CEPIS UPGRADE is the European Journal for the Informatics Professional, published bi-monthly at <http://cepis.org/upgrade>

Publisher
CEPIS UPGRADE is published by CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>), in cooperation with the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática,* <http://www.ati.es/>) and its journal Novática

CEPIS UPGRADE monographs are published jointly with Novática, that published them in Spanish (full version printed; summary, abstracts and some articles online

CEPIS UPGRADE was created in October 2000 by CEPIS and was first published by Novática and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>)

CEPIS UPGRADE is the anchor point for UPENET (UPGRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
- inforewiew, magazine from the Serbian CEPIS society JISA
- Informatica, journal from the Slovenian CEPIS society SDI
- Informatik-Spektrum, journal published by Springer Verlag on behalf of the CEPIS societies GI, Germany, and SI, Switzerland
- ITNOW, magazine published by Oxford University Press on behalf of the British CEPIS society BCS
- Mondo Digitale, digital journal from the Italian CEPIS society AICA
- Novática, journal from the Spanish CEPIS society ATI
- OCG Journal, journal from the Austrian CEPIS society OCG
- Pliroforiki, journal from the Cyprus CEPIS society CCS
- Tölvumál, journal from the Icelandic CEPIS society ISIP

Editorial correspondence: Llorenç Pagés-Casas <pages@ati.es>
Advertising correspondence: <info@cepis.org>

Subscriptions
If you wish to subscribe to CEPIS UPGRADE please send an email to info@cepis.org with 'Subscribe to UPGRADE' as the subject of the email or follow the link 'Subscribe to UPGRADE' at <http://www.cepis.org/upgrade>

ISSN 1684-5285

Monograph of next issue (April 2011)
"Software Engineering for e-Learning Projects"
(The full schedule of **CEPIS UP**GRADE is available at our website)

CEPIS UPGRADE
The European Journal for the Informatics Professional
http://cepis.org/upgrade
Vol. XII, issue No. 1, February 2011

Monograph
Internet of Things
(published jointly with Novática*)
Guest Editors: *Germán Montoro-Manrique, Pablo Haya-Coll, and Dirk Schnelle-Walka*

UPENET (UPGRADE European NETwork)

CEPIS NEWS

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>.

A Semantic Resource-Oriented Middleware for Pervasive Environments

Aitor Gómez-Goiri, Mikel Emaldi-Manrique, and Diego López-de-Ipiña

Pervasive environments are highly dynamic with lots of heterogeneous devices which share information through increasingly interconnected networks. In this context semantic models can be used to describe the context that surrounds them in a very expressive manner, usually stored in centralized knowledge bases. The applications built on top of these knowledge bases use heterogeneous protocols to transmit their data, but do not capture the dynamicism of the network. The presented middleware facilitates the exchange of knowledge between different sensors and actuators in a highly distributed, decoupled and resource-oriented manner following the Triple Space paradigm. This middleware has been tested on a stereotypical scenario, which illustrates how different peers can exchange data whilst keeping them autonomous and yet with a reasonable footprint for devices with reduced computational capabilities.

Keywords: Distributed, Embedded, Mobile, Semantics, Triple Space.

Authors

Aitor Gómez-Goiri is a PhD candidate from the University of Deusto, Spain, who works at the INTERNET unit within DeustoTech – Deusto Institute of Technology. His research focuses on designing semantic middleware suitable for mobile and embedded devices. <aitor.gomez@deusto.es>.

Mikel Emaldi-Manrique is a research intern at Deustotech - Deusto Institute of Technology, Spain. His research focuses on developing integration middleware for embedded devices. <m.emaldi@deusto.es>.

Diego López-de-Ipiña holds a PhD in Engineering from the University of Cambridge, United Kingdom. He is the Principal Researcher of the INTERNET unit within DeustoTech – Deusto Institute of Technology, Spain. His research focuses on applying middleware and web techniques in order to more easily enable Intelligent Environments. <dipina@deusto.es>.

1 Introduction

In context-aware environments, a lot of devices communicate with each other and share changes of state in order to trigger actions within the environment. Different approaches to modelling and storing context data have been presented in several works [1], coming to the conclusion that ontology-based models are the most expressive models and fulfil most of the requirements of these environments. The blackboard model is one of the main context management models [2] which post messages into a shared media, usually centralized in a server.

Triple Space computing is a coordination paradigm based on tuplespace-based computing, which comes from the parallel computing language Linda [3]. In tuplespace computing the communication between processes is performed by reading and writing data structures in a shared space, instead of exchanging messages. The Semantic Web vision aims to offer machine-understandable persistent data forming a knowledge network for machines instead of the current World Wide Web which is more human-centered and require user intervention (web services offer remote functionality to machines, but they are not really Web-based since they are driven by message exchange). Triple Space (TS) computing performs a tuplespace based communication using RDF triples, in which the information unit has three dimensions: "subject predicate object", to express this semantic data. TS offers reference autonomy (processes can communicate without knowing anything about each other), time autonomy (because of the asynchronous communication) and space autonomy (processes can be executed in very different computational environments), which cannot be achieved by message exchange-driven communication.

In this paper middleware for pervasive environments which uses the blackboard model through a Triple Space decentralized implementation is presented. This middleware is specially designed to run over devices with limited computational resources and embedded devices which may be part of the Internet of Things where common objects share their contextual information on the network.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents our middleware. Section 4 details an experimental environment. Section 5 examines the results of using the proposed solution in the experimental environment. Finally, Section 6 concludes and outlines the future work.

2 Related Work

Several approaches exist in the field of semantic tuplespace [4]. Conceptual Spaces, or cSpaces, were born to study the applicability of semantic tuplespaces to different scenarios including ubiquitous computing. Semantic Web Spaces propose some new primitives defining two different data coordination views: data view (with syntacti-

> “Pervasive environments are highly dynamic with lots of heterogeneous devices which share information through increasingly interconnected networks”

cally valid RDF and Linda primitives) and information view (with consistent and satisfiable data and new primitives). sTuples was conceived by Nokia Research Center as a pervasive computing work and provides description logics reasoning and a semantic extension of JavaSpace tuplespace middleware.

None of this projects was fully distributed and they were deployed over more or less a client-based architecture which did not implement the tuplespace paradigm itself in mobile peers and which restricted the reasoning process to few powerful devices.

In Triple Space Computing the tuples are expressed in the form of triples. Currently, two main pure Triple Space Computing middleware implementations exist: tsc++ and TripCom.

TripCom has different kernels hosted in servers which can distribute the semantic data through themselves, but once again, is too server centered. TripCom clients are not part of the space and they could hardly be, because of the complexity of this software which is oriented to run on powerful machines (it is designed to be able to run even different modules of the same kernel in different machines).

The first Triple Space project was called TSC. In TSC, triples can be interlinked to form graphs and semantic algorithms are implemented for template matching. It also offers a transactional context and a simple form to publish and subscribe to certain patterns. tsc++ [5] is a new version of the former TSC project [6] which basically offers the same API in a distributed way. To do that, tsc++ uses Jxta Peer To Peer framework to perform the coordination and Sesame [7] and Owlim [8] to store triples of each peer.

The nodes in tsc++ not only can query the space, but they can also store their own information, enabling the distribution of the space over all the peers by means of the strategy known as negative broadcast. In negative broadcasting the nodes of a group write the information locally and read querying to the rest of the nodes. This seems to adapt to ubiquitous system, where different devices share heterogeneous data entering and leaving the system, compromising data consistency and availability. In this aspect a sensor can provide information, but when it leaves the space, its information is automatically removed from there since it is no longer available to the rest of the nodes. However, as will be explained below in Section 3.4, negative broadcasting needs to be adapted to those cases in which a device wants to remotely change the state of actuators managed by another device.

Nevertheless, tsc++ lacks some of the advantages of other alternatives: it does not make inferences, it does not allow expressive querying and last but not least, it has not been designed for devices with reduced computing capabilities, because tsc++ middleware is focused on architecture and implementation in large scale and we focus on small scale aspects (local area networks with an intelligent environment).

Our work also aims to build an Internet of Things where everyday objects have connectivity and share their data through it. Thus, some of our work aspects have been presented in other solutions which belong to this area such as [9] or the Web of Things (WoT) paradigm [10]. The first one describes how the Jxta framework can facilitate the communication between objects with connectivity capabilities, and even if our solution uses Jxta as communication protocol, it is more centered on sharing knowledge rather than on the underlying layers. The second solution advocates the convenience of making objects part of the web to use existing web techniques to create applications and therefore they focus on embedding web servers on them. Despite the simplicity of this approach, there are certain aspects which are not considered as the discovery method of new devices (other works try to correct this limitation of the web approach as [11]), the instability of the nodes which serve webs in a network or the use of semantics.

As it has been seen, even if some works have analysed the convenience of the semantic tuplespace approach in Ubiquitous computing, to the best of our knowledge TS has never been specifically designed and implemented to use mobile and embedded devices as another peer of these spaces and not only as simple clients. This approach allows heterogeneous devices communicate with each other limiting the necessity of fixed infrastructure and previous configuration enabling more dynamic environments.

3 Infrastructure Description

Our main goal was making Triple Space middleware suitable for pervasive environments. To do this, it should enable the communication between devices of heterogeneous nature (mobiles, embedded devices, PDAs, Tablets, even PCs) through different communication links using standard

> “In context-aware environments, a lot of devices communicate with each other and share changes of state in order to trigger actions within the environment”

Space management	createSpace(space) joinSpace(space) leaveSpace(space)
Querying	query(space,template): triples read(space,graph): triples read(space,template): triples take(space,graph): triples take(space,template): triples queryMultiple(space,sparql): triples
Writing	write(space,triples): URI demand(space,template,timeout)
Subscriptions	subscribe(space,template,listener): URI unsubcribe(space,URI) advertise(space,template): URI unadvertise(space,URI)
Services	register(space,service) unregister(space,service) invoke(space,invocation,listener): URI

Table 1: Primitives of the designed Triple Space Implementation grouped by their Nature. *space* is the URI which identifies the knowledge of a group of nodes, *graph* is an URI which identifies a set of triples written into the space, *triples* is a set of triples and *template* expresses a sequence of adjacent triple patterns which specify WHERE-clauses of SPARQL queries. *Service* and *invocation* are interfaces which express the data needed to define a service and its invocation.

communication protocols and also still be connected to Internet. These devices would communicate though a push-and-pull process using semantic and in a very decoupled fashion.

So, as one of the main concerns was allowing mobile and embedded devices to run that Triple Space middleware, effort was put into developing a Java-embedded distributed triple space implementation, namely *tscME*, which was still compatible with tsc++ [5]. It has also been developed a module to allow embedded devices such as SunSPOTs or XBee sensors which do not support IP stack be part of that space through a gateway. Finally, the former Triple Space API adopted from [5] was also enhanced to offer more expressive queries which are spread through all devices within the space, service management primitives and a new writing approach.

In the following sections the API and the different stages of the implementation of the middleware are described.

3.1 API

In order to interact with the Triple Space, an API is offered to the developer. Although it was originally inspired in the tsc++ API, it has been adapted taking into account the nature and restrictions of pervasive environments. It is composed by several primitives which are presented in Table 1.

The space management primitives allow the nodes to share knowledge with different groups of nodes. *Query* retrieves all the triples which match an specific template in a given space, while *queryMultiple* divides a SPARQL query into templates which are sent to all the nodes, the responses obtained from those nodes are merged and the query is made again over them potentially obtaining new results. *Read* retrieves just one complete graph which contains at least a triple which matches the template and *take* does the same but it removes that graph from the semantic repository of

Figure 1: Subscribe and Advertise Use Example.

the node which had it. Both read and take are overloaded to retrieve an specific graph identified by its URI.

Although *write* primitive was used to write triples in a local repository returning the URI which identified the new graph in which they had been stored, its implementation has been modified not to always follow the negative broadcasting strategy. *Demand* lets the developer announce to other nodes that it will be responsible for the graphs which match the given template for a period defined by the timeout parameter. The solution will be explained further in Section 3.6, below.

Subscribe primitive is used to made a node aware of the notifications made over the given template or any particularization of it. *Advertise* lets to a node propagate a notification to all the nodes in an space to warn them about something. Those primitives can be used to build a notification system upon the space (see Figure 1).

Finally, a service API was conceived in top of other basic primitives as a first approach to solve the problems described in the Section 3.4 [12]. With *register* and *unregister* the notation of a service (inputs and outputs) is stored in a node and advertised to the rest of nodes. With *invocation* some inputs are written into the space and advertised to the service provider. The service provider can perform that invocation and, optionally, retrieve the output of the service. Most of the time, the output may imply changes in the knowledge base of the provider as a result of change in the context. This process will be explained in detail in Section 3.4, below.

3.2 1st Stage: tscME

The first step towards implementing our middleware was creating a library for a MIDlet which could be deployed on Java ME CLDC compliant JVM called tscME. This version provides space management, querying (except for *queryMultiple*), writing and subscription primitives in a native way. The communication has done using Jxme (Java ME's Jxta framework) and the semantic data was managed using Microjena [13].

Due to the fact that Jxme does not use multicast, an special Jxta element called Rendezvous must be used to propagate mobiles' messages. This element centralizes a little bit the solution and makes the mobile nodes dependent on them, but whenever multicasting is implemented on Jxme, this problem should be automatically avoided.

Since Jxta communication method was slightly different from the one used in tsc++ peers, a new communication layer was attached to those peers. To made subscriptions and advertising compliant with the method used by tsc++, a tsc++ node must act as a gateway for tscME nodes breaking the distribution principle in mobile nodes. Subscription

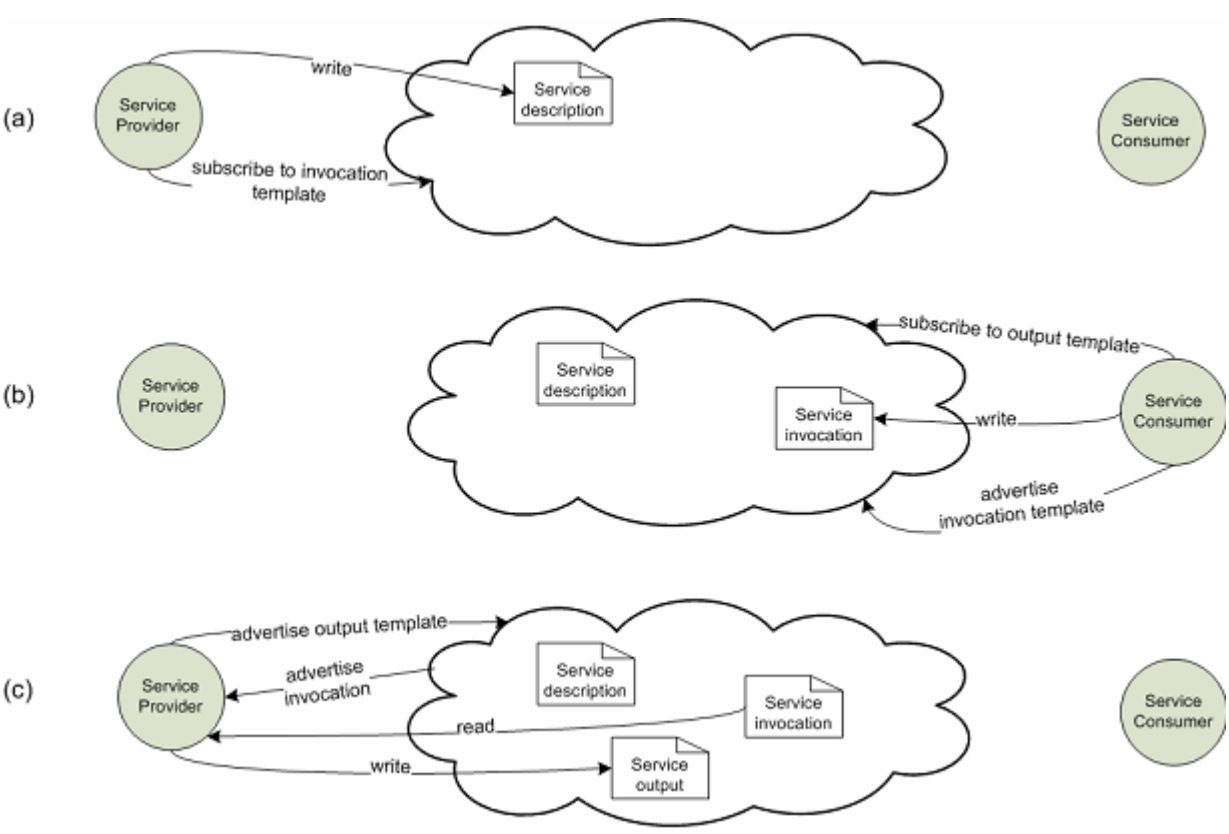

Figure 2: Services over tsc++: (a) registration, (b) invocation from the consumer point of view, and (c) invocation from the service provider point of view.

> “Our work also aims to build an Internet of Things where everyday objects have connectivity and share their data through it”

primitives are, however, not part of the Triple Space paradigm themselves.

Finally, the basic template has recently been improved, thus creating a new comparison template both in tscME and in the expanded tsc++ versions. This new template allows one to check whether a triple matches it by comparing its literal value.

Comparison Template Examples

The first example is equivalent to *? s ? p ? o* . The second one checks whether a triple has *myont:lenght* predicate and a numeric literal object not equal to 3. The last one checks if a triple has *:aitor* as the subject, *:has_age* as the predicate and a numeric literal object less than or equal to 30.

1. `[?s ?p ?o . , ?o <= ?lit .]`
2. `[?s myont:lenght ?o . , ?o != "3"^^xsd:int .]`
3. `[:aitor :has_age ?o . , ?o <= "30"^^xsd:int .]`

3.3 2nd Stage: tsc++ extension

As described above, to make tsc++ compatible with tscME a new communication layer was attached. Apart from that, a service API and queryMultiple was provided in these nodes. Service API is described in the following section.

QueryMultiple uses an SPARQL Construct query as input and splits it up into basic templates which are spread to the rest of nodes of the space. Then it filters all the received responses locally with the original query. Different strategies to propagate the queries have been proposed in [14] and [15], but due to the nature of negative broadcasting the nodes do not know anything about each other and therefore they cannot redirect those basic templates to an specific node.

Since the SPARQL query decomposition cannot be done in mobile peers, this primitive is not mandatory.

3.4 3rd Stage: Service API

The necessity of providing this service infrastructure in Triple Space or not could be argued since the knowledge can be directly obtained from the space or written into it, working with resources in a very RESTful way. Although in Section 3.6 a more resource-oriented approach is described, in [12] some primitives to use services inside Triple Space were proposed.

In pervasive environments, the sensed data can be obtained querying the space, but some limitations when modifying actuators were discovered:

- **Security.** Since tsc++ does not implement any kind of access control list, somebody might modify more knowledge than he or she wanted by mistake.
- **Concurrency.** If two different peers modify the same information at the same time, what information should be taken into account?
- **Location of the information.** Due to the nature of tsc++, when any information which initially belongs to *peer a* is modified by *peer b*, it is stored in *peer b* instead of being stored in *peer a*. If *peer b* leaves the space, some crucial information about the actuator will disappear. It seems logical that the information about a sensor or an actuator should be stored in the device which manages it (in the example, *peer a*).

Our service solution aims to provide this control to the device which controls the actuators being respectful of the asynchronous nature of TS. For that purpose, a really simple A Service invocation approach, which is very independent of the way the semantic services are defined (we use our own service definition language for the scenario, but other standard languages can be used), was designed. First, the service provider should register its service in the space (see Figure 2a). The consumer would discover it querying the space, and then it would create an *invocation* using the master-worker pattern and advertise it (see Figure 2b). An invocation is basically composed by an URI identifier and the input data the service may need.

The service provider, which is subscribed to its services invocation templates, will notice the event (see Figure 2c) and will retrieve the input data and perform the service (typically, performing a change in the environment using an actuator). When the invocation has been completed, the provider may write some output triples into the space and advertise the consumer in a similar fashion to the invocation.

3.5 4th Stage: Embedded Devices’ Gateway

In this stage, we adapted a gateway which enables the integration of SunSPOTs devices into TS. We interact with the SunSPOTs through a gateway designed to follow the WoT paradigm [10]. It provides access to them and theirs sensors and actuators through RESTful services (see Figure 3).

Besides the default representations provided by the server (XML, JSON and HTML), we have added another one which returns a set of triples with the semantic data corresponding to the resource identified by the user’s given

> “Our main goal was making Triple Space middleware suitable for pervasive environments”

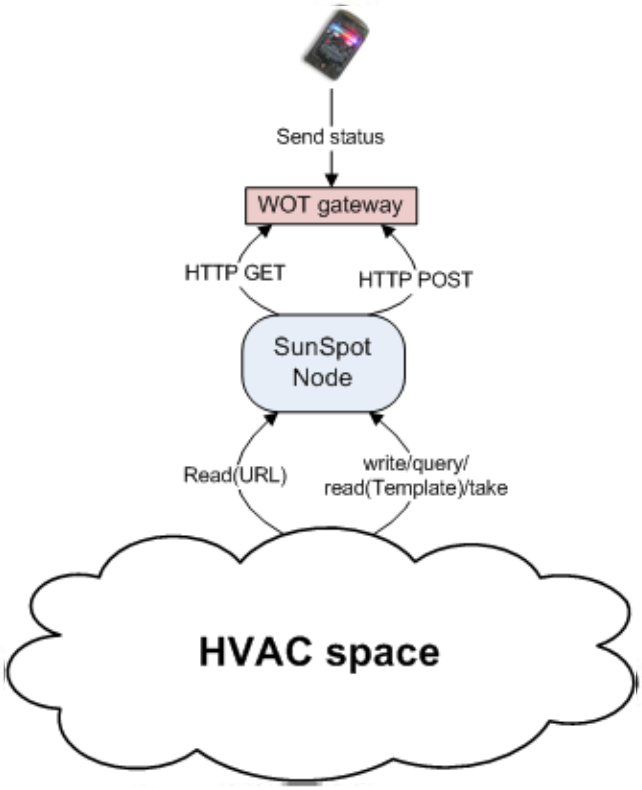

Figure 3: Message exchanging between the Nodes beyond the Space, the WOT Gateway and a SunSPOT.

URL. This representation allows the seamless integration of SunSPOT devices in the TS by simply replacing the layer which interacts with the semantic repository of any node with the one which translates TS primitives into HTTP calls addressed to the gateway. The translated primitives are *read*/*take*, *write* and *query*.

Read has two different implementations. The first one returns a graph identified by an URL requesting it to the gateway. For example, to get the graph which describes the temperature of a SunSPOT, the *read* primitive would be mapped to an HTTP GET to the following URL: `http://{host}:{port}/sunspots/{SpotName}/sensors/temperature/`

The second implementation, returns the first graph found in the gateway which has a triple that matches a given template. To do that, it requests the following URL passing the template and the space as parameter: `http://{host}:{port}/read?spaceURI={spaceURI}&template={"?s ?p ?o"}`

Internally, the gateway collects all currently available knowledge for SunSPOTs in a temporary repository and runs the template obtaining the desired result. The *take* primitive has been mapped to a *read* primitive since it does not make sense to remove a graph which is only generated on demand and therefore is not stored anywhere.

The *write* parses the contents of the triples passed to it extracting the values needed to make an HTTP POST request as follows: `http://{host}:{port}/sunspots/{SpotName}/leds/led{0-6}?switch={true/false}&redColor={0-255}&blueColor={0-255}&greenColor={0-255}`

The *query* gets all the triples that match a specific template in a given space. To do that, a GET request is addressed to a URL like `http://{host}:{port}/query? spaceURI={spaceURI}&template={"?s ?p ?o"}` and the server executes the template over all available graphs.

3.6 5th Stage: Remote Writing

Although negative broadcasting perfectly suited to scenarios where nodes frequently enter and leave the local net-

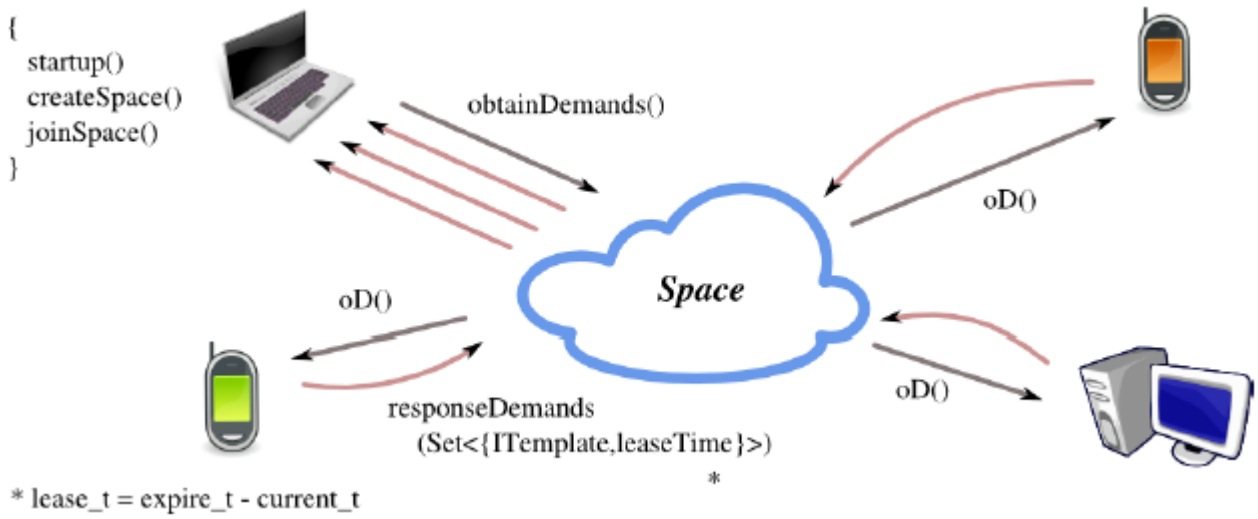

Figure 4: A Node which joins a Space asks to the Rest of the Members about the Demands they have received.

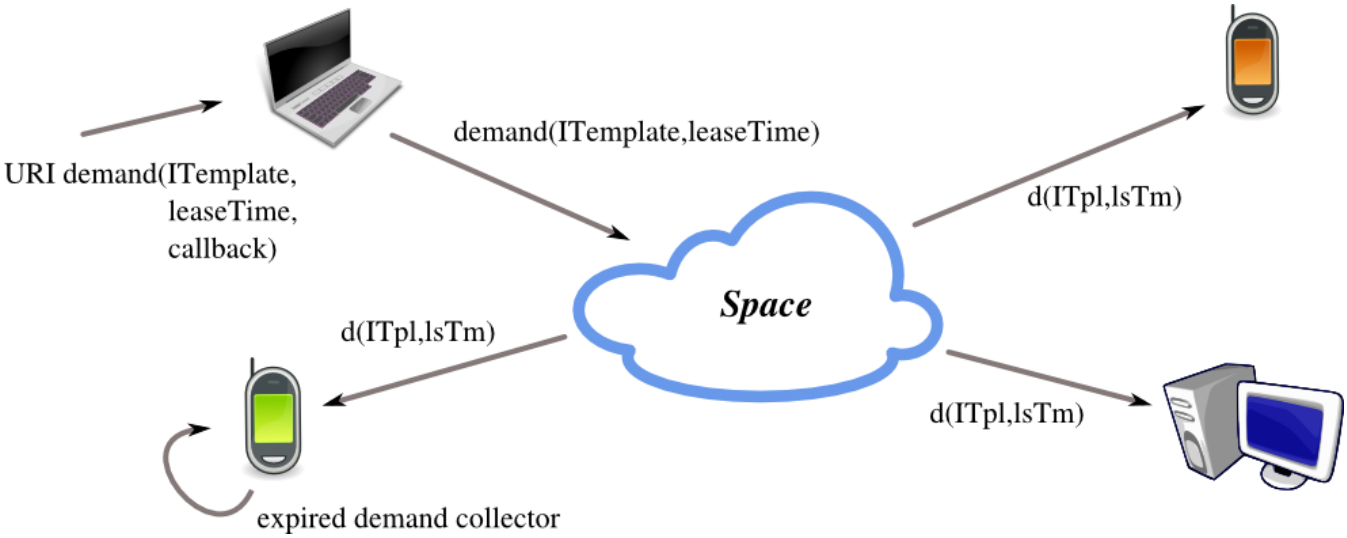

Figure 5: Each Hode sends Demands periodically to renew its Responsibility for an Item of Knowledge.

work sharing their own data (i.e. data from its sensors or the profile of the owner of the mobile phone), this approach presents a problem when a node changes the context of an actuator managed by another node (quite common on the Internet of Things). Those problems have already been discussed in Section 3.4.

The first approach to solve this problem was the use of services over Triple Space. The inherent problem with it is the same that exists between WS*-like web services and RESTful web services [16]: excessive complexity. As it was described in [17], Triple Space paradigm is basically a resource-oriented paradigm, and since all the primitives use semantic resources (triples) as basis, it seems logical to find a solution consistent with the paradigm triples to be used as base. To keep the API simple for the developer, we have modified the implementation of the *write* primitive in order for him not having to care about the write operation to be used.

In essence, a *demand* primitive has been created to allow each node to reflect its responsibility for a piece of knowledge by using a template. *Write* primitive has been overwritten to send such knowledge to other nodes of the space when the responsibility for a set of triples is known to belong to another node. See Figure 4.

The *demand* primitive has a maximum lifetime after which the nodes will remove such claim from their registers to avoid the indefinite accumulation of them. The nodes which join a space will interrogate others to get an idea of the responsibilities they have claimed.

Finally, the *write* will have the following behavior.

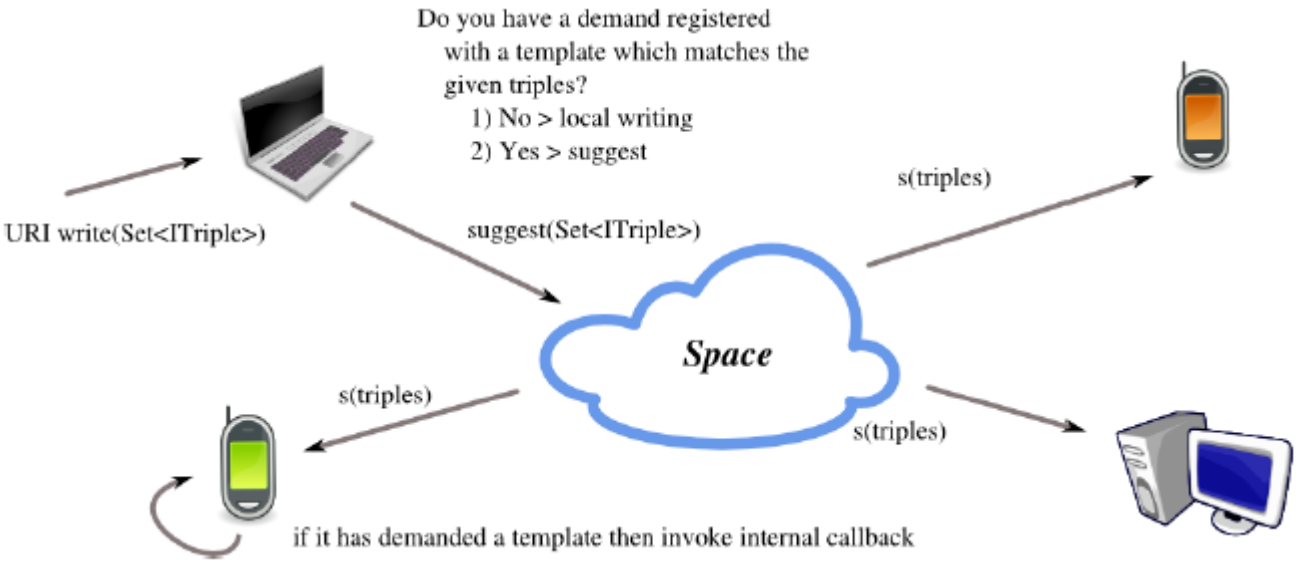

Figure 6: New Write Behavior.

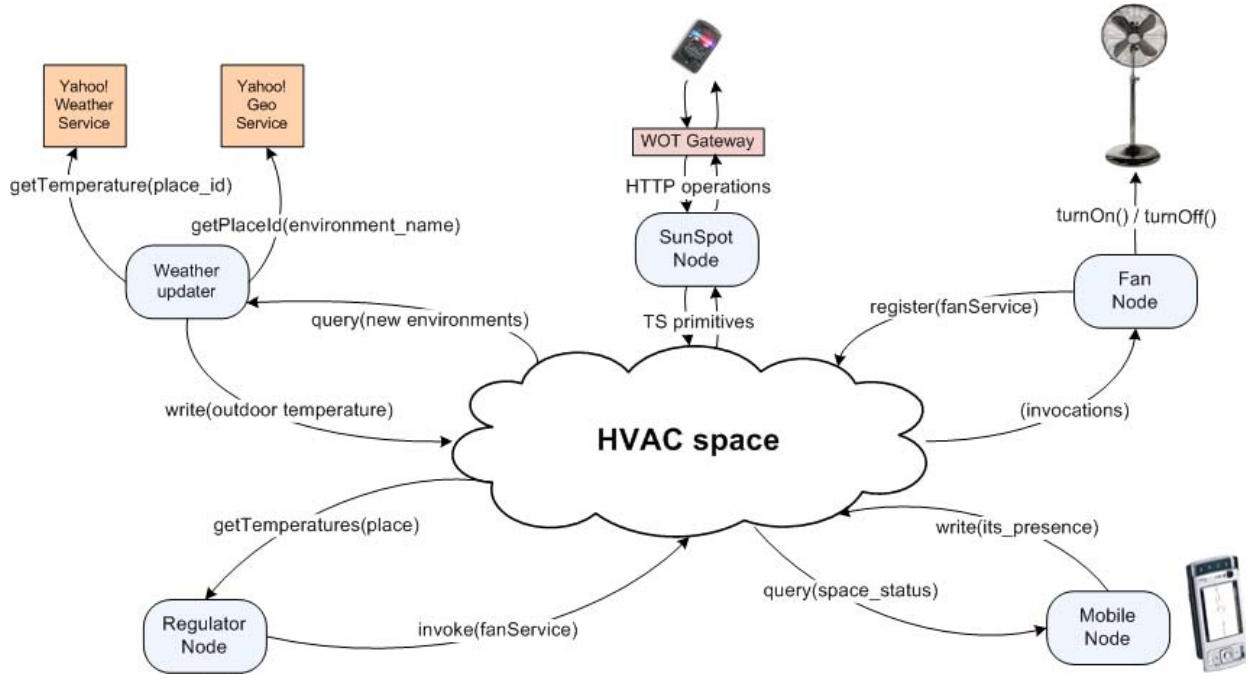

Figure 7: Implementation of the described Scenario using a Service-based Approach.

■ If a *node a* tries to write a set of triples and another node is known to be responsible for the knowledge which is *node a* trying to write (somebody has performed a *demand* with a template that at least one triple matches), it will be suggested to the other nodes that the *node a* wants to modify that knowledge with that set of triples. This is done using an operation transparent to the users of the middleware called *suggest* (see Figure 5).

■ Each node which has performed a demand that matches the given triples, will call a callback method passing the original set of triples to it. In this method, ideally the environment should be changed using an actuator and,

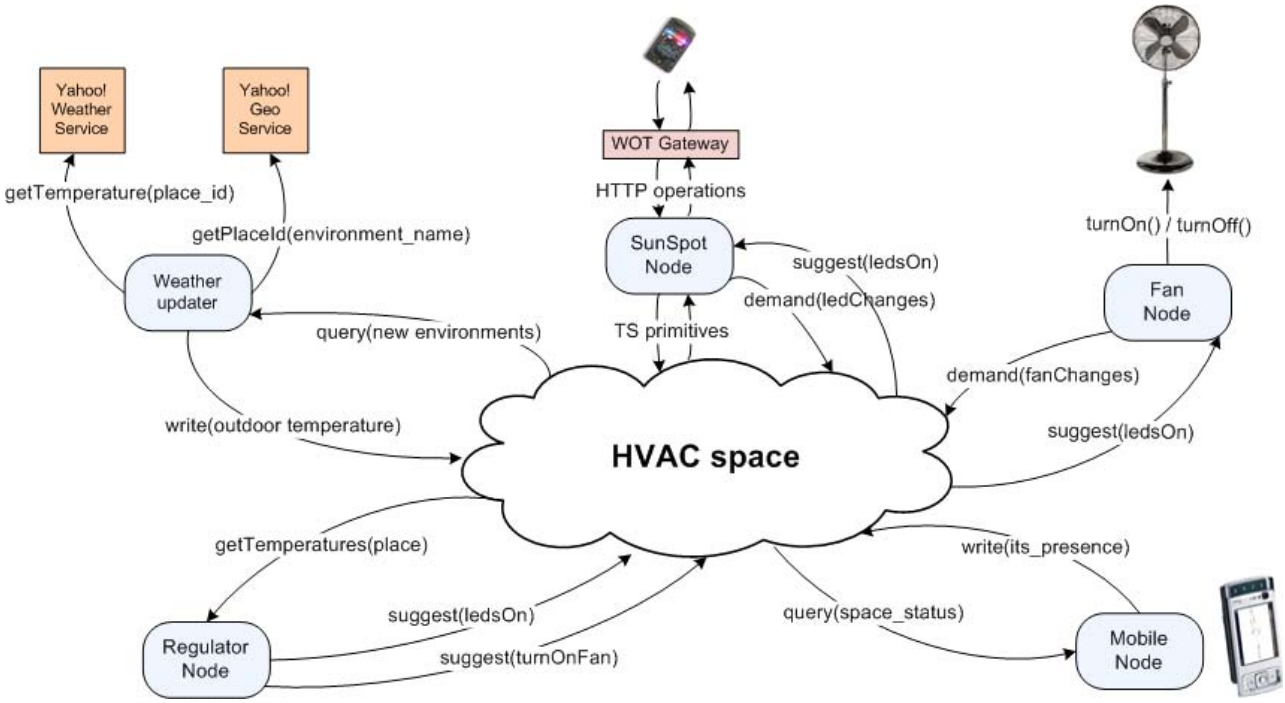

Figure 8: Implementation of the described Scenario using a Resource-based Approach.

Number of registered demands		10	20	30
Former version	write	0.004	0.004	0.004
Current version	demand	0.096	0.099	0.098
	write	0.006	0.015	0.027
	suggest	0.133	0.159	0.186

Table 2: Demand Primitive Performance in Mobile Node (in seconds).

if necessary, reflect that change in the knowledge base.

- If nobody claims responsibility for this knowledge, it will be written in the local semantic repository (see Figure 6).

4 Experimental Environment

There are many home automation or urban instrumentation scenarios where the proposed middleware could be used. One of these stereotypical cases could be the control of the room temperature. In this scenario, which uses all the primitives described previously, there are at least five peers, which are shown in Figures 7 and 8.

There are two different context providers: SunSPOTs and weather provider. The first types are physical devices which share their sensed data through the node described in Section 3.5. The weather provider, on the other hand, is a virtual context provider which gets information from the Internet and writes it as semantic information into the space. To do that, the peer polls for new environments in the space, and checks if Yahoo! has a place id for this environment name. If so, it queries the Yahoo! Weather service and it gets the current temperature in this location. The space has also an actuator device: an air conditioning system which will decrease the temperature. It can be turned on or off to try to regulate indoor temperature. Current temperature can be also monitored with a mobile phone.

The regulator node uses all the sensed information whenever someone is in a place (inferred when there is a device who belongs to somebody in the space) to decide when to cool it. To regulate the temperature, this peer checks whether there is an air conditioning system in the location it wants to regulate and turns it on.

Basic Indoor Temperature Control Algorithm

```
while( indoort>26 ) // unbearable temperature
if( outdoort<26 ) // user should open windows
else // turn on the air conditioning
```

In order to implement this scenario two different approaches can be used depending on what method they use to perform actions over the environment. The first approach (Figure 7) uses the described service primitives to create a service which turns on or off the connected air conditioning. Then the regulator node invokes it whenever it is necessary.

The second approach (Figure 8), the recommended one, relies in the demand primitive explained above. The SunSPOT node demands a template related with the LEDs of the controlled sunspots and the air conditioning node does the same with a template related with itself. When the regulator node decides that the indoor temperature should be decreased, it tries to write that knowledge on the space but since both SunSPOT and air conditioning nodes have claimed to be responsible of that knowledge the primitive is transformed on a *suggest* primitive. Then, SunSPOT and air conditioning nodes decide what to do with these information. In this example, SunSPOT node turns on or off all the LEDs of all the SunSPOTS connected to them using the gateway, so their status is automatically updated and the air conditioning node turns it on or off and updates its status.

This scenario is only a proof of concept. Technically, a SWRL (Semantic Web Rule Language) could be introduced to define the temperature control rules in a more expressive and decoupled way.

5 Experimentation

In this section, the new *demand* primitive in conjunction with the *write* primitive and the SunSPOT gateway will be evaluated. The rest of the primitives have already been assessed in [12] for tscME and in [5] for tsc++.

Number of graphs		10			
Number of Spots		1	2	3	4
SunSPOTs	write	0.037	0.036	0.045	0.037
	read(URL)	0.020	0.017	0.029	0.028
	read(template)	0.198	0.304	0.567	0.586
	take	0.164	0.253	0.557	0.597
	query	0.170	0.282	0.396	0.507

Table 3: WOT Gateway Performance (in seconds).

Finally, the stereotypical scenario presented in the previous section has been implemented using the *demand* primitive and will be compared with the service-based solution assessed in [12].

5.1 Demand Primitive

In Table 2 we can appreciate how the *demand* primitive behaves as a networking primitive and therefore it takes the time which is needed to propagate over the network until the other node is reached, and it may vary depending on the network status. The *write* is slightly penalized by the demands checking process, when the node decides whether it should write on the repository or propagate a *suggest* primitive, but anyway it is nearly insignificant (even for 30 demands, it just takes 20ms more comparing with the measures obtained in the previous version of this primitive). When the *write* is propagated using *suggest*, the time needed is also similar to the propagation time.

According to those measures, we can assume that the new implementation of the *write* is fast enough to be run instead of the former implementation, with "always local writing" behavior.

5.2 SunSPOT Gateway

As can be seen in Table 3, the number of connected SunSPOTs does not affect *write* and *read(URL)* primitives performance, because the gateway accesses directly to the request URL provided by them. With the *read(template)*, the *take* and the *query*, however, the delay increases whenever more SunSPOTs are available since the gateway must collect the graphs from all the resources of each SunSPOT to filter the results using the given template.

5.3 Scenario

In order to deploy the scenario in the two ways described in the previous section, all the nodes in this test have been configured to wait for up to one second for responses (due to its asynchronous nature, we cannot predict when all the responses will arrive). In any case, the performance of this scenario could be improved by decreasing the waiting time.

> “There are many home automation or urban instrumentation scenarios where the proposed middleware could be used, e.g. the control of the room temperature”

Action	Demand based implementation	Service based implementation
Discover new locations in the space	5.38	
Update weather measures for an unknown location	1.91	
Update weather measures for a known location	1.18	
Check changes on indoor and outdoor temperature	10.5	
Air conditioning activation since temperature manager invokes its service	0.90	1.45

Table 4: Time Measures for proposed Scenario (in seconds).

Table 4 shows the time required for activating the air conditioning in each implementation. With demand based implementation, the invocation is done faster. Moreover, this difference will become more noticeable if more air conditioning machines are activated. While the service-based invocation will take as much time as needed to perform each *invoke* primitive, *write* will still need a single call taking the same time.

Finally, from a qualitative point of, it has been experienced that the demand-based approach is a much easier way to develop the scenario than the service-based approach.

6 Conclusions and Future Work

This paper explores the possibility of bringing semantic tuplespace-based distributed computing to ubiquitous systems, where many heterogeneous devices share knowledge asynchronously and in a resource-oriented manner, which fits perfectly with the idea of the Internet of Things.

The results obtained in our stereotypical scenario have proved that the middleware has an acceptable performance. However a more exhaustive evaluation of the middleware should be done to check scalability issues and to explore how the intensive use of it would affect mobile and embedded nodes (battery consumption, usefulness of the middleware...).

In addition, some implementation problems found on mobile and embedded nodes can be corrected by applying effort in improving the P2P framework (using multicast) and creating a "reasoning" feature for mobile and embedded devices. Otherwise, the usefulness of the proposed middleware and resulting applications will be limited.

For our future work, we are planning to create new gateways for other embedded devices using the same approach as for SunSPOTs, developing a mobile "reasoning" feature, considering new alternatives for the network layer and exploring security issues in the middleware. Finally, a performance analysis both in a simulator and in a heavily instrumented deployment scenario should be taken into account.

Acknowledgments

This project has been financed under grant PC2008-28 A by

the Department of Education, Universities and Research of the Basque Government, Spain, for the period 2008-10.

References

[1] T. Strang, C. Linnhoff-Popien. A context modeling survey. Workshop on Advanced Context Modelling, Reasoning and Management as part of UbiComp, 2004.

[2] T. Winograd. Architectures for context. Human-Computer Interaction, 16(2):401-419, 2001.

[3] D. Gelernter. Generative communication in linda. ACM Transactions on Programming Languages and Systems (TOPLAS), 7(1):80-112, 1985.

[4] L. J.B. Nixon, E. Simperl, R. Krummenacher, 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.

[5] R. Krummenacher, D. Blunder, E. Simperl, M. Fried. An open distributed middleware for the semantic web. International Conference on Semantic Systems (I-SEMANTICS), 2009.

[6] D. Fensel. Triple-space computing: Semantic web services based on persistent publication of information. IFIP Int'l Conf. on Intelligence in Communication Systems, page 43-53. Springer-Verlag, 2004.

[7] J. Broekstra, A. Kampman, F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. The Semantic Web-ISWC 2002, pp. 54-68, 2002.

[8] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - a pragmatic semantic repository for OWL. Web Information Systems Engineering -WISE 2005 Workshops, pp. 182-192, 2005.

[9] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. C Hugly, E. Pouyoul. Project JXTA-C: enabling a web of things. Proceedings of the 36th Annual Hawaii International Conference on System Sciences, page 9, 2003.

[10] Dominique Guinard, Vlad Trifa, Erik Wilde. A resource oriented architecture for the web of things. Proceedings of Internet of Things 2010 International Conference (IoT 2010), Tokyo, Japan, November 2010.

[11] K. A Hua, R. Peng, G. L Hamza-Lup. WISE: a web-based intelligent sensor explorer framework for publishing, browsing, and analyzing sensor data over the internet. Web Engineering, page 762, 2004.

[12] Aitor Gómez-Goiri, Diego López-de-Ipiña. A triple Space-Based semantic distributed middleware for internet of things. International Workshop on Web-enabled Objects (TouchTheWeb'10), 2010.

[13] Fulvio Crivellaro, Gabriele Genovese. µJena : Gestione di ontologie sui dispositivi mobili, 2007.

[14] G. Kokkinidis, V. Christophides. Semantic query routing and processing in P2P database systems: The ICS-FORTH SQPeer middleware. In Current Trends in Database Technology-EDBT 2004 Workshops, pp. 433-436, 2005.

[15] L.N.P Obermeier, L. Nixon. A cost model for querying distributed rdf-repositories with sparql. Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense Tenerife, Spain, June 2, 2008.

[16] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine, 2000.

[17] J. Riemer, F. Martin-Recuerda, Y. Ding, M. Murth, B. Sapkota, R. Krummenacher, O. Shafiq, D. Fensel, E. Kühn. Triple space computing: Adding semantics to space-based computing. The Semantic Web-ASWC 2006, pp. 300-306, 2006.

