UNIVERSIDAD DE GRANADA

Programa Oficial de Posgrado en Desarrollo de Sistemas de Software
(P39.56.1)

T E S I S D O C T O R A L

Plataforma de Servicios Semánticos Sensibles al Contexto para Sistemas de Inteligencia Ambiental

Defendida por
Sandra Rodríguez-Valenzuela

Director
Juan A. Holgado-Terriza

Departamento de Lenguajes y Sistemas Informáticos
Grupo de Investigación en Sistemas Concurrentes

Diciembre, 2015

Editor: Universidad de Granada. Tesis Doctorales
Autora: Sandra Sheila Rodríguez Valenzuela
ISBN: 978-84-9125-460-7
URI: http://hdl.handle.net/10481/42153

La doctoranda Sandra Rodríguez-Valenzuela y el director de la tesis Juan A. Holgado-Terriza. Garantizo, al firmar esta tesis doctoral, que el trabajo ha sido realizado por la doctoranda bajo la dirección del director de la tesis y hasta donde nuestro conocimiento alcanza, en la realización del trabajo, se han respetado los derechos de otros autores a ser citados, cuando se han utilizado sus resultados o publicaciones.

Granada, 30 de octubre de 2015

Director de la Tesis — Doctoranda

Fdo.: Juan A. Holgado Terriza — Fdo.: Sandra Rodríguez-Valenzuela

La realización de esta Tesis ha sido posible gracias a la financiación del Programa Estatal de Promoción del Talento y su Empleabilidad a través del Subprograma de Formación de Profesorado Universitario (FPU) (Referencia AP2009-2239).

A mi familia.

Índice general

Resumen

Se ha diseñado y desarrollado una plataforma de servicios semánticos sensibles al contexto considerando las restricciones propias de los sistemas ubicuos. La plataforma posee una arquitectura descentralizada y distribuida basada en SOA. Las decisiones de diseño tomadas durante su construcción han estado muy ligadas al concepto de servicio como elemento fundamental.

Así, se ha diseñado un modelo de composición entre servicios basado en grafos dirigidos acíclicos. Dicho modelo de composición asegura la inexistencia de bucles indefinidos en la colaboración entre servicios. Dicha colaboración es estática, pues el usuario define a-priori los servicios que componen el grafo de composición, con el pre-requisito de satisfacer las restricciones impuestas por el grado de complejidad de las operaciones involucradas. El grado de complejidad de una operación es un valor entero que establece el máximo número de operaciones en cascada que son invocadas por un servicio, desde el inicio hasta el final de la ejecución de una operación colaborativa.

Una vez definido un modelo de composición verificable, se ha dotado a los servicios de propiedades de calidad de servicio QoS. Considerando la propiedad de tiempo real, y partiendo de la base de que el modelo de composición ya está establecido y validado, es posible determinar el tiempo de ejecución de cada servicio y añadir esta información al árbol de composición, de forma que el tiempo de ejecución de la aplicación total queda acotado por el tiempo de ejecución de cada servicio que la compone.

A más alto nivel, se ha dotado a los servicios de propiedades semánticas haciendo uso de las ontologías planteadas por el estándar OWL-S. La conjugación de la anatomía de servicios planteada, las propiedades de calidad de servicio o QoS y su representación semántica, posibilita la selección dinámica de servicios. También a este nivel, se ha considerado una ontología para modelar el contexto y dotar a los servicios de propiedades de sensibilidad al contexto. En base a esto, los usuarios pueden determinar el comportamiento proactivo de los servicios mediante la configuración de perfiles de comportamiento personalizados que establecen sus preferencias con respecto al comportamiento del sistema.

La implementación de la plataforma y su utilización en el desarrollo de distintos casos de estudio ha permitido llevar a cabo la evaluación de los principales elementos de la plataforma, como son el modelo de composición y los perfiles de comportamiento proactivo. Las conclusiones obtenidas de dicha evaluación avalan el uso de la plataforma desarrollada para la construcción de aplicaciones para computación ubicua.

Lista de Figuras

Lista de Tablas

Lista de Algoritmos

Lista de Códigos

Lista de Acrónimos

AAL Ambient Assited Living

ADC Analog-to-digital converter

API Application Programming Interface

BIGroup Business Informatic Group

BWCET Best Worst-Case Execution Time

CAS Context Aware Service

CDC Connected Device Configuration

CLDC Connected Limited Device Configuration

COBRA Context Broker Architecture

COBRA-ONT Ontology for Context Broker Architecture

CoDAMoS Context-Driven Adaptation of Mobile Services

CONON CONtext ONtology

CSC Computación Sensible al Contexto

CU Computación Ubicua

DAML+OIL DARPA Agent Markup Language + Ontology Inference Layer

DCU Dublin City University

DOHA Dynamic Open Home Automation

DPWS Device Profile for Web Services

DQ Data Quality

DS Design Science

GPIO General-purpose input/output

GUI Graphical User Interface

HNS Home Network System

IoT Internet of Things

J2ME Java 2 Microedition

JavaES Java for Embedded Systems

JDOMO Java Home-Automation

M2M Machine-to-Machine

MDA Model-Driven Architecture

MDD Model-Driven Development

MOF Meta-Object Facility

MOM Message Oriented Middleware

NAT Network Address Translation

OWL Ontology Web Language

OWL-S Web Ontology Language for Services

OWLS OWL Schema

P2P Peer-to-Peer

PARC Xerox Palo Alto Research Center

PFC Proyecto Fin de Carrera

PFM Proyecto Fin de Máster

QoS Quality of Service

RDF Resource Description Framework

RDFS RDF Schema

RRM Request/Response Mode

SCA Server Centralized Architecture

SCAS Semantic Context Aware Service

SenSE SENsitive Service ENvironment: capa semántica de la plataforma de servicios

SGML Standard Generalized Markup Language

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SOUPA Standard Ontology for Ubiquitous and Pervasive Applications

SPARQL SPARQL Protocol and RDF Query Language

SS Semantic Service

SSOA Semantic Service Oriented Architecture

TIC Tecnologías de la Información y las Comunicaciones

TICS Tecnologías de la Información y las Comunicaciones

UDDI Universal Description, Discovery and Integration

UML Unified Modeling Language

URI Uniform Resource Identifier

URN Uniform Resource Name

VM Virtualized Mode

W3C World Wide Web Consortium

WCET Worst Case Execution Time

WS4D Web Services for Devices

WSDL Web Services Description Language

WWW World Wide Web

XmI XML Metadata Interchange

XML Extensible Markup Language

XSD XML Schema Definition

Parte I

Preliminares

Capítulo 1

Introducción

Research is what I am doing when I don't know what I am doing.

Wernher von Braun (March 23, 1912 - June 16, 1977) In an interview in the New York Times, 16 December 1957.

1.1. Dónde estamos y hacia dónde vamos

Nuestro mundo está cada vez más interconectado. A nuestro alrededor están proliferando gran cantidad de dispositivos de pequeño tamaño con capacidad de comunicación y computación denominados dispositivos empotrados [Gubbi et al., 2013].

Paradigmas como la domótica u hogar-digital, de gran auge unos años atrás, han quedado relegados en pro de otras propuestas cada vez más ambiciosas surgidas de la mano de esta tendencia. Entre estas destacan el Internet de las Cosas, conocido como Internet of Things (IoT) [Atzori et al., 2010], y la comunicación entre dispositivos o Machine-to-Machine (M2M) [Kim et al., 2014]. Aunque estos aspectos ya eran conocidos por gran parte de la sociedad gracias a su presencia en la ciencia ficción [Dourish and Bell, 2014], a lo largo de las dos últimas décadas se han realizado grandes esfuerzos a nivel de investigación, político y social para conseguir convertir esta tendencia en una realidad [Aarts and de Ruyter, 2009]. Sin embargo, encontrar la convergencia perfecta entre las diferentes tecnologías implicadas a nivel hardware, software y de redes de comunicaciones se ha convertido en un escollo difícil de solventar que requiere la coordinación efectiva entre campos como la electrónica, automática, telecomunicaciones e informática. Y en este punto, a pesar de la gran cantidad de nuevos paradigmas y tendencias de investigación en el área, el concepto que mejor define a esta tendencia es el de computación ubicua (CU) [Zhang et al., 2013].

A pesar de los esfuerzos realizados y del gran número de propuestas que han surgido desde sectores académicos, industriales y políticos, el sector no ha sido capaz de definir un estándar considerado "universal" [Stavropoulos et al., 2013]. Son muchos los factores que han influido en ralentizar o dificultar la aceptación e implantación de este tipo de sistemas. Desde los conflictos suscitados en torno a los conceptos de seguridad y privacidad, hasta la adecuación de su uso al usuario final, que aunque cada vez está más familiarizado con la tecnología, es a su vez más exigente con ella.

La complejidad subyacente en este tipo de sistemas es también un hándicap. El complejo entramado de dispositivos que da soporte a estos sistemas suele ser heterogéneo y diferir además en sus características software, e incluso también en las redes de comunicaciones utilizadas [Branca and Atzori, 2012]. Si además tenemos en cuenta que para el usuario final estos dispositivos ofrecen un conjunto de servicios, debemos considerar que estos pueden multiplicarse exponencialmente a raíz de la aparición de nuevos servicios como resultado de la composicionalidad de otros más simples. Todo ello hace que estos sistemas puedan convertirse en algo inabordable tanto desde el punto de vista del desarrollo como de su utilización.

A nivel de investigación se han realizado grandes esfuerzos para adecuar el desarrollo de software a los requisitos específicos de este tipo de sistemas. Uno de los paradigmas de uso más extendido en el desarrollo de sistemas ubicuos es Service Oriented Architecture (SOA) [Erl, 2005], en el cual se basa el desarrollo propuesto en esta tesis. Mediante la aplicación de una arquitectura SOA se pretende que la funcionalidad ofrecida por los nodos computacionales se interprete como un servicio que puede combinarse con otros para obtener aplicaciones más complejas.

1.2. Computación ubicua

El término *Computación Ubicua* se acuñó a partir de los artículos publicados por Mark Weiser a principios de los años 90 [Weiser, 1991][Weiser, 1993]. A él se le considera el padre de un paradigma que 20 años después sigue generando controversia a nivel de investigación. En 1988, cuando Weiser trabajaba en el Xerox Palo Alto Research Center (PARC), vaticinó, influenciado por campos tan dispares como la filosofía, antropología, psicología, y haciendo claras alusiones al ideal de invisibilidad propuesto en la novela Ubik de Philip K. Dick, que en un futuro no muy lejano estaríamos rodeados de dispositivos, todos ellos inteligentes e interconectados, desde los pomos de las puertas, hasta el papel higiénico. Aunque la investigación no avanza a la par de la ciencia ficción, se han realizado grandes avances en el área, siendo la computación ubicua el eje central de muchos de los proyectos de investigación actuales y uno de los objetivos a alcanzar en el ámbito de las ciencias de la computación en los próximos años [Commission, 2014].

El término computación ubicua está ligado a otros tantos, como por ejemplo computación pervasiva, inteligencia ambiental, computación móvil y computación sensible al contexto, entre otros. Considerando todas las definiciones contenidas en la literatura, para nosotros un espacio ubicuo es aquel que está repleto de dispositivos de computación, transparentes para los individuos y con los que pueden interactuar de forma natural, sin ser completamente conscientes de la complejidad subyacente.

Uno de los principios fundamentales de la computación ubicua y de la inteligencia ambiental es que los dispositivos dejen atrás su comportamiento automático, su papel de artefactos que el usuario puede utilizar, y pasen a tener un comportamiento activo, con consciencia del entorno en el que se encuentran, en definitiva, un comportamiento inteligente [Weiser, 1993]. Con recursos muy limitados y pequeño tamaño,

los dispositivos se presentan como objetos autónomos que requieren una mínima intervención de los usuarios. Se encuentran distribuidos e interconectados físicamente y desarrollan un comportamiento colaborativo, son capaces de formar coaliciones para ofrecer más funcionalidad al usuario [Geyik et al., 2013]; y proactivo, son sensibles a los cambios en la información del entorno, como puede ser la localización y estado de los usuarios y dispositivos [Saha and Mukherjee, 2003]. Se pretende que la funcionalidad ofrecida por los nodos computacionales se interprete como un servicio que puede combinarse con otros para obtener aplicaciones más complejas, lo cual, como ya se ha comentado, encaja adecuadamente dentro del paradigma SOA.

Una diferencia fundamental entre la programación tradicional y las aplicaciones para entornos ubicuos reside en la capacidad de adaptación al contexto, pues es necesario que estos sistemas sean sensibles a los cambios en la información del entorno. El concepto de computación sensible al contexto o context-aware ha cobrado una gran relevancia en nuestro tiempo, asociada al uso de dispositivos móviles y entornos dinámicos [Lassila, 2005] [Perera et al., 2014]. El conocimiento del entorno, la forma de representar el contexto y la información del mundo físico que es interpretable por parte de un sistema repercutirá en su grado de "inteligencia" ambiental e independencia del usuario. Debido a la diversidad de la información de contexto, la mayoría de los enfoques aplicables a este aspecto tratan de obtener una representación uniforme del entorno, utilizando lenguajes de consulta y algoritmos de razonamiento. Utilizar información contextual del mundo físico tiene una serie de implicaciones a tener en cuenta relacionadas con la captación y procesamiento de la información, pues se debe distinguir qué tipo de información es factible detectar, cuál es la manera de adquirir la información de los sensores y cómo se razona sobre la información obtenida para inferir contexto [Loke, 2006] [Baldauf et al., 2007]. En la bibliografía especializada se distinguen varios modelos en función de la estructura de datos utilizada para el manejo de la información contextual, de entre los que se deben destacar aquellos que sean más factibles en nodos de cómputo con recursos muy limitados.

Otro tipo de investigaciones, también relacionadas con la utilización de información de contexto, tratan de estandarizar términos y sembrar las bases estables de un desarrollo sostenible en la materia. Es el caso de [Ou et al., 2006], que enfatiza la importancia de emplear criterios estándares para llevar a cabo el proceso de desarrollo de los sistemas sensibles al contexto. En su trabajo, Ou aplica MDA para hacer frente al desarrollo de estas aplicaciones, estableciendo un modelo con dos niveles para la obtención de una ontología de contexto, que junto a un modelo de integración, permita generar implementaciones de forma semiautomática.

Dentro de las características básicas de un sistema de computación ubicua se pueden destacar las siguientes:

- Posee una interfaz invisible que exige una simplificación de la utilización de los abundantes procesadores que forman parte del entorno y que utiliza métodos de captura novedosos.
- Utiliza estándares abiertos, lo cuál es muy aconsejable cuando se trabaja con

equipos de gran cantidad y diversidad de desarrolladores, multitud de fabricantes de dispositivos, cuyo número es desconocido a priori y que requieren una mínima o nula configuración manual.

- Es necesario que el hardware a utilizar sea de bajo coste, pues el número de dispositivos a utilizar será elevado; sea miniaturizado, para garantizar la transparencia; y de bajo consumo, pues el sistema deberá estar funcionando constantemente.

- El sistema debe ser reactivo, es decir, reaccionar al ocurrir determinados eventos, y proactivo, inferir los deseos y necesidades del usuario, anticipándose a ellos y reduciendo al máximo la emisión consciente de órdenes por parte del usuario, a ser posible, aprendiendo de los posibles errores cometidos.

- Lleva a cabo acciones de forma local, es decir, se llevan a cabo dentro del espacio perspicaz tras la detección de la presencia del individuo y de sus necesidades, en función de las cuales se llevará a cabo una u otra funcionalidad. También se pretende que haya una transición suave entre distintos entornos, es decir, que se minimicen las fronteras entre los espacios que forman parte del entorno ubicuo.

1.3. Plataformas de Servicios

Tanto los sistemas de inteligencia ambiental como la computación sensible al contexto giran en torno al concepto de computación ubicua. Weiser define las principales líneas de investigación en computación ubicua [Weiser, 1991], centradas inicialmente en problemas relacionados con el hardware: reducción de tamaño y consumo de energía, capacidad de procesamiento, protocolos de comunicación inalámbricos e invisibilidad, convertir los dispositivos en objetos imperceptibles. La vertiginosa evolución de los dispositivos empotrados ha dirigido la atención a otras áreas como la computación móvil y las redes de sensores. El incremento en recursos del hardware, tanto en procesamiento y memoria, como en calidad de las comunicaciones, permite aprovechar los avances en tecnología software alcanzados en campos de aplicación como sistemas de información empresariales y sistemas distribuidos.

Las características fundamentales de un entorno ubicuo son: capacidad de reconfiguración dinámica, modularidad, extensibilidad y portabilidad [Weiser, 1993] [Weiser and Brown, 1997]. Es evidente que tanto el modelo de arquitectura como la infraestructura software utilizadas juegan un papel esencial. Las características de los entornos ubicuos hacen que su desarrollo requiera de nuevos modelos de computación específicos, y por tanto, de nuevas infraestructuras software que den soporte a este tipo de aplicaciones, teniendo en cuenta que estas deben ser integradas en entornos de ejecución empotrados, dispositivos con recursos limitados. El cumplimiento de estos requisitos pasa por la utilización de una arquitectura software adecuada. Actualmente los principios del paradigma SOA son los más extendidos y utilizados y

establecen una serie de consideraciones o atributos que deben presentar los servicios desarrollados según este enfoque [Stojanovic and Dahanayake, 2005].

Dependiendo de donde se encuentre el componente principal o controlador, la arquitectura de un sistema puede ser centralizada o descentralizada [Sommerville, 2005]. En la arquitectura centralizada un controlador central recibe información de múltiples sensores y, una vez procesada, genera las órdenes oportunas para los actuadores. El hecho de que no exista conexión directa entre todos los componentes hace que un fallo en la unidad central provoque el fallo de todo el sistema. En las arquitecturas descentralizadas, no existe la figura del controlador centralizado, sino que toda la inteligencia del sistema está distribuida entre todos los módulos, sean sensores o actuadores. Desde el punto de vista de un sistema ubicuo, este tipo de arquitectura facilita la escalabilidad del sistema y la incorporación y eliminación sencilla de componentes. Otro aspecto a tener en cuenta es el modelo de programación a utilizar. En computación ubicua el más extendido es el modelo distribuido. Por su utilización destacan los modelos centralizados de pizarra y de repositorio, y los cliente/servidor y peer-to-peer como descentralizados.

La selección de un estilo arquitectónico y un modelo de programación concretos, requiere además de un middleware adecuado que permita implementar los componentes de la arquitectura del sistema y el modelo de comunicación entre componentes. La plataforma middleware de un sistema ubicuo y de inteligencia ambiental debe resolver problemas como el descubrimiento dinámico de servicios, la selección, composición y adaptación de los servicios; y la interoperabilidad, heterogeneidad y transparencia en cuanto a los dispositivos [Banavar et al., 2000][Guinard et al., 2010].

1.4. Información, semántica y contexto

No existe una definición generalmente aceptada y consensuada que permita definir el término de contexto en un entorno de computación ubicua. Sin embargo, las definiciones existentes coinciden en que este concepto está muy ligado al conjunto de propiedades que caracterizan al entorno, con condiciones interrelacionadas en las que tienen lugar eventos, acciones o situaciones que también afectan a los individuos, y que no intervienen de manera explícita en la resolución de un problema, pero sí la restringen. El concepto de contexto es muy relativo y se podrían distinguir distintos tipos posibles, como por ejemplo computacional, físico, de usuario, de tiempo o de grupo.

La Computación Sensible al Contexto (CSC) trata situaciones contextuales, en las que confluyen un conjunto diverso de parámetros cambiantes, mediante la detección de la información del contexto. Esta información, una vez recogida, es utilizada en beneficio del sistema que será capaz de reaccionar de forma adecuada [Raz et al., 2006] [Madkour et al., 2013]. Existen muchos campos de aplicación de este tipo de información, como por ejemplo el estudio y tratamiento de localización para aplicaciones móviles o la computación emocional/afectiva. Todo ello con la filosofía

subyacente de utilizar el contexto para interpretar las necesidades e intenciones del usuario, y adaptar la aplicación y su interfaz de acuerdo a estas.

Los términos de computación ubicua y computación sensible al contexto están muy relacionados. Mientras que la CU pretende ofrecer ayuda lo más personalizada posible a los usuarios, la CSC tiene como objetivos principales tareas como la identificación de los usuarios, averiguar su ubicación, inferir sus deseos y necesidades, y actuar proactivamente, todo ello mediante el manejo de la información contextual.

El conocimiento del contexto del sistema debe permitir a los servicios tomar medidas automáticamente, reducir la participación directa de los usuarios, disminuyendo la carga comunicativa del sistema, y la prestación de asistencia proactiva inteligente [Vazquez et al., 2007] [Santofimia et al., 2011]. En una plataforma de servicios se deben tener en cuenta aquellos aspectos del entorno que pueden favorecer el comportamiento inteligente de los servicios, fomentando en su diseño los aspectos relacionados con la sensibilidad al contexto. El conocimiento del entorno, la forma de representar el contexto y la información del mundo físico que es interpretable por parte de un sistema repercutirá en el grado de "consciencia" que tengan sus elementos. Por ello, la forma de modelar este entorno es importante.

1.5. Objetivos y Estructura de esta Tesis

En este trabajo se plantea como objetivo principal el diseño de una plataforma de servicios para ambientes de interacción basado en espacios de inteligencia ambiental, ubicua y móvil que de cobertura especialmente a los espacios habitables que se demandan en la nueva generación de sistemas domóticos. La plataforma de servicios proporcionará metodologías y estrategias para la definición, diseño e implementación de un ambiente inteligente en base a la abstracción del concepto de "servicio" basada en una arquitectura orientada a servicios. El desarrollo de ambientes inteligentes no constituye únicamente un problema de diseño de software y su integración con redes de comunicaciones y con dispositivos hardware heterogéneos, sino que además obliga a definir el dominio semántico de información en el que se estructurarán tanto los dispositivos como los servicios, y donde se podrán establecer de manera natural las posibles interacciones entre los usuarios y el sistema. Para ello será necesario determinar una metodología que permita manejar la información semántica y llevar a cabo un comportamiento proactivo por parte de los servicios del sistema.

La plataforma de servicios proporcionará un marco de trabajo software que garantice propiedades como la interoperabilidad, la interconectividad, la integración entre distintos tipos de componentes hardware-software, la adaptabilidad, la extensibilidad, la escalabilidad y la flexibilidad, entre otras, según las necesidades específicas que se requieran en el diseño de aplicaciones de alto nivel. Además debe incorporar modelos de adaptabilidad y reconfiguración dinámica para permitir que el sistema pueda evolucionar de acuerdo a las necesidades de los usuarios y del entorno, así como potenciar su autonomía. Por otra parte debe garantizar la construcción de sistemas robustos, seguros y flexibles, capaces de adaptarse a las circunstancias con-

cretas del entorno de interacción, y posibilitar la extensión del sistema a través de la integración de nuevos dispositivos y servicios, a ser posible de forma dinámica, y en tiempo de ejecución sin necesidad de detener el sistema.

Los servicios de la plataforma podrán llevar a cabo un comportamiento colaborativo. Para ello la plataforma dará soporte a la composición de servicios teniendo en cuenta las restricciones temporales de los servicios. Esto permitirá diseñar aplicaciones que satisfagan restricciones de tiempo real no estricto.

La plataforma de servicios utilizará toda la información contextual del mundo físico que pueda obtener a partir de los sensores. Esto implica considerar problemas como la conexión entre la información contextual de la que son conscientes los servicios y la información que proporcionan los sensores, la forma de adquirir la información del sensor por parte de los servicios y la manera en que los servicios utilizan esta información como información de contexto para desempeñar un comportamiento proactivo en base a ella.

La presente memoria doctoral ha sido estructurada en tres partes bien diferenciadas: preliminares, contribución y conclusiones. En cada una de estas partes se ha documentado de la forma más completa posible el modo en que todos y cada uno de los objetivos indicados han sido abordados durante el desarrollo de esta tesis.

En la *Parte I* de esta memoria se encuentran los capítulos considerados como *preliminares*. En el capítulo 1 se realiza una introducción a la temática de la tesis y a los principales objetivos abordados durante su desarrollo. En el capítulo 2 se ofrece un detallado estado del arte en aquellas tecnologías involucradas en la investigación. Además se introducen una serie de conceptos y antecedentes relacionados con la investigación cuyo conocimiento previo es importante para la correcta interpretación de los capítulos siguientes. En el capítulo 3 se especifica detenidamente la metodología de investigación que ha conducido el desarrollo del trabajo realizado.

En la *Parte II* se encuentran los capítulos considerados como principal *contribución* de la tesis. El capítulo 4 introduce los conceptos principales y los objetivos de la plataforma de servicios desarrollada. El capítulo 5 presenta los conceptos relativos al modelado, análisis y diseño de la plataforma de servicios colaborativos DOHA. El capítulo 6 analiza los aspectos relativos al desarrollo de la capa semántica SenSE que proporciona mecanismos para la implementación de servicios semánticos sensibles al contexto. Finalmente, el capítulo 7 aborda la metodología de desarrollo que un ingeniero debería llevar a cabo para desarrollar una aplicación basada en servicios para computación ubicua utilizando la plataforma de servicios, así como un análisis de los principales elementos de la plataforma a partir de diversos casos de estudio.

Para finalizar, en la *Parte III* se encuentra el Capítulo 8 que presenta las conclusiones, trabajos futuros y aportaciones derivadas de esta tesis.

Capítulo 2

Estado del Arte y Antecedentes

Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.

Albert Einstein (March 14, 1879 - April 18, 1955

2.1. Introducción

La penetración de las Tecnologías de la Información y las Comunicaciones en la sociedad está produciendo cambios significativos en el modo de concebir el trabajo, el descanso, el ocio o la comunicación. Esto se hace cada vez más constatable por la tendencia creciente hacia escenarios tecnológicos en los que dispositivos digitales inundan el entorno con multitud de servicios y funciones accesibles a través de novedosas y variadas formas de interacción. Los entornos del hogar constituyen un ejemplo claro de este tipo de escenarios tecnológicos, en los que se hace necesaria la convivencia, colaboración e interacción de los múltiples dispositivos heterogéneos que conforman el entorno para poder ofrecer nuevas funcionalidades y servicios que permitan mejorar la calidad de vida de las personas [Bourcier et al., 2011].

A lo largo de las dos últimas décadas se han realizado grandes esfuerzos en el desarrollo de sistemas del hogar desde el punto de vista académico [Fundación OPTI. Consejería de Innovación, 2008] [Das and Cook, 2006] [Ricquebourg et al., 2006] e industrial [Gárate et al., 2005b] [Gárate et al., 2005a], centrándose en buscar la convergencia entre las diferentes tecnologías implicadas a nivel hardware, software y redes de comunicaciones. Esto además requiere una coordinación efectiva de campos como la Electrónica, la Automática y la Informática. A pesar del gran número de propuestas y prototipos funcionales que se han desarrollado, tanto a nivel nacional como a nivel europeo, y que poco a poco se han aglutinado en torno al concepto de "Hogar Digital" [Dourish and Bell, 2011], el sector no ha sido capaz de definir un estándar "universal" que determine qué tipo de tecnologías software, hardware y redes de comunicaciones son las más apropiadas para este tipo de entornos. Varios son los factores que han dificultado o ralentizado la aceptación e implantación de estos sistemas, tanto por parte de instaladores, promotores y usuarios finales, como por la falta de seguridad e incluso en ocasiones de privacidad, y de los altos costes derivados de su instalación, puesta en funcionamiento y mantenimiento.

Sin embargo, hay otro elemento clave que creemos va a ser el impulsor de la nueva generación de sistemas domóticos y que debemos tener en cuenta: "el usuario".

Los sistemas domóticos tienen que trabajar con un mundo complejo de dispositivos heterogéneos que utilizan distintos tipos de tecnologías, por lo que el desarrollo no es fácil, y requiere interoperar entre diferentes plataformas hardware, software y posiblemente entre distintas redes de comunicación [Romero Morales et al., 2010]. Si además tenemos en cuenta que el resultado supone para el usuario contar con sistemas complejos de manejar con múltiples servicios, tantos como dispositivos domóticos individuales interconectados, más añadidos los nuevos servicios que surgen de la composicionalidad de los servicios más simples, el sistema se convierte en algo inabordable tanto desde el punto de vista del desarrollo como de su utilización. Los usuarios necesitan sistemas que promuevan la simplicidad frente a la tecnología. Por tanto, el conjunto de servicios que se ofrecen debe estar adaptado a las características y preferencias de los usuarios, incluso para personas con algún tipo de discapacidad motora, visual o auditiva. Los servicios y dispositivos domóticos que se incorporan al sistema deben tener un comportamiento más "inteligente" y autónomo previendo las acciones de los usuarios e incluso anticipándose a sus necesidades mediante comportamientos proactivos sobre el entorno. Se deben promover interfaces persona-ordenador más naturales buscando una mayor transparencia tanto en el uso de la tecnología como en la interacción que se produce entre los usuarios y el sistema. Y, por último, los sistemas deben adaptarse a los nuevos modos de funcionamiento en cuanto a distribución, ubicuidad y movilidad.

Todas estas nuevas necesidades suponen la base de lo que se viene a denominar "Inteligencia Ambiental". Por tanto, un sistema de inteligencia ambiental define un espacio de interacción ubicuo en el que los dispositivos, los servicios y las aplicaciones con necesidad de uso de servicios puedan convivir, colaborar e interaccionar, adaptándose a las necesidades concretas del usuario de forma natural [Saha and Mukherjee, 2003]. Es decir, son capaces de capturar el conocimiento que necesitan según el contexto y son capaces, por tanto, de tomar decisiones autónomas sobre las acciones a llevar a cabo reduciendo la intervención del usuario [Weiser, 1991].

En este tipo de sistemas la infraestructura software juega un papel esencial. Esta facilita la integración de los dispositivos que conforman el espacio habitable ocultando la heterogeneidad subyacente en cada uno de ellos, en cuanto a su arquitectura hardware o recursos hardware disponibles, además de posibilitar la interoperabilidad entre las distintas redes de comunicaciones utilizadas. En definitiva permite unificar en un marco lógico los servicios y funcionalidades que se pueden proporcionar directamente a través de los dispositivos o mediante la colaboración de dichos servicios. El desarrollo de sistemas para entornos ubicuos de inteligencia ambiental requiere de nuevos modelos de computación y paradigmas específicos y, por tanto, de nuevas infraestructuras software que den soporte a dichas aplicaciones, teniendo en cuenta los recursos limitados en cuanto a memoria o procesamiento de muchos de los dispositivos que interaccionan en dicho espacio.

El cumplimiento de estos requisitos pasa por la utilización de una arquitectura software adecuada. Actualmente los principios del paradigma orientado a servicios, SOA, son los más extendidos y utilizados, y establecen una serie de consideraciones o atributos que deben presentar los servicios desarrollados según este enfoque,

como son acoplamiento débil, encapsulación, abstracción, reusabilidad, composicionalidad, autonomía, optimización y capacidad de descubrimiento [Stojanovic and Dahanayake, 2005]. El uso de arquitecturas de servicios SOA ha sido ampliamente utilizado para el diseño de sistemas de negocio como una tecnología madura que resuelve los problemas de interoperabilidad entre tecnologías de comunicación y posibilita la dinamicidad y evolución del sistema a partir de la colaboración entre servicios [Van Der Aalst, 2013]. Su aplicación a otros dominios como los entornos de computación ubicua en general, y en entornos de inteligencia ambiental está dando resultados muy interesantes [Kiani et al., 2005] [Ejigu et al., 2008] [Kostelnik et al., 2011], aunque plantea algunos problemas. Por ejemplo, la dificultad de utilizarlo sobre dispositivos con pocos recursos computacionales y la necesidad de centralizar elementos importantes de la arquitectura, como los repositorios de servicios.

2.2. Arquitectura Orientada a Servicios

La Arquitectura Orientada a Servicios SOA, acrónimo del inglés Service Oriented Architecture, es un concepto de arquitectura software que define la utilización de servicios para dar soporte a los requerimientos de software del usuario. SOA establece una serie de principios específicos que deben tenerse en cuenta en el diseño y definición de los servicios de un sistema que siga esta orientación, y que influyen de forma directa en el comportamiento del sistema. Existen distintas definiciones del concepto de "servicio" en SOA, aunque todas ellas coinciden que es una entidad sin estado, auto-contenida, que acepta peticiones y devuelve respuestas mediante una interfaz bien definida, y que no depende del estado de otras entidades o procesos. La especificación de una descripción completa de los servicios mediante un contrato de servicio y la obtención de características como acoplamiento débil, encapsulación, abstracción, reusabilidad, composicionalidad, autonomía, optimización y descubrimiento, son elementos fundamentales para la obtención de una plataforma de servicios conforme a los principios SOA [Stojanovic and Dahanayake, 2005].

Las arquitecturas orientadas a servicios son arquitecturas "débilmente acopladas" en las que el enlace a los servicios puede cambiar dinámicamente durante la ejecución. Como características principales del concepto de servicio podríamos decir que (a) es una representación estándar para cualquier recurso computacional o de información que pueda ser utilizado por otras aplicaciones, (b) la provisión del servicio es independiente de la plataforma asociada a la aplicación que lo proporciona, y (c) las aplicaciones se construyen enlazando servicios desde varios proveedores utilizando lenguajes de programación, como por ejemplo Java. Otros conceptos relacionados son los de interfaz de servicio, que define los datos disponibles para/por el servicio y cómo se puede acceder a ellos; proveedor de servicio, que oferta un servicio a las aplicaciones definiendo su interfaz y su funcionalidad; autoadaptabilidad de las aplicaciones, que modifican su comportamiento de acuerdo con su entorno de ejecución, enlazando con diferentes servicios según este cambia; basado en estándares, los servicios-software se han desarrollado guiados por estándares desde su nacimiento

(SOAP, WSDL, UDDI, etc.); e hibridación, algunas aplicaciones se construirán solamente usando servicios y otras los mezclarán con servicios desarrollados localmente [Erl, 2005].

2.2.1. Qué es

La arquitectura SOA proporciona un entorno de servicios independientes, al que nuevos servicios pueden acceder sin necesidad de conocer la implementación o el funcionamiento de la plataforma subyacente. La clave para el desarrollo de un sistema bajo los principios SOA es obtener interfaces de servicio independientes cuyas tareas pueden invocarse de una forma estándar, sin que el servicio tenga conocimiento de las llamadas a la aplicación, y sin que la aplicación necesite conocer cómo este desempeña sus tareas.

El paradigma SOA pretende establecer las bases para garantizar la comunicación entre los distintos componentes de un sistema, de forma que los mensajes tengan una estructura y sintaxis específica y se expresen en un vocabulario compartido por todos. El vocabulario se define en un esquema, cuya extensibilidad es imprescindible para garantizar un amplio abanico de ámbitos de aplicación. Puesto que todas las aplicaciones con una semántica concreta constan de elementos de datos, como los mensajes, un servicio puede ser caracterizado por el conjunto de mensajes que puede interpretar adecuadamente. Estos mensajes están especificados en base a un determinado esquema que se puede considerar como una definición abstracta de un servicio. El consumidor de un servicio solamente conoce su definición abstracta, donde el esquema especifica, para cada servicio, la funcionalidad que tiene asociada dentro del dominio de una aplicación concreta.

2.2.2. Principios SOA

El paradigma SOA establece una serie de principios específicos que deben tenerse en cuenta en el diseño y definición de los servicios de un sistema que siga esta orientación, y que influirán en su comportamiento.

Como podemos observar en la Figura 2.1 , SOA define tres tipos de actores, el Service Consumer o consumidor de servicios, el Service Provider o proveedor de servicios y el Service Directory o directorio de servicios [Stojanovic and Dahanayake, 2005]. El proveedor es responsable de ofrecer una funcionalidad específica a sus consumidores encapsulada en una unidad software autónoma. El consumidor es el cliente que solicita la funcionalidad de un determinado servicio. Por último, el directorio posee un rol de intermediario entre ambos, proveedor y consumidor. El directorio de servicios contiene una descripción actualizada del conjunto de servicios proveedores disponibles; además dispone de mecanismos para la publicación, descubrimiento y eliminación de servicios si es necesario. Los servicios proveedores publican su descripción en el directorio de servicios con el objetivo de que los consumidores puedan localizarlos y utilizarlos. Una vez que la descripción de los proveedores está disponible, los clientes pueden acceder a ellos y solicitar su funcio-

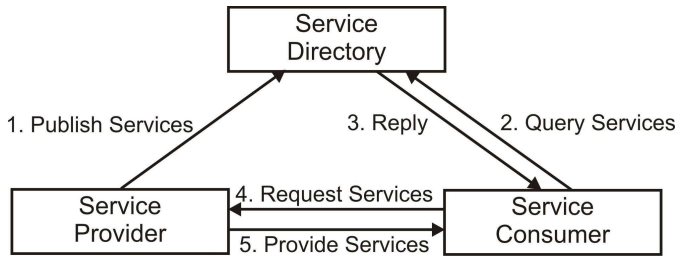


Figura 2.1: Entidades principales de SOA

nalidad.

El paradigma SOA impone el uso de una serie de restricciones en el diseño y organización de los servicios. Como consecuencia directa de estas restricciones es necesario establecer una separación entre el comportamiento interno y externo en el desarrollo de los servicios. De acuerdo con [Erl, 2005] y [Stojanovic and Dahanayake, 2005], las principales propiedades que deben satisfacer los servicios son las siguientes.

Contrato de servicio. Un contrato de servicio puede ser una descripción, técnica o no, de las características del servicio y su comportamiento. Es una buena práctica en sistemas distribuidos que cuando dos elementos del sistema van a comunicarse, utilicen un estándar, cuyo papel jugaría en este caso el "contrato de servicio". Así, los servicios y usuarios de un mismo conjunto de servicios deben seguir el mismo "contrato de servicio".

Bajo acoplamiento. El nivel de acoplamiento entre servicios puede entenderse también como el nivel de dependencia, que en programación distribuida es deseable que sea el mínimo posible. Por ello, los servicios deben establecer relaciones que minimicen las dependencias con otros servicios o usuarios, de los que solo se requiere conocer su existencia. El cliente, ya sea otro servicio, usuario final, o un programa externo, no tiene por qué ser consciente de cómo un servicio lleva a cabo sus responsabilidades. Las estructuras internas de datos, las invocaciones a otros servicios, la gestión de transacciones y los requisitos de almacenamiento deberían estar ocultos al cliente. Para garantizar que se cumpla, este principio debe satisfacerse tanto en el diseño del servicio como en sus relaciones con otros servicios o usuarios, lo cual afectará también al diseño del contrato de los servicios.

Encapsulación. Durante el diseño de los servicios se debe tener en cuenta este principio y establecer de forma clara una separación del comportamiento interno y externo. El servicio solo debe definir como observable en el sistema su comportamiento externo, no la forma en que se realiza la conducta, y reflejarlo en el "contrato de servicio".

Abstracción. Por encima de lo que se describe en su contrato, los servicios deben ocultar su funcionamiento lógico al mundo exterior, es decir, deben ocultar la información que no es absolutamente necesaria para utilizar de manera eficaz el servicio por parte de otros servicios o usuarios. Para garantizar este principio es aconsejable seguir el enfoque de "caja negra" en el diseño de los servicios. La

información que es deseable abstraer al aplicar este principio se denomina metainformación, y abarca los siguientes aspectos del servicio: tecnología, funcionalidad, lógica de programación y calidad del servicio. Los niveles de abstracción establecidos deben tenerse en cuenta a la hora de especificar el contenido del "contrato de servicio".

Reusabilidad. Uno de los principales objetivos de la utilización de un modelo de programación SOA es la reutilización, pues se divide la lógica de una aplicación concreta en servicios con el fin de garantizar este principio.

Composicionalidad. Este principio pretende que dada una colección de servicios, puedan ser coordinados y ensamblados para formar nuevos servicios compuestos.

Autonomía. La autonomía de un servicio representa el nivel de independencia con el que este puede llevar a cabo su lógica de actuación, para lo cual es necesario tener en cuenta requerimientos asociados a la infraestructura del servicio en el diseño del mismo. Este principio apoya los principios de "reusabilidad" y "composicionalidad".

Optimización. Se debe garantizar la opción de incluir modificaciones y mejoras en los servicios, a fin de mantener los servicios en el mejor estado posible.

Descubrimiento. Los servicios deben diseñarse para que sean lo más descriptivos posible hacia el exterior, de forma que puedan ser encontrados y evaluados a través de los mecanismos de descubrimiento disponibles. El cliente no tiene por qué ser consciente de dónde reside o se ejecuta el servicio. El objetivo fundamental de este principio es el de mejorar la calidad de las comunicaciones entre servicios y usuarios.

2.2.3. Beneficios de SOA

Los beneficios que puede obtener un sistema de computación ubicua que adopte un enfoque SOA son:

- Optimización del tiempo necesario para la realización de cambios en los procesos.
- Facilidad para evolucionar los modelos, dinamicidad.
- Facilidad para abordar modelos de comportamiento colaborativos.
- Poder para reemplazar aplicaciones SOA sin contrapartida en el resto del sistema.
- Facilidad para la integración de tecnologías heterogéneas.

En un ambiente SOA, los nodos de la red comunican su disponibilidad y sus recursos a otros participantes en la red como servicios independientes a los que tienen acceso de un modo estandarizado. La mayoría de las definiciones de SOA

identifican la utilización de Servicios Web, empleando SOAP y WSDL, en su implementación. No obstante, se puede implementar SOA utilizando cualquier tecnología basada en servicios, siempre aconsejándose el uso de estándares para garantizar la extensibilidad de los sistemas.

Al contrario de las arquitecturas orientadas a objetos, las arquitecturas SOA están formadas por servicios de aplicación débilmente acoplados y altamente interoperables. Para comunicarse entre sí, estos servicios se basan en una definición formal independiente de la plataforma subyacente y del lenguaje de programación, por ejemplo WSDL. La definición de la interfaz encapsula u oculta las particularidades de una implementación, lo que la hace independiente del fabricante, del lenguaje de programación o de la tecnología de desarrollo, garantizando la heterogeneidad y la interoperabilidad. Con esta arquitectura, se pretende que los componentes software desarrollados sean muy reusables, ya que la interfaz se define siguiendo un estándar.

2.3. Frameworks para el desarrollo de Sistemas Ubicuos

A continuación se presenta una breve descripción de las tecnologías basadas en Java más utilizadas para el desarrollo de sistemas ubicuos, como son Jini, JavaSpaces, MOM, Web Services y JXTA. La tabla 2.1 resume las características de cada uno de los framework presentados, características que serán discutidas a continuación.

La tecnología *Jini* favorece la construcción de sistemas distribuidos basados en el concepto de "federación de servicios" con una arquitectura cliente/servidor [Apache, 2014b]. Esta arquitectura proporciona un framework para registrar y descubrir dinámicamente los servicios. El *Lookup Service* juega el rol de directorio de servicios según SOA. En este caso los proveedores de servicio se comunican con los consumidores a través de unos objetos Java especiales, denominados objetos *proxy*, que están alojados en el *Lookup Service*. Estos objetos implementan una interfaz Java que representa su contrato de servicio. El servicio de descubrimiento es flexible, pero su carácter centralizado lo hace vulnerable ante la posible desconexión del nodo central que alberga al *Lookup Service*, lo que provocaría que el funcionamiento del sistema en su conjunto no fuese el esperado. Algunos trabajos de investigación que utilizan Jini como framework de soporte para sistemas ubicuos son [Kiani et al., 2005] [Lee et al., 2006a] [Xu et al., 2007].

La utilización de *JavaSpaces* en el desarrollo de aplicaciones distribuidas proporciona un espacio virtual entre proveedores, clientes y recursos de red u objetos. Dicho espacio virtual se construye en base a un espacio de memoria compartida con tuplas. Un objetivo de los servicios en JavaSpaces es notificar a las entidades interesadas en los objetos cuándo estos están disponibles en el espacio compartido. Este espacio actúa globalmente a modo de servicio de directorio, en el cual los servicios pueden llevar a cabo tareas de publicación y descubrimiento, ya que todos los servicios lo comparten. Los trabajos de [Rakotonirainy et al., 2001] y [Lee et al., 2006b] son ejemplos de utilización de este framework.

Tabla 2.1: Comparación de distintas tecnologías Java aplicables a sistemas ubicuos

	Jini	JavaSpaces	MOM (JMS)	Web Services (DPWS)	JXTA
Data representation	Java Marshalled Object	Collections of information	Messages	XML	XML
Transport	Based RMI protocols	Serialization	Based RMI protocols	SOAP over HTTP	Usually TCP/IP or http
Services description	Java interface	JavaSpace interface contract	Administered objects	WSDL	Advertisement
Services location	Jini Lookup Service	Entries	Publish/ Subscribe or point to point	UDDI	Discovery Service
Language bindings	Java	Java	Java	Java, .NET, Perl	Java, C
Remote reference	Proxy objects	Proxy objects	Identification	URL	Peer Rendezvous or Relay
Synchronicity	Sync/Async	-	Async	Sync	Sync/Async
Arquitecture type	Client-Server	Client-Server	Client-Server	Client-Server	Peer-to-Peer

El middleware de colas de mensajes distribuidas *Message Oriented Middleware* (MOM) establece una comunicación basada en mensajes asíncronos. La técnica de paso de mensajes no bloqueantes puede ser utilizada en un sistema según los principios SOA para facilitar el bajo acoplamiento entre aplicaciones o servicios, así como para simplificar la coordinación entre los mismos a través de la red. Este es el caso del trabajo de investigación presentado por Drosos [Drosos et al., 2005].

Probablemente, la alternativa más difundida en el desarrollo de sistemas SOA, tanto en el ámbito de la computación ubicua como fuera de él, sea el uso de Web Services [Curbera et al., 2002]. Por lo general, los servicios web proporcionan un framework con el que desarrollar de forma natural plataformas basadas en SOA definiendo los mecanismos necesarios para describir, localizar y comunicar a las aplicaciones entre sí. Estas aplicaciones se convierten así en componentes que forman un servicio web en la red de servicios. Los servicios web poseen una descripción realizada en WSDL que hace las veces del contrato de servicio establecido por los principios de SOA. Esta descripción se publica asociada a un identificador unívoco UDDI para que otros servicios consumidores puedan descubrirla y hacer uso de ella para interaccionar con el servicio y requerir su funcionalidad. El protocolo de

comunicación utilizado para llevar a cabo la comunicación entre servicios es SOAP. Son muy numerosos los trabajos de investigación que utilizan este tipo de framework en el desarrollo de sistemas para computación ubicua [Amoretti et al., 2008].

Por último, cabe mencionar los frameworks basados en P2P que también permiten llevar a cabo las tareas de publicación, descubrimiento, colaboración e interacción de servicios de acuerdo a los principios SOA. Una ventaja considerable en este tipo de arquitectura es que todos los nodos pueden desempeñar cualquier tipo de rol, ya sea cliente o servidor. La comunicación basada en peers establece una comunicación entre iguales basada en nodos independientes, lo cual a su vez favorece la construcción de aplicaciones distribuidas. Una infraestructura comúnmente utilizada para el desarrollo de sistemas P2P es JXTA [Oracle, 2014]. Además de las ventajas ya comentadas sobre el uso de una arquitectura P2P, JXTA requiere menos tareas de gestión que otras tecnologías, como por ejemplo Jini, lo cual la convierte en un mejor candidato en cuanto a dispositivos con recursos limitados se refiere. Además, JXTA proporciona un esquema de indexación de publicación y consultas basado en super-nodos, *Rendezvous* y *Relay* peers [Buford and Yu, 2010], lo cual elimina los problemas derivados del uso de enfoques más centralizados. De hecho, el framework JXTA también ha sido utilizado en numerosos trabajos de investigación sobre computación ubicua basados en redes de sensores, agentes o servicios [Bagci et al., 2005] [Krco et al., 2005] [Ejigu et al., 2008].

2.4. Servicios Colaborativos en Entornos Ubicuos

Una cuestión concreta con una relevancia cada vez mayor en el ámbito de la computación ubicua y la inteligencia ambiental es la evolución de los dispositivos desde un comportamiento automático a un comportamiento proactivo [Weiser, 1991]. La naturaleza automática de estos dispositivos implica que pueden ser utilizados por los usuarios como artefactos con los que obtener información, como el valor de temperatura de una habitación, o para obtener una funcionalidad específica, como controlar la temperatura de la habitación. Sin embargo, el progreso de la inteligencia ambiental ha transformado a los dispositivos en elementos sensibles al contexto capaces de colaborar entre ellos sin la intervención del usuario. Son objetos autónomos, distribuidos, interconectados y capaces de llevar a cabo acciones colaborativas y de formar grupos para ofrecer nuevas características a los usuarios [Banavar et al., 2000]. Además, los dispositivos ubicuos deben ser imperceptibles para los usuarios, de pequeño tamaño y optimizando el uso de los recursos limitados que poseen. Por lo tanto, los métodos de composición de servicios existentes ligados a procesos de negocio de alto nivel no son viables en computación ubicua, donde las limitaciones de los espacios ubicuos y sus dispositivos son un hándicap [Kakousis et al., 2010].

La funcionalidad ofrecida por los dispositivos ubicuos puede representarse como componentes software débilmente acoplados denominados servicios. La idea principal del paradigma SOA es ensamblar diferentes componentes a través de una red de servicios débilmente acoplados para crear procesos flexibles y dinámicos. Esto

refuerza el papel del servicio como abstracción software para facilitar el desarrollo de aplicaciones ubicuas distribuidas. Sin embargo, la aplicación de los principios de SOA sin tener en cuenta determinados requisitos no funcionales –en términos de atributos de calidad (rendimiento o ancho de banda) y de configuración (tecnología utilizada o plataformas)– conduce a la obtención de sistemas inviables que no ofrecen ninguna garantía de solidez, fiabilidad, disponibilidad y escalabilidad, todos ellos atributos requeridos en sistemas ubicuos. Asimismo, lejos de seguir un modelo de interacción tradicional entre usuarios y dispositivos, la computación ubicua trata de desarrollar comportamientos más complejos por medio de la colaboración entre servicios.

La composición de servicios aborda cómo llevar a cabo la composición de los servicios existentes en un sistema a fin de lograr una funcionalidad más compleja que, normalmente, no puede ser llevada a cabo por un único servicio de forma aislada [Pessoa et al., 2008]. Por medio de la composición de servicios un determinado servicio puede complementar su funcionalidad colaborando con otros servicios. De este modo, a través de la composición, un conjunto de servicios puede crear espacios más inteligentes que ofrecen una funcionalidad nueva y más compleja de forma transparente para los usuarios [Bronsted et al., 2007]. Los sistemas ubicuos son sistemas altamente dinámicos que requieren procedimientos y mecanismos capaces de integrar y combinar los servicios disponibles que se encuentran en dispositivos distribuidos. En consecuencia, la composición de servicios juega un papel muy importante en este ámbito de investigación. Trabajar en el ámbito de la composición de servicios implica abordar varias cuestiones importantes, como cuándo se lleva a cabo la composición, quién/qué es responsable de la misma, dónde se lleva a cabo, y cómo se gestiona.

Dependiendo de *cuándo* se establece la decisión de colaborar entre dos o más servicios, se pueden distinguir dos enfoques posibles: composición de servicios estática o dinámica [Dustdar and Schreiner, 2005b]. Se denomina composición de servicios estática cuando la colaboración entre servicios se planifica en la etapa de diseño, previa al despliegue y ejecución de los servicios. Por el contrario, la composición de servicios se considera dinámica cuando no se especifica *a priori* qué servicios estarán implicados en un procedimiento de colaboración. Es decir, los servicios que colaborarán en el proceso de composición se seleccionan en tiempo de ejecución, una vez que todos ellos ya se están ejecutando en el sistema. Se puede intuir que la composición dinámica resulta más flexible que la composición estática, ya que el procedimiento de colaboración puede ser modificado, ampliado o adaptado en tiempo de ejecución. Para ello es necesario hacer uso de una entidad adicional de más alto nivel capaz de identificar la funcionalidad de cada servicio individual y llevar a cabo la composición final, por ejemplo, un razonador. Sin embargo, la relevancia de la composición estática como predecesora de un proceso de composición dinámica debe ser tenida muy en cuenta, ya que puede ser la base de un buen modelo de composición dinámica. Si la composición estática falla, la composición dinámica no funcionará en absoluto.

La colaboración entre los servicios puede ser manejada por un solo servicio, por

el conjunto de servicios que intervienen en la composición, o incluso por un elemento adicional que no forma parte de la colaboración (*quién/qué*). Los modelos de composición de servicios más comunes se basan en la orquestación o en la coreografía de servicios, especialmente en contextos de tecnologías cliente-servidor como los Web Services. En la orquestación de servicios el control del flujo de ejecución es siempre responsabilidad de una de las partes implicadas en la colaboración –se trata de un único servicio que controla la ejecución del flujo de trabajo. Por el contrario, en la coreografía de servicios cualquier entidad involucrada en la colaboración puede tomar parte en la interacción sin necesidad de considerar un elemento de control –todos los servicios tienen la misma responsabilidad en el flujo de ejecución de la composición [Peltz, 2003]. Sin embargo, son muchos los aspectos críticos de los sistemas de computación ubicua, como el uso de dispositivos empotrados, caracterizados por su gran movilidad y recursos reducidos, tanto a nivel de procesamiento como de memoria, que son un gran hándicap.

La composición de servicios puede llevarse a cabo de forma centralizada o descentralizada (*dónde*). En los modelos de composición de servicios centralizados existe un nodo o dispositivo en el cual se realizan todas las composiciones, siempre en el mismo contexto. Para lograr esto, el dispositivo debe tener información sobre todos los servicios que potencialmente se pueden combinar. Un ejemplo son los sistemas basados en OSGi [Dohndorf et al., 2010]. En los modelos de composición de servicios descentralizados, los servicios están distribuidos en diferentes nodos o dispositivos y la composición se lleva a cabo por cada servicio en su propio nodo/dispositivo. Sin embargo, también existen enfoques que consideran un modelo híbrido entre ambos. Para ello consideran un nodo especial que interviene en la sincronización de los servicios entre diferentes redes. En este caso la composición se denomina semicentralizada [Gharzouli and Boufaida, 2010].

Existen diversos métodos formales y herramientas para diseñar *cómo* se realiza la composición [Karunamurthy et al., 2012] [Ter Beek et al., 2007]. Algunas de estas herramientas también permiten generar el código de los servicios en diferentes lenguajes de implementación. Estas herramientas se basan, por lo general, en lenguajes de modelado formal o semi-formal, tales como UML, redes de Petri o CSP, o bien utilizan una representación semántica de los servicios, tales como ontologías o razonadores.

La corrección de la composición de servicios puede evaluarse de acuerdo a diferentes aspectos relacionados con el comportamiento colaborativo de los servicios. Dado que la composición de servicios da lugar a sistemas complejos con gran número de servicios ejecutándose concurrentemente, un aspecto fundamental a verificar es si la interacción entre servicios a través del protocolo de comunicación establecido es correcta; dicha corrección puede evaluarse localmente entre dos servicios y globalmente si se tienen en cuenta todos los servicios que forman parte de la composición. Por lo general, los métodos formales permiten verificar la satisfacción de gran cantidad de las propiedades deseables en este tipo de sistemas, tales como la seguridad, vivacidad y confiabilidad [Ter Beek et al., 2007].

El comportamiento colaborativo en sistemas ubicuos ha sido estudiado desde

la perspectiva de varias estrategias de investigación. Existen middleware específicamente diseñados en base a las necesidades de computación ubicua [Barolli and Xhafa, 2011]; encontramos también ontologías y modelos semánticos que buscan enriquecer semánticamente la información utilizada por los sistemas ubicuos [Vazquez et al., 2007]; e incluso estudios que abordan la composición como si de servicios de negocio se tratase y que utilizan paradigmas tales como BPEL, OWL-S o PSML [Li et al., 2012]. En este sentido, a pesar de que podríamos utilizar modelos de composición específicamente creados para ser aplicados a servicios de negocio, creemos firmemente que es beneficioso para el desarrollo de sistemas ubicuos contar con métodos ligeros, que optimicen el uso de los recursos disponibles en los dispositivos empotrados y que den soporte a la dinamicidad de los espacios ubicuos.

Una estrategia comúnmente utilizada para el desarrollo de sistemas ubicuos en el ámbito de la domótica ha sido utilizar una arquitectura centralizada. En el trabajo de Nakamura [Nakamura et al., 2008] se propone un modelo de composición de servicios basado en gateway y utilizando perfiles de comportamiento creados a partir de un grafo de colaboración y almacenados en un archivo de configuración centralizado.

Los modelos basados en la maximización de la calidad de servicio, conocida como QoS, acrónimo de Quality of Service, también se utilizan en composición de servicios ubicuos. Los parámetros de calidad de servicio constituyen información muy útil a tener en cuenta en el diseño de métodos de composición. El trabajo de Chang et al. propone una metodología de composición de servicios orientada a la calidad por medio de una herramienta llamada *Event-driven Web Services Composer* [Chang and Lee, 2010]; Lv et al. también define un modelo de composición de servicios basado en la selección de servicios a partir de sus atributos de calidad [Lv et al., 2009]; y Éstevez-Ayres et al. utilizan los parámetros de calidad de servicio para diseñar un modelo de composición de servicios con requisitos de tiempo real y alta tolerancia a fallos [Estévez-Ayres et al., 2009].

El uso de ontologías y propiedades semánticas está también presente en numerosos trabajos de investigación que tratan de resolver los problemas ligados al comportamiento colaborativo de los servicios ubicuos. Las ontologías se utilizan para representar la información semántica asociada a los escenarios ubicuos y permiten asociar metadatos a la definición de los servicios [Kostelnik et al., 2011]. Modelos relacionados con la composición dinámica [Niu et al., 2011], la sensibilidad al contexto [Bernardos et al., 2009] y el Internet de las cosas [Kiritsis, 2011], son algunas de sus aplicaciones.

En el caso del desarrollo middleware muchas soluciones basadas en servicios utilizan Web Services [Corredor et al., 2012] [Villanueva et al., 2009]. Hay otras aportaciones que optan por otro tipo de arquitecturas, como por ejemplo las Peer-to-Peer (P2P), para el desarrollo de sistemas basados en servicios para computación ubicua [Gerke et al., 2007]. Las arquitecturas distribuidas y descentralizadas permiten hacer un mejor uso de los recursos heterogéneos, facilitan el acceso a los datos, mejoran la escalabilidad y reducen los cuellos de botella [Zhou et al., 2010].

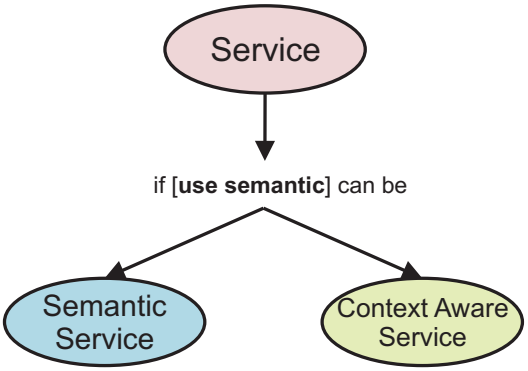


Figura 2.2: Inclusión de semántica a los servicios

2.5. Semántica en Entornos Ubicuos

La información semántica debe ser interpretable y procesable por los dispositivos sin necesidad de la intervención humana. Así, es posible que los servicios puedan buscar y seleccionar de forma dinámica y automática los dispositivos y/o servicios con los que pueden colaborar y llevar a cabo acciones sobre el entorno sin la intervención de los usuarios, entre otros. Para ello, es importante utilizar modelos de representación del conocimiento que nos permitan obtener información del dominio, en nuestro caso dispositivos y servicios del espacio ubicuo, utilizando lenguajes que permitan expresar semánticamente tanto las entidades del dominio como las relaciones existentes entre ellas. Esto permitirá que se pueda consultar, interpretar y procesar dicha información de forma automática. También sería posible obtener nueva información que se almacenaría en bases de conocimiento que pueden tratarse de forma centralizada o distribuirse parcialmente entre los dispositivos del espacio ubicuo [Rodríguez and Holgado, 2009].

Como puede observarse en la Figura 2.2 la inclusión de semántica en los servicios de una plataforma basada en SOA hace que estos se transformen en servicios de mayor valor añadido. Cuando la semántica complementa la información de los propios servicios, por ejemplo, indica las propiedades de calidad de servicio que estos satisfacen, hablaríamos de Servicios Semánticos. En cambio, cuando la semántica alude a la información externa al servicio pero relacionada con su contexto de ejecución, estaríamos ante Servicios Sensibles al Contexto.

2.5.1. Web Semántica

La semántica como elemento conductor de la computación ubicua actual guarda una estrecha relación con el nacimiento de la Web 2.0 y el desarrollo de la Web Semántica. Son muchos los trabajos de investigación que en la actualidad giran en torno al desarrollo de lenguajes y plataformas sobre el concepto de semántica, donde un papel predominante lo ocupan los lenguajes utilizados para la representación de los conceptos, la mayoría de ellos construidos sobre la base de XML u otros lenguajes

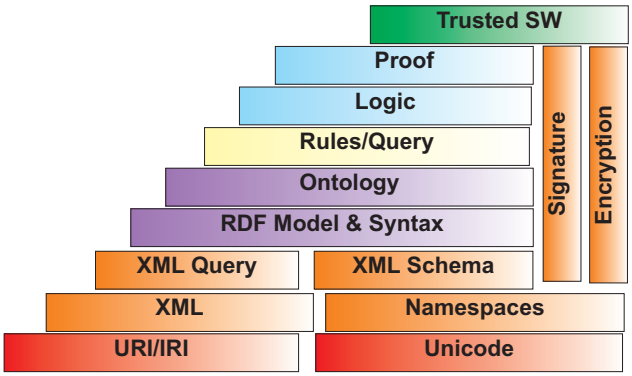


Figura 2.3: Estructura de la Web Semántica para el W3C

derivados de él. XML, acrónimo de Extensible Markup Language, es un metalenguaje de etiquetas extensible desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación del SGML, Standard Generalized Markup Language, y permite definir la gramática de lenguajes específicos. Por lo tanto, XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Se pretende que el desarrollo de la Web Semántica sea incremental, por lo que se está produciendo en forma de capas, de modo que el desarrollo de nuevas capas se realiza sobre las ya existentes. En la actualidad, el W3C maneja una estructura semejante a la que se muestra en la Figura 2.3 [W3C, 2014c].

Aunque analizaremos con mayor nivel de detalle algunos de los elementos presentes en esta estructura, vamos a comentar brevemente los más representativos en el ámbito que nos ocupa, así como la relación entre ellos. Por un lado, XML aporta la sintaxis necesaria para construir documentos estructurados, pero sin restricciones con respecto a su significado, que puede utilizar XML Schema como lenguaje para definir la estructura de los documentos. A más alto nivel, RDF es un modelo de datos que permite definir los recursos y las relaciones que se puedan establecer entre ellos, que aporta una semántica básica que puede representarse mediante XML. RDF Schema es un vocabulario para describir las propiedades y las clases de los recursos RDF, con una semántica para establecer jerarquías de generalización entre dichas propiedades y clases. En el nivel de ontología estaría OWL, que añade más vocabulario para describir propiedades y clases. Aporta mejoras de expresividad como definir relaciones entre clases, cardinalidad, igualdad, tipologías de propiedades más complejas, caracterización de propiedades o clases enumeradas.

Las principales ventajas de la Web Semántica son el ahorro de tiempo en el procesado de los datos, la obtención de resultados de búsqueda más adecuados y la mejora de la comunicación entre servicios. Las ontologías juegan un papel protagonista en este sentido. Los lenguajes de ontologías, como RDF y OWL, surgen como un instrumento para dotar de capacidades semánticas a los sistemas computacionales. Desde el punto de vista de SOA, la potenciación semántica de los servicios implica la inclusión de metadatos en la definición del servicio. Esto se puede hacer dentro

de la definición del servicio o dentro de la infraestructura de descubrimiento del servicio. En un entorno de servicios web, se pueden usar las extensiones de WSDL, o se puede ampliar la UDDI para ofrecer detalles adicionales a nivel semántico.

Las tecnologías basadas en la Web Semántica, como OWL-S, WSMO y SAWSDL, llevan ya un tiempo siendo utilizadas por los desarrolladores para dotar de mayor riqueza semántica a la especificación de servicios [Martin et al., 2005]. A continuación se comentarán con más detalle algunos de los elementos más importantes en el desarrollo de aplicaciones que consideren semántica para su desarrollo y ejecución, como son RDF, OWL, OWL-S y Jena.

2.5.1.1. RDF

Se podría considerar a RDF, Resource Description Framework , como el lenguaje base de la web semántica [W3C, 2014b], siendo considerado como modelo estándar para el intercambio de datos en la Web. Tiene características que facilitan la fusión de datos, incluso si los esquemas subyacentes difieren, y soporta la evolución de esquemas en el tiempo sin necesidad de efectuar cambios en los consumidores.

El concepto de tripleta o terna RDF permite dotar de significado a los datos representados, considerándose esta la principal diferencia con respecto a XML. Utilizar RDF como mecanismo de representación de la información implica seguir una estructura basada en la terna *sujeto − predicado − objeto* [W3C, 2014b]. Dentro de esta terna, el objeto encaja con el concepto de "valor" del elemento del dominio, mientras que sujeto representaría al elemento en sí mismo, siendo el predicado la relación entre ambos.

2.5.1.2. OWL

OWL es el acrónimo del inglés Ontology Web Language, un lenguaje de marcado para publicar y compartir datos usando ontologías en la Web. OWL tiene el objetivo de facilitar un esquema de marcas XML sobre RDF. OWL extiende a RDF y permite expresar relaciones complejas entre diferentes clases con mayor precisión en las restricciones entre clases y propiedades específicas. Los creadores de OWL se inspiraron en DAML+OIL como antecedente. Este y otros lenguajes utilizados anteriormente para desarrollar herramientas y ontologías específicas no fueron definidos para ser compatibles con la arquitectura de la WWW, World Wide Web , y la Web Semántica.

Actualmente, OWL tiene tres variantes: OWL Lite, OWL DL y OWL Full. Estas variantes incorporan diferentes funcionalidades con un grado de completitud ascendente, es decir, OWL Lite es más sencillo que OWL DL, y OWL DL es a su vez más sencillo que OWL Full.

La elección del acrónimo OWL ha sido algo controvertida desde su origen. El acrónimo correcto para Web Ontology Language debería ser WOL en lugar de OWL, hecho que se utiliza en curiosos foros sobre el lenguaje para justificar que el orden ha sido elegido en honor del personaje *Owl* de *Winnie the Pooh*. Dicho personaje escribe

su nombre WOL en lugar de OWL. Realmente, OWL fue propuesto originalmente por Tim Finin como acrónimo fácilmente pronunciable en inglés, y que a su vez facilitase el marketing mediante la generación de logos comerciales y su relación con el prestigioso proyecto de representación del conocimiento *One World Language*, promovido por William A. Martin desde el MIT en los años setenta [Finin][Hitzler et al., 2009].

2.5.1.3. OWL-S

OWL-S, Web Ontology Language for Services, anteriormente DAML-S, es una ontología de servicios que permite descubrir, invocar, componer y monitorizar los recursos ofrecidos por determinados servicios y que poseen propiedades particulares, en función de ciertas bases de conocimiento predeterminadas [W3C, 2014a]. En esencia, OWL-S es una ontología que contiene los elementos fundamentales que caracterizan y que permiten describir las capacidades de los servicios de forma semántica.

Una característica a destacar en OWL-S es la distinción que hace entre servicios "atómicos" y servicios "compuestos". Los servicios atómicos son aquellos programas aislados, accesibles por la Web o a través de un dispositivo como un sensor o un actuador, que son invocados mediante un mensaje de solicitud, y responden con un único mensaje cuando han llevado a cabo su tarea. Por el contrario, los servicios compuestos o complejos son aquellos formados por múltiples servicios básicos que interaccionan con el solicitante hasta la finalización de la tarea. El objetivo último de OWL-S es hacer posible el descubrimiento, la invocación, y la composición automática de servicios.

En esencia, OWL-S es una ontología que contiene los elementos fundamentales que caracterizan un servicio y que permite describir las capacidades que sustentan su comportamiento. Esta ontología distingue entre tres tipos de conceptos fundamentales asociados a los servicios, como puede observarse en la Figura 2.4: (a) qué es lo que hace el servicio o *ServiceProfile*, (b) cómo se usa el servicio o *ServiceModel*, y (c) cómo interaccionar con el servicio o *ServiceGrounding*. Para dotar a una plataforma de servicios de características semánticas se pueden utilizar las ontologías establecidas por OWL-S durante el diseño de los servicios.

Cada servicio posee un conjunto de *ServiceProfiles*, de forma que pueda proporcionar diferentes funcionalidades a los clientes. El descubrimiento automático de servicios se hace mediante los perfiles definidos en el sistema. Cada servicio requiere de un único *ServiceModel*, que indica su funcionamiento, y se comunica con el cliente mediante un único *ServiceGrounding*.

El perfil de cada servicio o *ServiceProfile* consta de la información del servicio como proveedor, como es el nombre del servicio, descripción textual breve e información de contacto, las características o propiedades del servicio y su funcionalidad. El servicio se modela como un proceso de interacción entre el cliente y el servicio mediante el *ServiceModel*, de forma atómica, compuesta o simple. El conocimiento base o *ServiceGrounding* especifica cómo acceder e invocar al servicio. Este último realiza un descripción de las relaciones existentes entre la descripción del proceso realizado

por el servicio y la descripción real de los elementos necesarios para interaccionar con él, como por ejemplo el protocolo de comunicación.

Existen algunas herramientas interesantes para facilitar la utilización de OWL-S, como son OWL-S Editor, desarrollado por la Universidad de Malta [UniversityOfMalta, 2014]; OWL-S Java API, desarrollada por Mindswap para ejecutar servicios OWL-S [Sourceforge, 2014]; y OWL-S Editor, basado en Protegé [SemWebCentral, 2014].

2.5.1.4. Jena

Jena es una API de código abierto desarrollada por un grupo de especialistas en Web Semántica de los laboratorios de HP. Jena, como framework, provee de un entorno de desarrollo para RDF, RDFS, OWL y SPARQL, además de disponer de inferencia basada en reglas [Apache, 2014a]. La principal motivación de la creación de Jena fue obtener capacidad para manipular metadatos desde una aplicación Java.

La API RDF de Jena permite crear y manipular modelos RDF desde una aplicación Java. Para ello proporciona clases para representar declaraciones, recursos, propiedades, literales y modelos como conjuntos de declaraciones. La API de OWL de Jena utiliza un lenguaje neutral, es decir, los nombres de las clases no hacen mención al lenguaje subyacente que están representando, como OWL o RDF. Por ejemplo, la clase *OntClass* puede hacer referencia a una Clase OWL o de RDF Schema. Para representar las diferencias entre los distintos lenguajes, cada uno posee un perfil o *Profile* que lista todos los constructores permitidos, los nombres de las clases y las propiedades. Cada uno de los lenguajes de ontología tiene un perfil, en el que se permite la construcción de clases y propiedades más concretas. Así, en el perfil de DAML, el URI de una propiedad a nivel de objeto es *daml:ObjectProperty*, en el perfil de *OWL* es *owl:ObjectProperty* y en el perfil de *RDF* es nulo, ya que RDF no permite definir propiedades del tipo *objectProperty*.

El perfil está unido al modelo ontológico u *OntModel* que es una versión extendida de la clase *Model* de Jena. La clase *OntModel* extiende el modelo básico de RDF añadiendo a los objetos aquellas propiedades que se esperan de una ontolo-

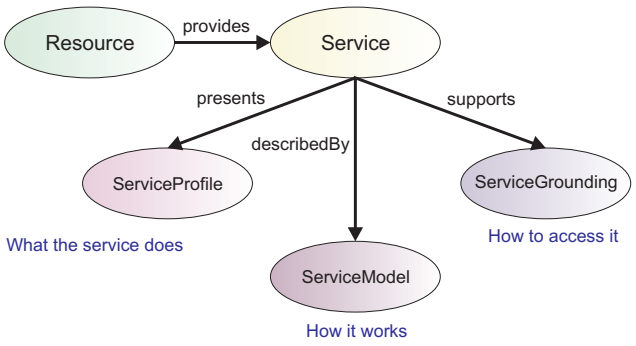


Figura 2.4: Estructura de OWL-S

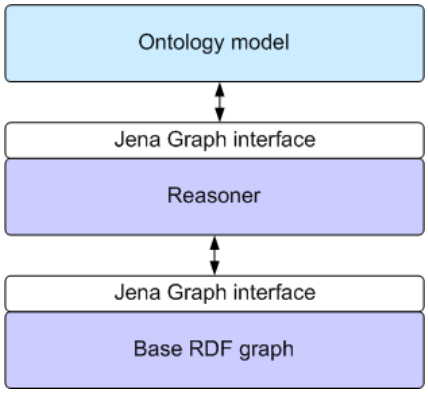


Figura 2.5: Relación entre el modelo ontológico en Jena y su representación en RDF [Apache, 2014a]

gía, como clases, propiedades e individuos. Cuando se trabaja con una ontología en Jena, toda la información de estado sigue siendo codificada como tripletas RDF almacenadas según este modelo. Es decir, la API de ontología de Jena no cambia la representación RDF de las ontologías, lo que sí hace es añadir una serie de clases y métodos que hacen que sea más fácil la manipulación de ontologías. Por ejemplo, supongamos que tenemos un objeto *OntClass* representando una determinada clase de nuestra ontología. Esta clase tiene un método para determinar sus superclases, *listSuperClasses()* que se corresponde al valor de la propiedad *rdf:SubClassOf* de RDF. Cuando llamemos a dicho método, Jena obtendrá la información desde el modelo subyacente, es decir, desde las tripletas RDF.

En la Figura 2.5 podemos ver la relación existente entre el modelo ontológico en Jena y su representación en base a RDF.

Por otro lado el subsistema de inferencia de Jena, *Jena Inference Support*, está diseñado para permitir trabajar con distintos tipos de razonadores y motores de inferencia. Jena ofrece mecanismos para añadir nuevos razonadores e incluye un conjunto básico de estos formado por *OWL Reasoner*, *DAML Reasoner* y *RDF Rule Reasoner*. Estos motores se utilizan para derivar nuevas sentencias RDF a partir de las ya definidas, junto con cualquier otra información de la ontología, los axiomas y las reglas asociadas con el razonador. Al utilizar este mecanismo se apoya el uso de lenguajes como RDF y OWL, que permiten deducir nuevos hechos a partir de instancias y descripciones de clases. La estructura general de inferencia en Jena puede observarse en la Figura 2.6.

Normalmente, el acceso de las aplicaciones al motor de inferencia de Jena se lleva a cabo mediante el *ModelFactory* que asocia un conjunto de datos con un razonador concreto para crear un nuevo modelo. Las consultas sobre el modelo de conocimiento creado no solo devolverán los datos iniciales de las ontologías, sino que también se podrán obtener declaraciones adicionales que se puedan haber derivado de los datos mediante las reglas de inferencia o mediante otros mecanismos aplicados

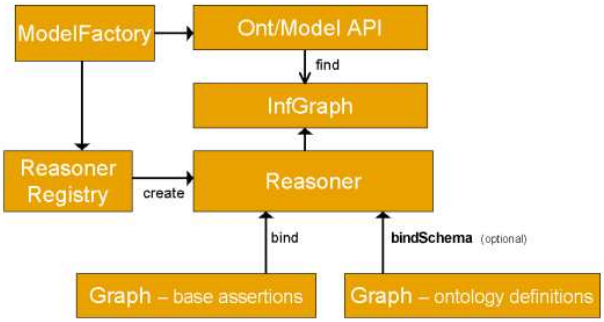


Figura 2.6: Esquema del motor de Razonamiento de Jena [Apache, 2014a]

por el razonador.

Como se observa en la Figura 2.6, el mecanismo de inferencia se aplica realmente a nivel de grafo, a fin de que cualquiera de las diferentes interfaces del modelo puedan construirse en torno a este. En particular, la API de ontología proporciona formas adecuadas de vincular razonadores a los modelos de ontologías u *OntModels* que construye. Como parte de la API RDF general, también está disponible un modelo de inferencia o *InfModel*, que extiende la interfaz común del modelo para proporcionar control y acceso al grafo para llevar a cabo inferencia. También incluye una clase estática a través de la que es posible registrar nuevos tipos de razonadores y realizar búsqueda de razonadores de un determinado tipo dinámicamente, se trata del *ReasonerRegistry*.

2.5.2. Sensibilidad al Contexto

Los sistemas sensibles al contexto son capaces de adaptar su funcionamiento al contexto del usuario, sin necesidad de la intervención explícita del mismo. Su objetivo principal es aumentar la facilidad de uso y transparencia de los sistemas de inteligencia ambiental gracias a la información de contexto [Baldauf et al., 2007]. De este modo, es lógico que uno de los primeros pasos a llevar a cabo para el diseño y desarrollo de un middleware sensible al contexto sea obtener una definición adecuada de qué es el "contexto". En sus inicios, el desarrollo de plataformas sensibles al contexto estaba ligado principalmente con la geolocalización de usuarios. Más tarde, la definición de "context awareness" ha evolucionado mucho y en la actualidad el término incluye no solo la ubicación o el tiempo, sino también cualquier otra información que pueda ser relevante en la interacción entre usuarios y aplicaciones. Sin embargo, esta información debe ser representada de un modo que sea compresible por los sistemas software, y es en este punto donde las técnicas de representación de la información ligadas a los sistemas de información aparecen como modelos de representación del contexto.

2.5.3. Modelos para la Representación del Conocimiento

Son diversos los modelos de representación del conocimiento a considerar en función de la estructura de datos utilizada para el manejo de la información semántica, de entre los que destacan aquellos que mejor se adaptan a los nodos de cómputo con recursos limitados, propios de los entornos ubicuos. Desde que en 2004 Strang and Linnhoff-Popien [Strang and Linnhoff-Popien, 2004] publicasen su estudio detallado sobre modelado de contexto, son varios los artículos de investigación de tipo *survey* dedicados a analizar la adecuación de los distintos modelos de representación de la información en computación ubicua. En estos trabajos se realiza una comparación, principalmente, entre los modelos key-value, los basados en lenguajes de marcas, los modelos gráficos, orientados a objetos y basados en ontologías.

Los modelos key-value utilizan una representación del conocimiento basada en pares clave-valor, de forma que cada variable asociada a un elemento concreto posee un valor. Es un modelo sencillo y fácil de manejar, pero inadecuado para emplear estructuras que posibiliten la utilización de algoritmos eficientes de recuperación de información de contexto. Además, establece una concordancia exacta entre los elementos y sus valores, sin dar pie a posibles ambigüedades en estas relaciones debidas a algún tipo de interpretación condicional, lo cual es frecuente en la representación de la información de contexto.

Los lenguajes basados en marcas emplean una jerarquía de etiquetas para los atributos y el contenido de los elementos del modelo de conocimiento. Un ejemplo son los perfiles XML, que están asociados a posibles estados del entorno, y se intercambian ante transformaciones en el estado físico del contexto. El problema de estos modelos es que la forma de capturar las relaciones de contexto más complejas es difícil y poco intuitiva. Algunos ejemplos de modelos de este tipo son Centaurus Capability Markup Language (CCML) [Kagal et al., 2001], Comprehensive Structured Context Profiles (CSCP) [Buchholz et al., 2004] y Pervasive Profile Description Language (PPDL) [Ferscha et al., 2006].

Los modelos gráficos emplean UML o diagramas entidad-relación para representar y estructurar la información existente. Los modelos orientados a objetos son similares a los modelos gráficos, y además combinan la información de cada dispositivo representado como un objeto para obtener el concepto de contexto como una abstracción superior. También los enfoques basados en la lógica se emplean para modelar el contexto utilizando propiedades matemáticas para representar de forma abstracta las entidades del entorno.

Por último, los modelos basados en ontologías garantizan un modelo uniforme de representación del conocimiento, que pueden compartir todos los elementos del entorno ubicuo. Las ontologías proporcionan un modelo global de los diferentes tipos de información de contexto que se pueden utilizar para describir las situaciones particulares de los dominios del contexto. Una vez representada la información del contexto, se puede aplicar razonamiento sobre esta de un modo lógico y diferentes entidades pueden entender y utilizar el conocimiento generado. Ejemplos de modelos de este tipo son CoBrA system [Chen et al., 2004b], ASC model of Context Ontology

Tabla 2.2: Idoneidad de los modelos para sistemas ubicuos

Requeriment Approach	Distributed Composition	Partial Verification	Richness and Quality of Information	Incompleteness and Ambiguity	Level of Formality	Applicabity to existing frameworks
Key-Value Models	-	-	–	–	–	+
Markup Scheme Models	+	++	-	-	+	++
Graphical Models	–	-	+	-	+	+
Object Oriented Models	++	+	+	+	+	+
Logic Based Models	++	-	-	-	++	-
Ontology Based Models	++	++	+	+	++	+

Language (CoOL) [Strang et al., 2003] y CONON [Wang et al., 2004].

Podemos hacer una comparación entre los modelos indicados según los requisitos fundamentales que habría que exigir a un modelo de contexto para su uso en un entorno de computación ubicua. Dicha comparación puede consultarse en la tabla 2.2 [Strang and Linnhoff-Popien, 2004].

Un trabajo de investigación similar es el de Gasevic et al. [Gasevic et al., 2006], en el que también se plantean distintas técnicas de representación del conocimiento. El autor comienza introduciendo las tripletas *objeto-atributo-valor*, similares a los modelos key-value, que representan datos sobre los objetos y sus atributos. Una tripleta *O-A-V* afirma que el objeto O posee un valor V para el atributo A. Se obtiene una extensión de este modelo al dotar de incertidumbre a los objetos, asignando un factor numérico de certeza a las tripletas en función del grado de creencia en su declaración. De una forma similar, la lógica fuzzy representa la incertidumbre utilizando la imprecisión y ambigüedad de los términos comunes en lenguaje natural. Otro tipo de representación del conocimiento serían los modelos basados en reglas, que relacionan una o más premisas con una o más situaciones, consecuencias o acciones, de la forma *If-Then-Else*. Las redes semánticas o mapas conceptuales son una técnica de representación del conocimiento que trata de reflejar el modelo psicológico asociativo de los humanos, representando gráficos formados por objetos, conceptos, y relaciones entre ellos. Otra técnica de representación de conocimiento son los *frames*, similares a las clases y objetos en la programación orientada a objetos. En esta técnica se distinguen los *class frames* que representan características comunes de un conjunto de objetos similares y los *instance frames*, casos concretos de las clases. Por último, también se introducen las *ontologías* como modelo de representación del conocimiento, enfatizando sus características de expresividad, pues proporcionan un vocabulario compartido de representación del conocimiento

en un dominio determinado; coherencia, gracias a la categorización o clasificación jerárquica de las entidades dentro del dominio de aplicación; y compartición de conocimiento y reutilización, pues una ontología proporciona una descripción de los conceptos y relaciones que pueden existir en un dominio y que se pueden compartir y reutilizar entre agentes inteligentes y aplicaciones.

A la vista de lo anterior, parece que las ontologías abarcan los requisitos fundamentales para la representación del conocimiento en sistemas de computación ubicua. Además, una ontología común permite el intercambio de conocimientos de una manera abierta y dinámica en sistemas distribuidos y permite interoperar a los elementos del sistema y a los dispositivos aunque no hayan sido expresamente diseñados para trabajar juntos [Chen et al., 2004b][Almeida and López-de Ipiña, 2012], compartiendo un entorno de conocimiento semánticamente bien definido [Ye et al., 2011].

2.5.4. Ontologías como Dominio de Conocimiento

Aunque el concepto de ontología ha estado presente desde hace mucho tiempo en la Filosofía, actualmente también se utiliza en TIC para definir vocabularios que las máquinas puedan entender y que se especifiquen con la suficiente precisión como para permitir diferenciar términos y referenciarlos de manera precisa. Una ontología representa una forma de manifestar el conocimiento común y compartido de un dominio concreto, interpretable por sistemas computacionales, lo cual permite que pueda ser usada por aplicaciones informáticas. Representa así un modelo de datos ligado a un dominio de información y las relaciones entre los objetos de este dominio, con el fin de realizar razonamiento sobre el mismo. Un sinónimo que usualmente se utiliza para referirse a este término es el de "conceptualización", pues una ontología representa un modelo abstracto de un entorno concreto del mundo, en el que se identifican los conceptos, sus propiedades y relaciones. También es muy usual tratar a las ontologías como "sistemas de representación de conocimiento", lo cual está muy ligado a la inteligencia artificial. Para trabajar con ontologías es necesario utilizar un lenguaje de ontología adecuado, y la opción que actualmente es más utilizada en los proyectos de investigación relacionados con las tecnologías de Web Semántica es OWL.

En una plataforma de servicios ubicuos en la que se tengan en cuenta aspectos de la semántica del entorno, las ontologías representan el modelo de datos, en el sentido de que todos los recursos y las descripciones de todos los datos intercambiados durante el uso de un servicio están basados en ontologías. Según la definición dada por Maedche "Una ontología es una teoría lógica formada por un vocabulario y un lenguaje lógico. En un dominio de interés formaliza signos que describen cosas en el mundo real, permitiendo una correspondencia entre los signos y las cosas tan exacta como sea posible" [Maedche and Staab, 2004]. El amplio uso de ontologías permite mayor procesamiento semántico de la información y da soporte a la interoperabilidad [Pogorelc et al., 2012].

Existen trabajos de investigación que definen su propio vocabulario de ontología

y los que abogan por la reutilización de ontologías ya existentes, siempre que estas cubran las necesidades del entorno en el que serán aplicadas. Uno de los principales problemas derivados de la utilización de ontologías, al igual que con el resto de lenguajes basados en XML, es que la reutilización no es directa, si no que los desarrolladores pueden necesitar ampliar constantemente los modelos existentes para aplicarlos a dominios de conocimiento más concretos. La reutilización de ontologías es un tema de investigación actual, difícil de abordar y en torno al cual giran los estudios de Web Semántica, que va más allá de esta tesis.

Dentro del ámbito de la computación ubicua son numerosos los proyectos que definen ontologías como modelo de representación de la información de contexto con vocación de reutilización, de entre las que destacan CONON [Wang et al., 2004], CoDAMoS [Preuveneers et al., 2004], COBRA-ONT [Chen et al., 2004b] y SOUPA [Chen et al., 2004a].

La ontología CONON, acrónimo de *CONtext ONtology*, define conceptos generales, como localización, actividad, persona o entidad, de forma que pueda ser extensible añadiendo nuevos conceptos específicos del ámbito de aplicación. Para ello los autores dividen la ontología en dos, una ontología general y de más alto nivel que especifica los conceptos generales, junto con ontologías específicas que cubran los detalles de cada subdominio.

Por otro lado, CoDAMoS, acrónimo de *Context-Driven Adaptation of Mobile Services*, define cuatro entidades centrales, que son: el usuario, el entorno, la plataforma y el servicio. Esta ontología ha sido diseñada con el objetivo de resolver la adaptabilidad de aplicaciones, la generación automática de código, así como su movilidad y adecuación a distintos dispositivos.

El objetivo principal de la ontología COBRA-ONT es permitir el intercambio de conocimientos y el razonamiento basado en ontologías dentro de la infraestructura COBRA, *Context Broker Architecture*.

Esta última es una extensión de SOUPA, acrónimo de Standard Ontology for Ubiquitous and Pervasive Applications. SOUPA pretende definir un vocabulario genérico y universal para distintas aplicaciones de computación ubicua. Además el núcleo de SOUPA puede extenderse haciendo uso de distintas ontologías en función del dominio de contexto, como es el caso de COBRA-ONT.

En función de qué ontología se esté analizando, cobrarán fuerza determinadas clases y relaciones presentes en su dominio de conocimiento [Poveda Villalon et al., 2010]. En la tabla 2.3 se presenta un breve resumen de los dominios de conocimiento característicos de las aplicaciones de computación ubicua presentes en las distintas ontologías analizadas. Como podemos observar, ninguna de ellas los considera todos, por lo que considerarlas en un desarrollo podría implicar su extensión para adecuarlas a las necesidades de cada proyecto.

En torno al concepto de ontología giran numerosos proyectos de investigación que han tratado de desarrollar plataformas eficientes e innovadoras en el ámbito de la computación ubicua sensible al contexto. En el proyecto de Machuca et al. [Machuca et al., 2005] se emplea una plataforma de agentes inteligentes con una adaptación en tiempo real a los cambios en el contexto, para proporcionar una adaptación de

Tabla 2.3: Dominios de conocimiento considerados en las distintas ontologías de contexto analizadas

	CONON	CoDAMoS	Cobra-ONT	SOUPA
Device	X	X	X	-
Environment	-	X	X	-
Interface	-	-	-	-
Location	X	X	X	X
Network	X	-	-	-
Role	-	X	X	-
Service	X	X	-	-
Time	-	X	X	X
User	X	X	X	X

sus servicios a la ubicación de los usuarios, sus preferencias o el estado de los dispositivos. En [Vazquez et al., 2006] se presenta SOAM, como un modelo experimental para la creación de objetos inteligentes que utilizan ontologías en la web, es decir, las tecnologías de Web Semántica, para permitir la comunicación entre la semántica del contexto y los procesos de razonamiento, con el fin de proporcionar una adaptación del entorno a las preferencias de los usuarios. También utiliza perfiles de comportamiento en base a los cuales se establece un modelo colaborativo entre los distintos objetos semánticos del entorno. En el proyecto de [Ejigu et al., 2008] se presenta una plataforma de servicios basada en un modelo de contexto híbrido. En dicha plataforma el funcionamiento está basado en el razonamiento sobre la información de contexto, su semántica y las normas y políticas relacionadas, combinando los modelos de representación relacional y ontológico de colaboración. De este modo, el funcionamiento está basado en el razonamiento sobre la información de contexto, su semántica y las normas y políticas relacionadas. Utiliza una arquitectura de peer-to-peer para implementar la colaboración entre peers vecinos. Como vemos, son numerosos los proyectos de investigación que emplean las ontologías para modelar el contexto del entorno ubicuo de sus aplicaciones sobre diversos tipos de arquitecturas, basadas en servicios, agentes, objetos o peers.

En [Becker and Nicklas, 2004] se aboga por la utilización de ontologías en el modelado de contexto y se realiza un análisis de las ventajas que podría tener para la utilización de la información de contexto la combinación de dos modelos: el modelo de contexto espacial y el modelo ontológico. De esta forma, se ganaría la eficiencia en la gestión del contexto del modelo espacial y la expresividad semántica de las ontologías. La combinación de varias técnicas de representación es una opción acertada, pues permite aunar las ventajas de cada una de forma aislada, para construir un modelo más completo y consistente.

En [Chen et al., 2003] se apoya la utilización de ontologías en un sistema sensible al contexto sobre una base computacional de agentes inteligentes, en el que se considera que la omnipresencia de los agentes con recursos limitados necesita apoyo externo para su desarrollo. En este trabajo desarrollan un agente basado en

la arquitectura Context Broker Architecture (CoBrA) para facilitar las tareas de adquisición, razonamiento y compartición de la información del contexto. La ontología que emplean está definida utilizando OWL, y modela los conceptos básicos del contexto como personas, agentes, lugares, eventos y presentación. En este caso, la ontología también está ligada a la arquitectura del sistema desarrollado, basada en agentes.

Otro tipo de investigaciones, también relacionadas con la utilización de ontologías y el razonamiento a nivel semántico sobre el contexto, tratan de estandarizar términos y sembrar las bases estables de un desarrollo sostenible en la materia. Es el caso de [Ou et al., 2006], que enfatiza la importancia de emplear criterios estándar para llevar a cabo el proceso de desarrollo de los sistemas sensibles al contexto. En su trabajo, Ou aplica MDA para hacer frente al desarrollo de estas aplicaciones, estableciendo un modelo con dos niveles para la obtención de la ontología de contexto, que junto a un modelo de integración, permita generar implementaciones de forma semiautomática.

Convertir las propiedades del mundo real en datos comprensibles por una máquina es una tarea compleja, que puede llevarse a cabo a través de las ontologías, pero que requiere además que el modelo de razonamiento utilizado para procesar la información del contexto considere la incertidumbre inherente al proceso de captación, para lo cual es necesario introducir técnicas de control y deducción [Cook and Das, 2007]. En el trabajo de [Simpson et al., 2006], los autores plantean como la inteligencia artificial podría no solo inferir las actividades de los usuarios, sino también completar determinadas tareas si fuese necesario. El razonamiento temporal combinado con un sistema basado en reglas se utiliza en [de Mántaras and Saitta, 2004] para identificar situaciones peligrosas y restablecer la seguridad en el entorno del usuario. En [Hagras et al., 2004] se monitoriza el estado del entorno usando reglas fuzzy obtenidas a través de la observación del comportamiento de los habitantes. Estas reglas pueden agregarse, modificarse o incluso suprimirse, permitiendo una adaptación del entorno a los cambios en el comportamiento de sus habitantes. Estos trabajos, apoyan la utilización de reglas y lógica fuzzy en el proceso de razonamiento a partir de la información obtenida del entorno, lo cual puede ser de gran utilidad combinado con una representación del conocimiento adecuada al modelo de contexto a utilizar.

Capítulo 3

Metodología de Investigación

Thus not only the mental and the material, but the theoretical and the practical in the mathematical world, are brought into more intimate and effective connexion with each other.

· Ada Lovelace ·
Notes upon L. F. Menabrea's "Sketch of The Analytical Engine Invented by Charles Babbage" (1842)

3.1. Tipo de estudio

El trabajo de investigación realizado durante el desarrollo de esta tesis puede considerarse una mezcla equilibrada entre investigación cuantitativa, cualitativa y experimental. Desde la especificación de los objetivos de la tesis se ha seguido un plan de actuación planificado para llevarse a cabo durante los años de realización del trabajo.

Las tareas llevadas a cabo han sido de base tecnológica y estrechamente ligadas con la innovación en las tecnologías de la información y las comunicaciones (TIC). La incorporación del conocimiento científico y tecnológico al proceso de desarrollo de los objetivos planteados al inicio del trabajo, persigue la obtención de un resultado que cumpla un fin valioso y necesario.

3.2. Método científico y Design Science

La aplicación de una metodología de investigación bien establecida durante el proceso de desarrollo de cualquier trabajo de investigación facilita su estructuración, correcto desarrollo y finalización. En el caso de esta tesis la metodología utilizada ha sido principalmente el método científico. Sin embargo, debido a una serie de colaboraciones establecidas con otros grupos de investigación durante estos años, el trabajo aquí expuesto también se ha visto influenciado por la metodología conocida como *Design Science* (DS).

La aplicación del método científico es una metodología básica y presente en muchos de los trabajos de investigación actuales. En dicho método se parte de una cuestión bien formulada a la cual se trata de dar respuesta a lo largo de la investigación mediante la observación sistemática, la experimentación y el planteamiento de

una hipótesis. El testeo y la experimentación alrededor de dicha hipótesis permite al investigador obtener gran cantidad de información que ayude a verificar o refutar la cuestión de partida. A menudo, este proceso iterativo implica incluir cambios en la hipótesis derivados de las conclusiones obtenidas a raíz de los resultados de los testeos. La modificación o reformulación no resulta en absoluto negativo para la investigación, sino que ayuda a arrojar luz sobre la cuestión de partida generando conocimiento y dando cuerpo a los posibles resultados de la tesis en la cual se engloban. En nuestro caso, la cuestión de partida de la investigación ha estado claramente definida desde sus inicios.

> "Existen gran cantidad de trabajos de investigación orientados a la obtención de entornos ubicuos inteligentes. El uso de plataformas basadas en servicios es muy frecuente en dichos trabajos. Analizando las necesidades y peculiaridades específicas en este tipo de sistemas, detectamos una coincidencia muy recurrente centrada en la teorización del desarrollo sin tener en cuenta que los dispositivos sobre los que se ejecutarán los programas son dispositivos empotrados con recursos, tanto de potencia de cálculo como de memoria, muy limados, además de otras restricciones propias de este tipo de sistemas, como son la dinamicidad y el carácter distribuido de los servicios, la colaboración entre servicios y el uso de la información ambiental para llevar a cabo un comportamiento proactivo sin la intervención del usuario. Entonces, como ingenieros software, ¿podríamos clarificar y establecer cuáles son las pautas a tener en cuenta así como los elementos y requerimientos más influyentes en el proceso de desarrollo de software para entornos ubicuos inteligentes y sensibles al contexto? Nuestro enfoque está centrado en diseñar un modelo software que permita el desarrollo de una plataforma de servicios para computación ubicua completa, teniendo en cuenta las restricciones propias de este tipo de sistemas."

Considerando esta hipótesis de partida, partimos de SOA como enfoque ampliamente aceptado para el desarrollo software de sistemas ubicuos, pasando por todos los niveles de abstracción: desde la interacción con el hardware, pasando por la comunicación entre servicios, y llegando hasta la interacción proactiva entre los mismos. Una vez establecida dicha cuestión, sobre la cual se asientan las bases de esta tesis, se ha llevado a cabo un proceso de investigación guiado por el método científico. Sin embargo, a lo largo de todo este tiempo las influencias sobre el trabajo de investigación han sido muchas, entre las cuales no solo se encuentran las relativas a la temática concreta del proyecto, sino también relativas a la metodología. Es el caso de *Design Science* (DS).

El primer acercamiento a esta metodología se produjo durante la estancia de investigación realizada en la Dublin City University (DCU). El grupo de investigación en sistemas de información en el que he realizado una estancia de investigación, el Business Informatics Group (BIGroup), utiliza DS como metodología de

investigación y su entusiasta defensa de la misma, no como alternativa, sino como complemento para mejorar el resultado final de mi tesis me ha hecho tenerla en cuenta.

La metodología de *Design Science* (DS) propone un conjunto de técnicas de análisis para la realización de la investigación en base al diseño de artefactos innovadores y el análisis de la utilización de dichos artefactos para mejorar y comprender el comportamiento de determinados aspectos de las tecnologías de la información y las comunicaciones. Así, el principal objetivo de DS es la definición de un artefacto innovador y con un claro objetivo. Dicho artefacto debe ser evaluado con el fin de asegurar su utilidad real en el dominio de aplicación específico del problema. Además, este artefacto debe o bien resolver un problema hasta ahora no resuelto, o bien, aportar una mejora factible a las soluciones previamente planteadas. Un artefacto puede ser desde un algoritmo, hasta una compleja metodología de diseño de sistemas software.

Tras llevar a cabo un análisis de los principios de DS y su adecuación para ser tenidos en cuenta como metodología complementaria de esta tesis, determinamos su utilización en el diseño de desarrollo de la capa semántica del middleware desarrollado. Para ello, determinamos los pasos a llevar a cabo para el desarrollo del artefacto obtenido como resultado:

- Definir los criterios que se van a utilizar para llevar a cabo la evaluación del sistema. Escribir una definición clara de cada uno de ellos. Analizar como los requisitos operacionales afectan a dichos criterios, para lo cual es necesario identificar previamente dichos requisitos operacionales (usabilidad, eficiencia y efectividad).
- Definir un caso de estudio.
- Definir una ontología para representar la información del entorno.
- Analizar las distintas partes a tener en cuenta en el desarrollo, teniendo en cuenta que los requisitos serán una parte dinámica relativa a las mismas:
  - Perfiles de datos o *data profiling*.
  - Información del entorno.
  - Preferencias de usuario: haciendo uso de los perfiles de datos y de la información del entorno representada mediante ontologías, el usuario establecerá sus preferencias con respecto al comportamiento proactivo del sistema.

3.3. Desarrollo de la tesis

El trabajo de investigación y los resultados recogidos en esta tesis se han llevado a cabo siguiendo una planificación estructurada en base a los objetivos planteados inicialmente. Para ello, las tareas que se han incluido en el trabajo a realizar durante

este tiempo han estado ligadas a un proceso de desarrollo e investigación iterativo, en el que el análisis de trabajos relacionados o búsqueda bibliográfica, el planteamiento de modelos de base teórica, la implementación de las teorías planteadas y la realización de pruebas y evaluación de resultados ha sido un denominador común. Y por último, y como ocurre en todo trabajo de investigación, la publicación de resultados ha formado parte de todas y cada una de dichas tareas.

3.3.1. Proceso iterativo: PFC, Máster y PhD

La consecución de esta tesis ha sido el resultado de un proceso de desarrollo e investigación iterativo que comenzó con la realización del Proyecto Fin de Carrera (PFC) de la titulación de Ingeniería Informática. En dicho proyecto, titulado "*Control domótico del hogar utilizando JXTA*" y finalizado en 2007, la temática predominante fue el desarrollo de una plataforma software basada en servicios con el fin de promover los espacios ubicuos habitables. Una de las principales características innovadoras presentadas en la plataforma propuesta en dicho PFC fue el uso de una comunicación entre servicios distribuida y basada en P2P, rompiendo así con el clásico binomio cliente-servidor presente en este tipo de sistemas.

Tomando como base dicho PFC, se inició un proceso de investigación más detallado y profundo en el área de los sistemas ubicuos e inteligencia ambiental con el desarrollo del Proyecto Fin de Máster (PFM) del Máster en Desarrollo de Software de la Universidad de Granada. Como resultado del PFM se obtuvo el proyecto "*Plataforma de Servicios Sensibles al Contexto. Aplicación a la Oficina Domótica*", que fue desarrollado durante el año 2008. Muchos de los problemas analizados durante el desarrollo del proyecto estuvieron centrados en buscar soluciones software a los problemas presentes en los nuevos espacios de interacción.

Una vez finalizados ambos proyectos, PFC más centrado en el desarrollo de una plataforma y PFM más ligado a la investigación del estado actual de la investigación en el área, el interés por las líneas de investigación definidas quedó patente y decidí comenzar el proyecto de tesis. Así, a finales del año 2010 me embarqué en un nuevo proyecto, a caballo entre la investigación y el desarrollo, que ha dado como resultado el texto actual que recoge los resultados más destacados obtenidos durante mi doctorado. Como ingeniera e investigadora, el modo de afrontar y resolver los distintos problemas que se han ido planteado durante todo este proceso ha estado basado en el uso de metodologías software para el desarrollo de modelos teóricos basados con los principios de la computación distribuida que puedan ser implementables y testeables en herramientas o plataformas software reales, ya sean en modo prototipo o bien a nivel de explotación real, una vez superado un periodo de depuración.

Como principal hilo conductor tanto de PFC, PFM y Tesis, podríamos destacar el uso de los principios de SOA para llevar a cabo el diseño y desarrollo de plataformas software en computación ubicua. Con este trasfondo, otros conceptos han ido haciéndose presentes en el proceso de investigación y añadiendo complejidad y a la vez contenido a los resultados obtenidos. Por ejemplo la consideración de las restricciones hardware de este tipo de sistemas, con dispositivos embebidos de pequeño

tamaño y recursos tanto de procesamiento como de memoria muy restringidos, la composición de servicios capaces de llevar a cabo acciones colaborativas más complejas que para las que inicialmente fueron diseñados, y la inclusión de información semántica capaz de enriquecer el contenido de los servicios y modificar su comportamiento hacia acciones proactivas, propias de la sensibilidad al contexto. Así, durante este proceso hemos ido respondiendo a distintas preguntas para las cuales no siempre hemos encontrado una única y refutable respuesta sino un conjunto de consideraciones a tener en cuenta en el desarrollo de aplicaciones. Algunas de ellas son:

- ¿Podemos construir la lógica de un sistema de inteligencia ambiental en entornos ubicuos compatible con SOA?
- ¿Es viable implementar esta lógica en dispositivos empotrados de pequeño tamaño y escasos recursos?
- ¿Pueden los servicios obtenidos colaborar entre ellos para llevar a cabo operaciones complejas con restricciones de Tiempo Real?
- ¿Pueden los servicios reconocer la información del entorno e interpretarla?
- ¿Pueden los servicios desempeñar un comportamiento proactivo inteligente sin la intervención del usuario?

En esta tesis están incluidas todas las conclusiones y resultados obtenidos tras estos años de análisis, investigación y desarrollo relativos a estas cuestiones y muchas otras cuestiones que han ido surgiendo por el camino. Aunque el proceso de realización del doctorado parece finalizar con la presentación de este trabajo, nuestra intención es sin duda continuar con el proceso iterativo comenzado ya años atrás y en el que estamos seguros aún queda mucho camino que andar.

3.3.2. Búsqueda bibliográfica

La revisión de trabajos relacionados se ha llevado a cabo en base a los principales tópicos planteados en los objetivos de investigación iniciales así como en criterios de calidad de los mismos.

Algunos de los tópicos que han estado presentes en la revisión realizada han sido Ambient Assited Living (AAL), Agents (agentes), Ambient intelligence (inteligencia ambiental), Context Awareness (sensibilidad al contexto), Design Science (metodología DS), Graphs (grafos), Home-Automation (domótica), IoT (internet de las cosas), JXTA (framework de desarrollo P2P), Ontologies (ontologías), Pervasive/Ubiquitous computing (computación ubicua o pervasiva), QoS (calidad de servicio), Service Composition (composición de servicios), Service Selection (selección de servicios), SOA-P2P (P2P en plataformas SOA), SOA-Semantic Services (servicios semánticos en plataformas SOA) y Tuple Spaces (espacios de tuplas).

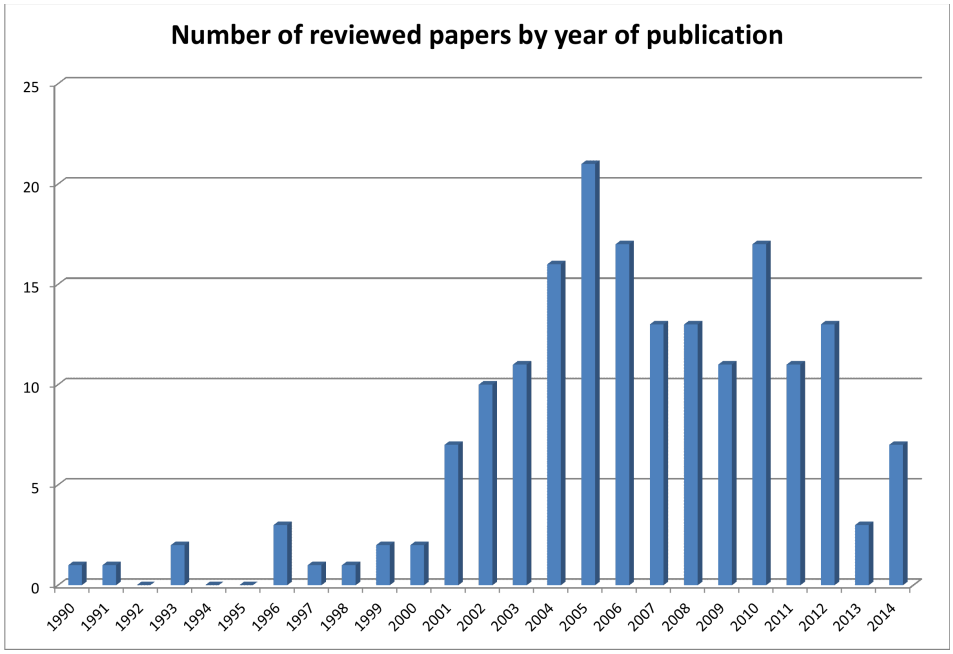


Figura 3.1: Estadística de artículos revisados por año de publicación

A lo largo de esta tesis están citados todos aquellos artículos de investigación, libros, páginas web, reportes técnicos, manuales, etc., que se han utilizado durante su realización. A modo de estadística se presentan las figuras 3.1 y 3.2 en las cuales se representa un resumen numérico de los artículos de investigación revisados por año de publicación y por temática principal, respectivamente.

3.3.3. Implementación, pruebas y evaluación

La visión del uso de los computadores es muy distinta cuando hablamos de computación ubicua. Como ya hemos visto, las principales características de este tipo de sistemas giran en torno al manejo de gran cantidad de información proveniente de sensores dispersos por el entorno, muchas veces imperceptibles para los usuarios, información que debe ser interpretada de forma homogénea, independientemente de su origen. Otra característica igualmente importante es la dinamicidad, pues tanto los dispositivos físicos, sensores o actuadores, como los servicios, pueden estar activos o no en función de la percepción de los usuarios. Para garantizar la obtención de características como las mencionadas es necesario un entorno de computación que combine el control y acceso a los dispositivos hardware, sensores y actuadores, la comunicación entre servicios, el descubrimiento dinámico de nuevas funcionalidades, y la disponibilidad para los usuarios, en un middleware completo que de soporte a la interoperabilidad inherente en este tipo de sistemas. Por otra parte, se espera que los dispositivos se comporten de modo inteligente, capturando el estado del entorno habitable y en función de las necesidades de los usuarios anticiparse a sus

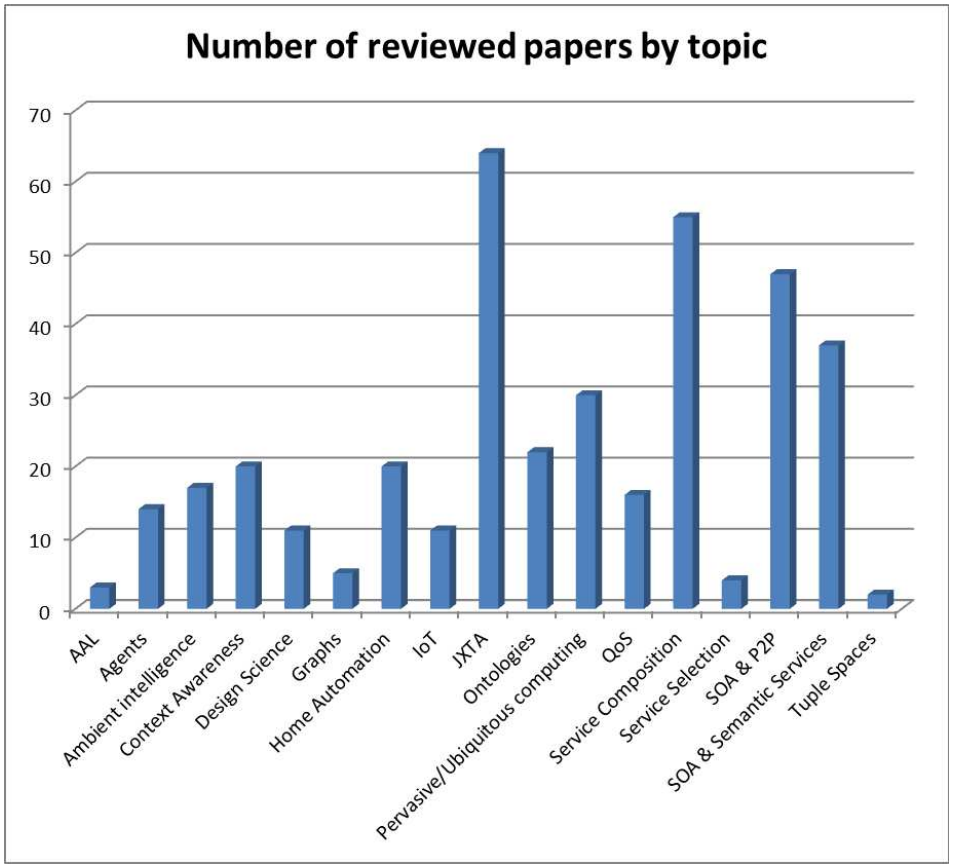


Figura 3.2: Estadística de artículos revisados por tópico principal asociado

necesidades y modificar si es necesario el estado de dicho entorno.

La programación de entornos ubicuos bajo un enfoque SOA posee ventajas ya mencionadas, como la facilidad para integrar tecnologías heterogéneas, evolución de los modelos de forma dinámica y capacidad de reemplazar aplicaciones SOA sin contrapartida sobre el resto, entre otras. Pero los requerimientos de la computación ubicua son mayores, y la programación en base los principios SOA también posee aspectos complejos de resolver como por ejemplo cómo encontrar cuál es el servicio que necesitamos entre todos los servicios de la red o cómo podemos componer servicios sencillos para formar otros servicios más complejos. Estas cuestiones no son tareas triviales, y para resolverlas están tomando protagonismo los servicios semánticos, en los que se enriquece a los servicios con información procedente de una ontología. La semántica permite expresar formalmente los conceptos y relaciones en los que están basados los datos reales de un sistema, con el objetivo de que, a partir de esta semántica, puedan desarrollar un comportamiento más inteligente, llegando a combinar tareas hasta conseguir un objetivo, descubrir errores e incluso desempañar acciones proactivamente.

La utilización de información semántica en computación ubicua es un tema de investigación candente, que puede ser abordado bajo enfoques muy dispares, y en el que creemos que la estandarización tiene un papel muy importante. La utilización de modelos semánticos, ya sean ontologías, lenguajes o arquitecturas, que puedan ser reutilizados con posterioridad garantiza un desarrollo abierto, capaz de avanzar hacia modelos más complejos y completos. Lejos de la consecución de un objetivo concreto, la utilización de semántica en las plataformas de servicios promueve el uso de servicios abiertos, totalmente escalables y reutilizables.

3.3.4. Publicación de resultados

Durante el desarrollo de la tesis se han ido seleccionando un publicando los resultados obtenidos de mayor relevancia para la comunidad investigadora. Durante el procedimiento de publicación de resultados se ha participado en distintos congresos y conferencias, tanto nacionales como internacionales, así como en revistas científicas.

Por otro lado, durante el desarrollo de la tesis se han compaginado tareas de docencia, por lo que los resultados obtenidos también han repercutido en sendas publicaciones docentes y de divulgación científica en el ámbito de la computación distribuida. Estas otras publicaciones, que aún no estando directamente relacionadas con los temas tratados en esta tesis, han contribuido a mi formación en el área del Desarrollo de Software, los Sistemas Concurrentes y Distribuidos, y de los Lenguajes y Sistemas Informáticos.

La lista total de publicaciones derivadas de la tesis puede consultarse en el capítulo 8.

Parte II

Contribución de la Tesis

Capítulo 4

Plataforma de Servicios para Computación Ubicua

Next comes ubiquitous computing, or the age of calm technology, when technology recedes into the background of our lives.

Mark Weiser (23 July 1952 – 27 April 1999)
The Computer for the 21st Century

4.1. Introducción

El control automático del hogar o de forma más general la domótica es un dominio de aplicación de la Ingeniería Informática con un creciente interés. El ámbito de aplicación de esta modalidad de desarrollo se centra en la automatización y el control del hogar y de los dispositivos electrónicos disponibles en entornos habitables, ya sean hogares residenciales, oficinas o incluso fábricas. Los objetivos principales de esta tendencia giran en torno a la satisfacción del confort, el entretenimiento, la seguridad y las comunicaciones de los usuarios, así como facilitar una mejor conservación, eficiente energéticamente y cuidado del espacio.

Las demandas de los usuarios pueden ser cubiertas por dispositivos individuales localizados en el entorno o bien por aplicaciones con propósitos específicos de control, supervisión y monitorización. Por ejemplo, una cámara de control, un sistema de climatización o de aire acondicionado, la alarma de un reloj, o un equipo de audio y televisión. Sin embargo, cuando el número de dispositivos crece, gestionar la interacción entre los mismos de forma sostenible facilita enormemente su gestión.

Tanto desde el ámbito de investigación universitario como industrial, se están llevando a cabo grandes esfuerzos con el objetivo de desarrollar innovadores sistemas que incrementen el nivel de autonomía e incluso "inteligencia" en los sistemas de control domótico. Estos sistemas se diseñan con el objetivo de habilitar la interconexión entre numerosos dispositivos, además de añadir servicios de valor añadido a los mismos. Por ejemplo, facilitar la colaboración entre dichos dispositivos para llevar a cabo una funcionalidad determinada de acuerdo con las necesidades y deseos de los usuarios.

Como consecuencia de estos esfuerzos, el mercado está siendo saturado con múltiples propuestas que soportan un amplio rango de tecnologías (HAVi, Jini, OSGi, UPnP), redes de comunicaciones cableadas (X10, EIB/KNX, Insteon, LonWorks) e

inalámbricas (WiFi, Bluetooth, ZigBee, Z-Wave). Del mismo modo, son muy abundantes los dispositivos electrónicos disponibles, la mayoría de ellos incompatibles entre sí, así como las aplicaciones de control desarrolladas tanto software como hardware [Das and Cook, 2006] [Gárate et al., 2005b] [Gárate et al., 2005a] [Haraikawa et al., 2005] [Ricquebourg et al., 2006].

A pesar de la gran diversidad de consorcios, grupos de trabajo y organizaciones internacionales que han participado en la redacción de especificaciones que ayuden a establecer un estándar en el ámbito, las expectativas no han evolucionado como era de esperar [Wacks, 2002]. Son muchas las razones que han contribuido a este hecho. Tal vez, la razón más obvia es la gran heterogeneidad de dispositivos, redes de comunicaciones, plataformas hardware e infraestructuras software existentes.

Los sistemas de automatización pueden llegar a convertirse en entidades aisladas dentro del entorno en el que ofrecen su funcionalidad. Esto se debe a que los sistemas suelen estar compuestos por gran cantidad de dispositivos autónomos conectados a una red específica. Esta conexión es a menudo solo física, pues los dispositivos continúan siendo incompatibles entre sí. La interconexión perfecta entre dispositivos es muy limitada y problemática, estando además restringida la extensibilidad del sistema a la oferta tecnológica proporcionada por el fabricante concreto. Por ejemplo, un sistema de seguridad en el hogar requiere colocar gran cantidad de sensores, como sensores de presencia o cámaras IP, para la gestión de la vigilancia de la casa. A su vez, el sistema de aire acondicionado suele estar gestionado por un controlador remoto independiente. Por lo tanto, aún estando ambos sistemas instalados en el mismo entorno, no hay ninguna interacción entre ellos. En definitiva, no hay conectividad ni interoperabilidad. La incorporación de nuevos servicios implica inevitablemente la configuración, supervisión y control de nuevos controladores, lo cual, a su vez, complica la interacción con el usuario.

La utilización de gateways es uno de los enfoques más utilizados para mejorar la conectividad y la interoperabilidad entre distintas redes de dispositivos [Bourcier et al., 2006] [Umberger et al., 2009] [Valtchev and Frankov, 2002]. Sin embargo, esto requiere llevar a cabo un complejo proceso de integración en el que se deben crear adaptaciones software personalizadas para cada arquitectura de sistema [Gacitua-Decar and Pahl, 2008] [Lalanda et al., 2006] [Pinto et al., 2003]. En muchos casos, la arquitectura del sistema, la topología de la red o la infraestructura software son demasiado rígidas para permitir la interoperabilidad entre distintos dominios de red. El desarrollo de dispositivos reconfigurables puede ser una opción, pero este enfoque está restringido en muchos casos debido a las particularidades de la topología de cada red [Bobda and Ahmadinia, 2005].

La proliferación de distintos modelos de comunicación y el uso generalizado de Internet en cualquier dispositivo, como es el caso de los dispositivos móviles, complican un poco más la escena relacionando los sistemas de automatización con los procesos de negocio [Lalanda et al., 2006]. Los procesos de negocio ofrecen servicios complejos y sofisticados basados en Internet, entre los que se encuentra la automatización del entorno cotidiano. En este sentido, es viable llevar a cabo la integración en el proceso de nuevos dispositivos de cómputo con un funcionamiento autónomo en

base al contexto existente en el entorno [Park et al., 2003]. La convergencia entre la funcionalidad ofrecida por los distintos dispositivos y los procesos de negocio necesita de nuevos paradigmas de computación donde la interacción entre el funcionamiento del dispositivo y el proceso de negocio resulte más natural y espontánea.

Para superar estas dificultades, nuestra visión de la automatización del entorno debe cambiar. Creemos que el entorno cotidiano debe ser visto como un espacio de interacción abstracta poblado de dispositivos interconectados, servicios que encapsulan funcionalidad y múltiples interfaces a través de las cuales el usuario puede interactuar con estos dispositivos fácilmente, de acuerdo con la visión de computación ubicua. La computación ubicua o pervasiva contempla espacios de interacción repletos de dispositivos, cada vez más pequeños, de menor costo, y con un consumo energético muy reducido. En ellos, la interacción se lleva a cabo de forma natural, no intrusiva y transparente para el usuario y para al resto de dispositivos o servicios. Los dispositivos, que a bajo nivel tienen una funcionalidad básica, relegan en una abstracción de nivel superior su control a alto nivel. En los servicios SOA.

El desarrollo de plataformas software bajo el paradigma SOA debe gestionar la heterogeneidad y movilidad de dispositivos y usuarios, permitir el acceso constante a la funcionalidad de los servicios y a los recursos de información disponibles, garantizar la seguridad y fiabilidad de las comunicaciones con tolerancia a fallos, la interoperabilidad entre diferentes redes, la extensión, adaptación y evolución de la funcionalidad, y por último, la existencia de interfaces multimodales de usuario. Algunos de estos requerimientos son también propios de la computación ubicua y a su vez gestionados por otros paradigmas, como la computación distribuida. Es el caso de la comunicación remota, la tolerancia a fallos, la alta disponibilidad y el acceso seguro a la información remota y distribuida [Satyanarayanan, 2001]. También en el dominio de la computación móvil se gestionan las redes móviles, el acceso a la información móvil, el uso de aplicaciones adaptativas, el control del consumo eléctrico y la sensibilidad a la localización de los usuarios [Banavar et al., 2000]. Sin embargo, los entornos ubicuos requieren de modelos de computación novedosos y más específicos, y por lo tanto, nuevas infraestructuras software para apoyar la escalabilidad, la heterogeneidad, la integración, la invisibilidad y la percepción [Saha and Mukherjee, 2003], y nuevos dispositivos empotrados con alta conectividad a pesar de poseer recursos limitados, baja potencia y escasa memoria [Weiser, 1993].

4.2. Objetivos de la Plataforma de Servicios

El propósito central de esta tesis ha sido desarrollar una plataforma de servicios abierta, confiable y dinámica que proporcione soporte a los sistemas de computación ubicua como un espacio de interacción en el que conviven servicios y dispositivos accesibles por parte de los usuarios y los propios dispositivos. La plataforma de servicios semánticos sensibles al contexto es capaz de manejar la heterogeneidad en los dispositivos conectados en cuanto a su hardware, software y/o protocolos de comunicación utilizados y facilita la escalabilidad de las aplicaciones que se construyan

sobre ella.

Para el diseño de la arquitectura se ha considerado un enfoque orientado a servicios conforme a los principios SOA [Erl, 2005]. La ventaja fundamental de dicho diseño es que favorece la partición de los sistemas y aplicaciones (especialmente complejos) en entidades cohesivas de bajo acoplamiento y autónomas con capacidad para intercomunicarse entre sí en tiempo de ejecución. Además es posible agregar propiedades semánticas a dichos servicios de forma que estos puedan adaptarse automáticamente al contexto en el que se ejecutan según las preferencias de los usuarios. El resultado obtenido es una plataforma de servicios que cubre las necesidades de los desarrolladores de aplicaciones para computación ubicua desde el acceso al hardware, pasando por la interacción entre servicios, y llegando al comportamiento proactivo de los mismos. La plataforma de servicios conjuga a dos capas software autocontenidas pero complementarias la una de la otra, DOHA, Dynamic Open Home-Automation, y SenSE, SENsitive Services Environment.

Los principales objetivos que han guiado las decisiones de diseño que se han tomado durante el proceso de construcción de la plataforma de servicios son:

1. **Interoperabilidad.** La construcción de aplicaciones a partir de la plataforma de servicios proporciona capacidad de interacción a cualquier dispositivo facilitando en cualquier instante la intercomunicación entre los distintos tipos de servicios y dispositivos heterogéneos. Por tanto, se han abordado mecanismos para integrar diferentes tecnologías e infraestructuras hardware y software dentro de la plataforma.

2. **Acceso a dispositivos físicos.** La gran cantidad de dispositivos distribuidos en el entorno y la diversidad de características de los mismos añade complejidad a la construcción de plataformas middleware para computación ubicua. Además se demanda un comportamiento cada vez más inteligente a dichos dispositivos por lo que es necesario que puedan interactuar con el resto de dispositivos del entorno. Por lo tanto, es necesario por un lado manejar la heterogeneidad de los diferentes dispositivos del hogar y disponer de abstracciones y mecanismos ligeros que faciliten el acceso a la capa física de los dispositivos (hardware y software de sistemas) tanto para capturar información del entorno como para actuar sobre el entorno. En estos casos los servicios encapsulan los recursos físicos que proporcionan los dispositivos sobre los que se ejecutan.

3. **Colaboración entre servicios.** La construcción de aplicaciones basadas en servicios colaborativos plantea numerosos problemas que tienen que considerarse, y cuyo resultado se plasma en el diseño de la plataforma de servicios. Para llevar a cabo la interacción entre servicios de forma acotada en el tiempo y dotar a la plataforma de características de Soft Real-Time se ha diseñado un modelo de composición estático basado en grafos dirigidos acíclicos, que además asegura que la composición de servicios es siempre válida sin la posibilidad de existir invocaciones erróneas entre operaciones de distintos servicios que produzcan ciclos infinitos de ejecución. Este componente de estaticidad

permite en todo momento acotar el tiempo de ejecución de la colaboración entre servicios y permite también verificar el comportamiento del sistema en su conjunto.

4. **Servicios Semánticos.** Con la utilización de servicios semánticos se pretende potenciar la comunicación y colaboración entre servicios de forma más natural. La descripción semántica de los servicios es procesable por las máquinas, lo cual permite automatizar los procesos de descubrimiento, composición y ejecución de servicios, lo que posibilita a los dispositivos y equipos el inicio de operaciones sobre el entorno del hogar de forma automática sin necesidad de la intervención del usuario.

5. **Sensibilidad al contexto.** La consideración de características de sensibilidad al contexto permite obtener un sistema que se anticipará a las necesidades de los usuarios y actuará en consecuencia gracias a la información obtenida del entorno. Los servicios sensibles al contexto son más personales y adaptados, de forma que son capaces de proporcionar el servicio más correcto, a la persona adecuada, en el mejor momento y a través del canal de información idóneo.

6. **Perfiles de comportamiento.** Los usuarios pueden indicar sus preferencias en cuanto al comportamiento autónomo de los servicios a través de perfiles de comportamiento que emplean la semántica de la plataforma establecida por sus ontologías. De este modo, el comportamiento proactivo de las aplicaciones construidas sobre la plataforma de servicios se adecuará a los gustos y necesidades de sus usuarios.

7. **Restricciones de tiempo real.** En un sistema de computación ubicua los dispositivos también pueden interactuar en el espacio de interacción y colaborar con otros dispositivos para conseguir un determinado fin en un intervalo de tiempo acotado. Por ejemplo, un regulador software puede necesitar acceder a sensores y actuadores para controlar el funcionamiento de un servicio del hogar como el sistema de iluminación dentro de una determinada consigna de luz. En estos casos, si hay una realimentación en el sistema, es necesario conocer las restricciones temporales que ofrecen los servicios para optimizar su funcionamiento. La plataforma podría garantizar una ejecución de tiempo real no estricta de servicios teniendo en cuenta las condiciones de calidad del servicio del sistema y de los servicios involucrados.

4.3. Arquitectura de la Plataforma de Servicios

La arquitectura software de la plataforma de servicios se estructura en base a distintos niveles de abstracción. En la Figura 4.1 se muestran los distintos niveles considerados y la relación existente entre ellos.

A bajo nivel la plataforma considera tanto los dispositivos hardware disponibles en el entorno, ya sean sensores o actuadores, como los sistemas operativos y redes,

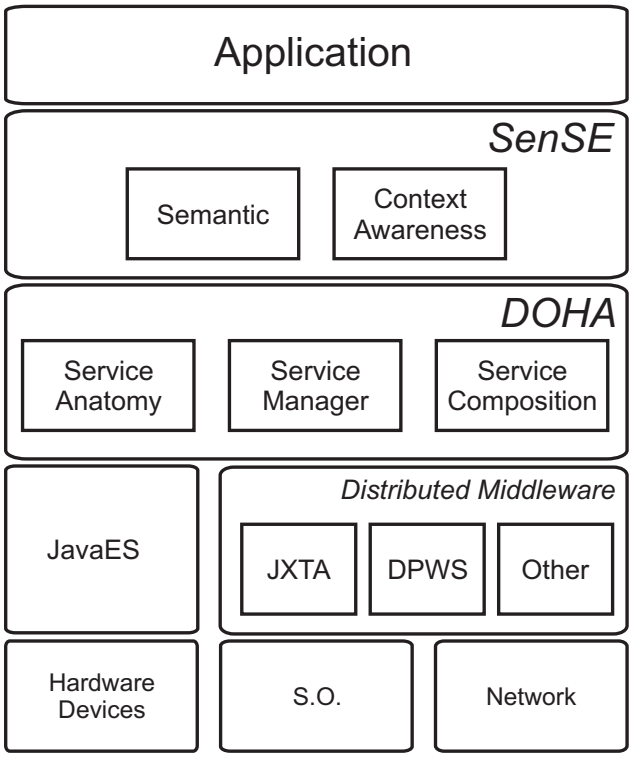


Figura 4.1: Arquitectura abstracta de la plataforma de servicios

cableadas o no, que lleven a cabo la interconexión de los mismos.

Por encima, se considerará el middleware de comunicación distribuido sobre el que se implemente la plataforma de servicios. Actualmente hay disponible una implementación sobre JXTA y otra sobre DPWS, así como un nuevo proyecto para la implementación sobre servicios REST. A este nivel hemos considerado también a JavaES. Se trata de un middleware que permite abstraer la complejidad y heterogeneidad de los dispositivos hardware empotrados y trabajar con ellos como objetos de alto nivel.

Sobre estos middleware se encuentra la plataforma de servicios, a la que hemos denominado DOHA. El elemento principal de la plataforma es el concepto de servicio, siendo la anatomía de servicios la característica fundamental para su construcción. En base a dicha anatomía y la definición de servicios que se realiza en base a ella, la plataforma permite llevar a cabo la comunicación y colaboración entre servicios.

A más alto nivel se encuentra la capa semántica de la plataforma, a la que hemos denominado SenSE. A este nivel se considera la semántica del entorno ubicuo y de los servicios construidos sobre él. Así, gracias a la consideración de semántica por parte de la plataforma se podrán diseñar servicios semánticos y servicios sensibles al contexto.

Sobre la plataforma de servicios está el nivel de aplicación. Los desarrolladores

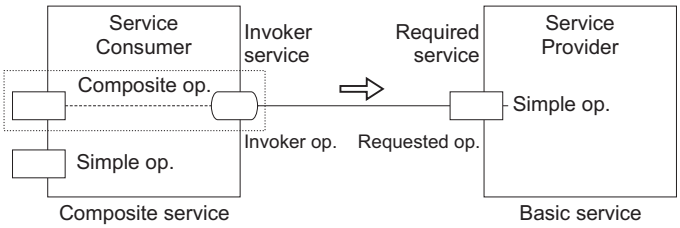


Figura 4.2: Principales elementos involucrados en la comunicación entre servicios

podrán construir distintos tipos de aplicaciones y configurar su funcionalidad haciendo uso de las facilidades de diseño e implementación que aporta la plataforma de servicios.

4.4. El concepto de Servicio

El servicio es el elemento principal del paradigma SOA [Erl, 2005]. Un servicio, *service*, es un componente autónomo y autocontenido capaz de llevar a cabo actividades específicas o funciones independientemente y que acepta una o más peticiones tras las que produce una o más respuestas a través de una interfaz estándar y bien definida. Un servicio puede asumir diferentes roles, en base al contexto en el cual se esté utilizando. Un servicio proveedor, *service provider*, es responsable de proporcionar una funcionalidad específica al resto de servicios, encapsulada en una unidad software autónoma. Un servicio consumidor, *service consumer*, es un servicio que solicita una funcionalidad concreta de los servicios proveedores. Así, la comunicación entre servicios y los conceptos involucrados en la misma, como muestra la Figura 4.2, son esenciales en la construcción de una plataforma basada en servicios.

Los servicios ofrecen su funcionalidad a través de operaciones. Las operaciones de un servicio son públicas y accesibles desde cualquier servicio consumidor. Una operación simple, *simple operation*, es una transacción simple que el propio servicio puede llevar a cabo por sí mismo – i.e., el servicio posee todos los recursos necesarios para llevar a cabo la operación y no requiere de la interacción con ningún otro servicio para ello. Por otro lado, las operaciones compuestas, *composite operation*, requieren la ejecución de un conjunto de operaciones formado por una o más operaciones en al menos un servicio proveedor.

Podemos denominar como *servicio básico*, *basic service*, a aquellos servicios que únicamente poseen operaciones simples, mientras que a un servicio que posee operaciones compuestas lo denominaremos *servicio compuesto*, *composite service*. Un servicio consumidor es siempre un servicio compuesto. Este debe implementar operaciones compuestas para invocar las operaciones de los servicios proveedores. Sin embargo, un servicio proveedor puede ser básico o compuesto. Las operaciones de un servicio proveedor pueden ser operaciones compuestas que requieren la invocación de otras operaciones en otros servicios proveedores para completar su propia funcionalidad. Para ejecutar una operación compuesta, el servicio que actúa como consumidor

solicita la ejecución de otras operaciones en los servicios proveedores. Por lo tanto, en una operación compuesta, siempre habrá un servicio invocador, *invoker service*, que actúe con rol cliente en la comunicación con el servicio proveedor. Del mismo modo, siempre existirá al menos un servicio que actúe como proveedor o servicio requerido, *required service*, cuya funcionalidad es necesaria para completar la funcionalidad del servicio consumidor. No todas las operaciones compuestas tienen que ser públicas e invocadas desde otros servicios consumidores. Cuando una operación compuesta no puede ser invocada desde otra operación compuesta en un servicio consumidor, se denomina operación compuesta final, *final composite operation*.

En el contexto de la interacción, las operaciones también pueden jugar distintos roles en función de en qué parte de la comunicación se encuentren. Las operaciones que están del lado del servicio consumidor se denominan operaciones invocadoras, *invoker operation*, mientras que las operaciones del lado de los servicios proveedores se denominarán operaciones requeridas, *requested operation*.

Los servicios pueden tener diferentes tipos de responsabilidad en el sistema. En la plataforma de servicios podemos clasificarlos en base a diferentes criterios:

1. *Naturaleza física del servicio.* El servicio puede ser un servicio dispositivo cuando encapsula alguna funcionalidad que requiere el acceso a la capa física de un dispositivo. Si el servicio encapsula alguna funcionalidad que hace referencia a recursos de datos o un algoritmo los denominamos servicios lógicos.

2. *Naturaleza semántica del servicio.* Cuando el servicio proporciona capacidad semántica lo denominamos servicio semántico, mientras que en otro caso el servicio solo tendría capacidad sintáctica (o servicio de interacción). Se profundizará en el concepto en la siguiente sección.

3. *Naturaleza de sensibilidad al contexto.* En este caso el servicio puede proporcionar capacidad reactiva frente al estado del entorno captado.

4.5. Colaboración de Servicios con Restricciones de Tiempo

La composicionalidad entre servicios permite ofrecer una funcionalidad más compleja y de valor añadido que aquella para la que los servicios han sido diseñados de forma aislada. Para ello, desarrollar una plataforma de servicios requiere afrontar el diseño del modelo de composición a aplicar para llevar a cabo la colaboración entre servicios.

En un entorno ubicuo, el modelo de composición debe satisfacer determinadas restricciones adicionales que a priori no se suelen considerar en otros ámbitos, como los procesos de negocio. En el diseño de la plataforma de servicios se ha considerado como requisito fundamental para la definición del modelo de composición la satisfacción de restricciones de tiempo en la colaboración entre servicios.

Considerar restricciones de tiempo real "soft" asociadas al modelo de composición supone asociar una cota de tiempo a las operaciones de los distintos servicios. Esta cota de tiempo podrá ser empleada por los desarrolladores a modo de función determinista para poder determinar si la ejecución del servicio es factible o no. Así mismo, le permitirá planificar la ejecución de los servicios, predecir el tiempo de ejecución necesario para llevar a cabo una tarea ligada a un proceso colaborativo, y considerar hasta qué punto los recursos de los que dispone el hardware son suficientes para ello.

Más allá de la ejecución de un servicio concreto, la cota de tiempo asociada a los servicios colaborativos permite también realizar planificación de sistemas. Puesto que cada servicio involucrado en un proceso colaborativo tendrá asociado un tiempo de ejecución determinado, el desarrollador podrá establecer el tiempo de ejecución del sistema en su conjunto.

La información del tiempo de ejecución de un servicio será también útil para otros servicios que quieran utilizarlo. Además, las aplicaciones podrán evolucionar añadiendo nuevos servicios a los ya existentes manteniendo sus restricciones de tiempo real. Los nuevos servicios consultarán los parámetros temporales de los ya existentes y actualizarán a su vez sus tiempos de ejecución.

4.6. Servicios Semánticos

La incorporación de semántica en una arquitectura de servicios nos lleva a lo que se conoce como Arquitecturas Semánticas Orientadas a Servicios, SSOA por su acrónimo en inglés . SSOA introduce capacidades semánticas en los servicios, de forma que una entidad consciente del modelo semántico puede combinar automáticamente servicios de forma dinámica para satisfacer sus objetivos [Sycara and Martin, 2006].

En un sistema de servicios basado en ontologías, estas se utilizarían como modelo de representación del conocimiento. En base a ellas, los servicios podrán especificar sus características semánticas e interpretar las de los demás servicios. A partir de la combinación de ontologías y SOA, se encapsula el concepto de servicio dentro de la ontología, relación que se convierte en un servicio más de SOA, que podríamos denominar "servicio semántico" (SS) [Korotkiy and Top, 2006]. En este sentido, además de que la ontología especifica los servicios, el lenguaje establecido por la ontología debe utilizarse para expresar los mensajes entre servicios. El conocimiento que el consumidor tiene de un servicio concreto es sólo el que obtiene a través de la ontología.

Para que un servicio pueda comprometerse con una ontología determinada es suficiente con que este sea capaz de interpretar adecuadamente un mensaje creado de acuerdo a la misma. De esta manera se abstraen las características sintácticas y estructurales de los mensajes, centrándose en los aspectos semánticos definidos en la ontología. Se incorpora también el concepto de servicio a la ontología. El usuario puede introducir nuevos conceptos ontológicos relacionados con la semántica "operacional" de los conceptos. La semántica "operacional" se refiere a la manera en

que un concepto afecta a la conducta de un servicio.

La utilización de SS potencia que la funcionalidad de los servicios pueda ser descubierta e invocada dinámicamente desde los programas sin necesidad de indicar expresamente el servicio invocado en el código del programa. La creación de los SS requiere un esfuerzo previo para llevar a cabo la definición del dominio de datos mediante ontologías, pero una vez está disponible, se consigue una mayor automatización y un mejor dinamismo en la ejecución de los servicios, por ejemplo, permitiendo una selección dinámica de servicios.

La estructuración del conocimiento en torno a ontologías, posibilita una definición semántica de los servicios que facilita diferentes actividades automáticas en una arquitectura orientada a servicios fundamentadas en la recuperación, interpretación y procesado de la información recogida en las ontologías, de entre las cuales destacan:

- Publicación de servicios en un registro semántico.
- Descubrimiento automático en función de la petición de servicio y de la descripción semántica publicada del servicio.
- Selección de servicios cuando se pueda producir un conflicto entre varios servicios que satisfagan una petición.
- Composición, un framework de SS permitirá la colaboración entre servicios, de forma que se puedan componer los resultados de varios servicios para obtener una funcionalidad de un nivel más alto.
- Invocación de SS, validación de los parámetros de entrada contra la ontología del servicio y ejecución del servicio.
- Despliegue.
- Gestión de las ontologías.

La arquitectura de una plataforma de SS tendría que incorporar una serie de componentes para llevar a cabo razonamiento automático en función de la información que tiene del dominio. Un componente sería la *ontología* del servicio que represente sus capacidades y sus restricciones de uso. Esta ontología integrará la semántica del servicio con su descripción y considerando tanto la información funcional del servicio (entradas, salidas, precondiciones y postcondiciones) como de la información no funcional (categoría, coste y calidad de servicio). Otro componente a considerar sería un *Razonador* que permita realizar consultas en base a la descripción semántica de los servicios e interpretarlas en función de esta.

4.6.0.1. Colaboración de Servicios Semánticos

Un aspecto clave en el que se incide especialmente durante el desarrollo de plataformas de sistemas basados en SOA es en la orquestación y coordinación de servicios.

Los servicios pueden establecer mecanismos de colaboración basados en orquestación o coreografía, donde la orquestación implica que el control del flujo de ejecución siempre corre a cargo de una de las partes involucradas en la comunicación, mientras que la coreografía conlleva un grado de colaboración mayor, permitiendo a cada servicio gestionar libremente su parte en la interacción [Peltz, 2003].

En un escenario real, una aplicación basada en los principios SOA exige, por un lado, modelos de interacción complejos y con el más bajo acoplamiento posible, establecer condiciones de sincronización asociadas a la interacción entre servicios que faciliten su coordinación, además de la posibilidad de establecer relaciones de composición más complejas y dinámicas entre servicios.

En la actualidad están en desarrollo diversos frameworks para trabajar con SS, aunque hay varias propuestas en activo basadas en la utilización de OWL-S. La ontología propuesta por OWL-S está formada por tres componentes fundamentales: el service profile, empleado para el descubrimiento y selección de servicios; process model, ontología encargada de describir con detalle las operaciones procesadas por un servicio; y el service grounding, que describe como interactuar con el servicio, a través de mensajes. Sin embargo, para que una plataforma de SS pueda llevar a cabo un comportamiento colaborativo, será necesario además que cumpla con las siguientes características:

1. *Comunicación e interacción*: es fundamental que el modelo de comunicación e interacción entre los servicios colaborativos sea correcto para garantizar la adecuación de las comunicaciones y evitar errores.

2. *Organización*: es necesario también gestionar dinámicamente la colaboración del conjunto de servicios de forma global para distribuir información a todos los miembros de una manera eficiente.

3. *Compartir conocimiento y recursos*: para ello se debe tener en cuenta la forma de presentar este conocimiento o recursos, pues debe ser de tal forma que todas las entidades involucradas en la colaboración los descubran e interpreten de una forma homogénea. Además, la forma de generar y modificar nuevo conocimiento o recursos debe ser coherente en todos los servicios colaborativos, preservando la integridad de la información y garantizando que se satisfacen determinadas condiciones de seguridad en su almacenamiento sin que afecten a su disponibilidad.

4. *Contexto*: todos los servicios colaborativos deberán compartir el mismo espacio de contexto, alegóricamente, como si se encontrasen en la misma habitación virtual.

4.7. Servicios Sensibles al Contexto

En una plataforma de servicios sensibles al contexto el conocimiento del entorno permite a los servicios tomar medidas automáticamente, reducir la participación

directa de los usuarios, disminuyendo la carga comunicativa del sistema, y la prestación de asistencia proactiva inteligente. El conocimiento del entorno, la forma de representar el contexto y la información del mundo físico que es interpretable por parte de un sistema repercutirá en el grado de "consciencia" de sus elementos; por ello, la forma de modelar este entorno es importante.

El concepto de sensibilidad al contexto o *context awareness* está muy presente en la investigación actual sobre sistemas de computación ubicua e inteligencia ambiental e implica dotar a estos sistemas de consciencia del entorno en el que se ejecutan, es decir, convertirlos en servicios sensibles al contexto. Una posible definición de contexto sería: "Contexto es la información que puede ser usada para caracterizar la situación de una entidad. Entendiendo que entidades son las personas, localizaciones u objetos que son considerados como relevantes para el comportamiento de una aplicación, y que por sí mismas son consideradas como parte de su contexto" [Becker and Nicklas, 2004]. La información de contexto no solo viene determinada por el entorno, sino también por los elementos presentes en él, que pueden variar en el tiempo, actuando de forma directa sobre el estado del mismo.

Una plataforma de servicios sensible al contexto debe utilizar toda la información contextual del mundo físico que pueda obtener a partir de los sensores, lo cual implica considerar problemas como: (a) la conexión entre la información contextual de la que son conscientes los servicios y la información que proporcionan los sensores, (b) la forma de adquirir la información del sensor por parte de los servicios y (c) la manera en que los servicios razonan sobre la información obtenida desde los sensores para inferir contexto. No toda la información que puede ser obtenida a través de sensores debe considerarse como información útil, sino aquella que permita representar el estado concreto del entorno, los estados emocionales de los usuarios o sus movimientos.

4.7.1. Elementos de un sistema sensible al contexto

Los sistemas software de inteligencia ambiental son sistemas proactivos, autónomos, comunicativos e inteligentes. Para dotar a estos sistemas de context awareness se deben convertir las propiedades del mundo real en datos comprensibles por una máquina. Según Loke, un *context-aware pervasive system* debe poseer tres elementos básicos: sensores, procesamiento (pensamiento) y actuadores [Loke, 2006].

Los sensores deben permitir la adquisición de datos o información desde el mundo físico. Una combinación de múltiples sensores puede proporcionar una visión mucho más completa del mundo físico al sistema. La complejidad estriba en concretar qué tipo de información deben capturar estos sensores y en qué lugar, dónde deben situarse estos para facilitar la comprensión del mundo físico y la adquisición de información por parte del sistema. Podríamos distinguir dos formas de adquisición de conocimiento, el conocimiento obtenido mediante el raciocinio, aplicando la razón, o el conocimiento empírico, obtenido a través de la experiencia y los sentidos. El conocimiento empírico está en constante ampliación, pues siempre estamos captando información a través de los sentidos, que razonamos y utilizamos como conocimiento

nuevo. Este es el caso de los datos o la información que los sistemas sensibles al contexto obtienen del mundo físico a través de los sensores y que deben procesar para convertir en conocimiento.

Los datos del mundo físico pueden venir dados para el sistema como valores discretos o continuos. El sistema puede emplear distintas técnicas para obtener, clasificar y procesar la información que le aporten los sensores hardware, como por ejemplo modelos físicos y matemáticos, como el filtrado de Kalman [Sarkka et al., 2012]; técnicas de inferencia, como reconocimiento de patrones, redes neuronales y razonamiento Bayesiano [Ramakrishnan et al., 2013]; o modelos cognitivos, como bases de conocimiento y lógica difusa [Kapitanova et al., 2012]. Sea cual sea el modelo de razonamiento utilizado para procesar la información del contexto es muy importante tener en cuenta la incertidumbre existente en el sistema. Para hacer frente a esta es posible utilizar redes Bayesianas o incluir la participación de los usuarios mediante diálogo entre el usuario y el sistema que ayude a la resolución de la ambigüedad. Puesto que incluir al usuario en la toma de decisiones puede ser tedioso para el mismo, se debe minimizar esta interacción empleando, por ejemplo, técnicas de redundancia o uso de valores por defecto.

Una vez que se ha recopilado la información del contexto y se ha reconocido la situación, es necesario tomar las medidas adecuadas según la aplicación. Es conveniente que las acciones se lleven a cabo antes de que la información recopilada sobre la situación cambie, siempre procurando que el usuario conserve el control del sistema de forma que pueda anular, cancelar, detener o invertir el efecto de estas acciones.

4.7.2. Contexto centrado en el usuario o en los sensores

La información que puede obtenerse del mundo físico y ser interpretada como contexto en los sistemas con context awareness puede ser muy diversa en función de la orientación que siga la aplicación concreta. En un sistema centrado en el usuario, la información del contexto siempre girará en torno a este, desde la captación del estado del entorno para adaptarlo a las preferencias del usuario, hasta la observación del estado anímico del usuario para establecer un contexto favorable al mismo [Raz et al., 2006].

Puesto que el usuario está en constante movimiento, el contexto no será estático, sino que estará allí donde se encuentre el usuario en cada momento. Esto añade complejidad extra a la hora de implementar un sistema sensible al contexto pues, al considerar un mundo físico tan cambiante, el conjunto de objetos que pueden involucrarse en la detección del contexto es mucho mayor. Todos aquellos aspectos relacionados con la usabilidad son de gran importancia en un diseño centrado en el usuario, como son el acceso permanente al sistema, el empleo de comunicación multimodal o la adaptación a los conocimientos e intereses de los usuarios.

Otra opción sería la realización de un diseño centrado en los sensores. De esta forma, no estaríamos considerando un mundo físico tan cambiante, sino que el mundo se reduciría al conjunto de sensores que sean conocidos en el modelo de espacio.

En este caso, cobraría más importancia la localización de los sensores, siendo de especial importancia la localización en el mundo físico de cada uno de los objetos del entorno. Puesto que en todo momento el sistema debe ser abierto y extensible, la consideración de un modelo de datos en el que se especifiquen todos los posibles objetos del mundo real que pueden existir en el sistema, debe ser modificable y ampliable.

Ambos enfoques para modelar el contexto, centrado en el usuario y centrado en los sensores, son válidos de forma independiente y a la vez complementarios, por lo que lo ideal sería combinarlos de forma que se tomen los aspectos más relevantes de ambos enfoques para definir un modelo híbrido más completo.

4.7.3. Influencia activa versus Influencia pasiva

La computación ubicua está ligada a la Inteligencia Artificial, hasta el punto de que ambas comparten como objetivo final el conseguir un comportamiento inteligente. Quizás, no más haya de la realidad, el hito de obtener un sistema inteligente esté más ligado a la ciencia ficción que al pragmatismo de la ciencia, pero son numerosos los enfoques que trabajan para acercarse al ideal. Es el caso de los sistemas sensibles al contexto, cuyo objetivo es manifestar un comportamiento proactivo dentro del entorno de los usuarios. La proactividad es un término acuñado por Victor Frankl, un neurólogo y psiquiatra austriaco que sobrevivió a los campos de concentración Nazis, en su libro *Man's Search for Meaning*, "El hombre en busca de sentido" [Frankl, 1984]. La palabra procede del latín, y está compuesta de dos palabras: *pro*, raíz latina "pro-", que significa "antes de" *actividad*, raíz latina "activitas, -atis", que significa facultad de obrar, diligencia o eficacia. La proactividad es una actitud en la que el sujeto asume el pleno control de su conducta vital de modo activo, lo que implica la toma de iniciativa en el desarrollo de acciones creativas y audaces para generar mejoras. El concepto opuesto es el de reactividad, o tomar una actitud pasiva y ser sujeto de las circunstancias.

Cuando el funcionamiento de un sistema depende de las solicitudes realizadas por los usuarios, se dice que este es reactivo. En este caso el usuario tiene una influencia activa sobre el sistema, según el modelo tradicional de acción-respuesta, donde el usuario es el que debe iniciar explícitamente las operaciones de los servicios. Cuando un sistema desarrolla una influencia pasiva no es necesaria la invocación de órdenes por parte de los usuarios para que el sistema lleve a cabo acciones sobre el entorno, sino que reacciona en respuesta a los cambios en el contexto, de forma más o menos esperada por el usuario [Vazquez et al., 2007]. Este tipo de influencia es muy ventajosa para los usuarios, pues se reduce al máximo su participación explícita y aumenta la asistencia proactiva inteligente desde el sistema. No obstante, se debe tener en cuenta la capacidad de control de los usuarios, la opción del usuario siempre es la preferente. Además, el sistema debe evitar ser intrusivo, pues para el usuario puede resultar negativa su actuación de forma inadecuada o contradictoria a sus preferencias.

A continuación se presenta un esquema de los dos tipos de influencias comen-

tados. En la Figura 4.3 se puede observar como el usuario ejecuta una orden sobre una entidad del sistema que conlleva la ejecución de una determinada acción sobre el entorno, en definitiva, el usuario actúa directamente sobre las entidades.

Figura 4.3: Esquema del modelo de influencia activa sobre el sistema

En cambio, en la Figura 4.4 se muestra como el usuario no interactúa directamente con el sistema ni ejecuta órdenes en el mismo, sino que las entidades son capaces de percibir la información del entorno asociada al usuario y llevar a cabo acciones sobre este, sin necesidad de que el usuario las solicite explícitamente, este solo es un sujeto pasivo del sistema.

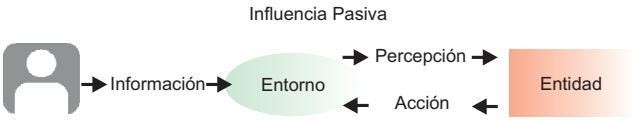


Figura 4.4: Esquema del modelo de influencia pasiva sobre el sistema

El objetivo de la computación sensible al contexto es combinar ambas perspectivas, de forma que el usuario siempre mantenga el control del sistema pudiendo llevar a cabo la emisión de determinadas órdenes solicitando acciones concretas, mientras que el sistema, además, implementa un modelo de inteligencia ambiental basado en la información del entorno y en mecanismos de razonamiento. De esta forma el comportamiento del sistema estará influenciado pasivamente por los cambios en el contexto, es decir, reaccionará ante cambios en el entorno llevando a cabo determinadas acciones según unos perfiles de comportamiento preestablecidos, sin la emisión explicita de órdenes por parte de los usuarios, siempre evitando ser intrusivo de cara a estos.

4.8. Discusión

En el desarrollo de una plataforma de servicios para computación ubicua entran en juego gran cantidad de conceptos importantes. Durante el modelado y el análisis de la plataforma presentada en esta tesis se ha prestado una mayor atención al concepto de servicio en sí y a la colaboración entre servicios como valor añadido de la plataforma. En el Capítulo 5 se analiza con mayor detalle cómo se ha llevado a cabo la construcción de DOHA como plataforma de servicios colaborativos con restricciones temporales. A más alto nivel, la consideración de la semántica ha sido también un elemento crucial de esta investigación. Cómo un servicio puede ofrecer

un valor añadido a nivel software convirtiéndose en un servicio semántico o en un servicio sensible al contexto. En el capítulo 6 se muestran todos los detalles relativos a la consideración de semántica en la plataforma de servicios, así como las decisiones de diseño que se han tomado para la construcción de SenSE, la capa semántica de la plataforma de servicios.

Capítulo 5

Plataforma de Servicios Colaborativos

Abstract: Este capítulo presenta los detalles de diseño de la plataforma de servicios desarrollada en base a los principios SOA y a una arquitectura descentralizada. Se ha denominado a la plataforma DOHA, acrónimo de *Dynamic Open Home Automation*, dado que está orientada al desarrollo de sistemas domóticos en espacios de interacción ubicuos en base a una arquitectura basada en servicios. Por otro lado, DOHA da soporte a la composición de servicios, de forma que sea posible ofrecer funcionalidad compleja a través de la colaboración de varios servicios, todo ello de manera transparente para el usuario. Los servicios DOHA encapsulan los dispositivos hardware del entorno a modo de recursos, lo cual hace más fácil su gestión. Estos y otros detalles serán discutidos a fondo en las siguientes secciones.

5.1. Introducción

La gran complejidad de los sistemas ubicuos conlleva a su vez una dificultad asociada al desarrollo de aplicaciones que satisfagan determinados requisitos no funcionales. Una estrategia recurrente para evitar dicha complejidad es el uso de tecnologías middleware y arquitecturas software. La arquitectura de un sistema distribuido puede ser centralizada o descentralizada, en base a la localización del componente principal del sistema, también conocido como controlador. Es frecuente encontrar propuestas de desarrollo de middleware para computación ubicua basadas en arquitecturas centralizadas [Nakamura et al., 2008]. Sin embargo, la tendencia actual en el desarrollo de sistemas ubicuos es la utilización de arquitecturas distribuidas y descentralizadas.

En esta línea, se ha desarrollado una plataforma de servicios según los principios establecidos por el paradigma SOA. El middleware obtenido se ha denominado DOHA, acrónimo de Dynamic Open Home Automation. Así, DOHA proporciona una plataforma distribuida, descentralizada, abierta y dinámica para facilitar el acceso, control y gestión de espacios ubicuos considerando cualquier dispositivo de

cómputo, desde un sensor vestible, a un dispositivo móvil o un ordenador personal. La plataforma DOHA ha sido diseñada específicamente para el desarrollo de aplicaciones de automatización en entornos ubicuos, ocultando la complejidad subyacente y ligada a la utilización de dispositivos heterogéneos con distintos protocolos de comunicación. Como resultado se ha obtenido una plataforma interoperable que permite llevar a cabo la comunicación entre servicios utilizando distintas tecnologías middleware de comunicaciones.

El elemento principal de DOHA es el servicio. Los servicios de DOHA pueden interactuar entre ellos, comunicarse y colaborar para desempeñar una funcionalidad más compleja que aquella para la que han sido inicialmente diseñados. La plataforma permite la composición de servicios, lo cual a su vez conlleva la adición de nueva funcionalidad al sistema en tiempo real. Todo ello con bajo coste computacional, y garantizando la escalabilidad y flexibilidad del sistema.

La naturaleza distribuida y descentralizada de la arquitectura de DOHA favorece su dinamicidad. La plataforma está abierta a constantes cambios y modificaciones. Nuevos dispositivos y servicios puedan añadirse al sistema en cualquier momento.

Un servicio proporciona su funcionalidad a través de un nodo que da soporte físico a su ejecución, es decir, un elemento real del entorno sobre el que el servicio implementa su funcionalidad. El uso de una arquitectura descentralizada y distribuida reduce el cuello de botella que suele existir normalmente en las arquitecturas centralizadas, ya que gestionar infraestructuras altamente dinámicas basadas en un servidor central es un claro punto débil de su planteamiento. Además, DOHA es una plataforma abierta que favorece la integración de entidades diferentes en el mismo espacio de interacción abstracto, ya sean servicios, dispositivos físicos o aplicaciones que ofrecen su funcionalidad a través de la interacción con los servicios de la red. A su vez, los servicios DOHA abstraen la complejidad de los dispositivos físicos del entorno ubicuo. La interoperabilidad de la plataforma DOHA permite que las aplicaciones construidas sobre ella puedan ser ejecutadas en todo tipo de dispositivos, ya sean ordenadores personales o dispositivos empotrados con recursos limitados.

5.2. Arquitectura de la plataforma DOHA

La plataforma DOHA ha sido diseñada con el objetivo de desacoplar al máximo el uso de servicios, el espacio lógico de aplicación y el dominio físico de los dispositivos reales. De hecho, la funcionalidad del sistema se lleva a cabo en base a la interacción entre servicios por medio de la composición de servicios. La necesidad de interacción con el entorno requiere además interaccionar con los dispositivos reales distribuidos por el mismo. Esta interacción supone establecer un vínculo efectivo entre los servicios y los dispositivos que permita obtener información del entorno real o regular algún aspecto del mismo. Esto significa que los dispositivos son un socio imprescindible para la construcción del sistema de automatización ubicua. Aunque no se consideren como una entidad del sistema software, su conexión con los servicios será muy relevante a la hora de desempeñar la funcionalidad del sistema.

Las aplicaciones en DOHA pueden tener diferentes roles y, en consecuencia, actuar con roles diferentes en el sistema. Estos roles definen cómo se construyen los distintos tipos de servicios. Dependiendo del rol de un servicio determinado, este requerirá la implementación de un componente determinado o no. En base a su rol, podemos distinguir tres tipos de servicios específicos en DOHA: servicios generales o *General Service* que ofrecen su funcionalidad a otros servicios, servicios dispositivo o *Device Service*, y servicios consumidores puros o *Pure Consumer Service*. Los servicios dispositivo encapsulan a dispositivos físicos reales con los que pueden interactuar o a los que pueden controlar. Por otro lado, los servicios consumidores puros no ofrecen funcionalidad a otros servicios sino que actúan como punto de acceso a la funcionalidad de otros servicios, o bien poseen interfaces gráficas de acceso para los usuarios.

La heterogeneidad de los dispositivos hardware subyacentes queda simplificada gracias a la abstracción en base a servicios realizada por DOHA. La Figura 5.1 muestra las tres capas de abstracción que forman DOHA: servicios a alto nivel, nodos en el nivel virtual y dispositivos en el nivel físico. La cooperación entre servicios a más alto nivel conlleva la comunicación entre nodos a nivel virtual, así como la comunicación punto a punto entre dispositivos a más bajo nivel, que además pueden formar parte de distintas subredes. La utilización de distintos niveles de abstracción proporciona un bajo acoplamiento a nivel de servicios.

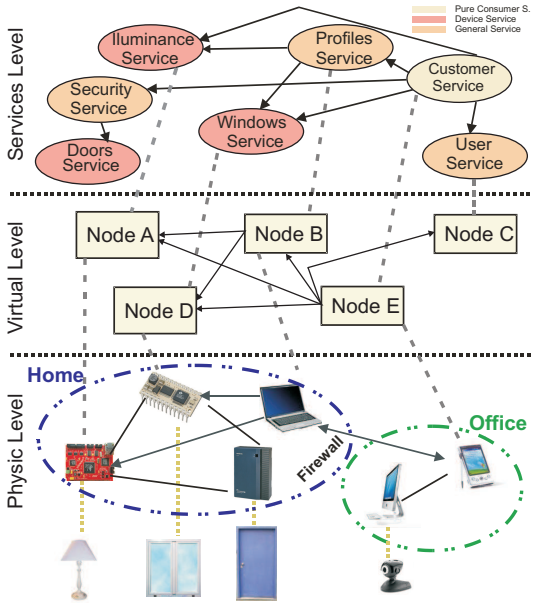


Figura 5.1: Niveles de abstracción de la plataforma DOHA

El diseño de aplicaciones en DOHA se llevará a cabo teniendo en cuenta la estructura impuesta al considerar estas tres capas de abstracción. El nivel más alto de la arquitectura es el nivel de aplicación. Los distintos servicios construidos sobre la plataforma DOHA estarán a este nivel, que encapsulará la funcionalidad ofreci-

da a los usuarios. A nivel intermedio se encuentra DOHA, que actúa como capa de interconexión o middleware entre la capa de aplicación a más alto nivel y los dispositivos físicos del entorno a más bajo nivel. Esta capa da soporte a la publicación, descubrimiento dinámico y la comunicación entre servicios, de acuerdo con los principios de SOA. Los desarrolladores DOHA también podrían incluir a este nivel nuevos servicios generales de soporte a los servicios de aplicación, cuya funcionalidad sea genérica y de utilidad para todo tipo de aplicaciones. Por ejemplo, un servicio de logger que almacene la traza de uso de los distintos servicios activos en la plataforma.

El modelo de comunicación de DOHA está basado en el esquema de publicación, descubrimiento e invocación establecido por los principios de SOA. A nivel virtual, los elementos que darán soporte al nivel de comunicación de la capa de servicios dependerán del middleware subyacente.

Por último, la capa de más bajo nivel está formada por los elementos hardware físicos del entorno ubicuo, considerando tanto los dispositivos hardware como sus sistemas operativos y las redes de comunicaciones existentes. La complejidad existente a este nivel, debida a la gran heterogeneidad de dispositivos, se simplifica gracias a las abstracciones implementadas en los niveles superiores de la plataforma. Son estas capas superiores las que facilitan el trabajo a los desarrolladores, dotándoles de elementos software de alto nivel que permiten acceder y controlar la inmensidad de elementos y dispositivos físicos que forman el mundo real. A este nivel DOHA emplea el framework *JavaES*. Se trata de una plataforma que permite abstraer los dispositivos físicos de control, facilitando el acceso al hardware y siguiendo un modelo homogéneo e independiente del tipo de dispositivo con el que se interactúe [Holgado-Terriza and Viúdez-Aivar, 2012]. Lo servicios dispositivo, con rol *Device Service*, utilizan este framework para acceder a los elementos del entorno real.

La Figura 5.2 muestra la arquitectura multicapa de cada servicio en relación con el rol del servicio concreto. Cada servicio implementa su propia funcionalidad en la capa de aplicación o *Application Layer*. Esta capa oculta al exterior cómo lleva a cabo su funcionalidad y es accesible únicamente a través de la capa de interfaz o *Interface Layer* del servicio. Los servicios consumidores puros o *Pure Consumer Services* pueden además ofrecer una interfaz gráfica para facilitar a los usuarios el acceso a la funcionalidad de la plataforma DOHA. Estos servicios no ofrecen funcionalidad, solo actúan como puerta de entrada a la funcionalidad del resto de servicios existentes. Para ello emplean la capa de interacción o *Iteraction Layer*, a través de la cual colaboran entre ellos y ofrecen una funcionalidad basada en operaciones compuestas. En el caso de los servicios dispositivo o *Device Services* también existe un componente particular, el *Device Interaction Layer*, que será el encargado de utilizar JavaES para gestionar los dispositivos físicos encapsulados por el servicio.

Como puede observarse en la Figura 5.3, esta estructura multicapa queda también reflejada en el diagrama de clases UML de la plataforma. En dicho diagrama aparece la clase Servicio, en torno a la cual giran las demás clases y relaciones existentes.

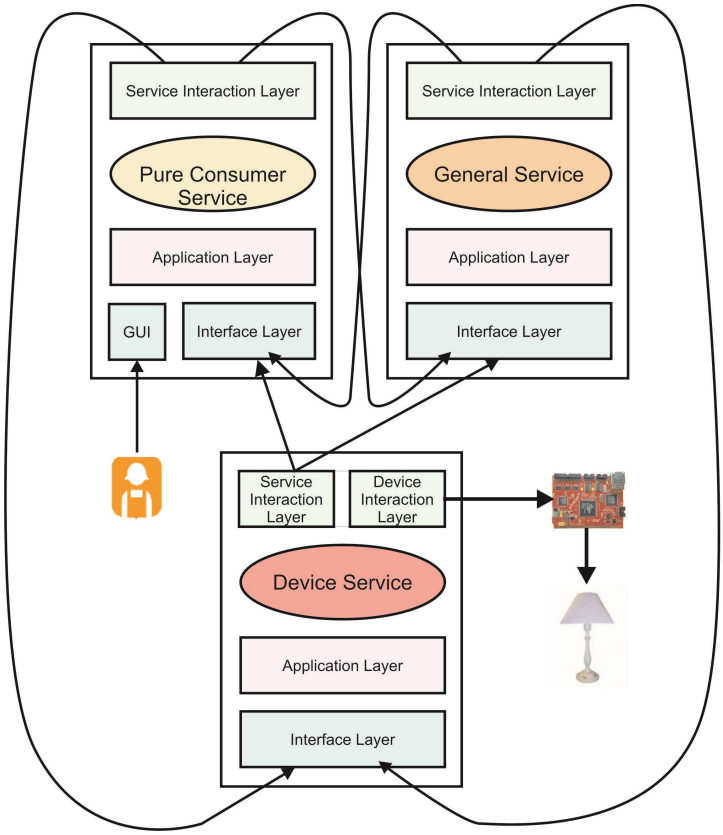


Figura 5.2: Interconexión entre las capas software de los servicios DOHA durante el proceso de comunicación

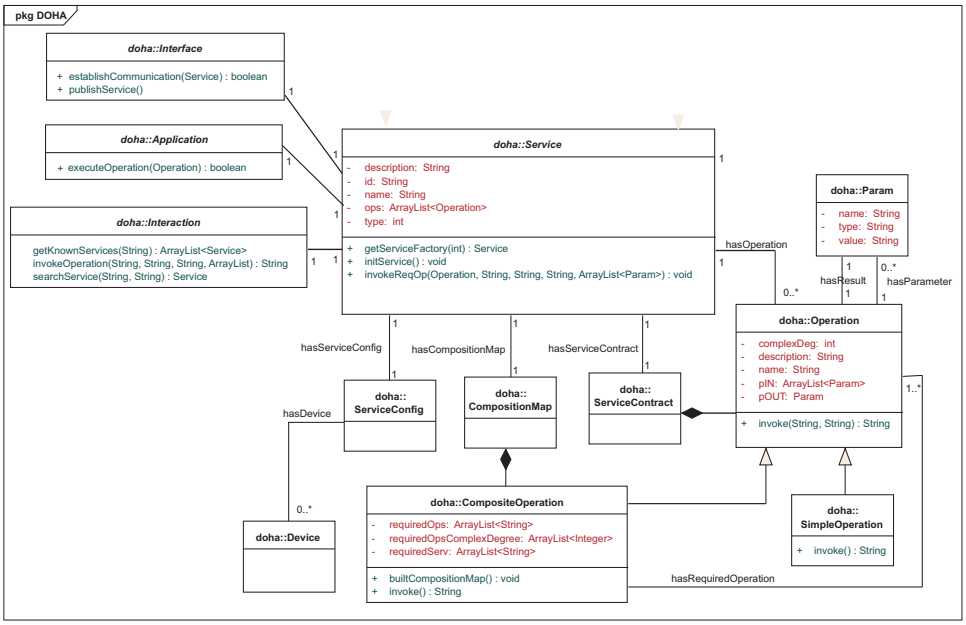


Figura 5.3: Diagrama de clases UML de DOHA

5.2.1. Definición de un Servicio DOHA

Como ya se ha comentado con anterioridad, el servicio es el elemento principal del paradigma SOA [Erl, 2005] y, por tanto, también de DOHA. Un servicio DOHA se define como un componente autónomo y autocontenido capaz de llevar a cabo actividades específicas de forma independiente, que acepta una o más peticiones y devuelve una o más respuestas a través de una interfaz bien definida. Los servicios pueden representar dispositivos hardware, recursos software, o cualquier objeto que pueda ser identificado y con el que se pueda interactuar.

En base a esta definición, un servicio DOHA requiere de una serie de elementos que lo caractericen con respeto al resto de elementos de la red. Formalmente, podemos representar un servicio DOHA como una tupla de 5 elementos, tal y como muestra la Definición 5.1, donde Id_i representa la identificación del servicio, Ps_i su propósito, Ip_i la interfaz proporcionada, Ir_i la interfaz requerida, y por último At_i un conjunto de atributos.

$$Service_i =< Id_i, Ps_i, Ip_i, Ir_i, At_i > \tag{5.1}$$

Veamos a continuación el propósito de cada uno de estos elementos como parte de la definición de servicio.

Para empezar, un servicio requiere de identificación – es decir, un nombre o identificador único que lo defina unívocamente entre todos los demás servicios. Cada servicio podrá tener distintas instancias, es decir, podrán coexistir distintas ejecuciones del servicio corriendo en distintas localizaciones, pero ofreciendo exactamente la

misma funcionalidad. El nombre o *id* debe ser único para cada instancia del servicio por lo que se ha decidido basar la identificación de servicios DOHA en URNs, acrónimo de Uniform Resource Name, un tipo de URI que permite definir identificadores de recursos independientemente de su localización.

En segundo lugar, el servicio debe de especificar cuál es su propósito – es decir, cuál es su funcionalidad y cuáles son sus responsabilidades. La funcionalidad del servicio puede representarse en términos de requisitos funcionales en lenguaje natural. Sin embargo, hemos optado por una definición más formal en base al conjunto de operaciones que el servicio ofrece al resto de servicios de la plataforma. Estas operaciones son públicas y accesibles para cualquier otro servicio que las requiera.

Un servicio puede asumir distintos roles. Dependiendo de cuál sea su papel en la interacción entre varios servicios, un servicio puede actuar como proveedor o como cliente, *service provider* o *service consumer*. A su vez, un servicio puede llevar a cabo dos tipos de operaciones, simples o compuestas. Mientras que una operación simple puede ser llevada a cabo completamente por el servicio, cuando se trata de ejecutar una operación compuesta el servicio actuará como consumidor de al menos un servicio proveedor. Por ello, en la definición de servicio se han considerado también las interfaces proporcionada y requerida. Dichas interfaces deben reflejar la capacidad de interacción del servicio con el resto de servicios de la red, distinguiendo cuando dicho servicio actúa como cliente o como proveedor. La interfaz proporcionada, *Provided Interface*, muestra las distintas funciones u operaciones que el servicio ofrece y que son accesibles por otros servicios. Es decir, será la interfaz utilizada cuando el servicio actúe con rol proveedor. Bajo este rol, el servicio es responsable de recibir mensajes de otros servicios, que actuarán como consumidores, y gestionar la ejecución de las operaciones invocadas, dando una respuesta a cada uno de ellos cuando sea necesario. Por el contrario, la interfaz requerida, *Required Interface*, incluye las funciones u operaciones que el servicio invocará en otros servicios de la red actuando como consumidor. Por lo tanto, ambas interfaces, proporcionada y requerida, pueden representarse como un conjunto de operaciones que el servicio debe ejecutar localmente, proporcionada, o en otros servicios, requerida.

Por último, la definición de servicio incluye un conjunto de propiedades que influyen sobre su ejecución. Estas propiedades son atributos del servicio que deben conocer los servicios consumidores antes de utilizarlo. En general, es necesario proporcionar información general sobre el contenido del servicio (datos, algoritmos o recursos software/hardware), además propiedades de configuración que puedan influir sobre el rendimiento del servicio. Algunas de estas propiedades son definidas específicamente por las instancias del servicio mientras que otras pueden ser definidas a alto nivel y ser comunes para todas las instancias del mismo.

En base a la definición formal de servicio podemos representar su estructura interna a modo de "anatomía del servicio". La Figura 5.4 muestra la anatomía de un servicio DOHA organizada por componentes constituidos en base a una estructura multicapa. La estructura software multicapa permite desacoplar las tareas del servicio en base a componentes conexos. La conexión entre dichos componentes da lugar a la anatomía del servicio completa.

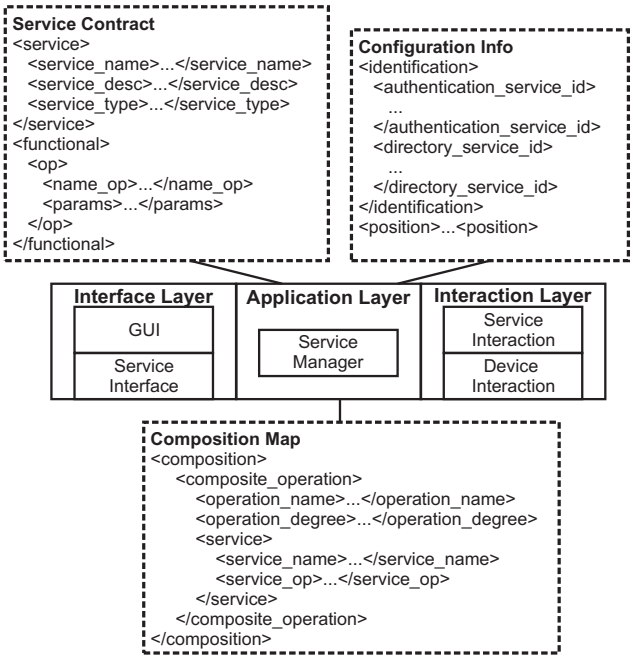


Figura 5.4: Anatomía de un servicio DOHA

La capa de interfaz o *Interface Layer* es el punto de acceso a los servicios y establece cómo estos pueden comunicarse entre sí. Es decir, permite el acceso a los servicios desde cualquier otro servicio del sistema. Los servicios pueden tener una interfaz gráfica, la cual está representada por el componente GUI, acrónimo de *Graphical User Interface*. No es obligatorio que la capa de interfaz contenga un componente GUI en todos los servicios. Es más, este componente será implementado únicamente por aquellos servicios que deban proporcionar una interfaz gráfica a los usuarios. Por ejemplo, podría ser el caso de un servicio que actúa exclusivamente como consumidor y que no ofrece funcionalidad al resto de servicios de la red. Su función en el sistema sería la de ofrecer a los usuarios una interacción amigable con el resto de servicios existentes. Sin embargo, aunque el componente GUI no es obligatorio, todos los servicios DOHA poseen el componente *Service Interface*. Este componente establece el modo de comunicación entre los servicios y qué operaciones del servicio se pueden invocar desde otros servicios consumidores. Esta información vendrá determinada por las operaciones indicadas en la interfaz proporcionada de la definición del servicio S_i, representada por Ip_i.

La capa de aplicación o *Application Layer* abstrae la funcionalidad de los servicios. El elemento principal de esta capa es el componente denominado *Service Manager*, que actúa como gestor del servicio. Este componente ha sido definido en base al principio de "stateless" o "sin estado" establecido por el paradigma SOA. Dicho principio evita que el uso de información de estado de los servicios, o la dependencia de dicha información, pueda afectar a la disponibilidad o escalabilidad de

los mismos. Para evitarlo, la parte dependiente del estado en los servicios DOHA queda bien limitada y encapsulada por la funcionalidad de cada servicio implementada en el *Service Manager*, siendo esta imperceptible desde el exterior. El *Service Manager* gestiona las peticiones recibidas en el servicio, implementando las operaciones que satisfacen la funcionalidad de los servicios. Cuando un servicio DOHA ejecuta su *Service Manager* para efectuar una operación determinada está cambiando su estado, aunque esto no sea visible desde el exterior. El servicio actúa como una caja negra. Aunque esté ejecutando una operación determinada a través de su *Service Manager*, puede continuar recibiendo peticiones de funcionalidad a través de la capa de interfaz o *Interface Layer*.

Por último, la capa de interacción o *Interaction Layer* contiene la lógica necesaria para llevar a cabo la comunicación y colaboración con otros servicios. El componente de esta capa denominado *Service Interaction* permite a los servicios DOHA actuar como clientes de otros servicios, permitiendo establecer una colaboración entre servicios a través de la composición de sus operaciones. La colaboración con otros servicios vendrá determinada por la interfaz requerida de la definición del servicio, Ir_i. Por otro lado, el componente *Device Interaction* de esta capa permite acceder a los dispositivos físicos del entorno. Dicho componente estará presente en todos los servicios DOHA que encapsulen un dispositivo existente en el entorno, por ejemplo, un sensor de temperatura o un regulador de luminosidad. Los dispositivos hardware encapsulados por el servicio S_i vendrán determinados por el contenido del elemento At_i de la definición del servicio.

Los beneficios aportados por la selección de esta y no otra arquitectura radican en la reusabilidad de los distintos componentes definidos, su adaptabilidad y su modularidad. Cualquier cambio en alguno de los componentes no afectaría a los demás. Es más, se podría reemplazar cualquiera de los componentes de forma transparente, siempre que se preserve la interfaz del servicio. Además, la estructura basada en componentes permite gestionar el flujo de ejecución de los servicios en cada paso de su ejecución.

Cada servicio especificará su definición y la descripción de su funcionalidad, su capacidad de colaboración y sus propiedades de configuración a través de distintos elementos adicionales. Estos elementos están también representados en la Figura 5.4 que muestra la anatomía del servicio. Son el contrato de servicio, *Service Contract*, el mapa de composición de servicios, *Service Composition Map*, y el archivo de configuración, *Service Configuration*.

A partir de la Definición 5.1 de servicio y teniendo en cuenta los conceptos de contrato, mapa y configuración propios de cada servicio, podemos realizar una definición adicional de servicio que encapsule estos tres elementos de su anatomía, tal y como se muestra en la Definición 5.2.

$$Service\_Anatomy_i =< contract(S_i), map(S_i), config(S_i) > \qquad (5.2)$$

La especificación de estos elementos, contrato, mapa y configuración, se almacena en documentos XML conforme al esquema XSD. A continuación analizaremos las

peculiaridades de cada uno de ellos individualmente.

5.2.1.1. Contrato de Servicio

El contrato de servicio o *Service Contract* describe el propósito del servicio definido por $Ps(S_i)$. El conjunto de operaciones es equivalente al contenido de la interfaz proporcionada por el servicio y especificada como $Ip(S_i)$ en la Definición 5.1. La descripción del servicio es pública y accesible desde cualquier servicio consumidor que desee interactuar con el servicio.

Un contrato de servicio DOHA está estructurado en dos partes bien diferenciadas: información general del servicio e información funcional. La información general del servicio viene determinada por su nombre y descripción. La sección de información funcional viene determinada por los requisitos funcionales del servicio y está formada por la lista de operaciones o métodos que pueden ser invocados por otros servicios que actúen como aplicaciones cliente. Cada operación está determinada por su nombre y los parámetros de entrada que requiere. Esta información oculta la lógica del servicio y la implementación que se realiza en el *Service Manager*. El contrato de servicio debe ser lo más descriptivo posible hacia el exterior manteniendo oculta toda la información sensible, como localización o estado de ejecución, lo que a su vez favorece las propiedades de encapsulación y abstracción de la plataforma.

El contrato de servicio es su carta de presentación. Este proporciona información sobre la funcionalidad del servicio, pero no cómo se lleva a cabo. El contrato de servicio es único para cada servicio. Pueden existir numerosas instancias de un mismo servicio pero todas tendrán el mismo contrato de servicio asociado. Esto implica que pueden existir numerosas ejecuciones del mismo servicio que ofrecen exactamente la misma funcionalidad al resto. La distinción entre ellos vendrá determinada por los requisitos no funcionales de cada una de las instancias, que veremos cómo considerar un poco más adelante.

En el fragmento de Código 5.1 se presenta un esquema de un contrato de servicio. En él podemos ver la especificación del servicio *LightSensorService*. Este servicio interactúa con un dispositivo físico y gestiona uno o varios sensores de luz. Las operaciones proporcionadas por este servicio son *getDevices()*, que devuelve una lista de identificadores con los sensores de luz gestionados por el servicio, y *getLightValue(id)*, que devuelve la intensidad de luz capturada por el sensor de luz identificado por el valor pasado como parámetro a la operación.

Código 5.1: Estructura XML de un contrato de servicio

```
1 <service>
2    <service_name>LightSensorService</service_name>
3    <service_desc>Manage light sensors</service_desc>
4    <service_type>Device Service</service_type>
5 </service>
6 <functional>
7       <op>
8            <name_op>getDevices</name_op>
9            <desc_op>See the devices managed</desc_op>
```

```
10            <params></params>
11            <return>List of identifiers of the light sensors</return>
12        </op>
13        <op>
14            <name_op>getLightValue</name_op>
15            <desc_op>Query in a sensor light</desc_op>
16            <params>
17                <param>
18                    <name_param>LightSensorID</name_param>
19                    <type_param>int</type_param>
20                </param>
21            </params>
22            <return>double</return>
23    </op>
24</functional>
```

5.2.1.2. Mapa de Composición

El mapa de composición del servicio o *Service Composition Map* define las operaciones compuestas del servicio. Cada operación compuesta posee su propio mapa de composición, por lo que el documento del mapa será un conjunto de mapas de operaciones compuestas. El conjunto de operaciones requeridas del mapa de composición será equivalente a la interfaz requerida $Ir(S_i)$ de la Definición 5.1. Este documento es privado y accesible únicamente por el propio servicio.

El mapa de composición de servicios determina la gestión del comportamiento colaborativo de los servicios. Dicho mapa de composición establece las relaciones entre los servicios y qué operaciones van a ejecutarse colaborando con otros servicios. Para ello debemos distinguir dos posibles tipos de operaciones en DOHA: operaciones simples y operaciones compuestas. En una operación simple el servicio es autosuficiente y no requiere colaborar con ningún otro servicio del entorno. En cambio, en una operación compuesta el servicio interaccionará con otros servicios del entorno invocando sus operaciones. Por ello, una operación compuesta está formada por una o varias operaciones de otros servicios. Esta información es la que está contenida en el mapa de composición, es privada y accesible únicamente por el propio servicio. Solamente él conoce con qué otros servicios interactúa para llevar a cabo una operación compuesta. El mapa de composición es el mismo para los servicios con igual contrato de servicio. Es decir, existe un mapa de composición para cada contrato de servicio con operaciones compuestas.

Una vez que el servicio está ejecutándose, este puede interactuar con otros servicios del entorno para llevar a cabo sus operaciones compuestas. El servicio es consciente de la existencia de los servicios requeridos en sus operaciones compuestas, pues estos son necesarios para llevar a cabo su funcionalidad. La comunicación entre servicios en base a la información existente en el mapa de composición permite llevar a cabo una interacción entre servicios autónoma, sin la intervención de los usuarios.

El Código 5.2 muestra la estructura XML de un mapa de composición propio de

un servicio de control de luminosidad. En cada una de las operaciones compuestas de este documento aparecen tantas etiquetas $< service\_name >$ como servicios con los que dicha operación debe interactuar para completarse. En este caso el mapa de composición consta de tres operaciones compuestas. La primera operación compuesta *getLight()* requiere la utilización de una sola operación, por lo que colabora con un único servicio, se trata de la operación *getLightValue(value)* del servicio *LightSensorService*. La operación compuesta *setLight()* requiere la utilización de la operación *setLightValue(value)* del servicio *LightRegulatorService*. Y, por último, la operación *turnOffLight()* también interactúa con el servicio controlador de luz *LightRegulatorService* e invoca a su operación *turnOff()*.

Código 5.2: Estructura XML de un mapa de composición

```
<composition>
      <composite_operation>
        <operation_name>getLight</operation_name>
        <operation_degree>1</operation_degree>
        <service>
            <service_name>LightSensorService</service_name>
            <service_operation>getLightValue</service_operation>
        </service>
      </composite_operation>
      <composite_operation>
        <operation_name>setLight</operation_name>
        <operation_degree>1</operation_degree>
        <service>
            <service_name>LightRegulatorService</service_name>
            <service_operation>setLightValue</service_operation>
        </service>
      </composite_operation>
      <composite_operation>
        <operation_name>turnOffLight</operation_name>
        <operation_degree>1</operation_degree>
        <service>
            <service_name>LightRegulatorService</service_name>
            <service_operation>turnOff</service_operation>
        </service>
      </composite_operation>
</composition>
```

5.2.1.3. Información de Configuración

La configuración del servicio o *Service Configuration* especifica los parámetros que influyen sobre su ejecución así como atributos propios del nodo físico que lo aloja. Por ejemplo, la localización física del nodo donde el servicio se ejecuta, el tipo de conectividad de dicho nodo, o propiedades QoS propias de su ejecución, entre otras. Por lo tanto, esta información puede ser diferente para distintas instancias del mismo servicio. Las propiedades de configuración incluidas en este documento son equivalentes a los elementos contenidos en el conjunto $At(S_i)$ incluido en la Definición 5.1. En algunos casos, además, el archivo de configuración puede contener

información adicional como documentos pdf o imágenes que pueden ser descargadas a través de una URL.

En el fragmento de código 5.3 podemos ver un ejemplo de la estructura general de un archivo de configuración.

Código 5.3: Estructura XML de un archivo de configuración

```
<identification>
  <authentication_service_id>
  ...
  </authentication_service_id>
  <directory_service_id>
  ...
  </directory_service_id>
</identification>
<hardware>
  <sensor>...</sensor>
  <regulator>...</regulator>
</hardware>
<qos>
  <wcet>...</wcet>
  <bwcet>...</bwcet>
  <throughput>...<throughput>
  <availability>...<availability>
  <reliability>...<reliability>
</qos>
```

5.3. Interacción con el mundo real

La interacción con el hardware es uno de los aspectos más conflictivos en los sistemas de automatización. Dicha interacción determina cómo los servicios pueden acceder al mundo real. El entorno que compone los espacios ubicuos suele estar plagado de dispositivos físicos muy diferentes entre sí. Estos dispositivos pueden clasificarse, a priori y de modo muy general, en dos grandes grupos: dispositivos autónomos y dispositivos interconectados a través de una red.

Por dispositivos autónomos podemos entender que son todos aquellos elementos individuales limitados para su interacción directa con el usuario. Como por ejemplo, un panel rígido. En estos casos la interacción con los dispositivos no es posible, ya que no existe una interfaz de programación o un protocolo de comunicación que podamos utilizar para invocar a un servicio concreto que se ejecute sobre los mismos. Por lo tanto, para incluir este tipo de dispositivos en el sistema será necesario establecer con ellos una conexión por cable o bien conectarlos directamente a un pin de entrada/salida analógico o digital de uno de los dispositivos empotrados conectados a la red y que si formen parte del sistema.

El segundo conjunto de dispositivos se refiere a todos aquellos que están conectados a la red del entorno. En este caso, la capacidad de los dispositivos para formar parte de la red del entorno ubicuo estará directamente ligada con el protocolo de comunicación de red que utilicen. Por ejemplo, TCP-IP, X-10, Konnex o Lonworks.

Generalmente, para implementar una red de dispositivos de automatización en el entorno del hogar o Home Network System (HNS) se opta por la utilización de arquitecturas centralizadas basadas en un servidor central del que dependen el resto de elementos del sistema [Nakamura et al., 2008]. A este tipo de soluciones se las denomina *Server Centralized Architecture*.

La plataforma DOHA lleva a cabo la interacción con el hardware a través de servicios específicos denominados *Device Services*. Con este enfoque se propone un procedimiento para el acceso al hardware inusual que difiere de los ya comentados servidores centralizados. Con los servicios de tipo dispositivo de DOHA lo que se propone es ascender el nivel de dispositivo hardware hasta el nivel de servicios. Es decir, encapsular en la propia lógica de cada servicio las peculiaridades de los dispositivos hardware del mundo real que este abstrae. Para llevar esto a cabo se incorpora a la lógica de control de los servicios dispositivo un subsistema con la responsabilidad específica de gestionar y manejar los dispositivos físicos que encapsula y, por lo tanto, llevar a cabo la interacción con el entorno. Desde el punto de vista estructural de DOHA, dicho subsistema se encuentra en la capa de interacción de la arquitectura, como puede verse en la Figura 5.4 que muestra la anatomía de un servicio DOHA.

La metodología de interacción con el hardware propuesta en DOHA conlleva ventajas considerables en el ámbito de la computación ubicua. Por ejemplo, se reduce el salto semántico entre el nivel de servicios y el nivel de dispositivos, facilitando la coordinación y sincronización entre ambos. El subsistema encargado de gestionar la interacción con el hardware en los servicios, al que se ha denominado *Device Manager*, trabaja en el contexto del servicio, no en el del dispositivo, tal y como podemos observar en la Figura 5.5.

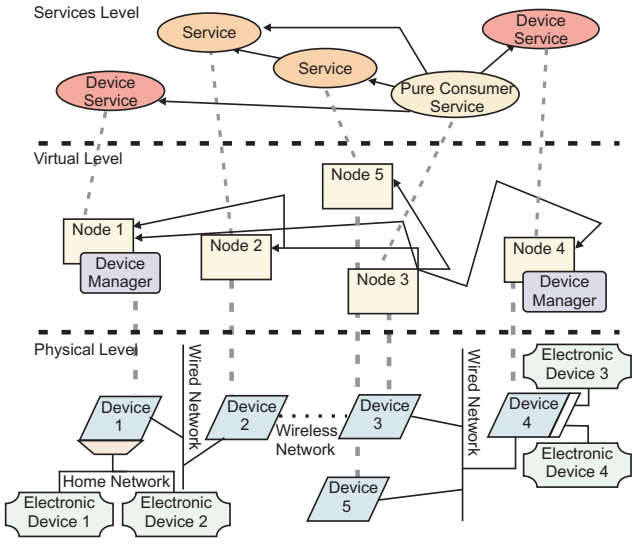


Figura 5.5: Niveles de abstracción de la arquitectura desde el punto de vista de los *Device Services* en DOHA

La capa de interacción del servicio ha sido implementada mediante la combinación de dos infraestructuras software, JavaES y JDOMO [Holgado-Terriza and Viúdez-Aivar, 2012]. JavaES (Java for Embedded Systems) es un framework basado en Java que proporciona una plataforma adaptable, flexible y robusta para facilitar el desarrollo de aplicaciones Java teniendo en cuenta los recursos computacionales limitados de los sistemas empotrados. Además, JavaES abstrae la complejidad y heterogeneidad existente entre los dispositivos hardware que podemos encontrar en un entorno ubicuo. JavaES proporciona mecanismos de alto nivel que facilitan la utilización de los distintos dispositivos hardware, por ejemplo puertos de entrada/salida, relojes y contadores, entre otros, independientemente de la arquitectura propia del sistema empotrado. Las características de todos los microcontroladores soportados actualmente por JavaES puede consultarse en [Holgado-Terriza and Viúdez-Aivar, 2009].

A través de una capa de abstracción que facilita la independencia del entorno físico concreto, JavaES trabaja con plataformas virtualizadas tanto de recursos hardware (capacidad de procesamiento, memoria o periféricos) como infraestructuras software (sistema operativo, máquina virtual o drivers). De este modo, es posible desacoplar el desarrollo de aplicaciones con respecto a su despliegue, preservando la portabilidad del código. Esto permite a los desarrolladores controlar los recursos hardware de la plataforma, pudiendo modificar, actualizar o extender las posibilidades del sistema hardware en base a las necesidades de la aplicación.

Sobre JavaES se utiliza JDOMO (Java Home-Automation). De forma similar a JavaES, JDOMO proporciona un conjunto de clases y paquetes que facilitan el control, acceso y gestión de los dispositivos físicos a alto nivel.

El conjunto formado por las infraestructuras JavaES-JDOMO facilita que la interacción con el hardware se lleve a cabo a través del concepto de objeto virtual. Un objeto virtual no es más que una abstracción lógica del periférico de entrada/-salida que representa. Por ejemplo, un objeto virtual de un sensor de temperatura representa a un sensor de temperatura físico conectado a un pin GPIO. Desde el punto de vista del desarrollador, se trabaja con un objeto virtual como cualquier otro, es decir, una instancia de un objeto. Con la peculiaridad de que los métodos y funciones que este proporciona repercuten en el entorno de forma directa gracias a la vinculación establecida con el dispositivo físico real, tal y como podemos observar en la Figura 5.6.

La utilización de objetos virtuales facilita desacoplar la abstracción software de dispositivo con respecto al dispositivo físico real en dos niveles. En el primer nivel, el objeto virtual contiene una interfaz de alto nivel que proporciona todas las funciones necesarias para acceder al dispositivo físico. Por ejemplo, la lectura de la temperatura en un sensor de temperatura a alto nivel implica la lectura del nivel de voltaje en el sensor, ya sea directamente en el sensor o a través de un bus de comunicación, y su conversión a un valor mediante un ADC. En un segundo nivel, el objeto virtual encapsula el estado real del dispositivo físico. Esto significa que se puede desacoplar completamente la lectura real del estado del dispositivo con respecto a la interfaz que ofrece el objeto virtual para consultar su estado. Por lo tanto, el objeto virtual puede

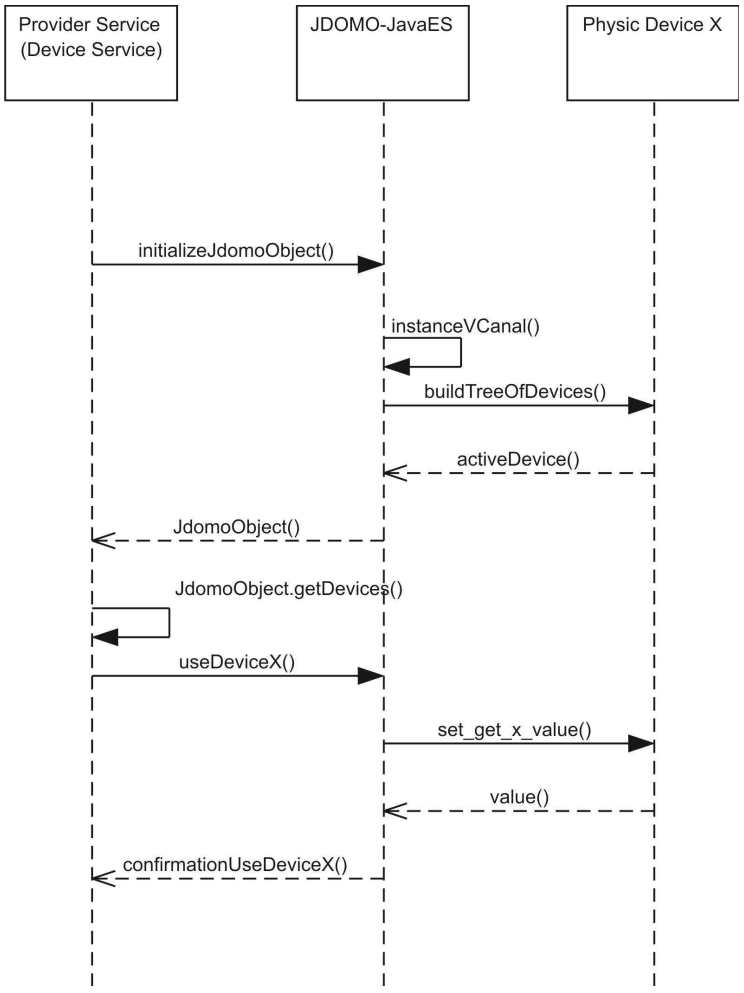


Figura 5.6: Interacción entre un *Device Service*, JDOMO-JavaES y un dispositivo físico

separar la lectura sobre el pin del dispositivo físico con respecto a la actualización del estado del objeto virtual. De este modo, la invocación del objeto virtual supondría una actualización del mismo, pero no una lectura inmediata del dispositivo físico.

El uso de JavaES con respecto al conexionado de dispositivos físicos aporta también flexibilidad a DOHA. La reconfiguración de los dispositivos físicos ligados a los objetos virtuales puede realizarse de forma estática o dinámica. Para ello existe un mecanismo denominado "lazy initializing". Dicho mecanismo facilita la ejecución de servicios DOHA sobre los objetos virtuales y que estos, más tarde y una vez estén ya en ejecución los servicios, sean linkados con el dispositivo físico real.

Los tipos de objetos virtuales que pueden ser gestionados por JavaES-JDOMO son varios y se organizan en base a una estructuración jerárquica que puede observarse en la Figura 5.7. Los elementos más relevantes de dicha jerarquía son:

- Objetos virtuales de tipo sensor, *Sensor*, que representan dispositivos de captura de información el entorno.

- Objetos virtuales de tipo actuador, *Actuator*, que representan dispositivos que pueden modificar el entorno llevando a cabo acciones sobre el mismo.

- Objetos virtuales de tipo controlador o regulador, *ComplexDevice*, que representan dispositivos con un comportamiento determinado y cerrado que no puede ser modificado, tan solo pueden monitorizarse o utilizarse como controladores o reguladores.

5.4. Implementación de DOHA

La plataforma que proporciona DOHA se ha diseñado para dar soporte al desarrollo de aplicaciones de computación ubicua conforme a los principios de SOA. La implementación de DOHA está sustentada a través de un middleware de comunicaciones que proporciona un modelo de comunicación que establece los mecanismos utilizados para la interacción entre las aplicaciones, las redes y el software de sistemas.

En nuestro caso, la implementación de DOHA es agnóstica e independiente de la tecnología utilizada para la comunicación utilizando como referencia los principios de SOA. Esto nos permite proporcionar un modelo de programación sólido y con el suficiente nivel de abstracción para dar soporte a la implementación de los servicios como elementos principales de DOHA. Ahora bien, para conseguir que el modelo sea aplicable sobre diferentes tecnologías distribuidas necesitamos poder adaptarlo al paradigma de comunicación concreto a través de la utilización de un middleware. En nuestro caso, se han seleccionado dos paradigmas de comunicación bien diferentes: DPWS y JXTA. DPWS proporciona un paradigma de comunicación cliente-servidor basada en servicios web por ser una tecnología que se adapta de forma natural al

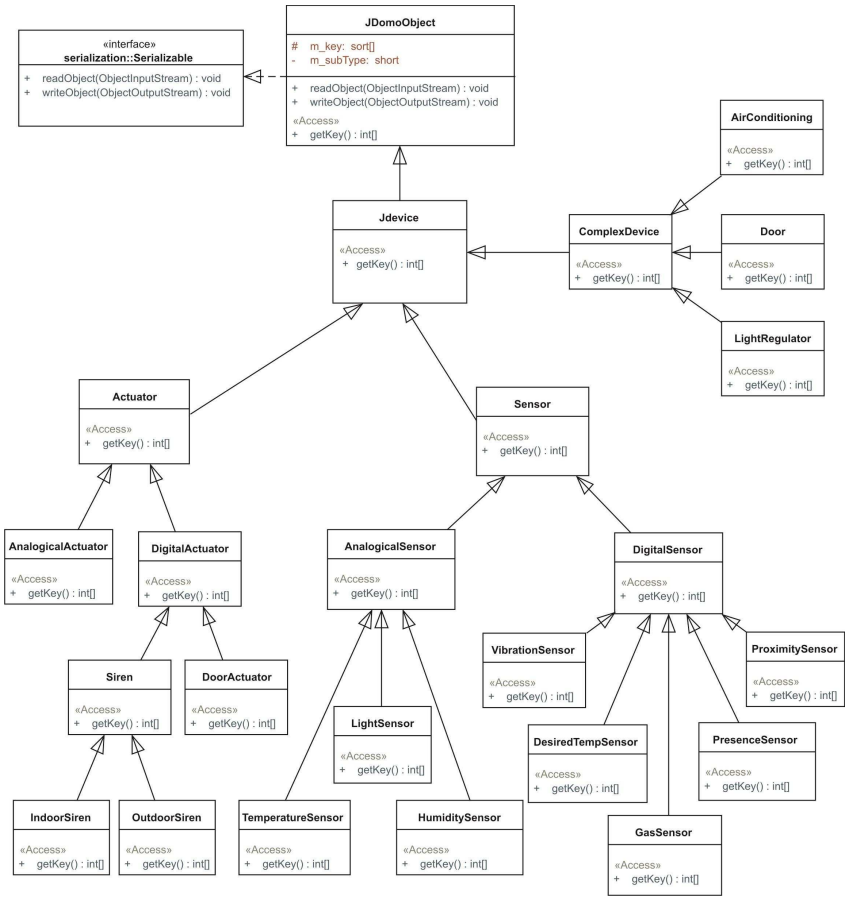


Figura 5.7: Diagrama de clases de los objetos virtuales contenidos en JDOMO-JavaES

modelo de DOHA; en este caso hemos utilizado para la implementación del middleware *Web Services for Devices* (WS4D). En cambio, JXTA es una tecnología que utiliza un paradigma de comunicación basada en P2P.

Por tanto, la plataforma de servicios DOHA se convierte así en una especie de "middleware de middlewares". En la Figura 5.8 se puede observar la arquitectura general de DOHA con respecto a estos middleware de comunicación, así como a la red, dispositivos o sistemas operativos que estos ejecutan.

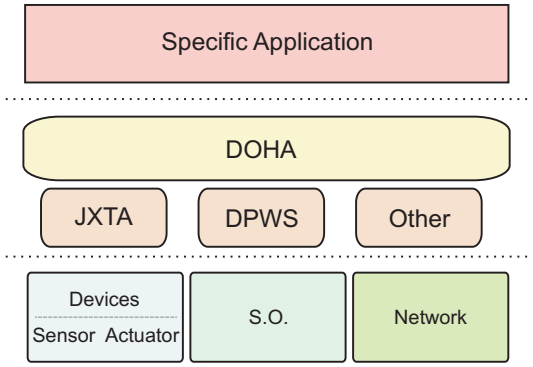


Figura 5.8: Arquitectura de la plataforma de servicios DOHA con respecto al middleware de comunicación subyacente

La plataforma de servicios DOHA modela las aplicaciones a nivel de servicios, tal y como se comentó al presentar el diagrama UML que mostraba su arquitectura (Figura 5.3). Sin embargo, a nivel de implementación se debe distinguir entre los conceptos de JXTA o DPWS para llevar a cabo la comunicación, tal y como muestra la Figura 5.9. La utilización del patrón factoría permite crear y utilizar los objetos Servicio del mismo modo e independientemente del middleware de comunicación subyacente. Que el Servicio utilice a bajo nivel JXTA o DPWS no repercutirá en cómo el programador emplee la plataforma de servicios. Sin embargo, las peculiaridades de cada middleware sobre el que se sustenta la plataforma de servicios sí que influirán a nivel de ejecución. Veremos a continuación las peculiaridades de cada uno.

En el caso de JXTA el elemento principal que da soporte virtual a los servicios de DOHA es el peer JXTA. Cada peer utiliza los mecanismos y protocolos de JXTA según corresponda, por ejemplo, implementa pipes JXTA para comunicarse con otros peers. El *JXTA Pipe Service* es utilizado por los peers para crear un canal de comunicación punto a punto que define el pipe y establecer la comunicación entre ellos. El *JXTA Discovery Service* y el *JXTA Advertisement Service* son usados por los peers para crear y publicar anuncios en la red. Adicionalmente, el *Rendezvous Service* permite a los peers de distintas subredes descubrirse y comunicarse. Estos y otros detalles se analizarán más detenidamente en la sección 5.4.1.

Cuando el middleware de comunicación de soporte a DOHA es DPWS, los elementos principales son *Device*, *Hosted and Hosting Services* y *Message*. El *DPWS*

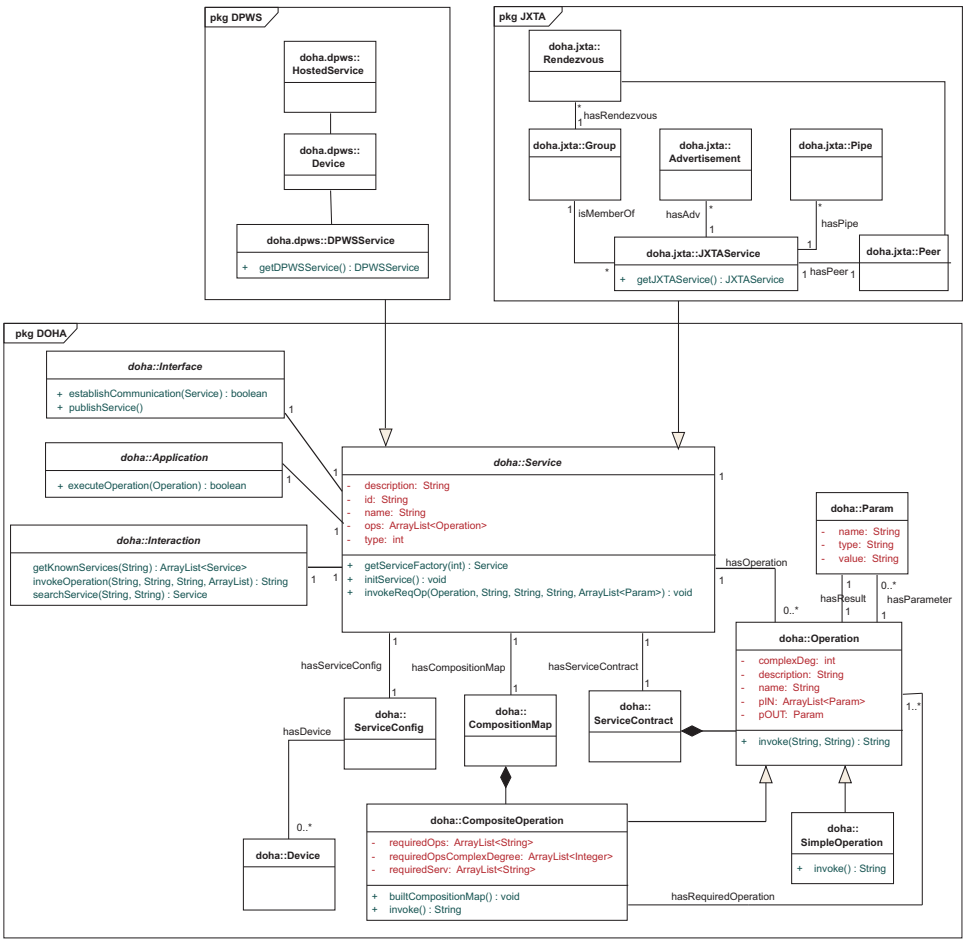


Figura 5.9: Diagrama de clases UML de DOHA considerando las implementaciones JXTA y DPWS

Device es el encargado de alojar los distintos servicios que se pueden invocar desde fuera, actuando como *Hosting Service*. Los servicios ofrecidos se denominan *Hosted Services*, pues se considera que están alojados por el *DPWS Device* al que pertenecen. Estos y otros detalles se analizarán más detenidamente en la sección 5.4.2.

5.4.1. DOHA sobre JXTA

El framework JXTA es una plataforma P2P que permite que cualquier dispositivo conectado a la red pueda colaborar y comunicarse como un peer. Fue inicialmente creado por *Sun Microsystems* y proporciona los protocolos, servicios, modelos de programación y mecanismos necesarios para desarrollar aplicaciones P2P de forma sencilla. El término JXTA es un acrónimo de *juxtapose*, y refleja el objetivo de la plataforma de romper con el clásico paradigma cliente-servidor utilizado en numerosos desarrollos de computación distribuida [Traversat et al., 2003].

Una evolución de JXTA, ligada al uso de dispositivos empotrados con escasos recursos, es JXME. Se trata de una versión optimizada de JXTA para dispositivos que ejecuten Java 2 Microedition (J2ME), tanto para la configuración CDC como CLDC. Esta versión de JXTA ofrece prácticamente la totalidad de la funcionalidad de la versión original, pero adaptada a las restricciones, tanto de memoria como de procesamiento, características de los dispositivos empotrados [Knudsen, 2002]. La arquitectura de JXME relega la mayor parte del procesamiento pesado que pueda sobrecargar a este tipo de dispositivos a otros peers JXTA que se ejecuten en nodos con mayor capacidad de cómputo, como por ejemplo los *Relay Peers* [Kawulok et al., 2005].

Existen diversos trabajos de investigación que también han empleado JXTA como middleware de comunicaciones base para el diseño de plataformas colaborativas basadas en servicios y que están relacionados con el trabajo presentado. Por ejemplo, *JXTA-Overlay* es una plataforma P2P basada en JXTA diseñada por Barolli et al. [Barolli and Xhafa, 2011] y testeada en dispositivos de cómputo potentes. En la literatura especializada también se ha trabajado relacionando los web-services y la comunicación P2P en entornos distribuidos. Es el caso de Gharzouli et al. [Gharzouli and Boufaida, 2010], que presenta una arquitectura distribuida para servicios web semánticos basada en una red P2P, pero centralizando el uso de determinados recursos. Veremos a continuación como el diseño de DOHA sobre JXTA, además de estar orientado a su utilización en dispositivos empotrados, respetando sus características, carece de elementos totalmente centralizados, respetando el modelo P2P. En adelante, denominaremos a esta implementación DOHA-JXTA.

Elementos básicos de JXTA. Los principales elementos de JXTA son los advertisements, peers, peer groups, pipes y protocolos. El concepto de anuncio o *advertisement* identifica cualquier tipo de informacón que se transmite en la red. Los *advertisements* son documentos XML que se emplean también para describir todos los recursos de la red JXTA, como peers, grupos, pipes o servicios. La red de dispositivos físicos del espacio ubicuo queda representada virtualmente a través de la

interconexión de los *peers* JXTA. Un *peer* es cualquier nodo o dispositivo conectado a la red que implemente uno o más protocolos JXTA y que se ejecuta de forma independiente y asíncrona con respecto al resto de peers. En la implementación de DOHA sobre JXTA, el concepto de servicio DOHA recae sobre el de peer JXTA.

Descubrimiento de servicios. Los servicios DOHA publican sus anuncios en la red para facilitar que otros servicios puedan descubrirlos. La relación publicación y descubrimiento requiere de mecanismos específicos para poder llevarse a cabo. Asimismo, determinados servicios que interactúen entre sí no tienen por qué estar en la misma subred, por lo que la utilización de mecanismos que permitan facilitar la comunicación en estos casos es también relevante. En JXTA podemos distinguir entre los peers generales, o *edge peers*, y otro tipo de peers específicos especialmente diseñados para ser usados en estos casos, son los *Relay Peer* y *Rendezvous Peer* [Meshkova et al., 2008]. La utilización de estos peers especiales permite llevar a cabo el descubrimiento de anuncios entre peers, incluso estando estos localizados en distintas subredes.

El *Rendezvous Peer*, mantiene una cache de anuncios, puede reenviar solicitudes de descubrimiento a otros peers y poner a disposición los recursos compartidos por los peers, incluso más allá de la red local. Los *Relay Peer* son de especial utilidad cuando la red de peers está formada por distintas subredes físicas o lógicas con NAT o cortafuegos. Estos peers especiales almacenan la localización de los peers conocidos y hacen el papel de nexo de unión entre distintas subredes.

Cuando un servicio DOHA descubre un nuevo anuncio en la red, este es almacenado en la cache local de su peer. Si el servicio además está conectado a un servicio de directorio, que sería un nodo del tipo Rendezvous Peer, podrá además realizar la búsqueda y descubrimiento de anuncios en otras redes remotas distintas de su propia red local. Los peer de tipo Rendezvous formarán una coalición que compondrá el servicio de directorio de DOHA. Estos peers especiales tendrán la responsabilidad de coordinar todos los peers de la red JXTA y propagar en las distintas redes remotas sus anuncios y mensajes. Por otro lado, si alguna de las redes posee protección por firewall o NAT será necesario hacer uso de un peer de tipo Relay.

Comunicación entre servicios. Los servicios DOHA publican y realizan descubrimiento de anuncios o *advertisement* utilizando el *JXTA Peer Discovery Protocol*. Las solicitudes se llevan a cabo a través de pipes de comunicación haciendo uso de los protocolos *JXTA Pipe Biding Protocol* y *JXTA Peer Endpoint Protocol*. JXTA proporciona tres mecanismos de comunicación entre peers básicos, cada uno de ellos con un nivel de abstracción diferente. A más bajo nivel se utiliza el *endpoint service*, sobre el que está el *pipe service*, y finalmente, a más alto nivel, estarían los *JXTA sockets* [Antoniu et al., 2005]. En la implementación de DOHA realizada con JXTA se ha utilizado el nivel intermedio formado por *pipe service* para llevar a cabo la comunicación entre peers.

Un pipe establece una comunicación punto a punto entre dos peers. Cada peer

posee un conjunto de pipes con nombre, identificador y un anuncio de pipe para cada uno. Esta información es publicada por el peer mediante anuncios (advertisements) y conocida por el resto de peers de la red. Además, dicha información puede trascender de la propia red local gracias a la difusión que de la misma realizan los peers Rendezvous. Cuando un servicio desea establecer comunicación con otro, primero busca el anuncio de este en la cache local de su peer y después en el servicio de directorio, por si este estuviese en una subred distinta. A partir de la información contenida en el anuncio, se lleva a cabo la creación del pipe que permite establecer la comunicación entre servicios.

El peer asociado con cada servicio posee un pipe especial denominado pipe de interacción o *PipeI*. Este pipe es utilizado para colaborar con otros servicios del mismo modo que si se tratase de un pipe de entrada a un servicio consumidor. El *PipeI* permite a los servicios actuar como clientes en la red, acercándose más al comportamiento propio de los P2P que al cliente/servidor.

La comunicación estándar entre servicios DOHA se lleva a cabo de forma síncrona. Esto se debe a que los servicios interaccionan entre ellos mediante el envío de mensajes de solicitud y respuesta. Sin embargo, la plataforma DOHA también soporta un mecanismo de comunicación asíncrona entre servicios. La comunicación asíncrona es de utilidad especialmente para el envío de alertas o mensajes de aviso sin solicitud previa. La implementación de este tipo de comunicación en los servicios DOHA se lleva a cabo mediante el uso de otro peer especial denominado pipe asíncorno o *PipeA*.

La Figura 5.10 muestra el flujo de mensajes establecido en una comunicación estándar entre servicios DOHA-JXTA. En la primera parte de la comunicación denominada *Publish Stage* todos los servicios publican sus anuncios, es decir, cada peer publican sus advertisements en el JXTA PeerGroup. En la segunda parte denominada *Locaton Stage* se realiza una búsqueda de servicios disponibles. Dicha búsqueda se lleva a cabo de forma local, buscando los servicios disponibles en el JXTA PeerGroup, y de forma remota, solicitando al Rendezvous Peer, representado como *Directory Service*, aquellos anuncios que coincidan con los criterios de búsqueda a pesar de formar parte de una red remota. Por último, durante la fase de *Request Stage*, un servicio puede comunicarse con otro haciendo uso de la información contenida en dichos anuncios, para lo cual desde su peer creará un pipe de comunicación con el peer del servicio requerido que usará para el envío y recepción de mensajes.

Llevar a cabo un análisis y seguimiento del flujo de comportamiento de un servicio DOHA-JXTA nos ilustra cómo este lleva a cabo su comportamiento dinámicamente en el sistema global. A continuación se presenta un diagrama de actividad, Figura 5.11 que representa el flujo de actividades que lleva a cabo un servicio en DOHA-JXTA a alto nivel. En dicho diagrama de actividad, se puede distinguir cómo varía el comportamiento del servicio en base a la capa de abstracción en la que se encuentre su ejecución, ya sea interfaz, aplicación o interacción.

Los servicios ejecutan la actividad de "waiting for messages" en la capa de interfaz. Durante esta actividad los servicios delegan en sus peers la escucha en los pipes de entrada esperando recibir mensajes de peticiones desde otros servicios. Una vez

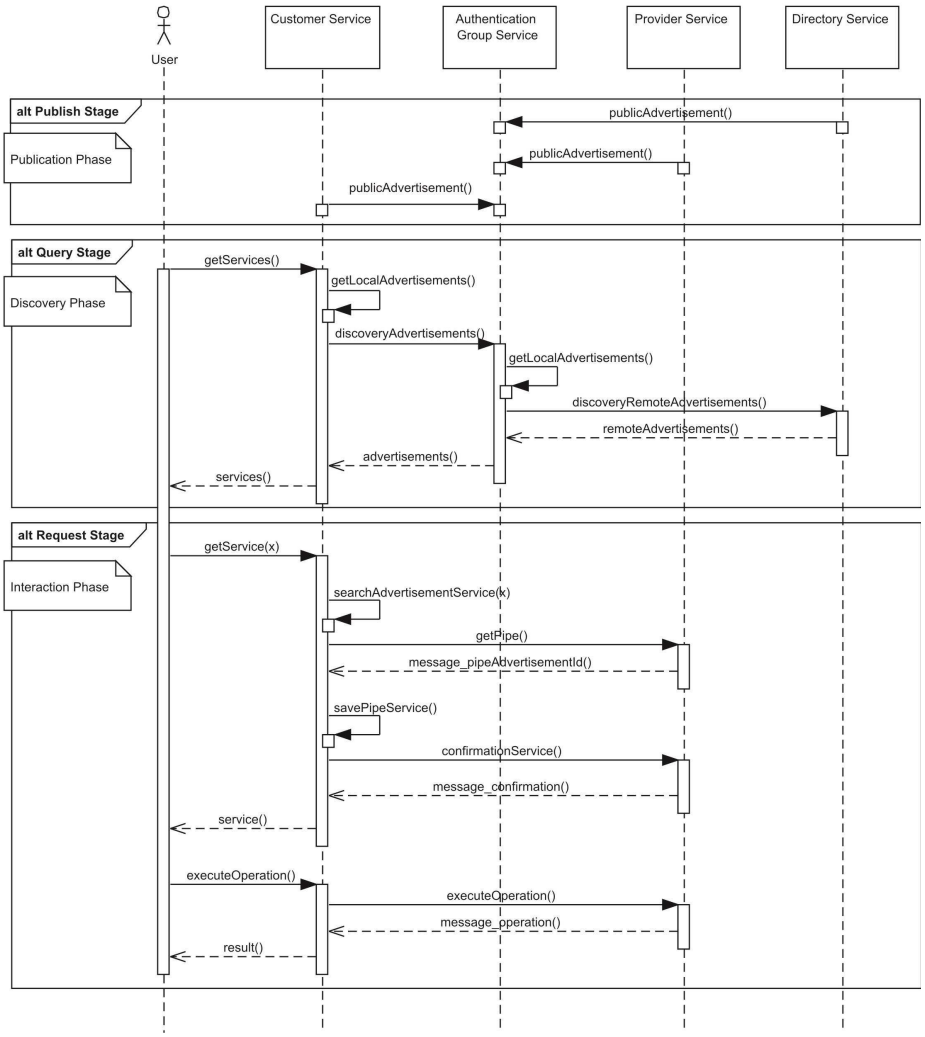


Figura 5.10: Modelo de comunicación entre servicios DOHA-JXTA

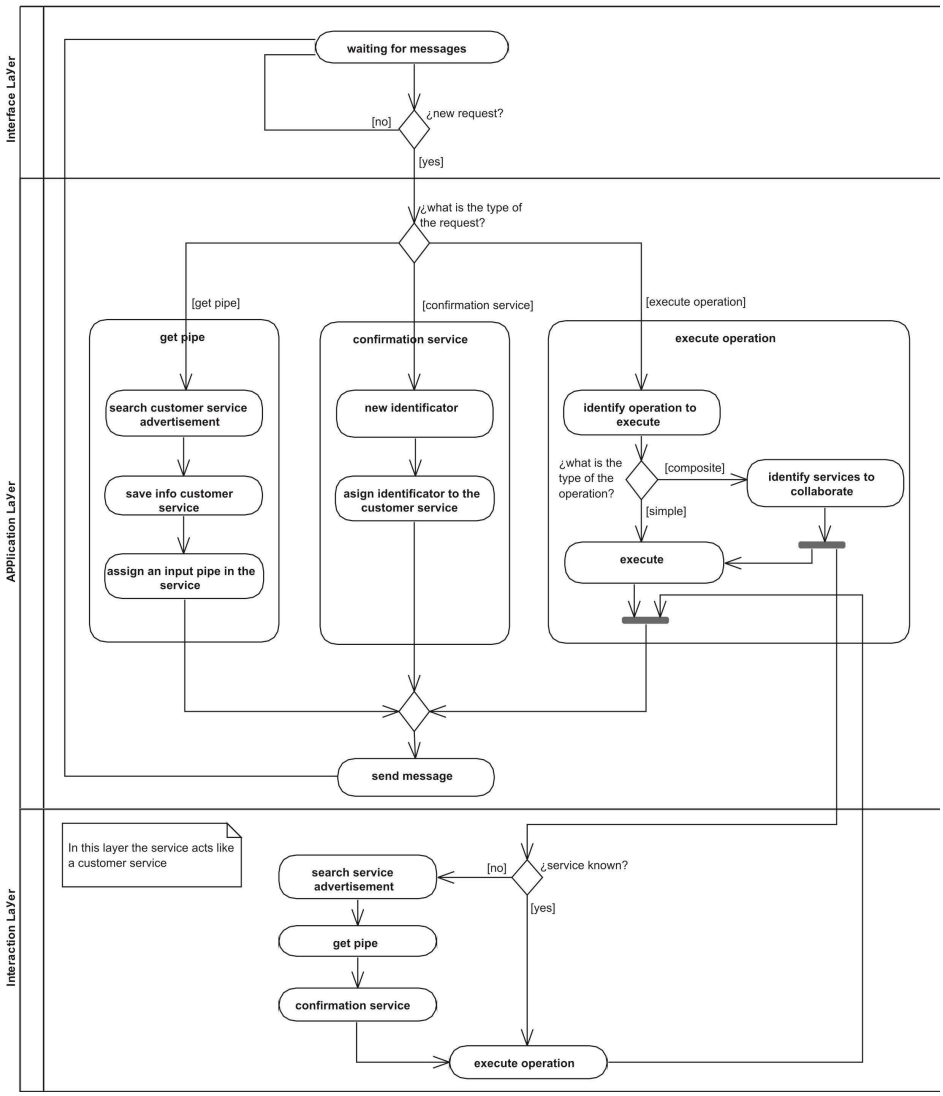


Figura 5.11: Comportamiento de un servicio DOHA-JXTA en base a sus actividades

que llega un mensaje de esta tipo, la actividad cede el control a la capa de aplicación, la cual es responsable de llevar a cabo el procesamiento del mensaje.

La mayor complejidad en cuanto a la ejecución de los servicios DOHA se encuentra encapsulada en la capa de aplicación. Es aquí donde se determina la lógica de aplicación de los servicios. De este modo, cuando una petición llega a la capa de interfaz y el flujo de ejecución se pasa a la capa de aplicación, se determina cuál es el tipo de la petición recibida. El tipo de petición puede variar entre *get pipe*, *confirmation service* y *execute operation*, cada una de las cuales contendrá una subconjunto de actividades que el servicio llevará a cabo para darles respuesta. Todas las peticiones finalizan con el envío de un mensaje de respuesta al servicio cliente por parte del servicio que ha sido requerido.

La actividad *get pipe* tiene la función de asociar un pipe de entrada en el servicio para facilitar la comunicación con el servicio cliente. Para ello, se realiza la búsqueda del anuncio del servicio cliente en la red para obtener información del mismo, como por ejemplo, de qué servicio en concreto se trata y cuál es su pipe de comunicación. A partir de esta información se puede crear un canal de comunicación o pipe entre ambos y se considera establecida la comunicación.

La actividad *confirmation service* permite a los servicios clientes verificar si el servicio que requieren está disponible para la comunicación. Esta actividad asigna un identificador unívoco a cada servicio cliente en el servicio requerido a modo de registro. Así, el servicio tiene localizados a los potenciales consumidores de su funcionalidad para agilizar la comunicación con los mismos.

Por último, la actividad *execute operation* es la que lleva a cabo la ejecución de una de las operaciones del servicio, que corresponderá con la que haya sido requerida por el servicio cliente. Primero se determina el tipo de operación. Si se trata de una operación simple, el servicio la ejecutará completamente en su capa de aplicación. Si por el contrario, se trata de una operación compuesta, se deben identificar primero los servicios con los que se colabora en dicha operación y pasar el flujo de ejecución a la capa de interacción.

Finalmente, en la capa de interacción, se encapsula la funcionalidad relativa a la colaboración con otros servicios. A través de esta capa el servicio actúa como cliente de otros servicios proveedores. En esta capa el servicio lleva a cabo una serie de actividades tal y como si de un servicio cliente se tratase, como buscar el anuncio del servicio con el que desea interactuar, obtener su pipe, confirmar la conexión con el mismo y ejecutar la operación. Tras cada operación, el flujo de ejecución vuelve de nuevo a la capa de aplicación en la cual se van recopilando todos los resultados de las operaciones requeridas por la misma operación compuesta antes de dar una respuesta al servicio consumidor.

El desacoplamiento existente entre las capas de aplicación e interfaz facilita el control de la propiedad de "stateless" o servicios sin estado establecida por SOA. Todos los servicios DOHA son sin estado desde un punto de vista externo. Esto se debe a que la capa de interfaz provee siempre una respuesta a cualquier solicitud realizada desde un servicio cliente.

Por otro lado, los servicios de tipo dispositivo pueden llevar a cabo una virtua-

lización de los dispositivos físicos con los que interaccionan almacenando su estado en una variable temporal. Sin embargo, el objetivo de este estado es muy distinto al mencionado anteriormente y no afecta en absoluto al exterior, para el que es completamente imperceptible. Por ejemplo, cuando un servicio virtualiza el estado del dispositivo físico sensor de temperatura encapsulando su valor en una variable cualquiera del servicio, facilita la comunicación con los potenciales servicios clientes que requieran dicha información. Si por el contrario, cada vez que recibiese una solicitud tuviese que testear el dispositivo físico, el tiempo de respuesta podría verse muy mermado.

Gestión de errores de red. La calidad y simplicidad en las comunicaciones de la red de nodos es de gran importancia en un entorno de computación ubicua. La naturaleza dinámica y distribuida de las redes P2P no están libres de errores debidos a sobrecarga de la red, fallos en la comunicación o pérdida de mensajes. Debido a esto, se han incorporado a DOHA-JXTA algunos mecanismos para tratar de mitigar las posibles consecuencias de estos casos.

Por un lado, es importante garantizar que los servicios no permanezcan infinitamente bloqueados esperando la recepción de un mensaje en su pipe. La plataforma JXTA proporciona distintos métodos para implementar la recepción de mensajes. A través del método *InputPipe.waitForMessage()*, el peer que lo ejecuta espera indefinidamente la recepción de un mensaje en su pipe. Si ocurre un error en la red y el mensaje no llega, este peer se bloquearía, dejando inutilizable al servicio DOHA que lo aloja. Para evitar este error en DOHA-JXTA se ha optado por emplear el método *InputPipe.poll(timeout)*. Este provee una protección para el peer implementando un mecanismo preventivo de tolerancia a fallos.

La publicación y descubrimiento de mensajes JXTA también puede ser una fuente de problemas. Los servicios publican anuncios en la red, pero no hay garantía de que estos anuncios lleguen finalmente a sus destinatarios. De ser así, los anuncios quedarían vagando por la red. Para evitar la excesiva propagación de solicitudes a través de la red JXTA, se ha asociado un tiempo de vida limitado a los anuncios de DOHA-JXTA. Una vez expirado dicho tiempo, el anuncio que sigue en la red muere, desaparece. Cuando esto ocurre, el servicio debe actuar en consecuencia, volviendo a publicar dicho anuncio si fuese necesario. Asimismo, los servicios de la red deberán realizar búsquedas regulares de anuncios en la red para poder descubrir posibles nuevos anuncios.

5.4.2. DOHA sobre DPWS

El framework DPWS, acrónimo de *Device Profile for Web Services*, es una plataforma basada en Web Services especialmente diseñada para su uso en dispositivos empotrados con recursos limitados. El proyecto fue inicialmente impulsado por Microsoft junto con otras compañías. Su rápida evolución lo convirtió en un estándar OASIS en 2009 [OASIS, 2009]. El principal objetivo de DPWS es facilitar la construcción de aplicaciones basadas en Web-Services sobre dispositivos empotrados,

teniendo en cuenta que estos están caracterizados por poseer recursos limitados, tanto de procesamiento como de memoria.

Llevar a cabo una implementación de DOHA sobre DPWS respeta las pautas de diseño establecidas, gracias a la estrecha relación de esta plataforma con los conceptos SOA. En adelante, denominaremos a esta implementación DOHA-DPWS.

Elementos básicos de DPWS. Son varios los conceptos introducidos por DPWS y que están presentes en la implementación de DOHA-DPWS. En DPWS el elemento principal es el *Device* en torno al cual giran los servicios, que tendrán rol de *Hosting Service* y *Hosted Service*. El *Device* es el punto de acceso público al cual se debe acceder para utilizar los servicios hospedados en él, es decir, los *Hosted Services*. Por ello, el dispositivo actúa como huésped de estos servicios y actúa con un rol de *Hosting Service*. En la Figura 5.12 podemos ver un esquema de estos elementos.

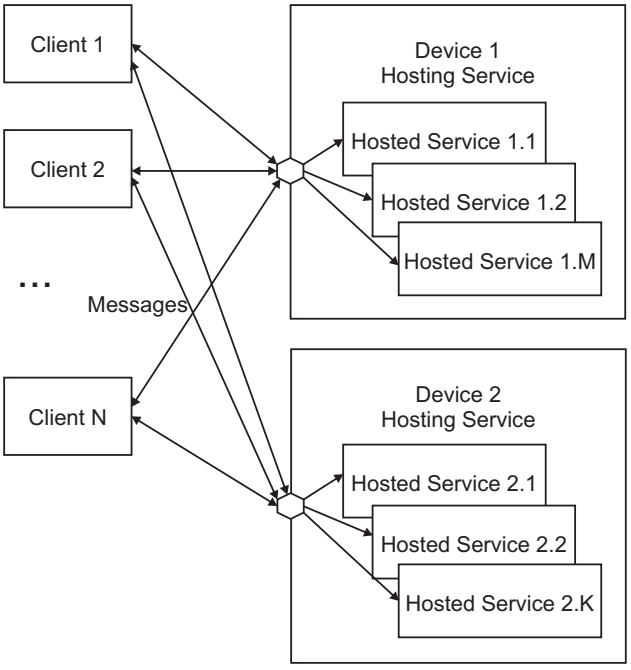


Figura 5.12: Elementos principales de un servicio DPWS

Gracias a la distinción entre servicios que realiza DPWS, *Hosted* and *Hosting services*, podemos asegurar que la propiedad de *stateless* o sin estado requerida por los servicios DOHA se cumple en esta implementación. El *Device* que actúa como *Hosting Service* resuelve las solicitudes externas y redirige al *Hosted Service* adecuado cada una de las peticiones. El *Device* está funcionando siempre y siempre puede gestionar las peticiones que llegan, actúa como un gestor de peticiones, pues la funcionalidad está encapsulada en los *Hosted Services*.

Descubrimiento de servicios. La plataforma DPWS proporciona un mecanismo de descubrimiento basado en la captación de mensajes [Ribeiro et al., 2008]. Para ello los *Device* de DPWS publican mensajes autodescriptivos, es decir, informan sobre su nombre, UUID y sus operaciones.

Dichos mensajes llegan a todos los servicios que se encuentren en la misma subred. Sin embargo, DPWS no resuelve el problema de la captación de mensajes entre distintas redes. Para resolver este contratiempo, se ha implementado un servicio especial en DOHA que actúa de intermediario entre subredes, funcionando como un directorio de servicios.

Comunicación entre servicios. La comunicación entre servicios DPWS se lleva a cabo a través de mensajes SOAP [Zeeb et al., 2007], pudiendo llevar a cabo tanto comunicación síncrona como asíncrona. Para implementar una comunicación asíncrona entre servicios se utiliza la suscripción a eventos, mientras que para la comunicación síncrona se utilizan mensajes de solicitud y respuesta [Zeeb et al., 2010].

Gestión de errores de red. Para evitar, en la medida de los posible, que los errores de red puedan afectar al funcionamiento de los servicios DOHA-DPWS hemos utilizado algunos de los mecanismos que ofrece DPWS, como son el uso de los mensajes *Hello* y acotar el tiempo de espera tanto en la recepción como en el envío de mensajes.

Los mensajes *Hello* y *Goodbye* de DPWS permiten a los servicios autopublicitarse en la red. Dichos mensajes son capturados por el resto de dispositivos que están escuchando en la red, de forma que los servicios que estén interesados en el servicio que acaba de llegar y publicarse pueden conocerlo y comunicarse con él. Del mismo modo, cuando un servicio abandona la red envía un mensaje de *Goodbye*, para que los servicios con los que colabora puedan saber que no estará disponible a partir del instante en el que se lanza dicho mensaje. Gracias a estos dos tipos de mensajes especiales, los servicios DOHA-DPWS podrán determinar cuándo la comunicación con otros es o no posible, en base a si están o no disponibles en la red.

Por otro lado, acotar el tiempo de espera en el envío y recepción de mensajes pretende evitar el bloqueo de los servicios ante pérdidas de mensajes inesperadas. Para ello se inicia un temporizador con cada petición. Si la solicitud no llega antes del tiempo establecido, se omite la petición y se continúa con la ejecución del servicio.

5.5. Composición de Servicios

El modelo de composición de DOHA se basa en las actividades que un servicio lleva a cabo cuando necesita colaborar con otros servicios. Dicho modelo se ha definido teniendo en cuenta las características de los entornos ubicuos para ser aplicado en cualquier middleware o plataforma distribuida basada en los principios de SOA.

5.5.1. Modelo de Composición basado en Grafos Dirigidos

Las operaciones compuestas son la base del modelo de composición de DOHA. Dicho modelo de composición se ha definido utilizando los conceptos propios de la teoría de grafos [Jensen, 1996] [Bender and Ron, 2002]. El uso de grafos en la especificación del modelo ha venido determinada por la estrecha relación existente entre el concepto de grafo y el flujo de ejecución de una operación compuesta, siendo natural la representación de dicho flujo de ejecución mediante grafos.

Cada operación compuesta *op* de un servicio S puede ser definida por un grafo dirigido en el que:

$$G_{op_S} = (o_{op_S}, V(G), L(G), E(G)) \tag{5.3}$$

- o_{op_S} es el vértice principal del grafo y se corresponde con el servicio origen S de la operación compuesta op_S.
- $V(G)$ es el conjunto de vértices del grafo. Cada uno de los vértices del conjunto representa a un servicio requerido, *required service*. En cada uno de estos servicios requeridos será invocada una operación desde la operación compuesta op_S.
- $L(G)$ es un conjunto de etiquetas en el que cada una representa una operación solicitada, *requested operation*, en un servicio requerido de $V(G)$.
- $E(G)$ es el conjunto de arcos o enlaces entre el vértice origen o_{op_S} y el vértice destino en $V(G)$, etiquetados por elementos de $L(G)$. En consecuencia, cada uno de los elementos de este conjunto se define con la siguiente función *edge*:

$$edge_i(o_{op_S}, op_{i_{v_j}}, v_j) \tag{5.4}$$

  Donde o_{op_S} es el servicio origen y op_i es la operación solicitada en el servicio requerido v_j, verificando que $E(G) \subseteq o \times L(G) \times V(G)$.

En el grafo de una operación compuesta, los arcos u operaciones entre vértices siempre tienen un inicio o emisor y un fin o receptor. El punto de inicio se corresponderá con el servicio invocador o *invoker service*, el cual a su vez será el poseedor de la operación compuesta que llevará cabo la solicitud. El destino será el servicio requerido o *required service*, en el cual se encuentra la operación solicitada o *requested operation*. Los servicios requeridos no son conscientes de los servicios invocadores ni tampoco de las operaciones compuestas que los solicitan. El grafo de una operación compuesta puede contener llamadas a numerosas operaciones solicitadas, pero estas operaciones se llevan a cabo de un modo secuencial, no anidado ni en paralelo. Esto implica que la invocación de operaciones solicitadas debe ser ejecutada siguiendo un orden secuencial en el que cada operación debe haber finalizado completamente antes de comenzar la ejecución de otra operación solicitada.

El modelo de composición de DOHA está centrado en la caracterización de cualquier operación compuesta individual en un servicio y su interacción con el resto de operaciones solicitadas en los servicios requeridos. Sin embargo, es adecuado realizar una definición formal en el contexto de un servicio para representar la capacidad de interacción del mismo en base a las operaciones que este provee al resto de servicios que puedan actuar como sus consumidores.

Podemos establecer el grafo de composición de un servicio S como un supergrafo dirigido o $map(S)$ formado por la unión de los grafos de todas sus operaciones compuestas, del siguiente modo:

$$\begin{aligned} map(S) &= G_{op_{1_S}} \cup G_{op_{2_S}} \cup G_{op_{3_S}} \cup ... \cup G_{op_{n_S}} \\ &= \cup_i G_{op_{i_S}} \end{aligned} \tag{5.5}$$

Dado un servicio S, los grafos de sus operaciones compuestas, $G_{op_{i_S}}$, y el grafo de composición del propio servicio, $map(S)$, tienen el mismo vértice de origen, el propio servicio S como servicio invocador. La Definición 5.5 representa formalmente la capacidad de colaboración de un servicio determinado. El mapa de composición de un servicio o *service composition map* define las operaciones compuestas que un servicio puede llevar a cabo de forma colaborativa utilizando un representación basada en grafos. Este grafo enlaza los servicios invocadores y sus operaciones compuestas con los servicios requeridos y las operaciones solicitadas en los mismos.

El nivel de profundidad del grafo de composición de un servicio es siempre de nivel uno, ya que el servicio únicamente puede conocer sus operaciones solicitadas y sus servicios requeridos, pero no más allá. En otras palabras, cuando un servicio comienza una operación compuesta, este no puede saber si las operaciones solicitadas serán a su vez compuestas. En este caso, se podría dar la ejecución de varias operaciones compuestas entre distintos servicios requeridos en cascada. Para manejar a alto nivel la profundidad de dicha cascada de operaciones hemos establecido un nivel de complejidad asociada a las operaciones compuestas.

Cada operación tendrá asignado un valor que representará su grado de complejidad o *complexity degree*. El grado de complejidad de una operación en DOHA es un valor entero que establece el máximo número de operaciones en cascada que son invocadas desde un servicio requerido desde el inicio hasta el final de la ejecución de la operación. Se ha definido la función $complexDeg(op_S)$ como la función que determina el grado de complejidad de la operación de un servicio op_S. Formalmente, $complexDeg(op_S) : L(G) \rightarrow \mathbb{N}$, $complexDeg(op_S) = k \; where \; k \rightarrow \mathbb{N}$. Siempre y cuando $k = 0$, op_S será una operación simple, mientras que cuando $k > 0$, la operación op_S será una operación compuesta.

Solo las operaciones compuestas tienen grafo de composición. El grado de complejidad de una operación compuesta en base a su mapa de composición G_{op_S} puede definirse como:

$$\begin{aligned} complexDeg(G_{op_S}) &= max(complexDeg(op_i)) + 1 \\ &\quad \forall op_i \in L(G) \end{aligned} \tag{5.6}$$

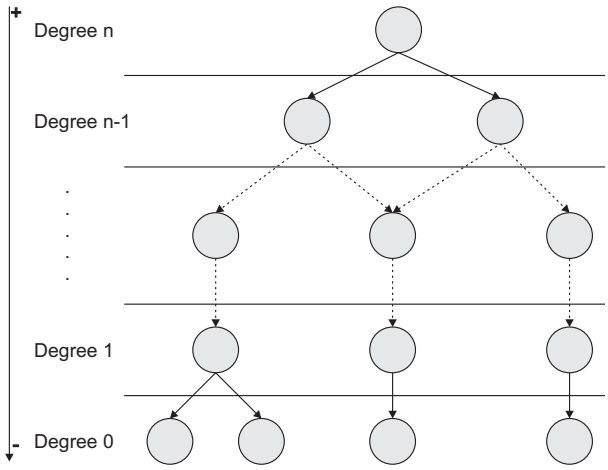


Figura 5.13: Representación jerárquica del grado de complejidad de una operación compuesta

Como consecuencia, el grado de complejidad de una operación compuesta es el mismo que el grado de complejidad de su grafo de composición – es decir, $complexDeg(op_S) = complexDeg(G_{op_S})$. Debido a esto, el grado de complejidad de un servicio es igual al máximo grado de complejidad de todas las operaciones compuestas de su mapa de composición, tal y como podemos observar en la Figura 5.13, es decir:

$$complexDeg(map(S)) = max(complexDeg(G_{op_{i_S}})) \quad \forall G_{op_{i_S}} \subseteq map(S) \tag{5.7}$$

La función $complexDeg()$ se emplea para asegurar la aciclicidad del grafo de composición, para lo cual además se establecen las siguientes restricciones:

- Todas las operaciones compuestas deben finalizar en una operación simple.
- Una operación compuesta puede invocar una o más operaciones siempre y cuando estas tengan menor grado de complejidad. Por ejemplo, una operación compuesta con grado de complejidad 1 solo puede invocar operaciones simples – es decir, operaciones con grado de complejidad 0. Una operación compuesta con grado de complejidad n puede invocar operaciones con un máximo grado de complejidad igual a $n-1$ – es decir, al menos una de las operaciones solicitadas por la operación con grado de complejidad n tendrá grado de complejidad igual a $n-1$.
- Las operaciones invocadas por una operación compuesta deben formar parte de un servicio distinto al servicio al que pertenece la operación compuesta. Es decir, una operación no puede invocar operaciones en su propio servicio.

En base a estas restricciones, se derivan importantes consecuencias que deben satisfacer los enlaces o arcos del grafo de una operación compuesta. Dado $op_{i_{V_1}}$ una operación compuesta en el servicio V_1 (vértice origen del grafo de composición) y op_k una operación solicitada en el servicio V_2 (vértice destino del grafo de composición), podemos definir el siguiente axioma:

$$edge(o_{op_{i_{V_1}}}, op_{k_{V_2}}, V_2) \rightarrow complexDeg(op_{i_{V_1}}) > complexDeg(op_{k_{V_2}}) \quad (5.8)$$

El Axioma 5.8 establece que el grado de complejidad de la operación compuesta $op_{k_{V_2}}$ del servicio V_2 debe ser estrictamente inferior que el grado de complejidad de la operación que la ha invocado – es decir, la operación $op_{i_{V_1}}$ del servicio V_1.

5.5.2. Mapa de Composición Completo

Cada servicio tiene su propio grafo de composición, $map(S)$, que modela todas sus operaciones compuestas. El conjunto de todos los grafos de composición de una aplicación forma el flujo colaborativo que tiene lugar durante la ejecución de dicha aplicación. Por construcción, podemos definir el grafo de composición global, que denominaremos grafo de composición del sistema o *system composition graph*, de una aplicación como la unión de todos los grafos de composición individuales de los servicios de la aplicación *app*. Esto implica:

$$\begin{aligned} map(app) &= \cup_i map(S_i) \\ &= \cup_{i,j} map(G_{op_{j_{S_i}}}) \\ &= \cup_{i,j}(o_{op_{j_{S_i}}}, V_{op_{j_{S_i}}}, L_{op_{j_{S_i}}}, E_{op_{j_{S_i}}}) \end{aligned} \quad (5.9)$$

El grafo de composición del sistema puede ser construido dinámicamente para identificar los mapas de composición de los servicios que se están ejecutando en una aplicación en un momento determinado. Esta información puede ser de utilidad en estudios de rendimiento, fiabilidad, carga de trabajo, retrasos o cualquier otro parámetro relacionado con la calidad de servicio o QoS.

El mapa de composición de una aplicación completa tiene propiedades interesantes que pueden resultar beneficiosas a la hora de establecer la corrección de la composición llevada a cabo entre los servicios del sistema. Consideraremos un servicio S_1 con dos operaciones compuestas op_A y op_B. El grafo que representa el mapa de composición del servicio S_1 se muestra en la Figura 5.14. Las operaciones invocadas por la operación compuesta op_A serán $op_{C_{V_1}}$, $op_{D_{V_1}}$ y $op_{E_{V_2}}$. $E(G)_{op_{A_{S_1}}}$ incluye el conjunto de dichas operaciones solicitadas por op_A como el conjunto de arcos o enlaces del grafo de composición de op_A:

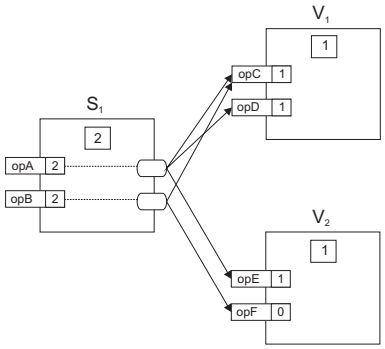


Figura 5.14: Mapa de composición del servicio de ejemplo S_1

$$E(G)_{op_{A_{S_1}}} = (edge(op_{A_{S_1}}, op_{C_{V_1}}, V_1), edge(op_{A_{S_1}}, op_{D_{V_1}}, V_1), edge(op_{A_{S_1}}, op_{E_{V_2}}, V_2)) \quad (5.10)$$

Las operaciones invocadas desde la operación compuesta op_B serán $op_{C_{V_1}}$ y $op_{F_{V_2}}$, por lo que:

$$E(G)_{op_{B_{S_1}}} = (edge(op_{B_{S_1}}, op_{C_{V_1}}, V_1), edge(op_{B_{S_1}}, op_{F_{V_2}}, V_2)) \quad (5.11)$$

El mapa de composición del servicio S será un super-grafo, de acuerdo a la Definición 5.5. Este grafo incluirá además los arcos o enlaces de cada una de las operaciones compuestas de S – es decir, todas las operaciones solicitadas invocadas por las operaciones compuestas de S.

$$E(G)_{map(S)} = (edge(op_{A_{S_1}}, op_{C_{V_1}}, V_1), edge(op_{A_{S_1}}, op_{D_{V_1}}, V_1), edge(op_{A_{S_1}}, op_{E_{V_2}}, V_2), edge(op_{B_{S_1}}, op_{C_{V_1}}, V_1), edge(op_{B_{S_1}}, op_{F_{V_2}}, V_2)) \quad (5.12)$$

Para demostrar cómo el grafo de composición de una aplicación compuesta puede formarse, vamos a examinar también los mapas de composición de los servicios requeridos V_1 y V_2. Consideramos ahora las operaciones compuestas op_C y op_D del servicio V_1 de acuerdo con el mapa de composición representado en la Figura 5.15. Las operaciones solicitadas del mapa de composición de V_1 que estarán incluidas en $E(G)$ serán:

$$E(G)_{map(V_1)} = (edge(op_{C_{V_1}}, op_{G_{W_1}}, W_1), edge(op_{C_{V_1}}, op_{I_{W_2}}, W_2), edge(op_{D_{V_1}}, op_{I_{W_2}}, W_2)) \quad (5.13)$$

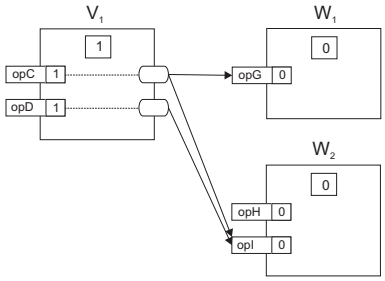


Figura 5.15: Mapa de composición del servicio de ejemplo V_1

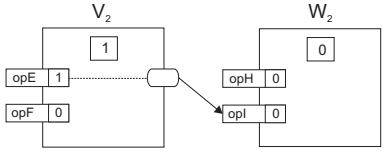


Figura 5.16: Mapa de composición del servicio de ejemplo V_2

Por otro lado, el mapa de composición del servicio V_2 de acuerdo a la Figura 5.16 incluirá la invocación de las operaciones solicitadas op_E y op_F. En este caso, la operación $op_{E_{V_2}}$ es una operación compuesta y la operación $op_{F_{V_2}}$ es una operación simple. El conjunto de arcos o enlaces contenidos en el $E(G)$ correspondiente al mapa de composición de V_2 son:

$$E(G)_{map(V_2)} = (edge(op_{E_{V_2}}, op_{I_{W_2}}, W_2)) \tag{5.14}$$

Podemos representar el grafo de composición de una aplicación compuesta mediante la unión de todos los grafos independientes, de acuerdo a la Definición 5.9 – es decir, S_1, V_1 y V_2, tal y como podemos ver gráficamente en la Figura 5.17. A partir de ambas representaciones, tanto la realizada matemáticamente como la gráfica, nos muestran que el grafo final no contiene elementos repetidos. Es decir, $map(S_1) \cap map(V_1) \cap map(V_2) = \emptyset$. Cada servicio proporciona únicamente el conjunto de operaciones solicitadas invocadas por cada una de sus operaciones compuestas con un nivel de profundidad 1. En conclusión, podemos probar que los grafos de composición son disjuntos – es decir, la intersección de todos los grafos de composición es siempre vacía y el mapa de composición global no contiene elementos repetidos.

$$\begin{aligned} \cap_i map(S_i) &= \cap_{i,j} map(G_{op_{j_{S_i}}}) \\ &= \cap_{i,j} (o_{op_{j_{S_i}}}, V_{op_{j_{S_i}}}, L_{op_{j_{S_i}}}, E_{op_{j_{S_i}}}) \\ &= \emptyset \end{aligned} \tag{5.15}$$

La corrección del modelo de composición de servicios puede establecerse a dos niveles, estático y dinámico [Dustdar and Schreiner, 2005a]. A nivel estático, el modelo

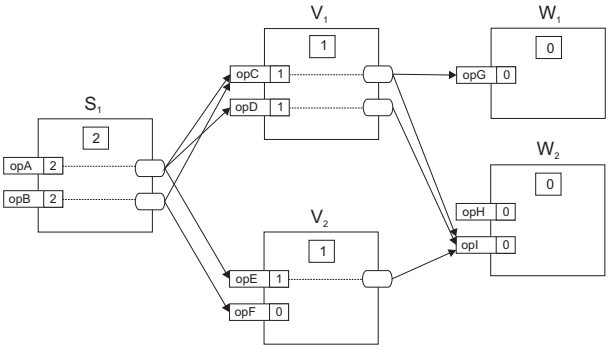


Figura 5.17: Mapa de composición completo que contiene a todos los servicios de ejemplo

de composición de servicios debe definir cómo se establecen las invocaciones entre servicios. La composición de servicios se considerará bien formada cuando podamos analizar de forma finita todas las posibles conexiones factibles entre servicios. Cuando el modelo de composición estático está bien definido, este puede ser la base para diseñar una estrategia de composición dinámica en la que los mapas de composición de los servicios se podrían construir en tiempo de ejecución.

El mapa de composición de servicios a nivel de sistema de DOHA proporciona una gestión eficiente. El conjunto de nodos es disjunto. No es necesario llevar a cabo un proceso de verificación global para comprobar la validez de todas las operaciones compuestas y el conjunto de sus operaciones solicitadas. La verificación puede llevarse a cabo independiente por parte de cada uno de los servicios, y por extensión, quedaría también validado el sistema en su conjunto. En sistemas ubicuos, es importante que la verificación se pueda llevar a cabo en cualquier momento durante la ejecución del sistema, ya que estos sistemas son altamente dinámicos y la composición puede evolucionar con el tiempo. El modelo de composición propuesto es sólido y escalable por construcción, por lo que se reduce el tiempo necesario para llevar a cabo la verificación del sistema, pues cualquier modificación que afecte a la composición entre servicios deberá satisfacer las restricciones y propiedades impuestas por el mismo.

La corrección de un sistema puede ser analizada estudiando la aciclicidad del conjunto de operaciones solicitadas de una operación compuesta. La propiedad de aciclicidad queda demostrada implícitamente y por construcción en el modelo de composición propuesto, gracias al uso del grado de complejidad de cada operación. En la siguiente sección vamos a analizar detalladamente las repercusiones de dicha propiedad en sistemas basados en servicios colaborativos, en los que nuevos servicios con nuevas operaciones compuestas pueden ser añadidos satisfaciendo las restricciones relativas al grado de complejidad que han sido definidas.

5.5.3. Propiedad de Aciclicidad

La aciclicidad es una de las principales características del grafo de composición de las operaciones compuestas, servicios y aplicaciones. La aciclicidad es una propiedad importante de los grafos que garantiza, una vez verificada, que el grafo no contiene ciclos. Además, también garantiza un número finito de vértices y aristas en el nodo.

El grafo de composición de una operación compuesta y el mapa de composición de un servicio son intrínsecamente acíclicos gracias a que el grado de complejidad en las conexiones entre sus nodos verifica el Axioma 5.8 siempre para un nivel uno de profundidad. Sin embargo, el mapa de composición completo podría ser susceptible a tener bucles, debido a que la profundidad del mismo es mayor a uno. A continuación vamos a probar que el mapa de composición completo de un sistema es correcto – es decir, el mapa de composición del sistema no contiene ciclos.

Vamos a utilizar el Axioma 5.8 para probar la propiedad de aciclicidad en el mapa de composición del sistema completo o *Full Composition Map*. En el caso de ejemplo del servicio V_1, con un grado de complejidad igual a 1, tiene una o más operaciones compuestas con grado de complejidad 1. Estas operaciones compuestas solo pueden invocar a operaciones simples con grado de complejidad 0, lo cual coincide además con el nivel de profundidad uno del mapa de composición. Por lo tanto, la aciclicidad en este caso sencillo se cumple por construcción y no pueden existir bucles en la composición.

Veamos ahora un caso un poco más complejo con distintos servicios. Consideraremos los servicios V_1, V_2 y V_3, que interaccionan entre sí a través de las operaciones compuestas op_1, op_2 y op_3, respectivamente. En este caso podemos definir dos conexiones, $edge(o_{op_{1_{V_1}}}, op_2, V_2)$ y $edge(o_{op_{2_{V_2}}}, op_3, V_3)$. De acuerdo con el Axioma 5.8, $complexDeg(op_1) > complexDeg(op_2)$ y $complexDeg(op_2) > complexDeg(op_3)$, respectivamente. Por transitividad, entonces también se verifica que $complexDeg(op_1) > complexDeg(op_3)$.

Si existiese un ciclo, tendría que existir una conexión adicional – por ejemplo, entre el último servicio y el primero, $edge(o_{op_{3_{V_3}}}, op_1, V_1)$. Pero esto implicaría que $complexDeg(op_3) > complexDeg(op_1)$, lo cual incumple el Axioma (5.8). Por lo tanto, la aciclicidad queda también demostrada por reducción al absurdo.

Por lo tanto, podemos afirmar que no importa cuán complejo sea el mapa de composición de un servicio, pues el grafo asociado al mismo debe ser siempre un grafo dirigido y acíclico.

5.5.4. Otras propiedades

El modelo de composición propuesto pretende proveer a los desarrolladores de software de aplicaciones ubicuas una herramienta válida con la que implementar un comportamiento colaborativo entre servicios mediante operaciones compuestas. Podemos asegurar la utilidad del modelo en entornos ubicuos analizando y verificando desde esta perspectiva los teoremas y propiedades definidas en las secciones anteriores y sobre las cuales se sustenta el modelo.

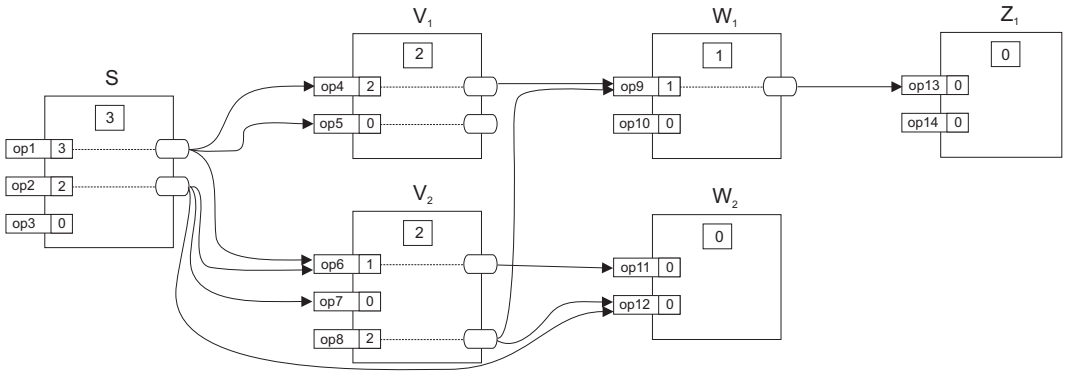


Figura 5.18: Mapa de composición de un ejemplo extendido que considera numerosos servicios con un comportamiento colaborativo

Para realizar dicho análisis, vamos a emplear un ejemplo extendido con distintos servicios compuestos. El escenario utilizado en esta sección está representado en la Figura 5.18. En el ejemplo pueden observarse hasta seis servicios distintos, S_1, V_1, V_2, W_1, W_2 y Z_1, con numerosas operaciones compuestas que establecen el comportamiento colaborativo entre ellos.

El modelo de composición proporciona un método para diseñar sistemas ubicuos que verifica que el grafo de composición de las operaciones compuestas, servicios y aplicaciones satisface el modelo. En particular, establece determinadas restricciones relativas al valor de grado de complejidad.

Un sistema ubicuo es por definición un sistema escalable y altamente dinámico, por lo que el modelo de composición debe adaptarse a cambios en la interacción entre servicios. En un sistema dinámico, como primera alternativa, podríamos establecer un valor inicial de grado de complejidad a las operaciones y servicios, a partir del cual estudiar cómo evoluciona en el tiempo, con la consiguiente actualización de los grafos de composición.

Otra alternativa sería dotar al sistema de mecanismos para determinar el grado de complejidad adecuado para cada operación y servicio. Cuando un servicio se inicia, el grado de complejidad de las operaciones simples puede asignarse *a priori* con un valor igual a 0. Sin embargo, en el caso de las operaciones compuestas, el grado de complejidad debe calcularse usando sus grafos de composición y analizando el grado de complejidad de sus operaciones solicitadas en los servicios requeridos a partir de las Ecuaciones 5.6 y 5.7. En ambos casos, el mapa de composición de los servicios es fácilmente actualizable y siempre reflejará el estado correcto del sistema en términos de composición. El mapa de composición puede almacenarse para evitar tener que volver a ser recalculado mientras no haya cambios en el sistema.

Otro aspecto a considerar es que la invocación de operaciones es unidireccional, desde un servicio que actúa como consumidor hacia un servicio que actúa como proveedor, ya que los grafos son orientados. La operación solicitada invocada por el servicio consumidor no conoce cuál ha sido la operación en este servicio que la ha requerido. Para ilustrar mejor este concepto vamos a analizar dos posibles

situaciones: a) dos operaciones solicitadas desde diferentes operaciones compuestas en distintos servicios consumidores; y b) dos operaciones solicitadas desde distintas operaciones compuestas del mismo servicio.

El primer escenario ocurre, por ejemplo, cuando la operación op_9 del servicio W_1 es solicitada por la operación op_4 del servicio V_1 y por la operación op_8 del servicio V_2, tal y como podemos ver en la Figura 5.18. Esta situación está correctamente identificada en el modelo de composición de servicios, que especifica una conexión diferente entre cada operación solicitada y el servicio consumidor que la invoca (desde un grafo de composición distinto para cada operación compuesta).

$$edge(o_{op_4}, op_9, W_1) = edge(V_1, op_9, W_1) \in E(G)_{op_{4_{V_1}}} \quad (5.16)$$

$$edge(o_{op_8}, op_9, W_1) = edge(V_2, op_9, W_1) \in E(G)_{op_{8_{V_2}}} \quad (5.17)$$

El segundo escenario se plantea, por ejemplo, cuando la operación op_6 es solicitada por las operaciones compuestas op_1 y op_2.

$$edge(o_{op_1}, op_6, V_2) = edge(S, op_6, V_2) \in E(G)_{op_{1_S}} \quad (5.18)$$

$$edge(o_{op_2}, op_6, V_2) = edge(S, op_6, V_2) \in E(G)_{op_{2_S}} \quad (5.19)$$

En ambos casos, el servicio invocador es el mismo, el servicio S, pero no podemos saber cuál es la operación invocadora, op_1 o op_2, pues ambas tienen diferentes grafos de composición para cada operación compuesta. Podemos justificar esta conclusión porque los servicios no deben saber qué operación es la que solicita su funcionalidad – es decir, en el proceso de llamada el servicio invocador es conocido; sin embargo, no se conoce la operación específica del servicio invocador que ha realizado la solicitud, lo cual es acorde a los principios establecidos por el paradigma SOA.

También es conveniente identificar las diferencias existentes entre el grado de complejidad y el grado de profundidad de una operación compuesta en el grafo de composición. El grado de complejidad de una operación compuesta muestra el máximo número de operaciones invocadas en cascada desde que la operación se inicia, al menos por una de las ramas del grafo, pero no necesariamente por todas las ramas. Por ejemplo, la operación op_8 del servicio V_2 tiene un grado de complejidad 2, lo que significa que existe al menos una rama en su grafo de composición con al menos dos operaciones ejecutadas enlazadas por su ejecución en distintos servicios. De hecho, una de las ramas desde la operación op_8 invoca a la operación op_9 del servicio W_1 con grado de complejidad 1, que a su vez invoca a la operación op_{13}

del servicio Z_1 con grado de complejidad 0. En este caso, en la otra rama de la operación op_8 únicamente se invoca a una operación, la operación op_{12} del servicio W_2 con grado de complejidad 0. Todas las ramas de la operación op_8 en el mapa de composición del sistema finalizan en una operación simple, verificando a su vez uno de los requisitos que hemos visto previamente a lo largo de la sección. Sin embargo, no todas las ramas tienen que invocar operaciones compuestas en todos los niveles conforme al grado de complejidad de la operación op_8. El grado de complejidad de una operación compuesta coincide con la profundidad de dicha operación compuesta en el mapa de composición del sistema o *full composition map* para la rama más larga.

Por último, cabe mencionar que el modelo de composición garantiza la inexistencia de ciclos siempre desde las operaciones compuestas, pero no desde los servicios. El grado de complejidad asociado a un servicio está ligado al grado de complejidad de las operaciones, pero es también una propiedad del propio servicio. Sin embargo, el grado de complejidad de un servicio no es relevante en cuanto a la satisfacción de la propiedad de aciclicidad en el grafo de composición del sistema. Por ejemplo, en la operación op_1 del servicio S solamente se invocará a la operación op_5 del servicio V_1, y en la operación op_2 del servicio S únicamente se invocará a la operación op_7 del servicio V_2. Por lo tanto, el grado de complejidad del servicio S será igual a 1, el máximo grado de complejidad de las operaciones solicitadas más 1. Sin embargo, los servicios requeridos, V_1 y V_2, tienen un grado de complejidad igual a 2, es decir, el grado de complejidad de los servicios requeridos puede ser superior al de los servicios invocadores. Esta situación nunca se podrá dar a nivel de operaciones, en el que el grado de complejidad de una operación solicitada nunca puede ser superior al de la operación invocadora, debido a la imposición del Axioma 5.8.

5.5.5. Gestión del Modelo de Composición en Entornos Dinámicos

El carácter dinámico del modelo de composición propuesto depende de la asignación inicial del grado de complejidad realizada para cada una de las operaciones compuestas de los servicios del sistema, así como de su actualización en tiempo de ejecución. El uso de este mecanismo de composición en DOHA requiere que cada valor de grado de complejidad relativo a las operaciones compuestas se mantenga actualizado siempre individualmente para garantizar la consistencia del conjunto. El proceso de actualización del grado de complejidad a nivel de operación compuesta a su vez permite verificar la aciclicidad del sistema completo. Sin embargo, no es necesario establecer complejos procedimientos a llevar a cabo a lo largo de la ejecución del sistema, gracias a las propiedades aportadas por el modelo, como:

- Ligereza: No es necesario construir o verificar el mapa de composición completo de todo el sistema. Un servicio proveedor no necesita conocer qué servicios consumidores pueden requerir su funcionalidad, sino solo los servicios proveedores con los que tiene que interactuar para conseguir dicha funcionalidad.

- Descentralización: cada servicio dispone de la información necesaria para ofrecerse a cualquier servicio consumidor que requiera su funcionalidad o los recursos que encapsula. Por tanto, la información del sistema está diseminada a través de sus servicios. Con la información de cada servicio y el mapa de composición, los servicios pueden llevar a cabo su comportamiento colaborativo, sin necesidad de conocer el mapa de composición de todo el sistema.

- Eficiencia: la corrección en la colaboración entre servicios queda asegurada por construcción. Por lo tanto, esta puede ser analizada sin complejos procedimientos para la detección de errores con respecto al modelo.

- Escalabilidad: un servicio compuesto puede interaccionar con nuevos servicios y su mapa de composición puede ser actualizado fácilmente para añadir nuevas operaciones y servicios requeridos.

Para facilitar las acciones de inicialización y actualización de las variables involucradas en el modelo de composición, principalmente el grado de complejidad de las operaciones, se han diseñado una serie de algoritmos. Estos algoritmos se han denominado *dynamic algorithms*, ya que garantizan la estabilidad del modelo en sistemas con un carácter altamente dinámico, como es el caso de los sistemas ubicuos. Su ejecución será limitada, para reducir el exceso de operaciones de control relativas al mantenimiento de la aciclicidad del sistema. Podemos distinguir dos situaciones específicas en las que el grado de complejidad puede ser actualizado:

1. Primera inicialización del sistema.

2. Chequeos parciales del sistema cuando un servicio se añade, elimina o modifica.

En el primer caso, se lleva a cabo un procedimiento de asignación del grado de complejidad *bottom-up*, desde más bajo a más alto nivel. Se inicia con la asignación de grado de complejidad a los servicios básicos con menor grado de complejidad. A continuación, se evalúan los servicios con operaciones compuestas para asignarles grado de complejidad. De este modo, la verificación de la aciclicidad se lleva a cabo *on-the-fly*, en tiempo real, ya que la asignación a los servicios de más bajo nivel primero y después a los de más alto nivel la garantiza. Para facilitar este procedimiento, los servicios deben tener una operación que permita consultar su grado de complejidad desde otros servicios invocadores.

El segundo caso es algo más complejo de analizar, ya que un cambio en el grado de complejidad de una operación puede afectar a las operaciones compuestas de otros servicios que la requieran. Por lo tanto, la operación de actualización tiene que ser ejecutada solamente en los servicios invocadores que puedan tener conflictos con respecto al grado de complejidad de sus operaciones compuestas para reducir el tráfico de comunicación en la red.

En las subsecciones siguientes se analizan los algoritmos diseñados para cubrir estos aspectos.

5.5.5.1. Asignación del Grado de Complejidad

La primera vez que se ejecuta un servicio es necesario establecer el grado de complejidad de sus operaciones analizando las operaciones solicitadas en los servicios requeridos.

Para llevar a cabo esta tarea se propone un algoritmo con dos etapas. En este algoritmo se establece el grafo de composición de cada operación compuesta del servicio para determinar cuál es su grado de complejidad final. El Algoritmo 5.1 muestra el procedimiento llevado a cabo.

En la primera parte del algoritmo se asigna un grado de complejidad 0 a todas las operaciones simples y un valor predefinido a las operaciones compuestas. El grado de complejidad de las operaciones compuestas es salvado por cada servicio y su valor puede ser recuperado cuando el servicio para o se reinicia algún tiempo después, evitando así repetir completamente el proceso de inicialización. Este valor se obtiene a través de la función *readComplexDeg()* que devuelve -1 si es la primera vez que se ejecuta el servicio. Si no es así, la función devuelve el grado de complejidad del servicio almacenado.

En la segunda parte, si se trata de una operación compuesta con un valor de grado de complejidad inicial igual a -1, este valor se chequeará y actualizará. El servicio consultará a sus servicios requeridos para obtener el grado de complejidad de cada una de sus operaciones solicitadas, y el valor obtenido se verificará aplicando el Axioma 5.8. Si el grado de complejidad de las operaciones compuestas del servicio ha cambiado después de esta parte del algoritmo, los nuevos valores se almacenarán a través de la función *writeComplexDeg()*. Los cambios en el grado de complejidad producidos pueden afectar a otros servicios en la red cuando actúan como servicios consumidores del servicio. En este caso, el servicio usará la función *sendAsyncEventChange*() para comunicar los cambios producidos a todos sus posibles consumidores en la red. Esta función envía eventos asíncronos con información sobre los cambios producidos, como el servicio concreto, la operación compuesta que ha cambiado y cuál es su nuevo grado de complejidad. Veremos a continuación en detalle cómo esta información es de utilidad durante la ejecución del procedimiento de mantenimiento del grado de complejidad, en la sección 5.5.5.2.

En un escenario real, los servicios se inician y caen muy frecuentemente. El algoritmo permite que cada servicio chequee su propio mapa de composición – es decir, el algoritmo afecta a un nivel de profundidad en cada servicio con respecto al mapa de composición completo del sistema o *full composition map*. Esta verificación parcial asume que el grado de complejidad de las operaciones solicitadas es también correcto. Por lo tanto, desde el punto de vista del servicio iniciador, se considera que el algoritmo *CIA* ha sido ejecutado recursivamente en cada uno de sus servicios requeridos.

5.5.5.2. Actualización del Grado de Complejidad

Un escenario típico de computación ubicua es un espacio de interacción dinámico al que los servicios pueden unirse o dejar espontáneamente en cualquier momento.

Algoritmo 5.1: Algoritmo de inicialización del grado de complejidad - Complexity degree Initialization Algorithm (CIA)

Data: S1 start
Result: Complexity degree of composite operations of S_1 initialized.

```
1  changed[0..n] = false;
2  for op_i : map(S_1) do
       /* Stage 1: Initialize                                   */
3      if ∃L(G_op_i) then
4          complexDeg(op_i) = readComplexDeg(op_i);
5      else
6          complexDeg(op_i) = 0 ;
       /* Stage 2: Update                                       */
7      if complexDeg(op_i) == −1 then
8          for op_j : L(G_op_i) do
9              writeComplexDeg(op_j, getComplexDeg(op_j));
10         complexDeg(op_i) = maxComplexDeg(L(G_op_i)) + 1;
11         writeComplexDeg(op_i, complexDeg(op_i));
12         sendAsyncEventChange(op_i, complexDeg, S_1);
```

En estos casos, el modelo de composición propuesto ayuda a preservar la aciclicidad de la composición ajustando el grado de complejidad de los servicios invocadores. Las operaciones compuestas que invocan la funcionalidad proporcionada por los nuevos servicios deben satisfacer sus propios grados de complejidad con respecto a la nueva operación solicitada.

Hay dos posibles formas de chequear dinámicamente los cambios producidos en el grado de complejidad de un servicio consumidor, síncrona o asíncronamente. En el primer caso, el servicio consumidor debería verificar periódicamente la corrección del grado de complejidad de su mapa de composición. Por lo tanto, el servicio consumidor debería chequear todas sus operaciones solicitadas en todos sus servicios requeridos cada cierto periodo de tiempo, que denominaremos *time-check*. Sin embargo, la consulta periódica puede llevar a sobrecargar la red cuando una proporción grande de servicios deben ir actualizándose, incluso aunque no se hubiesen producido cambios en las operaciones solicitadas. Por lo tanto, esta no es una buena opción en un entorno ubicuo, en el que estaríamos inundando la red de mensajes inútiles debido a un procedimiento preventivo.

En el segundo caso, los servicios proveedores se comunicarían asíncronamente mediante eventos, haciendo un broadcast en la red cuando se produzcan cambios en el valor del grado de complejidad de alguna de sus operaciones. En este caso, son los servicios consumidores los que deben capturar estos eventos y determinar si están asociados a alguna de sus operaciones solicitadas. En este caso, esto indicaría que el grado de complejidad de sus operaciones solicitadas ha sido modificado, lo

cual podría afectar a su propio valor de grado de complejidad, por lo que se debe analizar cómo afecta este cambio.

Algoritmo 5.2: Algoritmo de mantenimiento del grado de complejidad - Complexity degree Maintenance Algorithm (CMA)

Data: Asynchronous event with a change has been received. PRE-Condition: $op_{event} \in L(map(S))$
Result: Complexity degree of composite operations of S_1 checked and modified if it is necessary.

```
1 writeComplexDeg(op_event, complexDeg(op_event));
2 for op_i : map(S_1) do
3     if op_event ∈ L(G_op_i) then
4         oldComplexDeg_op_i = complexDeg(op_i);
5         complexDeg(op_i) = maxComplexDeg(L(G_op_i)) + 1;
6         if complexDeg(op_i)! = oldComplexDeg_op_i then
7             writeComplexDeg(op_i, complexDeg(op_i));
8             sendAsyncEventChange(op_i, complexDeg, S_1);
```

Hemos diseñado el Algoritmo 5.2 utilizando la segunda opción propuesta. La elección de esta opción se debe a que reduce sustancialmente el número de mensajes enviados a través de la red, pues este se lleva a cabo cuando realmente ha ocurrido una modificación. Los pasos a llevar a cabo durante la ejecución del algoritmo son los siguientes:

- Se recibe un evento asíncrono con información sobre el nuevo grado de complejidad de una operación compuesta de un servicio de la red. Si la operación compuesta del evento forma parte del $map(S)$, el algoritmo se ejecutará.
- Los valores del grado de complejidad relativos a la operación invocadora en S relacionada con la operación asociada al evento recibido op_{event} se chequearán haciendo uso del Axioma 5.8. Si fuese necesario, el grado de complejidad de la operación invocadora en S sería actualizado.
- Si hay un cambio en el valor del grado de complejidad, la función *sendAsyncEventChange*() enviará un evento asíncrono que difundirá a todos los servicios de la red la información sobre dicho cambio que incluirá la identificación del servicio, de la operación compuesta y el nuevo valor de su grado de complejidad.

5.5.5.3. Verificación de la Aciclicidad del Mapa de Composición

El Algoritmo 5.3 nos permite además verificar la aciclicidad de los mapas de composición. Este se aplica a un único servicio y considera únicamente un nivel de profundidad en el mapa de composición del sistema completo. La aciclicidad del

servicio S se chequea utilizando el $map(S)$. El algoritmo verifica que el grado de complejidad de todas las operaciones solicitadas es estrictamente inferior al grado de complejidad de las operaciones invocadoras en $map(S)$.

Para determinar la aciclicidad del grafo de composición de todo el sistema se chequeará la aciclicidad de cada uno de los servicios que lo componen. Se ejecutará el Algoritmo5.3 sobre el $map(app)$ siguiendo un mecanismo similar al esquema del flujo de ejecución o *workflow* seguido en otras tecnologías aplicadas o basadas en procesos de negocio, como BPEL.

Algoritmo 5.3: Algoritmo de verificación de aciclicidad - Acyclicity Verification Algorithm (AVA)

Data: $map(S)$, the composition map of the service S
Result: Verify the acyclicity of the composition map. Return true or false.

```
1 acyclic = true;
2 for op_i : map(S_1) do
3     for op_j : L(G_{op_i}) do
4         if getComplexDeg(op_j) ≥ complexDeg(op_i) then
5             acyclic = false;
6             break;
7 return acyclic;
```

5.6. Propiedades de Calidad de Servicio QoS

Considerar propiedades QoS en un sistema de colaboración de servicios permite establecer prioridades y/o restricciones en la ejecución de dichos servicios. Asimismo, permite convertir el modelo de composición en un modelo de selección dinámica de servicios en base a dichas propiedades. Además, considerar propiedades como tiempo real o latencia, dota a los desarrolladores de herramientas para configurar la ejecución del sistema con restricciones de soft real-time. Esto también permite analizar los sistemas de forma previa a su despliegue en escenarios reales, optimizando así el uso del sistema en su conjunto y definiendo, por ejemplo, nuevas propiedades para servicios específicos que permitan optimizar aún más el rendimiento del mismo.

Para identificar las propiedades QoS a considerar para caracterizar a las distintas instancias de los servicios DOHA hemos utilizado como referencia los trabajos de investigación previos de Moser et al. [Moser et al., 2012] y Vanitha and Palanisamy [Vanitha and Palanisamy, 2010]. En dichos trabajos se recomienda diferenciar entre propiedades locales o globales. Entendemos por propiedad QoS global que puede influenciar en la selección dinámica de servicios aquella que es propia del servicio general y por lo tanto común a todas sus instancias. Son propiedades deterministas, se conocen antes de invocar al servicio. Por otro lado, las propiedades locales o no deterministas, son propias de cada instancia de servicio y se conocen tras llevar a

cabo un proceso de observación continua del comportamiento del servicio o bien son preestablecidas a priori.

En el modelo de composición de DOHA tenemos ambos tipos, propiedades deterministas o globales, como es el grado de complejidad de una operación compuesta, y propiedades no deterministas o locales, como son el tiempo real, el rendimiento, la disponibilidad y la fiabilidad.

La primera propiedad QoS definida es la de tiempo real, *Real Time*. En este caso, considerar la selección de servicios en base a su tiempo de ejecución puede interpretarse como diseñar un planificador de servicios válido para DOHA. Considerando exclusivamente la propiedad de tiempo teal, y partiendo de la base de que el modelo de composición ya está establecido y validado, se podría determinar el tiempo de ejecución de cada servicio y añadir esta información al árbol de composición, de forma que el tiempo de ejecución de la aplicación total quede acotado por el tiempo de ejecución de cada servicio que la compone.

Cada instancia de servicio tendrá asociado un tiempo de ejecución máximo muy estrechamente ligado a su mapa de composición de servicio. Dicho tiempo de ejecución depende, además de las características hardware propias del nodo físico sobre el que se ejecuta, de la complejidad de su funcionalidad. Para determinar esta propiedad QoS se etiquetará el worst-case-execution-time (WCET) de cada instancia de servicio en base al mapa de composición.

La Definición 5.20 establece cómo se calcula para cada instancia de servicio el valor de WCET de cada operación.

$$WCET = Time\_process(op) + Time\_results(op)(milliseconds) \tag{5.20}$$

Es de esperar que el WCET de una operación compuesta sea elevado, pues incluirá el tiempo de ejecución de todas las operaciones que la formen. El etiquetado de dicha propiedad permitirá determinar si es posible asumir dicho tiempo de ejecución o si por el contrario es mejor buscar otra instancia de servicio que pueda llevar a cabo la funcionalidad esperada en un tiempo inferior.

Al igual que el tiempo de respuesta será propio de cada instancia de servicio, también consideraremos el rendimiento de dichas instancias o cuántas solicitudes son capaces de gestionar por unidad de tiempo. Como propiedad QoS se considera el rendimiento o *Throughput* de los servicios. La Definición 5.21 establece cómo se determina su valor.

$$Throughput = n^o request/second(request/second) \tag{5.21}$$

Ambas propiedades, tiempo de respuesta y rendimiento, son propiedades ligadas al funcionamiento de las instancias de cada servicio y para determinar su valor será necesario llevar a cabo la observación de la ejecución de dichos servicios e ir actualizando su valor cada cierto número de ejecuciones.

También consideraremos propiedades QoS la disponibilidad o *Availability* y la confiabilidad *Reliability* del servicio, definidas en 5.22 y 5.23. Dichas propiedades son no funcionales y pueden ser comunes a distintas instancias de un mismo servicio.

$$Availability = UpTime(op)/TotalTime(op)(\%) \tag{5.22}$$

$$Reliability = n^o success_request(op)/n^o total_request(op)(\%) \tag{5.23}$$

Todas las propiedades QoS definidas, WCET, Thorughput, Availability y Reliability, son propiedades locales o no deterministas. Es decir, cada instancia de servicio debe chequearse a sí misma en un proceso inicial de testeo o check-in para poder dar valor a estas variables.

5.6.1. Definición de Servicios con Propiedades QoS

Cada instancia de servicio establece sus propiedades QoS como parte de sus atributos At_i, que forman parte de la Definición 5.1 de servicio. Puesto que la primera vez que se ejecuta el servicio dichas propiedades son desconocidas, se consideran de partida inicializadas a -1. Será la instancia del servicio la encargada de actualizar esta información en tiempo de ejecución y periódicamente para que se mantenga siempre lo más actualizada posible.

Puesto que DOHA es una plataforma dinámica, se debe dar soporte para que los servicios puedan salvar los valores de sus propiedades una vez se han inicializado. Esto posibilitará que puedan omitir de nuevo el proceso de inicialización ante una caída temporal. Para ello las instancias de servicio harán uso del archivo de configuración 5.2.1.3. Será aquí donde las distintas instancias almacenen toda la información relativa a sus propiedades QoS. El proceso de actualización periódica afectará también al almacenamiento persistente de las propiedades, para garantizar que estas hayan sido salvadas ante una posible caída del servicio.

Será el desarrollador de aplicaciones sobre DOHA el responsable de configurar cuándo se realiza el salvado. Es decir, para determinar el periodo de actualización/salvado de las propiedades QoS, el desarrollador establecerá un tiempo máximo. Para la elección de dicha cota de tiempo, el desarrollador tratará de no sobrecargar los servicios pero, a su vez, asegurar que la información de los atributos QoS se mantenga lo más actualizada posible y no se pierda.

No obstante, bajo determinadas circunstancias será necesario volver a realizar el proceso de check-in y la correspondiente actualización de propiedades. Las restricciones que llevarán a efectuar un reseteo de las propiedades QoS serán también establecidas por los desarrolladores.

5.7. Selección dinámica de Servicios

La plataforma de servicios DOHA posee un modelo de composición de servicios bien definido y evaluado como correcto. Dicho modelo de composición está basado en grafos dirigidos acíclicos mediante los cuales se asegura la inexistencia de bucles indefinidos en la colaboración entre servicios. La colaboración entre servicios se considera estática, pues el usuario define a-priori los servicios que componen el grafo

de composición, con el pre-requisito de satisfacer las restricciones impuestas por el grado de complejidad de las operaciones involucradas en la colaboración. Esto es especialmente en el diseño de aplicaciones en el que hay que garantizar restricciones de tiempo real. Una vez definido un modelo de composición verificable, se ha definido un modelo de selección dinámica de servicios basado en propiedades de calidad de servicio o QoS.

En un sistema de composición dinámica, el servicio y las operaciones invocadas son seleccionados en tiempo de ejecución [Dustdar and Schreiner, 2005a]. Sin embargo, existe cierta controversia sobre qué, y qué no es, composición dinámica de servicios. Existen, a nuestro parecer, numerosos trabajos de investigación que confunden la composición dinámica con la selección dinámica de servicios. Cuando la información y la funcionalidad de los servicios y de sus operaciones es conocida, en un sistema de composición dinámica se decidirá qué servicios en concreto utilizar para llevar a cabo una funcionalidad determinada. Por el contrario, en un sistema de selección dinámica, la composición de la funcionalidad está establecida a-priori y en tiempo de ejecución se decide qué instancia en concreto del servicio se ejecutará de entre todas las instancias disponibles. La composición dinámica es una cualidad muy potente en un sistema basado en servicios, pero aumenta la complejidad del sistema y su capacidad de adaptación en tiempo de ejecución.

Un ejemplo de confusión entre composición y selección dinámica es la propuesta realizada por Estévez-Ayres et al. [Estévez-Ayres et al., 2009]. En dicho trabajo de investigación se afirma que el modelo de composición presentado es un modelo de composición dinámica. Para ello se distingue entre el servicio como ente de alto nivel que proporciona una funcionalidad y la implementación de dicha funcionalidad concreta como *Service Implementation*, SI. Por ejemplo, se considera por un lado el servicio de luminosidad y las distintas implementaciones de dicho servicio, como la implementación del servicio de luminosidad en la cocina y en el comedor. La composición dinámica de servicios que se presenta en dicho trabajo gira en torno a estas implementaciones de servicio. Sin embargo, no se especifica cómo se modela la composición entre servicios de alto nivel. Es decir, cómo se determina qué servicio ofrece la funcionalidad necesaria para incluirse dentro del grafo de composición. Se da por supuesto que el algoritmo de selección dinámica de los SI se basa en un grafo de composición prefijado por el usuario. Aunque pueda parecer una mera cuestión de nomenclatura, es erróneo denominar composición dinámica a este tipo de modelos, pues la funcionalidad está pre-establecida a priori, y lo que se realiza en realidad es una selección dinámica de la implementación concreta del servicio a utilizar.

El mapa de composición de servicios construido hasta ahora establece de forma estática cuales son los servicios y las operaciones que determinarán la funcionalidad de una operación compuesta. Hasta aquí hablamos de composición estática. Si consideramos que un mismo servicio puede tener distintas instancias en un mismo escenario, y que las propiedades QoS variarán entre estas instancias, necesitamos un mecanismo de selección que permita determinar qué instancia del servicio seleccionar para la ejecución de una operación compuesta concreta. Es aquí donde llegamos al concepto de selección dinámica. Por lo tanto, es importante diferenciar

correctamente un servicio y las distintas instancias del mismo.

1. Cada servicio involucrado en el mapa de composición puede tener distintas implementaciones que denominaremos instancias del servicio. Es decir, puede haber dos o más instancias de un mismo servicio.

2. El algoritmo de selección entre distintas implementaciones de un mismo servicio deberá seleccionar una instancia concreta en base sus propiedades QoS.

A partir de la Definición 5.1, podemos considerar que distintas instancias de un mismo servicio tendrán en común tanto su funcionalidad o propósito, Ps_i, como las interfaces proporcionada y requerida, Ip_i e Ir_i. Por el contrario, cada instancia de servicio tendrá su propio identificador unívoco, Id_i, y lo que es esencial para definir el algoritmo de selección, establecerá sus propiedades QoS como parte de sus atributos At_i.

5.7.1. Algoritmo de selección dinámica de servicios en base a propiedades QoS

Para diseñar el algoritmo de selección dinámica de servicios en DOHA partimos del modelo de composición y de las características de los servicios DOHA. Además, se asumen una serie de restricciones implícitas a partir de las consideraciones previas, de las propiedades QoS y de la forma de asociar dichas propiedades a los servicios DOHA, tales como:

1. El modelo debe ser adaptable y abierto para poder añadir, eliminar o cambiar los atributos QoS.

2. El algoritmo de selección de servicios debe ser también adaptable y soportar cambios en los servicios.

3. Tanto los cambios en el modelo de atributos QoS como en los servicios deben ser soportados por el algoritmo de selección dinámica en tiempo de ejecución y no afectar negativamente al modelo de composición de DOHA.

4. La implementación del algoritmo de selección de servicios en DOHA debe ser autocontenida, totalmente independiente de DOHA, pero a la vez directamente acopable a la plataforma. De esta forma se previene que posibles modificaciones del modelo de selección dinámica afecten a la plataforma en su conjunto.

Para determinar que instancia de un servicio en concreto se utilizará para llevar a cabo una operación concreta se analizarán los valores de las propiedades QoS asociados a dicha instancia y operación. El Algoritmo 5.4tratará de optimizar el tiempo de ejecución, sopesando a su vez el resto de propiedades QoS establecidas. Para que todas las propiedades QoS entren en juego en la selección, a cada instancia del servicio se le asociará una puntuación determinada. De entre todas las instancias de servicio que satisfagan unos límites y mínimos de puntuación, se escogerá aleatoriamente la instancia de servicio seleccionada como resultado del algoritmo.

Algoritmo 5.4: Algoritmo de selección dinámica de servicios en base a propiedades QoS - Dynamic Selection Algorithm (DSA)

Data: $map(S)$, the composition map of the service S to select.
Result: An available instance of the service S.

$Id_s elected = -1$;
for $Id_i where Ps_i \subseteq map(S)$ **do**
    $rt = min(responseTimeMs(Id_i), maxResponseTime)$;
    $th = max(throughput(Id_i), minThroughput)$;
    $av = max(availabilityPercent(Id_i), minAvailability)$;
    $re = max(reliabilityPercent(Id_i), minReliability)$;
    $score[Id_i] = rt * (th + av + re)/100$;
$Id_s elected = getBest(score)$;
return $Id_s elected$;

5.8. Discusión

Se ha diseñado una plataforma de servicios descentralizada y distribuida basada en SOA. El concepto de servicio es el elemento fundamental en torno al que han girado las principales decisiones de diseño tomadas. Se ha establecido una anatomía de servicios determinada por la capacidad de los servicios de interactuar entre sí y de actuar tanto como clientes como proveedores, manteniendo su funcionalidad encapsulada como una "caja negra".

A partir de la anatomía de servicio se ha diseñado un modelo de composición entre servicios basado en grafos dirigidos acíclicos. Dicho modelo de composición asegura la inexistencia de bucles indefinidos en la colaboración entre servicios. La colaboración entre servicios en este sentido es estática, pues el usuario define a-priori los servicios que componen el grafo de composición, con el pre-requisito de satisfacer las restricciones impuestas por el grado de complejidad de las operaciones involucradas en la colaboración. Del grado de complejidad se puede obtener información extendida de una operación concreta. A mayor grado de complejidad, mayor interacción con otros servicios, mayor posibilidad de problemas de red y mayor tiempo de ejecución.

Para la definición del modelo de composición se ha tenido en cuenta la necesidad de satisfacer restricciones temporales ligadas a la colaboración entre servicios. Gracias a ello es posible determinar el tiempo de ejecución de cada servicio y añadir esta información al árbol de composición, de forma que el tiempo de ejecución de la aplicación total queda acotado por el tiempo de ejecución de cada servicio que la compone.

La colaboración que propone el modelo está siempre dirigida por el proceso que actúa como cliente. Es más, puesto que el servicio consumidor puede consultar el tiempo de ejecución de los posibles servicios proveedores antes de requerirlos, puede planificar la ejecución a llevar a cabo y determinar su coste en tiempo. Esto es una

ventaja con respecto a otros modelos de composición, como por ejemplo los utilizados habitualmente en procesos de negocio. En ellos, los servicios consumidores desconocen cuando un servicio proveedor depende de un tercero cuya ejecución influirá en el tiempo de obtención de un resultado válido. Sin embargo, con el modelo de composición propuesto la cota de tiempo aporta a los servicios consumidores y a los desarrolladores información suficiente para determinar la idoneidad o no de invocar a un servicio proveedor determinado.

Capítulo 6

Capa Semántica de la Plataforma de Servicios

Only one's imagination could limit the possibilities of systems that can be aware of people's situations or their contexts and do things for them.

· Seng Loke ·
Context-Aware Pervasive Systems: Architectures for a New Breed of Applications

Abstract: La inclusión de semántica asociada a la funcionalidad de los servicios de la plataforma desarrollada y el uso de ontologías para la representación de la información de contexto la convierten en una plataforma de Servicios Semánticos Sensibles al Contexto a la que hemos denominado SenSE. Gracias a ello, los servicios de la plataforma son capaces de captar información, procesarla, razonar sobre ella y actuar autónomamente en consecuencia. Todo ello ejecutándose en dispositivos con recursos limitados y poca capacidad de procesamiento.

6.1. Introducción

La domótica es producto de la progresiva inclusión de la computación en la sociedad, hasta el punto en que los dispositivos domóticos estén inmersos en los elementos de uso más cotidiano. Con un reducido tamaño, estos dispositivos pueden incrustarse en cualquier objeto de uso común, hasta formar parte del entorno de una manera transparente y ubicua [Weiser, 1993]. Además, la domótica puede enriquecerse con la inteligencia ambiental, donde los dispositivos dejan atrás su comportamiento automático, su papel de artefactos que el usuario puede utilizar, y pasan a tener un comportamiento activo, con consciencia del entorno en el que se encuentran, en definitiva, un comportamiento inteligente. Los dispositivos se presentan como objetos autónomos que requieren una mínima intervención de los usuarios. Distribuidos e interconectados físicamente, desarrollan un comportamiento colaborativo, es decir,

son capaces de formar coaliciones para ofrecer funcionalidad de valor añadido al usuario [Urbieta et al., 2007].

La computación ubicua se aleja de la idea de ordenador personal como elemento con el que interacciona el usuario, e introduce el uso de dispositivos móviles y embebidos como algo natural y cotidiano en el entorno de los usuarios. Estamos rodeados de numerosos elementos con capacidad de procesamiento, no solo PCs, PDAs o teléfonos móviles. Ejemplos de ello son los coches, televisión, dispositivos multimedia portátiles, videojuegos, electrodomésticos, etc. Imaginémonos, como apuntaba ya Mark Weiser en 1991, rodeados de interfaces inteligentes e intuitivas embebidas en los objetos cotidianos que nos rodean, en un ambiente que reconoce y responde a nuestra presencia de una forma invisible [Weiser, 1991]. Pero un ambiente plagado de dispositivos con funcionalidades muy concretas, aislados entre sí y sin colaboración con el resto, lleva consigo un difícil mantenimiento, además de no favorecer la escalabilidad, aspecto de gran importancia en computación distribuida. Aplicando el paradigma SOA en computación ubicua podemos obtener nodos computacionales que ofrecen su funcionalidad mediante abstracciones basadas en servicios, de forma autocontenida y débilmente acoplados, pudiendo combinarse con otros para obtener aplicaciones más complejas.

El problema que presentan las arquitecturas SOA es que mediante los contratos de servicio se facilita una descripción exclusivamente sintáctica de los servicios, que es insuficiente en los entornos de inteligencia ambiental. Así, una diferencia fundamental entre la programación tradicional y las aplicaciones para entornos ubicuos reside en su capacidad para llevar a cabo un funcionamiento autónomo o independiente del usuario. Para que los servicios de un entorno ubicuo puedan tener un comportamiento proactivo no es suficiente con hacer una descripción de las características y funcionalidades del servicio y los protocolos de comunicación utilizados en un contrato de servicio, sino que es necesario disponer de una descripción con más expresividad semántica que permita tener un conocimiento preciso de los datos que se utilizan y las funcionalidades que se ofertan. Para que esto sea posible se debe trabajar con información adicional de alto nivel relacionada con la funcionalidad del sistema, información que representa el "conocimiento" que se posee del mundo y que también se puede representar de forma semántica.

A partir del diseño que hasta ahora se tiene de la plataforma de servicios, el siguiente reto en el desarrollo ha sido dotarla de las características necesarias para que pueda dar soporte a dispositivos semánticos inteligentes que comparten el conocimiento del entorno y actúan en consecuencia. Para ello se ha diseñado una capa semántica que extiende la arquitectura hasta ahora diseñada y conocida como DOHA, la cual la ha convertido en una plataforma de servicios semánticos sensibles al contexto. Hemos denominado a dicha capa semántica como SenSE (SENsitive Service Environment) .

6.2. Clasificación y uso de la Información Semántica en SenSE

Para que la plataforma de servicios diseñada posibilite la creación de aplicaciones de inteligencia ambiental se va a incorporar semántica al concepto de servicio. Como ya se comentó en el Capítulo 4, incorporar información semántica a los servicios permite obtener servicios semánticos y sensibles al contexto.

Desde nuestro punto de vista, la diferencia fundamental entre un servicio semántico y un servicio sensible al contexto radica en el fin con el que ambos emplean la información semántica de la que disponen. Un servicio semántico realiza una representación semántica de su información funcional y no funcional para que otros servicios puedan interpretarla y utilizar el servicio en base a ella. En cambio, un servicio sensible al contexto realiza una representación semántica de la información de contexto, entendida esta como la información del entorno del servicio, que está relacionada con su funcionalidad. El servicio sensible al contexto podría llevar a cabo un comportamiento proactivo si dispone de las herramientas necesarias para relacionar su funcionalidad con esta información de contexto.

En el caso de los servicios semánticos, se va a llevar a cabo una representación de su información semántica haciendo uso de la ontología OWL-S, que establece una ontología general que será instanciada por cada servicio con su información particular, como su funcionalidad y sus restricciones de uso. La información semántica que se incluirá en la instanciación de la ontología para cada servicio vendrá determinada por su descripción y su información funcional (entradas, salidas, precondiciones y postcondiciones), así como su información no funcional (categoría, coste y calidad de servicio). Esta información se obtendrá a partir de los elementos de la definición del servicio, que son el contrato de servicio, mapa de composición y archivo de configuración. Por ejemplo, a partir del contrato del servicio se obtendrá su funcionalidad, del mapa de composición los grados de complejidad de sus operaciones, y del archivo de configuración sus propiedades de calidad de servicio o QoS.

En cuanto a los servicios sensibles al contexto, la información semántica que utilizan está relacionada con el entorno en el que se encuentran. Para llevar a cabo esta relación se va a emplear una ontología de contexto que represente la información del entorno propia de una plataforma de servicios ubicuos para inteligencia ambiental. Cada servicio sensible al contexto llevará a cabo una instanciación de esta ontología a partir de la cuál establezca los valores del entorno que son relevantes para su funcionalidad, a pesar de que todos los servicios comparten el mismo espacio de contexto, alegóricamente, como si se encontrasen en la misma habitación virtual. Por ejemplo, un servicio de luminosidad gestionará la ontología de contexto instanciando los elementos del entorno relativos a la iluminación del entorno. También empleará conceptos como usuario, localización o tiempo. Sin embargo, no necesitará conocer la información semántica relativa a la temperatura del entorno, concepto que si será relevante para otro tipo de servicios, como por ejemplo un servicio de climatización.

Sin embargo, para los servicios sensibles al contexto no es suficiente con re-

presentar semánticamente la información del entorno a través de una ontología de contexto, sino que es necesario además relacionar dicha información con su funcionalidad. Esto permitirá a los servicios llevar a cabo un comportamiento proactivo. Para ello vamos a definir *perfiles de comportamiento*. Estos perfiles establecerán bajo qué condiciones del entorno el servicio debe ejecutar sus operaciones, sin necesidad de que el usuario intervenga explícitamente para ello.

6.3. Mapa Semántico del Contexto

La inclusión de semántica asociada al contexto en el que se ejecutan los servicios de la plataforma requiere establecer con claridad el significado de los distintos conceptos a considerar, de forma que no quede lugar a la ambigüedad. Para ello en este apartado abordaremos cuales son estos elementos y cuál será la interpretación que de ellos se hará en el desarrollo de servicios sensibles al contexto sobre la plataforma de servicios.

6.3.1. El Entorno y sus Entidades

Dentro de la representación del espacio ubicuo además del entorno en sí mismo, cobran protagonismo los elementos existentes en él. No existe una representación ontológica estándar para definir las entidades que pueden formar parte del espacio, serán los desarrolladores de servicios sobre la plataforma los encargados de gestionar la nomenclatura a seguir para definir los objetos con los que trabajar. No obstante, las entidades que formen parte del sistema deben estar identificadas de forma unívoca, ya sea mediante instancias de las ontologías que definen la estructura del contexto o bien mediante URIs, como se llevaría a cabo en un entorno web. Por ejemplo, $owl : NamedIndividualrdf : about = "\&Context; LightD1" > ... < /owl : NamedIndividual >$ y $http : //core.ugr.es/doha/onto/codamos/Context.owl\#LightD1$ son dos ejemplos de identificación de la entidad $LightD1$.

La condición fundamental para distinguir unívocamente las entidades del entorno es que sea cual sea la forma de identificarlas, no deben existir dos entidades distintas bajo la misma denominación. Este requerimiento es especialmente importante y debe tenerse en cuenta a la hora de llevar a cabo la identificación de las distintas entidades existentes.

6.3.2. Información de Contexto

La información del contexto es un concepto clave en un servicio que pretenda satisfacer la característica de "sensibilidad al contexto". Este puede organizarse de forma jerárquica mediante un grafo que puede mapearse, por la utilización de ontologías como medio de representación, en un grafo RDF. Mediante RDF se representaría la situación de las entidades del entorno en un determinado instante

de tiempo, asociando a cada una de estas entidades un elemento del dominio de conocimiento y un valor concreto.

6.3.3. Dominio de Conocimiento

La representación del *dominio de conocimiento* se lleva a cabo mediante el concepto de *ontología*, que es el modelo de datos que representa un dominio de información y las relaciones entre los objetos de este dominio, con el fin de realizar el razonamiento sobre el mismo. Una cuestión a tener en cuenta cuando se está trabajando con ontologías OWL o reglas de dominio es determinar cómo el gestor semántico del sistema, en nuestro caso el razonador, obtiene estas ontologías y reglas. Por ejemplo, un determinado servicio sensible al contexto puede estar encargado de la luminosidad de un nodo, pero ¿cómo identifica, localiza y aplica el razonador de este servicio las ontologías y reglas apropiadas para crear la relación entre sus conceptos conocidos y la información que capta del entorno?

En el caso de la utilización de ontologías, se debe indicar explícitamente en los documentos RDF mediante la cláusula RDF $< owl : imports >$, que permite obtener las ontologías o bien utilizando una copia de la ontología en almacenamiento local, o bien descargando la ontología desde un servicio remoto, ya sea vía *http*, *cms* o *ftp*. Cuando se analiza un documento OWL y se encuentra la cláusula $< owl : imports >$, se buscan las ontologías de forma local y se cargan, y en caso de que no se disponga de la ontología en local, se deberá obtener remotamente. Algo similar ocurre con las reglas asociadas al dominio de conocimiento. En este caso, se debe tener en cuenta su relación con el escenario concreto al que están asociadas y las ontologías a las que se refieren.

6.3.4. Elementos y Valores en el Dominio

La utilización de RDF como mecanismo de representación de la información de contexto implica seguir una estructura basada en la terna $sujeto - predicado - objeto$. Dentro de esta terna, el sujeto representa a un elemento concreto del dominio, el predicado denota un rasgo o un aspecto del sujeto que se expresa como una relación entre el sujeto y el objeto, y el objeto representa el concepto de *valor* del elemento del dominio. Así, por ejemplo, la terna sensorCocina-temperatura-100 nos está indicando que el sensor de Cocina ha detectado una temperatura de 100°C (se necesitaría otra terna para saber la unidad de temperatura). Por tanto, es necesario establecer con claridad que elementos de las ontologías tomarán el papel de cada uno de los elementos de esta terna. En el caso de SenSE, un elemento del dominio de conocimiento se corresponde con la combinación $sujeto - predicado$ de la tripleta RDF. Por ejemplo, $LightD1 - hasValue$, representa el valor real de luz que emite en el entorno el regulador de luz $LightD1$.

6.3.5. Capacidad de Percepción

La capacidad de percepción está muy relacionada con el conjunto de elementos del dominio de conocimiento que una entidad es capaz de percibir, ya sea de forma directa o indirecta. En el apartado anterior se ha expuesto que un elemento del dominio de conocimiento se corresponde con la combinación $sujeto - predicado$ de la tripleta RDF, de forma que el conjunto de los objetos o valores que pueden completar las ternas que forman el dominio de conocimiento conformarán la capacidad de percepción del conjunto. Por lo tanto, cada entidad sensible al contexto podrá declarar su capacidad de percepción sobre la información del contexto como el conjunto de valores u objetos con los que se relaciona. Por ejemplo, en el caso del ejemplo del regulador de luz, este instanciaría como sigue su relación con el valor real del entorno $< Context : hasValuerdf : resource = "\&Context; Value" / >$, siendo $Value$ un elemento del tipo $< rdf : typerdf : resource = "\&units; IntegerValue" / >$.

6.3.6. Capacidad de Actuación

Determinar cuál es la capacidad de operación asociada al dominio de conocimiento del modelo es análoga a la que hemos seguido para determinar cuál es la capacidad de percepción. Las tripletas RDF pueden utilizarse para representar la información de contexto que una entidad es capaz de obtener a partir de sus operaciones, ya sea de forma directa o indirecta. Por ejemplo, se pueden representar a través de tripletas RDF las operaciones de un servicio. Volviendo al ejemplo del regulador de luminosidad, la siguiente tripleta RDF especificaría su capacidad de actuar sobre el sensor LightD1 $ServiceLightRegulatorOperationTurnOnLightLightD1$.

6.3.7. Condiciones, Reglas y Predicados

Una condición puede interpretarse como una restricción particular sobre el conjunto de valores posibles que puede adquirir un elemento concreto del dominio de conocimiento. Por ejemplo, a la característica de iluminación se pueden adherir condiciones del tipo $0 < iluminacion < 100$, que indicaría que la iluminación de los elementos del modelo variará siempre entre esos márgenes. Este tipo de condiciones limitan el conjunto de los posibles valores que puede tomar un elemento del dominio de conocimiento a un subconjunto del total, aquellos que satisfacen la condición.

Para relacionar estas condiciones con la capacidad de percepción y actuación de los servicios se van a definir *perfiles de comportamiento*. En ellos, se declarará en base a reglas la relación existente entre la información del contexto y la funcionalidad del servicio. Entraremos más adelante en este capítulo en su diseño, pues será clave para delimitar la capacidad de actuación proactiva del sistema.

6.4. Procesamiento Semántico de la Información de Contexto

Los servicios sensibles al contexto están ligados a la ontología definida sobre el entorno y a los cambios que se produzcan sobre este. Para que esta relación sea veraz, los servicios activos en la plataforma realizarán consultas periódicas sobre el estado del entorno y comprobarán si esta información está actualizada. El tiempo que pasa desde que el servicio verifica la información de contexto e inicia el procedimiento, hasta que lo vuelve a realizar, puede establecerse en base al WCET de sus operaciones. De este modo, la información del entorno siempre estaría actualizada. Otra opción sería, en lugar de hacer una comprobación periódica, diseñar un servicio centralizado encargado de la gestión de la información de contexto al que los demás servicios le notificasen a través de eventos los cambios en su estado. En SenSE hemos optado por la opción distribuida y que cada servicio sea responsable de esta tarea.

Cuando se detecten cambios en la información del entorno, se evaluarán las condiciones de los perfiles de comportamiento para determinar si es necesario o no llevar a cabo una acción sobre el entorno de forma proactiva. Un esquema básico del proceso a llevar a cabo para procesar la información semántica relativa al contexto de los servicios se muestra en la Figura 6.1 y se resume a continuación.

1. **Update Context – Actualización de la información de contexto.** Los servicios comprueban la información de contexto que les es relevante. El servicio posee capacidad de percepción directa si puede consultar o actualizar los dispositivos físicos. En cambio los servicios con capacidad de percepción indirecta, es decir, sin acceso a los dispositivos físicos, tendrán que consultar otros servicios como fuentes de la información de contexto. Este procedimiento se llevará a cabo a través de operaciones compuestas.

2. **Context Information Retrieval – Obtención de la información de contexto.** La ejecución de esta tarea implica la recuperación de información de contexto real de las fuentes determinadas en el paso anterior.

3. **Reasoning – Razonamiento.** Este proceso implica la gestión de la información de contexto obtenida contrastándola con los perfiles de comportamiento en base a la ontología que definen las características del entorno.

4. **Analysis – Análisis.** En esta etapa se procede a interpretar la información obtenida del contexto junto con la obtenida a partir del proceso de razonamiento, para determinar cuál es el comportamiento proactivo más adecuado a llevar a cabo.

5. **Action – Actuar sobre el entorno.** Por último, se lleva a cabo la ejecución de las operaciones que se hayan determinado en la fase anterior. Se debe determinar si el servicio posee la capacidad de ejecutar tales operaciones, ya

sea de forma directa sobre los elementos del entorno, en caso de los servicios de tipo *Device Service*, con acceso directo al hardware, o bien indirectamente a través de otros servicios mediante operaciones compuestas.

6.5. Arquitectura de SenSE

Para definir la arquitectura software de la capa semántica SenSE debemos considerar aquellos elementos esenciales para la representación y uso de la información semántica de los servicios. La Figura 6.2 muestra un esquema de la arquitectura software de SenSE en base a estos elementos representados como componentes.

La representación de la información semántica que van a utilizar los servicios se llevará a cabo en base a ontologías. Así, se va a emplear dos ontologías, una para representar la información semántica propia del servicio y otra para representar la información de contexto.

Además de la representación de la información, se van a diseñar perfiles de comportamiento ligados a la ontología de contexto y a la funcionalidad de los servicios

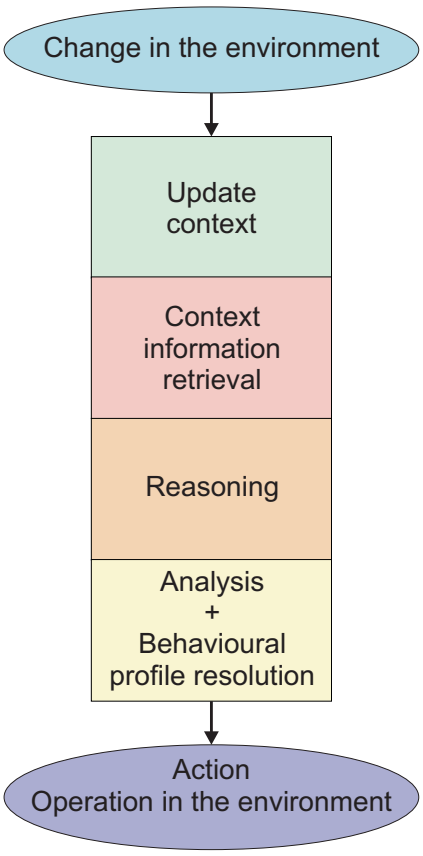


Figura 6.1: Procesamiento de la información de contexto y comportamiento proactivo de los servicios

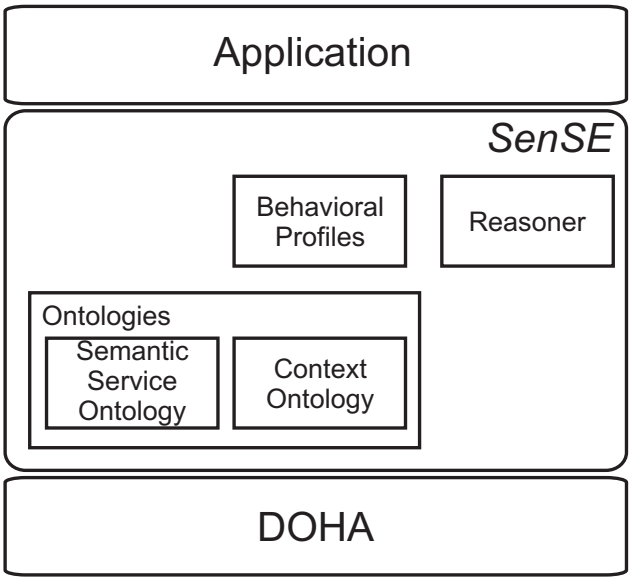


Figura 6.2: Arquitectura abstracta de la capa semántica de la plataforma de servicios

para determinar su comportamiento proactivo. A través de estos perfiles se establece bajo qué circunstancias se desea que la situación del entorno cambie en algún sentido. Para ello, el *Perfil de Comportamiento* se define en base a *Precondiciones* y *Acciones*. Un perfil de comportamiento estará compuesto por un conjunto de *Precondiciones*, las cuales establecen una serie de restricciones a modo de reglas que especifican bajo qué circunstancias se deben de llevar a cabo las *Acciones* del perfil de comportamiento, que también están expresadas como reglas.

Para poder manipular la información, tanto de las ontologías como de los perfiles de comportamiento se utilizará un razonador. Dicho razonador se implementará utilizando Jena y con él también se podrá llevar a cabo inferencia de nuevo conocimiento haciendo uso de la información de contexto obtenida por los sensores junto con las especificaciones de los perfiles de comportamiento establecidos por los usuarios.

Puesto que el rendimiento de SenSE estará estrictamente ligado al uso que los servicios hagan de su capa semántica, se ha tomado la decisión de incluir un gestor semántico al que hemos denominado *SemanticManager*. Este componente será el encargado de gestionar la funcionalidad de la capa semántica y de dotar a los servicios de la capacidad de sensibilidad al contexto. Por lo tanto, la capacidad de representación de la información, razonamiento y comportamiento proactivo de SenSE girará en torno al *SemanticManager*.

En este punto, podemos decir que SenSE dispondrá de al menos los siguientes elementos: SemanticManager, Ontology, BehavioralProfile, Rule y JenaReasoner.

Una vez considerados los distintos elementos necesarios en SenSE se ha procedido a construir el diagrama de clases de análisis de la capa semántica, tal y como

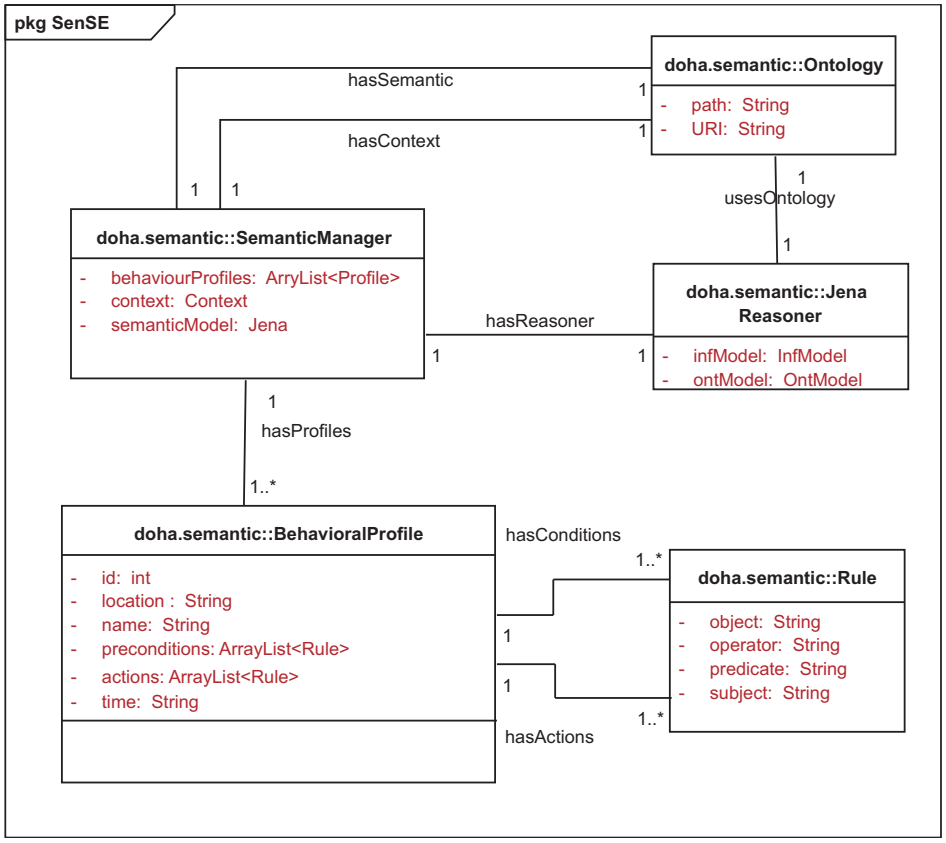


Figura 6.3: Diagrama de clases UML de SenSE

muestra la Figura 6.3. Las clases que se han incorporado a nivel de análisis a SenSE cumplen una finalidad específica y están ahí para cubrir las necesidades derivadas de la incorporación de semántica en la plataforma.

Gracias a este diseño la capa semántica de la plataforma de servicios, es autocontenida y autónoma. Los servicios DOHA podrán o no instanciarla, dependiendo de sus necesidades y requerimientos. Puede que existan servicios que hagan uso de la información semántica para llevar a cabo un comportamiento proactivo y otros cuya funcionalidad sea mucho más simple y no la necesiten. La Figura 6.4 representa el diagrama de clases final y completo de la plataforma de servicios, incluyendo a SenSE, donde además se puede observar cómo se enlazan ambas capas para satisfacer este requisito.

6.5.1. Definición de un Servicio Semántico Sensible al Contexto SenSE

Al incorporar propiedades semánticas a los servicios de la plataforma se debe adecuar la definición formal de servicio que teníamos hasta ahora (Capítulo 5, Sección 5.2.1). Para ello, en primer lugar vamos a definir que es un servicio semántico y

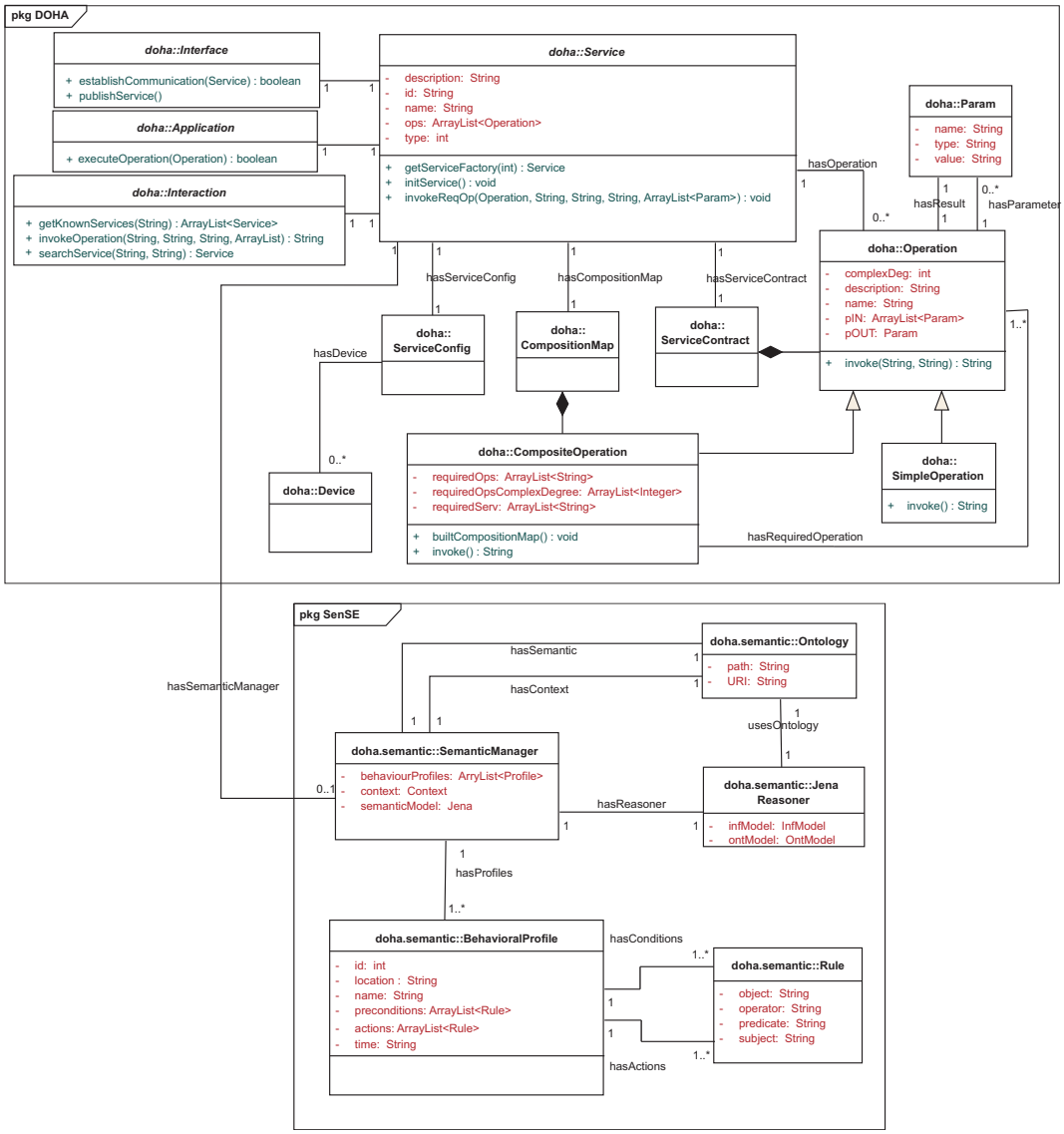


Figura 6.4: Diagrama de clases UML de DOHA + SenSE

qué es un servicio sensible al contexto. Ambos son complementarios y no excluyentes. La tipología del servicio dependerá de la naturaleza del componente semántico en cada caso, tal y como muestra la Figura 6.5. Puede que un servicio semántico no sea sensible al contexto y viceversa. Y puede que tengamos servicios semánticos sensibles al contexto.

Consideramos un servicio semántico, *Semantic Service* (SS), como aquel que representa las propiedades funcionales del servicio, así como aquellas propiedades no funcionales relacionadas con su funcionamiento, en base a una ontología. Esto es, por ejemplo, las características QoS que pueden delimitar que sea seleccionado entre un conjunto de instancias en tiempo de ejecución.

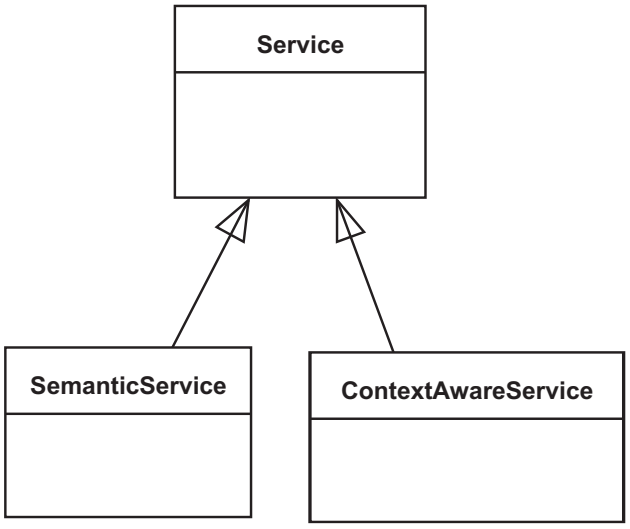


Figura 6.5: Tipos de servicios según la naturaleza de su componente semántica

Por otro lado, un servicio sensible al contexto, *Context Aware Service* (CAS), es aquel que considera la información no funcional del servicio para llevar a cabo un comportamiento proactivo sobre el entorno. Para ello también considera una ontología que representa semánticamente la información que el servicio es capaz de manejar e interpretar. Se entiende como *comportamiento proactivo* aquellas acciones que el servicio lleva a cabo sobre el entorno sin la intervención del usuario. Por ello, para los servicios sensibles al contexto no es suficiente con representar semánticamente la información del entorno a través de una ontología de contexto, sino que es necesario además relacionar dicha información con su funcionalidad. Esto permitirá a los servicios llevar a cabo un comportamiento proactivo, es decir, actuar sobre el entorno sin la intervención explícita de los usuarios. Para ello vamos a definir *perfiles de comportamiento*. Estos perfiles establecen bajo qué condiciones del entorno el servicio debe ejecutar sus operaciones proactivamente.

Puede ser que un servicio sea semántico y sensible al contexto, en cuyo caso se le denominará *Semantic Context Aware Service* (SCAS).

La ontología o conjunto de ontologías que determinan la semántica en este tipo de servicios son un elemento fundamental que debe ser incorporarlo a la definición de servicio semántico y sensible al contexto. En el caso de que el servicio posea un comportamiento proactivo, es decir, sea sensible al contexto, cobran también importancia los perfiles de comportamiento. Este conjunto de reglas serán configuradas por los usuarios y determinarán el comportamiento autónomo del servicio en tiempo de ejecución.

Por lo tanto, ambos conceptos se han considerado para especificar la definición formal de servicio semántico, Definición 6.1, y de servicio sensible al contexto, Definición 6.2. En ambas definiciones se puede observar cómo ambos tipos de servicios extienden la definición inicial de servicio para incorporar a ella la ontología u onto-

logías que determinan la semántica del servicio, Ont_i, y los perfiles que determinan el comportamiento proactivo del servicio, Bp_i por sus siglas en inglés *Behavioral Profiles*, en el caso del CAS. En caso de que se trate de servicio semántico y sensible al contexto, se empleará la Definición 6.2, siendo el componente Ont_i de la definición un conjunto de ontologías. Cuando los desarrolladores de aplicaciones deseen incorporar semántica a sus servicios optarán por estas definiciones de servicio para llevar a cabo su especificación.

$$SService_i =< Id_i, Ps_i, Ip_i, Ir_i, At_i, Ont_i > \tag{6.1}$$

$$CAService_i =< Id_i, Ps_i, Ip_i, Ir_i, At_i, Ont_i, Bp_i > \tag{6.2}$$

La extensión del concepto de servicio afecta también a la estructura del servicio. Aunque la estructura de capas compuesta por interfaz, aplicación e interacción se mantiene, los elementos externos que estas capas van a emplear se verán extendidos. Así, además del contrato de servicio, mapa de configuración y archivo de configuración, se unen ahora las ontologías y los perfiles de comportamiento, como elementos adicionales necesarios para llevar a cabo la funcionalidad del servicio semántico sensible al contexto. De este modo, la Definición 5.2 que se presentaba también en el Capítulo 5, Sección 5.2.1) en base a estos elementos ligados a la anatomía de servicio, se ve extendida al considerar la semántica y se presenta en las Definiciones 6.3 y 6.4.

$$\begin{aligned} SService\_Anatomy_i \;=\; & < contract(S_i), \\ & map(S_i), \\ & config(S_i), \\ & ontology(S_i) > \end{aligned} \tag{6.3}$$

$$\begin{aligned} CAService\_Anatomy_i \;=\; & < contract(S_i), \\ & map(S_i), \\ & config(S_i), \\ & ontology(S_i), \\ & behavioral\_profile(S_i) > \end{aligned} \tag{6.4}$$

6.6. Implementación de SenSE

6.6.1. Servicios Semánticos

Para convertir los servicios de la plataforma en servicios semánticos, SS, se va a emplear OWL-S [W3C, 2014a]. OWL-S proviene de un proyecto promovido desde el W3C para fomentar la estandarización en la utilización la semántica en la Web. De

este modo, SenSE utilizará el paradigma de programación de los SS para facilitar la automatización de los procesos de descubrimiento, composición, invocación y monitorización de los servicios.

Actualmente el procedimiento de descubrimiento de servicios en SenSE es dinámico, pero no la colaboración entre servicios. Cuando los servicios están disponibles en la plataforma se anuncian en un registro de servicios (Directorio de servicios), para que otros puedan descubrirlos e interaccionar con ellos, pero cada servicio preestablece con qué otros servicios de la plataforma va a interactuar de forma estática. Al considerar OWL-S para la conversión de los servicios en SS, SenSE posibilita la construcción de un modelo de colaboración dinámica basado en semántica.

Para convertir un servicio de DOHA a un servicio SS partimos de su contrato de servicio DOHA. El contrato de servicio de un servicio DOHA establece cuál es la funcionalidad del servicio de una forma estrictamente sintáctica. Al convertir dicho servicio en SS, se llevará a cabo un proceso de conversión del contrato de servicio en WSDL a OWL-S, para lo cual se han utilizado sus tres ontologías, ya descritas con anterioridad, *service profile*, *process model* y *service grounding* [Martin et al., 2005]. En resumen, el perfil se utiliza para expresar "lo que hace un servicio", a efectos de publicación y solicitudes de funcionalidad frente al servicio. El modelo de proceso describe "cómo funciona", con el objetivo de permitir la invocación, colaboración, composición, y seguimiento de los servicios. Y, por último, la base grounding especifica los detalles a tener en cuenta en la utilización de los servicios, como por ejemplo los formatos de mensaje o protocolos a seguir.

Para llevar a cabo la conversión de un servicio a SS en base a las ontologías de OWL-S se ha utilizado Jena. Como paso previo para la construcción de las ontologías OWL-S de cada servicio es necesario identificar los componentes del servicio involucrados en la ontología a partir de su contrato de servicio en WSDL, como son WSDLTypes, WSDLMessage, WSDLportTypes, WSDLbinding y WSDLService. Una vez organizados los componentes se crean las ontologías de cada servicio a partir de las fuentes de OWL-S y utilizando el plugin java basado en Jena.

Además de la información funcional que puede extraerse del contrato de servicio en WSDL al generar las ontologías OWL-S, hay más información en la definición de los servicios DOHA a considerar para su inclusión como información semántica. Es el caso del grado de complejidad de las operaciones de los servicios. Dado que esta información es relevante para su ejecución y permite a los desarrolladores valorar la capacidad colaborativa de los servicios, también será incorporada al perfil del servicio en OWL-S. Para ello se inspeccionará el mapa de composición y se etiquetará el grado de complejidad asociado a cada una de las operaciones del servicio. También se puede extraer información semántica interesante del archivo de configuración, como es el caso de las propiedades de calidad de servicio que también se incorporarán al perfil del servicio en OWL-S. En el fragmento de Código 6.1 se muestra un fragmento de la ontología Profile.owl de un servicio semántico con propiedades de calidad de servicio. En él se muestra cómo se definen las propiedades *hasRestrictionMaxAnswerTime_Ri* y *hasRestrictionMaxTimeExecution_Ci*, y cómo se les da valor para la instancia del servicio concreta a través de su ontología OWL-S

Código 6.1: Fragmento de código de la ontología Profile.owl de un Servicio Semántico de ejemplo

```
1  ...
2  <rdf:Description rdf:about="http://localhost/SEMANTIC/
       ServicioSensorHumedad2_Profile.owl#hasRestrictionMaxAnswerTime_Ri">
3      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"
           />
4      <rdfs:domain rdf:resource="http://localhost/SEMANTIC/
           ServicioSensorHumedad2_Profile.owl#QoS"/>
5      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
           ObjectProperty"/>
6  </rdf:Description>
7  <rdf:Description rdf:about="http://localhost/SEMANTIC/
       ServicioSensorHumedad2_Profile.owl#
       hasRestrictionMaxTimeExecution_Ci">
8      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"
           />
9      <rdfs:domain rdf:resource="http://localhost/SEMANTIC/
           ServicioSensorHumedad2_Profile.owl#QoS"/>
10     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
           ObjectProperty"/>
11 </rdf:Description>
12 ...
13 <rdf:Description rdf:about="http://localhost/SEMANTIC/
       ServicioSensorHumedad2_Profile.owl#
       ServicioSensorHumedad2_ProfileQoS">
14     <j.0:hasRestrictionMaxAnswerTime_Ri rdf:datatype="http://www.w3.org
           /2001/XMLSchema#string">1</j.0:hasRestrictionMaxAnswerTime_Ri>
15     <j.0:hasRestrictionMaxTimeExecution_Ci rdf:datatype="http://www.w3.
           org/2001/XMLSchema#string">2</j.0:
           hasRestrictionMaxTimeExecution_Ci>
16     <rdf:type rdf:resource="http://localhost/SEMANTIC/
           ServicioSensorHumedad2_Profile.owl#QoS"/>
17 </rdf:Description>
```

6.6.2. Ontología de contexto

Durante el desarrollo de SenSE se han analizado las distintas ontologías existentes para representar la información de contexto relativa a sistemas de inteligencia ambiental. Previamente, en el Capítulo 2, se presentaron las características más representativas de las mismas. Desde nuestro punto de vista creemos que es viable considerar la reutilización de ontologías en el ámbito de la computación ubicua, pues facilita el trabajo de los desarrolladores que pueden utilizar una ontología ya testeada, considerando todos los conceptos que soporta, su taxonomía y sus axiomas. Por lo tanto, si una ontología existente para nuestro dominio de aplicación funciona y se adecúa a nuestras necesidades, podemos utilizarla, no es necesario reinventar la rueda. De entre las ontologías analizadas se ha seleccionado para ser la ontología de referencia en SenSE la propuesta por Preuveneers et al. [Preuveneers et al., 2004] y denominada CoDAMoS.

La ontología CoDAMoS, acrónimo de Context-Driven Adaptation of Mobile Services, es el resultado de un ambicioso proyecto de investigación titulado con el mismo nombre [CoDAMoS, 2004]. En dicho proyecto participaron cuatro grupos de investigación diferentes tratando de resolver los principales desafíos de la Inteligencia Ambiental. Entre ellos, estaba la forma de representar el contexto en un sistema AmI. Así nació la ontología CoDAMoS, con el objetivo de representar los conceptos y definiciones relativos al contexto de un sistema de Inteligencia Ambiental.

Para ello CoDAMoS utiliza distintos dominios de conocimiento en torno al concepto de contexto. Uno de los elementos principales en CoDAMoS es el usuario, representado por la clase *User*. En la Figura 6.6 podemos ver sus relaciones en la ontología. La clase *User* posee cuatro relaciones distintas con cuatro clases diferentes. Dichas relaciones caracterizan las distintas instancias de la clase *User* que se creen en base a la ontología. De este modo, todo usuario representado en CoDAMoS tendrá un rol, *Role*; realizará una tarea, *Task*; utilizando un dispositivo determinado, *IODevice*; y siempre considerando su estado de ánimo, *Mood*.

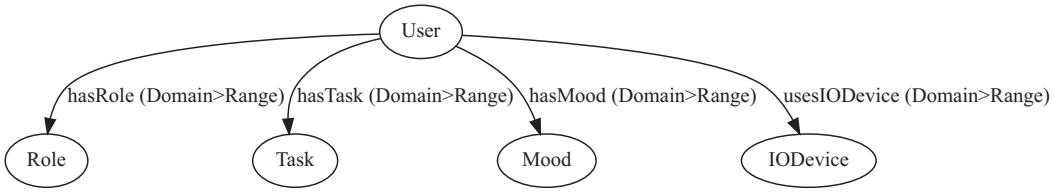


Figura 6.6: Ontología CoDAMoS: User

Las tareas, *Task*, que realizan los usuarios, *User*, están caracterizadas en CoDAMos por dos relaciones diferentes. Toda tarea poseerá un conjunto de actividades, *Activity*, y se llevará a cabo haciendo uso de un servicio determinado *Service*. En la Figura 6.7 podemos visualizar las relaciones de la clase *Task* en CoDAMoS.

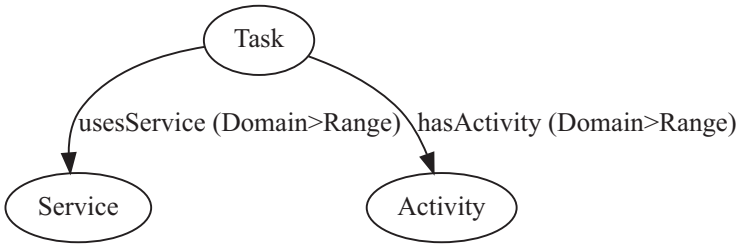


Figura 6.7: Ontología CoDAMoS: Task

Los servicios, *Service*, que realizan las tareas especificadas por los usuarios están caracterizados en CoDAMoS en base a las especificaciones establecidas por OWL-S. De este modo, todo *Service* poseerá *ServiceProfile*, *ServiceModel* y *ServiceGrounding*, como podemos ver en la Figura 6.8. Así, la ontología además de cubrir las necesidades relativas al contexto para dotar a los servicios de SenSE de *context-*

awareness, también aporta el conocimiento necesario para convertir a los servicios en SS, como ya se mencionó en la sección 6.6.1.

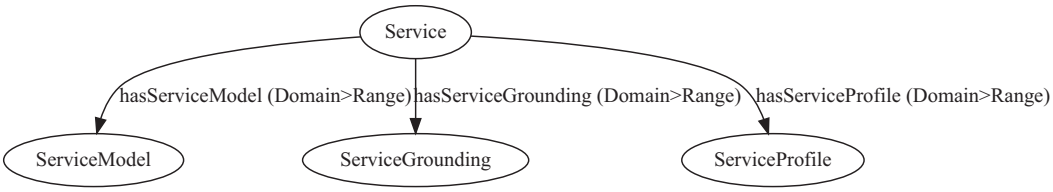


Figura 6.8: Ontología CoDAMoS: Service

Los elementos de la clase *Service* en CoDAMoS forman parte de la plataforma, *Platform*, que representa al sistema de inteligencia ambiental representado por la ontología en su conjunto, por lo que cobra un mayor protagonismo. A partir de las relaciones existentes en la clase *Platform* se establece qué tipo de *Software* y *Hardware* proporciona el sistema, así como el entorno, *Environment*, en el que dicha plataforma se está ejecutando. Como vemos en la Figura 6.9, la clase *Software* requiere además la clase *Platform*, existiendo una relación de reciprocidad entre ambas clases.

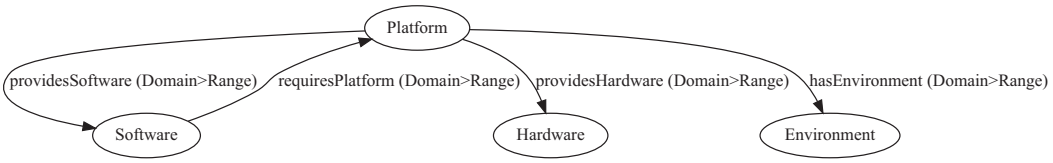


Figura 6.9: Ontología CoDAMoS: Platform

El entorno, *Environment*, en el que se ejecuta la plataforma establece las condiciones que el usuario interpretará como "contexto", como son tiempo, *Time*; localización, *Location*; y demás condiciones ambientales, como presión, humedad, luminosidad, temperatura o ruido, *Pressure*, *Humidity*, *Lighting*, *Temperature* y *Noise*. La Figura 6.10 representa cómo se relacionan estas clases entre sí. Como se puede apreciar en dicha figura, las distintas condiciones ambientales recogidas en CoDAMoS forman parte de la ontología como subclases de la clase *EnvironmentalCondition*, por lo que podrían añadirse fácilmente nuevas características ambientales si fuese necesario.

Con respecto a la clase *Software*, puede ser de tipo *Middlware*, *Library*, *VirtualMachine*, *RenderingEngine* u *OperativeSystem*. A su vez, la clase *Software* tiene relación con las clases *Platform* y *Service*. Como ya se mencionó al analizar la clase *Platform*, entre ambas existe una necesidad bidireccional, la clase *Platform* provee *Software*, y la clase *Software* requiere de *Platform*. En la Figura 6.11 podemos ver con detalle todas estas relaciones.

Al igual que la clase *Software* puede ser de distintos tipos, la clase *Hardware* puede ser de tipo *IODevice* o *Resource*, relación que se establece en la ontología

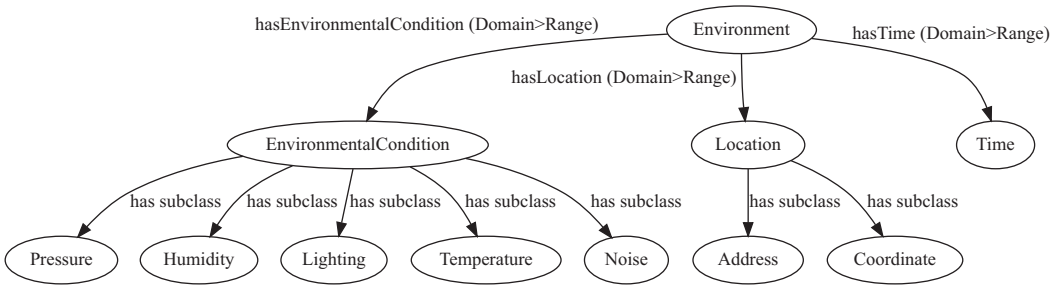


Figura 6.10: Ontología CoDAMoS: Environment

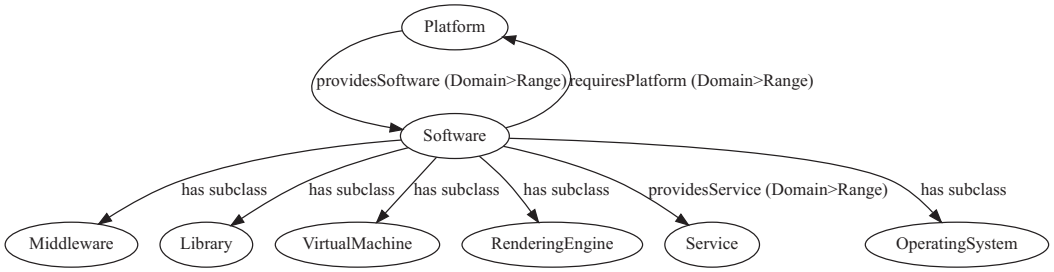


Figura 6.11: Ontología CoDAMoS: Software

mediante relaciones de subclase, como se puede ver en la Figura 6.12. A su vez, los elementos de la la clase *IODevice*, se reconocen como dispositivos de entrada o de salida, y se distinguen a través de las clases *InputDevice* u *OutputDevice*.

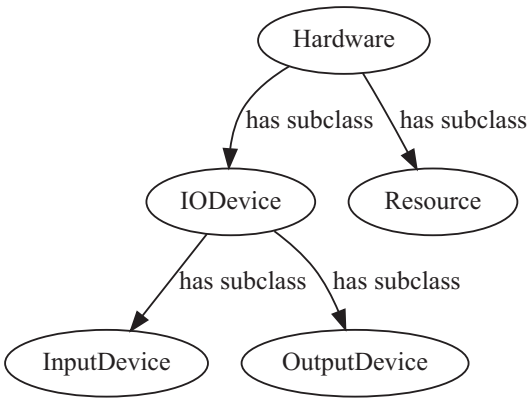


Figura 6.12: Ontología CoDAMoS: Hardware

Los distintos recursos hadware gestionados por la ontología son muy diversos, por lo que analizaremos de forma independiente las clases y relaciones derivadas de la clase *Resource*, representada en la Figura 6.13. Los distintos tipos de recursos están representados en la figura por las subclases de *Resource*, que son *PowerResour-*

ce, *MemoryResource*, *StorageResource*, *CPUResource* y *NetworkResource*. La clase *CPUResource* posee además otras tres relaciones funcionales que caracterizan a los elementos de ese tipo y que establecen los valores que pueden tomar en función de su tipo. Así, un elemento de la clase *CPUResource* poseerá un valor *branchPrediction* e *instructionSet*, ambos de tipo *String*; y un valor de *cpuCache* de tipo *Integer*. También la clase *NetworkResource* posee características específicas y los elementos de esta clase poseerán un valor de *latency* y de *reliability*, ambos de tipo *Integer*.

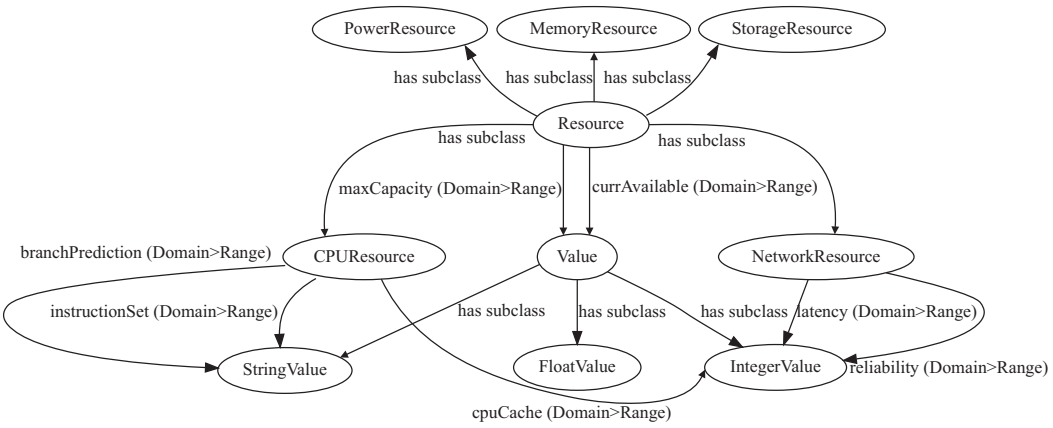


Figura 6.13: Ontología CoDAMoS: Resource

Paralelamente a estas clases, CoDAMoS define la clase *Profile* que permite especificar a través de la ontología cuáles son las preferencias de los usuarios. En la Figura 6.14 pueden observarse la relación funcional entre la clase *Profile* y la clase *ProfileItem*, así como que la clase *ProfileItem* puede ser a su vez de tipo *PreferenceItem*, restricción establecida por la relación de subclase entre ambas. La clase *PreferenceItem* posee además dos relaciones funcionales que permiten establecer las preferencias de los usuarios a través de su instanciación, como son *prefCondition*, *prefProperty* y *PrefValue*. Sin duda, esto nos recuerda a las reglas de comportamiento basadas en reglas que hemos mencionado con anterioridad.

Por último, CoDAMoS también permite representar la versión de la representación del contexto realizada en la ontología a través de la clase *Version*, 6.15.

Puesto que se va a emplear una ontología ya definida y verificada no vamos a llevar a cabo el análisis formal de su validez. Se considera que las propiedades transitivas que la ontología define sobre las distintas clases son válidas y suficientes para la representación del contexto en un entorno ubicuo.

6.6.3. Perfiles de Comportamiento y Reglas Lógicas

Para que el sistema sea capaz de desarrollar una actitud proactiva debe poseer algún elemento de donde inferir que su actuación es adecuada en un momento concreto sin ser intrusivo para los usuarios. Este elemento que indica a los servicios cuando actuar son los perfiles de comportamiento. En estos perfiles se indican, me-

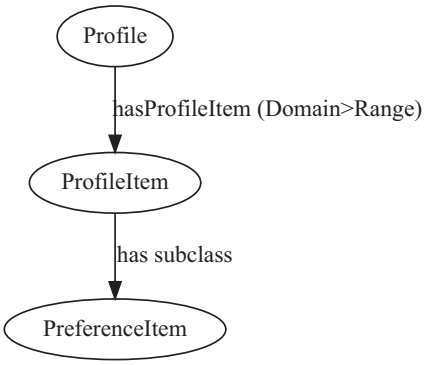


Figura 6.14: Ontología CoDAMoS: Profile

Figura 6.15: Ontología CoDAMoS: Version

diante reglas lógicas, las situaciones ante las que el servicio puede llevar a cabo una determinada operación sin la solicitud previa de esta por parte del usuario.

En relación con un espacio domótico ubicuo, estos perfiles de comportamiento estarán ligados a las preferencias de los usuarios sobre el estado de su entorno y las restricciones de los propios dispositivos. A través de estos perfiles, los usuarios pueden determinar sus preferencias con respecto al comportamiento que están definiendo en el servicio sensible al contexto en cuanto a iluminación, temperatura o seguridad, entre otros. De esta forma, una vez implementado el perfil del comportamiento el usuario se "desentiende" del servicio sensible al contexto, dado que ya puede por si solo ser capaz de interpretar sus deseos y adecuar su entorno para que le sea lo más confortable posible.

Para desarrollar los perfiles de comportamiento de la plataforma en base a reglas lógicas se podría utilizar un modelo ya existente o bien diseñar un modelo propio de perfiles en el que se introducen "manualmente" las reglas lógicas a aplicar sobre las ontologías. Esta segunda opción tiene un pequeño inconveniente. Las reglas de los perfiles de comportamiento quedarán ligadas con las clases y relaciones de las ontologías, de forma que los cambios en la ontología afectarán a los perfiles de comportamiento. Como contrapartida, el diseño particular de la estructura de reglas es más ligero que utilizar un framework específico que lo lleve a cabo.

En el caso de SenSE hemos optado por diseñar un modelo propio de perfiles de comportamiento. Un perfil de comportamiento en SenSE estará formado por reglas. Estas reglas indican las condiciones establecidas por el usuario sobre su entorno o incluso por los propios dispositivos sobre su funcionamiento. Para construir las reglas se seguirá la nomenclatura de las ternas RDF. Es decir, cada regla estará formada por una terna $sujeto - predicado - objeto$, como se muestra en la Figura 6.16, junto

con un operador de comparación, *greaterThan*, *lessThan* o *Equal*. El operador de comparación se aplicará para comparar el significado de la regla con el estado real del entorno. De dicha comparación el servicio puede deducir si es necesario o no llevar a cabo una acción sobre el entorno.

Los sujetos de las ternas RDF de cada regla están directamente relacionados con los elementos del entorno representados como instancias de las clases de la ontología de contexto CODAMOS como por ejemplo "luminosidad de esta habitación". En este caso el sujeto está explícitamente determinado por "esta habitación", pero también podría ser más genérico, como "luminosidad actual". En este caso el sujeto existe, pero no de forma explícita y es susceptible a interpretación. Podría tratarse de la luminosidad de la habitación en la que se encuentre el usuario, o bien del edificio. Por ello, y para evitar ambigüedades, para identificar los elementos del entorno relativos a la ontología de contexto a partir de la terna RDF de cada regla se utilizará tanto el sujeto como el predicado, por ejemplo "estaLuz - luminosidad". Por último, el *objeto* de la terna RDF de cada regla especificará el valor concreto que se espera tenga el elemento del entorno identificado por el sujeto y el predicado. El valor del *objeto* podrá ser un literal u otro elemento del dominio de conocimiento establecido por la ontología de contexto.

Código 6.2: Instancias de la ontología de contexto utilizadas en una regla simple

```
1 <!-- http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1 -->
2 <owl:NamedIndividual rdf:about="http://core.ugr.es/doha/onto/codamos/
      Context.owl#LightD1">
3         <rdf:type rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#EnvironmentalCondition"/>
4         <rdf:type rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#Lighting"/>
5         <rdf:type rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#OutputDevice"/>
6         <hasLocation rdf:resource="http://core.ugr.es/doha/onto/codamos
              /Context.owl#ThisRoom"/>
7 </owl:NamedIndividual>
8
9 <!-- http://core.ugr.es/doha/onto/codamos/Context.owl#ThisRoom -->
```

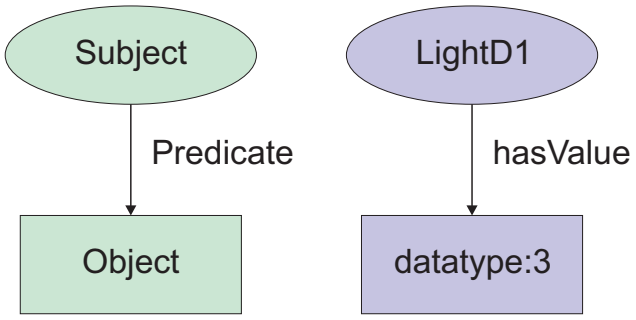


Figura 6.16: (a) Tripleta RDF genérica. (b) Ejemplo de tripleta RDF en SenSE.

```
10 <owl:NamedIndividual rdf:about="http://core.ugr.es/doha/onto/codamos/
       Context.owl#ThisRoom">
11         <rdf:type rdf:resource="http://core.ugr.es/doha/onto/codamos/
               Context.owl#Location"/>
12         <hasEnvironmentalCondition rdf:resource="http://core.ugr.es/
               doha/onto/codamos/Context.owl#LightD1"/>
13     </owl:NamedIndividual>
```

Las reglas de un perfil de comportamiento se estructuran en base a precondiciones o estados deseables y acciones a llevar a cabo cuando alguna de las precondiciones no se cumple. Si una regla como precondición no se satisface, habrá una regla como acción que determinará cómo debe actuar el servicio en consecuencia. Para determinar si las reglas contenidas en una precondición se cumplen o no, los servicios deben consultar la información del entorno, es decir, deben actualizar la información real de su contexto. Si la precondición no se cumple, es decir, las reglas no se corresponden con la información real del contexto, entonces se activan las acciones del perfil de comportamiento. Activar la acción supone llevar a cabo una operación sobre el entorno, dando lugar al comportamiento consciente del contexto de las entidades involucradas.

Formalización de los perfiles de comportamiento y su relación con la metodología de Data Quality. Para construir un perfil de comportamiento en SenSE se debe utilizar la ontología concreta que se ha seleccionado para representar el contexto de la plataforma, que ha sido CODAMOS. La construcción de las reglas se hará en base a las clases, relaciones e instancias de dicha ontología. La organización de reglas en precondiciones y acciones establecerá el comportamiento que desarrollará el servicio ante cambios en el entorno. Cuando se interprete la información del entorno, si no se comprueba la veracidad de las precondiciones del perfil, puede llevarse a cabo la acción, que conllevará una operación sobre el entorno. Se espera que, si el sistema funciona correctamente, una vez se ha activado un perfil de comportamiento, el estado del entorno satisfaga las precondiciones de dicho perfil.

En este sentido, la utilidad de los perfiles de comportamiento es doble, no solo permite llevar a cabo un comportamiento proactivo sobre el entorno, sino que además representa una forma de aplicar una metodología de evaluación del sistema basada en Data Quality. Que un perfil de comportamiento se active debido al incumplimiento de sus precondiciones, significa que el sistema no satisface una serie de restricciones. Del mismo modo, tras la ejecución de las acciones asociadas al perfil de comportamiento, el estado del sistema debe ser el determinado por las precondiciones. Si es así, las operaciones involucradas en las acciones del perfil funcionan adecuadamente y por tanto, los datos se consideran correctos. Si por el contrario, tras la ejecución de las operaciones del perfil el estado del sistema no satisface las precondiciones, la evaluación basada en *Data Quality* habrá fallado y será necesario determinar el motivo del funcionamiento erróneo del sistema.

Ejecución de múltiples perfiles de comportamiento. La existencia de múltiples perfiles de comportamiento puede llevar asociado contradicciones entre las precondiciones que afecten al comportamiento proactivo del servicio. Para evitar estos conflictos proponemos a los desarrolladores diseñar los perfiles de comportamiento tal y como en robótica se llevan a cabo la programación de los robots en base al concepto de "Behavior Programming" Lejos [2014].

La evaluación de los perfiles de comportamiento forma parte de las etapas del procesamiento de la información de contexto (Sección 6.4). Así, cada vez que se lleve a cabo este proceso, el servicio evaluará secuencialmente el conjunto de perfiles de comportamiento que tenga definidos. Esto significa que evalúa el primer perfil y, si se activa, es decir, no se satisfacen sus precondiciones, el servicio lleva a cabo las operaciones incluidas en la acción. Una vez ejecutadas las acciones del perfil, el servicio dará por concluido su comportamiento proactivo hasta que se vuelva a ejecutar otra iteración del procesamiento de la información de contexto.

Esto implica, que para que el comportamiento proactivo del servicio sea lo más adecuado posible se deben definir los perfiles de comportamiento en base a su generalidad. Primero se definirían las excepcionalidades y por último los casos más generales. Dependiendo de donde esté en la lista completa, un perfil de comportamiento tendrá mayor o menor prioridad de ejecución. Además, los perfiles deben estar bien definidos y no contradecirse con los que se hayan definido anteriormente.

Así, las condiciones para la definición y ejecución de los perfiles de comportamiento asociados a los servicios SenSE son:

- El servicio solo puede activar un perfil de comportamiento en una iteración del procesamiento de la información de contexto.
- Cada perfil de comportamiento tiene una prioridad fija determinada por su orden en el conjunto de perfiles. El primer perfil de la lista tiene la máxima prioridad. El último perfil tiene la mínima prioridad.
- Cada perfil de comportamiento debe definir adecuadamente sus precondiciones de activación.
- Una vez que se active un perfil de comportamiento su ejecución es prioritaria ante cualquier otra acción del servicio.

Un ejemplo de conflicto que resuelve esta forma de diseñar los perfiles de comportamiento es la existencia de distintos usuarios de un mismo servicio. Si cada usuario define su perfil de comportamiento deseado, la prioridad de su perfil vendrá determinada por sus privilegios en el sistema. Por ejemplo, imaginemos un servicio de confort en el hogar. Todos los habitantes de la casa definirán sus preferencias en cuanto a confort. Cada habitante tendrá un nivel de privilegios que, supongamos, estará determinado por la edad de cada uno, considerando un individuo adicional como "visitante" que tendrá el mínimo de privilegios posible. La evaluación de los perfiles se llevaría a cabo, tal y como muestra la Figura 6.17, en función de la prioridad de los individuos que los han definido.

Consideraciones acerca de la información de un perfil de comportamiento. Las reglas contenidas tanto en las precondiciones como en las acciones de los perfiles de comportamiento emplean la información con la que cuenta el servicio. Esta información puede estar relacionada con los datos que el propio servicio posee del entorno, con sus características o bien con su funcionalidad, es decir, con sus operaciones.

La información que el servicio posee del entorno estará determinada por los sensores/actuadores hardware que controle, en caso de ser un servicio dispositivo, o bien por la información que obtiene a través de la colaboración con otros servicios, en caso de ser un servicio compuesto. Por lo tanto, un servicio dispositivo que controle la luminosidad conocerá el valor de esta propiedad del contexto a través de los sensores que encapsula, mientras que un servicio de gestión del confort también conocerá esta propiedad del contexto, pero a través de la interacción con el servicio dispositivo correspondiente. Esta interacción se llevará a cabo a través de las operaciones compuestas del servicio.

Del mismo modo, cuando las precondiciones del perfil de comportamiento de un servicio no se satisfagan, el servicio ejecutará las acciones. Estas acciones formarán parte del conjunto de operaciones del servicio determinado en su definición (Definición 5.1) y pueden ser operaciones simples o compuestas, en caso de que no se trate de un servicio dispositivo. En caso de que la acción de un perfil de comportamiento involucre a una operación compuesta, la activación del perfil desencadenará la colaboración con otros servicios para satisfacer sus precondiciones y que el estado del contexto sea el adecuado.

Ejemplo de perfil de comportamiento. A continuación se presenta un ejemplo de perfil de comportamiento en el que se observa la estructura de etiquetas XML correctas. En el ejemplo se va a mostrar un perfil asociado a un servicio dispositivo

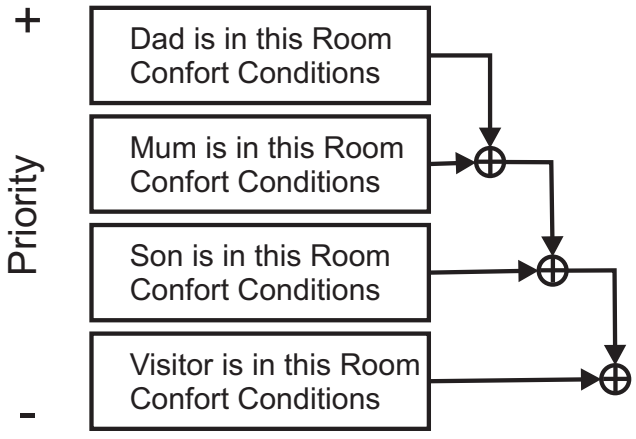


Figura 6.17: Evaluación de un conjunto de perfiles de comportamiento en base a prioridades

Tabla 6.1: Reglas del perfil de comportamiento del Servicio Regulador de Iluminación (SRL)

Type	Subject	Predicate	Operator	Object
Pre	ThisRoom	hasEnvironmentalCondition	Equal	LightD1
Pre	LightD1	hasValue	GreaterThan	2
Action	LightD1	OperationTurnOnLight	Equal	ON
Action	LightD1	OperationSetLight	Equal	3

sencillo denominado *Servicio Regulador de Luminosidad*, SRL. Dicho servicio controla un dispositivo físico del entorno, *LightD1*, sobre el que se actuará en caso de que se satisfagan las precondiciones relativas a luminosidad del entorno establecidas por el perfil.

Primero instanciamos el servicio en la ontología, su funcionalidad y, en este caso, los dispositivos hardware que utiliza también. En el fragmento de Código 6.3 se puede observar cómo se establecen estas relaciones a través de las clases y métodos transitivos establecidos por la ontología de contexto CODAMOS. En el fragmento de Código 6.3 ya se mostró la instancia de LightD1, por lo que lo omitimos aquí.

Código 6.3: Instancia de la ontología de contexto utilizada para definir un servicio dispositivo sencillo

```
1 <!-- http://core.ugr.es/doha/onto/codamos/Context.owl#SRL -->
2 <owl:NamedIndividual rdf:about="http://core.ugr.es/doha/onto/codamos/
      Context.owl#SRL">
3         <rdf:type rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#Service"/>
4         <usesIODevice rdf:resource="http://core.ugr.es/doha/onto/
              codamos/Context.owl#LightD1"/>
5         <hasTask rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#OperationSetLight"/>
6         <hasTask rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#OperationTurnOffLight"/>
7         <hasTask rdf:resource="http://core.ugr.es/doha/onto/codamos/
              Context.owl#OperationTurnOnLight"/>
8 </owl:NamedIndividual>
```

Una vez se ha definido el servicio y su funcionalidad, se puede crear el perfil de comportamiento que enlace la información del contexto, en este caso luminosidad, con la funcionalidad del servicio.

El perfil está compuesto por dos reglas precondición y dos reglas acción. SenSE contrastará la precondición con la información del entorno, y cuando la luminosidad de la localización en la que se encuentre el servicio sea inferior a 2 no se verificará la regla, y se ejecutará la acción. Las reglas de la acción son también dos, lo que implica que la ejecución del perfil conlleva la ejecución de dos acciones sobre el entorno, establecer a *on* el estado del dispositivo, en este caso esto implica encender la luz *LightD1*; y dar un valor concreto a su propiedad principal, en este caso establecer la luminosidad de la luz a 3.

A continuación, en el Código 6.4, se puede observar cómo se pueden mapear las reglas anteriores en código XML interpretable por SenSE. La estructuración en etiquetas permite delimitar las reglas como *precondition* o *action*. Dentro de cada regla se define la tripleta RDF mediante las etiquetas *subject*, *predicate* y *object*, y el operador de comparación a emplear que viene determinado por la etiqueta *operator*.

Código 6.4: Perfil de comportamiento del Servicio Regulador de Iluminación (SRL)

```
1  <?xml version="1.0"?>
2  <behavioralProfile>
3    <precondition>
4      <rule>
5        <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#
             ThisLocalion</subject>
6        <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
             hasEnvironmentalCondition</predicate>
7        <operator>Equal</operator>
8        <object>LightD1</object>
9      </rule>
10     <rule>
11       <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1
             </subject>
12       <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
             hasValue</predicate>
13       <operator>GreaterThan</operator>
14       <object>2</object>
15     </rule>
16   </precondition>
17   <action>
18     <rule>
19       <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1
             </subject>
20       <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
             OperationTurnOnLight</predicate>
21       <operator>Equal</operator>
22       <object>ON</object>
23     </rule>
24     <rule>
25       <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1
             </subject>
26       <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
             OperationSetLight</predicate>
27       <operator>Equal</operator>
28       <object>3</object>
29     </rule>
30   </action>
31 </behavioralProfile>
```

A través de las reglas de este perfil de comportamiento los servicios que estén influenciados por la luminosidad, como un servicio encargado de la regulación de la iluminación o bien un servicio de más alto nivel que utilice como parte de su comportamiento el estado de la luminosidad en el entorno, puede interpretar que la iluminación del dispositivo físico *LightD1* debe ser siempre superior a 2, que se

representaría en forma de pseudocódigo como se muestra en el Código 6.5. Puesto que la evaluación de las reglas se realiza en negativo, para que se determine la activación del perfil, estas se representarían negando el predicado, tal y como se muestra en el fragmento de código.

Código 6.5: Interpretación del Perfil de Comportamiento

```
1 IF <ThisLocation> !<hasEnvironmentalCondition> <Lighting> ||
2    <LightD1> <LessThan> 3 ] THEN
3   [ <LightD1> <OperationTurnOn> <on> ]
4   [ <LightD1> <OperationSetLight> <3> ]
5 END
```

Con la interpretación del perfil como tripletas RDF se obtiene que la precondición está formada por dos reglas.

$< subject = ThisLocation >$ - $!< predicate = hasEnvironmentalCondition >$ - $< object = Lighting >$

$< subject = ThisLocation >$ - $< predicate = LessThan >$ - $< object = 3 >$

Y por otras dos reglas en la acción:

$< subject = LightD1 >$ - $< predicate = OperationTurnOn >$- $< object = on >$ $< subject = LightD1 >$ - $< predicate = OperationSetLight >$ - $< object = 3 >$

La utilización de este modelo de reglas en SenSE permite representar mediante un esquema sencillo, como es el de etiquetas XML, las relaciones existentes entre la información del entorno físico y su correspondencia con la representación del dominio de conocimiento que se tiene en el sistema. Además, también se representan las acciones que se pueden llevar a cabo de forma automática en base a cambios en la información del entorno. La representación de la información de contexto procesable en reglas basadas en las ontologías del dominio y su estructuración en precondiciones y acciones, juega un papel esencial en el comportamiento proactivo de la plataforma SenSE, por lo que su diseño es importante a la hora de desarrollar aplicaciones con sensibilidad al contexto sobre la plataforma.

6.6.4. Razonamiento e Inferencia

Considerar la semántica como parte directora del diseño de una plataforma basada en servicios implica que esta debe poseer mecanismos suficientes para obtener, interpretar y actuar en función de ella. Estos mecanismos conforman lo que se denomina *Motor de Razonamiento*. Partiendo de unas reglas de inferencia, un motor de razonamiento puede usar los datos de una base de conocimiento, como por ejemplo una ontología, para inferir conclusiones dentro de ese mismo dominio.

Para comprender bien esta sección, debemos aclarar que se empleará el término *inferencia* para referirse al proceso abstracto de obtener información adicional y *razonador* para referirnos al componente que realiza la tarea de inferencia.

Para llevar a cabo el procedimiento de razonamiento sobre la información los servicios deben llevar a cabo una serie de procesos que les permitan (a) obtener información del dominio, (b) interpretar la información obtenida, (c) procesarla e (d)

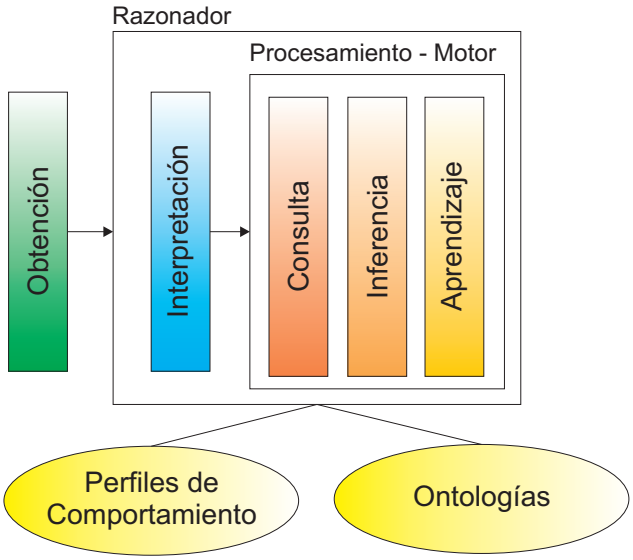


Figura 6.18: Etapas del Proceso de Razonamiento

inferir nueva información útil, como podemos ver en la Figura 6.18. Se denomina a este conjunto de tareas como *Proceso de Razonamiento*, debido a que en su conjunto actúan como un *Razonador* sobre la información obtenida.

Durante el proceso de obtención de información tiene un papel importante la forma de captar la información del entorno real. Por ejemplo, cuando se trate de un servicio de tipo dispositivo, los sensores que este encapsula serán los encargados de captar esta información. Sin embargo, cuando se trate de un servicio general, este tendrá que llevar a cabo la interacción con otros servicios a través de sus operaciones compuestas para poder percibir información del contexto en el que se encuentra. Los datos que toman los elementos receptores pueden venir dados de muy diversas formas, como valores digitales o analógicos, o tramas de bytes. No obstante, no toda la información del mundo real es representativa en un sistema, por lo que es importante que las entidades que captan la información sean receptivas exclusivamente a la información relevante para, cribando y desechando la información innecesaria. La ejecución del proceso de criba de la información útil puede realizarse en este proceso, o bien junto con el proceso siguiente, responsable de llevar a cabo la interpretación de la información.

Una vez que se ha recogido la información del entorno es necesario interpretarla y establecer una relación entre esta información del mundo real y el modelo de representación del conocimiento, en nuestro caso la ontología de contexto CODAMOS. Aquí juega un papel importante la semántica utilizada, la completitud y corrección del modelo y su correspondencia con el dominio de conocimiento real. Una interpretación adecuada de la información del dominio facilita su posterior procesamiento y utilización para generar nuevo conocimiento. Puesto que el modelo de representación de conocimiento está basado en una ontología, en este proceso se establece una

correspondencia entre la información obtenida y su representación ontológica, para que así sea procesable por el sistema y sea realmente de utilidad en los procesos posteriores.

A continuación se lleva a cabo el proceso de razonamiento. Dentro de este proceso se incluyen los procesos de inferencia y aprendizaje, pues el proceso de inferencia de nuevo conocimiento puede considerarse como una parte especializada en la deducción de nueva información. Para ello se emplean los perfiles de comportamiento, que a su vez están vinculados con la ontología de contexto. A partir de la relación entre la información del entorno, su representación mediante la ontología de contexto y las reglas establecidas en los perfiles de comportamiento, se puede inferir qué acción se va a llevar a cabo sobre el entorno. En base a estas acciones, el sistema puede memorizar sus acciones a modo de aprendizaje, para que sea capaz de adaptarse a los cambios del entorno de forma autónoma y sin la necesidad de evaluar las precondiciones de los perfiles de comportamiento.

En el Capítulo 2, Sección 2.5.1.4, ya se introdujo la API de código abierto Jena como un framework que provee un entorno de desarrollo para RDF, RDFS, OWL y SPARQL, además de disponer de inferencia basada en reglas [Apache, 2014a]. En el desarrollo de SenSE se ha considerado Jena como herramienta para llevar a cabo la implementación de la capa semántica y de su motor de razonamiento, ya que incluye una completa API para el manejo de ontologías y soporta el lenguaje OWL. En SenSE utilizaremos los razonadores predefinidos de la API de Jena para manejar RDF y OWL, así como el *Transitive Reasoner*, que implementa las propiedades de transitividad y simetría.

En el fragmento de Código 6.6 se muestra cómo se ha llevado a cabo la implementación del razonador de SenSE. En él puede observarse como cada "pollingPeriod" se lleva a cabo el procesamiento de la información ejecutando las distintas tareas que se han mencionado asociadas a la Figura 6.18.

Código 6.6: Código del razonador implementado en SenSE

```
1  public void run() {
2          int k = 0;
3          boolean action;
4          Object contextObj = null;
5          String logmsj;
6
7          logmsj = ("[semanticManager] SEMANTIC MANAGER RUNNING");
8          Logger.getLogger(SemanticManager.class.getName()).log(Level.
               INFO, logmsj);
9
10         while (running) {
11             try {
12                 logmsj = ("[semanticManager] ---> " + k + " Semantic
                       Manager Iter");
13                 Logger.getLogger(SemanticManager.class.getName()).log(
                       Level.INFO, logmsj);
14                 for ( int j = 0; j < behavioralProfiles.size(); j++) {
15                     BehavioralProfile profile = behavioralProfiles.get(
```

```
                        j);
16                  action = false;
17
18                  // Analysis
19                  for (int i = 0; i < profile.getConditions().size();
                        i++) {
20
21                      // Update context
22                      if (updateContext(profile.getConditions().get(i
                            ))) {
23
24                          // Context information retrieval
25                          contextObj = contextInformationRetrieval(
                                profile.getConditions().get(i));
26
27                          // Reasoning
28                          if (reasoning(profile.getConditions().get(i
                                ), contextObj) == false) {
29                              action = true;
30                          }
31                      }
32                  }
33
34                  // Behavioural profile resolution
35                  if (action) {
36                      for (int i = 0; i < profile.getActions().size()
                            ; i++)
37                          changeContext(profile.getActions().get(i),
                                contextObj);
38                  }
39                  logmsj = ("[semanticManager]        Profile " +
                        profile.getBehavioralProfile() + " revised");
40                  Logger.getLogger(SemanticManager.class.getName()).
                        log(Level.INFO, logmsj);
41              }
42              // Wait polling period
43              System.out.println("[semanticManager] ---> PAUSE");
44              Thread.sleep(1);//pollingPeriod);
45
46          } catch (InterruptedException ie) {
47              // just shut down the thread
48              stop();
49          }
50      }// while running
51      logmsj = ("[contextManager] SEMANTIC MANAGER STOPPED");
52      Logger.getLogger(SemanticManager.class.getName()).log(Level.
            INFO, logmsj);
53  }
```

6.7. Discusión

Aunque pueda parecer antagónico que dispositivos con recursos limitados y poca capacidad de procesamiento sean capaces de captar información, procesarla, razonar sobre ella y actuar autónomamente en consecuencia, en definitiva, mostrar un comportamiento inteligente, creemos que es posible conseguirlo. El desarrollo de la capa semántica SenSE ha sido guiado por un proceso de ingeniería consensuado sobre un orden de actuación. Para ello, a partir de la sensorización y los perfiles de comportamiento definidos por los usuarios, junto con la percepción del contexto del entorno basado en ontologías, se aplica un razonamiento fundamentado en reglas a partir del cual es posible obtener un comportamiento proactivo por parte de los servicios, cuyo resultado final será operar sobre los actuadores de forma inteligente. Todo ello sobre dispositivos ligeros, de pequeño tamaño y que consumen poca energía, embebidos en los objetos cotidianos, para evitar crear escenarios irreales y artificiales, alejados de la visión de la computación ubicua.

Capítulo 7

Metodología de Desarrollo y Caso de Estudio

Many prototypes are developed from scratch and "fade onto oblivion" when the corresponding research project ends. Moreover, it is often impossible to compare different approaches in a fair manner as experiments are incomparable or cannot be reproduced.

· Wil M.P. van der Aalst ·
Business Process Management: A Comprehensive Survey

Abstract: El desarrollo de la plataforma de servicios semánticos sensibles al contexto se ha llevado a cabo siguiendo un proceso de diseño basado en buenas prácticas de Ingeniería del Software. La toma de decisiones de forma paulatina durante el proceso ha hecho que el resultado final sea sólido y se convierta en una plataforma escalable y sostenible en el tiempo. El culmen de todo el proceso llega estableciendo una metodología para el desarrollo de aplicaciones sobre la plataforma que permita a los usuarios crear de forma rápida y sencilla nuevos servicios siguiendo unas pautas de diseño claras. En este capítulo se establecerán las premisas a considerar para el desarrollo de aplicaciones basadas en servicios sobre plataforma presentada, así como los pasos a seguir para que la construcción de aplicaciones sea un proceso sencillo y exitoso que satisfaga las necesidades de los usuarios que opten por ella como middleware para la construcción de sus aplicaciones. Además, se llevará a cabo el análisis de los aspectos más representativos de la plataforma a través de la evaluación de distintos casos de estudio.

7.1. Metodología de Desarrollo de Aplicaciones basadas en Servicios sobre la Plataforma

Una plataforma de servicios tiene que ofrecer facilidades para que los desarrolladores puedan implementar aplicaciones utilizando las abstracciones propias de la

plataforma, los modelos de programación en base a la API que proporciona, los mecanismos propios para resolver las necesidades de los desarrolladores e incluso patrones de diseño para simplificar la construcción de aplicaciones. En nuestro caso, DOHA se ha construido para facilitar la construcción de aplicaciones en base a una partición del sistema en servicios siguiendo los principios de orientación a servicios definidos en SOA y con el objetivo de cubrir las necesidades de sistemas de computación ubicua.

Se ha seguido un proceso de desarrollo paulatino de la plataforma con objeto de ganar claridad y transparencia tanto en el diseño de aplicaciones como en el manejo de la infraestructura. Las pautas a seguir para llevar a cabo el diseño e implementación de aplicaciones sobre la plataforma están bien definidas para simplificar todo el proceso.

El elemento fundamental de DOHA obviamente es el servicio, y en torno a él gira el desarrollo de todas las aplicaciones. El primer reto al que se tiene que enfrentar un desarrollador cuando tiene que realizar aplicaciones en DOHA es cómo va a particionar el sistema en servicios. Existen diferentes estrategias bien definidas que parten primero por identificar los componentes o módulos del sistema como las unidades cohesivas y de bajo acoplamiento que posteriormente pueden integrarse entre sí mediante un enfoque bottom-up o se puede llegar al conjunto de componentes mediante un enfoque top-down.

A diferencia de lo que ocurre en el desarrollo de sistemas de información, en los sistemas de inteligencia ambiental las aplicaciones están vinculadas a dispositivos y a la reactividad de los mismos frente a los cambios que se producen en el entorno, por lo que es determinante aislar la funcionalidad en aplicaciones autónomas que se ejecutan de modo natural sobre nodos diferentes de la red. Desde un punto de vista conceptual la abstracción basada en servicios permite ver a los servicios como entidades independientes que pueden coordinarse entre sí mediante un mecanismo de interacción que se ejecuta de forma transparente.

En la construcción de un servicio cobra importancia establecer una estructura como la que hemos identificado en la anatomía de un servicio formada por tres capas distintas: interfaz, aplicación e interacción. A su vez, distintos componentes de diseño están directamente relacionados con dicha anatomía, como son el contrato de servicio o *Service Contract*, mapa de composición o *Composition Map* y archivo de configuración o *Configuration Info*. En el caso que el servicio sea además semántico y sensible al contexto habrá que incluir las ontologías y los perfiles de comportamiento o *Behavioral Profiles*. En el contexto del desarrollo de servicios sobre la plataforma, estos elementos son la base para la construcción de un patrón de diseño. Considerar la estructura de la anatomía de un servicio como el patrón de diseño a seguir para la construcción de servicios también asegura a los desarrolladores que los servicios satisfarán los requisitos impuestos por la plataforma y por tanto que se considerarán válidos [Lethbridge and Laganière, 2001].

En la Figura 7.1 se puede observar la plantilla del patrón de diseño de servicios sobre la plataforma basado en la anatomía de un servicio. Inicialmente se diseña la capa de interfaz, para la cual el desarrollador debe establecer cuáles son los requi-

sitos funcionales del servicio y dar forma a estos en base a operaciones que otros servicios, servicios consumidores, puede requerir. Esta información está directamente relacionada con el contrato del servicio. Por otro lado, puede ser que el servicio, para desarrollar su funcionalidad, necesite interaccionar con otros servicios de la plataforma. Es decir, que algunas de sus operaciones sean operaciones compuestas. En este caso, el desarrollador tendrá que analizar cómo se llevará a cabo dicha interacción y establecer con qué servicios y qué operaciones se requerirán. Esta información está directamente relacionada con el mapa de composición del servicio. Finalmente, el desarrollador implementará la funcionalidad de las operaciones del servicio que formarán la capa de aplicación. La información de cómo se llevan a cabo las operaciones en los servicios es privada y está siempre oculta hacia el exterior, dado que el servicio se comporta como una caja negra. Una vez diseñado el servicio, el desarrollador puede lanzar distintas instancias del mismo. Dichas instancias pueden ejecutarse en distintos dispositivos, por lo que su ejecución podrá verse afectada por las características hardware de dichos nodos. Esta información puede recogerse en el archivo de configuración, así como el identificador de cada una de dichas instancias y sus propiedades de QoS, que inevitablemente estarán ligadas a las características hardware del nodo sobre el que se ejecuten. A continuación, el desarrollador puede definir la semántica del servicio instanciando las ontologías que dotarán de semántica al mismo y los perfiles de comportamiento que determinarán su comportamiento proactivo.

7.1.1. Cómo modelar servicios sobre DOHA *Paso a Paso*

La base fundamental del desarrollo de una aplicación en DOHA depende del desarrollo de servicios alineados con la estructura de servicio de DOHA tal y como se vislumbra. Para simplificar dicho proceso, en esta sección se van a presentar los pasos que deben seguir los desarrolladores, a modo de metodología de desarrollo, para la construcción de sus aplicaciones.

Una vez identificado el caso a resolver los desarrolladores habrán identificado diferentes servicios a diseñar. Cada uno de ellos cubrirá una funcionalidad determinada de la aplicación global. Para construir el sistema completo distinguiremos dos etapas. En la primera etapa, a la que hemos denominado etapa de "especificación", se determinará qué funcionalidad concreta desempeña cada uno de los servicios a construir y cómo interacciona con el resto de servicios, si lo hace. Del mismo modo, existirán servicios de tipo dispositivo que interaccionarán con el hardware y que será necesario considerar para establecer adecuadamente su configuración.

Una segunda etapa, denominada "semántica", en la que el desarrollador irá más allá de los aspectos funcionales y no funcionales, para adentrarse en un modelado a más alto nivel que le permitirá endorsar un comportamiento proactivo a los servicios diseñados durante la etapa de especificación. Para ello se crearán los perfiles de comportamiento. En esta etapa se determinarán también las propiedades de calidad de servicio o *QoS* que se añadirán a la configuración del servicio. Dichas propiedades podrán ser utilizadas también de forma semántica para la selección dinámica de

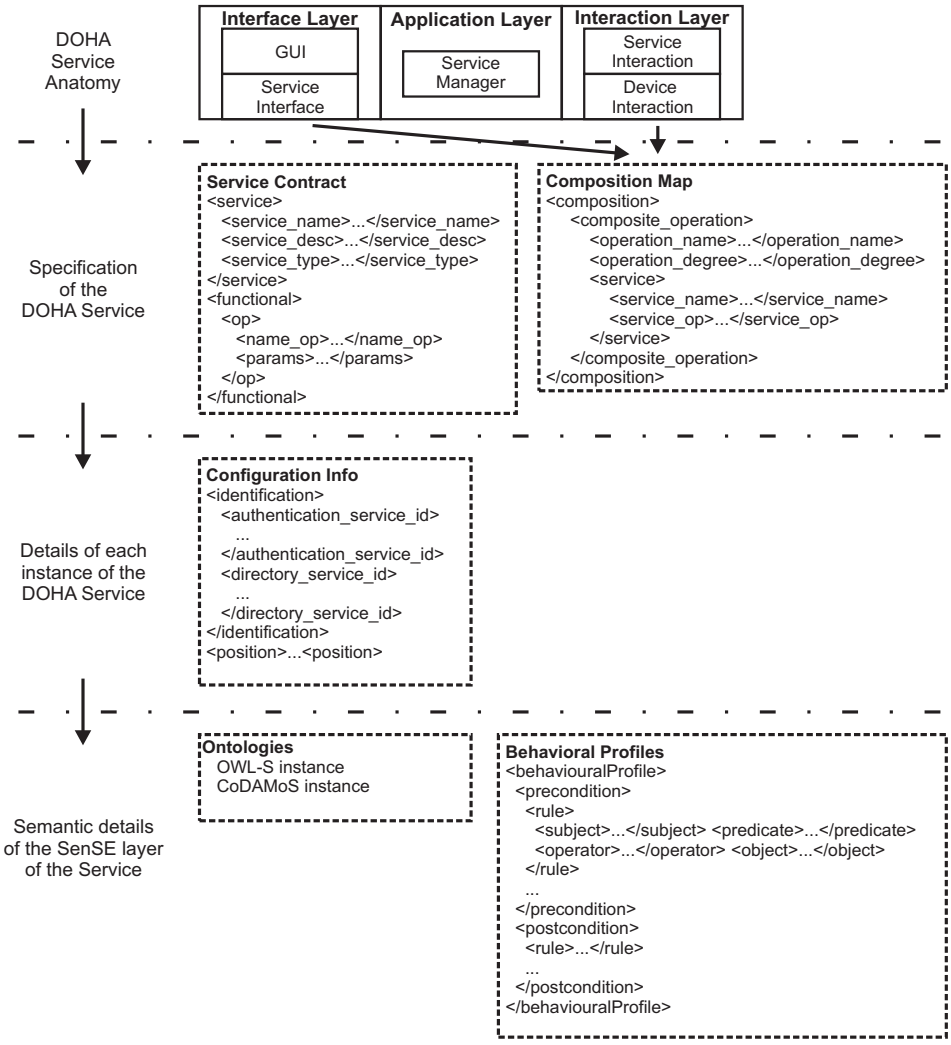


Figura 7.1: Patrón de diseño de los Servicios de la plataforma en base a la anatomía de servicios establecida

distintas instancias de un mismo servicio.

Determinados los dos grandes bloques de actividades a desempeñar, veamos paso a paso qué haría el desarrollador para construir un servicio DOHA:

1. Etapa de especificación

   a) Determinar qué servicios son necesarios y sus requisitos funcionales: construir el contrato de servicio de cada servicio.

   b) Determinar cómo se va a llevar la interacción entre servicios: construir el mapa de composición de las operaciones compuestas de cada servicio.

   c) Determinar cuántas instancias de cada servicio serán necesarias y sus requisitos no funcionales: construir los archivos de configuración de cada instancia.

2. Etapa de semántica

   a) Determinar las propiedades QoS asumibles por los servicios: completar el archivo de configuración de cada instancia del servicio.

   b) Determinar la semántica de cada servicio en base a la ontología de contexto: construir el mapa de contexto de cada servicio.

   c) Determinar los perfiles de comportamiento estándar de cada servicio: construir los perfiles de comportamiento.

7.2. Modos de Ejecución: Petición/Respuesta o Virtualización

Dada la relevancia de la colaboración entre servicios establecida en el mapa de composición, los desarrolladores deberán asumir una penalización en la ejecución de operaciones con un grado de complejidad alto. Para minimizar la repercusión que puede tener el grado de complejidad sobre el tiempo de ejecución global de las operaciones compuestas se propone virtualizar aquellas operaciones que lo permitan y siempre que no afecte a la percepción que el usuario final obtiene del sistema.

De este modo se distinguiría entre dos modos de ejecución posibles Petición/-Respuesta o *Request/Response* (RRM) y Virtualizado. La elección de uno u otro dependerá del tipo de servicio, sus restricciones y requisitos de rendimiento, siempre con el objetivo de optimizar la ejecución de operaciones con propiedades de soft real-time.

Request/Response Mode (RRM). En este modo, cuando una operación es invocada por un servicio consumidor, el servicio requerido ejecuta la implementación de dicha operación que, a su vez, puede incluir la invocación de otras operaciones compuestas si están especificadas en su mapa de composición, en caso de que esta sea una operación compuesta. Si las operaciones requeridas son también operaciones

compuestas, entonces estas serán invocadas provocando una anidación de operaciones compuestas. El flujo de ejecución del servicio consumidor que ha invocado a la operación sobre un servicio proveedor quedará bloqueado hasta que la operación haya finalizado y se haya devuelto una respuesta adecuada.

Virtualized Mode (VM). En este modo, el servicio virtualiza los recursos, el estado o los datos pertenecientes a otros servicios requeridos salvando una copia temporal de los mismos en memoria. De este modo, cuando una operación es requerida por un servicio consumidor, el servicio ejecuta al implementación de la misma sustituyendo la invocación de las operaciones requeridas definidas en el mapa de composición por una llamada a las operaciones locales encargadas de proporcionar los valores correspondientes y previamente salvados en memoria. Como consecuencia, la respuesta del servicio a una solicitud es prácticamente inmediata, pues al reducir la comunicación a través de la red con otros servicios para la ejecución de operaciones requeridas se reduce significativamente el tiempo de ejecución total de la operación. Para lograrlo, los servicios que virtualicen sus operaciones requerirán de un proceso en background que se ejecute periódicamente e invoque a las operaciones requeridas definidas en el mapa de composición para mantener la copia de los valores en memoria lo más actualizada posible. El periodo de ejecución de dicho proceso en background deberá estudiarse detenidamente para que su ejecución no sobrecargue la red más de lo que lo haría la ejecución del mapa de composición completo.

Ambos modos de ejecución poseen pros y contras y será el desarrollador el responsable de determinar cuál es más adecuado en cada caso. En la sección posterior se mostrarán resultados para mostrar las diferencias en tiempo de ejecución.

7.3. Implementación de Servicios

En los capítulos y secciones anteriores hemos visto las bases teóricas y formales de la plataforma, haciendo especial énfasis en aspectos relevantes como el modelo de composición o los perfiles de comportamiento. En esta sección veremos cómo el modelo teórico, partiendo de su correcta definición matemática, puede ser implementado utilizando un modelo software de alto nivel basado en el *Unified Modeling Language* (UML 2.0) [Arlow and Neustadt, 2002] .

UML ofrece a los desarrolladores un lenguaje de modelado universal y visual para el desarrollo de software orientado a objetos. En el desarrollo software se tiende a evitar desarrollar herramientas complejas con un ciclo de vida muy corto y ligadas a la resolución de un único problema. Es preferible que las herramientas software trabajen con artefactos generales en un formato estándar que otras herramientas puedan utilizar también. Esto aumenta los niveles de interoperabilidad del software y ayuda a desacoplar el desarrollo de aplicaciones software del tiempo de vida útil de la herramienta concreta con la que dicho software fue diseñado.

El *Desarrollo Dirigido por Modelos* (MDD) puede complementar el diseño, desa-

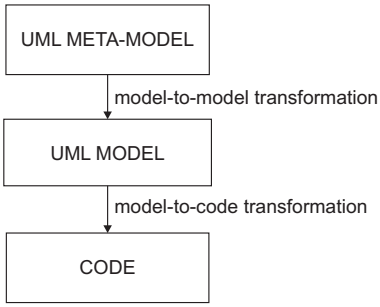


Figura 7.2: Procedimiento de metamodelado a llevar a cabo en el diseño y desarrollo de aplicaciones software

rrollo e implantación de sistemas basados en servicios colaborativos. MDD especifica las reglas de transformación del modelo para lograr la transformación automática del modelo de aplicación. MDD está estrechamente relacionado con *Model-Driven Architecture* (MDA) y *Meta-Object Facility* (MOF), el idioma de meta-modelo. MDD nos permite modelar la funcionalidad de un sistema y determinar la arquitectura general que este debe tener [Atkinson and Kuhne, 2003]. MDD también facilita el diseño de modelos arquitectónicos que permitan llevar a cabo la *autoadaptación* del sistema en tiempo de ejecución [Floch et al., 2006], aspecto de gran relevancia en el ámbito de la computación ubicua.

Nuestro objetivo es obtener un modelo software bien definido utilizando lenguajes estándares de desarrollo de software y las estructuras proporcionadas por UML 2.0. Desde el metamodelo de la plataforma podemos aplicar una transformación modelo a modelo, *model-to-model*, para obtener el diseño de una aplicación específica. Finalmente, con la transformación de modelo a código, *model-to-code*, podemos generar el código de la aplicación. La Figura 7.2 muestra el procedimiento de metamodelado a llevar a cabo en el diseño y desarrollo de aplicaciones software.

El desarrollo de una aplicación sobre la plataforma de servicios está relacionado con el ciclo de vida de los servicios implementados, así como en su especificación, tanto sintáctica como semántica, implementación, despliegue y ejecución [Koziolek, 2010]. Hemos definido la arquitectura específica a considerar por las aplicaciones que se implementen sobre la plataforma de servicios propuesta. Esta arquitectura incluye las tres fases representadas en la Figura 7.3: (1) análisis y diseño de servicios; (2) modelado de servicios; y (4) generación de código.

La fase de análisis y diseño del servicio, *service analysis and design*, abarca tanto el análisis de los requisitos del modelo como el diseño de las clases y los elementos externos necesarios para su desarrollo. En esta etapa es necesario considerar aspectos relativos a la semántica de los servicios además de los aspectos meramente funcionales. A su vez, establece el metamodelo que relaciona estos elementos con los elementos UML.

El procedimiento de modelado de servicios, *service modeling*, desarrolla la transición desde los servicios modelados como clases a componentes que pueden ser

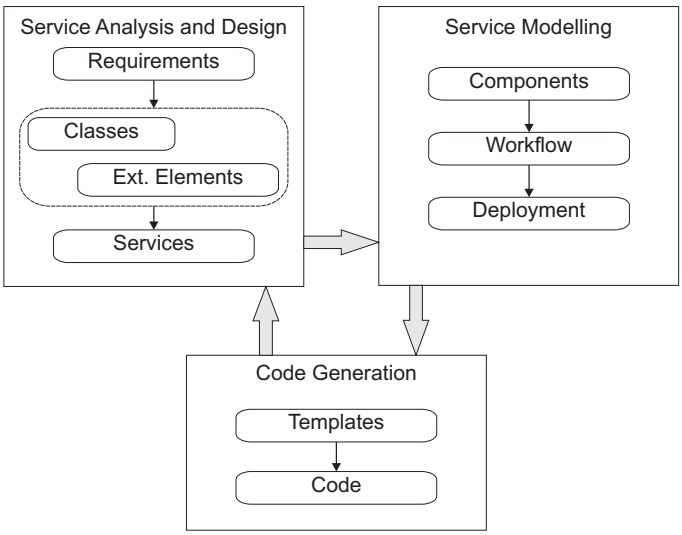


Figura 7.3: Arquitectura de sistemas establecida por la plataforma de servicios

desplegados en dispositivos reales.

La etapa de generación de código, *code generation*, genera el código de los servicios a través de plantillas *XmI* correspondientes con el diseño UML. Estas plantillas pueden realizarse en numerosos lenguajes de programación, como Java o C++.

7.3.1. Análisis y Diseño del Sistema

Los desarrolladores de aplicaciones que utilicen la plataforma de servicios deben conocer cómo diseñar un servicio, cómo modelar su contrato, su mapa de composición, es decir, cómo establecer el conjunto de operaciones compuestas a aplicar en un sistema real, el archivo de configuración, las ontologías y los perfiles de comportamiento.

Todos estos son elementos adicionales asociados con el servicio. Estos elementos están representados en la Figura 7.4 y determinan la configuración del servicio, la funcionalidad que ofrece, cuál es su comportamiento colaborativo, cómo representa su información semántica y cómo define su comportamiento proactivo.

Estos cuatro elementos, *Service Contract*, *Service Composition Map*, *Configuration Info*, *Ontology* y *Behavioral Profiles* son necesarios para llevar a cabo la correcta ejecución de los servicios de la plataforma.

Con respecto a la interacción entre servicios el mapa de composición juega un papel relevante. Una vez que el servicio se está ejecutando, este puede interactuar con los otros servicios de la plataforma para llevar a cabo operaciones compuestas. El servicio solo conoce a sus servicios requeridos; *a priori* desconoce la existencia del resto de servicios que se están ejecutando en la plataforma. La interacción entre los servicios sin la intervención del usuario permite a los desarrolladores crear aplicaciones que gestionan de forma autónoma la colaboración a nivel de servicios. El mapa de composición de servicios encapsula las operaciones compuestas. Además, cuan-

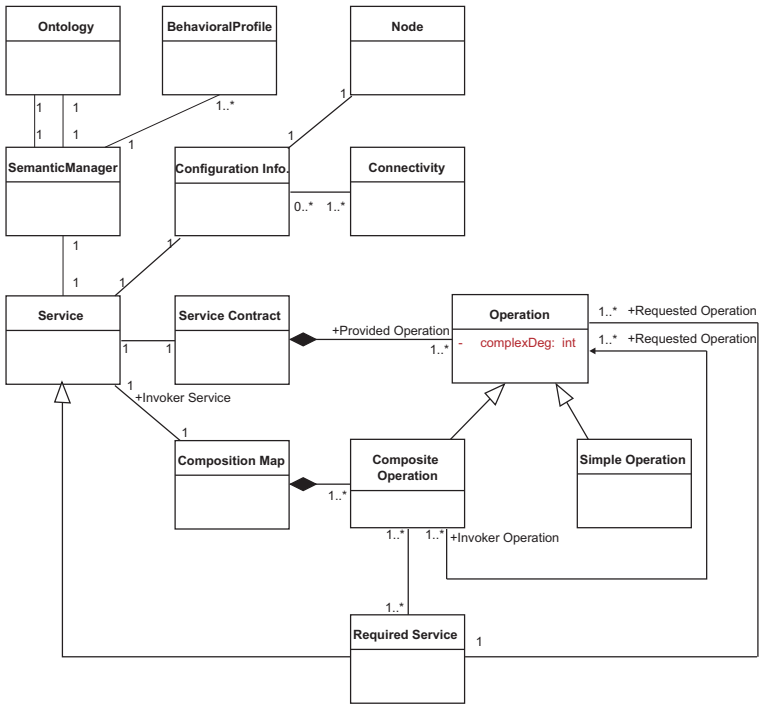


Figura 7.4: Elementos del modelo relevantes para diseñar un servicio en la plataforma

do el servicio ha sido publicado como un servicio disponible en la red de servicios, se permite que el resto de los servicios puedan invocar y ejecutar sus operaciones. Cuando se ejecuta una operación compuesta, el servicio interactúa con los servicios requeridos invocando la ejecución de sus operaciones solicitadas. Cuando el servicio invoca una operación solicitada pierde el foco de ejecución, pasando este al servicio requerido. Cuando la operación solicitada completa su ejecución el foco de ejecución vuelve de nuevo al servicio invocador. El servicio continúa ejecutando la operación compuesta invocando el resto de operaciones solicitadas que integran el mapa de composición de su operación.

7.3.2. Modelado de Servicios y Composición

Los desarrolladores pueden diseñar los servicios utilizando los componentes del modelado UML. Cada servicio sería un componente UML con la etiqueta de $<service\_name>$, como podemos ver en la Figura 7.5. Los desarrolladores realizan el diseño de los servicios y establecen las relaciones de colaboración entre ellos a través del mapa de composición; a continuación, aunque los servicios sufran cambios o actualicen su funcionalidad, la plataforma asegura que las propiedades establecidas por el modelo de composición se mantienen gracias a los algoritmos definidos.

El diagrama de componentes nos permite obtener una visión completa del conjunto de servicios que forman parte de una aplicación de servicios. Por medio de los componentes podemos representar el servicio como una caja negra, con una funcio-

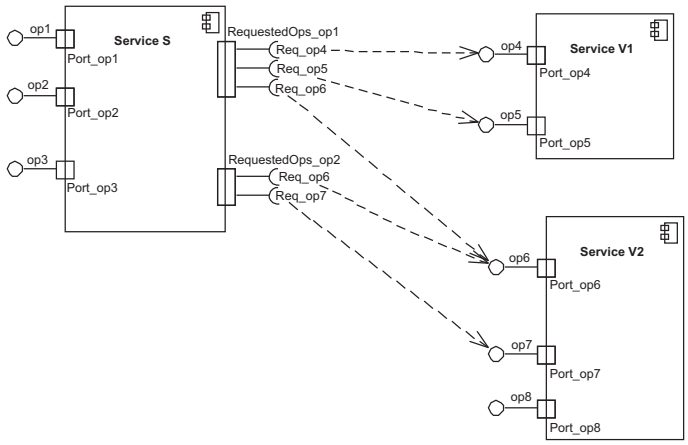


Figura 7.5: Modelo de componentes

nalidad completamente oculta al exterior. Los servicios que lo utilicen, o los servicios que este requiere, desconocen completamente cómo el servicio lleva a cabo sus operaciones, incluso si estas operaciones son simples o compuestas. Podemos observar también, que existe una clara relación entre la representación del servicio como un componente y la definición formal de servicio (Capítulo 5, Sección 5.1).

Los servicios consumidores únicamente conocen la interfaz proporcionada, *Provided Interface*, del servicio. Esta interfaz se utiliza para invocar su funcionalidad y se corresponde con las operaciones del servicio. Por otro lado, la interfaz requerida, *Required Interface*, se corresponde con aquellas operaciones que el servicio invoca en sus servicios requeridos – es decir, el servicio actúa como cliente de otros servicios a través de esta interfaz. Los puertos o *Ports* son los canales de comunicación disponibles hacia y desde el servicio. En la *Provided Interface* cada operación tiene un puerto independiente, lo cual establece un mecanismo de control y sincronización con respecto a las operaciones ejecutadas, ya que el servicio puede recibir diferentes peticiones asociadas a la misma operación de diferentes servicios consumidores. Cada puerto en la *Required Interface* incluye una operación solicitada en otros servicios.

Gracias a las interfaces proporcionada y requerida la funcionalidad del servicio queda bien identificada y se mantiene independiente de las operaciones solicitadas necesarias para la ejecución de las operaciones compuestas del mismo. El uso de los *Ports* y de la *Required Interface* establece una relación de dependencia entre las operaciones compuestas del servicio y la *Provided Interface* en sus servicios requeridos que debe ser utilizada para la invocación de las operaciones solicitadas.

Diseño del workflow de un servicio. A través de actividades es posible asociar un flujo de ejecución o *workflow* a cualquier elemento del modelo. El comportamiento de un servicio abarca las acciones llevadas a cabo desde que se invoca una operación compuesta hasta que finaliza la ejecución de todas las operaciones solicitadas en los servicios requeridos.

Los elementos que aparecen representados en la Figura 7.6 están relacionados

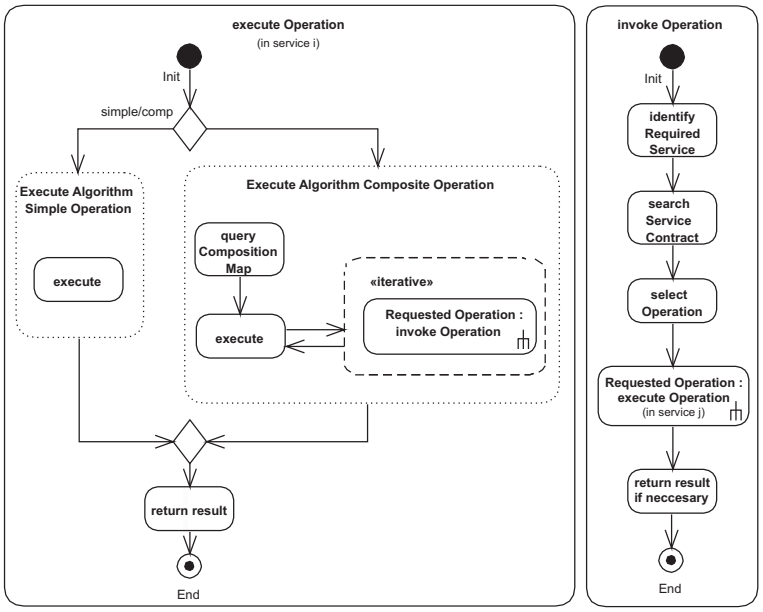


Figura 7.6: Acciones involucradas en la ejecución/invocación de una operación

con las clases de modelado definidas durante los procesos de análisis y diseño. Estos elementos tienen un papel activo en el modelo. El workflow representado en el diagrama de actividad nos permite mostrar las acciones realizadas por los servicios cuando estos actúan como clientes o como proveedores. Para ejecutar una operación en un servicio tras la recepción de una solicitud de ejecución a través de la "Provided Interface", el servicio ejecuta la acción *execute Operation*, representada en la primera parte de la Figura 7.6. En esta actividad, los elementos de diseño incluyen al "Composition Map" y al concepto de "Operation". Para cada operación solicitada, el servicio debe ejecutar la actividad "invoke Operation", representada en la segunda parte de la figura 7.6, y que cambia el flujo de ejecución, pasando este a ser controlado por el servicio requerido. En esta actividad los elementos de diseño involucrados son "Owner Service", "Service Contract" y "Operation". La finalización de esta actividad conlleva a su vez la ejecución de la actividad "execute Operation". Se crea así una estrecha relación entre la ejecución y la invocación de operaciones.

7.3.3. Despliegue de Servicios

Una vez que se ha realizado la especificación del servicio a través del diagrama de clases y se ha representado mediante el diagrama de componentes, debemos determinar cómo estos modelos software teórico/lógicos pueden ejecutarse en dispositivos reales. Para lograr esto, los diagramas de despliegue establecen cómo se distribuyen los servicios a través de los dispositivos del sistema y qué elementos son necesarios en la ejecución.

En la Figura 7.7 se ha representado el diagrama de despliegue del servicio S. El nodo dispositivo o *device node* representa el tipo de dispositivo físico en el que

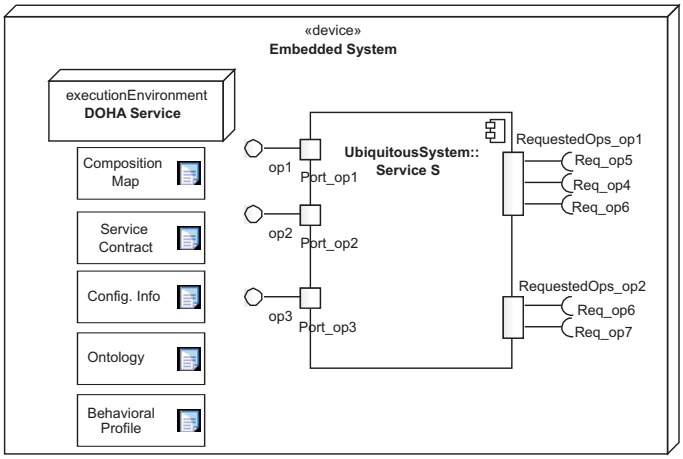


Figura 7.7: Despliegue de un servicio en un dispositivo

se ejecutará el servicio, que este caso es un dispositivo empotrado. El entorno de ejecución o *execution environment* identifica el tipo de entorno requerido para la ejecución del software. En este caso hemos utilizado un servicio DOHA o *DOHA Service*. Los documentos complementarios y externos al servicio que este necesite, como el *Service Contract*, *Service Composition Map*, *Configuration Info*, *Ontology*, *Behavioral Profiles* y el propio *Servicio S*, se representan por medio de "artefactos". Todos estos artefactos reflejan elementos del modelo lógico que pueden ser implementados utilizando la generación de código.

7.3.4. Generación de Código

Todos los diagramas UML podrían transformarse en documentos XML utilizando el meta-modelo establecido por MOF para cada tipo de diagrama. En nuestra representación tendríamos diferentes documentos para los diagramas de clases, componentes, actividad o de despliegue. Estos XML se suelen guardar en documentos XMI que son independientes de la herramienta CASE utilizada para generarlos. Esto significa que podemos utilizar cualquier herramienta CASE para modelar la colaboración entre los servicios porque podríamos generar código utilizando los esquemas de nuestro modelo de composición de servicios.

7.4. Evaluación del Rendimiento de la Plataforma de Servicios

Para evaluar el rendimiento de la plataforma de servicios hemos realizado distintas pruebas sobre la implementación realizada con JXTA y la realizada con DPWS. Las pruebas han consistido en medir el tiempo necesario para inicializar un servicio sobre cada uno de los middleware así como para realizar una comunicación punto a punto entre dos servicios.

Tabla 7.1: Evaluación de Rendimiento: Inicialización de DOHA-JXTA

	Configure JXTA	PeerGroup	Rdv	ModuleSpec	Start Service
AVT (ms)	345,30	112,83	89971,90	3,53	1,77
SD (ms)	79,54	4,86	5,33	0,82	0,50
WCET (ms)	609,00	125,00	89981,00	5,00	3,00

Tabla 7.2: Evaluación de Rendimiento: Inicialización de DOHA-DPWS

	Average Execution Time (ms)	Standard Deviation (ms)	WCET (ms)
Iter 1	14,57	6,86	28,00
Iter 2	15,67	6,86	38,00
Iter 3	11,27	4,80	32,00
Iter 4	14,73	5,25	29,00
Iter 5	13,17	2,85	21,00

En la medida del tiempo de inicialización de los servicios entran en juego las características propias de cada uno de los middleware subyacentes. Mientras que en la implementación DOHA-DPWS el objetivo de la inicialización es la creación de un objeto *Device* asociado al servicio, en la implementación DOHA-JXTA las tareas a llevar son muy numerosas. Es necesario configurar el peer JXTA sobre el que se ejecutará el servicio, conectarse al grupo de peers de DOHA si está ya creado, y si no crearlo, conectarse al peer *Rendezvous* más cercano, crear el anuncio del servicio de tipo *ModuleSpec* y, finalmente, publicar el servicio en la red. Solo en este punto se considerará el servicio inicializado. Por lo tanto, el rendimiento de ambas implementaciones está condicionado por el middleware que emplea y los resultados van a variar bastante comparativamente.

Las pruebas se han llevado a cabo sobre un Intel Core i5-3317U, 1.70GHz, 6GB RAM y JavaSE JDK 7. Para la implementación DOHA-JXTA se ha empleado JXTA 2.0, mientras que para la implementación DOHA-DPWS se ha utilizado WS4D-JMEDS beta 8.

La Tabla 7.1 muestra el resultado que se ha obtenido al realizar la inicialización de los servicios DOHA-JXTA. La tabla resume los tiempos empleados en llevar a cabo las tareas necesarias para considerar inicializado un servicio DOHA-JXTA. Se muestra el tiempo medio (AVT), la desviación típica (SD) y el tiempo de peor caso (WCET) asociado a cada tarea. Podemos observar como es especialmente representativo el tiempo empleado en la conexión con el peer *Rendezvous*, lo cual ralentiza notablemente el proceso.

El proceso de inicialización de los servicios DOHA-DPWS es considerablemente más liviano, como podemos observar en la Tabla 7.2, que muestra los resultados obtenidos en 5 iteraciones de 30 ejecuciones cada una, y en la Figura 7.8 que muestra los tiempos de cada una de estas ejecuciones. En este caso, aunque los tiempos son también del orden de las decenas de milisegundos, los WCET son más estables que en el caso de DOHA-DPWS.

En cuanto a la comunicación entre dos servicios, también encontramos diferencias

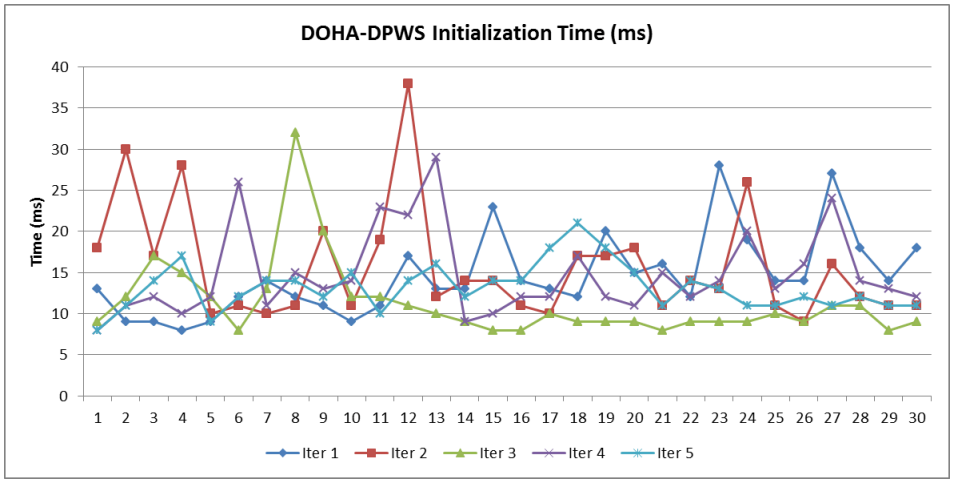


Figura 7.8: Evaluación de rendimiento: Inicialización de un Servicio DOHA-DPWS

Tabla 7.3: Evaluación de Rendimiento: Comunicación síncrona entre dos servicios DOHA-JXTA

	Average Execution Time (ms)	Standard Deviation (ms)	WCET (ms)
Iter 1	75,47	9,78	98,00
Iter 2	77,90	14,06	98,00
Iter 3	69,47	16,51	97,00
Iter 4	75,00	18,82	99,00
Iter 5	85,57	9,36	98,00

significativas en cuanto al rendimiento de las dos plataformas middleware. Mientras que la comunicación de dos servicios DOHA-DWPS se realiza de forma muy controlada con tiempos medios muy coherentes con los tiempos de peor ejecución, en el caso de DOHA-JXTA el tiempo empleado en la comunicación a través de los pipes JXTA se ve muy afectado por las condiciones de la red. Las Tablas 7.3 y 7.4 muestran los tiempos obtenidos en las 5 iteraciones realizadas, con 30 ejecuciones en cada iteración. La desviación típica asociada a cada implementación nos da una clara idea de la estabilidad de cada una de las plataformas. En las Figuras 7.9 y 7.10, asociadas a la comunicación en DOHA-JXTA y DOHA-DPWS respectivamente, se puede observar fácilmente la diferencia en cuanto a la fluctuación de los tiempos de cada iteración.

Por último, se ha medido el tiempo necesario para conectar con el framework JDOMO-JavaES en un servicio dispositivo y el tiempo de ejecución de la función *initJDomo* que carga en memoria el árbol de dispositivos conectados al nodo físico. Esta prueba es independiente del middleware subyacente, por lo que no se diferencia el uso de DPWS o JXTA en su implementación. El tiempo se ha medido haciendo uso de la función *initializeJdomoObject*, que crea una instancia de la clase *VCanal* que establece la comunicación entre dispositivos, inicializa JDOMO y crea la abstracción

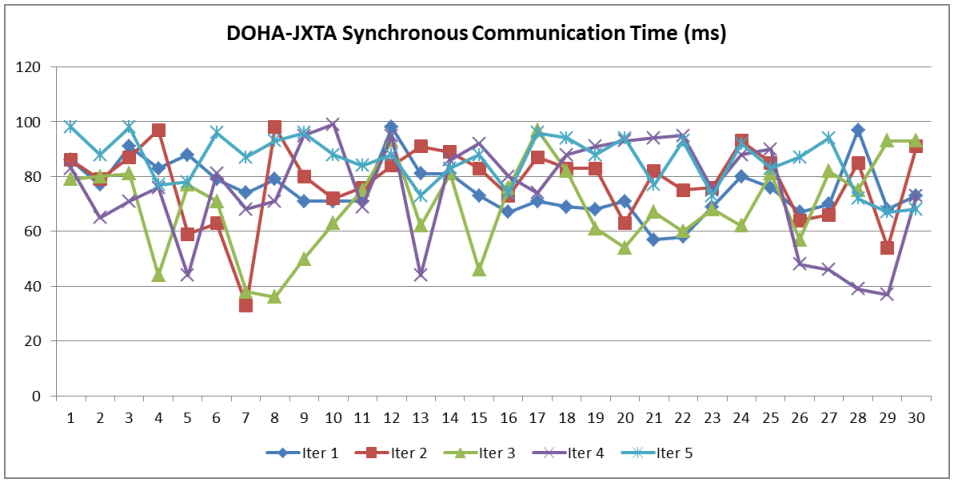


Figura 7.9: Evaluación de rendimiento: Comunicación Síncrona entre Servicios DOHA-JXTA

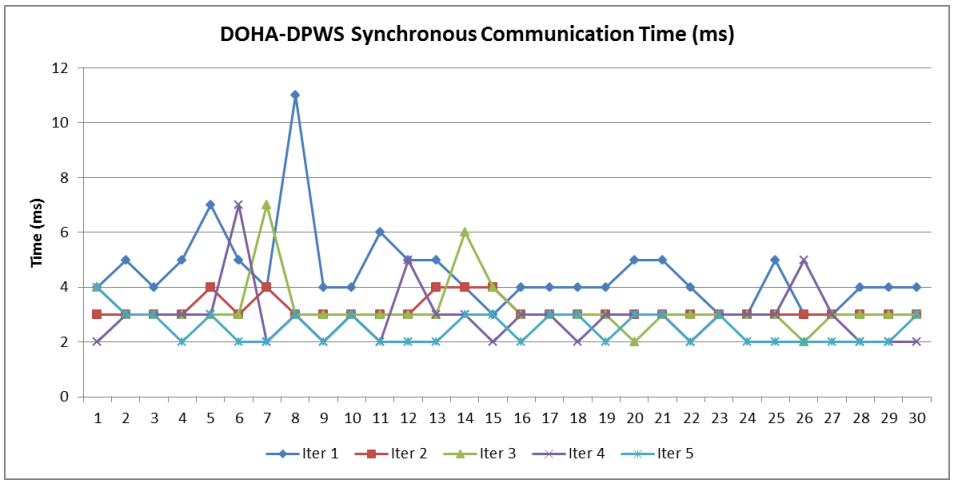


Figura 7.10: Evaluación de rendimiento: Comunicación Síncrona entre Servicios DOHA-DPWS

Tabla 7.4: Evaluación de Rendimiento: Comunicación síncrona entre dos servicios DOHA-DPWS

	Average Execution Time (ms)	Standard Deviation (ms)	WCET (ms)
Iter 1	4,50	1,53	11,00
Iter 2	3,17	0,38	4,00
Iter 3	3,23	0,97	7,00
Iter 4	2,93	1,08	7,00
Iter 5	2,50	0,57	4,00

Tabla 7.5: Evaluación de Rendimiento: Inicialización de JDOMO-JavaES

	Average Execution Time (ms)	Standard Deviation (ms)	WCET (ms)
Iter 1	4,80	2,51	17,00
Iter 2	4,27	0,74	6,00
Iter 3	4,17	0,87	6,00
Iter 4	4,60	1,16	7,00
Iter 5	4,97	1,43	9,00

de dispositivos físicos reales. Uno de los objetivos a alcanzar mediante la plataforma JDOMO-JavaES es abstraer la complejidad subyacente de los dispositivos físicos conectados a la red de servicios en el entorno real. Por lo tanto, es importante que su inicialización no afecte negativamente al rendimiento de la plataforma. Como podemos observar tanto en la Tabla 7.5, que muestra un resumen estadístico de los tiempo, como en la representación gráfica detallada de las 30 ejecuciones de cada una de las 5 iteraciones llevadas a cabo de la Figura 7.11, el tiempo de inicialización de JDOMO-JavaES es muy estable y los tiempos medios de las distintas iteraciones están bien delimitados con respecto al WCET.

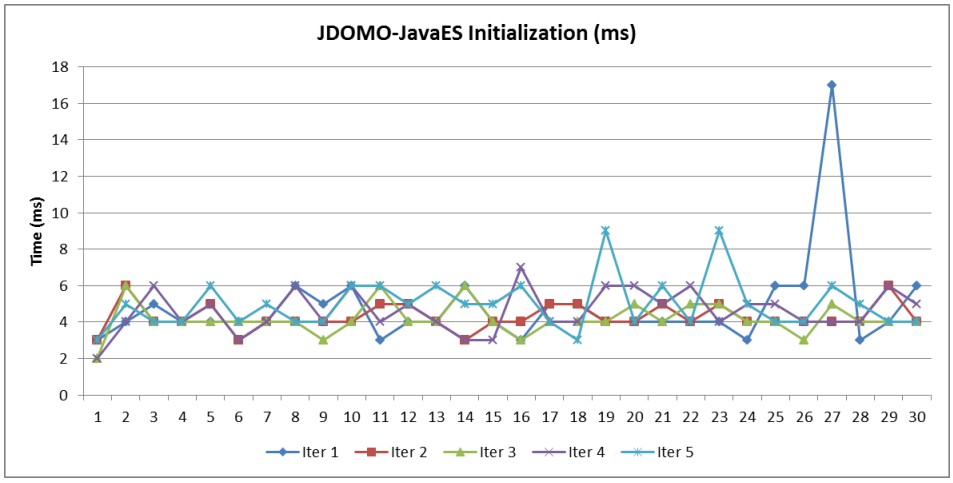


Figura 7.11: Evaluación de rendimiento: Inicialización de JODOMO-JavaES

7.5. Caso de estudio: Análisis del Modelo de Composición

Para evaluar el modelo de composición de DOHA se ha implementao un sistema que nos permite realizar predicciones de tiempo meteorológico en base a las condiciones atmosféricas, como humedad, presión y temperatura. Con el desarrollo de este sistema se pretende resolver un problema de fusión de datos en entornos de IoT utilizando el modelo de composición definido en DOHA que proporciona por un lado un modelo distribuido escalable para organizar los componentes (o servicios en nuestro caso) a partir de los cuales se identifican las fuentes de datos, y posteriormente se lleva a cabo el proceso de fusión de información, y por otra parte, establece un mecanismo determinista de acceso a las fuentes de datos que regula la rapidez con que se puede acceder a dichas fuentes mediante la definición de restricciones temporales [Rodríguez-Valenzuela et al., 2014].

Para poner en marcha el sistema de predicción meteorológica se han implementado distintos servicios *dispositivo* relacionados con la medida de las condiciones meteorológicas del entorno como fuente de datos directas, y otros servicios de más alto nivel que mediante composición de servicios fusionan la información obtenida por los primeros. Los servicios *dispositivo* se han replicados utilizando además distintos tipos de sensores. Es decir, para obtener el valor de temperatura a considerar en la predicción, se utilizará el valor de temperatura deducido a partir de como mínimo tres servicios de temperatura distintos. Se ha considerado la réplica de los servicios dispositivo por dos razones fundamentales: a) asegurar la corrección de las medidas, y b) detectar sensores o actuadores que no funcionen correctamente.

7.5.1. Dispositivos Hardware

Las pruebas se han ejecutado en seis sistemas empotrados diferentes. Cinco han sido Raspberry-Pi y una tablet Android. Cada Raspberry-Pi ha sido identificada con un índice, R1, R2, R3, R4 y R5, y las identificaremos así en adelante. La Raspberry-Pi es un dispositivo empotrado potente con un procesador ARM1176JZF-S 700 MHz, 512Mb RAM, Ethernet 100 y Java JDK 1.7 que ejecuta un Sistema Operativo Linux Debian [Raspberry-Pi, 2013]. La tablet Android, en adelante denominada AT, se ha utilizado como sistema con el que los usuarios puedan interactuar. Se trata de una T30s con cuatro núcleos, 1 Gb RAM y conexión WIFI que ejecuta un Sistema Operativo Android 4.1.2.

A las Raspberries se han conectado distintos tipos de sensores que permitan obtener los valores de humedad, presión y temperatura del entorno. Se han usado un barómetro digital MPL115A2 Miniature SPI (FreeScale) [FreescaleSemiconductor, 2013a] [FreescaleSemiconductor, 2013b], un sensor de presión digital AVR4201 (Atmel) [ATMEL, 2013] [Sensortec, 2013] y sensores de humedad que integra también la temperatura SHT71 y SHT75 (Sensirion) [Sensirion, 2013]. Estos sensores se han distribuido en tres de las Raspberries, R1, R2 y R3. El Atmel y un Sensirion SHT71 se han conectado a la R1 y R2, mientras que el FreeScale y un Sensirion

SHT75 se han conectado a la R3. La Figura 7.12 muestra las conexiones entre las tres Raspberries y los sensores.

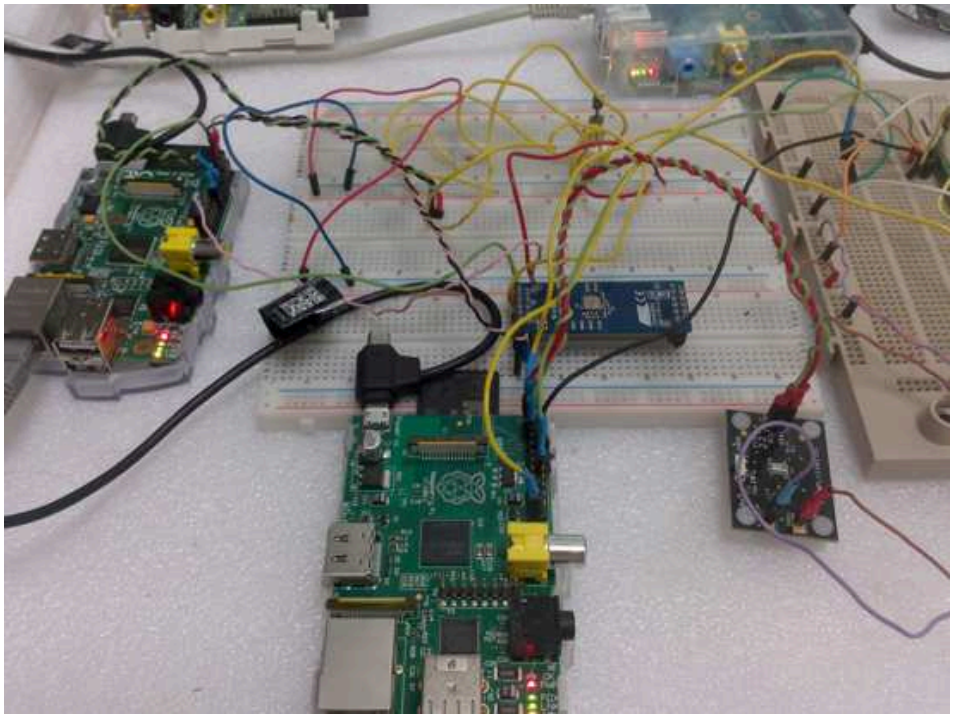


Figura 7.12: Caso de estudio 1: Dispositivos hardware utilizados

Las Tablas 7.6 y 7.7 muestran las propiedades de los sensores conectados a las Raspberries R1, R2 y R3, así como su relación con el tipo de valor que capturan, humedad, presión o temperatura. Los valores de humedad se han obtenidos a través del Sensirion. Los valores de presión con los sensores Atmel conectados a R1 y R2 y el FreeScale de la R3. Y finalmente, los valores de temperatura se han tomado como media de los obtenidos a partir del Atmel y Sensirion en R1 y R2, y del FreeScale en R3.

7.5.2. Servicios

Siguiendo una estrategia top-down se identificaron el conjunto de servicios necesarios para desarrollar el sistema de predicción meteorológica basado en fusión de datos. Para empezar, cada uno de los sensores se identifica claramente con un servicio dispositivo. Como consecuencia, se han considerado tres servicios dispositivo *Temperature Device Service* (TDS), *Humidity Device Service* (HDS) y *Pressure Device Service* (PDS). Para evitar inconsistencias en los valores del entorno [Hall and Llinas, 1997], se han considerado tres instancias distintas de cada uno de estos servicios. Es decir, cada servicio de temperatura, humedad y presión se considera replicado tres veces. En tiempo de ejecución existirán tres objetos diferentes con la

Tabla 7.6: Caso de estudio 1: Propiedades de los sensores conectados a las Raspberry-Pi 1 y Raspberry-Pi 2 obtenidas de su *datasheet*

	Humidity	Pressure	Temperature
Type	Sensirion	Atmel	Atmel/Sensirion
Sensor chip	SHT71	BMP085	BMP085/SHT71
Resolution	12bits	0.01	0.1/14bits
Accuracy	+-3.0	+-0.2	+-0.5/-
Repeatability	+-0.1	-	-/+-0.1
Output	2-wire interface	I2C	I2C/2-wire

Tabla 7.7: Caso de estudio 1: Propiedades de los sensores conectados a la Raspberry-Pi 3 obtenidas de su *datasheet*

	Humidity	Pressure	Temperature
Type	Sensirion	FreeScale	FreeScale/Sensirion
Sensor chip	SHT75	MPL115A2	MPL115A2/SHT75
Resolution	14bits	0.15	0.1/14bits
Accuracy	+-1.8	+-1kPa	+-0.1/-
Repeatability	+-0.1	-	-/+-0.1
Output	2-wire interface	I2C	I2C/2-wire

misma funcionalidad pero ejecutándose en tres dispositivos diferentes. Tres instancias del servicio de humedad, HDS1, HDS2 y HDS3; tres instancias del servicio de presión PDS1, PDS2 y PDS3; y tres instancias del servicio de temperatura, TDS1, TDS2 y TDS3.

La información obtenida de los servicios dispositivo se va a "fusionar" utilizando un algoritmo *N-Version*, que utiliza las N réplicas de cada tipo de servicio para evaluar la validez del valor final dado como resultado [Avizienis, 1985]. Los servicios que llevarán a cabo los algoritmos N-Version de cada tipo de servicio dispositivo serán *Humidity N-Version Service* (HNVS), *Pressure N-Version Service* (PNVS) y *Temerature N-Version Service* (TNVS), todos ellos con N igual a 3, es decir, consultarán tres instancias distintas de los servicios dispositivo requeridos. En el Algoritmo 7.1 se muestra el pseudocódigo de su funcionamiento. La operación *requestSensorValue()* corresponde con la operación específica a invocar en cada uno de los servicios dispositivo y que devolverá el valor del sensor concreto. En el caso del servicio de humedad, la operación *getHumidity()* invocará a la operación *getHValue()* en cada HDS; en el servicio de presión, la operación *getPressure()* invocará a la operación *getPValue()* en cada PDS; y por último, en el servicio de temperatura, la operación *getTemperature()* invocará a la operación *getTValue()* en cada TDS.

A más alto nivel, el servicio que llevará a cabo la predicción y al que se ha denominado *Weather Forecast Service* (WFS), utiliza los servicios N-Version. A partir

Algoritmo 7.1: Ejecución de los servicios *N-Version* - Operaciones *getHumidity()*, *getPressure()* y *getTemp()*

```
   Data: threshold (double)
   Result: value (double)
   /* Stage 1: request sensor values                                  */
 1 for s : Sensorsdo do
 2     value[s] = requestSensorValue();;
   /* Stage 2: validate sensor values using threshold                 */
 3 for n = 1tonumSensors − 1do do
 4     for i = n + 1tonumSensorsdo do
 5         if abs((value[n] − value[i]) > threshold) then
 6             version[n] = 0;;
   /* Stage 3: calculate nVersions value                              */
 7 for n = 1tonumSensors do
 8     if (version[n]! = 0) then
 9         sumVersion+ = version[n];;
10         nVersion + +;;
11 returnnSumVersions > 0?sumVersions/nVersions : 0; ;
```

de los valores obtenidos de estos servicios, el WFS aplicará un algoritmo basado en probabilidad para determinar la predicción meteorológica para el usuario. El algoritmo utilizado es una extensión del denominado *Zambretti forecaster* [Meteormetric, 2013]. La modificación con respecto al algoritmo de Zmbretti original consiste en utilizar además de los valores de presión, los valores de temperatura y humedad para mejorar la predicción. El Algoritmo 7.2 muestra el pseudocódigo de la operación *getPrediction()* de este servicio.

Los valores de humedad, presión y temperatura que el Algoritmo 7.2 recibe como entradas son los resultados obtenidos tras la ejecución de cada una de las operaciones N-Version en sus respectivos servicio. El valor de humedad será el resultado de ejecutar la operación *getHumidity()* en el servicio HNVS, el presión será el resultado de ejecutar la operación *getPressure()* en el servicio PNVS y, por último, el de temperatura será obtenido por la operación *getTemp()* en el TNVS.

Por último, el servicio *Climate Service* (CS) utiliza la información tanto del WFS como de los servicios N-Version para generar un pack de información completo al usuario, ofreciéndole sugerencias relativas a la predicción meteorológica realizada. La operación implementada en este servicio se denomina *getClimate()*.

La colaboración entre los servicios diseñados se lleva a cabo con el objetivo global de fusionar la información obtenida de los sensores para proporcionar una información mucho más completa a los usuarios. Cada servicio con operaciones compuestas tendrá su propio mapa de composición en el que se especificará su valor concreto del grado de complejidad, que dependerá del máximo grado de complejidad de sus ope-

Algoritmo 7.2: Ejecución del servicio *Weather Forecast* - Operación *getPrediction()*

Data: temperature (double), pressure (double), humidity (double)
Result: weather-forecaster-index (int)

```
   /* Stage 1: Initialize the top and bottom barometer values      */
1  baroTop = 1030;;
2  baroBottom = 950;;
   /* Stage 2: Calculate the Cloud Base value                      */
3  dewPoint = calculateDewPoint(temperature, humidity);;
4  cloudBase = calculateCloudBase(temperature, dewPoint);;
   /* Stage 3: Calculate the pressure                              */
5  pressure = calculatePressure(baroBottom, baroTop, pressure);;
   /* Stage 4: Calculate the prediction                            */
6  returnmatchPreddiction(pressure, cloudBase); ;
```

raciones requeridas. La Figura 7.13 muestra el mapa de composición completo del sistema y las relaciones existentes entre los grados de complejidad de cada servicio, sus operaciones y su posición dentro del mapa de composición del sistema completo. Tal como podemos ver en la figura, los servicios N-Version services, HNVS, PNVS y TNVS, y lo servicios WFS y CS son servicios compuestos. Además, sus operaciones compuestas fusionan la información obtenida de otros servicios con grado de complejidad inferior. Por lo tanto, en el caso de estudio planteado, todos los servicios compuestos actúan como servicios fusores.

El uso de operaciones compuestas en los servicios fusores como herramienta para la adquisición de los datos favorece la escalabilidad del sistema. No importa si se cambian los servicios fusores o si se incluyen nuevos servicios al sistema, pues podrán interaccionar con los servicios dispositivo a través del modelo de composición para obtener los datos. A su vez, un sistema así no necesita de un servicio centralizado para gestionar los datos, ya que cada servicio lo hace de forma distribuida.

El proceso de fusión de datos está en este ejemplo muy ligado al modelo de colaboración establecido en DOHA, por lo que se llevará a cabo de forma distribuida en la red de servicios. Además, se podría obtener de los servicios información relativa a los parámetros de QoS, de forma que se puedan acotar los tiempos de ejecución y garantizar propiedades de soft-real-time en la ejecución del sistema. En este caso, los servicios fusores pueden asegurar a sus clientes un tiempo máximo de ejecución cuando estos requieran alguna de sus operaciones.

En la Figura 7.14 se presenta el diagrama de despliegue de los servicios enumerados y su relación con los distintos dispositivos hardware considerados. Los servicios han sido implementados utilizando DOHA e interconectados a una red de area local. HDS1, PDS1 y TDS1 se han ejecutado en R1; HDS2, PDS2 y TDS2 en R2; HDS3, PDS3 y TDS3 en R3; HNVS, PNVS y TNVS en R4; WFS en R5, y CS en AT.

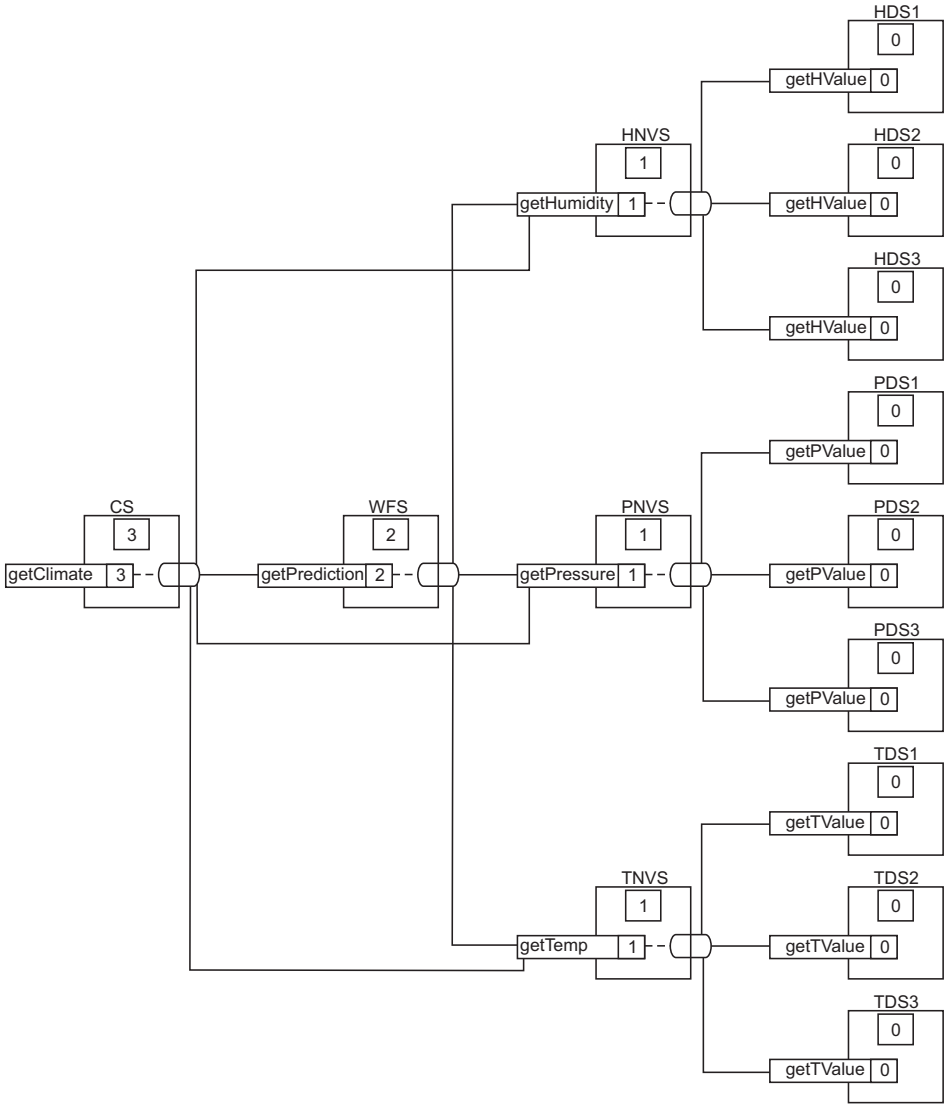


Figura 7.13: Caso de estudio 1: Mapa de composición del sistema completo

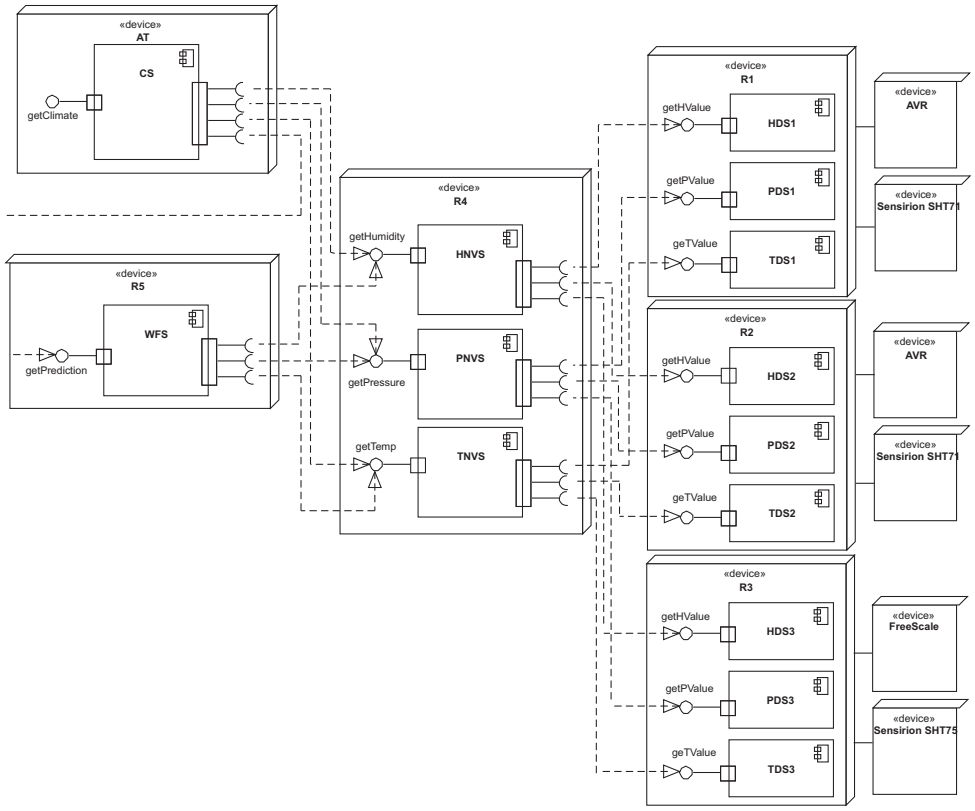


Figura 7.14: Caso de estudio 1: Diagrama de despliegue del sistema

7.5.3. Evaluación

El proceso de evaluación del sistema propuesto se ha llevado a cabo centrado en el análisis de los distintos tiempos de ejecución de las operaciones consideradas en el caso de estudio utilizando la implementación DPWS de DOHA. Controlar el máximo tiempo de respuesta de las operaciones puede ayudar a los desarrolladores a establecer propiedades de soft-real-time a sus sistemas. Además, esto nos da una idea de cómo se comporta el modelo de composición propuesto en DOHA aplicado a la adquisición y fusión de datos.

Durante el análisis se han ejecutado los servicios en los dos modos propuestos en la Sección 7.2, Petición/Respuesta (RRM) y Virtualizado (VM).

Las pruebas que se han realizado se pueden agrupar en dos tipos: a) análisis del tráfico de la red durante la ejecución de los servicios, y b) medida del tiempo de ejecución de cada operación.

7.5.3.1. Análisis del tráfico de red

El análisis del tráfico de red se ha llevado a cabo mientras la red estaba en reposo, durante el proceso de inicialización de los servicios y durante la ejecución de los servicios, distinguiendo los modos de ejecución de las operaciones implementados, RRM y VM. La Figura 7.15 muestra cómo varía el tráfico de red en cada uno de estos cuatro casos.

El gráfico Figura 7.15 (a) muestra el estado normal de la red cuando esta se encuentra en una situación de reposo, es decir, sin ningún servicio en ejecución. Sin embargo, en la Figura 7.15 (b) se observa un mayor movimiento en el tráfico de la red debido a la inicialización de los servicios. El incremento en el tráfico de red en este caso está directamente relacionado con la implementación de los servicios que se ha llevado a cabo y en la cual se ha empleado DPWS como middleware subyacente. Durante la inicialización de los servicios DOHA, DPWS envía mensajes "hello" por broadcast utilizando UDP, por lo que puede observarse su diferencia con respecto a la Figura 7.15 (a).

Con los servicios ejecutándose en modo RRM el tráfico de red presenta numerosas fluctuaciones, que se observan en forma de picos en la Figura 7.15 (c). Estos picos coinciden con la ejecución de las operaciones compuestas con alto grado de complejidad. Esto ocurre porque las operaciones compuestas con alto grado de complejidad suponen la anidación de otras operaciones compuestas, con el consiguiente aumento del número de mensajes en la red.

En el caso de la ejecución en modo VM, se observa en la Figura 7.15 (d) que el tráfico de red ha aumentado con respecto a al Figura 7.15 (a), que muestra la red en reposo, pero, sin embargo, no presenta grandes picos, a diferencia del modo RRM mostrado en la Figura 7.15 (c). En este caso la ejecución de las operaciones compuestas se lleva a cabo de forma periódica a través de un proceso en background que salva en memoria los resultados y de donde son consultados cuando las operaciones son invocadas. Esto favorece un balanceo más equilibrado de la carga de red así como un mejor control del tiempo de respuesta de las operaciones de los servicios.

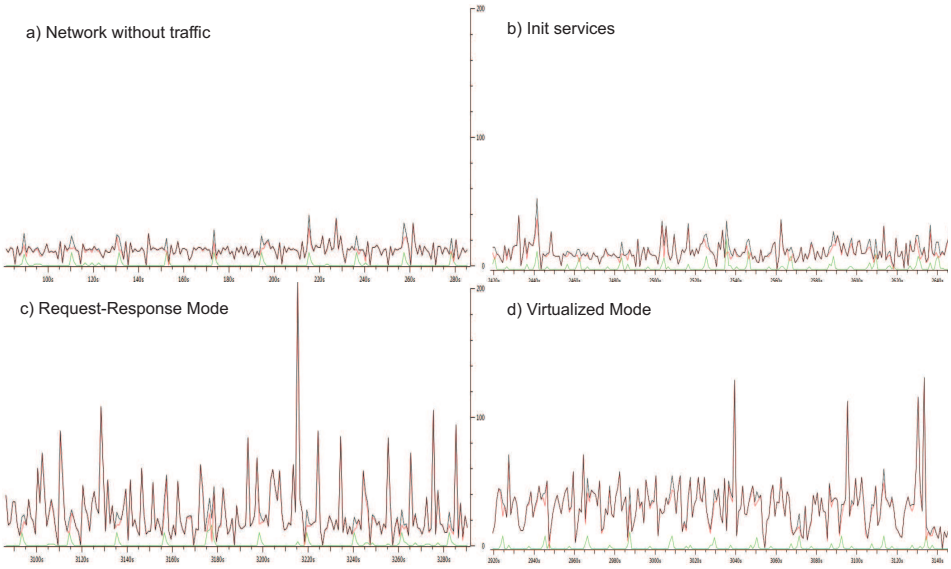


Figura 7.15: Caso de estudio 1: Evaluación del tráfico de red en 4 estados diferentes: a) sin ruido, b) durante la inicialización de servicios, c) durante la ejecución de los servicios en modo Request-Response y d) durante la ejecución de los servicios en modo Virtualized

7.5.3.2. Análisis de la ejecución de los servicios

Las pruebas realizadas para llevar a cabo el análisis de la ejecución de los servicios involucrados en el caso de estudio, en base a los dos modos de ejecución mencionados, RRM y VM, han consistido en llevar un seguimiento del tiempo de ejecución de las operaciones tomando medidas de forma repetitiva cada vez que estas se han ejecutado.

En base a dichas medidas de tiempo, la Tabla 7.8 incluye un resumen de los resultados obtenidos agrupados por el modo de ejecución, RRM o VM. En el caso del modo de ejecución VM se ha desechado el valor medido en la primera ejecución, pues este supone la inicialización de la red y representa un impacto negativo sobre las estadísticas finales. Además de la media del tiempo de ejecución, se presenta la desviación típica y el valor de peor tiempo de ejecución, WCET, en cada caso. Para que la visualización de los resultados sea más intuitiva se presentan a continuación distintas gráficas resumen asociados a estos valores y que nos ayudarán a discutir los resultados obtenidos.

Las Figuras 7.16a y 7.16b muestran los tiempos de ejecución de las operaciones simples localizadas en los servicios de tipo dispositivo HDS, PDS y TDS. La obtención de los valores de humedad, presión y temperatura se han llevado a cabo a través del bus I2C de la R1 y accediendo a los chip BMP085 y SHT71, por lo que la lectura de los valores de los sensores se lleva a cabo de forma secuencial. En este caso, el tiempo obtenido tras la ejecución de las operaciones en modo RRM equivale

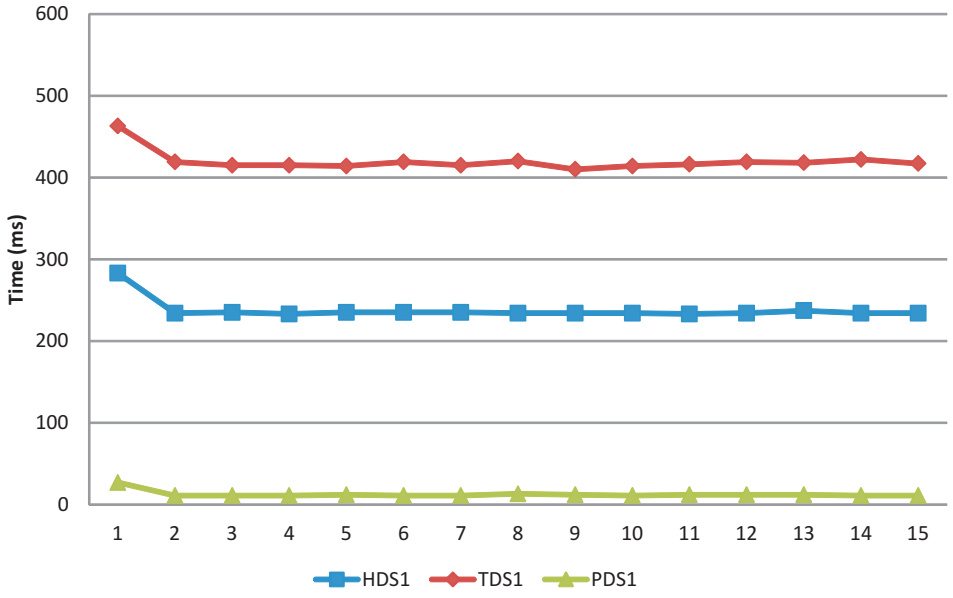
Tabla 7.8: Caso de estudio 1: Tabla resumen que muestra los valores de tiempo de ejecución medio, desviación estándar y WCET de cada operación de los servicios del sistema. Para el modo de ejecución RRM se incluye también la medida del BWCET.

IoT Service	Operation	RRM mode				VM mode		
		Average Execution Time (s)	Standard deviation (s)	WCET (s)	BWCET (s)	Average Execution Time (s)	Standard Deviation (s)	WCET (s)
PDS1	getPValue()	0,013	0,004	0,027	0,015	0,021	0,006	0,117
TDS1	getTValue()	0,420	0,012	0,463	0,428	0,026	0,024	0,111
HDS1	getHValue()	0,238	0,013	0,283	0,246	0,027	0,027	0,121
PNVS	getPressure()	0,910	1,975	8,001	2,224	0,313	0,409	1,687
TNVS	getTemp()	1,873	0,084	2,169	1,929	0,207	0,095	0,533
HNVS	getHumidity()	1,196	0,181	1,640	1,317	0,810	2,309	9,148
WFS	getPrediction()	3,812	0,500	5,135	4,144	0,390	0,278	1,353
CS	getClimate()	12,486	2,223	19,026	13,964	0,001	0,001	0,004

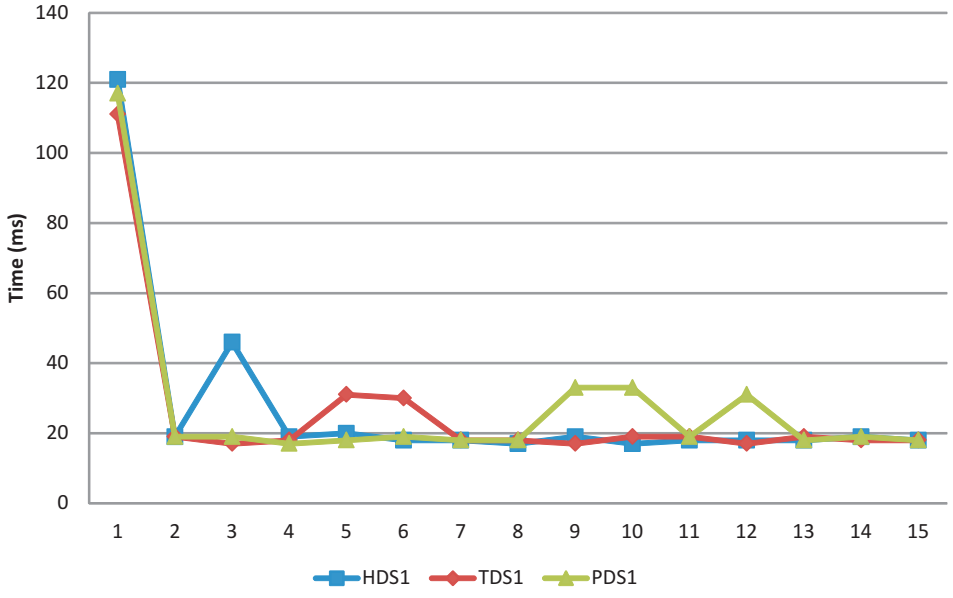
al tiempo necesario para leer el sensor físico a través del bus I2C. Sin embargo, en el modo de ejecución VM dicho valor está ya disponible en memoria, por lo que la ejecución de la operación es casi inmediata.

Si analizamos la Figura 7.16a junto con la información contenida en la Tabla 7.8 encontramos diferencias en el tiempo de ejecución de los distintos servicios dispositivo en modo RRM. A pesar de que ninguno de estos tiempos es demasiado elevado, vamos a analizar porqué se producen estas diferencias entre servicios a priori muy similares, solo diferenciados por el tipo de sensor que consultan. En primer lugar, existe un cuello de botella en el acceso al bus I2C para la lectura del valor del sensor, que irremediablemente dependerá del orden en el que lleguen las solicitudes a la R1. En segundo lugar, el tiempo que requiere la conversión analógico-digital (ADC) depende de la resolución y del tiempo de muestreo del sensor. Esto supone un mayor retardo en las operaciones del TDS y el HDS con respecto al ODS. Pero, además, el tiempo asociado al TDS es superior debido a que la temperatura se toma como la media de los valores de dos sensores. En el caso de la R1 se consideran los valores de temperatura obtenidos desde el Atmel y el Sensirion.

La Figura 7.17 muestra los tiempos de ejecución especificados en la Tabla 7.8 para las operaciones de los servicios CS, WFS, HNVS, TNVS y PNVS en ambos modos de ejecución, RRM y VM. En el caso de los servicios *N-Version* con grado de complejidad 1, HNVS, TNVS y PNVS, cada operación *get* requiere la invocación de una operación simple en tres instancias diferentes del mismo servicio dispositivo para llevar a cabo la fusión de datos, de acuerdo a los esquemas presentados en las Figuras 7.14 y 7.15. Por ejemplo, la operación *getPressure()* en el servicio PVNS debe invocar a la operación *getPValue()* en tres instancias diferentes del servicio PDS. En el modo de ejecución RRM, el tiempo de ejecución de cada operación N-Versión es elevado, debido a la invocación y ejecución secuencial de las operaciones requeridas en los servicios dispositivo. Del mismo modo, este comportamiento se presenta también en la ejecución de las operación *getPrediction()* del servicio WFS con 9 operaciones y *getClimate()* del servicio CS con 19 operaciones. Las fluctuaciones en los tiempos de ejecución del modo RRM que se observan en la Figura 7.17 (a) se deben principalmente a la existencia de ruido en la red.

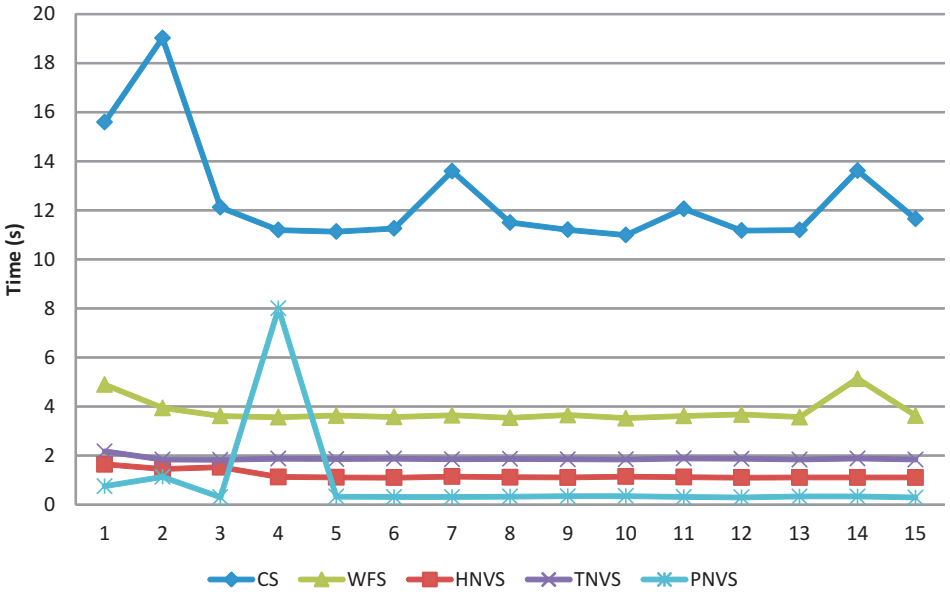


(a) RRM

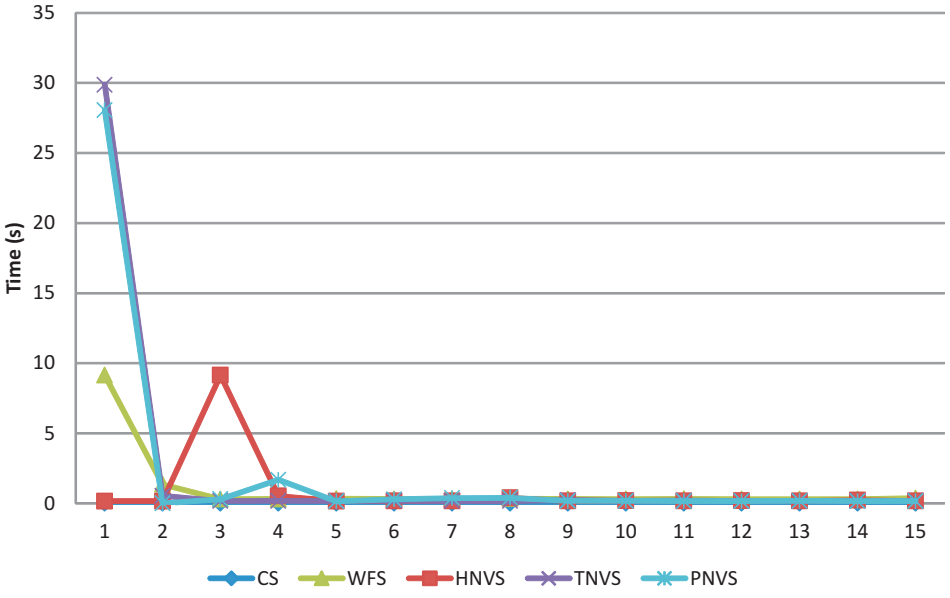


(b) VM

Figura 7.16: Caso de estudio 1: Tiempo de ejecución de los servicios de tipo *Device Service* en R1 considerando los modos de ejecución a) RRM y b) VM



(a) RRM



(b) VM

Figura 7.17: Caso de estudio 1: Tiempo de ejecución de los servicios compuestos considerando los modos de ejecución a) RRM y b) VM

A diferencia de la ejecución en RRM, con VM los tiempos de ejecución de las operaciones N-Version son mucho más bajos, pues la ejecución de las operaciones requeridas se lleva a cabo de forma previa a la invocación de las operaciones. Las fluctuaciones que se observan en los tiempos de ejecución recogidos en la Figura 7.17 (b) correspondientes al modo de ejecución VM, especialmente en las primeras ejecuciones, están relacionadas con la ejecución de los procesos en background responsables del salvado en memoria de los valores respuesta. Este comportamiento se observa también en la ejecución de las operaciones *getPrediction()* del WFS y *getClimate()* del CS.

Se puede concluir que el tiempo de ejecución de una operación en modo RR depende directamente del grado de complejidad de dicha operación. A mayor grado de complejidad, mayor tiempo de ejecución debido a que al aumento en el número de operaciones requeridas y por tanto de transacciones a través de la red. De hecho, el tiempo de ejecución de una operación se ve influenciado por el número de operaciones invocadas, pues además del tiempo de ejecución de cada una de ellas, se añade también un retardo asociado a la comunicación a través de la red por cada invocación. Sin embargo, en el modo de ejecución VM el tiempo de ejecución depende exclusivamente del tiempo de ejecución local de la operación, pues los valores de las operaciones requeridas se encuentran salvados en memoria.

En la Tabla 7.8 se observa que, en modo de ejecución RRM, las operaciones de los servicios dispositivo tienen un WCET con una magnitud de cientos de milisegundos, los servicios N-Version del orden de un segundo, el WFS de 5 segundos, y, finalmente, el CS del orden de cien segundos. Sin embargo, los WCET se han obtenido salvando los tiempos de ejecución de las operaciones durante su ejecución en las pruebas, y algunas de estas medidas pueden tener un impacto negativo sobre la estimación del WCET. Por lo tanto, el resultado es un valor muy pesimista, lejos del tiempo de ejecución medio, que supone además una pérdida importante de recursos. Por ejemplo, el WCET de la operación *getPressure()* del PNVS es mucho mayor que el WCET de la operación *getPrediction()* del WFS.

Para reducir el pesimismo que arroja el WCET se puede seleccionar una cota de WCET menor que llamaremos Best Worst-Case Execution Time (BWCET). El BWCET se determina evaluando el máximo del 99 % del intervalo de confianza para una distribución estándar de los tiempos de ejecución. Esto nos da una probabilidad del 99 % de que el tiempo de ejecución de la operación se ejecute dentro de dicho intervalo de tiempo. La Tabla 7.8 muestra los valores de BWCET para cada operación del caso de estudio, con un valor en todos los casos muy cercano al valor medio, permitiendo un mejor aprovechamiento de los recursos en comparativa con los valores de WCET.

Las medidas tomadas sobre los tiempos de ejecución de los distintos servicios nos pueden ayudar también a determinar los tiempos de ejecución periódicos de las operaciones requeridas por parte de los procesos en background en modo VM. Este tiempo puede definirse como tiempo de muestreo de los datos, un parámetro que puede ser muy relevante en caso de considerar el modo de ejecución VM en el algoritmo de fusión. Inicialmente, se podría considerar este tiempo de muestreo

igual al valor del WCET, ya que este garantiza una cota superior para cada posible ejecución de una operación dada. Dada esta cota, la ejecución de la operación nunca podría exceder este deadline por lo que los valores obtenidos se consideran válidos y actualizados en toda ejecución del algoritmo de fusión. En definitiva, el WCET, y mejor el BWCET, pueden utilizarse para especificar el periodo de muestreo de las operaciones requeridas en los servicios ejecutados en modo VM. En el modo de ejecución VM el intercambio de mensajes entre servicios es limitado en comparación con el modo RR;, ya que cada servicio es responsable de invocar sus operaciones requeridas regularmente para optimizar el tiempo de respuesta hacia sus clientes, ayudando además a satisfacer restricciones de soft-real-time sobre su ejecución.

En resumen, a partir del análisis de resultados obtenido de este caso de estudio se presentan las siguientes recomendaciones para los desarrolladores que utilicen DOHA para construir aplicaciones colaborativas que lleven a cabo fusión de datos:

1. **La construcción de un sistema escalable para procesos de fusión de datos es más seguro cuando se considera un modelo de composición de servicios.** Un modelo de composición de servicios válido proporciona una forma correcta para construir servicios escalables. En general, el modelo de composición de servicios proporciona mecanismos para mejorar la interconexión y la comunicación entre los servicios. En el modelo de composición propuesto en DOHA se va más allá proporcionando además una forma de verificar globalmente el sistema. Esto es particularmente interesante en la implementación de procedimientos de fusión, especialmente cuando el sistema se debe construir gradualmente, para no reimplementar el sistema cada vez que nuevos datos son añadidos al algoritmo de fusión implementado.

2. **Es preferible diseñar el sistema en base a servicios con grado de complejidad medio/bajo.** A menor grado de complejidad, menor número de interacciones y por tanto invocación de operaciones requeridas. Por lo tanto, se puede controlar mejor el tiempo de ejecución y también el periodo de muestreo. No obstante, el sistema es completamente escalable, por lo que es posible añadir nuevos servicios con mayor grado de complejidad asumiendo los costes que esto supone sobre el tiempo de ejecución final de las operaciones.

3. **Cuando un servicio contiene operaciones con un grado de complejidad elevado es preferible su ejecución en modo VM que en modo RRM.** Cuando el grado de complejidad de una operación es alto, la invocación de esta operación en el modo de RRM supone una penalización en tiempo de ejecución, principalmente por dos razones. En primer lugar, la invocación de este tipo de operaciones puede necesitar recursos de red suficientes, como un buen ancho de banda o ausencia de ruido en la comunicación, pues el número de mensajes transmitidos puede ser elevado en un intervalo de tiempo muy acotado, provocando picos de carga en la red. En segundo lugar, el flujo de control en el servicio invocador se bloquea debido a los intervalos de tiempo

necesarios para recibir las respuestas. Un tiempo de bloqueo excesivo del flujo de control favorece la construcción de sistemas infrautilizados.

La utilización del modo de ejecución VM de este tipo de operaciones beneficiaria al sistema en su conjunto ya que la carga de red se repartiría entre todos los servicios por igual en lugar de quedar ligada a los servicios específicos con operaciones con alto grado de complejidad. En VM cada servicio es responsable de mantener actualizada la información de las operaciones requeridas de acuerdo a su mapa de composición. Además, el bloqueo del flujo de control de las operaciones invocadoras ser reduce significativamente, lo que en consecuencia mejora la capacidad de respuesta del sistema global.

4. **En un contexto de ejecución con restricciones de tiempo real es preferible fijar el periodo de muestreo de cada operación de cada servicio con antelación**, especialmente en el caso de operaciones compuestas. Para ello, es recomendable utilizar los valores de WCET o mejor de BWCET, dependiendo de las exigencias de tiempo real a satisfacer. La propiedad de *idempotencia* de los servicios garantiza que se pueden gestionar varias solicitud al mismo tiempo sin que esto suponga cambios en los resultados ni afecte a los datos. Sin embargo, si una operación concreta se invoca continuamente con una frecuencia superior al periodo mínimo de muestreo podría sobrecargar al servicio sin que ello suponga obtener información más actualizada, pues el servicio no tendría tiempo suficiente de actualizar los resultados. La aplicación de restricciones de tiempo real en un sistema de fusión de datos puede beneficiar la ejecución global del sistema permitiendo realizar un mejor control de los recursos del sistema. Dependiendo de las condiciones de criticidad del sistema, el ingeniero puede establecer el período de muestreo con el WCET, cuando los datos gestionados o servicios implicados son críticos, o con BWCET en otros casos.

7.6. Caso de estudio: Análisis del Comportamiento Proactivo del Sistema

A nivel semántico, se ha llevado a cabo un caso de estudio relacionado con los perfiles de comportamiento de SenSE, la capa semántica de la plataforma de servicios. Dicho caso de estudio se ha realizado en colaboración con el *Bussines Informatics Group* (BIG) de la *Dublin City University* y está estrechamente relacionado con la evaluación de *Data Quality* o calidad de datos en plataformas de servicios [Rodríguez-Valenzuela et al., 2013].

Existe una relación clara entre los perfiles de comportamiento que utilizan la semántica establecida por las ontologías de contexto y el funcionamiento proactivo del sistema en su conjunto. Gracias a ello, los perfiles de comportamiento pueden interpretarse como esquemas de evaluación para determinar la calidad de los datos o de la información que la aplicación que se está ejecutando sobre la plataforma de

Tabla 7.9: Caso de estudio 2: Especificación de los perfiles de datos o "data profiling" asociados a la plataforma de servicios

Who	How	What		When
Service	Process	Information Type/Entity	Information Elements	Requirements
Identification of the Service	BP Action Rule Predicate	BP Action Rule Subject	BP Action Rule Object	BP Precondition

servicios genera o manipula. Es en base a esta relación sobre la que se plantea el caso de estudio.

Que un perfil de comportamiento se active debido a la veracidad de sus precondiciones, significará que el sistema satisface una serie de restricciones. Del mismo modo, tras la ejecución de las acciones asociadas al perfil de comportamiento y contenidas en sus acciones, el estado del sistema debe ser el determinado por las precondiciones. Si es así, las operaciones involucradas en las acciones del perfil funcionan adecuadamente y por tanto, los datos se considerarán correctos. Si por el contrario, tras la ejecución de las operaciones del perfil el estado del sistema no satisface las precondiciones, la evaluación basada en *Data Quality* habrá fallado y será necesario determinar el motivo del funcionamiento erróneo del sistema.

De este modo, se propone procesar la información que maneja el sistema a través de perfiles de datos o data profiling como metodología para determinar su corrección y la calidad de los datos (data quality, DQ).

7.6.1. Perfiles de datos como Herramienta de Evaluación

El proceso denominado *data profiling* es una etapa de la metodología de *Data Quality* en la cual se define qué se entiende por datos válidos o "calidad de datos" [Ora, 2008]. Los perfiles nos ayudarán a detectar posibles anomalías en el funcionamiento del sistema tras la activación de los perfiles de comportamiento.

Para establecer los "data profiles" asociados a la plataforma de servicios se han considerado los servicios y las condiciones establecidas por los usuarios en los perfiles de comportamiento. En la Tabla 7.9 se presenta una estructuración de estos elementos organizados siguiendo el patrón de metadatos 'who' (quién), 'how' (cómo), 'what' (qué) y 'when' (cuándo).

Para avalar la utilización de los perfiles de comportamiento, que a su vez son parte clave del razonador de la capa semántica de la plataforma, se presenta a continuación un resumen estadístico del tiempo de ejecución que requiere dicho procedimiento. La Tabla 7.10 muestra los valores de tiempo de ejecución medio, desviación típica y tiempo de peor caso en milisegundos. El procedimiento evaluado ha abarcado todas las tareas necesarias para la evaluación de los perfiles de comportamiento en el razonador de la plataforma, como son análisis de las precondiciones, actualización de la información del entorno, razonamiento y resolución, es decir, aplicación de las operaciones contenidas en las reglas de la acción si es oportuno. El servicio evaluado ha sido un servicio dispositivo de control de luminosidad denominado *Iluminance*

Tabla 7.10: Caso de estudio 2: Evaluación de rendimiento en base al tiempo de ejecución del razonador basado en perfiles de comportamiento de SenSE

	Average Execution Time (ms)	Standard Deviation (ms)	WCET (ms)
Iter 1	3,13	1,25	8,00
Iter 2	2,33	0,66	4,00
Iter 3	1,97	1,19	7,00
Iter 4	2,07	1,46	9,00
Iter 5	1,97	1,79	9,00

Service con sus operaciones implementadas en modo virtualizado. Como ya vimos en la Sección 7.2, la implementación en modo virtualizado de las operaciones de un servicio acorta los tiempos de respuesta del servicio. En la Figura 7.18 se muestran los tiempos de ejecución tomados en 5 iteraciones con 30 ejecuciones del razonador cada una de ellas.

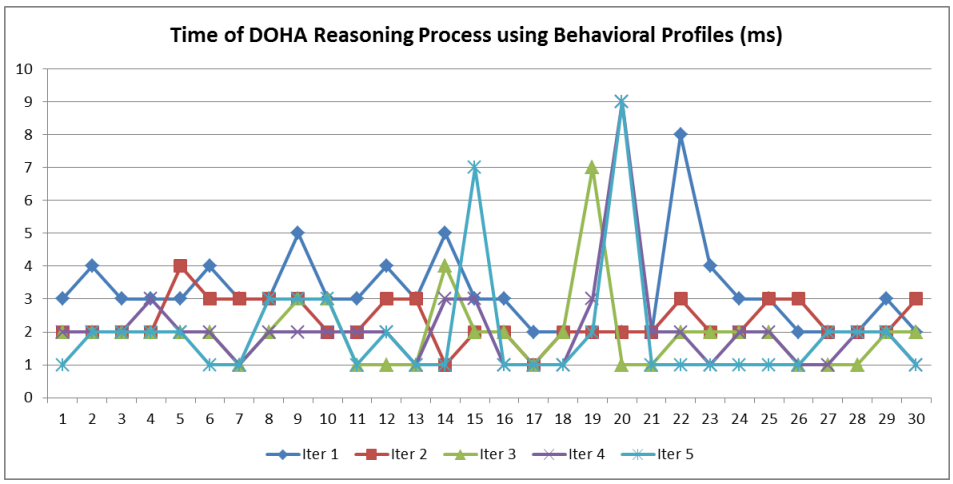


Figura 7.18: Caso de estudio 2: Evaluación de rendimiento considerando el tiempo de ejecución del razonador basado en perfiles de comportamiento de SenSE

7.6.2. Evaluación

Para llevar a cabo la evaluación se ha implementado un caso de estudio simple formado por cuatro servicios, de los cuales dos son de tipo dispositivo, uno compuesto y otro cliente puro. La Figura 7.19 muestra tanto los servicios, de luminosidad, calefacción, confort y usuario, como su relación con los elementos del mundo físico.

Para cada servicio se diseñó un perfil de comportamiento sobre el cual se definió el perfil de datos asociado. Veamos como ejemplo el perfil de comportamiento y el perfil de datos asociado a este para el servicio de iluminación, Código 7.1. El objetivo del perfil de comportamiento es que el servicio automáticamente mantenga siempre la luminosidad de la habitación con una intensidad de luz superior a $3lumens$, y en

base a esto establece la precondición del perfil. En caso de que la precondición no se cumpla, se regularía la luz de la habitación modificando su valor a través de las operaciones incluidas como reglas de la acción.

Código 7.1: Perfil de comportamiento del Servicio de Iluminción

```
<?xml version="1.0"?>
<behavioralProfile>
  <precondition>
    <rule>
      <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#
          ThisLocalion</subject>
      <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
          hasEnvironmentalCondition</predicate>
      <operator>Equal</operator>
      <object>LightD1</object>
    </rule>
    <rule>
      <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1
          </subject>
      <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
          hasValue</predicate>
      <operator>GreaterThan</operator>
      <object>2</object>
    </rule>
  </precondition>
  <action>
    <rule>
      <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1
          </subject>
      <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
          OperationTurnOnLight</predicate>
```

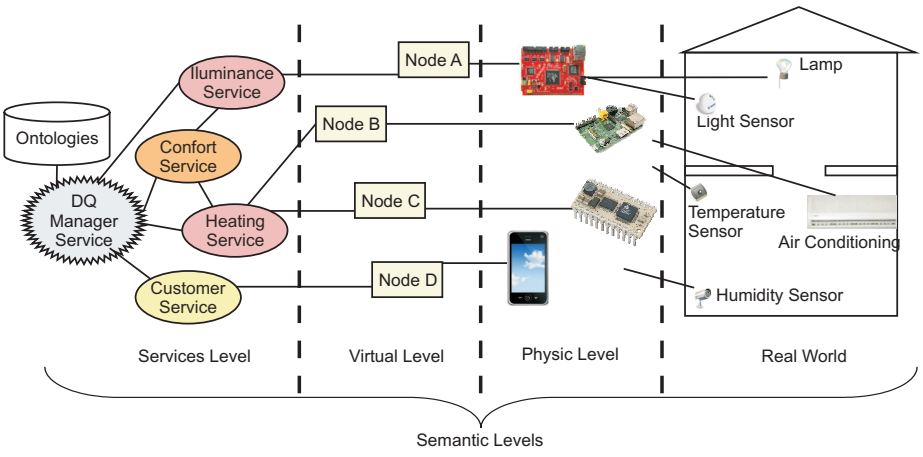


Figura 7.19: Caso de estudio 2: Diseño del caso de estudio relativo al uso de los perfiles de comportamiento como perfiles de datos para aplicar una metodología de evaluación basada en Data Quality

Tabla 7.11: Caso de estudio 2: Ejemplo de perfil de datos del Servicio de Iluminación

Who	How	What		When
Iluminance Service	OperationTurnOnLight() OperationSetLight()	LightD1	ON 3	LightD1.hasLocation()=ThisLocation LightD1.hasValue()=3

```
21        <operator>Equal</operator>
22        <object>ON</object>
23      </rule>
24      <rule>
25        <subject>http://core.ugr.es/doha/onto/codamos/Context.owl#LightD1
              </subject>
26        <predicate>http://core.ugr.es/doha/onto/codamos/Context.owl#
              OperationSetLight</predicate>
27        <operator>Equal</operator>
28        <object>3</object>
29      </rule>
30    </action>
31 </behavioralProfile>
```

En base a la información del perfil de comportamiento se obtendría el perfil de datos que se muestra en la Tabla 7.11. Como se puede observar, existe una relación directa entre los elementos contenidos en el perfil de comportamiento y el perfil de datos que evalúa la metodología DQ.

El proceso de evaluación se ha llevado a cabo mediante la implementación de un servicio especial al que se ha denominado *Data Quality Manager Service*. Conjugando la información semántica del sistema establecida por las ontologías y los perfiles de comportamiento proactivo de cada servicio, el "DQ Manager Service" aplica los perfiles de datos definidos y determina si el funcionamiento de cada servicio ha sido el correcto.

Las pruebas realizadas fueron satisfactorias y no se detectaron inexactitudes en el funcionamiento del sistema. Sin embargo, estos resultados estuvieron condicionados por las condiciones de ejecución, sin ruido ni perturbaciones severas.

Una ampliación de este caso de estudio está siendo implementado en un entorno real por parte del BIG en el entorno de la DCU. Aún no se han obtenido resultados concretos ni conclusiones definitivas que puedan ser incluidas aquí. Pero sin duda, es una muestra de la utilidad de la plataforma de servicios.

Parte III

Conclusiones y Aportaciones

Capítulo 8

Conclusiones, Trabajos Futuros y Lista de Publicaciones

Hay un momento en que todos los obstáculos se derrumban, todos los conflictos se apartan, y a uno se le ocurren cosas que no había soñado, y entonces no hay en la vida nada mejor que escribir.

· Gabriel García Márquez ·

8.1. Conclusiones y aportaciones

La realización y desarrollo de esta tesis doctoral nos ha permitido obtener las siguientes conclusiones como resultado:

Se ha diseñado y desarrollado una plataforma de servicios semánticos sensibles al contexto considerando las restricciones propias de los sistemas ubicuos. La plataforma posee una arquitectura descentralizada y distribuida basada en SOA. Las decisiones de diseño tomadas durante su construcción han estado muy ligadas al concepto de servicio como elemento fundamental.

Así, se ha diseñado un modelo de composición entre servicios basado en grafos dirigidos acíclicos. La aplicación de dicho modelo de composición asegura la colaboración entre servicios válida evitando la posible existencia de ciclos producidos por invocaciones recíprocas entre operaciones de distintos servicios.

La colaboración entre servicios es estática, pues el usuario define a-priori los servicios que componen el grafo de composición, con el pre-requisito de satisfacer las restricciones impuestas por el grado de complejidad de las operaciones involucradas en la colaboración. Del grado de complejidad se puede obtener información extendida de una operación concreta. A mayor grado de complejidad, mayor interacción con otros servicios, mayor posibilidad de problemas de red y mayor tiempo de ejecución.

Para la definición del modelo de composición se ha tenido en cuenta la necesidad de satisfacer restricciones temporales ligadas a la colaboración entre servicios. Gracias a ello es posible determinar el tiempo de ejecución de cada servicio y añadir esta información al árbol de composición, de forma que el tiempo de ejecución de la aplicación total queda acotado por el tiempo de ejecución de cada servicio que la compone.

La colaboración que propone el modelo está siempre dirigida por el proceso que actúa como cliente. Es más, puesto que el servicio consumidor puede consultar el

tiempo de ejecución de los posibles servicios proveedores antes de requerirlos, puede planificar la ejecución a llevar a cabo y determinar su coste en tiempo. Esto es una ventaja con respecto a otros modelos de composición, como por ejemplo los utilizados habitualmente en procesos de negocio. En ellos, los servicios consumidores desconocen cuando un servicio proveedor depende de un tercero cuya ejecución influirá en el tiempo de obtención de un resultado válido. Sin embargo, con el modelo de composición propuesto la cota de tiempo aporta a los servicios consumidores y a los desarrolladores información suficiente para determinar la idoneidad o no de invocar a un servicio proveedor determinado.

A más alto nivel, se ha dotado a los servicios de propiedades semánticas haciendo uso de las ontologías planteadas por el estándar OWL-S. La conjugación de la estructura de los servicios en base a la capa de interfaz, aplicación e interacción de la anatomía de servicios de DOHA planteada con propiedades semánticas permite añadir a los servicios propiedades de calidad de servicio o QoS, que además posibilita la selección dinámica de servicios. También a este nivel, se ha considerado una ontología para modelar el contexto y dotar a los servicios de propiedades de sensibilidad al contexto. En base a esto, los usuarios pueden determinar el comportamiento proactivo de los servicios mediante la configuración de perfiles de comportamiento personalizados que establecen sus preferencias con respecto al comportamiento del sistema.

El procesamiento de la información semántica de contexto se lleva a cabo siguiendo un procedimiento preestablecido. A partir de la sensorización y los perfiles de comportamiento definidos por los usuarios, junto con la percepción del contexto del entorno basado en ontologías, se aplica un razonamiento fundamentado en reglas a partir del cual es posible obtener un comportamiento proactivo por parte de los servicios, cuyo resultado final será operar sobre los actuadores de forma inteligente.

La implementación de la plataforma y su utilización en el desarrollo de distintos casos de estudio ha permitido llevar a cabo la evaluación de algunos de sus principales elementos, como son el modelo de composición y los perfiles de comportamiento proactivo. Las conclusiones obtenidas de dicha evaluación avalan el uso de la plataforma desarrollada para el desarrollo de aplicaciones para computación ubicua.

8.2. Trabajo futuro

Durante el análisis y diseño de la plataforma de servicios se han seguido pautas de Ingeniería del Software que han permitido obtener como resultado una plataforma sólida, que además es escalable y sostenible en el tiempo. La estructuración de la plataforma en distintas capas software permitirá que los ingenieros software puedan crear nuevas capas de funcionalidad compatibles. Así, la funcionalidad de la plataforma no quedará restringida a la desarrollada en esta tesis y podrá crecer de la mano de nuevos proyectos de investigación en el entorno de sistemas distribuidos para entornos ubicuos adaptándose a las necesidades que los nuevos tiempos vayan

demandando de ella.

Aunque la plataforma admite la inclusión de propiedades QoS en los servicios semánticos y la selección dinámica de sus instancias, no se ha llegado a diseñar un modelo de composición dinámica de servicios que satisfaga las restricciones de soft-real-time deseables. Como trabajo futuro podría estudiarse cómo extender DOHA-SenSE a nivel de colaboración de servicios para añadir un elemento al modelo a modo de gestor de composición que permitiese determinar en tiempo de ejecución qué servicios ofrecen la funcionalidad necesaria para desempeñar una determinada tarea y llevar a cabo su colaboración sin la necesidad de considerar un mapa de composición estático y predefinido.

En cuanto a la representación del contexto y su relación con los perfiles de comportamiento proactivo, la plataforma también podría ser mejorada. En la implementación actual los usuarios deben definir manualmente perfiles de comportamiento para que el sistema pueda llevar a cabo este tipo de funcionalidad, lo cual puede resultar tedioso y que para ello es necesario conocer todos los servicios disponibles y sus relaciones contextuales. Sería interesante proponer un elemento de alto nivel capaz de interpretar las acciones de los usuarios en el entorno y crear los perfiles de comportamiento en tiempo de ejecución. Evidentemente, inicialmente el comportamiento proactivo no sería posible, pues no habría información sobre los usuarios que emplear. Pero con el tiempo, el uso progresivo del sistema por parte de los usuarios podría generar información suficiente para llevar a cabo una deducción de sus necesidades o gustos particulares.

8.3. Lista de Publicaciones

En esta sección se recoge una lista de las publicaciones más relevantes en las que han aparecido los distintos resultados presentados en esta tesis. Se han agrupado dichas publicaciones en cuatro categorías correspondientes a revistas científicas, congresos internacionales, congresos nacionales y, por último, otras publicaciones que aún no estando directamente relacionadas con los temas tratados en esta tesis, han contribuido a mi formación en el área del Desarrollo de Software, los Sistemas Concurrentes y Distribuidos, y de los Lenguajes y Sistemas Informáticos.

Publicaciones en Revistas Científicas

1. Sandra Rodríguez-Valenzuela, Juan A. Holgado-Terriza, José M. Gutiérrez-Guerrero and Jesús L. Muros-Cobos (2014). *Distributed Service-Based Approach for Sensor Data Fusion in IoT Environments.* Sensors 2014, 14(10), 19200-19228; doi:10.3390/s141019200. ISSN: 1424-8220; CODEN: SENSC9 Índice de impacto (ISI): 2.245 (2014); 5-Year Impact Factor: 2.474 (2014)

2. Francisco Jesús Novo-Sánchez, Sandra Rodríguez-Valenzuela and Juan Antonio Holgado-Terriza (2014) *Multiprocessor support in Java.* Annals of Multicore and GPU Programming 2014, 1, 68-73. Editorial: Editorial Universidad de Granada ISSN: 2341-3158

Congresos Internacionales

1. Sandra Rodríguez-Valenzuela, Juan Antonio Holgado-Terriza, Plamen Petkov and Markus Helfert (2013). *Modeling Context-Awareness in a Pervasive Computing Middleware using Ontologies and Data Quality Profiles.* Evolving Ambient Intelligence Communications in Computer and Information Science - Volume 413, 2013, pp 271-282. 3rd International Workshop on Pervasive and Context-Aware Middleware (PerCAM13) in conjunction with Fourth International Joint Conference on Ambient Intelligence (AmI 2013). Dublin, 3-5 December 2013. ISBN: 978-3-319-04405-7 (2013)

2. Juan A. Holgado-Terriza and Sandra Rodríguez-Valenzuela (2011). *Services Composition Model for Home-Automation Peer-To-Peer Pervasive Computing.* Proceedings of the Federated Conference on Computer Science and Information Systems, pp. 527-536. 3rd International Symposium on Services Science. IEEE Computer Society Press. ISBN: 978-83-60810-39-2 (2011)

3. Sandra Rodríguez-Valenzuela and Juan A. Holgado-Terriza. *A Home-Automation Platform towards Ubiquitous Spaces based on a Decentralized P2P Architecture.* International Symposium on Distributed Computing and Artificial Intelligence, DCAI 2008. Advances In Soft Computing (Vol 50) (ISSN: 1615-3871), pp. 303-308, Springer Verlag. ISBN: 978-3-540-85862-1 (2009)

Congresos Nacionales

1. José Miguel Gutiérrez-Guerrero, Jesús Luis Muros-Cobos, Sandra Rodríguez-Valenzuela, Miguel Damas-Hermoso, Juan Antonio Holgado-Terriza. *Desarrollo de sistemas industriales mediante dispositivos empotrados basados en Java.* Actas de las IV Jornadas de Computación Empotrada (JCE), Madrid, 17-20 Septiembre 2013. ISBN: 978-84-695-8317-3 (2013)

2. Sandra Rodríguez-Valenzuela, Juan A. Holgado-Terriza, Jesús L. Muros-Cobos, José M. Gutiérrez-Guerrero. *Data Fusion Mechanism based on a Service Composition Model for the Internet of Things.* Actas de las III Jornadas de Computación Empotrada (JCE). ISBN: 978-84-695-4424-2 (2012)

3. Sandra S. Rodríguez-Valenzuela y Juan A. Holgado-Terriza. *A Services Platform for Home-Automation based on Decentralized P2P Architectures.* II Simposio en Desarrollo de Software, Universidad de Granada, Junio 2008, pp. 203-218. Departamento de Lenguajes y Sistemas Informáticos. ISBN: 978-84-96856-71-4 (2008)

4. Sandra S. Rodríguez-Valenzuela y Juan A. Holgado-Terriza. *Servicios Sensibles al Contexto en Sistemas de Computación Ubicua.* II Simposio en Desarrollo de Software, Universidad de Granada, Junio 2008, pp. 235-248. Departamento de Lenguajes y Sistemas Informáticos. ISBN: 978-84-96856-71-4 (2008)

5. Sandra S. Rodríguez-Valenzuela, María D. Serrano-Laguna y Juan A. Holgado-Terriza. *Control Domótico del Hogar a través de una Plataforma de Servicios Distribuida basada en JXTA*. II Simposio en Desarrollo de Software, Universidad de Granada, Junio 2008, pp. 219-234. Departamento de Lenguajes y Sistemas Informáticos. ISBN: 978-84-96856-71-4 (2008)

Otras Publicaciones

1. Manuel I. Capel Tuñón and Sandra Rodríguez-Valenzuela. *Distributed and Concurrent Systems (In Spanish): Teory and Practice. Vol. I and II.* Copycentro, Granada (Spain). ISBN: 84-15536-68-0 and 84-15536-69-7 (2012)

2. Manuel J. Baena-Toquero, Jesús L. Muros-Cobos, Sandra Rodríguez-Valenzuela and Juan A. Holgado-Terriza. *Towards sustainability in multimodal urban planners.* International Conference on Connected Vehicles and Expo (ICCVE), pp.492-497, 3-7 Nov. 2014. doi: 10.1109/ICCVE.2014.7297595

Bibliografía

Emile HL Aarts and Boris ER de Ruyter. New research perspectives on ambient intelligence. *JAISE*, 1(1):5–14, 2009.

Aitor Almeida and Diego López-de Ipiña. A distributed reasoning engine ecosystem for semantic context-management in smart environments. *Sensors*, 12(8):10208–10227, 2012.

Michele Amoretti, MariaChiara Laghi, Francesco Zanichelli, and Gianni Conte. Jxta-soap: Implementing service-oriented ubiquitous computing platforms for ambient assisted living. In Emile Aarts, JamesL. Crowley, Boris de Ruyter, Heinz Gerhäuser, Alexander Pflaum, Janina Schmidt, and Reiner Wichert, editors, *Ambient Intelligence*, volume 5355 of *Lecture Notes in Computer Science*, pages 75–90. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-89616-6. doi: 10.1007/978-3-540-89617-3_6. URL `http://dx.doi.org/10.1007/978-3-540-89617-3_6`.

G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance evaluation of jxta communication layers. In *2005 IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005*, volume 1, pages 251–258, 2005.

Apache. Apache jena. https://jena.apache.org/, 2014a. URL `https://jena.apache.org/`. [last visited on July 2014].

Apache. Jini technology. http://river.apache.org, 2014b. URL `http://river.apache.org`. [last visited on July 2014].

Jim Arlow and Ila Neustadt. *UML and the Unified Process. Practical Object-Oriented Analysis and Design.* Addison-Wesley, 2002.

Colin Atkinson and Thomas Kuhne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003. ISSN 0740-7459.

ATMEL. Avr4201: Pressure one (atavrsbpr1). Technical report, ATMEL, 2013. URL `http://www.atmel.com/Images/doc8355.pdf`.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010. ISSN 1389-1286.

A. Avizienis. The n-version approach to fault-tolerant software. *Software Engineering, IEEE Transactions on*, SE-11(12):1491 – 1501, dec. 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.231893.

F. Bagci, H. Schick, J. Petzold, W. Trumler, and T. Ungerer. Support of reflective mobile agents in a smart office environment. In *Lecture Notes in Computer Science*, volume 3432, pages 79–92, 2005.

Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4): 263–277, 2007.

G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges: An application model for pervasive computing. In *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*, pages 266–274, 2000.

L. Barolli and F. Xhafa. Jxta-overlay: A p2p platform for distributed, collaborative, and ubiquitous computing. *IEEE Transactions on Industrial Electronics*, 58(6): 2163–2172, 2011.

Christian Becker and Daniela Nicklas. Where do spatial context-models end and where do ontologies start? a proposal of a combined approach. In *Proceedings of the First International Workshop on Advanced Context Modelling, Reasoning and Management, in conjunction with UbiComp*, 2004.

Michael A Bender and Dana Ron. Testing properties of directed graphs: acyclicity and connectivity. *Random Structures and Algorithms*, 20(2):184–205, 2002.

A.M. Bernardos, P. Tarrio, and J.R. Casar. Casandra: A framework to provide context acquisition services and reasoning algorithms for ambient intelligence applications. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 372 –377, dec. 2009.

C. Bobda and A. Ahmadinia. Dynamic interconnection of reconfigurable modules on reconfigurable devices. *IEEE Design and Test of Computers*, 22(5):443–451, 2005.

J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-soa home control gateway. In *Proceedings - 2006 IEEE International Conference on Services Computing, SCC 2006*, pages 463–470, 2006.

Johann Bourcier, Ada Diaconescu, Philippe Lalanda, and Julie A McCann. Autohome: An autonomic management framework for pervasive home applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 6(1):8, 2011.

G. Branca and L. Atzori. A survey of soa technologies in ngn network architectures. *Communications Surveys Tutorials, IEEE*, 14(3):644–661, Third 2012. ISSN 1553-877X. doi: 10.1109/SURV.2011.051111.00127.

J. Bronsted, K. M. Hansen, and M. Ingstrup. A survey of service composition mechanisms in ubiquitous computing. In *Proceedings of UbiComp 2007 Workshop Innsbruck, Austria*, volume 4717, pages 87–92, 2007.

S. Buchholz, T. Hamann, and G. Hubsch. Comprehensive structured context profiles (cscp): design and experiences. In *Pervasive Computing and Communications*

Workshops, 2004. Proceedings of the Second IEEE Annual Conference on, pages 43–47, March 2004. doi: 10.1109/PERCOMW.2004.1276903.

J.F. Buford and H. Yu. Peer-to-peer networking and applications: Synopsis and research directions. In *Handbook of Peer-to-Peer Networking*, pages 3–45. Springer US, 2010. ISBN 978-0-387-09751-0.

H. Chang and K. Lee. A quality-driven web service composition methodology for ubiquitous services. *Journal of Information Science and Engineering*, 26(6):1957–1971, 2010. cited By (since 1996) 2.

Harry Chen, Tim Finin, and Anupam Joshi. Using owl in a pervasive computing broker, 2003.

Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18:197–207, 9 2004a. ISSN 1469-8005. doi: DOI:10.1017/S0269888904000025. URL `http://journals.cambridge.org/article_S0269888904000025`.

Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. Soupa: Standard ontology for ubiquitous and pervasive applications. In *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, pages 258–267. IEEE, 2004b.

CoDAMoS. Research project. created by davy preuveneers. https://distrinet.cs.kuleuven.be/projects/CoDAMoS/, 2004. URL `https://distrinet.cs.kuleuven.be/projects/CoDAMoS/`. [last visited on July 2014].

European Commission. Horizon 2020: The eu framework programme for research and innovation. http://ec.europa.eu/programmes/horizon2020/, 2014. URL `http://ec.europa.eu/programmes/horizon2020/`. [last visited on July 2014].

Diane J. Cook and Sajal K. Das. How smart are our environments? an updated look at the state of the art. *Pervasive Mob. Comput.*, 3(2):53–73, March 2007. ISSN 1574-1192. doi: 10.1016/j.pmcj.2006.12.001. URL `http://dx.doi.org/10.1016/j.pmcj.2006.12.001`.

Iván Corredor, Ana M Bernardos, Josué Iglesias, and José R Casar. Model-driven methodology for rapid deployment of smart spaces based on resource-oriented architectures. *Sensors*, 12(7):9286–9335, 2012.

Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: An introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2):86–93, March 2002. ISSN 1089-7801. doi: 10.1109/4236.991449. URL `http://dx.doi.org/10.1109/4236.991449`.

Sajal K. Das and Diane J. Cook. Designing and modeling smart environments (invited paper). In *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, WOWMOM '06, pages 490–494, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2593-8. doi: 10.1109/WOWMOM.2006.35. URL `http://dx.doi.org/10.1109/WOWMOM.2006.35`.

RL de Mántaras and L Saitta. *The use of temporal reasoning and management of complex events in smart homes.* IOS Press, 2004.

O. Dohndorf, J. Kruger, H. Krumm, C. Fiehe, A. Litvina, I. Luck, and F.-J. Stewing. Towards the web of things: Using dpws to bridge isolated osgi platforms. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 720 –725, 29 2010-april 2 2010.

Paul Dourish and Genevieve Bell. *Divining a digital future: Mess and mythology in ubiquitous computing.* MIT Press, 2011.

Paul Dourish and Genevieve Bell. resistance is futile": Reading science fiction alongside ubiquitous computing. *Personal Ubiquitous Comput.*, 18(4):769–778, April 2014. ISSN 1617-4909. doi: 10.1007/s00779-013-0678-7. URL `http://dx.doi.org/10.1007/s00779-013-0678-7`.

Nicolas Drosos, Eleni Christopoulou, and Achilles Kameas. Middleware for building ubiquitous computing applications using distributed objects. In Panayiotis Bozanis and EliasN. Houstis, editors, *Advances in Informatics*, volume 3746 of *Lecture Notes in Computer Science*, pages 256–266. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-29673-7. doi: 10.1007/11573036_24. URL `http://dx.doi.org/10.1007/11573036_24`.

Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005a.

Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005b.

D. Ejigu, M. Scuturici, and L. Brunie. Hybrid approach to collaborative context-aware service platform for pervasive computing. *Journal of Computers*, 3(1): 40–50, 2008.

T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. ISBN 0131858580.

I. Estévez-Ayres, P. Basanta-Val and M. García-Valls, J.A. Fisteus, and L. Almeida. Qos-aware real-time composition algorithms for service-based applications. *Industrial Informatics, IEEE Transactions on*, 5(3):278 –288, aug. 2009. ISSN 1551-3203. doi: 10.1109/TII.2009.2026422.

A Ferscha, M. Hechinger, A Riener, H. Schmitzberger, M. Franz, Md.S. Rocha, and A Zeidler. Context-aware profiles. In *Autonomic and Autonomous Systems, 2006. ICAS '06. 2006 International Conference on*, pages 48–48, July 2006. doi: 10.1109/ICAS.2006.18.

Tim Finin. Owl origin. URL `http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html`.

J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62 – 70, march-april 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.61.

Viktor E. Frankl. *Man's search for meaning: An introduction to logotherapy*. Simon & Schuster, 1984.

FreescaleSemiconductor. How to implement the freescale mpl115a digital barometer. Technical report, FreescaleSemiconductor, 2013a. URL `http://www.freescale.com/files/sensors/doc/app_note/AN3785.pdf`.

FreescaleSemiconductor. Miniature spi digital barometer. Technical report, FreescaleSemiconductor, 2013b. URL `http://cache.freescale.com/files/sensors/doc/data_sheet/MPL115A1.pdf?fpsp=1&WT_TYPE=Data%20Sheets&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation`.

Ciencia y Empresa de la Junta de Andalucía Fundación OPTI. Consejería de Innovación. *Estudio de prospectiva sobre el hogar digital*. Agencia de Innovación y Desarrollo de Andalucía IDEA, 2008. URL `http://books.google.es/books?id=2DF1QwAACAAJ`.

V. Gacitua-Decar and C. Pahl. Business model driven service architecture design for enterprise application integration. In *3rd Int. Conf. on Software and Data Technologies, ICSOFT 2008*, 2008.

Dragan Gasevic, Dragan Djuric, Vladan Devedzic, and Bran Selic. *Model driven architecture and ontology development*, volume 1. Springer, 2006.

J. Gerke, P. Reichl, and B. Stiller. Strategies for service composition in p2p networks. In *E-business and Telecommunication Networks*, volume 3, pages 62–77. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75993-5.

S.C. Geyik, B.K. Szymanski, and P. Zerfos. Robust dynamic service composition in sensor networks. *Services Computing, IEEE Transactions on*, 6(4):560–572, Oct 2013. ISSN 1939-1374. doi: 10.1109/TSC.2012.26.

M. Gharzouli and M. Boufaida. A distributed p2p-based architecture for semantic web services discovery and composition. In *New Technologies of Distributed Systems (NOTERE), 2010 10th Annual International Conference on*, pages 315–320, 2010.

A. Gárate, N. Herrasti, and A. López. Genio: An ambient intelligence application in home automation and entertainment environment. In *ACM International Conference Proceeding Series*, volume 121, pages 241–245, 2005a.

A. Gárate, I. Lucas, N. Herrasti, and A. López. Ambient intelligence as paradigm of a full automation process at home in a real application. In *Proceedings of IEEE International Symposium on Computational Intelligence in Robotics and Automation, CIRA*, pages 475–479, 2005b.

Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2013.01.010. URL `http://www.sciencedirect.com/science/article/pii/S0167739X13000241`. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications, Big Data, Scalable Analytics, and Beyond.

D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235, July 2010. ISSN 1939-1374. doi: 10.1109/TSC.2010.3.

Hani Hagras, Victor Callaghan, Martin Colley, Graham Clarke, Anthony Pounds-Cornish, and Hakan Duman. Creating an ambient-intelligence environment using embedded agents. *Intelligent Systems, IEEE*, 19(6):12–20, 2004.

D.L. Hall and J. Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1):6 –23, jan 1997. ISSN 0018-9219. doi: 10.1109/5.554205.

T. Haraikawa, K. Annen, G. Yamamoto, Y. Sakane, and Y. Takebayashi. Pibot: An adaptive and scalable appliance integration for a universal access of a home network. In *Digest of Technical Papers - IEEE International Conference on Consumer Electronics*, pages 407–408, 2005.

Pascal Hitzler, Markus Kroetzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press, 2009.

J. A. Holgado-Terriza and J. Viúdez-Aivar. A flexible java framework for embedded systems. In *ACM International Conference Proceeding Series*, pages 21–30, 2009.

JuanAntonio Holgado-Terriza and Jaime Viúdez-Aivar. Javaes, a flexible java framework for embedded systems. In M. Teresa Higuera-Toledano and Andy J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 323–355. Springer US, 2012. ISBN 978-1-4419-8157-8. doi: 10.1007/978-1-4419-8158-5_13. URL `http://dx.doi.org/10.1007/978-1-4419-8158-5_13`.

Finn Jensen. *An Introduction to Bayesian networks*. UCL Press, 1996.

L. Kagal, V. Korolev, H. Chen, A Joshi, and T. Finin. Centaurus: a framework for intelligent services in a mobile environment. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 195–201, Apr 2001. doi: 10.1109/CDCS.2001.918705.

Konstantinos Kakousis, Nearchos Paspallis, and George Angelos Papadopoulos. A survey of software adaptation in mobile and ubiquitous computing. *Enterprise Information Systems*, 4(4):355–389, 2010. doi: 10.1080/17517575.2010.509814.

Krasimira Kapitanova, Sang H. Son, and Kyoung-Don Kang. Using fuzzy logic for robust event detection in wireless sensor networks. *Ad Hoc Networks*, 10(4):709 – 722, 2012. ISSN 1570-8705. URL `http://www.sciencedirect.com/science/article/pii/S1570870511001326`. Advances in Ad Hoc Networks (II).

Rajesh Karunamurthy, Ferhat Khendek, and Roch H. Glitho. A novel architecture for web service composition. *Journal of Network and Computer Applications*, 35 (2):787–802, March 2012. ISSN 1084-8045. doi: 10.1016/j.jnca.2011.11.012.

L. Kawulok, K. Zielinski, and M. Jaeschke. Trusted group membership service for jxme (jxta4j2me). In *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005), IEEE International Conference on*, volume 4, pages 116 – 121 Vol. 4, 2005.

S. L. Kiani, M. Riaz, Y. Zhung, S. Lee, and Y. . Lee. A distributed middleware solution for context awareness in ubiquitous systems. In *Proceedings - 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 451–454, 2005.

Jaewoo Kim, Jaiyong Lee, Jaeho Kim, and Jaeseok Yun. M2m service platforms: Survey, issues, and enabling technologies. *Communications Surveys Tutorials, IEEE*, 16(1):61–76, First 2014. ISSN 1553-877X. doi: 10.1109/SURV.2013.100713.00203.

Dimitris Kiritsis. Closed-loop plm for intelligent products in the era of the internet of things. *Computer-Aided Design*, 43(5):479 – 501, 2011. ISSN 0010-4485. doi: 10.1016/j.cad.2010.03.002.

J. Knudsen. *Getting Started with JXTA for J2ME.*, 2002. URL `http://wireless.java.sun.com/midp/articles/jxme/jxta`.

M. Korotkiy and J. Top. Onto-soa: From ontology-enabled soa to service-enabled ontologies. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 124–124, Feb 2006. doi: 10.1109/AICT-ICIW.2006.141.

Peter Kostelnik, Martin Sarnovsk, and Karol Furdík. The semantic middleware for networked embedded systems applied in the internet of things and services

domain. *Scientific International Journal for Parallel and Distributed Computing*, 12(3):307–315, 2011. doi: 10.1080/17517575.2010.509814.

Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. ISSN 0166-5316.

S. Krco, D. Cleary, and D. Parker. Enabling ubiquitous sensor networking over mobile networks through peer-to-peer overlay networking. *Computer Communications*, 28(13 SPEC. ISS.):1586–1601, 2005.

P. Lalanda, L. Bellisard, and R. Balter. Asynchronous mediation for integrating business and operational processes. *Internet Computing*, 10(1):56–64, 2006.

Ora Lassila. Using the semantic web in mobile and ubiquitous computing. In Max Bramer and Vagan Terziyan, editors, *Industrial Applications of Semantic Web*, volume 188 of *IFIP The International Federation for Information Processing*, pages 19–25. Springer US, 2005. ISBN 978-0-387-28568-9.

S. Lee, Y. Lee, and H. Lee. *Jini-based ubiquitous computing middleware supporting event and context management services*, volume 4159 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. -, 2006a.

Y. Lee, S. Lee, and H. Lee. *Development of secure event service for ubiquitous computing*, volume 344 of *Lecture Notes in Control and Information Sciences*. -, 2006b.

Lejos. Behavioral programming: Programming behavior with lejos nxj. http://www.lejos.org/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm, 2014. URL `http://www.lejos.org/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm`. [last visited on July 2014].

Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. McGraw Hill, 2001.

Wu Li, Yann-Hang Lee, Wei-Tek Tsai, Jingjing Xu, Young-Sung Son, Jun-Hee Park, and Kyung-Duk Moon. Service-oriented smart home applications: composition, code generation, deployment, and execution. *Service Oriented Computing and Applications*, 6:65–79, 2012. ISSN 1863-2386. 10.1007/s11761-011-0086-7.

Seng Loke. *Context-aware pervasive systems: architectures for a new breed of applications*. CRC Press, 2006.

Qingcong Lv, Fan Yang, and Qiying Cao. A general qos-aware service composition model for ubiquitous computing. In *Intelligent Systems and Applications, 2009. ISA 2009. International Workshop on*, pages 1–4, 2009. doi: 10.1109/IWISA.2009.5072622.

Miriam Machuca, Miguel ángel López, Iván Marsá Maestre, and Juan Ramón Velasco Pérez. A contextual ontology to provide location-aware services and interfaces in smart environments. -, -:–, 2005.

Mohcine Madkour, Driss El Ghanami, Abdelilah Maach, and Abderrahim Hasbi. Context-aware service adaptation: An approach based on fuzzy sets and service composition. *Journal of Information Science and Engineering*, 29(1):1–16, 2013.

Alexander Maedche and Steffen Staab. *Ontology learning*. Springer, 2004.

David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing semantics to web services: The owl-s approach. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24328-1. doi: 10.1007/978-3-540-30581-1_4. URL `http://dx.doi.org/10.1007/978-3-540-30581-1_4`.

E. Meshkova, J. Riihijärvi, M. Petrova, and P. Mähönen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Computer Networks*, 52(11):2097–2128, 2008.

Meteormetric. Zambretti forecaster. Technical report, Meteormetric Limited, 2013. URL `http://www.meteormetrics.com/zambretti.htm`.

O. Moser, F. Rosenberg, and S. Dustdar. Domain-specific service selection for composite services. *Software Engineering, IEEE Transactions on*, 38(4):828–843, July 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.43.

M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K. Matsumoto. Constructing home network systems and integrated services using legacy home appliances and web services. *International Journal of Web Services Research*, 5(1):82–98, 2008.

Wenjia Niu, Gang Li, Zhijun Zhao, Hui Tang, and Zhongzhi Shi. Multi-granularity context model for dynamic web service composition. *Journal of Network and Computer Applications*, 34(1):312 – 326, 2011. ISSN 1084-8045. doi: http://dx.doi.org/10.1016/j.jnca.2010.07.014.

OASIS. Devices profile for web services version 1.1, 2009.

Oracle. Jxta technology. https://java.net/projects/jxta, 2014. URL `https://java.net/projects/jxta`. [last visited on July 2014].

Understanding Data Quality Management. In Oracle® Warehouse Builder User's Guide. Oracle Inc., 2008.

Shumao Ou, Nektarios Georgalas, Manooch Azmoodeh, Kun Yang, and Xiantang Sun. A model driven integration architecture for ontology-based context modelling and context-aware application development. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture, Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 188–197. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35909-8. doi: 10.1007/11787044_15. URL `http://dx.doi.org/10.1007/11787044_15`.

S.H. Park, S.H. Won, J.B. Lee, and S.W. Kim. Smart home: Digitally engineered domestic life. *Personal and Ubiquitous Computing*, 7(3):189–196, 2003.

C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys Tutorials, IEEE*, 16(1):414–454, First 2014. ISSN 1553-877X. doi: 10.1109/SURV.2013.042313.00197.

R. M. Pessoa, E. Silva, M. Van Sinderen, D. A. C. Quartel, and L. F. Pires. Enterprise interoperability with soa: A survey of service composition approaches. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pages 238–251, 2008.

G. Pinto, S. Medeiros, J. Souza, J. Strauch, and C. Marques. Spatial data integration in a collaborative design framework. *Commun. ACM*, 46:86–90, 2003.

Bogdan Pogorelc, Radu-Daniel Vatavu, Artur Lugmayr, Björn Stockleben, Thomas Risse, Juha Kaario, Estefania Constanza Lomonaco, and Matjaž Gams. Semantic ambient media: From ambient advertising to ambient-assisted living. *Multimedia Tools and Applications*, 58(2):399–425, 2012.

Maria Poveda Villalon, Mari Carmen Suárez-Figueroa, Raúl García-Castro, and Asunción Gómez-Pérez. A context ontology for mobile environments. In *Workshop on Context, Information and Ontologies (CIAO 2010) co-located with EKAW 2010, Lisbon*. CEUR-WS, 2010.

Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. In Panos Markopoulos, Berry Eggen, Emile Aarts, and JamesL. Crowley, editors, *Ambient Intelligence*, volume 3295 of *Lecture Notes in Computer Science*, pages 148–159. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23721-1. doi: 10.1007/978-3-540-30473-9_15. URL `http://dx.doi.org/10.1007/978-3-540-30473-9_15`.

A. Rakotonirainy, J. Indulska, S.W. Loke, and A.B. Zaslavsky. Middleware for reactive components: An integrated use of context, roles, and event based coordination. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 77–98, London, UK, 2001. Springer-Verlag. ISBN 3-540-42800-3.

A.K. Ramakrishnan, D. Preuveneers, and Y. Berbers. A loosely coupled and distributed bayesian framework for multi-context recognition in dynamic ubiquitous environments. In *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, pages 270–277, Dec 2013. doi: 10.1109/UIC-ATC.2013.66.

Raspberry-Pi. Raspberry-pi manual. Technical report, Raspberry-Pi, 2013. URL `http://www.raspberrypi.org/`.

Danny Raz, Arto Tapani Juhola, Joan Serrat-Fernandez, and Alex Galis. *Fast and efficient context-aware services.* John Wiley & Sons, 2006.

L. Ribeiro, J. Barata, A. Colombo, and F. Jammes. A generic communication interface for dpws-based web services. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 762–767, July 2008. doi: 10.1109/INDIN.2008.4618204.

V. Ricquebourg, D. Menga, D. Durand, B. Marhic, L. Delahoche, and C. Logé. The smart home concept: Our immediate future. In *2006 1st IEEE International Conference on E-Learning in Industrial Electronics, ICELIE*, pages 23–28, 2006.

S. Rodríguez and J.A. Holgado. A home-automation platform towards ubiquitous spaces based on a decentralized p2p architecture. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, pages 304–308. Springer Berlin / Heidelberg, 2009.

Sandra Rodríguez-Valenzuela, Juan A. Holgado-Terriza, Plamen Petkov, and Markus Helfert. Modeling context-awareness in a pervasive computing middleware using ontologies and data quality profiles. In Michael J. O'Grady, Hamed Vahdat-Nejad, Klaus-Hendrik Wolf, Mauro Dragone, Juan Ye, Carsten Rocker, and Gregory O'Hare, editors, *Evolving Ambient Intelligence*, volume 413 of *Communications in Computer and Information Science*, pages 271–282. Springer International Publishing, 2013. ISBN 978-3-319-04405-7. doi: 10.1007/978-3-319-04406-4_28.

Sandra Rodríguez-Valenzuela, Juan A Holgado-Terriza, José M Gutiérrez-Guerrero, and Jesús L Muros-Cobos. Distributed service-based approach for sensor data fusion in iot environments. *Sensors*, 14(10):19200–19228, 2014.

Cristobal Romero Morales, Fco. Javier Vazquez Serrano, and Carlos de Castro Lozano. *Domótica e Inmótica. Viviendas y edificios inteligentes.* RA-MA, 2010.

D. Saha and A. Mukherjee. Pervasive computing: A paradigm for the 21st century. *Computer*, 36(3):25–31+4, 2003.

Maria J Santofimia, Scott E Fahlman, Xavier del Toro, Francisco Moya, and Juan C Lopez. A semantic model for actions and events in ambient intelligence. *Engineering Applications of Artificial Intelligence*, 24(8):1432–1445, 2011.

S. Sarkka, V.V. Viikari, M. Huusko, and K. Jaakkola. Phase-based uhf rfid tracking with nonlinear kalman filtering and smoothing. *Sensors Journal, IEEE*, 12(5): 904–910, May 2012. ISSN 1530-437X. doi: 10.1109/JSEN.2011.2164062.

M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

SemWebCentral. Owl-s editor plugin to protege ontology editor. http://owlseditor.semwebcentral.org/index.shtml, 2014. URL `http://owlseditor.semwebcentral.org/index.shtml`. [last visited on July 2014].

Sensirion. Datasheet sht7x (sht71, sht75) humidity and temperature sensor ic. Technical report, Sensirion, 2013. URL `http://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/Humidity/Sensirion_Humidity_SHT7x_Datasheet_V5.pdf`.

Bosch Sensortec. Datasheet bmp085 digital pressure sensor. Technical report, Bosch, 2013. URL `http://www.adafruit.com/datasheets/BMP085_DataSheet_Rev.1.0_01July2008.pdf`.

Richard Simpson, Debra Schreckenghost, EdmundF. LoPresti, and Ned Kirsch. Plans and planning in smart homes. In JuanCarlos Augusto and ChrisD. Nugent, editors, *Designing Smart Homes*, volume 4008 of *Lecture Notes in Computer Science*, pages 71–84. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35994-4. doi: 10.1007/11788485_5. URL `http://dx.doi.org/10.1007/11788485_5`.

I. Sommerville. *Software Engineering*. Addison Wesley, 2005.

Sourceforge. Owl-s api. http://owlapi.sourceforge.net/, 2014. URL `http://owlapi.sourceforge.net/`. [last visited on July 2014].

ThanosG. Stavropoulos, Dimitris Vrakas, and Ioannis Vlahavas. A survey of service composition in ambient intelligence environments. *Artificial Intelligence Review*, 40(3):247–270, 2013. ISSN 0269-2821. doi: 10.1007/s10462-011-9283-1. URL `http://dx.doi.org/10.1007/s10462-011-9283-1`.

Z. Stojanovic and A. Dahanayake. *Service-oriented software system engineering: challenges and practices*. Idea, 2005.

Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *Workshop Proceedings*, 2004.

Thomas Strang, Claudia Linnhoff-Popien, and Korbinian Frank. Cool: A context ontology language to enable contextual interoperability. In Jean-Bernard Stefani, Isabelle Demeure, and Daniel Hagimont, editors, *Distributed Applications and Interoperable Systems*, volume 2893 of *Lecture Notes in Computer Science*, pages 236–247. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20529-6. doi: 10.1007/978-3-540-40010-3_21. URL `http://dx.doi.org/10.1007/978-3-540-40010-3_21`.

Katia Sycara and David Martin. Tools and technologies for semantic web services: An owl-s perspective. Technical report, Carnegie Mellon University and Artificial Intelligence Center (SRI International), 2006.

Maurice H Ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 1 (5):1–10, 2007.

B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.C. Hugly, E. Pouyoul, and B. Yeager. *Project JXTA 2.0 super-peer virtual network*, 2003.

M. Umberger, I. Humar, A. Kos, J. Guna, A. Zemva, and J. Beter. The integration of home-automation and iptv system and services. *Computer Standards and Interfaces*, 31(4):675–684, 2009.

UniversityOfMalta. Owl-s editor to semantically annotate web-services. university of malta. http://staff.um.edu.mt/cabe2/supervising/undergraduate/owlseditFYP/OwlSEdit.html, 2014. URL `http://staff.um.edu.mt/cabe2/supervising/undergraduate/owlseditFYP/OwlSEdit.html`. [last visited on July 2014].

A. Urbieta, G. Barrutieta, J. Parra, and A. Uribarren. Estado del arte de composición dinámica de servicios en entornos de computación ubicua. In -, 2007.

D. Valtchev and I. Frankov. Service gateway architecture for a smart home. *IEEE Communications Magazine*, 40(4):126–132, 2002.

W. Van Der Aalst. Service mining: Using process mining to discover, check, and improve service behavior. *Services Computing, IEEE Transactions on*, 6(4):525–535, Oct 2013. ISSN 1939-1374. doi: 10.1109/TSC.2012.25.

V Vanitha and V Palanisamy. A global qos-aware service composition in wireless sensor networks. *International Journal of Computer Applications*, 1-19:0975 – 8887, 2010.

J.I. Vazquez, I. Sedano, and D. López de Ipiña. Soam: A web-powered architecture for designing and deploying pervasive semantic devices. *International Journal of Web Information Systems*, 2(3):212–224, 2007.

Juan Ignacio Vazquez, Diego López De Ipiña, and Iñigo Sedano. Soam: an environment adaptation model for the pervasive semantic web. In *Computational Science and Its Applications-ICCSA 2006*, pages 108–117. Springer, 2006.

Felix Jesús Villanueva, Francisco Moya, Fernando Santofimia, David Villa, Jesús Barba, and Juan Carlos López. Towards a unified middleware for ubiquitous and pervasive computing. *IJACI*, 1(1):53–63, 2009.

W3C. Owl-s coalition: Owl-s: Semantic markup for web services. w3c member submission. http://www.w3.org/Submission/OWL-S/, 2014a. URL `http://www.w3.org/Submission/OWL-S/`. [last visited on July 2014].

W3C. Resource description framework (rdf). http://www.w3.org/RDF/, 2014b. URL `http://www.w3.org/RDF/`. [last visited on July 2014].

W3C. World wide web consortium. http://www.w3c.org, 2014c. URL `http://www.w3c.org`. [last visited on July 2014].

K. Wacks. Home systems standards: Achievements and challenges. *IEEE Communications Magazine*, 40(4):152–159, 2002.

X.H. Wang, Da Qing Zhang, Tao Gu, and H.K. Pung. Ontology based context modeling and reasoning using owl. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22, March 2004. doi: 10.1109/PERCOMW.2004.1276898.

Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, July 1993. ISSN 0001-0782. doi: 10.1145/159544.159617. URL `http://doi.acm.org/10.1145/159544.159617`.

Mark Weiser and JohnSeely Brown. The coming age of calm technology. In *Beyond Calculation*, pages 75–85. Springer New York, 1997. ISBN 978-0-387-98588-6. doi: 10.1007/978-1-4612-0685-9_6. URL `http://dx.doi.org/10.1007/978-1-4612-0685-9_6`.

B. Xu, Q. Gao, and X. Yang. Extensions to jini service architecture for pervasive computing. In *SPCA 2006: 2006 First International Symposium on Pervasive Computing and Applications, Proceedings*, pages 90–94, 2007.

Juan Ye, Graeme Stevenson, and Simon Dobson. A top-level ontology for smart environments. *Pervasive Mob. Comput.*, 7(3):359–378, June 2011. ISSN 1574-1192. doi: 10.1016/j.pmcj.2011.02.002. URL `http://dx.doi.org/10.1016/j.pmcj.2011.02.002`.

E. Zeeb, A. Bobek, H. Bohn, and F. Golatowski. Service-oriented architectures for embedded systems using devices profile for web services. In *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, volume 1, pages 956–963, May 2007. doi: 10.1109/AINAW.2007.330.

E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski. Ws4d: Toolkits for networked embedded systems based on the devices profile for web services. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 1–8, Sept 2010. doi: 10.1109/ICPPW.2010.11.

Daqiang Zhang, Hongyu Huang, Chin-Feng Lai, Xuedong Liang, Qin Zou, and Minyi Guo. Survey on context-awareness in ubiquitous media. *Multimedia Tools and Applications*, 67(1):179–211, 2013. ISSN 1380-7501. doi: 10.1007/s11042-011-0940-9. URL `http://dx.doi.org/10.1007/s11042-011-0940-9`.

J. Zhou, E. Gilman, M. Ylianttila, and J. Riekki. Pervasive service computing: Visions and challenges. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, pages 1335–1339, 2010. cited By (since 1996) 0.