexibility of their usage as users are able to give message's payloads meaning humanly or programmatically.

While it is possible to create message oriented structures in Web applications, development required are significantly more complex and not necessarily easy to conciliate with a proper MVC architecture. In a complete message oriented Web application, the client-side code will directly receive messages. Then the message will be processed and a new state will be deduced from it directly inside the client-side code meaning that most of the business logic will have to be embedded in the client code. This is hardly applicable to standard use cases which are mostly focused on display.

Thus, in our opinion, message oriented protocols are hard to use in standard Web applications. They are too complex to set up and require far too much adaptation of existing code or coding patterns. Note that the most spread Web protocols, HTTP and SOAP for example, are resource oriented.

B. Resources Identification

The WebSocket protocol handshake might be interpreted by HTTP servers, but it is an independent TCP-based protocol. In its scope, connections are endpoints identified only by the couple IP and port. This is sufficient for data transport and, in facts, message based infrastructure do not necessarily require more to identify message's recipients. Resource oriented protocols rely on a higher granularity.

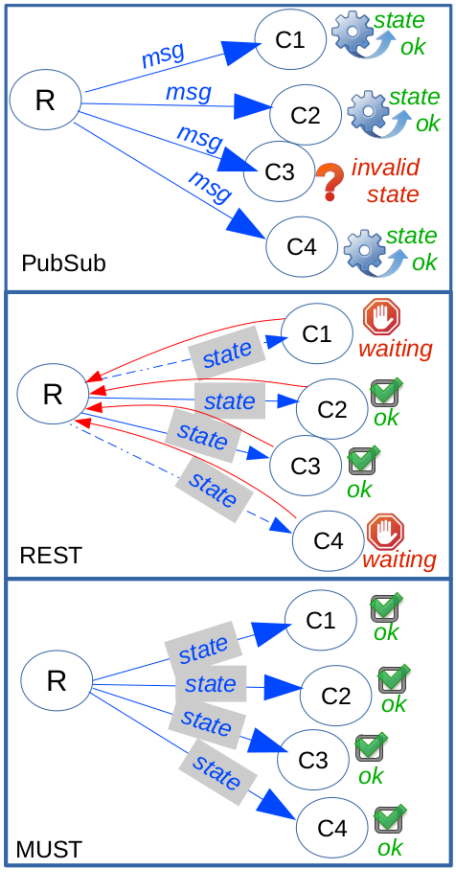

Figure 2. Comparison of PubSub, REST and MUST with 1 resource and 4 clients. PubSub mechanism sends asynchronous messages, and each client needs to rebuild its own version of the ressource. In our example, the client C3 is in a incoherent state. REST is a solution to send a valid representation of the resource state, but in a synchronous way. Clients need to query the resource, are blocked while waiting for the response, and are not aware of resource modification. MUST provides a non blocking mechanism that exchange a full representation of the resource state.

While a recipient can be determined only by the endpoint to which data should be sent in a message oriented architecture, the recipient will then have to use the message payload to determine the action it is supposed to do. Resource oriented architectures, and in particular RESTful ones, rely on a higher granularity for resource identifications. HTTP, for example, uses uniformed resource identifiers.

As MUST main goal is to be familiar, like in HTTP, it uses a single string as a unique name for its resources. While in the REST nomenclature, uniformed resource identifier structure is specified, MUST does not enforce the naming of resources.

C. Resource Manipulation

In message oriented architectures, the payload of a message will determine the action expected to be taken. But in resource oriented, operations have to be defined for each resource. MUST uses HTTP methods to define operation possible on a resource.

However MUST is aimed to be a bidirectional protocol. While HTTP methods are adapted to a request-response architecture, MUST is asynchronous and thus not bound to the same constraints. As the order in request and response is irrelevant, a single request can receive multiple responses. MUST extends HTTP methods with SUBSCRIBE and UNSUBSCRIBE. Subscribing to a resource is a way of keeping its state up-to-date. Server first replies with standard status codes and, after the initial request and response, server keeps replying to this specific request with the new 210 UPDATED status code.

This asymmetry in MUST allow clients to keep data up-to-date while bringing minimal constraints on server-side or client-side architecture. Most existing software code can completely use MUST without major modifications.

V. CONCLUSION

The wide-spread of HTTP as the common platform for the existing use of the Internet and its extension to new fields (i.e. the Internet of Things) lead to its constant adaptation to solve new issues. The dynamicity of the exchanges produced by new usages is addressed by the bi-directional communication offered by Websockets. A list of sub-protocols gives Websockets the ability to transport already existing application protocol. But these sub-protocols are mainly messages oriented. The need for a resource oriented solution (such as REST) is needed for Websockets.

This paper describes MUST (Mutable State Transfer). MUST is both an asynchronous and resource oriented WebSocket sub-protocol. Thus, it is tailored to represent volatile object states. Internet of Thing monitoring is an example of application for which it is aimed. Moreover, its resource oriented nature makes underlying concepts really easy to grasp for current and new web developers. Javascript and Play (Java) implementations of the protocol are available on GitHub.

REFERENCES

[1] Soap version 1.2 (w3c). http://www.w3.org/2002/07/soap-translation/soap12-part0.html.

[2] M. Belshe, M. Thomson, and R. Peon. Hypertext transfer protocol version 2 (http/2). 2015.

[3] T. Betz, L. Cabac, and M. Wester-Ebbinghaus. Gateway architecture for web-based agent services. In *Multiagent System Technologies*, pages 165–172. Springer, 2011.

[4] C. Doukas, L. Capra, F. Antonelli, E. Jaupaj, A. Tamilin, and I. Carreras. Providing generic support for iot and m2m for mobile devices. In *Computing & Communication Technologies-Research, Innovation, and Vision for the Future (RIVF), 2015 IEEE RIVF International Conference on*, pages 192–197. IEEE, 2015.

[5] I. Fette and A. Melnikov. The websocket protocol (no. rfc 6455). 2011.

[6] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2:115–150, May 2002.

[7] J. J. Garrett et al. Ajax: A new approach to web applications. 2005.

[8] V. Herwig, R. Fischer, and P. Braun. Assessment of rest and websocket in regards to their energy consumption for mobile applications. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015 IEEE 8th International Conference on*, volume 1, pages 342–347. IEEE, 2015.

[9] U. Hunkeler, H. Truong, and A. Stanford-Clark. Mqtt-s-a publish/subscribe protocol for wireless sensor networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798. IEEE.

[10] M. Jung, J. H. Kim, H. W. Wi, S. Kim, and M. Kovatsch. Things-to-cloud communication: technology overview and design considerations. 2015.

[11] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing*, 3(1):11–17, 2015.

[12] M. Kovatsch, Y. N. Hassan, and S. Mayer. Practical semantics for the internet of things: Physical states, device mashups, and open questions. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 54–61. IEEE, 2015.

[13] P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. 2011.

[14] T. Savolainen, K. Hartke, and B. Silverajan. Coap over websockets (ietf draft. 2015.

[15] Z. Shelby, B. Frank, and D. Sturek. Constrained application protocol (coap). *An online version is available at http://www.ietf.org/id/draft-ietf-core-coap-18.txt*, 2010.

[16] L. Stout, J. Moffitt, and E. Cestari. An extensible messaging and presence protocol (xmpp) subprotocol for websocket (no. rfc 7395). Technical report, 2014.

[17] V. Wang, F. Salim, and P. Moskovits. The websocket api. In *The Definitive Guide to HTML5 WebSocket*, pages 13–32. Springer, 2013.