novática
CEPIS UPGRADE Revista de la Asociación de Técnicos de Informática

Novática, revista fundada en 1975 y decana de la prensa informática española, es el órgano oficial de expresión y formación continua de **ATI** (Asociación de Técnicos de Informática), organización que edita también la revista **REICIS** (Revista Española de Innovación, Calidad e Ingeniería del Software). ***Novática*** co-edita asimismo **UP**GRADE, revista digital de **CEPIS** (*Council of European Professional Informatics Societies*), en lengua inglesa, y es miembro fundador de **UP**ENET (**UP**GRAD ***E***uropean ***NET***work).

<http://www.ati.es/novatica/>
<http://www.ati.es/reicis/>
<http://www.upgrade-cepis.org/>

ATI es miembro fundador de **CEPIS** (*Council of European Professional Informatics Societies*) y es representante de España en **IFIP** (*International Federation for Information Processing*); tiene un acuerdo de colaboración con **ACM** (*Association for Computing Machinery*), así como acuerdos de vinculación o colaboración con **AdaSpain**, **AI2**, **ASTIC**, **RITSI** e **Hispalinux**, junto a la que participa en **ProInnova**.

Coordinación Editorial, Redacción Central y Redacción ATI Madrid
Padilla 66, 3º, dcha., 28006 Madrid
Tlfn.914029391; fax.913093685 <novatica@ati.es>
Composición, Edición y Redacción ATI Valencia
Av. del Reino de Valencia 23, 46005 Valencia
Tlfn./fax 963330392 <secreval@ati.es>
Administración y Redacción ATI Cataluña
Via Laietana 46, ppal. 1ª, 08003 Barcelona
Tlfn.934125235; fax 934127713 <secregen@ati.es>
Redacción ATI Aragón
Lagasca 9, 3-B, 50006 Zaragoza.
Tlfn./fax 976235181 <secreara@ati.es>
Redacción ATI Andalucía <secreand@ati.es>
Redacción ATI Galicia <secregal@ati.es>
Suscripción y Ventas <http://www.ati.es/novatica/linteres.html>, ATI Cataluña, ATI Madrid
Publicidad
Padilla 66, 3º, dcha., 28006 Madrid
Tlnf.914029391; fax.913093685 <novatica@ati.es>
Imprenta: Derra S.A., Juan de Austria 66, 08005 Barcelona.
Depósito legal: B 15.154-1975 -- ISSN: 0211-2124; CODEN NOVAEC
Portada: El devorador de fantasías - Concha Arias Pérez / © ATI
Diseño: Fernando Agresta / © ATI 2003

Nº 209, enero-febrero 2011, año XXXVII

sumario

Monografía del próximo número:
"Ingeniería del Software en proyectos de e-Learning"

Middleware semántico orientado a recursos para entornos ubicuos

Aitor Gómez-Goiri, Mikel Emaldi Manrique, Diego López de Ipiña
Deusto Institute of Technology – Tecnológico Deusto, Universidad de Deusto, Bilbao

<{aitor.gomez,m.emaldi,dipina}@deusto.es>

Resumen: *En los entornos ubicuos, dispositivos de distinta naturaleza comparten información a través de redes altamente interconectadas. En este contexto, los modelos semánticos pueden utilizarse para describir el contexto que rodea dichos dispositivos de una manera muy expresiva, normalmente almacenada en bases de conocimiento centralizadas. Las aplicaciones construidas a partir de estas bases de conocimiento generalmente no son sensibles a la dinamicidad de la red. El middleware propuesto en este artículo facilita el intercambio de conocimiento entre diferentes sensores y actuadores de una manera altamente distribuida, con bajo acoplamiento y orientada a recursos, siguiendo el paradigma del Triple Space. Este middleware ha sido probado en un escenario estereotipado, mostrando cómo los diferentes nodos de una red pueden compartir información, conservando su autonomía y con un rendimiento razonable en los dispositivos con capacidad de computación reducida.*

Palabras clave: *distribuido, embebido, móvil, semántica, triple space.*

Autores

Aitor Gómez-Goiri es estudiante de doctorado en la Universidad de Deusto y trabaja en la unidad de INTERNET en el Tecnológico Deusto – Deusto Institute of Technology <http://www.tecnologico.deusto.es/>. Sus investigaciones se centran en el diseño de *middleware* semántico para dispositivos móviles y embebidos.

Mikel Emaldi Manrique es investigador interno en el Tecnológico Deusto – Deusto Institute of Technology <http://www.tecnologico.deusto.es/>. Sus investigaciones se centran en el *middleware* de integración para dispositivos embebidos.

Diego López de Ipiña es doctor en Ingeniería por la Universidad de Cambridge, Reino Unido. Es el investigador principal de la unidad de INTERNET en el Tecnológico Deusto – Deusto Institute of Technology <http://www.tecnologico.deusto.es/>. Sus investigaciones se centran en la aplicación de *middleware* y técnicas web a facilitar los entornos inteligentes.

1. Introducción

En entornos sensibles al contexto, los dispositivos se comunican entre sí, intercambiando modificaciones en sus estados y realizando acciones sobre el entorno. En diversos trabajos se han presentado diferentes aproximaciones para modelar y almacenar la información del contexto [1], llegando a la conclusión de que los modelos basados en ontologías son los más expresivos y los que más respetan los requisitos de los entornos sensibles al contexto. El modelo *blackboard*, uno de los principales modelos de gestión de contexto [2], publica mensajes en un medio compartido que normalmente se encuentra centralizado en un servidor.

La computación *Triple Space* es un paradigma de coordinación basado en la computación *tuplespace*, que a su vez deriva del programa de programación paralela Linda [3]. En *tuplespace*, la comunicación entre procesos se realiza a través de la lectura y escritura de tuplas de datos en un espacio compartido, en lugar del intercambio de mensajes. Al contrario que la World Wide Web, centrada en humanos y que requiere la intervención del usuario (los servicios web ofrecen funcionalidad remota a las máquinas, pero realmente no están basados en la Web ya que basan su funcionamiento en el intercambio de mensajes), la visión de la Web Semántica hace que la información sea comprensible por máquinas, formando una red de conocimiento para ellas. La computación *Triple Space* (TS) implementa la comunicación basada en el *tuplespace* utilizando tripletas RDF como unidad de intercambio básica, en las que las unidades de información están compuestas de tres dimensiones para poder expresar su información semántica: "sujeto predicado objeto". El TS ofrece autonomía de referencia (los procesos pueden comunicarse sin conocerse entre ellos), autonomía temporal (ya que se utiliza comunicación asíncrona) y autonomía de espacio (ya que los procesos pueden ejecutarse en entornos computacionales totalmente diferentes). Estas características no pueden darse en la comunicación basada en el intercambio de mensajes.

En este artículo se presenta un *middleware* para entornos ubicuos que utiliza el modelo *blackboard* a través de una implementación descentralizada de *Triple Space*. El *middleware* está ideado para ser desplegado en dispositivos con recursos computacionales limitados, como móviles o dispositivos embebidos, que comparten su información contextual a través de la red formando la *Internet of Things*.

El artículo está organizado de la siguiente manera. En la **sección 2** se analiza el trabajo relacionado. La **sección 3** presenta el *middleware* propuesto. En la **sección 4** se detalla el entorno experimental. En la **sección 5** se examinan los resultados de la utilización de la solución propuesta en el entorno experimental. Finalmente, en la **sección 6** se presentan las conclusiones y futuras líneas de trabajo.

2. Trabajo relacionado

Como hemos mencionado, los *Triple Spaces* derivan de *tuplespaces* y este paradigma a su vez procede del lenguaje de programación paralela Linda. En el campo de los *tuplespaces* semánticos existen múltiples soluciones [4]. Los *Conceptual Spaces*, o cSpaces, nacieron para estudiar la aplicabilidad de los *tuplespaces* semánticos en diferentes escenarios, incluyendo la computación ubicua. Los *Semantic Web Spaces* proponen nuevas primitivas para extender el modelo de coordinación Linda, definiendo dos vistas de coordinación de información diferentes: vista de datos (con grafos RDF sintácticamente válidos y primitivas propias de Linda) y la vista de información (con información consistente y satisfactible, y nuevas primitivas). sTuples fue concebido por el Nokia Research Center como un trabajo de computación ubicua que provee razonamiento de lógica descriptiva y extensión semántica del *middleware* de *tuplespace* JavaSpace.

Ninguno de los proyectos mencionados puede considerarse como totalmente distribuido. Su despliegue se realiza sobre una arquitectura en la que los móviles son simples clientes que no implementan el paradigma del *tuplespace*. Dicho de otra forma, estas soluciones restringen el proceso de computación a dispositivos con mayor potencia.

En *TS* las tuplas son expresadas en forma de tripletas. Actualmente existen dos implemen-

taciones del *middleware* de Computación *Triple Space*: tsc++ y TripCom.

TripCom[1] dispone de diferentes núcleos alojados en servidores que pueden distribuir la información semántica entre ellos, pero una vez más, está demasiado orientado hacia el servidor. En TripCom los clientes no son parte del espacio, y por la complejidad del software, orientado a ejecutarse en máquinas potentes donde los módulos que componen el núcleo pueden llegar a desplegarse en diferentes máquinas, sería imposible que lo fuesen.

TSC fue el primer proyecto *Triple Space*. En TSC, las tripletas se interrelacionan para formar grafos y permiten realizar consultas sobre ellos a través de plantillas. También ofrece un contexto transaccional y un procedimiento para publicar y suscribirse a diferentes patrones. Tsc++ [5] una nueva versión del proyecto TSC original [6], que básicamente ofrece la misma API de una manera distribuida. Para ello, tsc++ utiliza el *framework* Peer to Peer Jxta[2] para llevar a cabo la coordinación entre nodos y Sesame [7] y Owlim [8] para almacenar las tripletas en cada uno.

Los nodos en tsc++ almacenan su propia información permitiendo la distribución del espacio a través de todos los nodos que lo forman. Esta estrategia, donde la información se escribe en local y las consultas se propagan al resto de nodos de un grupo, es conocida como *negative broadcasting*. *Negative broadcasting* se adapta al funcionamiento de los sistemas ubicuos donde diferentes dispositivos comparten información heterogénea entrando y saliendo del sistema, comprometiendo la consistencia y la disponibilidad de la información.

En este aspecto un sensor puede proveer una determinada información, pero cuando abandona el espacio su información se elimina automáticamente, ya que no está disponible para el resto de nodos. De cualquier manera, como se explica en la **sección 3.4**, esta estrategia también presenta limitaciones en los casos en los que un dispositivo quiere modificar remotamente el estado de un actuador gestionado por otro dispositivo.

A pesar de todo ello, tsc++ carece de algunas de las ventajas que ofrecen otras alternativas: no realiza inferencia, no permite las consultas expresivas y por último, y no por ello menos importante, no ha sido diseñado para dispositivos con capacidad de computación reducida. El motivo principal es que está orientado a desplegarse en redes de área global y nuestra solución se centra en sistemas más reducidos (redes de área local con un entorno inteligente).

Nuestro trabajo tiene como objetivo facilitar la construcción de la *Internet of Things*, donde los objetos cotidianos disponen de conectividad para compartir información entre ellos.

Por esta razón, comparte características de otras soluciones propuestas en este área como [9] o la *Web of Things* [10]. La primera describe como el *framework* Jxta puede facilitar la comunicación entre objetos que disponen de conectividad, y aunque nuestra solución utiliza Jxta como protocolo de comunicación, está más centrada en el modo en el que compartiendo conocimiento se facilita la coordinación de dispositivos que en las capas inferiores que posibilitan este hecho. La segunda solución aboga por la conveniencia de que los objetos formen parte de la web utilizando las técnicas web existentes para crear aplicaciones, por lo que se centra en desplegar servidores web en dichos dispositivos. A pesar de la simplicidad de este último enfoque hay ciertos aspectos que no son tenidos en cuenta, como el método de descubrimiento de nuevos dispositivos (otros trabajos intentan corregir esta limitación del enfoque web, como [11]), la inestabilidad de los nodos que sirven webs en una red o el empleo de semántica.

Como se ha podido observar, a pesar de que algunos trabajos han analizado la conveniencia de la utilización del enfoque del *tuplespace* semántico en la computación ubicua, en la medida de nuestro conocimiento, TS nunca ha sido diseñado e implementado de manera específica para utilizar dispositivos móviles y dispositivos empotrados como un nodo más del espacio, y no como simples clientes. Este enfoque permite que dispositivos heterogéneos se comuniquen entre sí eliminando la necesidad de una infraestructura fija y de configuración previa, habilitando así entornos más dinámicos.

Gestión de espacios	createSpace(space)
	joinSpace(space)
	leaveSpace(space)
Consulta	query(space,template): triples
	read(space,graph): triples
	read(space,template): triples
	take(space,graph): triples
	take(space,template): triples
	queryMultiple(space,sparql): triples
Escritura	write(space,triples): URI
	demand(space,template,timeout)
Subscripciones	subscribe(space,template): URI
	unsubcribe(space,URI)
	advertise(space,template): URI
	unadvertise(space,URI)
Servicios	register(space,service)
	unregister(space,service)
	invoke(space,invocation,listener): URI

Tabla 1. Primitivas del *Triple Space* diseñado agrupadas según su naturaleza.

3. Descripción de la infraestructura

Nuestro principal objetivo ha sido la implementación de un *middleware* TS adecuado para entornos ubicuos. Para conseguirlo, hay que habilitar la comunicación entre dispositivos de naturaleza heterogénea (móviles, dispositivos empotrados, PDAs, Tablets, PCs), utilizando los protocolos estándar, y además lograr que permanezcan conectados a Internet. Estos dispositivos se comunicarán a través de *push-and-pull* logrando un bajo acoplamiento.

Del mismo modo, una de las principales preocupaciones fue habilitar la ejecución del *middleware* TS en dispositivos móviles y empotrados. Este esfuerzo dio lugar a una implementación reducida de Java con TS compatible con tsc++ [5], denominado *tscME*. También se desarrolló un módulo para permitir que dispositivos como las SunSPOTs o sensores XBee, los cuales no soportan el protocolo IP, sean parte del espacio a través de un *gateway*. Por último, la API *Triple Space* adoptada de [5] ha sido mejorada, ofreciendo consultas más expresivas que son propagadas a través de todos los nodos del espacio, primitivas para gestionar servicios y un nuevo enfoque de escritura.

En las siguientes secciones describimos la API y las diferentes etapas de implementación del *middleware*.

3.1. API

La API que permite interactuar con *TS* está inspirada en la de tsc++, pero ha sido adaptada teniendo en cuenta la naturaleza y restricciones de los entornos ubicuos. Está compuesta por las primitivas presentadas en el **tabla 1**. Destaquemos que *space* es la URI que identifica el conocimiento de un grupo de

nodos, *graph* es una URI que identifica una serie de tripletas escritas en el espacio, *triples* es un conjunto de tripletas y *template* expresa una secuencia de patrones adyacentes que especifican cláusulas WHERE de consultas SPARQL. *Service* e *invocation* son interfaces que expresan los datos necesarios para definir un servicio y su invocación.

Las primitivas de gestión de espacios permiten a los nodos compartir información con diferentes grupos de nodos. *Query* recupera todas las tripletas que coinciden con una plantilla determinada en un determinado espacio, mientras *queryMultiple* divide una consulta SPARQL en diferentes plantillas que son enviadas a todos los nodos del espacio, combinando las respuestas recibidas de dichos nodos y realizando la consulta de nuevo sobre estas respuestas, para filtrar los resultados. *Read* recupera un único grafo al completo que contenga al menos una tripleta que coincida con la plantilla especificada. Por su parte, *take* realiza lo mismo que *read* eliminando el grafo recuperado del repositorio semántico del nodo que lo almacena. Tanto *read* como *take* están sobrecargadas para obtener un grafo concreto identificado por una URI.

A pesar de que la primitiva *write* se utiliza para escribir tripletas en el repositorio local, devolviendo la URI que identifica el nuevo grafo en el que se van a almacenar las tripletas, su implementación ha sido modificada para que no siempre utilice la estrategia de *negative broadcasting*. *Demand* permite que el desarrollador anuncie al resto de nodos que es responsable de los grafos que contengan al menos una tripleta que coincida con una plantilla determinada, durante un periodo de tiempo definido por el parámetro *timeout*. Esta solución se explica con mayor detalle en la **sección 3.6**.

La primitiva *subscribe* se usa para que un nodo sea consciente de las notificaciones realizadas acerca de una plantilla determinada, o sobre alguna particularización de ésta. *Advertise* permite a un nodo propagar una notificación a todos los nodos del espacio informando sobre un evento determinado. Estas primitivas pueden ser utilizadas para construir un sistema de notificaciones dentro del espacio.

Finalmente, se ha creado una API de servicios [12] haciendo uso de las primitivas básicas, como primera aproximación para resolver los problemas descritos en la **sección 3.4**. Con *register* y *unregister*, se registra la descripción de un servicio en un espacio y se suscribe a invocaciones del mismo. Con *invocation*, se escriben en el espacio las posibles entradas que se necesitan para invocar el servicio y se anuncian al proveedor de servicios. El proveedor de servicios puede ejecutar dicha invocación y, opcionalmente, recuperar la salida del servicio. En muchas ocasiones, la salida del servicio implica cambios en la base de conocimiento del proveedor, como resultado del cambio en el contexto. Este proceso se explica en detalle en la **sección 3.4**.

3.2. 1ª etapa: tscME

El primer paso para implementar nuestro *middleware* fue crear una librería llamada tscME, para un MIDlet que será desplegado en Java ME CLDC. Esta versión proporciona gestión de espacios, consulta (excepto *queryMultiple*), escritura y suscripción de manera nativa. La comunicación se realizó[3] utilizando Jxme (versión de Jxta para Java ME)[3] y la información semántica se gestiona utilizando Microjena [13].

Debido a que Jxme no utiliza *multicast*, para propagar los mensajes de los móviles, se tiene que usar un elemento de la arquitectura Jxta denominado *Rendezvous*. Este elemento hace que la solución se centralice ligeramente, haciendo que los nodos móviles dependan de él, aunque en el momento en el que Jxme implementase *multicasting*, este problema dejaría de existir automáticamente.

Dado que el método de comunicación empleado por Jxta es ligeramente distinto al utilizado en los nodos tsc++, se ha acoplado una nueva capa de comunicación a estos últimos. Para que las suscripciones y anuncios sean compatibles con los métodos utilizados por tsc++, un nodo tsc++ debe actuar como *gateway* para los nodos tscME, rompiendo el principio de distribución en los nodos móviles. Aún así, no se debe ignorar el hecho de que las primitivas de suscripción no son parte del paradigma TS por sí mismas.

Finalmente, la plantilla básica de consulta ha sido mejorada creando una nueva plantilla de comparación tanto en tscME como en las versiones extendidas de tsc++. Esta nueva plantilla permite comprobar cuando una tripleta coincide con la plantilla, comparando su valor literal.

Ejemplos de plantillas de comparación. El primer ejemplo es equivalente a *?s ?p ?o*. El segundo comprueba que una tripleta tenga el predicado *myont:lenght* y el valor literal diferente a 3. El último comprueba que una tripleta tenga el sujeto *:aitor*, el predicado *:has_age* y el valor literal sea menor o igual a 30.

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

3.3. 2ª etapa: extensiones de tsc++

Como se ha descrito anteriormente, se ha añadido una nueva capa de comunicación para que tsc++ sea compatible con tscME. Además, se ha añadido un API de servicios y la primitiva *queryMultiple* a estos nodos. La API de servicios se explicará en la siguiente sección.

QueryMultiple utiliza una consulta *SPARQL construct* como entrada, dividiéndola en plantillas básicas que son propagadas a través del resto de nodos del espacio. A continuación, filtra localmente las respuestas recibidas a partir de la consulta original. En [14] y [15] se propusieron diferentes estrategias para propagar las consultas, pero debido a la naturaleza del *negative broadcasting*, los nodos no conocen nada sobre los demás, imposibilitando la redirección de las plantillas básicas a un nodo determinado. Ya que la descomposición de consultas SPARQL no es posible en nodos móviles, esta primitiva no es de obligatorio cumplimiento en los mismos.

3.4. 3ª etapa: API de servicios

La necesidad de ofrecer o no una infraestructura de servicios en *Triple Space* puede ser discutible dado que el conocimiento puede ser obtenido o escrito en el espacio directamente, trabajando con los recursos a través del paradigma REST.

A pesar de que en la **sección 3.6** se propone una aproximación más orientada a recursos, en [12] se propusieron ciertas primitivas para utilizar servicios dentro de TS.

En los entornos ubicuos, la información detectada puede ser obtenida consultando el espacio, pero al modificar actuadores se encontraron ciertas limitaciones:

- Propensión a errores. Dado que tsc++ no implementa ningún tipo de lista de control de acceso, cualquiera puede modificar por error más conocimiento del que desea sobreescribiendo un grafo con menos información de la que inicialmente contenía.
- Concurrencia. Si dos nodos diferentes modifican la misma información simultáneamente, ¿Qué información será tomada en cuenta?
- Localización de la información. Dada la naturaleza de tsc++, cuando una información que inicialmente pertenece al *nodo a* es modificada por el *nodo b*, se almacena en el *nodo b* en vez de almacenarse en el *nodo a*. Si el *nodo b* abandona el espacio, desaparecerá mucha información crucial sobre el actuador. Parece razonable que la información sobre un sensor o actuador se almacene en el dispositivo que la gestiona (en el ejemplo, el *nodo a*). Nuestra solución tiene como objetivo proporcionar el control al dispositivo que controla el actuador, respetando la naturaleza asíncrona del TS. Para alcanzar este objetivo, se ha tomado un enfoque de invocación de servicios realmente simple. Este enfoque es aséptico respecto a la manera en la que se

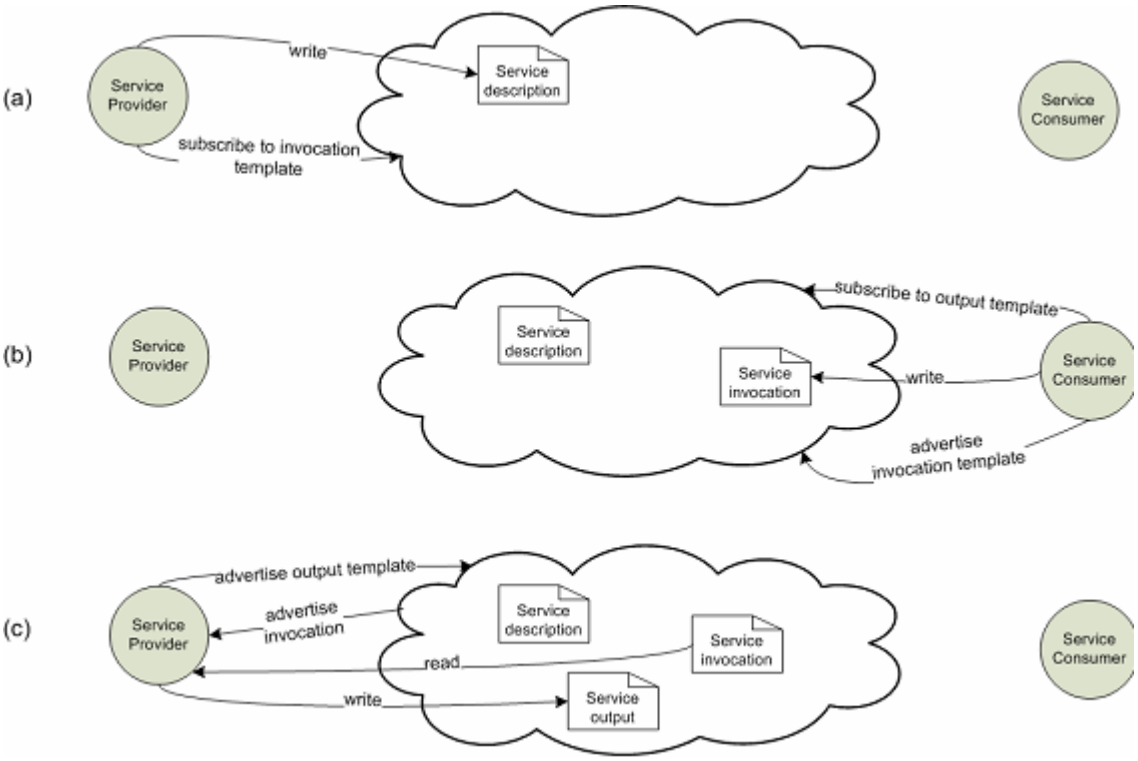

Figura 1. Servicios sobre tsc++: a) registro; b) registro desde el punto de vista del consumidor; c) invocación desde el punto de vista del proveedor de servicios.

definen los servicios semánticos. En el escenario, utilizamos nuestro propio lenguaje de definición de servicios, pero podrían utilizarse otros lenguajes.

En primer lugar, el proveedor de servicios debe registrar su servicio en el espacio (ver **figura 1**). El consumidor descubre el servicio buscándolo en el espacio. A continuación crea una invocación y utilizando el patrón *master-worker*, la escribe en el espacio y la anuncia al resto de miembros de ese espacio (ver **figura 1b**). Una invocación, básicamente, está compuesta por un identificador en forma de URI y la información de entrada que el servicio necesita.

El proveedor de servicios, que está suscrito a todas las plantillas de invocación de sus servicios, recibirá el evento (ver **figura 1c**), recuperará la información de entrada y ejecutará el servicio (normalmente realizando un cambio en el entorno usando un actuador). Cuando se completa la invocación, el proveedor podrá escribir ciertas tripletas de salida en el espacio y avisar al consumidor de manera similar a como se realizó la invocación.

3.5. 4ª etapa: *Gateway* para dispositivos empotrados

En esta etapa, hemos adaptado un *gateway* que permite la integración de los dispositivos SunSPOT en TS. El *gateway* empleado está diseñado para adaptarse al paradigma WoT [10], y provee acceso tanto a los sensores como a los actuadores de las SunSPOTs a través de servicios REST.

Además de las representaciones ofrecidas por el servidor (XML, JSON y HTML), hemos añadido una nueva que devuelve una serie de tripletas con la información semántica correspondiente con URL solicitada por el usuario. Esta representación permite la integración de los dispositivos SunSPOT en el TS simplemente reemplazando la capa que interactúa con el repositorio semántico de cualquier nodo, por otra capa que traduce las primitivas TS en llamadas HTTP dirigidas al *gateway*. Las primitivas traducidas son *read*, *write* y *query*.

La primitiva *read* tiene dos implementaciones diferentes. La primera de ellas devuelve un grafo identificado por una URI, solicitándolo al *gateway*. Por ejemplo, para obtener el grafo que describe la temperatura de una SunSPOT, la *read* debe acceder a la siguiente URL: `http://{host}:{port}/sunspots/{SpotName}/sensors/temperature/`

La segunda implementación devuelve el primer grafo del *gateway* que contenga una tripleta que coincida con la plantilla proporcionada. Para ello, solicita la siguiente URL enviando como parámetro la plantilla y el espacio: `http://{host}:{port}/read?spaceURI={spaceURI}&template={"?s ?p ?o"}` Internamente, el *gateway* recolecta todo el conocimiento disponible de las SunSPOTs en un repositorio temporal, realizando la consulta y obteniendo el resultado deseado. La primitiva *take* ha sido mapeada a *read*, ya que no tiene sentido eliminar un grafo que es generado bajo demanda en el *gateway*.

La *write* examina el contenido de las tripletas proporcionadas extrayendo los valores necesarios para realizar una POST de la siguiente manera: `http://{host}:{port}/sunspots/{SpotName}/leds/led{0-6}?switch={true/false}&redColor={0-255}&blueColor={0-255}&greenColor={0-255}`

La *query* obtiene todas las tripletas que coincidan con una plantilla específica en un espacio determinado. Para ello, se realiza una petición GET como `http://{host}:{port}/query?spaceURI={spaceURI}&template= {"?s?p?o"}` y el servidor ejecuta la consulta sobre todos los grafos disponibles.

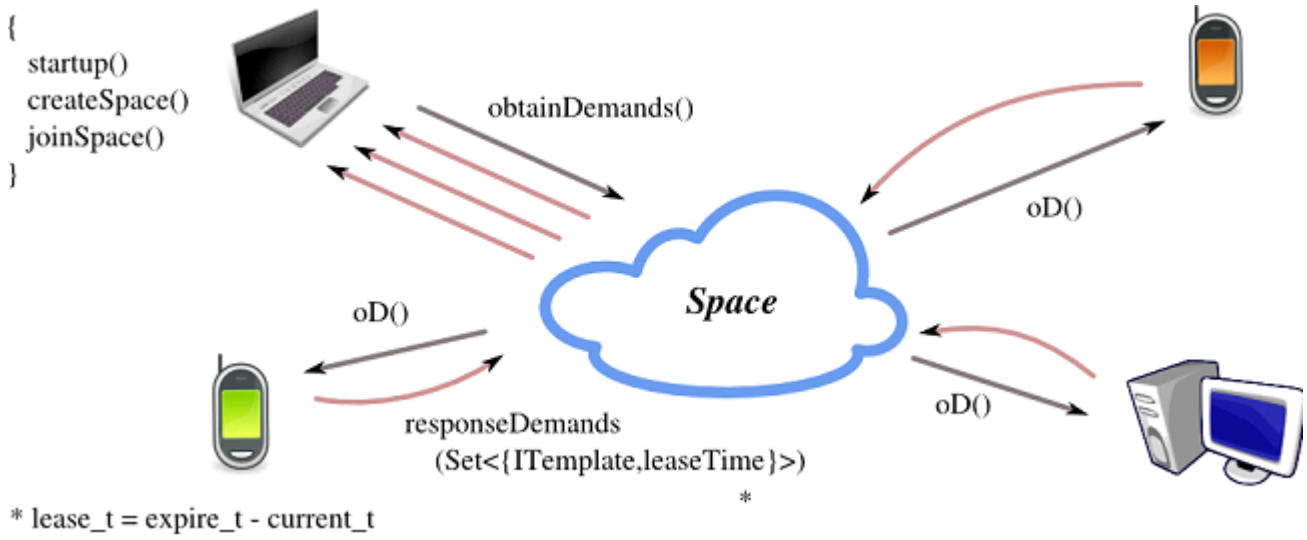

Figura 2. Un nodo que accede al espacio interroga al resto de nodos acerca de las *demands* que han recibido.

3.6. 5ª etapa: escritura remota

A pesar de que el *negative broadcasting* encaja perfectamente en escenarios en los que los nodos acceden y abandonan frecuentemente la red local compartiendo su información (por ejemplo, información acerca de sus sensores o el perfil del propietario del teléfono móvil), este enfoque presenta problemas cuando un nodo realiza cambios en el contexto a través de un actuador gestionado por otro nodo (algo muy común en *Internet of Things*). Estos problemas ya han sido discutidos en la **sección 3.4**.

La primera aproximación para resolver este problema, descrita anteriormente, consistía en la utilización de servicios sobre *TS*. El problema inherente es el mismo que existe entre grandes servicios web y los servicios REST [16], una excesiva complejidad. Como se describía en [17], el paradigma TS es básicamente un paradigma orientado a objetos, y ya que todas las primitivas utilizan recursos semánticos (triples) como base, resulta razonable buscar una solución consistente con el paradigma, usando tripletas como base. Para mantener una API simple para el desarrollador, hemos modificado la implementación de la primitiva *write* para que el desarrollador no deba contemplar el tipo de escritura a usar.

Para ello, se ha creado una primitiva *demand* que permite a cada nodo reflejar su responsabilidad sobre una porción de conocimiento utilizando una plantilla. La primitiva *write* se ha sobreescrito para enviar el conocimiento que se trata de escribir a otros nodos del espacio, siempre que se sepa que un conjunto de tripletas son responsabilidad de otro nodo. Para saber si el conocimiento a escribir es responsabilidad de otro nodo, basta con comprobar que alguna de las plantillas anunciadas con *demand* coincida con alguna de las tripletas que se pretenden escribir.

La primitiva *demand* tiene un tiempo de vida máximo, tras el cual el nodo la eliminará de su registro para impedir la acumulación indefinida de ellas. Los nodos que acceden a un espacio interrogan al resto para conocer las responsabilidades de otros nodos del espacio (ver **figura 2**).

Finalmente, *write* se comporta de la siguiente manera:

- Si el *nodo a* intenta escribir un conjunto de tripletas y conoce que otro nodo es responsable del conocimiento que está intentando escribir, el *nodo a* sugerirá al resto de nodos que quiere modificar ese conocimiento con ese conjunto de tripletas. Dicha sugerencia de cambio realiza a través de una operación transparente al usuario del *middleware* llamada *suggest* (ver **figura 3**). Cada nodo que haya ejecutado una *demand* que coincida con las tripletas proporcionadas, llamará a un método de *callback* pasándole el conjunto de tripletas original. En este método, se alterará el entorno utilizando un actuador y si es necesario, se reflejará dicho cambio en la base de conocimiento.

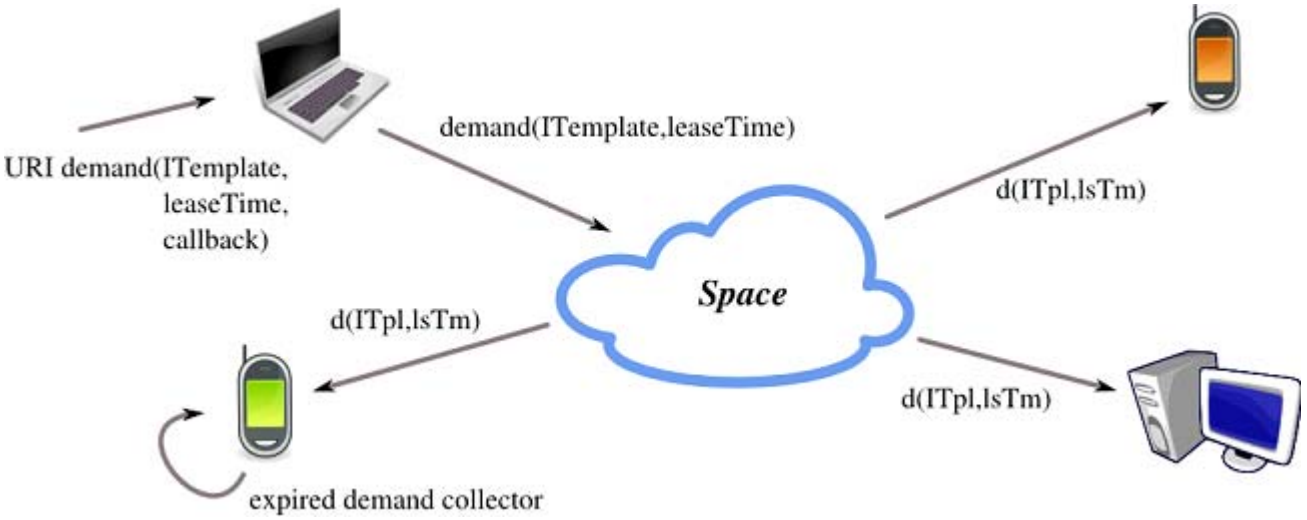

Figura 3. Cada nodo envía una *demand* periódicamente para renovar su responsabilidad sobre un fragmento de conocimiento.

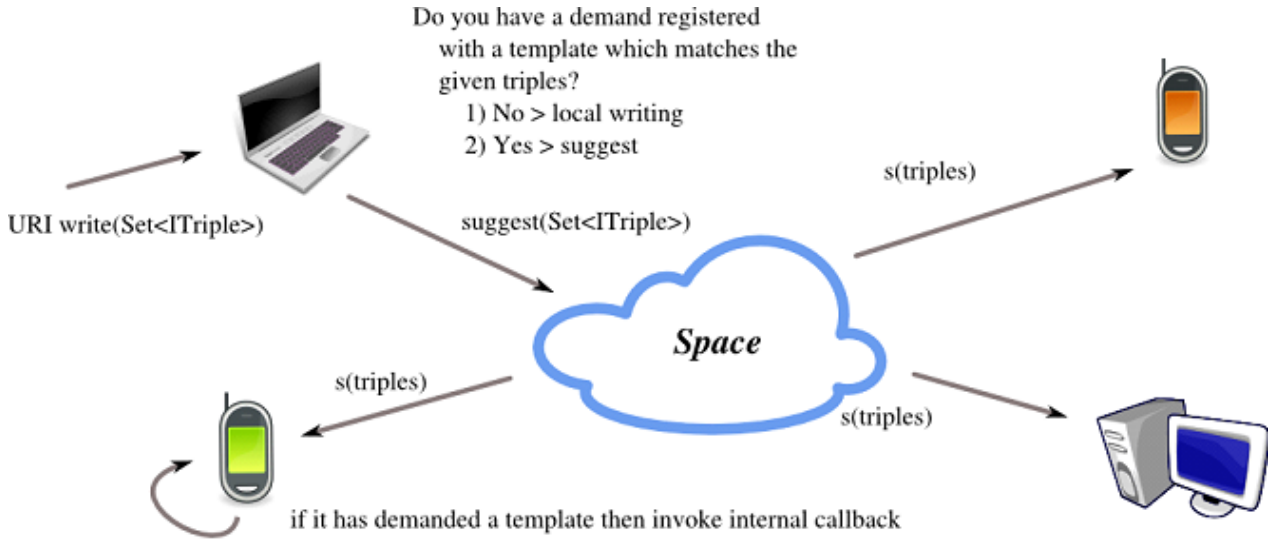

Figura 4. Nuevo comportamiento de escritura.

■ Si nadie ha reclamado la responsabilidad de este conocimiento se escribirá en el repositorio semántico local (ver **figura 4**).

4 . Entorno experimental

Existen muchos escenarios de automatización del hogar o de instrumentación urbana en los que el *middleware* propuesto será de utilidad. Uno de estos casos podría ser el control de la temperatura de una habitación. En este escenario, que utiliza todas las primitivas descritas anteriormente, hay por lo menos 5 nodos, que pueden observarse en las **figuras 5 y 6**.

Hay dos proveedores de contexto diferentes: las SunSPOTs y el proveedor meteorológico. Los primeros son dispositivos físicos que comparten la información detectada a través del nodo descrito en la **sección 3.5**. Por otra parte, el proveedor meteorológico es un proveedor de contexto virtual que obtiene la información de Internet y la escribe de forma semántica en el espacio. Para ello, el nodo sondea el espacio en busca de nuevos entornos, y comprueba si Yahoo! dispone de una *id* de ubicación para ese entorno. En caso afirmativo, consulta el servicio Yahoo! Weather[4], obtiene la temperatura actual para esa ubicación, crea las instancias de la ontología usada en el escenario que representan dicha información y escribe el conocimiento en TS. El espacio dispone también de un actuador, un aire acondicionado con el que se regulará la temperatura interior del entorno en el que se encuentre, activándolo en caso de necesitar enfriarlo. La temperatura actual del interior o exterior del entorno se puede monitorizar usando un teléfono móvil.

El nodo regulador utiliza toda esta información cada vez que alguien se encuentra en una ubicación para decidir si debe enfriarla. Para deducir la presencia de alguien en una sala se consultará en el espacio si hay un móvil en la misma que pertenezca a alguien, mientras que para regular la temperatura, el nodo regulador deberá comprobar si existe un sistema de aire acondicionado en la ubicación que se quiere regular, encendiéndolo en caso afirmativo.

Algoritmo básico de control de temperatura

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

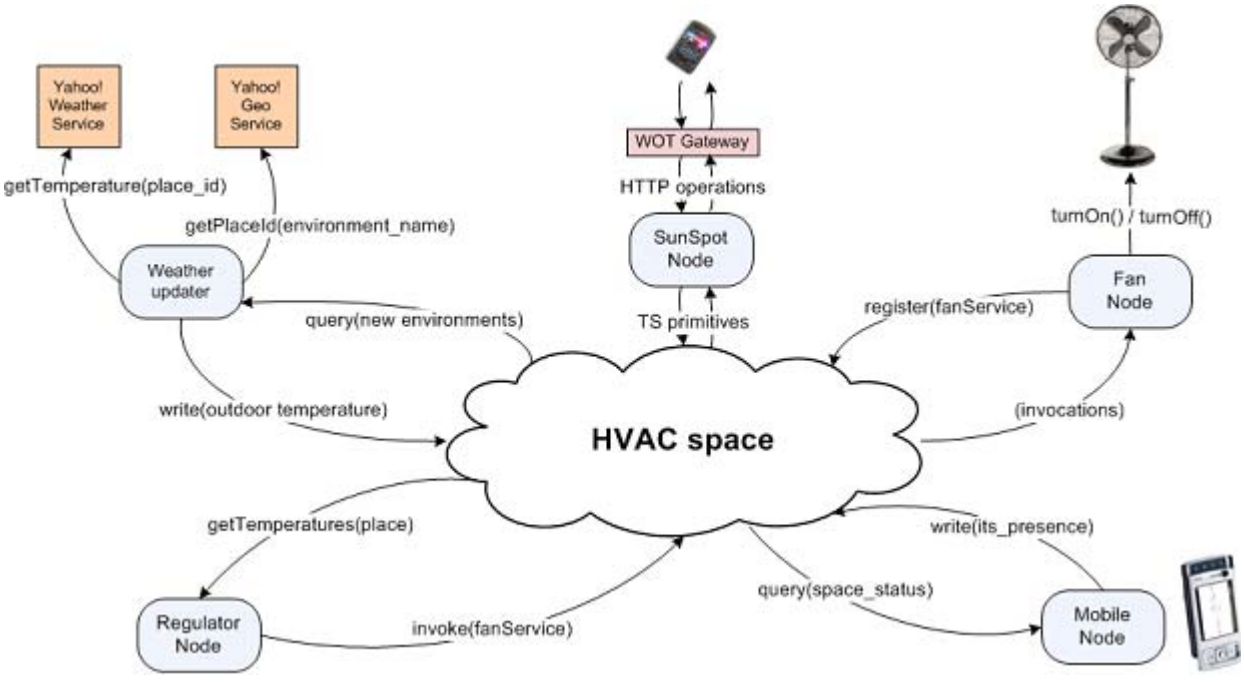

Figura 5. Implementación del escenario descrito utilizando la aproximación basada en servicios.

Para implementar este escenario se pueden utilizar dos enfoques diferentes, en función de qué método utilicen para realizar los cambios sobre el entorno. El primero de ellos (ver **figura 5**) utiliza las primitivas de servicio descritas anteriormente para crear un servicio que encienda o apague el aire acondicionado conectado al nodo. El nodo regulador consumirá dicho servicio siempre que es necesario. El segundo enfoque (ver **figura 6**) se basa en la primitiva *demand* explicada anteriormente. El nodo SunSPOT demanda una plantilla relacionada con los LEDs de las SunSPOTs controladas y el ventilador realizará lo propio con una plantilla relacionada consigo mismo. Cuando el nodo regulador decide que la temperatura interior debe decrementar, tratará de escribir dicho conocimiento en el espacio, pero si tanto los nodos de las SunSPOTs y el del ventilador han reclamado la responsabilidad sobre ese conocimiento, la primitiva se transformará en una primitiva *suggest*. Cuando reciban esa petición de cambio, los nodos SunSPOT y ventilador deberán decidir qué hacer con dicha información. En este ejemplo, el nodo SunSPOT enciende o apaga todos los LEDs de todas las SunSPOTs conectadas al *gateway*, por lo que su estado es actualizado automáticamente, y el nodo ventilador lo apaga o enciende actualizando su estado.

Este escenario es únicamente una demostración de conceptos. Técnicamente, se puede introducir un SWRL (*Semantic Web Rule Language*) para definir las reglas de control de temperatura de una manera más expresiva y con menor acoplamiento.

5. Experimentación

En esta sección, se evaluarán la nueva primitiva *demand* y el *gateway* SunSPOT. El resto de las primitivas ya fueron evaluadas en [12] para tscME y en [5] para tsc++. Finalmente, el escenario presentado en la sección anterior ha sido implementado utilizando la primitiva *demand* y ha sido comparada con la solución basada en servicios evaluada en [12].

5.1. Primitiva demand

En la **tabla 2** se puede apreciar como la primitiva *demand* se comporta como una primitiva de red, por lo que sus medidas representarán el tiempo que necesita para propagarse por la red hasta alcanzar al nodo responsable, dato que puede variar en función del estado de la red. La escritura queda ligeramente penalizada por el proceso en el que el nodo decide si debe escribir en su repositorio local o propagar una primitiva *suggest*. En cualquier caso, está cerca de ser insignificante: para 30 *demands* sólo se demora 20 ms. más en comparación con las mediciones obtenidas en la versión previa de esta primitiva. Cuando la escritura se propaga utilizando *suggest*, el tiempo necesario es similar al tiempo de propagación.

De acuerdo con estas mediciones, podemos asumir que la penalización sufrida al utilizar la nueva implementación de la *write* es suficientemente leve como para utilizarla en lugar de la implementación original.

5.2. *Gateway* SunSPOT

Como puede verse en la **tabla 3**, el número de SunSPOTs conectadas no afecta al rendimiento de las primitivas *write* y *read (dada una URL)*, ya que el *gateway* accede directamente a la URL requerida por ellas. Con la *read (dado una plantilla de consulta)* y *query*, sin embargo, el retardo aumenta en función del incremento de SunSPOTs conectadas, ya que el *gateway* debe recolectar todos los grafos de todos los recursos de cada SunSPOT para filtrar los resultados utilizando la plantilla proporcionada.

5.3. Escenario

Para poner en marcha los dos escenarios comentados en la sección anterior, se han configurado los nodos para demorar por un segundo la recepción de las respuestas, dado que el *middleware* es asíncrono y no se puede prever de cuando se recibirán las respuestas. En cualquier caso, el rendimiento de este escenario podrá ser mayor del que se ha mostrado disminuyendo dicho tiempo de espera.

En la **tabla 4** puede apreciarse el tiempo necesario para la activación del aire acondicionado en cada implementación del escenario. Con la nueva implementación de *write* la invocación se realiza más rápidamente. Además, esta diferencia se hace más notable en la medida en la que se intenten activar más máquinas de aire acondicionado. Mientras la invocación mediante servicios tardará lo que tarden en completarse tantas invocaciones como máquinas se quieran activar, la write seguirá necesitando una única llamada y tardando lo mismo.

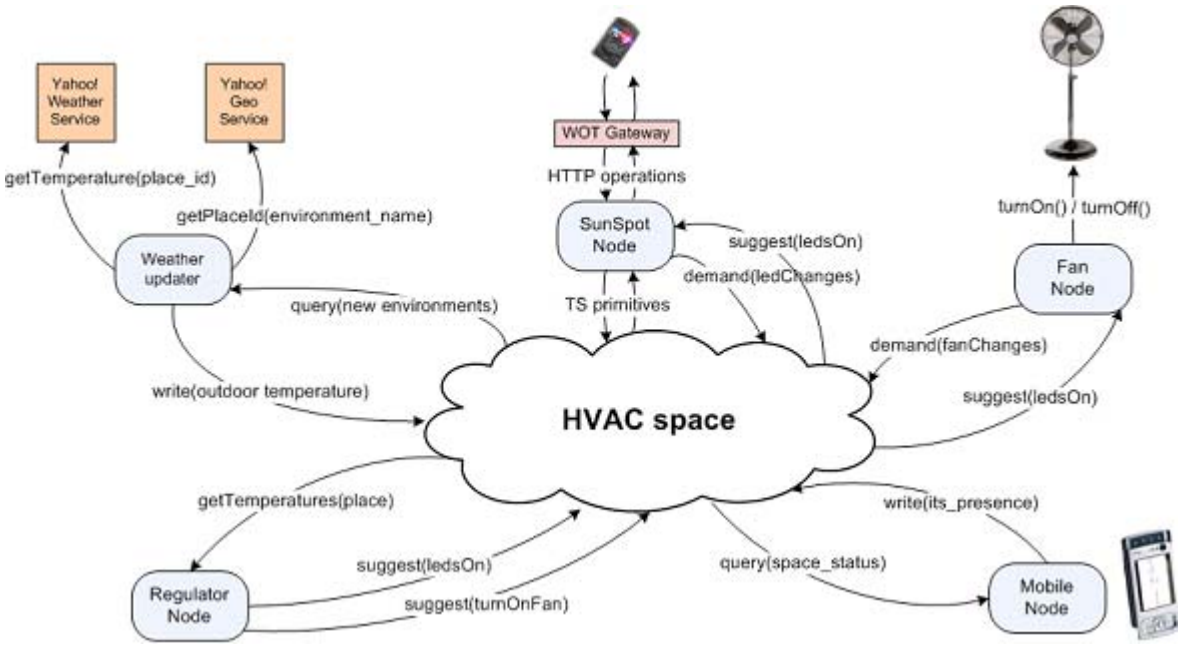

Figura 6. Implementación del escenario descrito utilizando la aproximación basada en recursos.

Núm. de *demands* registradas		10	20	30
Versión original	write	0.004	0.004	0.004
Versión actual	demand	0.096	0.099	0.098
	write	0.006	0.015	0.027
	suggest	0.133	0.159	0.186

Tabla 2. Rendimiento de la primitiva *demand* en un nodo móvil (en segundos).

Núm. grafos		10			
Núm. 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

Tabla 3. Rendimiento del WOT *gateway* (en segundos).

Acción	Implementación basada en *Demand*	Implementación basada en servicios
Descubrir nuevas localizaciones en el espacio	5.38	
Actualizar las medidas meteorológ. para una ubicación desconocida	1.91	
Actualizar las medidas meteorológ. para una ubicación conocida	1.18	
Comprobar cambios en las temperaturas. interior y exterior	10.5	
Activación del aire acond. Desde q el gestor de temp. invoca su servicio	0.90	1.45

Tabla 4. Medidas de tiempo para el escenario propuesto (en segundos).

Finalmente, desde el punto de vista cualitativo se ha experimentado mayor sencillez al desarrollar el escenario con la aproximación basada en *demand* que con la aproximación basada en servicios.

6. Conclusiones y líneas de trabajo futuras

Este artículo explora la posibilidad de trasladar la computación distribuida basada en *Triple Space* a los sistemas ubicuos, en el que un gran número de dispositivos heterogéneos comparten conocimiento de manera asíncrona y orientada a recursos, lo cual encaja perfectamente con la idea de *Internet of Things*.

Los resultados obtenidos de nuestro escenario estereotipado prueban que el *middleware* posee un rendimiento aceptable. Aún así, se debería realizar una evaluación más exhaustiva para comprobar otras características como la escalabilidad de la solución y explorar cómo afecta en los dispositivos empotrados el uso continuo del *middleware* en aspectos críticos como el consumo de batería.

Además, podría mejorarse fácilmente el *middleware*, haciendo a los dispositivos independientes del rendezvous utilizando multicast y asegurando que todos ellos pudiesen llevar a cabo algún tipo de razonamiento.

En el futuro, se contempla crear nuevos *gateways* para diferentes dispositivos empotrados con el mismo enfoque utilizado para las SunSPOTs, desarrollar un razonador para teléfonos móviles, considerar nuevas alternativas para la capa de red y explorar los problemas de seguridad del *middleware* propuesto. Finalmente, se realizará un análisis de rendimiento tanto en simulador como en un escenario real, con un numero de dispositivos más elevado.

7. Agradecimientos

Este proyecto ha sido financiado con la subvención PC2008-28 A del departamento de Educación, Universidades e Investigación del Gobierno Vasco para el periodo 2008-10.

Referencias

[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 et al., 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, D. Manov. OWLIM - a pragmatic semantic repository for OWL. *Web Information Systems Engineering -WISE 2005 Workshops*, pp. 182-192, 2005.

[9] B. Traversat et al., 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*, page 433â"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 et al., Triple space computing: Adding semantics to space-based computing. *The Semantic Web-ASWC 2006*, pp. 300-306, 2006.

Notas

[1] TripCom (IST-4-027324-STP), <www.tripcom.org>.
[2] <https://jxta.dev.java.net/>.
[3] <https://jxta-jxme.dev.java.net/>.
[4] <http://developer.yahoo.com/weather/>.