A Triple Space-Based Semantic Distributed Middleware for Internet of Things

Aitor Gómez-Goiri[1] and Diego López-de-Ipiña[1]

Deusto Institute of Technology - DeustoTech
University of Deusto
Avda. Universidades 24, 48007 Bilbao, Spain
{aitor.gomez,dipina}@deusto.es
http://www.morelab.deusto.es

Abstract. In the Internet of Things several objects with network capabilities are connected over a self-configured local network with other objects to interact and share knowledge. In this context, the Triple Space approach, where different processes share common semantic knowledge, seems to fit perfectly. In this paper we present our progress towards a semantic middleware which allows the communication between a wide range of embedded devices in a distributed, decoupled and very expressive manner. This solution has been tested in a stereotypical deployment scenario showing the promising potential of this approach for local environments.

Keywords: triple space, ubiquitous, mobile, embedded, semantic

1 Introduction

Triple Space computing is a coordination paradigm based on tuplespace-based computing. In tuplespace computing the communication between processes is performed by reading and writing data structures in a shared space, instead of exchanging messages [4]. The Semantic Web vision aims to offer machine-understandable persistent data forming a network for machines instead of the current World Wide Web which is more human-centered (web services offer remote functionality to machines, but they are not really Web-based since they are message exchange-driven). 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.

A Triple Space offers different type of autonomy which is not reachable with message exchange-driven communication, such as reference autonomy (the processes can communicate without knowing anything about each other), time autonomy (because of the asynchronous communication) and space autonomy (the processes can be executed in very different computational environments).

In context aware environments, a lot of devices communicate with each other and share their state in order to perform actions over the environment. Different approaches to define and store context data have been presented in several

works [12], coming to the conclusion that ontology based models are the most expressive models and fulfil most of the requirements of these environments. On the other hand, there are different context management models such as widgets, networked services or blackboard model [14]. The blackboard model post messages into a share media, which usually is stored in a centralized server. Triple Space not only expresses knowledge using the semantic model, but it also uses a similar mechanism to blackboard model, but in a decentralized fashion. The work presented in this paper describes an decentralized implementation over devices with limited computational resources.

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

2 Related work

A number of approaches in the field of semantic tuplespace exist [11]. Conceptual Spaces, or cSpaces, were born to study the applicability of semantic tuplespaces to different scenarios including ubiquitous computing [10]. In cSpace the tuples have 7 fields, and in one of them the data can be expressed using first-order logic (ideally), description logics or RDF triples. Even when cSpace can be applied to ubiquitous systems, it is mainly based on client-server architecture which does not implement tuplespace paradigm itself in mobile peers and which restricts the reasoning process to few powerful devices.

Semantic Web Spaces [13] propose some new primitives to extend Linda coordination model. Its tuples are formed by RDF triples and a URI that identifies the tuple, it divides the tuplespace in nested and hierarchical spaces which may have access constraints and it allows RDFS reasoning. In Semantic Web Spaces two views on the coordination are defined: data view and information view. The first one contains syntactically valid RDF data and implements Linda primitives and the second one consistent and satisfiable data and offers some new primitives. The prototype was not designed to be run in small devices.

sTuples [5] was conceived by Nokia Research Center as a pervasive computing work. It provides description logics reasoning and a semantic extension of JavaSpace tuplespace middleware. In sTuples there are managers, which mediates between clients, available services and agents (p.e. recommenders). The prototype was deployed over a centralized server.

As it has been previously explained, in Triple Space Computing the tuples are expressed in form of triples. Currently, two main pure Triple Space Computing middleware implementation exist: tsc++ and TripCom.

TripCom is an European research project which was finished in 2009. It has different kernels hosted in servers which can distribute the semantic data through themselves. TripCom has a query processor to optimize queries, a security manager, a transaction manager and a web service API, 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 (different modules of the same kernel can run in different machines).

The first Triple Space project was called TSC [3]. 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++ [7] is a new version of the former TSC project which basically offers the same API (without transactions) in a distributed way. To do that, tsc++ uses Jxta Peer To Peer [1] framework to perform the coordination and Sesame [1] and Owlim [6] 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 it is made up of (this strategy is known as negative broadcast). This seems to adapt perfectly to ubiquitous system, where different devices share heterogeneous data entering and leaving the system, compromising data consistency and availability. In this aspect with tsc++ a sensor can provide information, but when it leaves the space, its information is also removed from there.

Nevertheless, tsc++ lacks some of the advantages of other alternatives: it does not make inference, it does not allow expressive querying (although it has been solved at the same time in the current version locally) and last but not least, it has not been designed for devices with reduced computing capabilities, because tsc++ middleware focused on architecture and implementation in large scale and we focused in short scale (local area networks).

As it has been seen, even if some works have analysed the convenience of the Triple Space approach in Ubiquitous computing [9, 8] and others have offered solutions to this problem with tuplespaces [5], 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 improving already existing Triple Space middeware to make it more suitable for IoT, enabling devices talking through different communication links and of heterogeneous nature (mobiles, embedded devices, PDAs, Tablets, even PCs) to talk to each other using standard communication protocols and also still be connected to Internet. These heterogeneous 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 Triple Space middleware, effort was put into developing a Java embedded adaptation of tsc++ which was still compatible with standard tsc++ namely

[1] https://jxta.dev.java.net/

tscME. It has also been developed another version for embedded devices such as Sun SPOTs or ZigBee spots which do not support Java ME specification completely. Finally, the current tsc++ implementation was also improved offering a more sophisticated API to perform more expressive queries which are spread through all devices of the space and a service API.

Thus, the API of the proposed middleware is divided in two parts. In the most basic one, primitives to manage spaces, write and query triples (in a destructive or non-destructive way), and for a subscription mechanism are provided. In the complex one, the possibility of using more expressive queries and a Triple Space based web resource oriented approach has been included. In the next sections each of these elements will be discussed.

3.1 API

The tsc++ project and TripCom project, which have their bases on Linda language, have inspired the proposed API. Our API has been structured in two levels called basic API and extended API. In the basic API the tsc++ project API has been maintained for compatibility reasons and it can be implemented both by normal devices and by mobiles. In the extended API some advanced primitives have been provided to allow service management and more expressive queries by using basic primitives. Unfortunately, not all the devices will have the capacity to implement those primitives, and consequently, they are optional.

In the basic API the following primitives can be found:

- `URI` write(URI space, Set<ITriple> triples)
  When the write primitive is called, all the triples passed as parameter are stored together in the same graph associated to the specified space, which is identified by the returning URI (which would always be the same for the same set of triples). These triples are stored locally because *negative broadcast* is used.
- `Set`<ITriple> read(URI space, URI graph)
  `Set`<ITriple> read(URI space, ITemplate template)
  The read primitive returns a complete graph for a given graph URI or for a template which matches at least one of the triples within this graph. Reader must notice that only one graph is returned with read primitive, even if more than one graph in the space matches the graph. The remote semantic repositories will be checked before querying the local one.
- `Set`<ITriple> take(URI space, URI graph)
  Set<ITriple> take(URI space, ITemplate template)
  The take primitive is very similar to read. The main difference is that take reads a graph in a destructive way (removing it from the space).
- `Set`<ITriple> query(URI space, ITemplate template)
  The query primitive returns all the triples which match the given template, no matter what graph they belong to.
- `URI` advertise(URI spaceURI, ITemplate template) void unadvertise(URI spaceURI, URI advertisement)
  A template is advertised to the peers subscribed to a certain template.

– URI subscribe(URI spaceURI, ITemplate template,
INoticationListener listener)
void unsubscribe(URI spaceURI, URI subscription)
Subscribe primitive expresses the interest in events related with the given template. When a peer advertises this template or another template with matches with it, the listener is called to notify the event.

The previously mentioned ITemplate class expresses a sequence of adjacent triple patterns which specify WHERE-clauses of SPARQL queries. In a recent tsc++ version these templates have been improved allowing SPARQL queries, but the version currently supported only has triple patterns to ensure that all peers use the same API.

Our extended API defines the following primitives (more details about services in section 3.5):

– Set<ITriple> queryMultiple(URI spaceURI, ComplexTemplate template)
Given a SPARQL query, QueryMultiple splits it into different ITemplates and sends it to other peers as one primitive, receiving multiple results for each template (partial results from the point of view of the initial query). Once it has all these results, it merges them and it makes the query again over them. Doing this in contrast with tsc++ current solution, data stored in different peers needed by a SPARQL query can be retrieved potentially obtaining new results.
– void register(URI spaceURI, IService service)
void unRegister(URI spaceURI, IService service)
It registers/unregisters a service in the space.
– void invoke(URI spaceURI, IServiceInvocation invocation, IInvocationObserver observer)
It performs an invocation of a service over an space.

3.2 Mobiles

In tscME it has been attempted to provide as much capabilities as possible keeping a small footprint and memory consumption. Unfortunately, the Jxme library (Java ME version of Jxta) is not as mature as Jxse (Jxta for Java SE) and it does not have the same communication mechanisms implemented.

Firstly, a Jxme peer is not able to talk to other peers using multicast, so it relies on a Jxta special peer called Rendezvous which forwards its messages to other fully capable Jxta peers.

Secondly, a Jxme peer is not completely able to exchange all kind of advertisements, the base communication unit in tsc++, with other peers, so we had to use pipe based communication. In pipe based communication virtual channels between peers are established and therefore they are not as flexible as advertisement based communication since it can not be configured how they work. Because of that, tsc++ had to be altered internally so that it communicate seamlessly with both the tsc++ peers and the tscME peers.

Finally, since there are not Java ME semantic repositories right now, Micro-Jena [2] is used to express semantic triples and the RecordStore API to store them into disk whenever it is necessary (a memory *store* has been also performed because RecordStore caused a huge latency). Semantic reasoning has not been implemented yet in mobile peers since, to the best of our knowledge, it does not exist a low resource consuming semantic mobile reasoner publicly available.

3.3 Spots

As has been suggested, the initial goal was not making a proxy to allow the communication between really simple devices and more sophisticated ones, but both with Sun SPOTs and with XBee Gateway difficulties were found to implement a Jxta peer. To overcome this limitation, we focused on making spots communicate through some Proxies which were also Jxta peers.

The Sun SPOT (Sun Small Programmable Object Technology) [2] is a wireless sensor network mote, which can be developed in a limited version of Java ME using the Squawk Virtual Machine. Sun SPOTs do not support the IP protocol stack yet and therefore a tsc++ *gateway* to which the Sun SPOT base station, a special mote, is connected to has been used. In this gateway a really simple standard REST service server, which is also a normal peer of tsc++ network, has been developed using Jetty server and Jersey framework.

XBee Gateway [3] is a special device which can communicate with XBee motes [4] and can be developed with Python programming language. Given that there is not a Jxta protocol library for Python, this gateway was made to communicate via sockets with a server which is a normal tsc++ peer.

3.4 Normal nodes

To improve the tsc++ API, three issues have been taken into account: allowing semantic inference in each node, providing a primitive to make complex SPARQL queries over the space and offering a service API. The service API has been already outlined in section 3.1 and will be discussed further on the next section. To allow local inference in peers, the inference mechanism provided by Sesame (RDF and RDFS inference) and Owlim (OWL Horst inference) have been used.

To implement query decomposition, needed by *queryMultiple*, the SPARQL processor Jena ARQ has been used (Sesame's SPARQLParser would have been an option too) in order to split up each SPARQL query into subject-predicate-object templates that every peer in the space will answer.

Input query

```
CONSTRUCT {
  ?measure ismed:hasValue ?value .
```

[2] http://www.sunspotworld.com/
[3] http://www.digi.com/products/serialservers/connectport-ts-w.jsp
[4] http://www.digi.com/products/wirelessdropinnetworking/sensors/xbee-sensors.jsp

```
}
WHERE {
  ?measure rdf:type ismed:LightMeasure .
  ?measure ismed:hasValue ?value .
  ?measure ismed:hasDateTime ?datetime .
  OPTIONAL {
    ?measure2 rdf:type ismed:LightMeasure .
    ?measure2 ismed:hasDateTime ?datetime2 .
    FILTER(?datetime2 > ?datetime) .
  }
  FILTER( !bound(?datetime2) )
}
```

Result Templates after processing the query

```
?s rdf:type ismed:LightMeasure .
?s ismed:hasValue ?o .
?s ismed:hasDateTime ?o .
```

3.5 Services

Although tsc++ has not got any service invocation or registering method, 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.

More specifically 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 solution aims to provide this control to the device which has the actuators being respectful with the asynchronous nature of TS. For that purpose, a really simple 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 another standard languages can be used) has been designed. First, the service provider should register its service in the space (see figure 1a). 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 1b). An invocation is basically composed by an URI identifier and the input data the service may need.

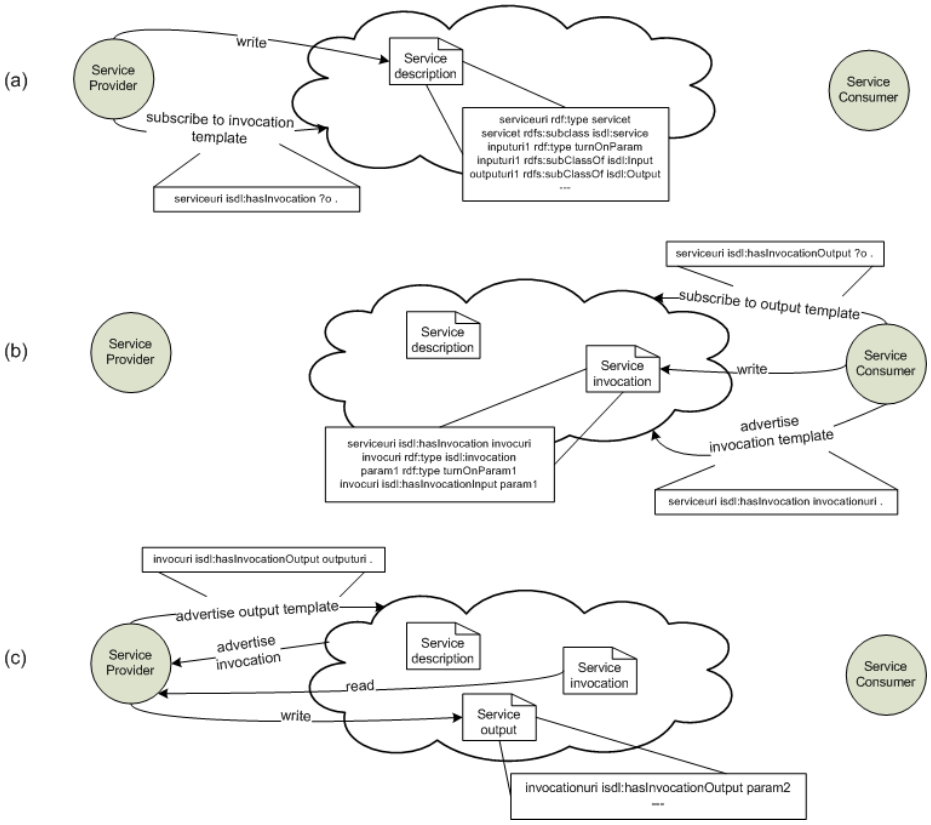

Fig. 1. Services over tsc++: a) registration, b) invocation from the consumer point of view and c) invocation from the service provider point of view.

The service provider, which is subscribed to its services invocation templates, will notice the event (see figure 1c) 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.

4 Experimental Environment

There are many home automation or urban instrumentation scenarios where the proposed middleware could be used. One of this stereotypical cases could be the

control of the room temperature. In this scenario, which uses all the primitives described previously, there are at least 5 peers, which are shown in the figure 2.

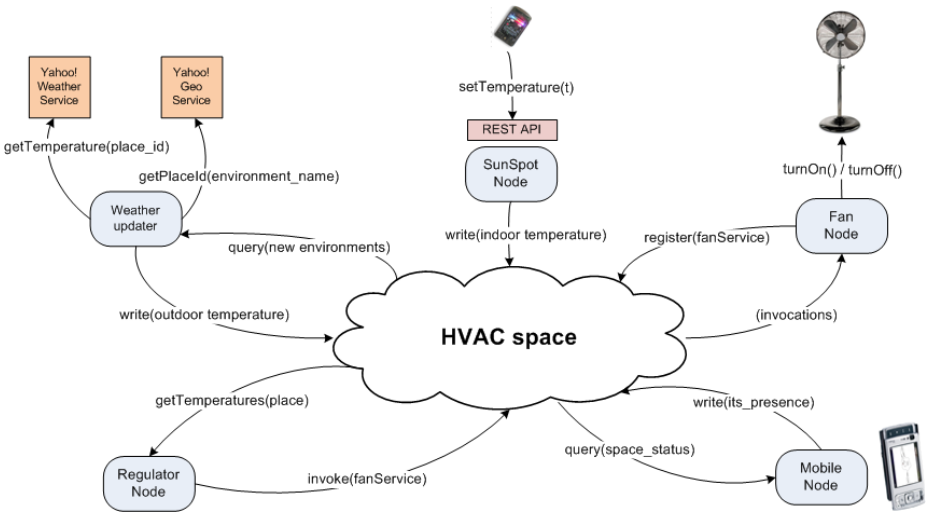

Fig. 2. Schematic view of implemented scenario.

There are two different context providers: Sun SPOTs and weather provider. The first ones are physical devices which set their sensed data via the REST API. The tsc++ peer writes then this information in semantic way into the space. 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: a fan which ideally will decrease the temperature. It can be turned on or off to try to regulate indoor temperature. It can be also monitored what is happening in the space with the mobile phone.

The regulator node uses all this information and whenever someone is in a place (inferred when there is a device who belongs to somebody in the space), takes into account that the Spanish Government regulates the temperature inside public buildings to be below 26 °C whenever the air conditioning is turned on and another law regulates that in office the temperature must be between 17 °C and 27 °C Our regulator controls the temperature in a place taking into account not only the indoor but also the outdoor temperature. If the outdoor temperature is lower than the indoor temperature, it does not make sense to turn on the fan when windows can be opened. Otherwise, the fan is turned on until the temperature reaches a lawful temperature.

So to regulate the temperature, this peer checks whether there is an actuator in the location it wants to regulate which provides a service related with the temperature. If so, a service invocation is carried out in case of needing to decrease the indoor temperature.

Basic indoor temperature control algorithm

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

This scenario is only a proof of concept. Obviously the fan is not the kind of device which can cool a room efficiently. Also, an easy improvement can be done introducing heating actuators and services in the space. 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

Since there are no equivalent implementations apart from tsc++ itself, we tried to enable the comparison between our middleware and the tsc++ assessment which can be found in [7] to evaluate tscME.

On one hand, to assess the performance of tscME, we launched 5, 10 and 20 mobile peers in several emulators running on different machines of the same local network, joined to 1, 5 and 10 spaces. Each emulator, which had 64MB heap space, kept 50 graphs with 5 triples in each graph distributed homogeneously over all its spaces. Measuring the time needed to query with different primitives (see table 1) it was appreciated that most of the time was spent parsing triples instead of waiting for answers. When about 10 responses were processed the time measure was quite poor, moreover, when the test was performed on a real phone the time needed increased about 2-3 seconds.

Kernels	1			10			20		
Spaces	1	5	10	1	5	10	1	5	10
read	0.23	0.22	0.26	3.48	2.99	3.04	10.01	9.97	9.8
take	0.2	0.21	0.28	3.40	2.89	2.61	10.28	9.93	11.07
query	0.4	0.27	0.24	7.05	3.66	3.34	24.83	11.9	10.56

Table 1. TscME networking evaluation results (in seconds)

On the other hand, it seemed interesting to check how much of this time was spent in querying the knowledge base itself. As mentioned before, two different implementations have been developed: a persistent one and a non persistent one. Each operation was measured over both implementations with 10, 50 and 100 graphs in each space and 10 triples in each graph (see the table 2).

Number of graphs		10	50	100
RecordStore	write	0.006	0.006	0.006
	read	0.127	0.486	1.381
	take	0.125	0.473	1.401
	query	0.226	0.934	3.001
Memory	write	0.004	0.004	0.004
	read	0.003	0.004	0.008
	take	0.007	0.010	0.017
	query	0.002	0.003	0.005

Table 2. TscME data access evaluation results (in seconds).

A first observation demonstrates that the performance obtained with few graphs in each space is good enough when triples are kept in the RecordStore. The memory implementation shows a good performance in all the cases, specially with the query which is very fast even for 100 or more triples because of all the triples are stored not only in the graphs they belong to, but also in a common graph which is used as an optimization only for the query primitive.

Finally, we have tested the time needed for the presented scenario by using four devices: two computers (one of them containing three peers which, basically, introduces data in the space, and another one with the regulator peer), a mobile phone (Nokia N95) and a Sun SPOT. The time needed for each action in the scenario can be seen in the table 3. All the peers in this test have been configured to wait up to second for responses, bringing us to the conclusion that the scenario is performed quick enough to be applied in this case.

Action	Time needed
Publish Sun SPOT indoor temperature in the space through REST API	4.69
Discover new locations in the space	5.74
Update weather measures for an unknown location	3.67
Update weather measures for a known location	2.03
Check changes on indoor and outdoor temperature	10.5
Fan activation since temperature manager invokes the its service	1.45

Table 3. Time measures for proposed scenario (in seconds)

6 Conclusions and Future Work

This paper explores the possibility of bringing tuplespace based distributed computing to ubiquitous systems, where a lot of heterogeneous devices would share information in semantic notation asynchronously. This fits perfectly with the idea of the Internet of Things.

The results obtained in our stereotypical scenario have proven that the middleware has a reasonable performance. However in a more complex one (e.g. with more mobile peers) some scalability issues might appear depending on the capabilities of each mobile and the implementation of the scenario itself.

In addition to implementation problems, some of the used devices had not enough capacity to be part of the P2P semantic network that we have used, and they are not capable of reasoning, limiting the usefulness of the proposed middleware and resulting applications.

For our future work, we are planning to increase the expressiveness of query templates to provide comparison between strings and numbers and to implement advertisement based communication on Jxme. Finally, an overall code optimization, the attachment of a semantic reasoner and a performance analysis in a heavily instrumented deployment scenario should be taken into account.

7 Acknowledgments

This project has been financed under grant PC2008-28 A by the Department of Education, Universities and Research of the Basque Government for the period 2008-10.

References

1. Broekstra, J., Kampman, A., Harmelen, F.V.: Sesame: A generic architecture for storing and querying rdf and rdf schema. The Semantic WebISWC pp. 54–68 (2002)
2. Crivellaro, F., Genovese, G.: Jena: Gestione di ontologie sui dispositivi mobili (2007)
3. Fensel, D.: Triple-space computing: Semantic web services based on persistent publication of information. In: Intelligence in Communication Systems. pp. 43–53. Springer-Verlag (2004)
4. Gelernter, D.: Generative communication in linda. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(1), 80–112 (1985)
5. Khushraj, D., Lassila, O., Finin, T.: sTuples: semantic tuple spaces (2004)
6. Kiryakov, A., Ognyanov, D., Manov, D.: OWLIMa pragmatic semantic repository for OWL. Web Information Systems Engineering Workshops pp. 182–192 (2005)
7. Krummenacher, R., Blunder, D., Simperl, E., Fried, M.: An open distributed middleware for the semantic web. International Conference on Semantic Systems (I-SEMANTICS) (2009)
8. Krummenacher, R., Kopeck, J., Strang, T.: Sharing context information in semantic spaces. On the Move to Meaningful Internet Systems 2005: OTM Workshops pp. 229–232 (2005)
9. Krummenacher, R., Strang, T.: Ubiquitous semantic spaces. In: Conference Supplement to the 7th Internationall Conference on Ubiquitous Computing) (2005)
10. Martin-Recuerda, F.: Towards CSpaces: a new perspective for the semantic web. Industrial Applications of Semantic Web pp. 113–139 (2005)
11. Nixon, L.J., Simperl, E., Krummenacher, R., Martin-Recuerda, F.: Tuplespace-based computing for the semantic web: a survey of the state-of-the-art. The Knowledge Engineering Review 23(02), 181–212 (2008)

12. Strang, T., Linnhoff-Popien, C.: A context modeling survey. In: Workshop on Advanced Context Modelling, Reasoning and Management as part of UbiComp (2004)
13. Tolksdorf, R., Bontas, E.P., Nixon, L.J.: A coordination model for the semantic web. In: Proceedings of the 2006 ACM symposium on Applied computing. p. 423 (2006)
14. Winograd, T.: Architectures for context. Human-Computer Interaction 16(2), 401–419 (2001)