Otsopack: Lightweight Semantic Framework for Interoperable Ambient Intelligence Applications

Aitor Gómez-Goiri[a], Pablo Orduña[a], Javier Diego[b], Diego López-de-Ipiña[a]

[a] *Deusto Institute of Technology - DeustoTech*
University of Deusto
Avda. Universidades 24, 48007 Bilbao, Spain
{aitor.gomez,pablo.orduna,dipina}@deusto.es
http://www.morelab.deusto.es
[b] *Logica*
Avda. Manoteras 32, 28050 Madrid, Spain
javier.diego@logica.com
http://www.logica.com

Abstract

In Ambient Intelligence environments machines proactively and transparently work on behalf of humans. The nature of these machines and the communication protocols they use is multifarious. Therefore, the applications running on top of them remarkably demand interoperability. The Triple Space Computing (TSC) paradigm addresses that problem by sharing information represented in a semantic format through a common virtual space. As long as application developers use standard ontologies, different applications using the same spaces will interact automatically. The focus of this paper is to present Otsopack, a fully distributed TSC middleware designed to meet the needs of mobile and resource constrained devices. Otsopack defines a simple HTTP interface for the TSC operations. This interface focuses on simplicity and modularity, so that two implementations that support different modules can still interact. To assess the middleware we provide time and load measurements, and we analyze two independent implementations.

Keywords: Ambient Intelligence; Semantic Web; Web of Things; Tuple Space; Internet of Things; Space-based computing; mobile computing; embedded devices; Triple Space Computing

1. Introduction

Ambient Intelligence (AmI) defends that devices should imperceptibly work on behalf of the humans. Thus, it pursues a natural and transparent interaction which minimizes the psychological impact of machine use. Furthermore, AmI can avoid people from doing low-level but yet time-consuming tasks. These people can now focus on tasks with a high aggregate-value where the importance of the human capital is vital (Pirkkalainen and Pawlowski, 2012).

To pursue these goals, AmI applications need to integrate and coordinate heterogeneous data sources or service providers. Current trends, such as the Web of Things (WoT) initiative (Guinard, 2011), propose a straightforward integration of devices with the web using RESTful services. The problem with this model is that it couples the communication between nodes. This coupling can be avoided by using indirect communication styles (Coulouris et al., 2012). Indirect communication can be space and/or time uncoupling. *Space uncoupling* is achieved when the sender does not need to know the receiver or receivers and vice versa. *Time uncoupling* happens when senders and receivers do not need to exists in the same time.

Independently of the model used, the data applications usually exchange is diverse and application domain dependent. This implies that data will not be meaningful in other domains unless a specialized system converts and reinterprets them. A way to solve this problem is annotating the data semantically as proposed by the WWW (Berners-Lee et al., 2001).

TSC is a coordination paradigm which promotes the indirect communication style and uses semantic data. The way it works is simple: each application writes semantically annotated information in a shared space, and other applications or nodes can query for it and even take it.

For instance, consider two mobile applications. The first one consumes information from professional social networks such as Academia.edu or LinkedIn (Benson et al., 2012; Varela-Candamio and García-Álvarez, 2012). Then, it represents this information using standard ontologies such as Friend of a Friend (FOAF) to link people or even authors of a paper written jointly by multiple authors. After that, the application stores this knowledge in the user's mobile phone through Triple Spaces. Since the application uses Triple Spaces, this information is available for other nodes in the shared space.

A second independent application may notify users when there is a friend in the same party or in the same building. This application may populate

the shared space dumping information retrieved from social networks such as Facebook or Orkut (Varela-Candamio and García-Álvarez, 2012). Prior to the dump, the application *semantizes* the information according to the FOAF ontology. Finally, it periodically looks into the space to check whom is friend of this user.

The interoperability is achieved when a user of the second application, which does not support Academia.edu or LinkedIn, automatically discovers a friend who is using the first application. This is possible because both applications share information in a common space and use the same ontology.

In order to achieve this interoperability through Triple Spaces, we propose a middleware solution called Otsopack. This solution provides two core features: a) it is designed to be simple, modular and extensible, and b) it runs in different computational platforms, including Java SE and Android. The underlying interface is based on HTTP and covers isolated features such as discovery, maintenance or data access. Different implementations can provide only certain features and still interact with each others. This way it is possible to embed it in other devices such as *Digi's ConnectPort X2* IP gateway[1] for *Digi's XBee Sensors*[2]. This gateway's platform only supports Python and requires a partial, ad-hoc implementation.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 describes the conceptual model for an HTTP based TSC solution. Section 4 details the implementation made to adapt it to the necessities of AmI needs. Section 5 explains two study cases where the middleware will be useful and measures the performance of the middleware described. Finally, Section 6 concludes and discusses future work.

2. Related work

In the following subsections we analyze other semantic solutions used in mobile and embedded devices. We study them from the more general ones to the ones which specifically follow the TSC paradigm. Among the analysis we compare our solution with the rest emphasizing their strengths and weaknesses.

[1] `http://tinyurl.com/connectportx2`
[2] `http://tinyurl.com/xbee-sensors`

2.1. Semantic middleware for resource constrained devices

A middleware is a software layer which provides a higher level of abstraction and masks the underlying heterogeneity. Coulouris et al. (2012) define two communication styles on the upper layer of a middleware: the remote invocation and the indirect communication. Using indirect communication one can develop less coupled solutions. However, remote invocation is widely used in applications all over the Internet in SOAP or RESTful (Fielding, 2000) approaches.

Our solution can be defined as an indirect communication middleware which uses a REpresentational State Transfer (REST) interface over HTTP as a baseline. We cover indirect communication solutions in the following subsection. Regarding REST, its use in resource constrained devices is a current trend defended by the WoT initiative (Guinard, 2011).

WoT proposes to embed web servers in everyday things. This objects expose their capabilities following the REST principles. In this way, they fully integrate with the web. This has several benefits:

- Availability of libraries and frameworks in most of the existing computing platforms.
- Reuse of mechanism which have made the web truly scalable. E.g. searching, caching, load-balancing or indexing.
- The users can interact with the objects through a familiar tool: the browser. They can browse or bookmark them, share on social networks, etc.
- Direct integration with other web applications.

The SPITFIRE European project[3] represents the most remarkable effort on gathering the WoT and full semantics. It focuses on fully integrating sensor data with the Linked Open Data (LOD). The LOD are datasets which follow a series of principles on how to open and publish data. The goal of the LOD is to publish linked terms using full semantics.

SPITFIRE shares with our solution the vision of a world populated by devices acting as semantic data providers no matter how small they are

[3] `http://spitfire-project.eu`

(Hasemann et al., 2012). Therefore, many of their efforts are complimentary to this work. However, our work proposes an indirect communication model on top of HTTP interfaces. In this way, our middleware does not only make data easily accessible through HTTP, but also offers higher level of decoupling with TSC.

But that decoupling comes at cost of requiring some extra-tasks to the nodes implementing our middleware. This added complexity is the main disadvantage of our solution comparing it with a plain REST interface. To minimize this problem, the design of the middleware must be simple and modular.

2.2. Semantic space-based computing

Tuple Space (TS), also called space-based computing, is a coordination paradigm based on the shared memory approach (Gelernter, 1985). TS works with semistructured data which is accessed in an associative manner. Several TS solutions have used semantics to enhance the shared data (Nixon et al., 2008).

sTuples was conceived for scenarios like the envisioned in this work (Khushraj et al., 2004). In sTuples, the clients access a centralized space through a communication gateway. The centralization completely simplifies the solution, but makes the whole system dependent on a single machine. Besides, our work avoids the need of gateways by requiring a prominent protocol (i.e. HTTP) for the communication between the nodes.

TripCom[4] distributes the space among different super-peers using distributed hash tables. Specifically, it uses a hash function over the subject, predicate, object and space URL to decide where to store each triple. TripCom draws a clear distinction between the clients which consume data and the devices where the space resides. Our middleware promotes the direct communication between devices. Doing so, they can access to the most updated data and manage their own data.

Finally, Smart-M3 (Honkola et al., 2010) constitutes a remarkable effort to bring the semantic space-based computing to many different devices and protocols. To that end, it distinguishes between two types of nodes: Knowledge Processors (KP) and Semantic Information Brokers (SIB). The SIBs manage the space. The KPs are the nodes which access to the information

[4]TripCom (IST-4-027324-STP, www.tripcom.org)

of the space. For the communication between both types of nodes the Smart Access Protocol (SSAP) is used. The SSAP can be implemented on top of different communication mechanisms. Although theoretically possible, to the best of our knowledge no results have been presented on the federation of two or more SIBs. This makes the solution *de facto* centralized. Apart from the distributed nature of our work, it also avoids the definition of any new communication protocol. Instead it assumes that all the nodes will be able to work at HTTP-level or have a gateway to do so on their behalf. Thanks to that and to the prominence of libraries and tools for this protocol the implementation on new platforms is greatly simplified.

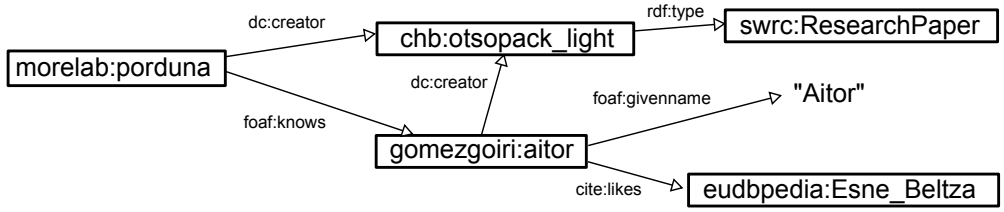
3. Designing Triple Spaces over a canonical HTTP interface

So far, the compatibility of TSC with the REST style has been proved from a formal Hernández and García (2010) point of view. In this section we discuss in detail our model towards the partial achievement of this mapping. To that end, we present a resource oriented HTTP API to expose the knowledge managed by our middleware.

3.1. TSC resources

As was previously stated, our API is based on the TSC paradigm. TSC is a TS variation where the information is stored in RDF. Three key concepts are important at this point: agents share information in a common **space**. A space is identified by an URI. Therefore, all the operations in TSC are performed against a particular space. By default, all applications connect to a common standard space, but they can optionally choose to connect to a particular private space. Within a space, the information is stored in sets of **triples** called **graphs**. Each graph can also be identified by an URI. The RDF **triples** are the underlying concept of all the Semantic Web (SW) languages. Each triple is composed by a subject (which is a URI), a predicate (also a URI) and a value (which can be a URI or a literal), as shown in the Figure 1.

As detailed later, the operations supported by Otsopack attempt to add or remove graphs, as well as to query for graphs or for sets of triples retrieved from different graphs. In order to perform the queries, which enable the selection of a subset of the semantic content hold in a given space, a **template**



subject	**predicate**	**object**
URI	URI	URI | literal
http://gomezgoiri.net/aitor	*cite:likes*	*eudbpedia:Esne_Beltza*
http://gomezgoiri.net/aitor	*foaf:givenname*	*"Aitor"^^xsd:string*
morelab:porduna	*foaf:knows*	*http://gomezgoiri.net/aitor*

Figure 1: Sample triples expressed both graphically and in the RDF triple form. At the bottom a template example is shown. Aliases for the beginning of most URIs, known as prefixes in most Semantic languages, are used to enhance the clarity.

is required. The wildcard templates used by default [5] are special triples with optional wildcard subject, predicate and/or object. For example, the template `?s foaf:knows gomezgoiri:aitor` could be employed to select instances which represent people who know Aitor (see Figure 1).

3.2. Adopted TSC primitives

TSC derives some primitives originally defined in the Linda language (Gelernter, 1985) to access to the semantic information hold in each graph. In this section, these primitives will be explained.

- The **write** primitive allows writing a graph into a given space (identified by its URI). The set of triples received by this primitive will be stored together in the same graph, returning the URI which identifies that graph. The graph URI can be used to access directly to that graph later on, or to create new triples and relate contents.

[5] More sophisticated query languages like SPARQL (`http://www.w3.org/TR/rdf-sparql-query/`) could be used. However, not many embedded platforms have parsers for these languages available. This could be an obstacle for the adoption of the middleware. Therefore, we have opted for using wildcard templates, which are in turn much more simple to process. In any case, the nodes able to parse these query languages can easily decompose a query on wildcard templates.

```
write(space_URI, triples): URI                    [1]
```

- The **read** returns a graph belonging to a given space which contains at least a triple matching the given template or has the given URI as its identifier. If more than one graph fulfill one of these conditions, just one of them is returned (nondeterministically). It should be remarked that it has been designed as a non blocking operation.

- The **take** primitive behaves like a destructive **read**, deleting the graph returned from the space.

```
read(space_URI, graph_URI): triples               [2]
read(space_URI, template): triples                [3]
take(space_URI, graph_URI): triples               [4]
take(space_URI, template): triples                [5]
```

- The **query** primitive aims to see the space as a whole triplestore, returning all the triples matching the given template.

```
query(space_URI,template): triples                [6]
```

- Space management primitives. A node can join or leave a space using `joinSpace(space_URI)` or `leaveSpace(space_URI)`.

3.3. HTTP API for TSC

In the same way TSC has the already explained primitives, HTTP has verbs to get, create, update or remove resources (GET, PUT, POST, DELETE). Consequently, the translation between these two worlds is straightforward.

The contents available on a node where Otsopack is deployed are completely *browseable*. The list of spaces a node is joined to are available under */spaces*. Each space is identified by an URI (e.g. `http://space1`). All the resources of that space, both real (i.e. graphs) or virtual (i.e. query) are listed under */spaces/*{*space_uri*} (summarized by *sp* from now on). Each graph is available on {*sp*}*/graph/*{*graph_uri*}. If we make an HTTP DELETE to that resource, we will be taking that graph from the space from the point of view of TSC. The rest of the mappings are shown in the Table 1 [6].

[6]Note that the write primitive is explicitly excluded from the table. This primitive could

Table 1: HTTP mapping for the primitives detailed in the Section 3.2. *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 an as *). 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	{sp}/graphs/{g}	[2]
GET	{sp}/graphs/wildcards/{s}/{p}/{o-uri}	[3]
	{sp}/graphs/wildcards/{s}/{p}/{o-type}/{o-val}	
DELETE	{sp}/graphs/{g}	[4]
DELETE	{sp}/graphs/wildcards/{s}/{p}/{o-uri}	[5]
	{sp}/graphs/wildcards/{s}/{p}/{o-type}/{o-val}	
GET	{sp}/query/wildcards/{s}/{p}/{o-uri}	[6]
	{sp}/query/wildcards/{s}/{p}/{o-type}/{o-val}	

3.3.1. Distribution

The HTTP mapping is done among a *client* and a *server*. However, as previously detailed, TSC provides reference autonomy, so a consumer will query the space without knowing the particular addresses of the nodes composing that space. This autonomy is managed at other upper and optional layers explained in the following section. However, this HTTP mapping is all a data provider needs to serve information in the space.

3.3.2. Status Codes

Otsopack is compliant with the standardized HTTP status codes. These codes are sent back in the response as part of the header. For instance, the middleware returns the 404 error when no significant result can be found for a primitive. This adoption, apart from enhancing the compatibility with other web applications, enables the modular adoption of our API. For example, if a node does not offer a wildcard based *query*, 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 ease the partial adoption on new platforms.

be directly mapped to a HTTP POST, but we defend that each node should manage its own information (see Section 4.2.1). Therefore, the HTTP API does not allow remote writing.

3.4. Content Negotiation

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 client adds an *Accept* field to the HTTP header with a weighted list of media types it understands. Then, the server will answer with the best possible format it knows about, specifying the *Content-type* in the response.

The benefits of using this mechanism in Otsopack are two-fold. Firstly, it enhances the browsability of the primitives with human understandable HTTP responses. Secondly, it allows different semantic representations (e.g. RDF/XML[7], N-Triples[8] or N3[9]). The latter characteristic becomes crucial since not all the nodes may understand all the formats (e.g. a mobile phone may not have a RDF/XML parser). In these cases, the compatibility of both sides can be ensured through a conversion carried out in the server side. Furthermore, expressing the preference for a semantic format can be useful too in other cases. For example, to obtain the less verbose answer.

4. Adapting the proposed interface to AmI

Once the conceptual model has been presented, its adaptation to the necessities of AmI applications which guided the architecture design will be described in depth. The implementation is characterized by the three big components described in the Figure 2: TSC API, data access and network layers. Besides, we have considered security transversally in all the layers. These components reside together in each node.

The TSC API is composed by the primitives described in Section 3.2. Developers use the middleware through this API. The implementation of the API is done using both the data access and the network layers. The network layer is divided in coordination and communication sublayers. The coordination layer handles aspects related with the node discovery on each space (see Section 4.2.2). The communication layer comprehends the HTTP API described in Section 3.3 and the mechanism to consumer other nodes' ones.

[7] `http://www.w3.org/TR/REC-rdf-syntax/`
[8] `http://www.w3.org/2001/sw/RDFCore/ntriples/`
[9] `http://www.w3.org/TeamSubmission/n3/`

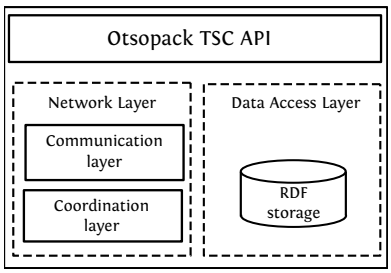


Figure 2: Basic architecture of Otsopack.

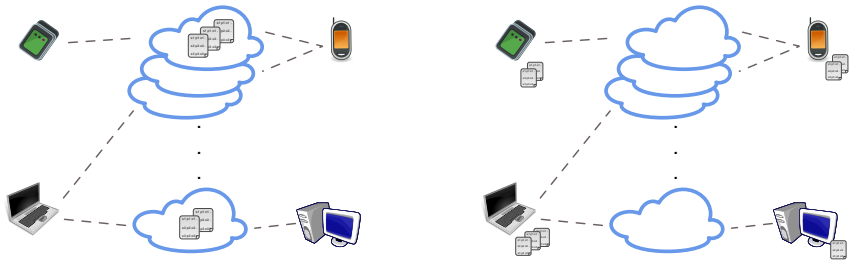
The data access layer is used both by the implementation of the TSC API and the communication layer. In the first case, it is used to provide results to the applications built on top of the middleware. In the second case, it provides these results to other nodes through HTTP responses.

4.1. Data access

The data access layer manages the triples stored in each Otsopack instance. They are stored using a semantic web framework. By default, microjena[10] is used, but Sesame[11] is available in the Java SE version of Otsopack. In any case, other frameworks could be integrated by implementing the access interface. These frameworks enable managing serialization formats and making different types of queries over the stored data.

Furthermore, if the semantic web framework supports reasoning, it can add new triples to the graph inferring them from the stored data. For instance, if it is stored that a thermometer A is on the platform B, given that "`onPlatform`" is the reverse property for "`attachedSystem`", it will infer that platform B has an attached system called thermometer A. Further queries requesting graphs for "`platformB attachedSystem ?`" would return the graph where this was inferred. This feature increases the expressiveness of the operations as well as the amount of possible interactions between different applications, given that more requests can match the same template.

[10] `http://poseidon.ws.dei.polimi.it/ca/?page_id=59`
[11] `http://www.openrdf.org`



(a) Conceptual scheme (b) Real scheme

Figure 3: Different views of the knowledge distribution.

4.2. Network

4.2.1. Knowledge distribution

As previously stated, TSC provides *space decoupling*, so applications do not know where the information is physically located. They just access *the space* requesting, removing and providing information. Therefore, the conceptual scheme the developer should have in mind is the one shown in Figure 3(a). In this scheme, multiple nodes interact through different spaces (represented by clouds in the Figure). Each space contains multiple graphs (represented by papers). These graphs are composed by a set of triples represented by lines within each paper.

However, AmI environments are mainly populated by mobile devices and sensors. These devices frequently join and leave the spaces and the information they hold constantly changes. Thus, AmI environments are highly dynamic. As a consequence, we decided to adopt a distributed strategy which locally stores, or even generates on demand, the information necessary to answer a query. Doing so, we ensure the freshness of the responses regarding the sensed data. The main drawback is that whenever a node is temporarily unavailable its contents become unavailable for the rest of the nodes too. However, this faithfully represents the actual state of the space.

Our TSC design does this by allowing each node, no matter how complex or simple it is, to manage its own information. Besides, it establishes 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 moment). Possible responses are received from them using the same communication channel. This real scheme where each node actually has the sets of graphs

locally is presented by the Figure 3(b). Gómez-Goiri and López-de Ipiña (2012) present further discussion about knowledge distribution strategies.

Implementation. HTTP is a client-server protocol and therefore is conceived to perform unicast communication. However, the semantic information is distributed among the different nodes belonging to a space and as a result more than a node may contain relevant information to answer to the read, take or query primitive. To overcome this discordance, multiple unicast channels should be established each time a node needs to execute one of these primitives.

The distribution module relies on internal parallel queues for performing the parallel requests. If there are 100 nodes in a space, the system will not perform 100 requests at the same time, but it will have several threads performing them and taking the rest from the queue. As soon as the conditions to return the results to the application are met, the rest of the request in the queue are canceled. For instance, in a *read* operation, a single graph is being searched, so as soon as it is found, the rest of the nodes will not be contacted.

Additionally, the primitives are guaranteed to be synchronous at node level, not at space level. As a result, during the processing of an operation where different nodes are contacted, the information might be retrieved in different moments, becoming weakly consistent from the overall view. If applications try to rely on Otsopack to use any coordination pattern, this must be done at graph level.

4.2.2. Discovery

TSC primitives are location agnostic. Therefore, it necessary to know how to address each request to the interested nodes. Since the HTTP protocol does not solve this problem, we design a simple but yet flexible discovery mechanism based on the so called Space Managers (SM). The simplicity is necessary to facilitate its adoption by a wide range of embedded platforms, whereas the flexibility enables many different scenarios.

The SM is a module which maintains a list of alive nodes per space. It can be deployed both with the rest of Otsopack or independently, remotely (using the HTTP API) or locally, in just a node or in every node and manage just one or many, making it suitable for many scenarios:

- scn1: each node has his local SM with a list of the remaining nodes

- scn2: each node is connected to a central SM
- scn3: each node is connected to several SM avoiding bottlenecks

Need of a new discovery mechanism. At first sight, implementing this module could seem a futile work since there are already many discovery mechanisms such as Digcovery (Jara et al., 2013) or Multicast DNS[12] (mDNS). Digcovery is a remarkable effort to enable discovery among heterogeneous systems which to the date has no publicly available implementation. MDNS on the other hand is a production-ready protocol which supports discovery in a multicast environment.

MDNS and other discovery protocols could be used at the *discovery layer* and later rely on HTTP to communicate semantic information with each node. However, they all rely in UDP to communicate with other nodes, which works very efficiently in local networks but it could be hardly used through different networks, specially when mixing different networks. For instance, many telecommunication providers configure network filters in their mobile networks (e.g. 3G). These filters impede the communication between a user's mobile device and a sensor deployed in a local area network using UDP-based protocols. While the Space Manager at this stage does not implement COMET (HTTP Server Push), it has been designed to support it in the near future. Therefore, the implementation of the Space Manager module can be seen as the first step towards supporting a rich variety of potential networks.

Node Discovery. To discover which nodes belong to each space, different strategies can be used alone or in conjunction with others by a SM. In the most basic one, the active nodes can be defined in the SM in memory or in a file. Other supported option is to actively join to a space manager. The system also supports active notifications from the nodes to state they are going to leave the space, or passive notifications, internally processed when the space manager is not pulled often enough by them or they are not reachable for long enough. In order to perform these operations, an optional HTTP interface is provided.

Space Manager Discovery. Otsopack's *discovery layer* manages the discovery of the SMs. Depending on the particular deployment, it is useful to have the URL of the SM hardcoded or in a configuration file, or available in

[12] `http://tools.ietf.org/html/rfc6762`

a QR code in the room where the system is deployed. However, in more dynamic situations, this static configuration may be a problem. Therefore, Otsopack provides a solution based on multicast sockets to discover SMs. As a result, it is possible to configure the application to enter in a network and automatically connect to the Space Managers located in it.

This way, limited sensors will only implement the basic primitives, and a SM will be configured to know that these sensors are in these addresses. The implementation running in the sensor does not need to implement any discovery mechanism to be reached by the rest of the nodes. In order to support limited actuators, there is another component called multicast gateway, which offers the HTTP interface detailed in the previous section, and which acts as a gateway to a particular space. This way, the limited actuator can also avoid implementing the discovery mechanisms letting the interaction with the SMs and the final nodes to the multicast gateway.

4.3. Security

Security is important at different levels in AmI applications. Given the data-centric nature of the framework, there are mainly two core concepts: 1) a data provider may only grant access to certain data to a certain set of users and 2) a data consumer may trust only a set of providers for certain set of acquired data. A derived issue is how to authenticate each other in such dynamic scenarios.

In order to support the first requirement, an OpenID-based[13] solution has been built. An Identity Provider securely identifies data consumers to the data providers. Data providers can establish which graphs can be accessed by which users. Therefore, the provider will return a restricted graph only if the valid user is requesting it. In other words, the same application can get different amounts of information depending on whether it provides credentials or not.

For the second requirement, work has been placed to make restrictions about who can be trusted for certain information. Besides, a lightweight server-to-client authentication (Naranjo et al., 2012) is being adopted in Otsopack.

[13] `http://openid.net`

5. Case study

Otsopack has been used in real scenarios, both in a supermarket[14] and in a hospital[15], within the ACROSS project (Castillejo et al., 2011). However, for the sake of brevity and clarity, two simple and not implemented applications have been designed for this contribution. Their ultimate aim is to show how the middleware solution can be used to achieve interoperability. Finally, in order to prove the feasibility of the implementation in limited devices, we present Otsopack's response time in real sensors.

5.1. Application 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 these measures go beyond a determined threshold, the company needs to take the proper action. To respond 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 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 Semantic Sensor Network Ontology (SSN)[16] or Semantic Web for Earth and Environmental Terminology (SWEET)[17] 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.

[14] `http://www.acrosspse.com/across/servlet/Noticias?id=33`
[15] `http://www.acrosspse.com/across/servlet/Noticias?id=35`
[16] `http://www.w3.org/2005/Incubator/ssn/ssnx/ssn`
[17] `http://sweet.jpl.nasa.gov`

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
...	...	...

5.2. Application 2: Home automation

On the one hand, a room is populated with several kind of sensors: *Digi's XBee Sensors* with a *Digi's IP gateway* and a FoxG20[18] embedded platform connected to sensors and actuators. Besides, an Android application semantically stores 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, Measurement Units Ontology (MUO)[19] or RECommendations Ontology (RECO)[20] 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

[18] http://www.acmesystems.it
[19] http://tinyurl.com/MeasurementUO
[20] http://tinyurl.com/RECommendationsO

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
...	...	...

5.3. Interoperability

Given that both systems use a common ontology called SSN and providing they use a common space in TSC, whenever the Security application asks for triples matching a template "`?s rdf:type sweet:Temperature`", the Home automation application would return "`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.

5.4. Feasibility in embedded devices

Finally, one of the challenges of Otsopack was to support limited devices such as low cost sensors. In order to do so, two devices are used: FoxG20[21] and *XBee sensors* with a *ConnectPort X2* IP gateway. XBee can only be programmed in Python, so a subset of the protocol was implemented in this language, only supporting that other nodes access sensor information in the space.

Therefore, the rest of the nodes located in the shared space would be able to query the space and the sensors would return the information. The

[21] `http://www.acmesystems.it`

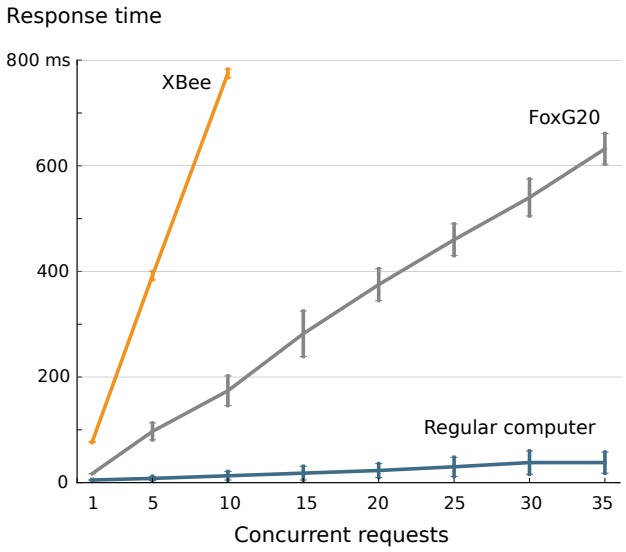


Figure 4: Response times of Otsopack implementations running on different devices.

requests performed by Android phones or PCs with Otsopack do not deal with the sensors in a particular way, neither the Space Managers or other Otsopack components. They only query for certain information to the space, and the sensors return the information if the query matches the information they contain.

To evaluate if this lightweight implementation of Otsopack fits, time measurements have been taken on both sensor platforms under different levels of stress (from 1 concurrent request to 35), and they have been compared with a regular PC running Otsopack (Java version), as shown in Figure 5.4. The results show that these sensors can support a wide number of concurrent requests using the presented middleware, which should be enough for any of the described scenarios.

6. Conclusions and further work

In this paper we have presented our work towards a lightweight flexible Triple Space Computing-based solution called Otsopack. This framework is specifically designed to adapt to AmI environments and aims to distribute the information among different types of nodes in a dynamic way. These design requirements lead to a modular interface, a flexible networking approach and a simple but yet trustful security layer.

The results show that firstly, the effort required to develop particular features of Otsopack on different platforms is low, due to its simplicity and the use of standard HTTP capabilities. Secondly, the networking layer shows its ability to adapt to a wide range of scenarios both in simulated and real environments. Finally, the distributed security schema allows to automatically expose and share the information managed by the applications so multiple applications that did not know each other can still interact.

To detail future directions, we aim to design a cloud computing implementation of Otsopack and compare it with the current solution. Regarding the security of the information, work has been done to ensure security in certain deployments (Naranjo et al., 2012). The next natural step is to integrate that solution in Otsopack.

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 CEN-20101019 (THOFU), funded by the Spanish Centro para el Desarrollo Tecnológico Industrial (CDTI) and supported by the the Spanish Ministry of Science and Innovation.

References

Benson, V., Morgan, S., Tennakoon, H., 2012. A framework for knowledge management in higher education using social networking. International Journal of Knowledge Society Research (IJKSR) 3, 44–54.

Berners-Lee, T., Hendler, J., Lassila, O., 2001. The semantic web. Scientific American 284, 3443.

Castillejo, E., Orduña, P., Laiseca, X., Gómez-Goiri, A., López-de Ipiña, D., Sergio, F., 2011. Distributed semantic middleware for social robotic services, in: Robot 2011, Seville, Spain.

Coulouris, G., Dollimore, J., Kindberg, T., Blair, G., 2012. Distributed Systems: Concepts and Design. Addison Wesley. 5 edition.

Fielding, R.T., 2000. Architectural styles and the design of network-based software architectures. Ph.D. thesis. University of California. Irvine.

Gelernter, D., 1985. Generative communication in linda. ACM Transactions on Programming Languages and Systems (TOPLAS) 7, 80112.

Gómez-Goiri, A., López-de Ipiña, D., 2012. Assessing data dissemination strategies within triple spaces on the web of things, in: 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp. 763 –769.

Guinard, D., 2011. A Web of Things Application Architecture Integrating the Real-World into the Web. Ph.D.. ETH Zurich.

Hasemann, H., Kroller, A., Pagel, M., 2012. RDF provisioning for the internet of things, in: Internet of Things (IOT), 2012 3rd International Conference on the, pp. 143 –150.

Hernández, A.G., García, M.N.M., 2010. A formal definition of RESTful semantic web services, in: Proceedings of the First International Workshop on RESTful Design, ACM, New York, NY, USA. p. 3945.

Honkola, J., Laine, H., Brown, R., Tyrkko, O., 2010. Smart-M3 information sharing platform, in: 2010 IEEE Symposium on Computers and Communications (ISCC), IEEE. pp. 1041–1046.

Jara, A., Lopez, P., Fernandez, D., Castillo, J., Zamora, M., Skarmeta, A., 2013. Mobile digcovery: discovering and interacting with the world through the internet of things. Personal and Ubiquitous Computing , 1–16.

Khushraj, D., Lassila, O., Finin, T., 2004. sTuples: semantic tuple spaces, in: The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004, pp. 268–277.

Naranjo, J.a.M., Orduña, P., Gómez-Goiri, A., López-de Ipiña, D., Casado, L.G., 2012. Lightweight user access control in energy-constrained wireless network services, in: Bravo, J., López-de Ipiña, D., Moya, F. (Eds.), Proceedings of the 6th international conference on Ubiquitous Computing and Ambient Intelligence. Springer Berlin Heidelberg. Lecture Notes in Computer Science, pp. 33–41.

Nixon, L.J., Simperl, E., Krummenacher, R., Martin-Recuerda, F., 2008. Tuplespace-based computing for the semantic web: a survey of the state-of-the-art. The Knowledge Engineering Review 23, 181212.

Pirkkalainen, H., Pawlowski, J.M., 2012. The knowledge intervention integration process: A process-oriented view to enable global social knowledge management. International Journal of Knowledge Society Research (IJKSR) 3, 45–57.

Varela-Candamio, L., García-Álvarez, M.T., 2012. Analysis of information and communication technologies in higher education: A case study of business degree. The International journal of engineering education 28, 1301–1308.