RESTful Triple Spaces of Things

Aitor Gómez-Goiri, Pablo Orduña, Diego López-de-Ipiña
Deusto Institute of Technology - DeustoTech
University of Deusto
Avda. Universidades 24, 48007
Bilbao, Spain
{aitor.gomez, pablo.orduna, dipina} @deusto.es

ABSTRACT

The demand for Internet-enabled objects which expose their content in a RESTful and web compliant manner is increasing. Consequently, these objects have to face well-known problems from the web world. The lack of expressiveness and human orientation of the syntactically described capabilities and contents of those resources is one of these difficulties. The Semantic Web on the contrary interlinks each object's data to one another, enabling its automatic process to reveal possible new relationships and therefore enhancing the interoperability of semantic-enabled objects. In this work we present a semantically enabled Web of Things compliant HTTP interface for Internet-enabled objects which uses Triple Spaces (TS) as a basis. Specifically, we address the adoption of this paradigm by a wide range of resource constrained devices assessing the feasibility of our middleware solution, focusing both on the web and on the semantic aspects. Besides, we stress the degree of interoperability achieved by the applications made using RESTful TS by describing two scenarios where it could be used.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.2.11 [**Software Architectures**]: Patterns—*blackboard*

Keywords

Web of Things, Semantic Web, Triple Space Computing

1. INTRODUCTION

As the Internet of Things (IoT) becomes more present, more initiatives are paying heed to the integration challenges that this network of heterogeneous devices and platforms creates. One notorious approach is the Web of Things (WoT) [5], where the capabilities of the devices are shared through RESTful web services provided from embedded web servers. As a result, the objects can be easily integrated with existing web applications and widespread RESTful services in Internet.

Another integration style, more focused on coordinating different devices than on just sharing information, is tuplespaces. In tuplespaces, different processes can exchange information reading and writing in a common space. A remarkable particularization of tuplespaces is Triple Space Computing (TS), where semantic RDF triples are shared in that common space. Due to the use of the Semantic Web -a common technique used to reach interoperability between different applications and machines- not explicitly stated content can be inferred by correlating different devices' semantic data.

In this work, we present our adaptation of Triple Spaces to resource constrained devices through an HTTP API. Thanks to this API, a wide range of devices can be part of TS in a WoT compliant manner [6]. To support this hypothesis the solution has been tested in different devices and in two different stereotypical scenarios.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 briefly explains our TS solution. Section 4 explains two study cases where the middleware could be beneficial in terms of interoperability. Section 5 evaluates the use of TS in the different devices employed in the scenarios. Finally, Section 6 concludes and discusses future work.

2. RELATED WORK

Many different styles can be used to integrate applications depending on application requirements. Hohpe et al. [7] enumerate batch data exchange, used to export data between two systems offline; a shared database, which eliminates synchronization issues having the data in a single place but enforces the use of a common data model; raw data exchange, where two entities agree the format and how the data is synchronously exchanged; remote procedure calls, which isolate the application from raw data exchange; and message exchange, which provides reliable asynchronous data transfer. HTTP has been frequently used in combination with different styles, as in the well known set of WS-*standards and most of services following the REST style. While the WS-* can be seen as both RPC or message exchange substyle, the REST style defines an elegant way to define resource-oriented interfaces which comply with the RPC idea of isolating application from exchange particularities [11]. The use of both substyles in resource-constrained devices is represented by *Device Profile for Web Services* (DPWS) [9], based on SOAP, and the Constrained Application Protocol

WoT 2012, June 2012; Newcastle, UK

(CoAP) [12] and the Web of Things initiative [5].

Other paradigms follow the idea proposed by Hohpe's *shared database* style of sharing a common space to coordinate different entities. One of those paradigms is tuplespace, where different processes read and write pieces of information called tuples in a space, possibly distributed over different devices. This paradigm has already been adapted to embedded and mobile devices [1] and in conjunction with the Semantic Web [10]. In contrast with traditional integration approaches, the Semantic Web enables the use of very expressive information and the inference of new implicit information. One of the approaches which mixes up Semantic Web with tuplespaces is Triple Space Computing, which uses similar primitives with RDF triples as exchanged units.

Although different semantic tupplespaces implementations exist, apart from our solution, only Smart-M3 [8] has been specifically designed to be run in devices with constrained capabilities. The most remarkable difference with Smart-M3 is the complete decentralization of the information. In Smart-M3 the spaces are managed by few machines and used by the rest of the devices which act as clients using an XML protocol. In this work every device is in charge of the knowledge it stores and it is responsible for querying the rest of the devices belonging to the same space using their HTTP API.

While the first approach is simpler and relieves the constrained devices from answering to the queries, we believe that the upcoming capabilities of embedded and mobile devices make possible a more autonomous space where any device is responsible of its information without relying in a third one and providing fresher sensed data. At the same time, our solution is browsable and *mash up able* since it enables any other WoT solution or web application to simply consume the information provided by each device in a RESTful compliant manner. To do that, the developer needs to know how to interpret the semantic response, which does not differ much from what a simpler JSON response requires.

3. TRIPLE SPACES OVER HTTP

So far, the compatibility of Triple Spaces with the HTTP RESTful style has been proved both from a formal [6] and qualitative point of view [3]. In this section our model towards the achievement of this mapping will be discussed in detail. This model has been implemented and is publicly available in the *Otsopack* Open Source project[1].

The Triple Spaces (TS) paradigm basically consists on writing and reading triples grouped in graphs in different spaces. Therefore, the resources in TS are RDF graphs identified by an URI. The most important primitives of our solution are *write*, *read*, *take* and *query*. *Write* adds new knowledge to a space writing together the given triples in a new RDF graph and returning its identifying URI. *Read* and *take* return a whole RDF graph which can be selected by its identifying URI or by a template. When a template is used, the graph is returned if it contains at least a triple which matches it. *Take* differs from *read* because it also subtracts the returned graph from the space. *Query* returns the triples which match a given template conceiving the space as a whole no matter to which graph they belong to.

One could argue that the query primitive does not really represent a resource since it returns RDF triples from different resources (i.e. graphs). In any case, we have included it because this primitive provides a more fine grained view of the knowledge hold in an space which sometimes may be useful for the developers using the middleware. The template used in these primitives can be as simple as a triple pattern with wildcards (e.g. "`?s rdf:type ont:Sensor`") or as complex as an SPARQL query, but in this work we have just considered the first ones.

[1] `http://code.google.com/p/otsopack`

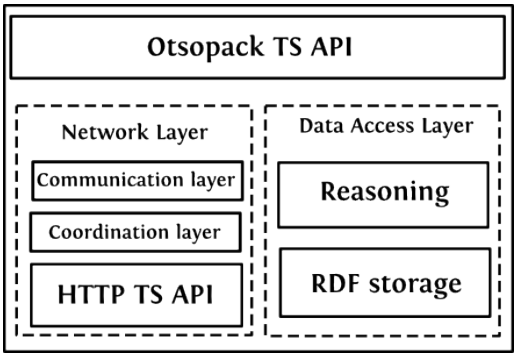


Figure 1: Internal structure of a typical Otsopack implementation.

Remarkably, TS guarantees four basic types of autonomies: a) *time*, since one application can store information in the common space and other applications consume it later in an asynchronous way; b) *space*, since applications can run in different environments as long as they support tuple spaces; c) *reference*, since applications should not need to know where the space is physically stored; and d) *vocabulary*, since the information is stored in a semantic format. This way, two applications using standard ontologies can interact among them automatically enriching one each other, as long as they use the same space and standard and linked ontologies.

The Otsopack middleware proposed contains the layers described in Figure 1. A TS API is provided to the developer using the middleware in each device, which has the primitives described above and employs two main functional blocks: network and data access layers. The network layer embraces a) the HTTP API provided by any device belonging to a space to made its data available to others and b) the strategy used to access the knowledge stored on other devices. At the same time, these strategies use the coordination layer to discover new devices and the communication layer to consume those devices' HTTP APIs.

The knowledge distribution strategies used in TS may vary from centralized to completely distributed ones. In the proposed middleware, the technique known as *negative broadcasting* is assumed, which implies that all write operations are executed locally at the node (i.e. the instance of the middleware running on a device) and all read and query operations are propagated to the rest of the nodes belonging to the same space. Nevertheless, the dissemination and discovery strategies can be considered transversal to the adoption of the proposed HTTP API by the resource constrained devices and therefore they are out of the scope of this paper. This way, the results obtained in the evaluation section are valid for any other web-enabled object which uses full semantics to describe the provided information. In the

Table 1: Examples of REST access to TS. *sp* is a space URI, *g* is a graph URI, *t* a set of triples, *s*, *p* and *o-uri* are subject, predicate and object uris or wildcards (represented with a *). When the template's object is a literal, it can be expressed specifying its value (*o-val*) and its type (*o-type*).

HTTP request	URL	Returns
GET	http://nodeuri/{sp}/graphs/{g}	`read({s},{g}): t`
GET	http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-uri} http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-type}/{o-val}	`read({sp},"{s} {p} {o} .":t`
DELETE	http://nodeuri/{sp}/graphs/{g}	`take({sp},{g}): t`
DELETE	http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-uri} http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-type}/{o-val}	`take({sp},"{s} {p} {o} .":t`
GET	http://nodeuri/{sp}/query/wildcards/{s}/{p}/{o-uri} http://nodeuri/{sp}/query/wildcards/{s}/{p}/{o-type}/{o-val}	`query({sp},"{s} {p} {o} .": t`

end, this is what an Otsopack node that does not perform queries and justs offers its data to other Otsopack nodes, or even to other WoT solutions, constitutes.

On the other hand, as many REST architectural styles use HTTP verbs to retrieve, create, modify or delete web resources and TS does the same with RDF graphs, both concepts can be easily mapped [6, 3]. More specifically, the mapping between both styles is summarized in Table 1. Note that the write primitive is intentionally excluded of this HTTP API because the writings are always locally invoked using the middleware's API.

Otsopack complies with the standardized HTTP status codes sent back as part of the header in the response (e.g. when no significant result can be found for a primitive the 404 error is returned). This adoption - apart from enhancing the compatibility with other web applications - enables the modular adoption of our API. For instance, if a node does not offer a wildcard based *query* and returns a 404 error, it will not affect the behavior of the rest of the nodes of an space. Instead, they will interpret these cases as empty responses. This modularity becomes crucial to enable the partial adoption of the API on new platforms.

Another key aspect of the HTTP protocol we have taken advantage of is the *content negotiation*. This mechanism allows to specify the desired representation for a content on the client side and to express what representation is sent as a response from the data provider side. For that purpose the consumer adds an *Accept* field to the HTTP header with a weighted list of media types it understands and the provider answers with the best possible format it knows about, specifying the *Content-type* in the response.

In Otsopack this mechanism not only enhances the browsability of the primitives with human understandable HTML responses, but they allow different Semantic representations (e.g. RDF/XML[2], N-Triples[3] or N3[4]). This characteristic becomes crucial since not all the nodes may understand all the languages (e.g. a mobile phone may not have a RDF/XML parser), even if they all use the same basic concepts: RDF Triples. The compatibility of both sides can be ensured through a conversion carried out in the provider side. On the other hand, even if both the provider and the consumer know how to use different languages, the preference of some of them can be easily expressed to achieve other goals (e.g. to obtain the less verbosed answer).

[2]`http://www.w3.org/TR/REC-rdf-syntax/`
[3]`http://www.w3.org/2001/sw/RDFCore/ntriples/`
[4]`http://www.w3.org/TeamSubmission/n3/`

In the data access layer, as any semantic-based solution the information stored can be expanded by a Semantic Reasoner. Even if at application level the data provider writes a few triples, this set can be expanded with inferred information and other nodes can perform queries on that information. Therefore, the amount of requests that can be answered is increased, even if they were not considered in the producer's design phase. For instance, if a sensor defines that *the kitchen light is on*, and the ontology defines that the *kitchen is next to the living room* and that the *light is a sensor*, a potential consumer may query for *all the sensors close to the living room*.

Although the main API implementation employs RESTlet (making it Java SE and Android compliant), it has been easily adapted to very limited devices using Python's standard libraries for this work.

4. STEREOTYPICAL SCENARIOS

Otsopack has been used in real scenarios both in a supermarket and in a hospital with more complex applications within the ACROSS project[5]. For the sake of brevity and clarity, two simple and not implemented applications will be described to show how this middleware can be used to achieve a higher degree of interoperability than syntactic WoT solutions.

The applications modeled are the following ones: *otsoSecurity* and *otsoHomeAutomation*. Both applications consume the data provided by the devices assessed in the evaluation section. In both scenarios they coordinate in a decoupled mode following the specialist pattern [2]. In this pattern a master writes a task into the space and waits for its result, which is performed by some of the workers specialized in this particular task (e.g. show a message or regulate the temperature).

4.1 Security

A security company can develop an application which monitors different parameters such as the temperature, the humidity or the CO_2 concentration with different sensors deployed over an industrial facility. Whenever any of this measures go beyond a determined threshold, the company needs to take the proper action. To answer to the potential risks the application creates tasks with different priorities: when a unimportant parameter is outside the expected boundaries the application can write a low priority task for the security manager into the space (e.g. the CO_2 is slightly

[5]`http://www.acrosspse.com`

higher than the normal one), but to warn about an emergency to the users in the facility a high priority one can be written (e.g when they must leave the building). Then, the message is consumed by different actuators according to its priority (e.g. in the manager's phone in a less intrusive manner or through visual or auditory alarms over the building).

The company can also develop a simpler version of the same application for the workers' personal mobile phones to ensure that they are warned even if the alarms of the main application fail. To implement both versions of the application, commonly used ontologies such as SSN (Semantic Sensor Network Ontology) or SWEET (Semantic Web for Earth and Environmental Terminology) can be used, storing and sharing the triples detailed in Listing 1 in a graph.

Listing 1: Sample triples provided by a NO_2 sensor deployed in the facility.

```
Subject      Predicate                  Object

wot:meas1    rdf:type                   ssn:Observation
wot:meas1    ssn:observedProperty       sweet:NO2
wot:meas1    ssn:observationResult      wot:outpt1
wot:outpt1   ssn:hasValue               wot:val1
wot:val1     ssb:QuantityValue          17
wot:val1     dul:isClassifiedBy
                 muo-ucum:microgram-per-cubic-meter
  ...          ...                        ...
```

4.2 Home automation

On the one hand, a room has been populated with several kind of sensors connected to XBee sensors[6] with an IP gateway[7], FoxG20[8] embedded platform connected to sensors and to an actuator. Besides, an Android application could be performed to semantically store the user's temperature preferences. An independent node (master node) continuously checks the room temperature using *read primitive* to get the first available graph where the last measure is defined (no matter which device provides that information) and the user's desired temperature. When the second one is below the first one, it generates a "decrease temperature during a certain period" task which can be consumed by different independent worker nodes. In this case, the FoxG20 periodically checks just for orders it can fulfill and it understands and consumes them with a *take primitive*.

Once again common ontologies such as SSN (Semantic Sensor Network Ontology), MUO (Measurement Units Ontology) or RECO (RECommendations Ontology) are used to express these relations. Sample triples provided by the mobile phone can be found in Listing 2.

Listing 2: Sample triples stored by the Home Automation application.

```
Subject      Predicate                  Object

ud:aigomez   reco:desireTowards         ud:pref1
ud:pref1     rdf:type                   reco:Preference
ud:pref1     ssn:observedProperty       swt:Temperature
ud:prefm     ssn:observationResult      ud:dout1
ud:dout1     ssn:hasValue               ud:dVal
ud:dVal      ssn:QuantityValue          20
...          ...                        ...
```

[6] `http://tinyurl.com/xbee-sensors`
[7] `http://tinyurl.com/connectportx2`
[8] `http://www.acmesystems.it`

Table 2: Technical characteristics of the assessed devices.

Device	Processor	RAM
XBee	-	8 MB
FoxG20	400Mhz Atmel ARM9	64 MB
Galaxy Tab	1 GHz Cortex-A8	512 MB
Regular computer	2.26 GHz Intel Core 2 Duo	4 GB

4.3 Interoperability

Given that both systems use a common ontology called SSN, and through Triple Spaces they can be using a common space, whenever the Security application asks for triples matching a template "`?s rdf:type sweet:Temperature`", the Home automation application would return that "`wot:mes3 rdf:type sweet:Temperature`" along with other information stored in that graph. Therefore, the Security application would be able to retrieve information from another application it does not even know. In the same way, it is feasible that the Home automation application also retrieves information stored by the Security application in the same or other nodes.

The key for this interoperability process is that both applications are using the same language, since both are using the same concepts of the same ontologies (e.g. SSN). Although this can be achieved mapping concepts from two different ontologies with a semantic web reasoner through the "`owl:sameAs`" property, it is habitual to use common ontologies. Furthermore, since all the applications should be interested in retrieving data from other potential ones, the developers should be willing to employ widely used ontologies to ease the information exchange among applications.

5. EVALUATION

In order to prove the easy adoption of the proposed middleware by different platforms and its feasibility, the performance of three different resource-constrained platforms has been evaluated: XBee, FoxG20 and Samsung Galaxy Tab (for further details see the Table 2). The performance of Otsopack in a regular computer is also provided as a baseline for the comparison.

For the RDFS reasoning evaluation, real examples from the SSN ontology[9] have been used. First the TBox[10] with the definition of the terms of this ontology has been loaded and then the writing of ten different measures was averaged. These measures are represented by graphs (ABox) with nearly 10 triples extracted from the Bizkaisense dataset[11].

Regarding the energy consumption of each implementation, it greatly depends on network traffic generated by the knowledge dissemination strategy selected, which has already been analyzed on a previous work [4].

[9] `http://www.w3.org/2005/Incubator/ssn/wiki/Semantic_Sensor_Net_Ontology`
[10] In the Semantic Web, TBox contains the knowledge which describes general properties of concepts or terminology (e.g. the type of devices and the sensors they have) and ABox contains knowledge that is specific to the individuals of the domain of discourse (e.g. the mobile brand is HTC or the sensed temperature is 3°C).
[11] `http://dev.morelab.deusto.es/bizkaisense`

5.1 Case of study 1: XBee

It is only possible to develop software in XBee using Python. Therefore we implemented a version of the middleware compatible with Otsopack in Python. This implementation was tested with the full Otsopack version successfully. However, when measuring it problems started to arise when more than 15 requests were performed concurrently in XBee -which is not a usual scenario-.

5.2 Case of study 2: FoxG20

Given that the Python implementation provides the core functionality of Otsopack, the server was lighter than the regular Android/Java SE version of Otsopack, and therefore it was the version used in FoxG20. FoxG20 is a more powerful platform than XBee. In fact, it can even perform OWL reasoning in Python using the Fuxi library [12]. However, as the Table 4 shows, the reasoning process in the FoxG20 takes a long time and therefore should be limited to special occasions.

5.3 Case of study 3: Tablet computer

To assess the Android implementation on a mobile platform, we selected the Samsung Galaxy Tab as a representative of the powerful mobiles which are becoming widely adopted. Its reasoning performance is halfway between the FoxG20 (seven times faster for small graphs and almost three times faster for longs graphs with the TBox), but yet far enough from the regular computer.

The tablet handles HTTP requests worse than the FoxG20. Although the web framework used in Android may not be as optimized as the Python standard one, the main cause might be that the FoxG20 does not implement all the modules defined by Otsopack. This may sound as a sign of incompleteness, but it is a desirable behavior for less powerful devices which does not affect to the rest of the nodes in the space.

5.4 Case of study 4: regular computer

To complete the evaluation, we have measured Otsopack in a regular environment using the Java SE implementation. Table 2(a) and Figure 2(b) summarize the data of these measurements, compared with the data retrieved by the Galaxy Tab, the FoxG20 and the XBee. As expected, the performance is notably higher in a computer rather than in an embedded device.

Although the time needed by the inference process varies significantly depending on the reasoning engine used, it generally requires much less time than with any other device. This stresses the open challenge it represents to reason in resource constrained devices.

5.5 Discussion

The proposed TS API over HTTP is lightweight enough to be successfully adopted by a range of devices. The response times are small and even the XBee, which takes around 77 milliseconds and 775 if 10 clients are performing concurrent requests, could be perfectly affordable in small or non-time critical scenarios.

In Table 3, we present a qualitative analysis of the different Otsopack technologies used for the implementations. As detailed in the table, we have tested four different architectures (Java SE, Android, FoxG20 and XBee) using two programming platforms (Java SE 6 and Android with Restlet and Python 2.4 with the Python Standard Library).

[12] https://github.com/RDFLib/FuXi

Table 3: Core libraries of Otsopack

	Platform version	REST libraries	Semantic libraries
Java SE	Java SE 6.0	Restlet	Rdf2Go
Android	Android 2.2	Restlet	Sesame
FoxG20	Python 2.5	Python Std Lib	Fuxi
XBee	Python 2.4	Python Std Lib	None

Table 4: Reasoning performance for a regular computer, a tablet computer and a FoxG20 (seconds)

Device	TBox	ABox
Regular computer (Sesame 2.6.4)	2.787	0.045
Samsung Galaxy Tab (Sesame 2.4.2)	17.342	0.225
FoxG20 (Fuxi)	48.939	1.443

Attending to these platforms, different reasoning levels can be achieved: Rdf2Go[13] in Java SE encapsulates different engines; Sesame[14] in Android supports RDFS and performs better than AndroJena[15], which additionally supports OWL; Fuxi supports OWL, and no reasoning engine could be executed in the XBee.

6. CONCLUSION

This paper shows that the Triple Space Computing paradigm can be adapted to constrained devices and therefore achieve application level interoperability. A RESTful interface associated to a fully distributed middleware for knowledge sharing among heterogeneous devices has been proposed. Different implementations of such middleware for various embedded platforms have been generated and evaluated. Concretely, the designed HTTP based TS API has been used in 4 different platforms, showing the adaptability of the proposed middleware to coordinate the operation of hardware of different nature.

Besides, in order to emphasize the advantages of this middleware in comparison to traditional syntactic WoT solutions, two scenarios which can transparently interoperate through standard ontologies and TS have been presented. The underlying idea is that even if for small scenarios using semantic data may appear complex, the richer descriptions are beneficial in the long term when more and more applications share and exploit their knowledge in spaces, in a cooperative manner. From the point of view of the developer, instead of using middleware to access each sensor data, they now use a middleware through which they query a conceptual space for the desired knowledge, no matter which device provides it. Still, each device can be treated as a browsable WoT device which serves semantic content to any web application.

For our future work, we are planning to further test the impact and benefits of the inference process on resource con-

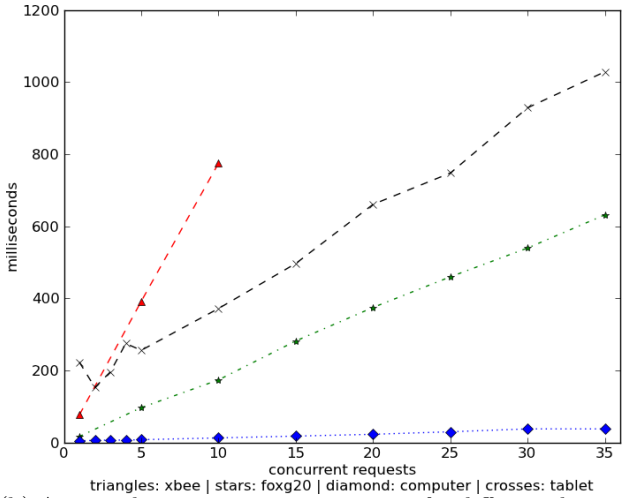
[13] http://semanticweb.org/wiki/RDF2Go

[14] http://code.google.com/p/android-sparql

[15] http://code.google.com/p/androjena

Concurrent requests	XBee	FoxG20	Regular computer	Samsung Galaxy tab
1	77 (1)	17 (0)	5 (1)	223 (349)
5	392 (8)	97 (16)	8 (4)	256 (76)
10	775 (8)	174 (28)	13 (8)	372 (171)
15	-	282 (43)	18 (13)	497 (191)
20	-	375 (30)	23 (13)	661 (444)
25	-	460 (30)	30 (18)	748 (288)
30	-	540 (35)	38 (22)	929 (805)
35	-	632 (29)	38 (20)	1029 (672)

(a) Mean of the measurements taken in different devices with the standard deviation (σ) in parenthesis (milliseconds).



(b) Averaged response time comparison for different devices (milliseconds).

Figure 2: Response times measured in different embedded devices.

strained devices. Particularly, we aim to propose a template-based strategy to reduce the number of times the reasoning process is triggered in each device.

7. ACKNOWLEDGMENTS

This work has been supported by research grants TSI-020301-2009-27 (ACROSS), funded by the Spanish Ministerio de Industria, Turismo y Comercio; TIN2010-20510-C04-03 (TALIS+ENGINE project), funded by the Spanish Ministry of Science and Innovation and IE11-316 (FUTURE INTERNET II project), funded by the Basque Government (ETORTEK 2011).

8. REFERENCES

[1] P. Costa, L. Mottola, A. Murphy, and G. Picco. Programming wireless sensor networks with the teeny lime middleware. *Middleware 2007*, pages 429–449, 2007.

[2] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, 1999.

[3] A. Gómez-Goiri and D. López-de Ipiña. On the complementarity of triple spaces and the web of things. In *Proceedings of the Second Intl Workshop on Web of Things*, WoT '11, pages 12:1–12:6, New York, NY, USA, 2011. ACM.

[4] A. Gómez-Goiri and D. López-de Ipiña. Assessing data dissemination strategies within triple spaces on the web of things. In *Proceedings of the Intl Workshop on Extending Seamlessly to the Internet of Things*, esIoT, 2012 (to be published).

[5] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the internet of things to the web of things: Resource oriented architecture and best practices. In *Architecting the Internet of Things*. Springer, 2011.

[6] A. G. Hernández and M. N. M. García. A formal definition of RESTful semantic web services. In *Proceedings of the First Intl Workshop on RESTful Design*, pages 39–45, New York, USA, 2010. ACM.

[7] G. Hohpe, B. Woolf, and K. Brown. *Enterprise integration patterns*. Addison-Wesley, 2004.

[8] J. Honkola, H. Laine, R. Brown, and O. Tyrkko. Smart-m3 information sharing platform. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 1041–1046. IEEE, 2010.

[9] G. Moritz, E. Zeeb, S. Pruter, F. Golatowski, D. Timmermann, and R. Stoll. Devices profile for web services and the REST. In *Industrial Informatics, 8th IEEE Intl Conf on*, pages 584–591, 2010.

[10] L. J. Nixon, E. Simperl, R. Krummenacher, and F. Martin-Recuerda. Tuplespace-based computing for the semantic web: a survey of the state-of-the-art. *The Knowledge Engineering Review*, 23(02):181–212, 2008.

[11] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceeding of the 17th Intl Conf on World Wide Web*, pages 805–814, 2008.

[12] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained application protocol (CoAP). https://datatracker.ietf.org/doc/draft-ietf-core-coap/, 2012.