Light-PubSubHubbub: A Lightweight Adaptation of the PubSubHubbub Protocol

Porfírio Dantas, Jorge Pereira, Everton Cavalcante, Gustavo Alves, Thais Batista
DIMAp – Department of Informatics and Applied Mathematics
UFRN – Federal University of Rio Grande do Norte
Natal, Brazil
{enghaw13, jorgepereirasb}@gmail.com, {evertonrsc, gustavo}@ppgsc.ufrn.br, thais@ufrnet.br

***Abstract*—The publish-subscribe communication paradigm is widely used in systems that require a loosely coupled asynchronous form of interaction. The PubSubHubbub protocol is a publish-subscribe protocol for the Web that involves publishers, subscribers, and hubs, which are the intermediate elements between publishers and subscribers. However, in the original implementation of the protocol, unnecessary computation and network traffic occur as the sequence of exchanged messages to subscribers to retrieve a message is not optimized. In this paper, we present a lightweight version of such a protocol, named Light-PubSubHubbub, by introducing the following changes to the communication process: (i) the publisher no longer needs to publish updated messages in a Web topic and then notify the hub since the messages are published in the hub itself; (ii) it uses the REST architectural style in order not to couple publishers, subscribers, and the hub; (iii) XML is the default format of the messages. This paper also presents the results of experiments comparing Light-PubSubHubbub with the original PubSubHubbub protocol and the JMS technology for asynchronous messaging. The obtained results have shown that Light-PubSubHubbub takes less time to answer to the client than PubSubHubbub and JMS.**

Keywords-asynchronous communication; publish-subscribe; PubSubHubbub; Light-PubSubHubbub

I. INTRODUCTION

In the traditional way of the client-server communication, the server is the main element involved in the communication that receives and handles requests from clients. Nevertheless, such a model has shown to be significantly inefficient in situations in which data is frequently updated or such frequency is undetermined. In this case, the client needs to periodically send requests (synchronous) to the server to obtain the data and to check for updates. Such method is called *polling* [1], and although it meets the purpose of obtaining updates, it is not appropriate for situations when data are updated with an unknown frequency. Fig. 1 illustrates an example in which a client interested in updated data periodically makes requests to check for updates on the server. However, in this example, the client only gets an update on the third request; so, the first, second, and the fourth requests would be unnecessary because they do not provide any new information.

Although such a method makes the communication process between the client and the server simpler, it raises the question about the ideal period of time for making such requests. If a very large time period is chosen, the time for obtaining an update may be high and then it is possible to use

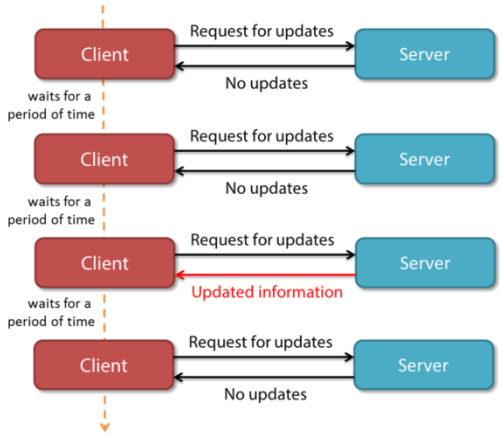

Figure 1. The polling method for obtaining information.

outdated information while an updated one is available. On the other hand, if the requests are performed in a very short period of time, unnecessary network traffic may be generated since the information may not be updated in such short time period [2]. In order to solve this problem, the *PubSubHubbub* protocol [3] was developed for dealing with event-oriented asynchronous requests [4] based on the *Publish-Subscribe* client-server communication model, in which *publishers* are responsible for sending messages to be consumed by *subscribers*. PubSubHubbub introduces a new element in such communication model called *hub*, which works as an intermediary between the publisher and subscriber elements. However, in the original implementation of such a protocol, the publisher needs to notify the hub that it has published an updated message in a Web *topic*, so that an additional processing must be performed by the hub in order to retrieve this new message and then forward it to the subscribers, thus generating unnecessary computation and network traffic. These limitations have motivated us to perform adaptations in the PubSubHubbub protocol to reduce its complexity, thus resulting in a lightweight protocol called *Light-PubSubHubbub*. In our approach, the publisher no longer needs to notify the hub that it has published a new message since publishers directly send the updated message to the hub instead of the Web topic. Therefore, no additional actions are performed to retrieve the published messages.

This paper is structured as follows. Section 2 gives an overview of the Publish-Subscribe communication model. Section 3 introduces the PubSubHubbub protocol. Section 4 presents the Light-PubSubHubbub protocol, resulted from adaptations in the original PubSubHubbub protocol. Section

 ISBN: 978-1-61208-304-9

5 presents a preliminary evaluation of the proposed protocol. Finally, Section 6 contains final remarks and future works.

II. The Publish-Subscribe Model

An *event* can be defined as a change in a state [4]. For example, when a person enters in his/her house, this action means a state change, i.e., an event. Events can be detected and dealt with applications through an *event-driven architecture* (EDA). Technically, such an approach enables the development of applications in which events trigger messages to be sent to independent modules of the application in an asynchronous way and according to the occurrence of such events.

In this context, the Publish-Subscribe model was developed as a model to deal with asynchronous messages in which *publishers* are responsible for sending messages that are consumed by *subscribers*. A great advantage of such model is the decoupling among its elements since publishers do not have knowledge about the subscribers registered for receiving their messages. However, the subscribers are able to choose what messages they want to receive from the publishers. Furthermore, subscribers receive only a subset of all published messages. The process of selecting such messages to be sent to the subscribers is called *filtering*, which can be *topic-based* or *content-based*. In a topic-based system, the messages are published in *topics*, which work as repositories of information of interest, so that subscribers will receive all messages published in the topic in which they have subscribed. In content-based systems, subscribers define constraints about the messages to be received, so that the messages are forwarded to them only if the message attributes or the content itself match the defined constraints.

In several systems that adopt the Publish-Subscribe model, there is an intermediary element called *broker* (or *event-bus*), which basically stores and forwards messages [5, 6]. In this kind of implementation, publishers publish messages in the broker, which forwards them to the subscribers that have been registered in the broker. There are also systems that do not use such intermediary element, so that publisher and subscriber share information (metadata) about themselves, thus forwarding messages based on the discovery of each other [6].

III. The PubSubHubbub Protocol

The *PubSubHubbub* protocol [3] is based on the Publish-Subscribe communication model and uses a broker element called *hub*. The hub is responsible for intermediating requests both from publishers (interested in distributing an updated information) and subscribers (interested in receiving the updates provided by the publishers), so that it receives update notifications from the publishers through an HTTP POST message, which informs the topic that has been updated. In a sequence, the hub makes a request to such topic in order to get the updated information. This request to the topic is performed through an HTTP GET message for obtaining updates, so that the updated information is forwarded to the subscribers through an HTTP POST message. Therefore, the PubSubHubbub protocol avoids that clients constantly perform checks for updates and it also eliminates the direct communication between the client and the server, which now is always intermediated by the hub (i.e., client–hub–server).

The PubSubHubbub protocol has four main elements:

1) The *topic* is the element in which the update information is published in the format of a feed by using the Atom [7] or Really Simple Syndication (RSS) [8] technologies. In general, the topic is publically available on the Web and can be accessed through an URL.

2) The *hub* is the element that works as an intermediary between the publisher and subscriber elements by: (i) receiving update notifications; (ii) accessing the topic provider in order to obtain updates; (iii) registering the subscribers, and; (iv) forwarding the updates to the subscribers.

3) The *publisher* is the element that publishes in the topic and is responsible for notifying the hub about the occurrence of an update. In the PubSubHubbub protocol, the publishers do not have to send the update to the hub. The publishers are only responsible for notifying it. The updates are published by the publisher as feeds, which is a data format used in communication transactions in which users frequently receive updated content.

4) The *subscriber* is the element that wants to receive updates regarding a given topic. In order to receive such updates, it is necessary that the subscriber has been subscribed in a topic of interest by making a request to the hub for subscribing to such topic. The hub will send to it the updates regarding the subscribed topic. The subscriber must be directly accessible through the network and identified by an URL.

PubSubHubbub works by performing three basic operations: (i) *discovery*; (ii) *subscription*, and; (iii) *publication*. In the discovery process the subscriber asks the publisher for a feed of a topic. Afterwards, the publisher sends the feed to the subscriber, which checks if there is an address regarding the hub used by the publisher for publishing updates in the topic and other important information, such as the update title and date when Atom feeds are used. If there is any reference to the hub in the feed sent by the publisher, then the subscriber can be subscribed to the referenced hub in order to obtain the updates whenever they are available. Otherwise, the subscriber must resort to the polling method or to other mechanism for obtaining updates regarding such topic since there is no reference to a hub in the feed, thus making impossible the use of the PubSubHubbub protocol.

In the subscription process, the subscriber requests the hub to subscribe to a topic by passing the address of the topic and the necessary information for sending the updates to the subscriber, and the hub confirms the subscription to the subscriber.

Fig. 2 illustrates the publication process in which the publisher publishes in the topic and immediately notifies the hub about the update by passing the address of the topic. In turn, the hub consults the address passed by the publisher and obtains the updated information for forwarding it to the interested subscribers. In such a communication model, the

 ISBN: 978-1-61208-304-9

update of information requires the following actions illustrated in Fig. 2:

(1) the publisher publishes a new information in the topic;
(2) the hub is notified about the update in the topic;
(3) the hub requests the topic about the new available information;
(4) the hub receives the new information from the topic, and;
(5) the update is distributed to the interested subscribers.

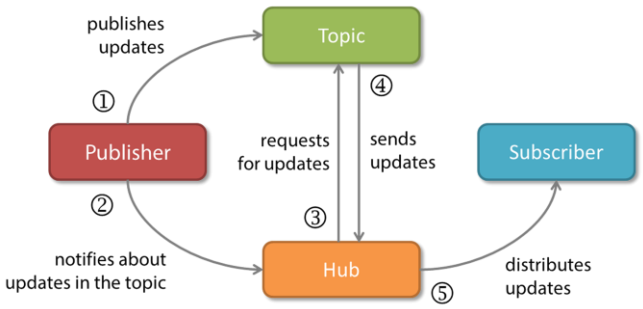

Figure 2. Processes performed by the PubSubHubbub protocol.

In addition, PubSubHubbub specifies operations based on the REpresentational State Transfer (REST) [9] architectural style, as means of establishing connections and requesting services, whether subscription or even publication requests. By using REST, PubSubHubbub can establish communications between hub, publishers, and subscribers by using only the HTTP protocol and several representation formats (e.g., XML, JSON [10] or plain text) without additional abstractions as in the SOAP protocol [11] for Web services.

IV. LIGHT-PUBSUBHUBBUB: AN ADAPTATION OF THE PUBSUBHUBBUB PROTOCOL

In PubSubHubbub, the publisher is not responsible for sending information to the hub when it is published. It is only responsible for notifying it. Afterwards, the hub makes a request to the topic in order to obtain the updated information through the informed URL. In the adoption of this model, some shortcomings can be observed, such as the unnecessary computation performed by the hub, which must access the topic at each publication, and the generation of unnecessary network traffic because the updated information must go to the topic and then be retrieved by the hub, as well as the possibility of the topic being unavailable when it is accessed.

In this perspective, the original PubSubHubbub protocol was modified, resulting in the *Light-PubSubHubbub* protocol [12]. In this new proposal, the hub is not responsible for accessing the topic (on the Web) in order to obtain the updates, thus eliminating the need of publically accessing the topic through an URL. In the publication request to the hub (implemented by following the REST architectural style), publishers send information to the topic that must be previously registered in the hub, not in a server on the Internet, so that the hub must forward the updated information to the subscribers to such topic.

In the Light-PubSubHubBub proposed protocol, the publisher sends the updated information and the identifier of a topic that is registered in the hub. Next, the hub checks if the passed identifier of the topic is already registered and if there are subscribers interested in such publication. If true, the hub sends the updated information to the subscribers that are registered for receiving it. In this new perspective, the communication process is as follows: in the publication request to the hub, publishers directly send the updated information to the hub by following the REST architectural style; next, the hub forwards the updated information to the subscribers.

Another change that was performed over the original PubSubHubbub protocol refers to the used topic. PubSubHubbub extends the Atom and RSS protocols by using them as means of obtaining updates about information hosted in a server on the Web. Nevertheless, as the publisher is used for sending update data to the hub in the implementation of the Light-PubSubHubbub protocol, it was observed that the Atom and RSS technologies originally used for sending information to the hub could be easily replaced by XML-based messages (extensively used for message exchanges in the Web) since additional information (e.g., the update date in the Atom protocol) would not be necessary because the hub has now control over the topic. Therefore, the hub receives a publication and forwards it to the subscribers, thus bringing a greater flexibility to the Light-PubSubHubbub protocol in terms of the message format (that can be represented as XML, JSON, plain text, etc.) since there is no restriction regarding it. However, the subscriber must know the message format in order to suitably and correctly parse it.

It is important to highlight that the hub just works as a distributer, i.e., it does not need to know the object format since it only receives and forwards a string that must be parsed by the subscribers. Hence, if the messages are represented in the XML format, for example, the transformations to (*marshaling*) and from (*unmarshaling*) the XML format must be respectively performed by the publishers and subscribers. In turn, the hub can distribute information in any text-based format.

The following subsections detail the communication processes in the Light-PubSubHubbub protocol.

A. Registration

Before making any publication, the topic must already be registered in the hub. To do that, a publisher requests the registration of a new topic through an HTTP request to the RESTful service that is responsible for registering new topics. In the registration request, the hub checks if the passed identifier has not already been registered and then makes and confirms the registration.

The registration process regarding a new topic is performed through an HTTP PUT request to the URL regarding the RESTful service that registers new topics in the hub. The HTTP PUT request for registering a new topic in the hub is made by the publisher to the following URL:

```
http://<hub's IP address>:<hub's port>/Hub/register
```

 ISBN: 978-1-61208-304-9

The identifier of the topic must be in the body of the HTTP message. Fig. 3 shows an example of an HTTP message sent to the hub aiming at registering a topic with the *sports* identifier. In Fig. 3, lines 1 to 7 correspond to the header of the HTTP message and line 9 corresponds to the body of the message with the identifier of the topic.

```
1: PUT /hub/register HTTP/1.1
2: Content-Type: text/plain
3: User-Agent: Java/1.6.0_43
4: Host: 127.0.0.1:8084
5: Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
6: Connection: keep-alive
7: Content-Length: 6
8:
9: sports
```

Figure 3. Example of HTTP PUT message sent to the hub for registering a topic with the *sports* identifier.

B. Subscription

The subscriber must be registered to receive the updates regarding its topics of interest. In this perspective, the subscriber sends to the hub an HTTP request to the RESTful service responsible by such requests and passes as parameters: (i) the identifier of the topic of interest, which is requested by the hub in order to identify which updates will be sent to the subscriber, and; (ii) an address and a port used for identifying to where the updates will be sent. Therefore, each subscriber is uniquely identified by a triple composed of its IP address, identifier of the topic of interest, and the port in which it will wait for the notifications. After receiving a new request for subscription, the hub checks if the identifier of the topic is already registered; if true, the informed address and the port are registered as interested in receiving updates regarding such topic.

For a new subscription, it is made an HTTP PUT request to the URL regarding the subscription in the hub:

```
http://<hub's IP address>:<hub's port>/Hub/subscribe
```

The body of an HTTP PUT request for new subscriptions must contain the identifier of the topic of interest, and the address and the port for receiving updates. Fig. 4 illustrates an example of an HTTP PUT request sent to the hub in order to make a subscription to the topic with the *sports* identifier. In Fig. 4, lines 1 to 7 correspond to the header of the HTTP message and line 9 corresponds to the body of the message with the identifier of the topic of interest, and the address and the port for receiving updates.

```
1: PUT /hub/subscribe HTTP/1.1
2: Content-Type: text/plain
3: User-Agent: Java/1.6.0_43
4: Host: 127.0.0.1:8084
5: Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
6: Connection: keep-alive
7: Content-Length: 57
8:
9: {"address":"127.0.0.1","port":"50355","topic":"sports"}
```

Figure 4. Example of HTTP PUT message sent to the hub for subscribing to the topic with the *sports* identifier.

C. Unsubscription

If a subscriber does not want to receive anymore updates regarding a topic, it is necessary to make an HTTP request to the RESTful service responsible for cancelling such action. As a client may be registered for receiving updates regarding more than one topic, it is necessary to specify the information about the client and the identifier of the topic. In order to perform such operation, the subscriber sends the identifier of the topic, and its address and port, so that the hub removes such client from the list of interested subscribers.

In order to cancel a subscription, an HTTP DELETE request to the URL regarding the RESTful service responsible for such operation is made by passing through such URL the information that uniquely identify the resource to be deleted. Unlike the previous operations in which the parameters can be directly sent in the body of the HTTP message, the information for this operation is passed in the URL itself due to limitations of the HTTP DELETE request. Such URL is as follows:

```
http://<hub's IP address>:<hub's port>/Hub/
unsubscribe/?address=<subscriber's IP address>
        &idTopic=<topic of interest>
          &port=<subscriber's port>
```

Fig. 5 illustrates an example of an HTTP DELETE request to the hub aiming at unsubscribing a subscriber from the topic with the *sports* identifier. In Fig. 5, lines 1 to 6 correspond to the header of the HTTP message, which can be empty because the information needed to cancel the subscription have already been sent in the URL of the request.

```
1: DELETE /hub/unsubscribe/?address=127.0.0.1&idTopic=sports&
2:     port=14746 HTTP/1.1
3: User-Agent: Java/1.6.0_43
4: Host: 127.0.0.1:8084
5: Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
6: Connection: keep-alive
```

Figure 5. Example of HTTP DELETE message sent to the hub for unsubscribing to the topic with the *sports* identifier.

D. Publication

The publication process only happens when the topic that is being updated is already registered in the hub, otherwise a "topic not found" exception is thrown. In order to make a publication, a publisher sends to the hub the identifier of the topic and the value to be published. The hub checks if the passed identifier is already registered, and if true, it stores the information contained in the request.

The publication is performed through an HTTP POST request to the hub containing the identifier of the topic of interest that must be updated. The request URL is as follows:

```
http://<hub's IP address>:<hub's port>/Hub/publish/
            <identifier of the topic>
```

After publishing, the hub sends the updated information to all subscribers registered for the current topic by using their respective address and port that were previously registered when subscribing. If there is no registered subscriber, the information is immediately discarded.

 ISBN: 978-1-61208-304-9

The body of the HTTP message for publishing a new content regarding a given topic must contain the value for update, which can be a string or even a XML representation of an object that must be parsed by the subscribers. Fig. 6 illustrates an example of an HTTP POST request to the hub in which the publisher wants to publish information in the topic with the *sports* identifier.

```
1:  POST /hub/publish/sports HTTP/1.1
2:  Content-Type: text/plain
3:  User-Agent: Java/1.6.0_43
4:  Host: 127.0.0.1:8084
5:  Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
6:  Connection: keep-alive
7:  Content-Length: 80
8:
9:  The 2014 FIFA World Cup will take place in Brazil from 12 June
10:     to 13 July 2014
```

Figure 6. Example of HTTP POST message sent to the hub for publishing information in the topic with the *sports* identifier.

In Fig. 6, lines 1 to 7 correspond to the header of the HTTP message, and lines 9 and 10 correspond to the body of the message that represents a new information to be forwarded to the subscribers. In such example, the update is regarding a message as a string.

V. EVALUATION

A. QoMonitor

The conducted case study consists of a ubiquitous oil and gas application that illustrates the need of monitoring the Quality of Service (QoS) and Quality of Context (QoC) of the services used by it. For monitoring such services, the *QoMonitor* [13, 14] system assesses, monitors, and makes available QoS and QoC metadata regarding services to be used by clients such as middleware platforms, Web services, applications, etc. *QoMonitor* handles synchronous and asynchronous requests from clients, both returning QoS and QoC metadata regarding a given service or a set of services. In synchronous requests, *QoMonitor* receives a request, processes the information, and answers to the client, i.e., the response time of such operation is the time for transporting the request/response over the network and the time for processing the request. In asynchronous requests, *QoMonitor* receives a subscription request, processes the information, and waits until a particular event (return condition) happens, and then asynchronously responds to the client, which needs to provide means of receiving responses from *QoMonitor*. To do that, the original implementation of *QoMonitor* uses a Java Message Service (JMS) [15] topic for forwarding the result of the subscription to the client when *QoMonitor* publishes in such topic. More details can be found at the URL http://consiste.dimap.ufrn.br/projects/lightpubsubhubbub/icsea2013.

However, the JMS technology generates a coupling between the clients and *QoMonitor* since JMS only works when the client is developed by using the Java programming language. In this context, the Light-PubSubHubbub protocol could have a key role since it enables the asynchronous communication between the clients and *QoMonitor* without generating a coupling because Light-PubSubHubbub was developed as a Web service.

B. Experiments and results

The performed experiments were aimed to address the overhead due to the use of the Light-PubSubHubbub protocol in comparison with the original PubSubHubbub protocol and the JMS technology, when *QoMonitor* notifies its clients about the event (return condition) regarding the asynchronous request. In the experiments, five different computers were connected to the same wired LAN network (in order to minimize the influence of the network) according to the experimental setup shown in Fig. 7. In order to calculate such overhead, a time Web service was developed for sharing the current time among the client, *QoMonitor*, and the used topic (JMS or hub). When *QoMonitor* publishes the notification in the topic, the time service is accessed for retrieving the current time and this time is stored. Afterwards, when the client receives the notification from JMS topic or the hub, it accesses the time service and the obtained time is subtracted from the time retrieved by *QoMonitor*, thus resulting in the time spent by the topic for answering to the client.

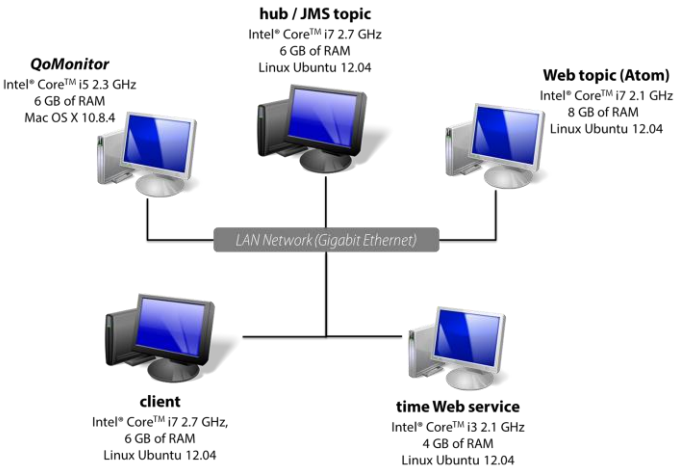

Figure 7. Infrastructure used in the evaluation of the Light-PubSubHubbub protocol compared with the original PubSubHubbub protocol and the JMS technology.

The experiments were conducted in three sequential phases. In the first one, the JMS technology was used by *QoMonitor* for communicating with the client, so that the client has performed an asynchronous request to *QoMonitor*, which has registered it in the JMS topic. Similarly, in the second and third phases, the PubSubHubbub and Light-PubSubHubbub protocols were respectively used, so that the client has performed an asynchronous request to *QoMonitor*, which has registered it in the hub. When the return condition was satisfied, *QoMonitor* answered to the client by using the used topic (JMS or hub). Twenty independent executions for the process of publishing and receiving the subsequent notification message were performed.

Table I presents the minimum, average, maximum, and standard deviation times spent (in milliseconds) by JMS, PubSubHubbub, and Light-PubSubHubbub within *QoMoni-*

 ISBN: 978-1-61208-304-9

tor. As can be observed in Table I, Light-PubSubHubbub takes less time to answer to the client than JMS and the original PubSubHubbub, thus resulting in a reduction of approximately 40% compared with JMS and 93% when compared to PubSubHubbub, on average.

TABLE I. TIME SPENT BY THE JMS TECHNOLOGY AND THE PUBSUBHUBBUB AND LIGHT -PUBSUBHUBBUB PROTOCOLS WITHIN QOMONITOR.

Technology	Minimum	Maximum	Average	Standard deviation
JMS	22.7671	30.6794	24.4792	1.7112
PubSubHubbub	173.4899	302.8242	209.085	33.2603
Light-PubSubHubbub	13.5101	19.3910	14.5962	1.3127

The considerable reduction observed when comparing Light-PubSubHubbub with the original version of the protocol is mainly due the fact that messages are directly sent to the hub instead of being posted to a Web topic, so that the hub can retrieve the message and then forward to the client, as in the original PubSubHubbub. Furthermore, as we have already argued, Light-PubSubHubbub does not generate a strong coupling between *QoMonitor* and the client since it was developed as a Web service, unlike the JMS technology that requires that the client be implemented by using the Java programming language.

VI. RELATED WORK

PubSubHubbub is a well-known protocol that has been used as a plug-in in several blog tools and content management systems (CMS) such as WordPress, Tumblr, Joomla, etc. Furthermore, there is also several works in the literature that have the same purposes of the PubSubHubbub protocol. For instance, the Java Message Service (JMS) [15] is a message-oriented middleware (MOM) that defines a set of interfaces that enable Java applications to communicate with each other. The JMS API enables asynchronism since it delivers the messages to consumers as soon as they are sent from the message producers, so that that consumers do not need to periodically request for the messages in order to receive them (as in the polling method). The JMS API also ensures that a message will be delivered one and only once, in a reliable way. The connection between consumers and producers can follow two basic models: (i) *point-to-point*, in which producers know consumers and directly deliver the message to them; or, (ii) *publish/subscribe*, in which publishers do not know subscribers and vice-versa since the communication among them is performed through the *JMS topic*, which receives the messages sent from publishers and forwards them to the interested subscribers. Since JMS is a Java technology, publishers and subscribers must be developed by using the Java programming language, thus generating a dependency in terms of technology, which does not happen in Light-PubSubHubbub.

Trifa [2] presents the Web Messaging System (WMS) protocol, which is based on the Publish-Subscribe model and is essentially similar to the PubSubHubbub protocol. WMS specifies the core functions of a Publisher-Subscribe system by using RESTful design patterns over HTTP interactions instead of developing a custom messaging protocol on the top of the HTTP protocol. In addition, it envisions a broker (very similar to the hub in the Light-PubSubHubbub protocol) that is responsible for storing the messages in an embedded database until their delivery to the subscribers. Despite of ensuring the delivery of the messages to the subscribers, this database storage may increase the latency for delivering the messages, as reported by the author.

In turn, Senn [16] uses the PubSubHubbub protocol in Wisspr (Web Infrastructure for Sensor Streams PRocessing), a Web-based framework for handling sensor data. Wisspr is built upon a Publish-Subscribe system in order to facilitate the development of event-driven and real-time processing applications for Web of Things by storing sensor data from different sources (e.g., mobile devices, home appliances, etc.) in a relational database. All sensor data are available from the PubSubHubbub protocol through a uniform RESTful interface, which enables to easily publish and consume data, as in Light-PubSubHubbub.

Another interesting publish-subscribe protocol is MobilePSM [17], which is intended to support mobile clients for publish-subscribe middleware. MobilePSM ensures that messages are not lost nor duplicated by temporarily storing them in a broker during the moving period, so that mobile clients can receive messages according to the sending order when a mobile client moves from one network to another or it is passively disconnected. This temporarily storage for providing reliability in terms of message delivering is an interesting feature that is not currently provided by Light-PubSubHubbub.

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented Light-PubSubHubbub [12], a lightweight version of the PubSubHubbub protocol [3] to deal with asynchronous message exchanges in the Internet. This new version introduced the following changes: (i) the publisher no longer needs to publish updated messages in a Web topic and then notify the hub since the messages are published in the hub itself; (ii) it uses the REST architectural style in order not to couple publishers, subscribers, and the hub, and; (iii) XML is the default format of the messages. We performed experiments to compare Light-PubSubHubbub with the original PubSubHubbub protocol and the JMS technology [15] for asynchronous messaging, and the obtained results have shown that Light-PubSubHubbub takes less time to answer to the client than PubSubHubbub (a reduction of 93% on average) and JMS (a reduction of 40% on average). In addition, the implementation is not constrained to a specific technology since JMS uses the Java programming language, whilst Light-PubSubHubbub is implemented as a Web service.

As ongoing work, we are investigating how to address some current limitations of Light-PubSubHubbub. The first one regards to the temporary persistence of the messages when the subscriber is busy or down. Moreover, Light-PubSubHubbub does not provide any mechanism to ensure that only the owner of a topic can publish updated in such topic; in the PubSubHubbub protocol, publishers receive keys when registering for a topic and must use them in order

 ISBN: 978-1-61208-304-9

to publish the updates. Finally, it is important to provide means of securing of the messages, in terms of crypto-graphing the exchanged messages.

ACKNOWLEDGMENT

This work is partially supported by the Brazilian National Agency of Petroleum, Natural Gas and Biofuels (ANP), through the PRH-22 Project.

REFERENCES

[1] H. Levi and M. Sidi, "Polling systems: Applications, modeling, and optimization", IEEE Transactions on Communications, vol. 38, no. 10, Aug. 1990, pp. 1750-1760.

[2] M. V. Trifa, Building blocks for a participatory Web of Things: Devices, infrastructures, and programming frameworks – PhD dissertation. Swiss Federal Institute of Technology Zurich, Switzerland, 2011.

[3] PubSubHubbub: http://pubsubhubbub.googlecode.com/ (access on September, 2013)

[4] K. Mani Chandy, "Event-driven applications: Cost, benefits and design approches", Gartner Application Integration and Web Services Summit, 2006.

[5] G. Cugola and H.A. Jac obsen, "Using Publish/Subscribe middleware for mobile systems", ACM SIGMOBILE Mobile Computing and Communications Review, vol. 6, no. 4, Oct. 2002, pp. 25-33.

[6] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec, "The many faces of Publish/Subscribe", ACM Computing Surveys, vol. 35, no. 2, Jun. 2003, pp. 114-131.

[7] Atom Publishing Protocol: http://www.ietf.org/rfc/rfc5023.txt (access on September, 2013)

[8] RSS Specification: http://www.rssboard.org/rss-specification (access on September, 2013)

[9] L. Richardson and S. Ruby, RESTful Web services. USA: O'Reilly, 2007.

[10] JSON: http://www.json.org/ (access on September, 2013)

[11] Simple Object Acess Protocol (SOAP) 1.2: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/ (access on September, 2013)

[12] Light-PubSubHubbub: http://consiste.dimap.ufrn.br/projects/lightpubsubhubbub/ (access on September, 2013)

[13] C. Batista et al., "A metadata monitoring system for Ubiquitous Computing", Proc. of the 6th Int. Conf. on Mobile Ubiqutious Computing, Systems, Services and Technologies (UBICOMM 2012). USA: IARIA, 2012, pp. 60-66.

[14] QoMonitor: http://consiste.dimap.ufrn.br/projects/qomonitor/ (access on September, 2013)

[15] Java Message Service (JMS): http://www.oracle.com/technetwork/java/jms/index.html (access on September, 2013)

[16] O. Senn, WISSPR: A Web-based infrastructure for sensor data streams sharing, processing and storage – Master's thesis. Swiss Federal Institute of Technology Zurich, Switzerland, 2010.

[17] T. Xue and T. Guan, "A protocol to support mobile computing for publish/subscribe middleware", Proc. of the 2012 Int. Conf. on Communication, Electronics and Automation Engineering, Advances in Intelligent Systems and Computing Series, vol. 181, G. Yang, Ed. Germany: Springer-Verlag Berlin/Heideberg, 2013, pp. 845-849.

 ISBN: 978-1-61208-304-9