Collaboration of Sensors and Actuators through Triple Spaces

Aitor Gómez-Goiri, Pablo Orduña, David Ausín, Mikel Emaldi and Diego López-de-Ipiña
Deusto Institute of Technology DeustoTech, University of Deusto
Avda. Universidades 24, 48007 Bilbao, Spain
Email: {aitor.gomez, pablo.orduna, david.ausin, m.emaldi, dipina} @deusto.es

Abstract— **In recent years, projects and initiatives under Internet of Things have focused mainly on establishing connectivity in a variety of challenging and constrained networking environments. Hence, a promising next step should be to build interaction models on top of this network connectivity and thus focus on the application layer, i.e. how to achieve useful aggregated functionality out of these Internet-connected ecosystems of sensors and actuators. This work analyses the adoption of Triple Spaces coordination language by very heterogeneous and resource-constrained devices and outlines how its primitives can help to develop fully distributed and very decoupled scenarios.**

I. Introduction

Currently, the Internet of Things (IoT) is becoming increasingly present thanks to the growing connectivity of everyday objects which embed sensors and actuators. So far, most initiatives have addressed connectivity issues in constrained networking environments. Therefore, little attention has been put on the interaction models which can help to achieve useful aggregated functionality out of these devices.

A successful achievement of the IoT vision will only be accomplished if those objects intelligently collaborate. A prerequisite to allow this is to understand the information they exchange in the form of services (functionality) and data (normally information of their surrounding context). A well-know current technique useful to provide data interoperability is annotating with semantic metadata this information to increase its expressivity. This semantic approach is used in the solution presented in this work which aims to progress from the current internet-connected objects that behave as isolated islands of functionality towards smarter internet connected objects that collaborate autonomously or under user intervention. To enable this collaboration, a new paradigm, namely Triple Space (TS), based on a shared storage area distributed among the objects and some coordination primitives to read and write data into it, is used. Remarkably, we have adopted a RESTful architectural style in TS to ensure its applicability in embedded devices and its compatibility with the *Web of Things* (WoT) emerging initiative [1].

II. Related work

Many different styles can be used to integrate applications depending on application requirements. Hohpe et al.[2] 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 to ensure the interoperability of different styles, as in the well known set of WS-*standards and most of services following the REST style. While the first one can be seen as both RPC or message exchange substyle, the second one defines an elegant way to define resource-oriented interfaces which comply with the RPC idea of isolating application from exchange particularities [3]. The use of both substyles in resource-constrained devices is represented by *Device Profile for Web Services* (DPWS) [4], based on SOAP, and the WoT [1].

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 so-called tuples in a space, possibly distributed over different nodes. This paradigm has been implemented to be run in embedded and mobile devices [5] and in conjunction with the Semantic Web [6]. In contrast with traditional integration approaches, using the Semantic Web, it 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 (TSC), which uses similar primitives with RDF triples as exchanged units. Although different Triple Spaces implementation exist, none of them has been specifically designed to be run in devices with constrained capabilities apart from our solution [7].

III. A TS API over HTTP

The Triple Space (TS) paradigm basically consists on writing and reading triples grouped in graphs in different spaces. Our implementation of TS does this by allowing each distributed node, no matter how complex or simple it is, to manage its own information and to establish a communication channel with the space it wants to join to, i.e. with each of the nodes belonging to it. Queries are propagated to other nodes which previously joined that space (regardless of who they are at each time) and possible responses are received from them using the same communication channel.

TABLE I

EXAMPLES OF REST ACCESS TO TS. *sp* IS A SPACE URI, *g* IS A GRAPH URI, *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}/query/wildcards/{s}/{p}/{o-uri} http://nodeuri/{sp}/query/wildcards/{s}/{p}/{o-type}/{o-value}	`query({sp},"{s} {p} {o} ."): triples`
GET	http://nodeuri/{sp}/graphs/{g}	`read({s},{g}): triples`
GET	http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-uri} http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-type}/{o-value}	`read({sp},"{s} {p} {o} ."): triples`
DELETE	http://nodeuri/{sp}/graphs/{g}	`take({sp},{g}): triples`
DELETE	http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-uri} http://nodeuri/{sp}/graphs/wildcards/{s}/{p}/{o-type}/{o-value}	`take({sp},"{s} {p} {o} ."): triples`

Remarkably, TS guarantees four basic types of autonomies: location autonomy (information providers and consumers are independent from where the data is stored), reference autonomy (nodes do not need to know each other), time autonomy (they communicate asynchronously) and data schema autonomy (it follows the RDF specication making it independent of nodes internal data schema). However, TS-based solutions are dependent on the ontology which defines the exchanged information or knowledge, i.e. they present data coupling. As we will explain below, the operations or primitives which can be issued over a Triple Space are independent of the semantic data being shared.

Anyhow, nodes will only manage to collaborate among them if they share a common pre-defined semantic vocabulary. This is a common problem in highly decoupled communications, such as Wire Admin Service [8] in OSGi, but in TS-based solutions the information stored can be expanded by a Semantic Reasoner. Even if at application level the data provider writes a small set of triples, these set can be expanded with inferred information and other nodes can perform queries on that information. Therefore, the amount of possible different requests that consumers can create are automatically 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*.

The most important primitives of our solution are *write*, *query*, *read* and *take*. *Write* adds new knowledge to a space writing together the given triples in a new RDF graph and returning its identifying URI. *Query* returns the triples which match a given template conceiving the space as a whole no matter to which graph they belong to. *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. The template used in these primitives can be as simple as a triple pattern with wildcards (e.g. ?s) or as complex as an SPARQL query, but in this work we have just considered the first ones.

When writing data into a space a technique known as *negative broadcasting* is used. It implies that all write operations are executed locally at the node, but all read and query operations are propagated to other nodes of a space. It suits perfectly to use cases where nodes create and manage their own information, such as a node in a mobile phone maintaining a user profile or embedded sensors managing their own generated data. This way, the primitives grant that there will not be concurrency issues attributed to shared information. Anyhow, this work will not cover the discovery process between different nodes.

As 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 [9]. More specifically, the proposed mapping between both styles is summarized in Table I and it is implemented in the Open Source project *Otsopack*[1], which adds other features not covered in this contribution such as deployment, discovery or security ones.

IV. STEREOTYPICAL SCENARIO

Two stereotypical scenarios for home automation had been devised to assess our collaborative middleware proposition for heterogeneous resource-constrained devices. In both scenarios the emphasis has been put in how those devices can coordinate in a decoupled mode thanks to it following the master-worker pattern [10].

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

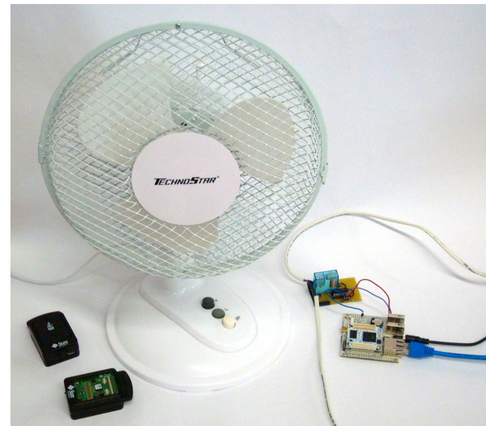


Fig. 1. FOX Board G20 connected to a fan and the SunSPOT, one of the devices responsible for capturing the temperature.

On the one hand, a room has been populated with several kind of sensors such as Oracle's SunSPOTs[2], XBee sensors with an IP gateway[3], the sensors on a KNX20 domotic bus and a fan connected to a FoxG20[4] embedded platform to act as an actuator (see Figure 1). Besides, an Android application has been 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 fulfil and it understands and consumes them with a *take primitive*.

On the other hand, a message delivery system has been designed using TS in order to avoid the sedentary lifestyle of a certain user by giving him different warnings. Taking into account the expected steps which should have been completed in each moment of the day (10.000 steps are recommended in average for an adult), different priority level messages are created to warn the user about his situation. To figure out this, a master node reads the number of steps covered by a user in that day[5] and his age both from the TS node deployed on his Android phone. To achieve it, the semantic information is used to find the accelerometers embedded in a user mobile's and his age. Depending on the priority of the messages, different devices which belong to that user, defined in the ontology, look for those messages. If the priority level is low, the user can be warned in a less intrusive way than if the priority is high. Hence, room's light brightness can be increased for low priority notifications, a chumby for normal priority ones or the message is shown directly in his mobile phone when the user should have covered many more steps than he has walked (high priority).

V. Evaluation

In order to prove that the proposed solution is light enough to run in small devices, we have measured the time taken in two of the devices detailed in the previous section: XBee and FoxG20.

A. Measuring times in 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-.

[2]http://www.sunspotworld.com
[3]http://www.digi.com/products/wireless-routers-gateways/gateways/connectportx2gateways.jsp
[4]http://www.acmesystems.it
[5]http://code.google.com/p/pedometer/

TABLE II
Measurements FoxG20 (milliseconds)

	REST			SOAP		
Concurrent requests	Mean	Std dev (σ)	Median	Mean	Std dev (σ)	Median
1	17	0	17	49	0	49
5	97	16	97	253	34	245
10	174	28	175	513	65	513
15	282	43	278	814	102	776
20	375	30	366	1021	149	997
25	460	30	455	1287	77	1287
30	540	35	539	1552	109	1539
35	632	29	627	1748	124	1751

So as to compare the approach taken of relying on REST with a possible SOAP based approach, we reimplemented the operations in SOAP[6] using the *SOAPpy* library. However, we could not make the SOAP version run in the device because it could not even load the required SOAP and XML libraries, even after stripping away those files in these libraries that were not required. XBee took around 77 milliseconds to return a response, and a mean of 775 milliseconds if 10 clients are performing concurrent requests, which is affordable.

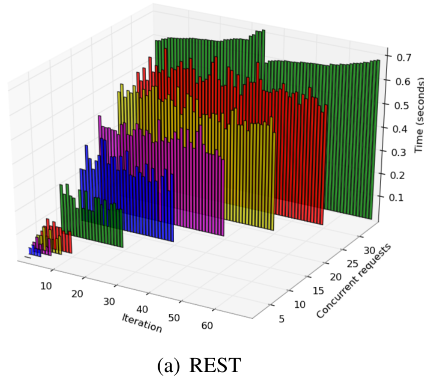
B. Measuring times in FoxG20

Given that the Python implementation of the Otsopack server was lighter than the regular Android/Java SE version of Otsopack, we used it again to measure the FoxG20. FoxG20 is a more powerful platform than XBee. In fact, it can even perform OWL reasoning in Python using the RDFClosure library [7].

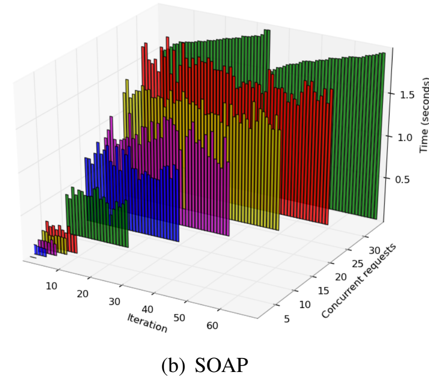
Using FoxG20, we could run it with 35 concurrent users both using the REST and the SOAP version. As seen in Table II, with SOAP it takes 2.6-3.0 times more than REST attending to the means. Figure 2 presents the full distribution using REST -Figure 2(a)- and SOAP -Figure 2(b)-. In this figure, every number of concurrent requests has been tested twice. Therefore, with 1 concurrent request there are 2 measures (iterations), while with 10 concurrent requests there will be 20 measures.

The SOAP version is the one detailed in the previous subsection, developed in Python and using *SOAPpy*. Given that FoxG20 can run Java SE and C code, it would be possible to measure other SOAP implementations that would be faster than our REST implementation. This would also require to redevelop Otsopack in those languages to make a fair comparison. However, the focus of this evaluation is to show that the proposed approach is light enough to be run in small sensors using standard libraries. Furthermore, the performance difference between SOAP and REST has already been addressed in the literature [11].

[6]DPWS could not be tested because no Python implementation is available.
[7]http://www.ivan-herman.net/Misc/2008/owlrl/



(a) REST



(b) SOAP

Fig. 2. Measures of the REST and SOAP versions in FoxG20 in milliseconds. Note that the time scale is different.

TABLE III
QUALITATIVE ANALYSIS OF OTSOPACK

	Java SE	Android	FoxG20	XBee
Platform version	Java SE 6.0	Android 2.2	Python 2.5	Python 2.4
REST Libraries	Restlet	Restlet	Python Standard Library	Python Standard Library
Semantic libraries	Rdf2Go	Microjena	RDFClosure	None

C. Summary of evaluation

The proposed TS API over HTTP is a lightweight technology, tested successfully in a range of devices. The response times are small.

In Table III, we present a qualitative analysis of the different Otsopack technologies used in the scenario. As detailed in the table, we have aimed four different architectures (Java SE, Android, FoxG20 and XBee) using two programming platforms (Java 6 + Restlet and Python 2.4 with the Python Standard Library). Attending to these platforms, different reasonement levels can be achieved: Rdf2Go[8] in Java SE encapsulates different engines; Microjena[9] in Android supports RDFS and performs better than AndroJena[10], which additionally supports OWL; RDFClosure supports OWL, and no reasoning engine could be executed in the XBee. However, those nodes which are only sensors can be programmed to infer the information they store manually, given that they only handle few triples.

VI. CONCLUSION

This paper shows how adopting HTTP for the communication layer and REST architectural style to define the access to Triple Space Computing primitives, can ease its adoption in very heterogeneous devices with limited computing capabilities. Besides, two scenarios in which each device can focus on achieving its own goal in a very decoupled way is presented.

[8]http://semanticweb.org/wiki/RDF2Go
[9]http://poseidon.elet.polimi.it/ca/?page_id=59
[10]http://code.google.com/p/androjena/

As can be appreciated, the designed HTTP based TS API has been used in 7 different kind of devices over two different scenarios, showing that the proposed TS API is a lightweight technology. For our future work, we are planning to further test the solution in mobile phones, specifically addressing the impact and benefits of the inference process on them.

ACKNOWLEDGMENT

This work has been supported by research grants TIN2010-20510-C04-03 (TALIS+ENGINE project), funded by the Spanish Ministry of Science and Innovation and TSI-020301-2009-27 (ACROSS project), funded by the Spanish Ministry of Industry, Tourism and Commerce.

REFERENCES

[1] 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, May 2011.

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

[3] 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 Conference on World Wide Web*, 2008, pp. 805–814.

[4] G. Moritz, E. Zeeb, S. Pruter, F. Golatowski, D. Timmermann, and R. Stoll, "Devices profile for web services and the REST," in *Industrial Informatics (INDIN), 8th IEEE Intl Conference on*, 2010, pp. 584–591.

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

[6] 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*, vol. 23, no. 02, pp. 181–212, 2008.

[7] A. Gomez-Goiri, M. Emaldi, and D. López-de-Ipiña, "A semantic resource oriented middleware for pervasive environments," *UPGRADE journal*, vol. 2011, Issue No. 1, pp. 5–16, Feb. 2011.

[8] O. Alliance, "Osgi service platform release 4 version 4.2 compendium specification," pp. 193–237, 2009.

[9] A. Gómez-Goiri and D. López-de Ipiña, "On the complementarity of triple spaces and the web of things," in *Proceedings of the Second International Workshop on Web of Things*. New York, NY, USA: ACM, 2011, pp. 12:1–12:6.

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

[11] D. Yazar and A. Dunkels, "Efficient application integration in ip-based sensor networks," in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, 2009, p. 4348.