UNIVERSIDAD DE SALAMANCA

Departamento de Informática y Automática

APIs Semánticas para la Web Orientada a Datos Enlazados

Tesis Doctoral

D. Antonio Garrote Hernández

Directora:

Dra. Dña. María N. Moreno García

Enero 2014

La memoria titulada **"APIs Semánticas para la Web Orientada a Datos Enlazados"**, que presenta D. Antonio Garrote Hernández, para optar al Grado de Doctor por la Universidad de Salamanca, ha sido realizada bajo la dirección de la Dra. Dña. María N. Moreno García, Profesora Titular de Universidad del Departamento de Informática y Automática de la Universidad de Salamanca.

Salamanca, Enero de 2014.

La directora,

Fdo: Dra. Dña. María N. Moreno García

El doctorando,

Fdo: D. Antonio Garrote Hernández

Agradecimientos

Me gustaría agradecer especialmente a mi directora de Tesis María N. Moreno García por su ayuda, paciencia y apoyo durante todo el proceso de elaboración de la tesis. Sin su auyuda, no hubiera sido posible terminar este trabajo. También me gustaría dar las gracias a mi familia: mis padres y mi hermano, mis compañeros de trabajo en Unkasoft en España, XING en Alemania, Forward en el Reino Unido, así como al resto de personas que me han apoyado durante estos años de trabajo e investigación, especialmente a Helena Martín Hernández y Ana María Flores Castro sin las cuales no habría podido terminar este documento.

APIs Semánticas para la Web Orientada a Datos Enlazados

por

Antonio Garrote Hernández

Resumen

Uno de los principales problemas que se plantean en el desarrollo actual de aplicaciones web y móviles consiste en la definición de Interfaces de Programación de Aplicaciones (APIs) entre clientes y proveedores de datos. Nuestra propuesta de solución para este problema se basa en el uso de componentes básicos desarrollados en el área de la Web Semántica, la iniciativa de Datos Enlazados Abiertos y los principios arquitecturales REST con el fin de desarrollar *APIs Semánticas de Datos Enlazados* que puedan usarse para implementar fácilmente interfaces de datos web y al mismo tiempo, asegurar la interoperabilidad entre APIs de diferentes proveedores de datos. En este documento caracterizamos el concepto de *API Semántica de Datos Enlazados* a través de la definición de un modelo formal que permite describir las interacciones entre clientes y servidores web de acuerdo con los principios REST para a continuación, transformar este modelo teórico en un modelo arquitectónico implementable haciendo uso de elementos semánticos esenciales. Por último, describimos nuestra implementación de algunos de los componentes software críticos en la arquitectura propuesta y mostramos algunos ejemplos de aplicación en los que estos componentes son usados para resolver problemas concretos de desarrollo web, con el fin de demostrar la aplicabilidad de la solución propuesta.

Supervisor de Tesis: María N. Moreno García

Semantic APIs for the Web of Linked Data

by

Antonio Garrote Hernández

Abstract

One of the main problems that can be found nowadays in the development of web and mobile applications is the definition of Application Programming Interfaces (API) between client and data providers. Our proposed solution to this problem is based on the use of basic components developed in the Semantic Web, the Linked Open Data fields and REST architectural principles as the foundation to build *Linked Data Semantic APIs* that can be used to easily develop web data intefaces and at the same time, enforce interoperability among different data providers. In this document the concept of *Linked Data Semantic API* is characterised by means of defining a formal model describing interactions between clients and web servers according to REST principles. This theoretical model is then transformed into a ready to implement architectural solution using essential semantic elements. The implementation of some of the most critical components of this architectural model is then described. Lastly, we show how the software implementation of the architecture can be used to solve real web development problems in order to demonstrate the applicability of the proposed solution.

Thesis Supervisor: María N. Moreno García

Índice general

Índice de figuras

Índice de cuadros

Lista de Acrónimos

API Application Programming Interface

BGP Basic Graph Pattern

CCS Calculus of Communicating Systems

CSP Communicating Sequential Processes

CSS Cascading Style Sheets

CURIEs Compact URIs

CWA Closed World Assumption

DNS Domain Name Service

DOM Document Object Model

EBNF Extended Backus-Naur Form

EL OWL 2 EL Profile

ESB Enterprise Service Bus

FOAF Friend Of A Friend

HATEOAS Hypermedia as the Engine of the Application State

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

ICV Integrity Constraint Validation

IP Internet Protocol

JSON JavaScript Object Notation

JSON-LD JSON for Linking Data

JSONP JSON with Padding

LRDD Links Based Descriptor Discovery

LUBM Lehigh University Benchmark

OLD Open Linked Data

OWA Open World Assumption

OWL Ontology Web Language

OWL-S Semantic Markup for Web Services

PEG Parsing Grammar Expression

QL OWL 2 Query Language Profile

R2RML RDF to RDF Mapping Language

RDF Resource Description Framework

RDFa RDF in Attributes

RDFS RDF Schema

REST Representational State Transfer

RL OWL 2 Query Rules Language

RPC Remote Procedure Call

SAWSDL Semantic Annotations for WDSL

SIOC Semantically-Interlinked Online Communities

SOA Service Oriented Architectures

SOAP Simple Object Access Protocol

SQL Structured Query Language

SVG Scalar Vector Graphics

TCP Transmission Control Protocol

UDDI Universal Description Discovery and Integration of Web Services

UNA Unique Name Assumption

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

WADL Web Application Description Language

WS-* Conjunto de especificaciones de Servicios Web básados en WSDL y SOAP

WS-BPEL Web Services Business Process Execution Language

WS-CDL Web Services Choreography Description Language

WSDL Web Services Description Language

WSMO Web Services Modeling Ontology

WSMX Web Services Execution Environment

XHTML eXtensible HyperText Markup Language

XLink XML Linking Language

XML eXtensible Markup Language

Capítulo 1

Introducción

En el periodo de tiempo transcurrido desde la generalización del uso de la World Wide Web, a mediados de la década de 1990 hasta el momento actual, la riqueza y sofisticación del contenido disponible para los usuarios de la web se ha incrementado. El salto de complejidad que media desde la concepción de la web como un conjunto de documentos HTML enlazados entre sí por hipervínculos, hasta la aparición de las primeras aplicaciones web o la construcción de las muchas redes sociales disponibles hoy en día, ha supuesto un desafío constante para las personas involucradas en el desarrollo de las tecnologías web que han hecho posible tal evolución.

Lejos de detenerse o incluso frenarse, el concepto de aplicación web sigue transformándose y enfrentándose a nuevos desafíos que deben ser solventados con el desarrollo de soluciones tecnológicas que hagan posible ofrecer nuevos servicios capaces de solucionar los problemas de los usuarios actuales de la web.

A continuación enumeraremos algunos de los principales problemas abiertos que encuentran los usuarios de aplicaciones web actuales y que no han sido todavía resueltos desde el punto de vista tecnológico de una forma óptima:

- **Agregación de fuentes de datos:** Un problema común entre los aplicaciones web actuales, especialmente en el área de las redes sociales, es el de la fragmentación de los datos del usuario: fotos, contactos, listas de reproducción musical, etc. entre diversos servicios, de forma tal que el intercambio de estos datos entre

estos diferentes servicios es complicado, cuándo no imposible, más allá de soluciones ad-hoc que no pueden ser reutilizadas entre aplicaciones o intentos de estandarización por parte de la industria de mecanismos de autenticación para un usuario entre los que cabe destcar *OAuth* [49].

- **Reconciliación de recursos:** El problema del intercambio de datos entre diferentes aplicaciones no se limita a la adquisición de estos datos y a la implementación de la correspondiente capa de privacidad y seguridad en el acceso, sino que tiene una implicación más fundamental al nivel semántico en cuanto a la denotación de la naturaleza e identidad de los recursos disponibles en los diferentes servicios. Esto significa que en un entorno abierto y cambiante, donde nuevos servicios y aplicaciones aparecen y desaparecen ofreciendo nuevos tipos de recursos, debe ser posible establecer asociaciones entre estos recursos así como determinar su naturaleza, para poder ofrecer a partir de ese plano semántico, algún tipo de servicio útil para el usuario final. Esto significa el establecimiento de un vocabulario común y extensible para los recursos web, así como el de un mecanismo igualmente extensible para la designación, identificación y recuperación de dichos recursos entre diferentes aplicaciones.

- **Serialización de datos:** Otro importante escollo para la construcción de aplicaciones web distribuidas viene dado por el hecho de que aún cuando sea posible conocer la identidad y naturaleza de un recurso disponible en una aplicación web, y el acceso a dicho recurso venga dado por un mecanismo estándar, todavía sería necesario ofrecer una representación concreta de dicho recurso para la aplicación cliente de forma tal que pueda satisfacer la funcionalidad deseada por el usuario. Es por lo tanto necesario ofrecer un mecanismo estándar y extensible para decidir el paso del nivel semántico a la representación sintáctica del recurso, es decir, su serialización, o si así se prefiere, el formato concreto en el que se van a consumir los datos asociados del recurso.

Estos tres problemas combinados afectan a la arquitectura, diseño e implementación de la mayoría de aplicaciones web y servicios sociales en uso hoy en día como

Facebook, *Twitter*, *Google+*, etc. Lo que supone importantes consecuencias para los usuarios finales de dichos servicios, ya que la complejidad en la interacción entre servicios hace muy difícil la implementación de nuevas funcionalidades capaces de combinar datos almacenados en distintos servicios con datos ajenos a ellos, ya que esto supone la implementación de los diferentes mecanismos de acceso, la elección o implementación de los componentes software capaces de interpretar las representaciones accesibles y traducir la semántica de los recursos recuperados a una ontología común que será utilizada para implementar la lógica del servicio agregador de datos. La complejidad de este proceso es un obstáculo para el usuario a la hora de utilizar los datos que han generado, y de los que es autor, fuera del servicio que usó para generar dichos datos y, por último, se traduce en una fragmentación de la identidad de un usuario en la web a través de una nube de aplicaciones y servicios, que no controla y de la que puede ser privado de forma arbitraria por los propietarios de dichos servicios.

Esta concepción de las aplicaciones y servicios web como sistemas cerrados, centralizados y no inter-operables, más allá de una forma muy limitada de conectividad, contrasta de forma marcada con la arquitectura de la web: abierta, distribuida y construida sobre estándares, que buscan asegurar la inter-operabilidad entre servicios, tal y como ha sido recogida y documentada por sus creadores [10] y en principios aceptados ampliamente por la comunidad dedicada al desarrollo web como los principios *REST* (Representational State Transfer) [40].

1.1. Desarrollo de la iniciativa Web Semántica

Al mismo tiempo que estos nuevos tipos de aplicaciones y servicios web eran construidos por el grueso de la comunidad de desarrolladores web, la comunidad dedicada a la investigación sobre Web Semántica, eminentemente académica, ha ido generando, desde su creación hace más de diez años, una serie de ideas que se han traducido más tarde en estándares que tienen el potencial para solucionar algunos

de de los problemas anteriormente mencionados. Entre estos estándares podemos mencionar las siguientes tecnologías:

- ***RDF* [72], un modelo de datos estándar y con una semántica formal:** que puede ser utilizado para poder integrar datos entre diferentes aplicaciones y servicios web, usando un grafo donde se establecen relaciones entre recursos usando el mecanismo estándar de la web, el hipervínculo.

- ***OWL* [83], un lenguaje extensible para la definición de ontologías:** definido sobre el modelo de datos propio de *RDF*, ofrece un mecanismo eficiente para que diferentes aplicaciones describan el contenido semántico de los datos ofrecidos mediante la construcción de ontologías o la reutilización de ontologías ya existentes. El uso de *RDF* como la base para la descripción de ontologías *OWL* asegura que la recuperación e integración de la ontología describiendo la semántica de un recurso web en particular sólo supone seguir un hipervínculo hasta el documento *RDF* que contiene la ontología quedando de este modo el recurso integrado en el grafo que describe todas las entidades que ya conoce el agente. *OWL* a su vez tiene una semántica formal de Mundo Abierto [65] definida estrictamente usando el modelo teórico que supone la Lógica Descriptiva, lo que permite comprobar la validez y consistencia del resultado de agregar ontologías provenientes de diferentes fuentes en un sólo modelo.

- ***SPARQL* [104], un lenguaje de consulta:** que permite realizar consultas sobre un grafo *RDF* incluyendo entidades provenientes de diferentes servicios y aplicaciones y que permite recuperar la información necesaria para que un determinado agente web lleve a cabo su cometido.

- **Ontologías *OWL* estándar como *Dublin Core* [131], *SIOC* [18], etc:** así como otro gran número de ontologías. Diferentes dominios de aplicación, desde el farmacéutico [33] hasta la venta en línea [54], pueden ser utilizados directamente por los desarrolladores de aplicaciones para describir su contenido usando un vocabulario común que facilite la inter-operabilidad entre los diferentes servicios web.

Sin embargo, y a pesar del potencial de estás tecnologías para solucionar algunos de los problemas más importantes de integración de datos en las aplicaciones web modernas, el uso de dichos estándares por parte de la gran mayoría de la comunidad de desarrolladores web ha sido muy limitado, viéndose el interés por la Web Semántica restringido a la comunidad académica y a desarrolladores en dominios de aplicación muy particulares, como el farmacéutico.
La principal objeción esgrimida por el grueso de desarrolladores web a la hora de intentar utilizar el conjunto de tecnologías desarrolladas para la web semántica se centra en la excesiva complejidad de dichas tecnologías. La base de esta crítica puede estar justificada por la fuerte influencia que la academia ha tenido en el desarrollo de estas tecnologías, haciendo hincapié en problemas de gran complejidad teórica, como la inferencia lógica, en detrimento problemas más sencillos, pero de aplicación potencial más generalizada o de los detalles concretos de implementación que son necesarios para la aplicación práctica de los desarollos teóricos realizados, como los formatos para la serialización de grafos *RDF*. Como consecuencia de todo esto, alternativas teóricamente menos idóneas desde el punto de vista formal se han impuesto como las opciones tecnológicas preferidas para el desarrollo web por la inmensa mayoría de programadores, por ejemplo, el uso de objetos *JSON* [29] sin concepto de identidad, no extensible, sin una posibilidad estándar para enlazar unos objetos con otros, como el formato de intercambio de datos universal de las aplicaciones web sociales, en vez de cualquier posible serialización de *RDF*.

1.2. El Enfoque *Open Linked Data*

Con el objetivo de intentar ofrecer una versión de la Web Semántica más pragmática y fácil de utilizar por la mayoría de desarrolladores web surge la iniciativa *Open Linked Data* o Datos Enlazados Abiertos. Desde este enfoque, la visión de lo que supone la Web Semántica se desplaza de problemas fuertes, como la inferencia lógica sobre los datos, hacia problemas más simples pero fundamentales para el desarrollo

web práctico, como el intercambio y la integración de datos.

Para conseguir este fin, la comunidad *Open Linked Data* ha adoptado aquellos estándares propuestos como parte de la Web Semántica y los ha adaptado a las necesidades del desarrollo web más genérico, por ejemplo, ofreciendo serializaciones de *RDF* simples basadas en *JSON* [120] o *HTML* [1], con el objetivo de que sean una opción realista para la gran mayoría de desarrolladores web. A su vez, la comunidad *Open Linked Data* también ha propuesto nuevos estándares para solucionar otros problemas básicos en la integración de datos en aplicaciones web como la autenticación, como por ejemplo *WebID* [62].

Tanto a la hora de adaptar tecnologías web semánticas como a la hora de proponer nuevos estándares, la comunidad *Open Linked Data* siempre ha intentado guiarse por los principios *REST* de arquitectura web, ampliamente aceptados por la mayoría de los desarrolladores web, incluyendo prácticas como la negociación de contenido [63] o clarificando la distinción entre recursos de información y no información [39]. Desde el punto de vista Open Linked Data, sus propuestas no son más que la extensión de los principios arquitecturales *REST* al intercambio de datos, reutilizando allí donde sea posible el trabajo llevado cabo por la comunidad impulsora de la Web Semántica, en lugar de proponer nuevas soluciones desde cero.

1.3. Objetivos de esta tesis

El objetivo que se propone esta tesis sigue la línea de trabajo propuesta por la iniciativa *Open Linked Data* pero situando nuestro foco de atención en la definición y desarrollo de *interfaces de programación de aplicaciones* (*APIs*) semánticas.

Desde nuestro punto de vista, las *APIs* de datos son el componente esencial de las aplicaciones web modernas.

La *API* de una aplicación determina los datos y recursos que van a estar disponibles para ser accedidos por otros agentes web con el fin de llevar a cabo sus funcionalidades y, por lo tanto, el grado en que los datos generados por una aplicación web pueden ser reutilizados e integrados con los datos de otras aplicaciones y servicios. Otros factores

que han incrementado la importancia del desarrollo de *APIs* de aplicaciones son el enorme crecimiento del mercado de aplicaciones para dispositivos móviles necesitan tener acceso a los datos almacenados en el *backend* de una aplicación web a través de una *API* de datos, así como el crecimiento de los clientes pesados *JavaScript* en las aplicaciones web de escritorio, que se conectan de la misma manera que un cliente nativo móvil a la *API* de datos de la aplicación.
Con el fin de alcanzar el objetivo de ofrecer una solución al problema de caracterización de las APIs semánticas el trabajo se ha abordado desde diferentes niveles de abstracción intentando alcanzar las siguientes metas:

- Definición de un **modelo formal** que permita expresar diferentes tipos de computación como un conjunto de servicios y agentes intercambiando datos a través de APIs semánticas.

- Especificación de una **arquitectura *software*** que ofrezca una marco para transformar el modelo formal anteriormente descrito en un conjunto de bibliotecas software con los que llevar a cabo la construcción de aplicaciones web. Dicha arquitectura se basa en los princios arquitectónicos *REST*, así como los desarrollos tecnlógicos en el área de la Web Semántica y la iniciativa de Datos Enlazados Abiertos.

- Diseño y construcción de aquellos **componentes software** que no se encuentren disponibles para llevar a cabo una implementación de la arquitectura software para la construcción de APIs semánticas que se ha especificado.

- Construcción de **aplicaciones web** reales que utilicén la implementación de la arquitectura de APIs semánticas desarrollada para demostrar como el uso de dichas APIs pueden solucionar algunos de los problemas que se pueden encontrar hoy en día en el desarrollo de aplicaciones web.

El resto de este documento se organiza de acuerdo con este plan de trabajo a través de los siguientes capítulos y secciones:

- El **capítulo 2** revisa el estado del arte en las diferentes areas sobre las que se basa el trabajo que se ha realizado para definir APIs semánticas. Estas areas incluyen las arquitecturas REST, computación distribuida, desarrollos recientes en Web Semántica y el enfoque alternativo conocido como *Open Linked Data* o Datos enlazados abiertos.

- El **capítulo 3** describe nuestra propuesta de solución a través de diferentes secciones que cubren un modelo formal para la descripción de la computación distribuida usando APIs semánticas, un modelo arquitectónico que puede ser usado para implementar el marco formal anteriormente descrito, algunos componentes software que han sido desarrollados para poder llevar a cabo la implementación de dicho marco formal y finalmente la descripción de ejemplos de aplicación en los que las herramientas construidas de acuerdo a nuestra concepción de las APIs semánticas son usadas para solucionar problemas concretos, como la interconexión de datos de diferentes servicios web o la construcción de visualizaciones de datos interactivas basadas en datos enlazados y expuestos a través de APIs semánticas.

- El **capítulo 4** extrae una serie de conclusiones finales sobre el trabajo realizado y expone las posibles líneas de trabajo que quedan abiertas para seguir desarrollando la investigación en este área.

Capítulo 2

Estado del Arte

Como se ha explicado en los capítulos previos, el objetivo del presente trabajo es presentar una arquitectura y herramientas que permitan la construcción de *APIs* semánticas para datos enlazados.

Para alcanzar este objetivo se ha hecho imprescindible un estudio pormenorizado del estado del arte en lo referente a la arquitectura de *APIs* de datos *REST*, modelos formales para la definición de sistemas distribuidos, avances en el desarrollo de la Web Semántica, los últimos avances alcanzados dentro de la iniciativa *Open Linked Data* así como de las herramientas y aplicaciones disponibles para implementar todos estos conceptos.

Algunas de estas áreas de investigación se encuentran directamente relacionadas, como por ejemplo, las iniciativas *Open Linked Data* y Web Semántica, mientras que otras se encuentran relacionadas de forma indirecta, como los trabajos en arquitecturas de servicios web *REST* que han influido de una forma directa en el desarrollo de muchas propuestas *Open Linked Data*, pero en algunos casos, como en la relación entre la teoría de la descripción formal de sistemas distribuidos, su relación con las otras áreas de investigación es mucho más difusa. Esto supone que ideas importantes para el presente trabajo aparecen en diferentes autores de manera complementaria pero usando muchas veces diferentes terminologías. En este capítulo intentaremos establecer estas relaciones entre las diferentes bases conceptuales sobre las que se ha construido esta tesis, así como definir una terminología común que se utilizará en el

resto de este documento.

En la primera sección de este capítulo, revisaremos los últimos avances en arquitecturas de servicios web *REST*, y los conceptos básicos sobre Web Semántica que se encuentran en el germen del desarrollo de esta tesis. A continuación, se examinan los trabajos sobre sistemas distribuidos que se han utilizado para realizar una descripción formal de las arquitecturas *REST* y por último, los principales desarrollos de la comunidad *Open Linked Data* en los que se trata de establecer puentes entre las tecnologías de la Web Semántica y las arquitecturas de servicios *REST*.

2.1. Arquitecturas de servicios web *REST*

En los últimos diez años, la visión de la Web como una plataforma sobre la que desplegar aplicaciones distribuidas altamente desacopladas y extensibles, compuestas por una multitud de servicios web, construidos sobre la pila de protocolos web (*HTTP/TCP/IP*) ha suscitado el interés tanto de la industria como de la comunidad de investigadores y que han sido comúnmente englobados bajo el epígrafe de *SOA* (*Service Oriented Architectures* o *Arquitecturas Orientadas a Servicios*) [100].

Un servicio web de datos en una arquitectura *SOA* se puede definir como unidades funcionales, atómicas, desacopladas y que pueden ser invocadas a través del protocolo *HTTP*. En la concepción *SOA* de las arquitecturas de servicios web, los servicios son interoperables gracias al uso de meta-datos, información adicional sobre el servicio que describe la funcionalidad ofrecida por el servicio así como los mensajes y el protocolo para acceder a ella.

Cuando la funcionalidad que se pretende alcanzar va más allá del simple acceso a un servicio web para recuperar datos, involucrando un gran número de servicios interaccionando entre ellos y con los clientes web, aparecen problemas complejos que debe ser resueltos: como la composición de un conjunto de servicios web para conseguir una determinada funcionalidad, problema conocido como *orquestación de servicios web*, o de cómo automatizar la interacción de un conjunto de servicios web sin un punto

central de control, problema conocido *como coreografía de servicios web.* [31]
Otros problemas asociados incluyen el catalogado de servicios web, el *descubrimiento de capacidades de servicios web* [66] así como los problemas de seguridad y autenticación en el acceso a dichos servicios.
El primer intento de solución para el conjunto de problemas que presentan las *arquitecturas orientadas a servicios* entre la industria vino dado por un grupo de estándares conocidos de forma genérica como arquitecturas *SOA WS-** [129]. Los componentes fundamentales de las arquitecturas *SOA WS-** consisten en un grupo de estándares entre los que se pueden destacar:

- ***SOAP* (*Simple Object Access Protocol*) [17]:** Un protocolo y formato de serialización de datos basado en *XML* y que puede ser utilizado sobre una capa de transporte como HTTP o SMTP para el intercambio de datos estructurados.
- ***WSDL* (*Web Services Description Language*) [26]:** Un lenguaje para la descripción de la funcionalidad ofrecida por un servicio web, incluyendo la localización del servicio, la forma de acceso y la descripción, usando *SOAP*, de los parámetros para su invocación así como el tipo de datos devuelto.
- ***UDDI* (*Universal Description Discovery and Integration*) [28]:** Una propuesta de catálogo de servicios web, construido sobre *SOAP* y *WSDL*, que puede ser usado por clientes web para encontrar y e invocar servicios web con el fin de llevar a cabo una determinada funcionalidad.

Sobre estos bloques básicos, la comunidad *WS-** ha propuesto especificaciones más complejas, que intentan solucionar algunos de los problemas anteriormente mencionados, como el de la coreografía de servicios web a través de la propuesta de estándar *WS-CDL* (*Web Services Choreography Description Language*) [69] o problemas tales como la descripción formal y automatizable de la interacción entre un determinado número de clientes y servicios web, expresada como un flujo de trabajo, para llevar a cabo algún tipo de funcionalidad compleja, con estándares como *WS-BPEL* (*Business Process Execution Language*) [96]. El objetivo último que se perseguía con la

especificación de este conjunto de estándares era la descripción de una arquitectura de distribuidos que se pudiese usar para la construcción de aplicaciones empresariales complejas, siendo el caso paradigmático la construcción de sistemas *ERPs* (*Enterprise Resource Planner*) [68], [79], conocida usualmente como *Enterprise Service Bus* (ESB) [114], por analogía con el *bus* de datos en las arquitecturas hardware, con un enfoque más ligero, extensible y fácil de mantener que alternativas anteriores basadas en otras propuestas tecnológicas como CORBA, gracias a las características de ocultación de la información, alta cohesión y bajo acoplamiento que teóricamente ofrecen los servicios web.

A medida que las implementaciones prácticas de arquitecturas *SOA* siguiendo los estándares *WS-** empezaron a hacerse realidad a partir del año 2005. Los resultados obtenidos dieron pie a posiciones críticas sobre la viabilidad de las arquitecturas *SOA* tal y como son concebidas por este cuerpo de estándares. Los principales inconvenientes señalados se pueden desglosar así:

- **Complejidad innecesaria:** las especificaciones *WS-** introducen un conjunto nuevo de formatos, protocolos, meta-servicios, etc. que añaden una capa de complejidad elevada encima de la relativamente sencilla capa de aplicación *HTTP*. Esta capa de complejidad hace que los sistemas *SOA WS-** sean costosos de desarrollar y mantener, necesitando casi de forma obligatoria herramientas de desarrollo automáticas capaces de generar todo el código intermedio requerido para realizar invocaciones a servicios o exponer incluso la más sencilla de las funcionalidades como un servicio web de acuerdo con los estándares. Esto evita que algunas de las promesas que las arquitecturas *SOA* prometían, como la facilidad de reemplazar implementaciones de servicios, sean difíciles de obtener en la práctica.

- **Problemas de eficiencia**: la complejidad introducida por los protocolos *SOA WS-** influye en las decisiones concretas de implementación de los servicios web siguiendo dichos estándares. Por ejemplo, *SOAP* describe el formato del

mensaje que debe ser intercambiado entre cliente y servicio como un documento *XML* complejo con un gran impacto en el tamaño final en *bytes* de los datos intercambiados, que se ven considerablemente incrementados. Esto supone que los servicios web *WS-** pueden ser menos eficientes desde el punto de vista computacional respecto a otras alternativas de implementación más ad-hoc, como los servicios sencillos *RPC* (*Remote Procedure Invocation*) [4], a pesar de ser estos últimos menos genéricos y extensibles.

- **Alto acoplamiento entre clientes y servicios:** a pesar de que los servicios *WS-** en sí no dependen unos de otros, los clientes intentando acceder a un servicio en particular dependen completamente del protocolo de acceso a ese servicio, lo que conlleva que cualquier cambio en los parámetros o tipos de entrada del servicio implican un cambio obligatorio del código del cliente.

Sin embargo, la principal crítica recibida por las arquitecturas *SOA WS-** vino dada por lo que se percibió como una falta de congruencia entre la capa adicional que es añadida por los estándares *WS-** sobre la capa web *HTTP* y los principios de diseño de dicha capa web.
Este conjunto de principios arquitecturales se encuentran ya definidos en documentos como los *Axioms of Web Architecture* de Tim Berners-Lee [10], pero fueron sistematizados en la tesis de Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures* [40]. En dicha tesis la arquitectura de la Web, basada en el protocolo *HTTP*, aparece descrita en el capítulo usando el acrónimo *REST* (*Representational State Transfer*). Las características de las arquitecturas *REST* tal y como aparecen descritas en la tesis de Fielding se enumeran a continuación:

- **Orientada a recursos:** el bloque básico de la arquitectura *REST* es el recurso, entendido como una relación conceptual entre identificadores y un conjunto de datos u otros identificadores asociados a ese identificador. La semántica d esta relación conceptual debe permanecer constante a lo largo del tiempo, aunque los datos asociados puedan variar.

- **Ausencia de estado mutable:** en la arquitectura *REST*, el servicio no almacena información sobre el estado del cliente entre peticiones. Esto supone una importante ventaja ya que permite que los sistemas web sean altamente escalables implementando componentes como cachés o aumentando el número de servidores ofreciendo un recurso particular sin necesidad de coordinar entre ellos estado mutable.

- **Uso de identificadores estándar:** basados en el estándar *URL*, que se puede utilizar para designar de forma única cualquier recurso expuesto a través de la Web.

- **Múltiples representaciones y negociación de contenido**: cuando un cliente web intenta acceder a un recurso usando el *URL* que lo identifica, cliente y servidor deben decidir qué representación particular del estado actual de ese recurso va a ser obtenida por el cliente. Esta representación es conocida en la arquitectura *REST* como *media type* y puede contener los bytes asociados a los datos del recurso, así como meta-datos sobre el recurso e incluso meta-datos sobre los meta-datos.

- **Interfaz uniforme *HTTP*:** en la arquitectura *REST* la interfaz para acceder a los recursos expuestos es uniforme y con una semántica estándar bien definida. Esta interfaz se compone de los métodos disponibles en el protocolo *HTTP*: *GET*, *POST*, *PUT*, *PATCH, DELETE*, *HEAD*, *OPTIONS* y *TRACE* a los que se les asigna la semántica de operaciones para obtener un recurso, crearlo, actualizarlo, destruirlo y obtener información asociada con él respectivamente.

El cuadro 2.1 muestra las principales diferencias entre las propuestas arquitecturales *REST* y *WS-**.

Cómo se puede observar, las principales diferencias vienen dadas por diferencias en principios conceptuales básicos entre ambos enfoques. Por un lado, la concepción del servicio web como una invocación de una operación remota arbitraria, en el caso

WS-*	***REST***
Cuerpo cerrado de estándares.	Estilo de arquitectura.
Orientada a la invocación remota de procedimientos.	Orientada a recursos.
Interfaz variable.	Interfaz uniforme.
Uso de HTTP opcional.	HTTP es el único protocolo.
Descubrimiento centralizado.	Descubrimiento a través de enlaces.
Uso de identificadores dependientes del contexto.	Uso de identificadores globales.
Puede almacenar estado.	No almacena estado.
Permite la descripción de recursos.	Los recursos son auto-descriptivos.

Cuadro 2.1: Comparativa de estilos arquitecturales *REST* y *WS-**.

de los servicios *WS-**, frente al servicio web como una petición *HTTP* estándar para recuperar o modificar un recurso usando de una forma estricta la semántica del protocolo *HTTP* en el caso de los servicios *REST*. En comparación, los servicios *WS-** usan una semántica variable para las invocaciones *HTTP*, que hace obligatoria la introducción de meta-datos y meta-servicios adicionales como *UDDI*, aumentando de esta forma la complejidad de la implementación de soluciones *WS-**.

El resultado último de la aparición de la propuesta arquitectural *REST* en el panorama de las arquitecturas orientadas a servicios fue que diferentes sistemas de computación empresarial basadas en soluciones *WS-** empezaron a ser reemplazados por soluciones construidas usando servicios web *REST*. Sin embargo, algunos patrones de integración en aplicaciones empresariales no son fácilmente traducibles usando los principios arquitectónicos *REST*, como por ejemplo, la distribución asíncrona y confiable de mensajes o los escenarios composición de servicios y descripción de flujos de procesos de negocio [102]. Para poder implementar algunos de estos casos de uso usando servicios web *REST*, es necesario recurrir a estándares adicionales como *WADL* [111] para la descripción de los recursos. En estos casos las soluciones *WS-** siguen siendo usadas profusamente.

Al mismo tiempo, la propia pila de protocolos y estándares *WS-** ha venido incorporando ideas y conceptos *REST* con el fin de ofrecer soluciones más simples y escalables, por ejemplo la versión 2.0 de *WSDL* introduce soporte para describir servicios *WS-** implementados sobre servicios web *REST* [124].

Fuera del ámbito de la computación empresarial, a mediados de la década de los 2000 y coincidiendo con el auge de la llamada Web Social o Web 2.0 [93], aparece la necesidad entre los desarrolladores de aplicaciones web de ofrecer el acceso a los datos almacenados en esas aplicaciones a los usuarios para que pudieran ser utilizadas por aplicaciones cliente móviles, aplicaciones software de terceros o dispositivos hardware, como cámaras fotográficas. También empezó a ser importante la integración de datos de diferentes aplicaciones para ofrecer aplicaciones, conocidas como mashups, que ofrecían una nueva funcionalidad a partir de la composición de diferentes fuentes. Esto suponía ofrecer interfaces de servicios web, englobadas en una *API* de datos, que pudiesen ser consumidas por los clientes web y otros servicios. Tras algunos intentos iniciales de utilizar *APIs* basadas en algunos estándares *WS-**, como *SOAP*, la arquitectura *REST* se ha impuesto como el marco conceptual elegido por la mayoría de aplicaciones Web 2.0 para implementar sus *APIs* de datos, con mayor o menor grado de fidelidad a los principios prescritos por dicha arquitectura.
La implementación en populares *frameworks* desarrollo web como *Ruby on Rails* o *Django* de estos principios también ha contribuido decisivamente a su difusión. La características comunes de este tipo de *APIs* de datos en aplicaciones web se enumeran a continuación:

- **Soporte parcial para la interfaz uniforme:** donde sólo los métodos *GET* y *POST*, los únicos disponibles en los navegadores web, son incluidos en la interfaz de acceso, desplazándose las operaciones asociadas al resto de operaciones de la interfaz *HTTP* a *URLs* especiales de la aplicación.

- **Ausencia de negociación de contenido:** Muchas *APIs* de datos no soportan más que una sola representación de los recursos expuestos. Cuando más de una representación está disponible, la solución más usada consiste en utilizar diferentes *URLs* para diferentes representaciones, rompiéndose de esta manera la identidad única del recurso.

- **Uso de JSON como un formato universal de intercambio:** La importancia de *JavaScript* como el lenguaje nativo del navegador web, así como la

sencillez del formato, su eficiencia y la presencia de buenas bibliotecas para serializar y deserializar datos en otros lenguajes de programación han hecho que *JSON* (*JavaScript Simple Object Notation*), se haya convertido en el formato por defecto en la mayoría de *APIs* de datos, desplazando a otras opciones más populares en el ámbito de los servicios web *WS-** como *XML*.

- **Uso limitado de las capacidades más avanzadas del protocolo *HTTP*:** Características propias del protocolo *HTTP* como el uso de las cabeceras de *caché* son ignoradas y otras veces, es frecuente encontrar usos erróneos de las mismas, como en el caso de los códigos de retorno de las peticiones *HTTP* o el uso de la cabecera de localización del recurso creado, muchas veces ignorados en favor del retorno del estado en el cuerpo de los datos recuperados.

- **Conflictos en la identidad de los recursos entre *URIs* e identificadores:** De acuerdo con los principios *REST*, un recurso web debería venir asociado a un o más de un *URI* estable. Sin embargo, la mayoría de *APIs* identifican los recursos no por un *URI* sino por un identificador que se introduce en la representación del recurso. Esto es debido a factores anteriormente comentados, como el uso de diferentes *URLs* para designar diferentes representaciones de un mismo recurso.

- **Uso de mecanismos ad-hoc para establecer enlaces entre recursos:** El uso de identificadores arbitrarios y relativos sólo a una determinada *API* de datos, hace imposible para el cliente usar un mecanismo estándar como una *URI* para identificar y obtener los recursos relacionados con un determinado recurso. Para obtener estos recursos, el cliente debe computar la *URL* desde la que el recurso relacionado estará disponible a partir de la información de estado del recurso actual de una forma específica a la *API* en la que están englobados.

- **Difícil inter-operabilidad entre *APIs*:** Las dificultades para establecer enlaces entre recursos de una misma *API* anteriormente comentadas, se hacen patentes también de forma todavía más evidente cuando se intentan enlazar

recursos entre *APIs* de diferentes proveedores, si ambos proveedores no están haciendo uso de *URIs* canónicos para identificar los recursos o si estos *URIs* no pueden ser insertados de una forma simple en la representación de los recursos.

Una consecuencia directa de las características anteriormente expuestas, es que la mayoría de *APIs* existentes hoy en día se encuentran aisladas unas de otras. Los datos expuestos por los diferentes servicios y aplicaciones web a través de *APIs* incompatibles y que no usan mecanismos estándar para identificar y describir los datos que ofrecen, y con ellos los usuarios de esos datos, se encuentran encerrados en silos de información o *walled gardens* [47] constituyendo islas de datos desde las que no se pueden establecer conexiones con otros servicios de datos. Esta situación es similar a la de la plétora de redes locales aisladas en la época anterior al protocolo *IP* de red y *HTTP* de aplicación que almacenaban repositorios de documentos aislados unos de otros, en formatos incompatibles y sin ofrecer la posibilidad de establecer enlaces entre documentos situados en dos subredes diferentes.
La búsqueda de mecanismos que permitan el enlazado y la conexión de *APIs* ofrecidas por diferentes servicios, de tal forma que se pueda usar un mecanismo común para la descripción de los datos y la identificación de los recursos propios y remotos, facilitándose de esta manera el descubrimiento y la automatización del consumo de servicios de datos ha sido el objeto de intensa investigación durante los últimos años. Algunas de las principales líneas de investigación que se encuentran actualmente abiertas y que son relevantes para la presente tesis, se analizan a continuación.

2.1.1. Descripción de *APIs REST*

Uno de los principales objetivos de diseño *REST* es el de que los servicios web construidos deben ser auto-descriptivos [10]. Esto quiere decir que un cliente que desee consumir un recurso a través de un servicio web *REST*, debería poder obtener toda la información necesaria para acceder al recurso o manipularlo de una forma tal que sea susceptible de ser automatizada. Esto supone poder obtener diferentes representaciones del mismo a través del propio servicio usando meta-datos asociados con el

recurso.

Debido a la simplicidad característica de los servicios web *REST* y al uso de convenciones, como la semántica asociada por el protocolo *HTTP* a las operaciones del protocolo, el único mecanismo estándar disponible en la arquitectura *HTTP* para describir las capacidades de un determinado servicio e intercambiar otros meta-datos sobe el servicio es el conjunto de cabeceras *HTTP* intercambiadas por clientes y servicios, junto a los códigos de respuesta establecidos en el protocolo.

Estas cabeceras constituyen un mecanismo que puede ser usado de forma eficaz para descubrir información sobre el servicio, como por ejemplo, en el mecanismo de negociación de contenido usando la cabecera de *media type* asociada a un recurso. Además las cabeceras *HTTP* son también un mecanismo extensible, ya que nuevas cabeceras no recogidas en el estándar *HTTP* pueden ser insertadas en la petición de un cliente o en la respuesta proveniente de un servicio.

Sin embargo, el uso de cabeceras por sí sólo no resuelve completamente el problema de la construcción de servicios auto-descriptivos, ya que constituye un mecanismo para el intercambio de meta-datos entre cliente y servicio pero no menciona como han de ser estos meta-datos, ni su formato, más allá de los especificado por las diferentes versiones del protocolo *HTTP*. El problema se puede extrapolar a la descripción de una *API* completa, así como la relación entre los recursos de esa *API*.

En la actualidad este es un problema abierto. Actualmente la mayoría de descripciones de *APIs REST* se realizan en texto plano, como documentación asociada al servicio que los desarrolladores del software que accederá a dichos servicios deben interpretar y transformar en la lógica de sus programas. Esta situación contrasta con los complejos mecanismos de descripción de servicios web *WS-** que hacen muy recomendable el uso de herramientas automáticas que generen el código necesario para realizar las invocaciones pertinentes a los servicios que se desea consumir.

La forma de encontrar formas eficientes de describir servicios *REST*, que puedan ser consumidas de forma automática por agentes software, sin incurrir en la complejidad asociada a los mecanismos de descripción de servicios *WS-** es todavía un campo activo de investigación tanto a nivel académico como industrial.

Un primer tipo de solución a la descripción de servicios web *REST* ha consistido en la proposición de diferentes lenguajes para la descripción de recursos, como *WADL* (*Web Application Description Language*) [111], que propone una serialización *XML* de la descripción de los recursos de una *API*, así como de las operaciones, parámetros de entrada y códigos de estado que serán devueltos por los diferentes servicios de la *API*. Esta descripción podía ser expuesta a su vez como un recurso más de la propia *API*. Los clientes podrían consumir este *meta-recurso* para descubrir los tipos de recursos disponibles y los requisitos para poder acceder a ellos.

WADL supone una simplificación de mecanismos similares propuestos en el mundo de los servicios web *WS-** com *WSDL*, y no ha conseguido el apoyo mayoritario de los desarrolladores de *APIs* web *REST*. El propio *WSDL* en su versión 2.0, permite la descripción de servicios web *REST*, como ya hemos mencionados, siendo esta versión ligera de *WSDL* mucho más usada en la actualidad. Además de *WADL*, otra gran cantidad de lenguajes de descripción de servicios *REST*, más o menos ad-hoc y centrados en dominios concretos de aplicación han sido propuestos en la literatura sobre la materia. Entre ellos podemos citar *RSWS* (*Really Simple Web Service Descriptions*) y *WDL* (*Web Description Language*) [77].

Frente a los esfuerzos por especificar un lenguaje, más simple que *WSDL*, para la descripción de servicios *REST* que pueda ser usado de forma genérica por los constructores de *APIs* basadas en servicios *REST*, una parte de la comunidad académica y de forma casi unánime la industria, ha argumentado que no existe la necesidad de un lenguaje de descripción en absoluto.

La justificación a esta postura ha venido dada por la importante cantidad de información acerca de la semántica de los servicios que se puede obtener a partir de las convenciones propias de las arquitecturas *REST*, como el uso de la interfaz uniforme *HTTP* [77].

2.1.2. Descubrimiento de servicios web *REST*

Un aspecto de gran importancia a la hora de implementar arquitecturas de servicios es la capacidad de descubrir automáticamente los servicios necesarios para llevar a cabo una determinada funcionalidad. La principal opción para el descubrimiento de servicios en el ámbito de las arquitecturas de servicios web *WS-** la constituye *UDDI* un registro centralizado de servicios. *UDDI* describe diferentes componentes para diferentes casos de uso en el descubrimiento de servicios. Puede ser usado como un servicio de *páginas blancas* con información sobre la organización que es proveedora de un tipo de servicio, haciendo posible encontrar servicios basándose en el nombre u otra información relativa al proveedor de dicho servicio. Otro caso de uso de *UDDI* es el de un registro de tipo *páginas amarillas*, donde dado un tipo de servicio dentro de una jerarquía de servicios permite encontrar proveedores para dicha categoría de servicios. Por último, *UDDI* puede ser usado como un servicio de *páginas verdes* almacenando los detalles técnicos y de contacto sobre un determinado servicio. De esta manera los clientes de un servicio pueden encontrar la información acerca de como dicho servicio debe ser consumido.

UDDI no ha encontrado una gran aceptación entre la industria, especialmente por su naturaleza centralizada así como por su complejidad. La comunidad *REST* ha intentando buscar en los últimos años alternativas a sistemas como *UDDI* que puedan usarse para descubrir servicios *REST* sin incurrir en el complejo diseño de esta tecnología. Algunas nuevas propuestas dentro de la comunidad de servicios web *WS-**, como *WSIL* (*Web Services Inspection Language* [6] han intentado dar respuesta a las limitaciones de *UDDI*, como por ejemplo, la arquitectura centralizada del servicio, pero sin conseguir a pesar de ello una importante aceptación.
Una de las soluciones exploradas ha sido el uso de un mecanismo fundamental en el diseño de la web como el sistema *DNS* (*Domain Name Service*) para aplicarlo al descubrimiento de servicios web *REST* [67] a través del uso de *DNS-SD* una extensión del estándar *DNS* para el registro de servicios de un determinado tipo en un deter-

minado dominio que pueden ser descubiertos por los clientes *DNS*. La descripción de los meta-datos del servicio se puede incluir en un registro DNS de tipo TXT o SRV y conducir a través de una petición *HTTP* al servicio o a una descripción del servicio usando alguno de los lenguajes de descripción de servicios mencionados anteriormente. Este sistema supone algunas importantes ventajas sobre tecnologías como *UDDI*: reusa una tecnología existente y aceptada por la industria, no es un sistema centralizado con lo que consigue una gran escalabilidad y es un sistema dinámico donde nuevos servicios pueden alterar su información sencillamente a medida que la computación evoluciona.

A pesar de las ventajas que una tecnología como *DNS* parece aportar como posible solución para el descubrimiento de servicios web, otra parte de la comunidad *REST* ha intentado construir mecanismos para el descubrimiento de servicios sin hacer uso de elementos externos a la pila de tecnologías que conforman la web. Un ejemplo se encuentra en [128], donde se describe un algoritmo para realizar el descubrimiento de servicios web *REST* con los que llevar a cabo una determinada tarea, partiendo de un enlace web inicial y utilizando para ello, la cabecera *OPTIONS* del protocolo *HTTP* como la forma con la que obtener los meta-datos asociados al servicio y el *REST* o de cabeceras del protocolo *HTTP* para negociar una posible representación o las capacidades asociadas al servicio. Un modelo alternativo para el descubrimiento de servicios *REST* descentralizado y basado puramente en tecnologías web que cumple con los estándares *REST*, viene descrito en [126]. En dicho modelo se distinguen tres capas principales: el conjunto de estándares web usados en el mecanismo de descubrimiento como *URIs* y el protocolo *HTTP*, una capa de referencia en la que se describe cómo a partir de un determinado *URI* se puede obtener la descripción asociada a ese servicio y por último la capa de descripción, donde se investiga que vocabulario es el adecuado para especificar la descripción del recurso.

Algunas de las propuestas de estándar que se han realizado para llevar a cabo la funcionalidad propia de la capa de referencia incluyen *XLink* (*XML Linking Language*) y *LRDD* (*Links based Resource Descriptor Discovery*) [48]. *LRDD* describe mecanismos alternativos para asociar descriptores con los recursos descritos: uso de

una etiqueta *LINK* en la representación *HTML* y *Atom* de un recurso que enlazaría al descriptor de ese recurso, el uso de una cabecera *HTTP* denominada *Link* que devolvería el enlace a la descripción del recurso cuando la *URI* del recurso es desreferenciada usando una petición *HTTP GET* o *HEAD* y, por último, el uso de un fichero *.well-known* disponible en una localización estándar para un dominio específico y que incluiría enlaces para las descripciones de los recursos disponibles en ese recurso.

2.1.3. Flujos de trabajos y *mashups* de servicios *REST*

En apartados anteriores hemos discutido los avances realizados en la descripción de meta-información acerca de recursos web, así como de mecanismos que permitan el descubrimiento de dichos servicios.

La combinación de ambas líneas de investigación abre la puerta a la automatización de interacciones complejas entre agentes y servicios web, en lo que se conoce como ejecución de flujos de trabajo. Para llevar a cabo uno de estos flujos, los agentes deben descubrir, seleccionar y consumir servicios en un determinado orden de forma autónoma siguiendo la guía ofrecida por una especificación de alto nivel donde se expone la lógica de negocio que se quiere obtener. El proceso de transformar de una forma automatizada esta descripción de la funcionalidad de negocio que se desea conseguir en un conjunto de interacciones entre servicios y agentes, se conoce como *orquestación de servicios web* [31].

Otro problema relacionado, aunque más sencillo en su planteamiento es el la descripción de interacciones entre servicios web a través de primitivas básicas, como la composición de servicios, que será luego ejecutada de una forma automática por parte de agentes software. Esta versión del problema se conoce como *coreografía de servicios web* [31].

En el mundo de los servicios web *WS-**, dos estándares cubren ambas áreas. *WS-BPEL* (*Business Processes Execution Language*) [96] permite describir escenarios de orquestación de servicios web para que puedan ser luego ejecutados automáticamente por un motor de ejecución de orquestación de servicios web. Por su parte *WS-CDL*

(*Choreography Description Language*) [69] permite describir las operaciones básicas que conforman la coreografía de servicios web para que puedan ser ejecutadas de una forma autónoma por un agente software.

Un problema que ha sido abordado por diferentes autores es el de la integración de servicios web *REST* con los estándares de coreografía y orquestación diseñados para los servicios web *WS-** [134].

Un primer paso para lograr esta integración ha venido dada por el soporte para servicios *REST* en las últimas versiones de *WSDL* [124], que es un componente esencial para las especificaciones tanto de coreografía *WS-CDL*, como de orquestación *WS-BPEL*.

Desde un punto de vista complementario también se ha intentado modelar las primitivas básicas de orquestación de un lenguaje como *BPEL* a las primitivas básicas de la interfaz uniforme *HTTP* que caracteriza a los servicios web *REST* [101].

Por último también se han diseñado diferentes lenguajes de orquestación y coreografía específicamente pensados para ser utilizados en un entorno de servicios web *REST*, entre los que podíamos destacar: *SWAP* (*Simple Workflow Access Protocol*), *Wf-XML* (*Workflow XML*), *AWSP* (*Asynchronous Web Services Protocol*) y *ASAP* (*Asynchronous Service Access Protocol*)[134].

El interés por integrar servicios web *REST* en las soluciones de automatización de flujos de trabajo *WS-**, ya sea desde la perspectiva de la orquestación o la coreografía, ha venido dada en la mayoría de los casos por miembros de la industria que han invertido tecnológicamente en soluciones *WS-** y desean integrar el, cada vez más rico, ecosistema de servicios web *REST* dentro de su infraestructura. Sin embargo, dentro de la comunidad *REST* y más concretamente dentro de la comunidad de desarrollo web, existen problemas relacionados con la composición de servicios web que han sido objeto de investigación tanto por la industria como en entornos académicos.

El caso más paradigmático es el de la construcción de *mashups* web [9]. Se trata de un caso particular de orquestación de servicios, donde diferentes servicios *REST* deben ser compuestos para obtener una funcionalidad web que será expuesta a un usuario. En la mayoría de los casos el consumo y la integración de los recursos ex-

puestos por los servicios *REST* será automática, pero el diseño de la forma en que estos recursos y servicios se agregan e interaccionan unos con otros debe ser llevada a cabo de una forma manual e interactiva por parte del usuario final a través de algún tipo de interfaz web. La mayoría de soluciones descritas en la literatura consisten en herramientas creadas ad-hoc para ofrecer una determinada funcionalidad con mayor o menor grado de interactividad en la composición del resultado de la agregación por parte del usuario final. Estas soluciones trabajan con un conjunto pre-determinado de servicios web *REST* que no pueden ser modificados.

Algunas otras herramientas han intentado aportar un mayor grado de genericidad en la construcción de *mashups* explotando las características de los servicios *REST*, como la homogeneidad en el acceso a la información. El trabajo más destacado es el de *Yahoo* con la construcción de *Yahoo Pipes* [105] una herramienta para la construcción de *mashups* web de una forma sencilla por parte del usuario final. La característica más interesante de Yahoo Pipes reside en el uso de un meta servicio de datos capaz de adaptar cualquier servicio web que se ajuste a la interfaz uniforme *HTTP* y que use *JSON* como el formato de datos para la representación de los recursos expuestos. La interpretación de la semántica de los datos *JSON* obtenidos desde el servicio debe ser aportada por el usuario final construyendo la *mashup* usando alguno de los elementos predefinidos que el sistema provee.

Por último, otra aportación interesante en el desarrollo de *mashups* web es el de la construcción de lenguajes de programación o *scripting* [112] específicamente diseñados para construir *mashups* web a través de la integración de diferentes servicios *REST*. En dichos lenguajes, las operaciones primitivas que se pueden aplicar a un recurso *REST* a través de la interfaz *HTTP*, así como la petición de determinadas representaciones y las operaciones necesarias para agregar estas representaciones son transformadas en primitivas computacionales de alto nivel en el lenguaje de programación, de forma tal, que la composición de servicios puede ser descrita de una forma sencilla a través del lenguaje de programación. Este programa puede ser a continuación ejecutado de forma automática por un agente software que cuente con un intérprete para dicho lenguaje de programación.

2.1.4. *HATEOAS*, *Hypermedia* como el Motor del Estado de la Aplicación

Un concepto clave que marca la diferencia entre el conjunto de especificaciones técnicas *WS-** y soluciones para la construcción de servicios web basados en la invocación remota de procedimientos (*RPC*) frente a los principios arquitecturales *REST* es el del uso de los tipos de datos y los enlaces para mantener el estado de la aplicación.

Este concepto aparece ya en la tesis original sobre *REST* de Roy Fielding [40], pero ha venido a consolidarse más tarde bajo el acrónimo *HATEOAS* (*hypermedia* como el Motor del Estado de la Aplicación). La consecuencia básica de este principio para la construcción de *APIs* de servicios web *REST* es que la mayoría de elementos del servicio, por ejemplo el protocolo de comunicación y su semántica, debe ser fijos y estándar, en este caso el protocolo *HTTP* y la semántica predefinida para las operaciones del protocolo. El único elemento sobre el que puede actuar el diseñador de la *API* es sobre el tipo de datos asignado a las diferentes representaciones de los recursos expuestos.

Desde este principio de arquitectura, el diseño de *APIs* es equivalente al diseño de tipos de datos, y conociendo como manipular un determinado tipo de datos, cualquier agente web debería ser capaz de utilizar una *API REST* partiendo de una *URI* inicial, dado que el resto de elementos de la interfaz van a ser homogéneos y constantes respecto a cualquier otra *API REST*. Esto último se logra a través de una segunda consecuencia básica derivada del principio de diseño *HATEOAS*, cualquier cambio en el estado de la aplicación se produce en el lado del cliente y es el resultado de seguir un enlace en la representación del recurso obtenida. De este modo, el servicio ofrece las opciones necesarias al cliente para que elija el siguiente estado de la aplicación de una forma estándar y el cliente debe limitarse a extraer estos enlaces, que también pueden incluir otros elementos de control, como en un formulario *HTML* y seleccionar

el siguiente estado.

Una línea de investigación fructífera dentro de la comunidad académica *REST* ha consistido en la aplicación de las restricciones que el principio arquitectónico *HATEOAS* impone a diferentes problemas en el desarrollo de servicios web. En la sección anterior hemos visto como de forma implícita *HATEOAS* se ha intentado aplicar a la orquestación y coreografía de servicios web *REST* a través del desarrollo de lenguajes específicos que tenían en cuenta la presencia enlaces en las representaciones de los recursos para descubrir mecanismos para manipular dichos recursos o recursos asociados [2].

Otro ejemplo de la importancia de *HATEOAS* para la definición de servicios web *REST* es evidente en la sucesión de artículos sobre como aplicar el principio *HATEOAS* a *APIs* ya existentes para transformarlas en *APIs* consistentes con los principios arquitectónicos *REST* tal y como se muestra en [80].

2.2. Web Semántica

La Web Semántica ha recorrido un largo trayecto desde su concepción hace más de diez años por parte de Tim Berners Lee como una versión de la web centrada en datos fácilmente procesables de forma automatizada por agentes software, en vez de una web únicamente de documentos para ser interpretados por usuarios humanos.

La visión última de la Web Semántica, en la que la inferencia lógica jugaría un papel destacado en la forma en que agentes software llevarían a cabo tareas determinadas usando los datos disponibles en la web, no ha sido alcanzada en su totalidad. Sin embargo, en estos años de investigación, numerosos resultados, estándares y tecnologías relevantes para la investigación expuesta en esta tesis han sido desarrollados en el seno de la comunidad de investigación sobre Web Semántica. En los siguientes apartados, se hará un repaso de los resultados más importantes.

2.2.1. Estándares Semánticos

La iniciativa Web Semántica ha sido liderada y llevada a cabo en su mayor parte al amparo del consorcio Web (*W3C*). Esto supone que la investigación más importante sobre Web Semántica ha cristalizado en el desarrollo de estándares web *W3C*. Algunos de estos estándares han tenido una gran influencia, no sólo dentro de la comunidad Web Semántica sino que ha tenido impacto en otras áreas de investigación relacionadas y en la industria en algunos casos de uso en los que era necesario trabajar con datos estructurados.

Un primer estándar es *RDF* (*Resource Description Framework* o *Framework* para la Descripción de Recursos). *RDF* describe simultáneamente un modelo de datos de una gran genericidad, consistente en un multi grafo dirigido, una serialización de dicho modelo de datos como una serie de expresiones sujeto-predicado-objeto y una semántica precisa para dicho modelo de datos [52].

RDF se encuentra en el nivel más básico de la pila de estándares propuestos por la comunidad Web Semántica, ya que es utilizado como el modelo de datos sobre el que se construyen el resto de estándares semánticos.

Un grafo *RDF* puede ser expresado usando diferentes sintaxis estandarizadas por *W3C*. La primera sintaxis propuesta estaba basada en *XML* (sintaxis *RDF/XML*) [7] y presenta una notable complejidad para los usuarios de dicha sintaxis, ya sea a la hora de generar la serialización de un grafo *RDF* o a la hora de construir un *parser* capaz de reconstruir un grafo *RDF* a partir de un documento *RDF/XML*. Un área de investigación destacable en estos años ha consistido en la búsqueda de mejores sintaxis para la serialización de grafos *RDF*, teniendo como resultado algunas propuestas de sintaxis simples para *RDF* como *N3* [11] o *Turtle* [8].

Otro estándar semántico de gran importancia es *SPARQL* (*SPARQL Protocol and RDF Query Language*) [104], un lenguaje que permite realizar consultas sobre un grafo *RDF* mediante expresiones consistentes en patrones de expresiones sujeto-predicado-objeto con variables. *SPARQL* es un lenguaje de consulta muy expresivo, que permite extraer partes de un grafo *RDF* o información particular de dicho grafo con una sin-

taxis similar a la de lenguajes de consulta para bases de datos relacionales como *SQL*. *SPARQL* ha sufrido una evolución importante desde su especificación inicial. Se han elaborado resultados acerca de la semántica formal y la complejidad computacional de las consultas *SPARQL* [103] y el estándar se ha extendido en su versión *SPARQL 1.1 UPDATE* [42] para dar soporte no sólo a la recuperación de información del grafo *SPARQL*, sino a la modificación del grafo mediante consultas específicas para la actualización, la inserción y el borrado de tripletes en un grafo *RDF*.

Otro conjunto clave de especificaciones semánticas lo constituyen los lenguajes para definir ontologías sobre grafos *RDF*. *OWL* (*Web Ontology Language*) [83] es el lenguaje estándar propuesto por *W3C* para la construcción de vocabularios y ontologías con los que describir recursos web. *OWL* extiende la semántica propia de *RDF* y de *RDFS* (*RDF schema*) [19] con primitivas ontológicas provenientes de la teoría formal de la Lógica Descriptiva [64].

En la primera versión del lenguaje, se especifican diferentes perfiles, que suponen un balance inverso entre expresividad, número de primitivas que se pueden usar en dicho perfil y complejidad algorítmica a la hora de llevar a cabo tareas de inferencia (clasificación, comprobación de consistencia, etc) sobre una ontología expresada usando ese perfil. En la segunda versión del lenguaje, *OWL 2* [92], los perfiles se han reorganizando, asignando las primitivas en base a casos de uso particulares (*QL*, *EL*, *RL*) [89] y no tanto un incremento simple de menor a mayor complejidad y expresividad. El lenguaje de consulta *SPARQL* también puede ser extendido [117], [44], para tener en cuenta el uso de un determinado perfil OWL, de tal manera que el conjunto de tripletes en el grafo *RDF* que van a ser recuperados por una consulta *SPARQL* se vea aumentado mediante el uso de la inferencia lógica disponible para el perfil de *OWL* seleccionado. La implementación de soluciones tecnológicas que permitan implementar las especificaciones tanto de *RDF*, como de *SPARQL* y *OWL* ha sido otra importante línea de investigación y desarrollo durante los últimos años de evolución de la iniciativa Web Semántica.

Se han propuesto diferentes soluciones tecnológicas para cada uno de los anteriores estándar: repositorios de tripletes *RDF*, motores de consulta *SPARQL* y razonadores

lógicos OWL. De entre ellas destacamos algunas interesantes arquitecturas para el almacenamiento de grafos *RDF* así como de motores de consulta *SPARQL* [50] [109].

2.2.2. Servicios Web Semánticos

La anotación semántica de los meta-datos asociados a un servicio web, de tal forma que sea posible que un agente software consuma de forma automatizada dicho servicio es uno de los problemas que han intentando solucionar las diferentes especificaciones para la construcción de servicios web. La solución que ofrece las tecnologías web semántica consiste en ontologías y modelos de servicio estándar que se pueden combinar con los estándares *WS-** o arquitecturas *REST* para solucionar el problema de la inter-operabilidad entre servicios.
Algunas de as principales propuestas de ontologías para la descripción de servicios web semánticos son las siguientes [76]:

- ***OWL-S* [82]**: Es propuesta de ontología construida usando el estándar semántico de *W3C OWL*, para la anotación de servicios web. *OWL-S* incluye vocabularios para la descripción del perfil del servicio (*service profile*), de tal forma que el servicio pueda ser descubierto por agentes software, un vocabulario para describir el modelo de proceso del servicio (*service process model*), los detalles operacionales concretos del servicio, incluyendo aspectos como el número de operaciones y los tipos de entrada y salida de los argumentos. Por último *OWL-S* incluye un vocabulario para la descripción del *grounding* del servicio, es decir, cómo los elementos del modelo de proceso se implementan como elementos del estándar de servicios web *WS-**, por ejemplo una descripción *WSDL*. La principal crítica recibida por *OWL-S* proviene de lo estático de su propuesta, ya que no recoge los elementos necesarios para describir variaciones temporales en las descripciones de los servicios ni un número arbitrario de operaciones no relacionadas.

- ***WSMO* [78]**: Es un intento de construir una especificación formal completa que

pueda ser usada en la descripción de servicios web semánticos. *WSMO* incluye un modelo formal y lenguaje de descripción (*WSML*) y una implementación de referencia para un entorno de ejecución de los servicios *WSMO* (*WSMX*). *WSMO* permite definir diferentes aspectos del entorno de uso de servicios web, incluyendo el dominio de aplicación de los servicios, los objetivos que los agentes software consumiendo los servicios necesitan completar y las descripciones de los servicios que pueden satisfacer esos objetivos. Una de las principales críticas recibidas por *WSMO* es que su especificación se llevó a cabo sin tener en cuenta los estándares semánticos como *OWL*, lo que supuso un inconveniente para que *WSMO* se convirtiese en un estándar web *W3C*. Una versión reducida del estándar *WSMO-Lite* [130] fue creada para solventar estos inconvenientes y ofrecer mejor compatibilidad con los estándares semánticos *W3C*.

- ***SAWSDL*** **[74]**: Es un intento por parte de *W3C* de especificar una ontología estándar para la anotación de servicios web definidos usando *WSDL* con información semántica. Extensiones al estándar describen elementos claves presentes en otros intentos previos, como el modelo de referencia para los servicios y vocabularios para el lifting y lowering de los servicios, es decir la relación existente entre los elementos descritos mediante la ontología de *SAWSDL* y los tipos básicos *XML* que se intercambiarán entre cliente y servicio cuando el servicio es consumido.

Así mismo, la comunidad Web Semántica también ha intentado incluir elementos semánticos en los meta-datos asociados a servicios web *REST*.
Algunas de las principales propuestas aparecidas en los últimos años son:

- ***hRESTS*** **[73]**: Es una propuesta para transformar la documentación *HTML* ad-hoc para humanos asociada a las *APIs REST* en meta-datos procesables automáticamente por agentes software usando microformatos. Los microformatos [70] son un intento de incluir meta-datos en documentos *XHTML* usando determinados atributos y etiquetas *HTML* siguiendo una serie de convenciones. Diferentes vocabularios de mxiroformatos (para información personal, tareas,

curriculum vitae, etc) definen cómo deben ser usados y qué valores deben tener. *hRESTS* consiste en una de dichas especificaciones que permite marcar los elementos básicos del modelo *REST* de servicios en la documentación, para que un agente software capaz de procesar la especificación *hRESTS*, pueda extraerlas y consumirlas.

- ***MicroWSMO*** **[75]**: Es una especificación que intenta anotar servicios web *REST* usando un modelo de servicio basado en *SAWSDL* [74] (modelo de servicio, *lifting* y *lowering*) usando *hRESTS* para anotar la documentación *HTML* asociada a los servicios. La gran ventaja de usar el vocabulario propuesto por *MicroWSMO* es que al ser una ontología compatible con *SAWSDL*, *MicroWSMO* puede usarse como un mecanismo para integrar servicios web *REST* con servicios *WS-**.

- ***SA-REST*** [115]: Una propuesta similar a *MicroWSMO* para anotar la documentación *HTML* asociada a un servicio web *REST*. La principal diferencia entre *SA-REST* y *MicroWSMO* reside en el hecho de que *SA-REST* utiliza *RDFa* [1], un estándar *W3C* que será revisado en la próxima sección para añadir meta-datos a un documento *HTML* en vez de usar un mecanismo no estándar como son los microformatos. En cuanto a la naturaleza de las anotaciones también son muy similares a las propuestas por *MicroWSMO*, consistentes en una descripción del modelo de servicio basada en componentes como tipos de entrada, salida, operaciones, acciones *HTTP*, errores y los mecanismos de *lifting* y *lowering*.

2.2.3. *ICV* Validaciones de Restricciones de Integridad

Un último desarrollo aparecido dentro de la comunidad Web Semántica es el de un mecanismo para interpretar una ontología construida usando estándares semánticos como *OWL* y *RDF Schema* como restricciones de integridad de datos que pueden ser validadas (*ICV*) [125], [91].

La base del mecanismo supone cambiar los axiomas fundamentales utilizados para

realizar inferencia lógica sobre una ontología: la Asunción de Mundo Abierto (*OWA*) y la ausencia de Asunción de Nombres Únicos (*UNA*) [118]. Ambos axiomas, heredados de la Lógica Descriptiva, permiten a un razonador *OWL* añadir nuevas inferencias cuando realiza la interpretación de un modelo dado por una ontología *OWL*.

La Asunción de Mundo Abierto implica que una proposición no puede ser inferida como falsa si no se cuenta con la información para probarlo de esta manera, por el contrario, el razonador puede añadir las proposiciones necesarias para que la interpretación sea verdadera. Por su parte la ausencia de Asunción de Nombres únicos significa que dos identificadores diferentes en dos proposiciones del modelo pueden interpretarse como referentes al mismo individuo para hacer la interpretación consistente. Estas asunciones son muy útiles en el contexto de un lenguaje de ontologías diseñado para ser usado en la web, un entorno altamente distribuido donde la información puede ser agregada de forma gradual desde diferentes servicios, pero contrastan marcadamente con las bases teóricas empleadas en otros sistemas de información más habituales, como los sistemas gestores de datos relacionales o en sistemas lógicos derivados de Datalog [90]. En dichos sistemas, la Asunción de Mundo Cerrado (*CWA*) en lugar de la Asunción de Mundo abierto y al mismo tiempo se observa la Asunción de Nombres únicos.

Esta última interpretación de un modelo de datos es la que resulta conveniente para validar la entrada de datos en un sistema en el que dichos datos deben cumplir unos requisitos mínimos de integridad. El desarrollo de *ICV* para *OWL* ofrece dicha interpretación para una ontología que puede ser usada al mismo tiempo para validar la integridad de los datos si se usa la interpretación *ICV* basada en CWA y UNA o para inferir nueva información si se usa la interpretación clásica de *OWL* basada en OWA y ausencia de UNA. Esta propuesta de añadir *ICV* a *OWL* ha sido recientemente implementada satisfactoriamente en algunos sistemas semánticos comerciales como *Pellet-ICV*, *Stardog* y se pretende que sea aceptada como una propuesta de estándar *W3C*.

2.3. Datos Enlazados Abiertos (*Open Linked Data*)

A pesar de la ingente cantidad de trabajo realizada por la comunidad Web Semántica, especialmente en las área de la inferencia lógica, la pila de estándares semánticos producidos por *W3C* no han encontrado un grado significativo de adopción en la industria del desarrollo web [61]. De un modo especialmente significativo, las propuestas de anotación de servicios web *REST* han quedado circunscritas al ámbito académico sin aplicación real.

Como respuesta a esta falta de adopción surge la iniciativa *OLD*, Datos Enlazados Abiertos (*Open Linked Data*) [15]. La comunidad *OLD* intenta reorientar el enfoque de la investigación en Web Semántica de las capas de razonamiento lógico hacia las capas más básica centradas en la disponibilidad de meta-datos procesables automáticamente en la web, mezclando aspectos básicos de las arquitecturas *REST* con los conceptos propios de la Web Semántica [99]:

- La importancia dada a los *URIs*, no sólo como un identificador en una ontología, sino como un mecanismo que puede ser desreferenciado mediante una petición *HTTP* para acceder a la información de un recurso identificado por ese *URI* [113].

- La necesidad de incluir enlaces entre recursos mediante el uso de *URIs* [51], aspecto no siempre explícito en los estándares semánticos, donde los *URIs* se contemplaban más como identificadores opacos que como mecanismos de desreferencición.

- Uso de las posibilidades propias del protocolo *HTTP*, desde una perspectiva arquitectónica *REST*, para, por ejemplo, decidir diferentes serializaciones de un grafo *RDF* asociado a un *URI* usando negociación de conenido [14].

- Un enfoque pragmático centrado en ofrecer soluciones válidas para la industria de desarrollo web y que puedan ser adoptadas de forma incremental en una

arquitectura que siga los principios de diseño *REST* [53].

El resultado de esta mezcla entre conceptos propios de la Web Semántica y principios arquitectónicos *REST*, es una vuelta a la concepción de la Web Semántica como una versión de la arquitectura actual de la web, basada en el protcolo HTTP y los conceptos arquitectónicos REST, pero centrada en el intercambio de datos enlazados entre servicios web.
En los siguientes apartados, enumeraremos algunos de los principales problemas que la comunidad académica relacionada con la iniciativa de Datos Enlazados Abiertos ha abordado en los últimos años.

2.3.1. Marcado Semántico

Uno de los problemas a los que se ha intentado encontrar solución desde la comunidad de Datos Enlazados Abiertos consiste en la búsqueda de mecanismos con los que insertar el contenido semántico estructurado en los documentos *HTML* generados para ser visualizados por humanos.
La meta última que se persigue con estos mecanismos es la de ofrecer dos versiones de la información en el mismo documento *HTML*, el contenido destinado a las personas, constituido por el conjunto de texto en lenguaje natural, imágenes y otros elementos de hipermedia, y al mismo tiempo, ofrecer una descripción de la semántica de esos contenidos que pueda ser automáticamente procesada por agentes software.
La primera solución propuesta a este problema, no proviene directamente de la comunidad de Datos Abiertos Enlazados, sino que es una propuesta surgida en el ámbito del desarrollo web. Esta propuesta viene dad por la comunidad de desarrollo de Microformatos [70], anteriormente mencionada.
Los Microformatos consisten en esquemas sencillos que combinan etiquetas *HTML*, atributos de esas etiquetas y ciertos valores para dichos atributos, con los que se intenta insertar contenido semántico en el código *HTML* de una página web. Existen diversas propuestas de microformatos para diferentes tipos de contenido semántico, incluyendo datos sobre personas, organizaciones, fechas, eventos, licencias etc.

El uso de Microformatos ha cosechado un éxito relativo dentro de la comunidad de desarrollo web, sin embargo, han sido, casi desde el inicio de su desarrollo, objeto de crítica por parte de la comunidad Web Semántica. Los principales inconvenientes que se han esgrimido en contra del uso de Microformatos [45] como mecanismo con el que añadir contenido semántico a documentos *HTML* incluyen su incompatibilidad con la pila de estándares semánticos propuesta por *W3C* así como el hecho de que la no observancia de ciertos estándares, como el uso de *URIs* y la preferencia por atributos en texto plano, provoca graves problemas de extensibilidad, que impiden que nuevos Microformatos puedan usarse de forma arbitraria para añadir contenido semántico a nuevos dominios de aplicación.

Estas críticas fueron recogidas por una parte de la comunidad Web Semántica primero, y de la comunidad de Datos Enlazados Abierto después, para proponer un mecanismo alternativo de marcado semántico para documentos *HTML*, extensible y que se integrase de forma satisfactoria con el resto de tecnologías semánticas de *W3C*. Este nuevo estándar conocido como *RDFa* (*Resource Description Framework in Attributes*) [1], permite la inserción de un grafo *RDF* dentro de un documento *HTML*. Para ello, *RDFa* usa una técnica similar a la propuesta por los Microformatos, usando algunos de los atributos *HTML* para señalar sujetos, predicados y objetos dentro de los tripletes *RDF* que conforman el grafo que se está insertando en el documento *HTML*. La principal diferencia respecto al uso de Microformatos, es que en lugar de usar texto plano para describir los atributos del vocabulario anotado, *RDFa* hace uso de *URIs* y de una notación especifica para *URIs* abreviados que pueden insertarse como valores de atributos *HTML* denominada *CURIEs* [123].

El resultado es que cualquier grafo *RDF* use el tipo de vocabulario que use, puede ser insertado en un document *XHTML* usando *RDFa*, con lo que se consigue una gran flexibilidad en el marcado de documentos.

A pesar de su expresividad como mecanismo de anotado semántico, *RDFa* ha sufrido críticas debido a ser considerado como de una complejidad demasiado elevada para encontrar una adopción significativa por parte de la comunidad de desarrolladores web. En respuesta a esta crítica algunas de las principales empresas web como *Google*,

Yahoo y *Microsoft* han propuesto, dentro del proceso de estandarización de *HTML5*, *Microdata* [56], un mecanismo alternativo a *RDFa* que intenta ofrecer una alternativa más simple para la inserción de contenido semántico en documentos *HTML*. Microdata es una propuesta de estándar que añade una serie de atributos nuevos a *HTML* con los que describir contenido semántico, de una forma extensible, cumpliendo algunos estándares *W3C*, como el uso de *URIs*, pero sin ofrecer toda la capacidad expresiva de *RDFa*. Microdata usa un modelo de datos, consistente en un árbol de relaciones entre elementos del documento *HTML*, diferente al modelo de grafo *RDF* que propone *RDFa*. A pesar de estas diferencias, algunos resultados recientes [59] han propuesto mecanismos para transformar un árbol de relaciones Microdata anotado en un documento *HTML* en un grafo *RDF*, con lo que sería posible integrar documentos Microdata con el resto de estándares semánticos *W3C*. Del mismo modo que exiten un determinado número de vocabularios propuestos para ser usados con Microformatos, algunos vocabularios han sido propuestos por las compañías detrás del esfuerzo de estandarización de Microdata para determinados dominios de aplicación [110]. Dichos vocabularios han sido también transformados en ontologías *RDF* que pueden ser utilizadas en documentos anotados con *RDFa*, consiguiéndose de esta manera un cierto grado de inter-operabilidad entre ambos estándares.

2.3.2. Desreferenciación de recursos web

Como se ha mencionado anteriormente, una de las principales aportaciones que la iniciativa Datos Enlazados Abiertos a aportado al enfoque tradicional de investigación en Web Semántica es el relativo a la importancia de los *URIs* como identificadores de recursos que pueden ser obtenidos mediante operaciones *HTTP*, equiparándolos de esta manera a recursos expuestos por servicios web, tal y como son descritos en las arquitecturas *REST*. El problema surge a la hora de encontrar una representación adecuada que devolver a un agente web que intenta desreferenciar una *URI* asociada a un recurso presente en una ontología *RDF* y que puede representar un concepto abstracto y no desreferenciable.

Este problema es conocido dentro de la comunidad de Datos Enlazados Abiertos como *HTTP Range-14* [39]. Se han propuesto diversas alternativas para intentar abordar este problema.

En primer lugar, se considera que recursos de información, como documentos *HTML* deben ser identificados por *URIs* que no acaben en un fragmento introducido por el carácter "#", mientras que *URIs* que identifican recursos abstractos que no son de información y no desreferenciables, deberían usar *URIs* con fragmentos. Esto permitiría al servicio que expone el recurso tomar una acción adecuada y responder con un código de estado *HTTP* y cabeceras adecuadas para que el agente web pueda acceder a una representación de dicho recurso.

En concreto, si el recurso es un recurso de información y puede ser desreferenciado, el servicio debe devolver un código *200* pero si el recurso no es de información y no puede ser desreferenciado, el servicio debe devolver un código de redirección *300* y ofrecer una cabecera con información sobre recursos de información asociados, que supongan una representación adecuada para el concepto abstracto que se intenta desreferenciar. Esta solución cuenta con el inconveniente de la gran cantidad de ontologías que ya han sido propuestas y que son ampliamente empleadas, que no utilizan el formato correcto para las *URIs* de conceptos abstractos.

2.3.3. RESTful *SPARQL*

Uno de los problemas a los que se ha enfrentado la comunidad de Datos Abiertos Enlazados ha consistido en exponer como servicios web ingentes cantidades de información codificada como grafos *RDF* procesables automáticamente. Como ya se ha mencionado en este documento, *SPARQL* constituye el estándar semántico básico usado para recuperar información en un grafo *RDF*. Una de los primeros intentos de solución propuestos por la comunidad de Datos Abiertos Enlazados consistió en exponer directamente grafos *RDF* a través de puntos *SPARQL* accesibles mediante el protocolo *HTTP*. Una propuesta de estándar web fue creada al respecto, especificando un protocolo [27] que permitía intercambiar consultas *SPARQL* y resultados entre un punto *SPARQL* y un cliente *HTTP*.

Sin embargo, este mecanismo de intercambio de consultas *SPARQL* entra en conflicto con los principios arquitectónicos básicos *REST*, ya que ignora el concepto de recurso latente en estándares semánticos como *RDF* y propone por el contrario un mecanismo basado en la invocación remota de procedimientos que se traducen en consultas *SPARQL* y la correspondiente respuesta *XML*.

Una posible solución para reconciliar el uso de *SPARQL* como un mecanismo viable para exponer grafos *RDF* a través de una interfaz *HTTP* viene de la mano del concepto de grafo con nombre introducida por la especificación de *SPARQL* [132]. *SPARQL* permite asociar un *URI* identificativo a un determinado grafo *RDF* y ejecutar una consulta sobre uno o más grafo. De esta manera, la unidad básica de las arquitecturas *REST*, el recurso web asociado a un *URI*, se asimilaría a la de grafo *RDF* con nombre en *SPARQL*, sobre el que se ejecutaría diferentes operaciones *HTTP*, cuya semántica quedaría recogida en las correspondientes consultas *SPARQL* y *SPARQL Update*, para recuperar información del grafo, eliminarla o modificarla. Esta línea de trabajo se ha formalizado en una propuesta de estándar web en *W3C*, conocida como Protocolo de Almacenamiento de Grafos para *SPARQL* 1.1 [97] o la arquitectura de servicios Pubby [30].

Dicha propuesta de estándar propone una traducción de las operaciones *HTTP* a diferentes consultas *SPARQL* y *SPARQL* Update para que sean ejecutadas sobre grafos *RDF* con nombre con una *URI* asociada y que se expondría a través del protocolo *HTTP* como recursos *REST*.

Una última alternativa más simple para la construcción de una interfaz *HTTP* para un grafo *RDF* consiste en la traducción del grafo entidades atributo-valor. En este tipo de *APIs*, el servicio oculta la naturaleza *RDF* de los datos expuestos, así como el modelo de datos *RDF*, traduciendo conjuntos de tripletes *RDF* a objetos que contienen pares clave valor. Estos objetos son codificados en los parámetros de las peticiones *HTTP* enviadas por los clientes y son devueltos en las respuestas *HTTP*, normalmente serializados como objetos *JSON* con atributos planos. Ejemplos de estas *APIs* son la Propuesta de *API* para Datos Enlazados (*LD-API*) [108] y las versiones que usan *RDF* como formato último de datos de la *API* para un Protocolo de Datos

Abiertos [37]. Las interfaces basadas en entidades atributo-valor son compatibles con los principios arquitectónicos *REST* y tienen una interfaz muy familiar para cualquier desarollador web. Normalmente introducen mecanismos ad-hoc para lidiar con problemas prácticos propios del desarrollo web, como la paginación de recursos en colecciones. Por otro lado, incurren en ciertos problemas desde el punto de vista de las recomendaciones sobre Datos Enlazados Abiertos, como el uso de *URIs* ocultos en los atributos de los objetos, lo que puede hacer imposible el enlazado efectivo entre diferentes *APIs*.

2.3.4. *JSON* Enlazado

Un último desarrollo interesante encontrado dentro de la comunidad de Datos Enlazados Abiertos, consiste, en la búsqueda de serializaciones válidas para grafos *RDF* como objetos *JSON* [29].

Como ya se ha mencionado, *JSON* es el formato de intercambio preferido por los desarrolladores web en la actualidad a la hora de desarrollar *APIs* de datos. Sin embargo, *JSON* cuenta con graves inconvenientes como serialización de recursos, entre ellos, la imposibilidad de identificar al recurso que ha generado el objeto *JSON* dentro del objeto de una forma estándar, así como la imposibilidad de denotar de una forma estándar *URIs* en los pares claves valor que conforman el documento *JSON*.

JSON-LD (*JSON* para Datos Enlazados) [120] ha sido una de las diferentes propuestas [3] para insertar grafos *RDF* dentro de documentos *JSON*. *JSON-LD* propone atributos especiales que denotan la identidad del recurso siendo serializado en el objeto, así como atributos que permiten identificar qué claves y qué valores del objeto son *URIs* y cuáles algún otro tipo de dato.

JSON-LD tiene un diseño muy flexible que permite serializar cualquier tipo de grafo *RDF* en una colección de objetos *JSON*, pero al mismo tiempo, el uso de valores convencionales por defecto, permite que en la mayoría de los casos, el aumento en complejidad que supone el uso de *JSON-LD* frente a un objeto *JSON* equivalente que no intente representar la serialización de un grafo *RDF*, se vea restringido al uso

de un par de atributos adicionales.

2.3.5. Autenticación y WebID

Un problema asociado a la construcción de servicios web en general y a la construcción de servicios web semánticos y *APIs* de datos enlazados abiertos, consiste en la autenticación de los agentes que intentan acceder a la información expuesta por el servicio. Diversos mecanismos de autenticación han sido propuestos dentro de la comunidad de desarrollo de aplicaciones web, siendo la más popular *OAuth* [49]. Diferentes proveedores de servicios web, han desarrollados mecanismos de autenticación para sus plataformas, que pueden ser utilizados como un servicio de autenticacón para realizar la autenticación de usuarios en aplicaciones desarrolladas por terceros. Un ejemplo de estos servicios es *Facebook Connect* [84], desarrollado por *Facebook*. Un intento de estandarizar y unificar estos servicios de autenticación se encuentra en el estándar *OpenID* [106] para ofrecer un servicio de autenticación único para usuarios de diferentes plataformas y aplicaciones, que sin embargo ha encontrado un grado de adopción pequeño por parte de los usuarios [122]. En los últimos meses, se asiste al desarrollo de nuevos protocolos de autenticación que hacen un especial énfasis en la consecución de un mecanismo de autenticación distribuído que reduce el papel de la autoridad central de autenticación, un ejemplo de estos mecanismos es *BrowserID* desarrollado por la *Fundación Mozilla* y basado en el uso de la dirección de correo electrónico y en la integración con el navegador web. Dentro de la comunidad *Open Linked Data* se ha desarrollado también un mecanismo de autenticación distribuido conocido como *WebID* [121]. Dicho mecanismo se basa en el uso del elemento basico de la tecnología web, el *URI*, al que se asocia una identidad usando criptografía asimétrica bajo la forma de un certificado público que los agentes web pueden usar al desreferenciar una URL. El servicio web que recibe la petición puede seguir la *URI* asociada al certificado, recuperando de esta manera un document *RDF* con la información necesaria para comprobar la autenticidad del certificado, así como información adicional sobre contactos, pérfil, etc. asociada a esa identidad. De esta manera

se consigue un mecanismo de autenticación verdaderamente distribuido y basado en los principios y elementos básicos web. El mecanismo a su vez se puede integrar fácilmente con el agente web más característico, el navegador web, debido al soporte que para el uso de certificados se encuentra ya integrado en los navegadores comerciales.

2.3.6. Equivalencia entre el modelo de datos *RDF* y el modelo relacional

Otras las áreas en las que la comunidad de Datos Enlazados Abiertos ha desarrollado su actividad es la de la búsqueda de mecanismos que permitan la transformación de datos expresados en el modelo de datos *RDF* como datos relacionales almacenados en un sistema gestor de bases de datos relacional.

Tradicionalmente, la comunidad Web Semántica ha trabajado en el desarrollo de tecnologías específicas para el almacenamiento y recuperación de datos desde un grafo *RDF*, buscando obtener la máxima eficiencia posible en el acceso a los datos semánticos.

Sin embargo, la tecnología más extendida en el mundo del desarrollo web para almacenar la capa de datos de una *API* de servicios web consiste en el uso de un sistema gestor de bases de datos relacional, que permite el acceso a los datos almacenados mediante el lenguaje de consulta *SQL*.

La comunidad de Datos Enlazados Abiertos ha intentado encontrar un mecanismo que permita trasladar el modelo de datos *RDF* al modelo relacional de forma que el almacenamiento de datos semánticos *RDF* pueda llevarse a cabo usando bases de datos relacionales, acercando de esta manera el uso de *RDF* a la mayoría de desarrolladores web. El resultado de este esfuerzo es un borrador de propuesta de estándar *W3C* denominada Lenguaje de Mapeado Relacional a *RDF* (*R2RML*) [32]. *R2RML* consiste en una ontología genérica que permite establecer una correspondencia entre columnas en un esquema relacional y los componentes de los tripletes de un grafo *RDF*., siendo este mecanismo lo suficientemente potente como para almacenar cualquier grafo *RDF* en las tablas de un sistema gestor de bases de datos relacionales.

R2RML, también incluye soporte para algunas ideas aparecidas dentro de la comunidad de Datos Enlazados Abiertos, como los mencionados grafos con nombre.

2.4. Computación Distribuida

El consumo de *APIs* de servicios web por parte de agentes software con el fin de llevar a cabo alguna finalidad determinado puede llegar a presentar situaciones de gran complejidad donde diferentes agentes y servicios intercambian peticiones que requieren la coordinación de las partes involucradas para ser llevadas a cabo con éxito. Nos encontramos pues ante un escenario donde la computación se realiza de forma distribuida entre todos los agentes y servicios involucrados en ellas y que puede ser analizada y modelizada usando las herramientas teóricas que han sido desarrolladas para el estudio dentro del area de investigación en computación distribuida.
En las siguientes secciones se detallan algunos de los modelos y desarrollos en este área que han sido utilizados para la elaboración de esta tesis.

2.4.1. Espacios de tuplas y espacios de tripletes

Los espacios de tuplas [23] es una propuesta de mecanismo de comunicación entre procesos que realizan algún tipo de computación distribuida. Un espacio de tuplas consiste en una memoria asociativa que puede ser accedida concurrentemente por diferentes procesos para almacenar, recuperar o eliminar unidades de información de composición variable conocidas como tuplas.
Cuando dos procesos necesitan coordinarse a través del espacio de tuplas, el proceso consumidor de información intenta realizar una operación de lectura en el espacio de tuplas para un determinado patrón de datos, quedando en un estado bloqueado si ninguna tupla que satisfaga las condiciones impuestas por el patrón de lectura se encuentra disponible en ese momento. Cuando el proceso productor desea notificar al proceso consumidor una condición determinada en la computación, puede insertar en el espacio de tuplas una nueva tupla de información que cumpla con el patrón

de lectura del proceso consumidor. En el momento en que la tupla es insertada en el espacio de tuplas, el proceso consumidor es desbloqueado y recibe la nueva tupla proveniente del proceso productor que quería señalar un determinado evento a los procesos consumidores. Este mecanismo, basado en una memoria asociativa usada como mecanismo de coordinación para los procesos involucrados en una computación distribuida, se conoce también como sistemas de pizarra (*blackboard systems*) [95].
Se ha demostrado que los sistemas de espacio de tuplas son Turing-completos [21]. La implementación práctica de las ideas expuestas en el modelo de computación dado por los espacios de tuplas, bajo la denominación de comunicación generativa, es Linda [43]. Las operaciones que sobre el espacio de tuplas ofrecía Linda se puede enumeran a continuación:

- **In:** Consume atómicamente, extrayéndola de la memoria, una tupla del espacio de tuplas.
- **Rd:** Lee atómicamente, sin extraerla de la memoria, una tupla del espacio de tuplas.
- **Out:** Escribe una nueva tupla en el espacio de tuplas.
- **Eval:** Crea un nuevo proceso que interaccionará con el espacio de tuplas usando alguna de las operaciones anteriormente mencionadas.

En su concepción original, las tuplas almacenadas en el espacio de tuplas, consisten en estructuras arbitrarias de datos, sin embargo, es posible adaptar el modelo de computación al uso de datos semánticos, compuestos por tripletes *RDF* con sujeto-predicado-objeto [38]. Desde este punto de vista, un espacio de tuplas se convierte en un espacio de tripletes, que puede ser manipulado por los agentes involucrados en la computación a través de las operaciones básicas de manipulación de espacios de tuplas.

2.4.2. Cálculos de procesos

Un problema básico dentro del estudio de la computación distribuida es el del modelado formal de los sistemas concurrentes, de tal forma que los problemas propios de este tipo de computación, como la bisimulación entre sistemas, puedan ser formalizados y analizados mediante una serie de transformaciones algebraicas definidas en dicho cálculo.
Existe un gran número de propuestas de diferentes cálculos de procesos en la literatura: *CSP* [60], *CCS* [85], *Ambient Calculus* [24], etc. Entre este grupo de formalismos destaca el Cálculo Pi (Pi-Calculus) [88]. En el Cálculo Pi, una computación concurrente es llevada a cabo por una red de procesos intercambiando información a través de canales. Los procesos pueden leer de forma bloqueante y escribir a través de estos canales, de tal forma que pueden ser utilizados como mecanismos de coordinación. La principal característica del Cálculo Pi es que los mismos canales también pueden ser enviados a través de los canales, de tal forma que los procesos que llevan a cabo la computación pueden ganar acceso a nuevos procesos a través del intercambio de canales, haciendo que la red de procesos cambie y evolucione en el tiempo.

Las principales primitivas presentes en el Cálculo Pi son las siguientes:

- **Ejecución concurrente de procesos:** múltiples procesos están siendo ejecutados de forma simultánea en el sistema.
- **Comunicación entre procesos a través de un canal:** la lectura y escritura bloqueante de datos a través de un canal permite la coordinación entre procesos.
- **Replicación:** primitiva que permite la creación de una nueva instancia de un proceso que se ejecutará de forma concurrente.
- **Restricción de nombres:** primitiva que permite a un proceso reservar un nuevo identificador constante que actuará como un nuevo canal por el que transmitir información.

- **Terminación:** Permite a un proceso terminar su ejecución, denotándose en el cálculo con el uso de un proceso especial 0.

Con estas primitivas básicas el Cálculo Pi constituye un modelo de computación Turing completo [86].
En el mundo de los servicios web *WS-**, el Cálculo Pi y los demás formalismos relacionados han sido utilizados reiteradamente como la base teórica sobre la que construir mecanismos automatizados para la composición y la orquestación de servicios web, un ejemplo paradigmático es el intento de usar el Cálculo Pi para definir la semántica de la propuesta de estándar para un lenguaje de orquestación de servicios web WS-BPEL [81]. El tratamiento formal que el uso de un cálculo de proceso aporta al estudio de los escenarios complejos de interacción entre agentes y *APIs* de servicios web lo han convertido en una base teórica atractiva con la que abordar problemas complejos como la simulación, la verificación y detección de errores en el consumo de servicios web [94], [13].

Capítulo 3

Descripción de la Solución

Nuestra meta en este capítulo es analizar la capa de servicios web de una aplicación, desde la definición formal de dicha interfaz hasta la implementación de herramientas y librerías software que permitan el desarrollo de *APIs* que combinen las tecnologías semánticas con los principios arquitecturales web *REST*, para ofrecer una alternativa pragmática y viable a desarrolladores web que solucione las limitaciones de las que adolecen las técnicas actuales de desarrollo de *APIs*.
Los principales elementos de este análisis que se desarrollarán en este documento son los siguientes:

- **Modelo formal** de la capa de servicios web y agentes consumiendo dichos servicios, que constituye la interacción básica de cualquier *API* de datos.
- **Diseño de una solución arquitectónica** basada en los principios de arquitectura *REST* que permita exponer datos enlazados en nuevas *APIs* y servicios. Algunas de las características con las que debe contar esta arquitectura pasan por la separación radical entre clientes y capa de servicio, el uso de la negociación de contenido para obtener la adecuada representación de los datos por parte del cliente, ser completamente agnóstica en cuanto al mecanismo de almacenamiento del grafo de datos enlazados usado en la implementación final y el uso del concepto de recurso web como la unidad fundamental de datos que

son expuestos y agregados.

- **Implementación componentes software** que permitan la fácil construcción de aplicaciones clientes y servidores que consuman datos enlazados a través de la arquitectura para *APIs* semánticas anteriormente especificada. Así mismo también se mostrará cómo se pueden adaptar tecnologías ampliamente usadas hoy en día para facilitar el almacenamiento de datos enlazados, de tal forma que puedan ser utilizadas como repositorios de datos para los recursos expuestos a través de la arquitectura para *APIs* desarrollada.

La meta última de este trabajo es aplicar dichas tecnologías y herramientas a la construcción de una aplicación web social, completamente descentralizada, donde la interconexión de diferentes servicios asociados a distintos usuarios, controlando en todo momento los datos que generan así como su identidad web, permita replicar la funcionalidad de las aplicaciones sociales centralizadas y difícilmente inter-operables disponibles hoy en día. Los ejemplos de aplicación que describiremos abordan estos probelmas y servirán como validación de la aplicabilidad de la propuesta de modelo arquitectónico que hemos desarrollado.

3.1. Model Formal

Como hemos visto en los apartados del estado del arte relacionados con la construcción de servicios web semánticos, existen diferentes propuestas, para añadir una capa semántica a las *APIs* de servicios web *REST* expuestas por diferentes aplicaciones, por ejemplo, *hRESTS* [73] o *SA-REST* [115]. Sin embargo, estas propuestas se centran en detalles técnicos como diferentes técnicas de marcado semántico y vocabularios específicos para describir los componentes esenciales de lo que supone el modelo *REST* de servicios web sin ofrecer un verdadero modelo formal que permita describir soluciones computacionales a diferentes problemas, mediante el uso de servicios web *REST* y meta-datos semánticos. Dicho modelo, debería ser capaz de permitir dicha descripción y servir de base sobre la que construir los diferentes componentes tec-

nológicos necesarios para implementar la solución descrita.
En este capítulo propondremos un posible modelo formal que puede ser utilizado para describir de forma genérica cualquier computación basada en el uso de servicios web *REST* semánticos.
Nuestro modelo se basa en dos referentes teóricos básicos dentro del área de la computación distribuida:

- El uso de espacios de tripletes [38] como un caso particular, adaptado al modelo de datos *RDF* de los espacios de tuplas [23], mecanismo de comunicación entre procesos mediante una memoria asociativa propuesto por Gelernter en sus artículos sobre el sistem Linda [43].

- El Cálculo Pi [88] un tipo de cálculo de procesos, en el que los procesos se comunican unos con otros a través del intercambio de mensajes a través de canales con nombre identificativo. Dichos mensajes pueden a su vez incluir los identificadores de los canales, con lo que los procesos pueden acceder a canales con los que comunicarse con nuevos procesos simplemente recibiéndolos en un mensaje, variando de esta forma dinámicamente con el tiempo la topología de la red de canales y procesos que llevan a cabo una determinada computación.

El uso de estas dos bases teóricas en nuestra propuesta de cálculo formal es debida a la identificación de los componentes principales de la computación mediante servicios *REST* con los componentes básicos de los modelos de espacios de tripletes y el Cálculo Pi.

Esta identificación se basa en los siguientes supuestos:

- Un recurso *REST* semántico consiste en un conjunto de tripletes *RDF* almacenados en en un repositorio accesible por los procesos participando en la computación. Un recurso *REST* semántico consiste en un espacio de tripletes.

- El espacio de tripletes que constituye el recurso *REST* semántico es accesible a través de un *URI* asociado que puede ser enlazado desde otros recursos

relacionados. Los *URIs* asociados a los recursos se pueden considerar pues canales con identificadores como los que se usan los procesos del Cálculo Pi para intercambiar mensajes.

- El espacio de tripletes que que contiene los tripletes *RDF* del recurso puede ser manipulado mediante peticiones *HTTP* sobre el *URI* asociado al recurso, de acuerdo con la semántica propia de la interfaz uniforme *HTTP*. De este modo, la semántica de la interfaz *HTTP* se puede traducir a las operaciones básicas del Cálculo Pi y por último a las operaciones elementales sobre un espacio de tripletes.

Ejemplos de entornos de servicios web que cumplen estos supuestos son, servicios web en los que se intercambian documento *HTML* con anotaciones semánticas insertadas usando el estándar *RDFa* o *APIs* de servicios web que permiten a acceder a grafos *RDF* almacenados en un repositorio de tripletes a través de una interfaz *HTTP REST*.

En los siguientes apartados explicaremos pormenorizadamente los detalles que permiten la identificación de un recurso *REST* semántico tanto con un espacio de tripletes como con un proceso dentro del Cálculo Pi y que forman la base de nuestra propuesta de cálculo de procesos para servicios web semánticos.

3.1.1. Recursos semánticos y espacios de tripletes

La base del cálculo es la manipulación de meta-datos semánticos representados como tripletes *RDF*. Los componentes de un triplete pueden consistir en *URIs* o literales. Cualquier computación descrita en el cálculo consiste en la manipulación de tripletes almacenados en espacios de datos compartidos denominados espacios de tripletes [38] por un conjunto de procesos distribuidos.

Este modelo de computación distribuida se conoce como comunicación generativa de acuerdo con la terminología propia de los sistemas de espacios de tuplas como Linda [43]. En estos sistemas se describe un conjunto finito de operaciones sobre el espacio de tuplas. Estas operaciones se pueden adaptar para manipular tripletes almacenados

en un espacio de tripletes:

- **Rd:** Operación que permite leer tripletes del espacio de tripletes sin eliminarlos del repositorio.
- **Rdb:** Versión bloqueante de la operación rd.
- **In:** Operación que permite leer tripletes del espacio de tripletes eliminándolos del repositorio al mismo tiempo.
- **Inb:** Versión bloquente de la operación in.
- **Out:** Operación que permite insertar tripletes en el espacio de tripletes.
- **Notify [22]:** Operación que permite a los procesos recibir una notificación cuando otros procesos manipulan el espacio de tripletes de una determinada manera.
- **Swap [12]:** Operación que combina lectura, borrado y escritura en una sola operación atómica sobre el espacio de tripletes.

Las operaciones definidas en el cáculo sobre espacios de tripletes aceptan como argumentos tanto conjuntos de tripletes (v), como patrones (p). La semántica particular del mecanismo de definición de patrones y su aplicación sobre el espacio de tripletes puede variar sin afectar a la semántica del cálculo. En este documento supondremos un mecanismo basado en un subconjunto de *SPARQL* consistente sólo en la aplicación de patrones básicos de grafo (**BGP**) con sustitución simple de nombres [104].
Un patrón recibido en una operación puede ser aplicado a un conjunto de tripletes ($< p, v >$) o a un espacio de tripletes ($< p, \theta >$), obteniéndose como resultado una colección de substituciones para las variables del patrón. Aplicando estas substituciones al patrón el conjunto de tripletes que satisfacen dicho patrón es obtenido finalmente. El resultado de la aplicación del patrón puede ser el conjunto vacío, si ninguna substitución es obtenida.

$$
\begin{array}{lcl}
P & ::= & 0 \mid T \mid P|P \mid !P \mid if\ T\ ?\ P.P \mid x ::= T \\
T & ::= & rd(\theta_i, p) \mid in(\theta_i, p) \mid out(\theta_i, v) \mid swap(\theta_i, p, v) \mid \\
 & & rdb(\theta_i, p) \mid inb(\theta_i, p) \mid notify(\theta_i, \rho, v) \\
 & & \\
\theta & ::= & \{\text{ triple spaces }\} \\
\rho & ::= & \{in, out\} \\
\mu & ::= & \{\text{URIs}\} \\
\lambda & ::= & \{\text{literals}\} \\
p & ::= & \{\text{patterns}\} \\
v & ::= & \{\text{values}\} = \{\mu\} \cup \{\lambda\} \cup < p, v > \cup < p, \theta_i >
\end{array}
$$

Cuadro 3.1: Sintaxis formal del calculo relativa al espacio de tripletes y elementos básicos.

Una definición exacta de la sintaxis y semántica operacional del cálculo, se pueden encontrar en los cuadros 3.1 y 3.2.

La semántica operacional del cálculo para la sintaxis formal aquí introducida sigue formalizaciones anteriores propuestas para otros sistemas de espacios de tuplas de tipo Linda [43]. En particular, hemos elegido una semántica con ordenación para las operaciones sobre el espacio de tripletes. Como consecuencia, la emisión y aplicación de mensajes se puede considerar como una única operación atómica. Se puede demostrar que los sistemas de tipo Linda con semanticas con con ordenación son Turing completos [21].

La semántica operacional descrita en el cuadro 3.2, define una relación ($\rightarrow$) para las operaciones como la más simple interpretación que satisface las reglas (1) (10). De éstas, las comprendidas entre la regla (4) y la regla (6) definen las principales operaciones sobre espacios de tripletes que modifican los tripletes almacenados.
La regla (7) muestra la semántica de la operación *swap* como una combinación en un sólo paso de reducción de la semántica de las operaciones *in* y out. Por su parte, las reglas (8) y (9) definen las semántica para la operación *notify*. Mostrando su relación con las operaciones *in*, *out* y con el patrón pasado como argumento a la operación *notify* aplicad a los tripletes insertados o eliminados por las operaciones *in* y *out*.

(1) $$\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

(2) $$\frac{P \rightarrow P'}{Q \rightarrow Q'} if\, P \equiv Q \,and\, P' \equiv Q'$$

(3) $$!P.Q \rightarrow Q \mid P$$

(4) $$\frac{rd(\theta_i,p).P}{rd(\theta_i,p).P \xrightarrow{<p,\theta_i>} P}$$

(5) $$\frac{in(\theta_i,p).P}{rd(\theta_i,p).P \xrightarrow{<p,\theta_i>} P, \theta_i = \theta_i - <p,\theta_i>}$$

(6) $$\frac{out(\theta_i,v).P}{out(\theta_i,v).P \xrightarrow{\overline{v}} P, \theta_i = \theta_i \cup v}$$

(7) $$\frac{swap(\theta_i,p,v).P}{swap(\theta_i,p,v).P \xrightarrow{<p,\theta_i>,\overline{v}} P, \theta_i = \theta_i - <p,\theta_i> \cup v}$$

(8) $$\frac{out(\theta_i,v).Q}{notify(\theta_i,out,p).P|out(\theta_i,v).Q \xrightarrow{\overline{v},<p,v>} P|Q}$$

(9) $$\frac{in(\theta_i,p).Q}{notify(\theta_i,in,q).P|in(\theta_i,p).Q \xrightarrow{<p,v>,<q,<p,v>>} P|Q}$$

(10) $$\frac{if\ T\ P.Q}{if\ T\ P.Q \xrightarrow{\overline{0}} Q}, \quad \frac{if\ T\ P.Q}{if\ T\ P.Q \xrightarrow{\overline{v}} P}$$

Cuadro 3.2: Semántica operacional del calculo relativa al espacio de tripletes y operaciones básicas.

Como ya mencionamos, nuestra propuesta de cálculo se basa en la identificación de los elementos básicos del modelo de servicios *REST* con los axiomas formales introducidos en el modelo formal. Desde este punto de vista las siguientes identidades entre cálculo y módelo *REST*:

- Un conjunto de tripletes en un espacio de tripletes puede ser identificado como un recurso *HTTP*.
- Un identificador del espacio de tripletes puede hacerse equivalente a un *URI* asociado a un recurso *HTTP*.
- Las operaciones *GET* del protocolo *HTTP* son equivalentes a operaciones *rd* sobre el espacio de tripletes.
- Las operaciones *POST* del protocolo *HTTP* son equivalentes a operaciones *out* sobre el espacio de tripletes.
- Las operaciones *PUT* del protocolo *HTTP* son equivalentes a operaciones *swap* sobre el espacio de tripletes.
- Las operaciones *DELETE* del protocolo *HTTP* son equivalentes a operaciones *in* sobre el espacio de tripletes.

3.1.2. Recursos semánticos y procesos en tiempo de ejecución

Las sintaxis y semántica introducidas en el cálculo en el punto anterior y relativas a la visión de los recursos *HTTP* semánticos como tripletes almacenados en un espacio de tripletes no es suficiente para describir una computación completa involucrando servicios web *REST* semánticos.

En primer lugar, en las concepciones originales de los espacios de tripletes, el espacio de tripletes era global, único y compartido. Sin embargo, para modelar recursos *REST* semánticos, es conveniente considerar cada recurso individual como un espacio de tripletes diferente e identificado por un *URI*. Del mismo modo, estos espacios de tripletes no serían estáticos, sino que podrían ser creados mediante operaciones *HTTP*

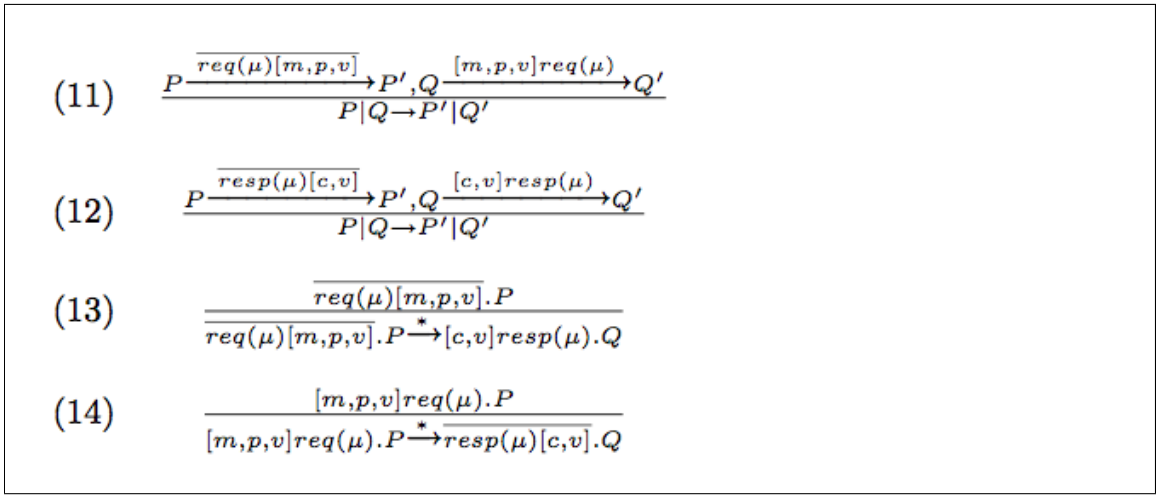
POST y destruidos mediante operaciones *HTTP DELETE*. La identificación de las operaciones *in* y *out* con los métodos *HTTP DELETE* y *POST* sería pues incompleta, ya que tendrían como objeto tripletes individuales y no el conjunto del espacio de tripletes. El formalismo puede ser extendido [116] añadiendo nuevas operaciones tanto sobre espacios de tripletes como sobre tripletes individuales. Teniendo en cuenta estas extensiones, los espacios de tripletes podrían ser concebidos como procesos en el cálculo que pueden ser creados, recibir mensajes de otros procesos y terminar su ejecución.

Otra importante identidad entre servicios *REST* y espacios de tripletes introducida ha sido la capacidad de asociar un *URI* a un espacio de tripletes. De este modo, los procesos pueden ganar acceso a los tripletes de un recurso almacenados en un espacio de tripletes a través de los *URIs* codificados en sus componentes. Desde este punto de vista, los espacios de tripletes pueden ser considerados no sólo procesos dinámicos, sino como procesos móbiles tal y como los descritos en el Cálculo Pi, ya que estos procesos podrían coordinarse a través de canales con nombre (*URI*) que se intercambiarían dentro de los componentes incluidos en los mensajes (tripletes) enviados y recibidos por los procesos.

En los cuadros 3.1 y 3.2 se definen la sintaxis y semántica operacional que permite describir los espacios de tripletes como procesos dinámicos mediante la definición de la composición paralela de procesos (1) así como la congruencia estructural de procesos (2) [88]. Por su parte (3) define el proceso de creación de un nuevo proceso y por último (10) define un tipo sencillo de ejecución condicional.

Además de estas primitivas para la creación y composición de procesos, el cálculo debe ser extendido con nuevas operaciones, como las mostradas en el cuadro 3.3, que corresponden a variaciones de las operaciones descritas en el Cálculo Pi Poliádico [87].

En dichas operaciones, los mensajes pueden ser enviados a través canales con nombre representados como *URIs* (μ). Los mensajes pueden ser de dos tipos, peticiones (req) o respuestas ($resp$) y los procesos pueden enviar y recibir ambos tipos de mensajes. A su vez, los mensajes se componen de un método (m), un patrón (p) y un



Cuadro 3.3: Semántica operacional del cálculo relativa a la comunicación entre procesos.

valor (v) en el caso de las peticiones y de un valor (v) y un código (c) en el caso de las respuestas. Adicionalmente, también se define una operación para introducir un nuevo identificador (URI) en la computación. ($new\, \mu\, in\, P$).
La semántica operacional para estas operaciones se introduce en el cuadro 3.3 de acuerdo con la semántica definida en el Cálculo Pi Poliádico. Las reglas (11) y (12) son una adaptación a la sintaxis de nuestro cálculo de la regla de comunicación del Cálculo Pi, mostrando la reacción entre procesos enviando y recibiendo mensajes. Las reglas (13) y (14), imponen restricciones sobre el orden en que peticiones y respuestas deben ser intercambiados. Según la semántica definida, sin un proceso envía un mensaje con una petición a otro proceso, debe esperar un mensaje con una respuesta tras un número finito de reducciones a través del mismo *URI*. De modo simétrico, si un proceso recibe una petición a través de un *URI* debe enviar una respuesta tras un cierto número de reducciones.

Esta semántica permite modelar de una forma más útil en el cálculo las peticiones *HTTP* a recursos *REST* semánticos como mensajes de petición enviados a procesos a través de canales definidos como *URIs* a recursos *REST*. Estos mensajes consisten en un método más un patrón o tripletes y la respuestas vendrían dadas por un código de respuesta más un conjunto de tripletes. Es posible crear nuevos recursos *REST* semánticos como resultado de peticiones POST asociados a nuevos *URIs* introducidos

en la computación y su ejecución podría terminar como resultado de una petición DELETE. El sistema compuesto por estos recursos *REST* semánticos sería un cálculo de procesos móbiles dado que en los mensajes de respuesta provenientes de un recurso, podrían encontrarse *URIs* asociados a diferentes recursos.

3.1.3. Modelado de recursos *REST* semánticos

En las dos secciones previas, hemos introducido la notación y la semántica de nuestro cálculo que permiten la descripción de los recursos *REST* semánticos usando dos formalismos diferentes, los espacios de tripletes y el cálculo de procesos móviles. Ambos mecanismos son complementarios siendo cada uno más adecuado para la descripción de diferentes aspectos de la computación con recursos *REST* semánticos. Los espacios de tripletes ofrecen una excelente descripción de los recursos *REST* semánticos como repositorios estáticos de datos semánticos, usando operaciones sobre los datos semánticos como el principal mecanismo de coordinación entre los procesos manipulando dichos repositorios. De modo similar, el cálculo de procesos es adecuado para describir los aspectos dinámicos del protocolo *HTTP*, como el mecanismo de paso de mensajes a través de canales con nombres, asociados a *URIs*, entre procesos agentes y procesos que constituyen los recursos *REST* semánticos.
En esta sección una se define formalmente de una forma definitiva el concepto de recurso *REST* semántico, que combina aspectos de ambos formalismos, usando para ello la sintaxis y la semántica introducidas en las secciones anteriores.

Las principales características de esta formalización de la computación basada en recursos *REST* semánticos se enumeran a continuación:

- Cualquier computación es llevada a cabo por procesos distribuidos: agentes y recursos.
- Cualquier proceso tiene un número de espacios de tripletes asociados que pueden ser manipulados por las operaciones sobre espacios de tripletes introducidas en

la sección 3.1.1.

- Un cierto número de procesos compartiendo el acceso al mismo espacio de tripletes se pueden agrupar en una localización computacional. Una aplicación web compuesta de multiples recursos web es un ejemplo de localización computacional. Un navegador web sería otro ejemplo de localización computacional compuesta únicamente de procesos agente.

- Los procesos recurso tienen al menos un canal con nombre asociado (URI) por el que pueden recibir mensajes de tipo petición (req) enviados por otros procesos.

- Los procesos agente no tienen un canal con nombre asociado (URI), por lo que no pueden recibir mensajes de tipo petición, pero pueden enviar mensajes de tipo petición y recibir respuesta (resp) a procesos de tipo recurso a través de las *URIs* almacenadas en sus espacios de tripletes.

- La coordinación entre los procesos dentro de la misma localización de computación se basa en las operaciones y semántica definidas en el cálculo para las operaciones sobre espacios de tripletes.

- Los canales con nombres ($URIs$) pueden ser intercambiados a través de mensajes de petición y respuesta y almacenados en espacios de tripletes como parte de los componentes de los tripletes.

- La única coordinación posible entre procesos en diferentes localizaciones de computación se basa en el paso de mensajes petición y respuesta a través de canales con nombres ($URIs$). Los identificadores de los espacios de tripletes no pueden ser intercambiados a través de mensajes petición y respuesta.

Usando este modelo, una posible formalización paramétrica de un proceso semántico *REST* $S_{REST}(\theta, \mu)$, con un espacio de tripletes asociado (θ), identificado por un URI (μ) y que sigue la semántica de la interfaz homogénea *REST*, se muestra en el cuadro 3.4.

$$
\begin{aligned}
R_{REST}(\theta,\mu) \quad ::= \quad & [m,v,p]req(\mu).\, if\ m = get\ ?\ R_{get}(\theta,\mu). \\
& if\ m = post\ ?\ R_{post}(\theta,\mu). \\
& if\ m = put\ ?\ R_{put}(\theta,\mu). \\
& if\ m = delete\ ?\ R_{delete}(\theta,\mu). \\
& \overline{resp(\mu)}[406,0].R_{REST}(\theta,\mu) \\
\\
R_{get}(\theta,\mu) \quad ::= \quad & x ::= rd(\theta,p).\overline{resp(\mu)}[200,x].R_{REST} \\
\\
R_{post}(\theta,\mu) \quad ::= \quad & new\ \nu\ in\ out(\theta,<p,\nu>).!R(\theta,\nu). \\
& \overline{resp(\mu)}[201,<p,\nu>].R_{REST} \\
\\
R_{put}(\theta,\mu) \quad ::= \quad & swap(\theta,p,v).\overline{resp(\mu)}[200,v].R_{REST} \\
\\
R_{delete}(\theta,\mu) \quad ::= \quad & in(\theta,p_{\mu}).\overline{resp(\mu)}[200,0].0
\end{aligned}
$$

Cuadro 3.4: Descripción paramétrica de un recurso *REST* semántico simple.

Según esta descripción, el proceso recurso recibe un mensaje de tipo petición y comprueba el tipo de la operación del mensaje. Si es una petición de tipo *GET* incluyendo un patrón p, utiliza una operación sobre el espacio de tripletes rd para leer sin extraer los tripletes que satisfagan el patrón y enviarlos a través de su *URI* asociado al proceso que realizó la petición a través de un mensaje de tipo respuesta.
Si la petición es de tipo *POST*, incluyendo un patrón p con una única variable, el proceso genera un nuevo *URI* μ y lo aplica al patrón para obtener un nuevo conjunto de tripletes que escribe en su espacio de tripletes asociado θ. Por último un nuevo proceso recurso para los tripletes, espacio de tripletes y *URI* asociada es iniciado por el proceso antes generar el mensaje respuesta al proceso que inició la petición.
Las peticiones *PUT* son definidas a través de una operación *swap* en la que se aplica el patrón y los nuevos valores contenidos en la petición enviada por un proceso cliente a los tripletes almacenados en el espacio de tripletes. Las peticiones *DELETE* por su parte, suponen la finalización de la ejecución del proceso y el borrado del espacio de tripletes asociado de todos los tripletes asociados al *URI* identificador del recurso.
La Figura 3-1 muestra una representación gráfica de un posible modelo de ejecución de computación semántica. Dicho modelo se compone de tres dominios que consti-

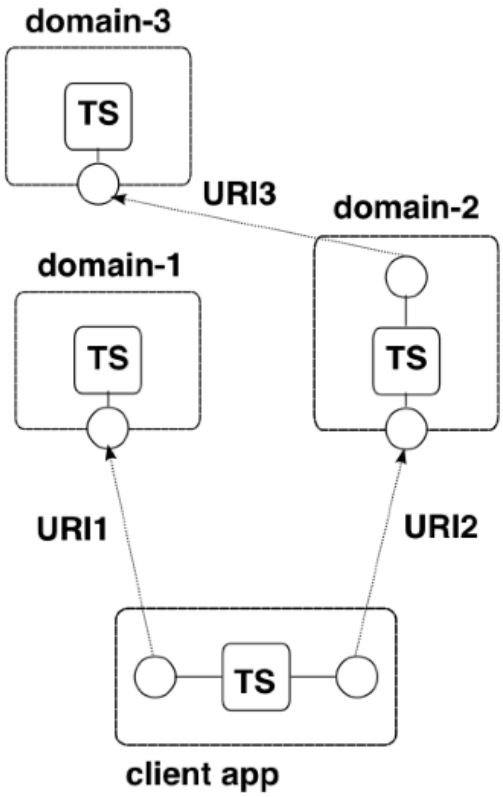


Figura 3-1: Ejemplo de computación *REST* semántica.

tuyen cada uno una localización de computación independiente con su espacio de tripletes.

Cada uno de ellos da soporte a un proceso recurso identificado por los *URIs* 1, 2 y 3. Además, la localización de computación constituida por el *dominio-2* incluye también un proceso agente capaz de comunicarse con el servicio en el *dominio-3* a través del *URI* 3 almacenado en el espacio de tripletes presente en el *dominio-2*. La figura también muestra una localización de computación adicional, que modela una aplicación web conectada con su propio espacio de tripletes y dos procesos agente. Dichos procesos agente pueden enviar mensajes de petición a los servicios en los dominios 1 y 2 a través de las *URIs* 1 y 2 almacenadas en el espacio de tripletes de la aplicación cliente. Por su parte el proceso recurso y el proceso agente localizado en el dominio-2 pueden coordinarse a través de operaciones sobre el espacio de tripletes del

dominio, por ejemplo a través de una operación *notify*, de tal modo que si el proceso recurso recibe un determinado mensaje de petición, esto desencadene una notificación en el cliente para que pueda realizar una operación como reacción, por ejemplo, enviar un mensaje de petición al proceso recurso en el *dominio-3*.

3.2. Modelo arquitectónico

En la sección previa, se ha introducido un modelo formal que puede ser empleado para describir rigurosamente una computación basada en recursos *REST* semánticos. Sin embargo, para transformar este modelo teórico en una implementación concreta de la solución, es necesario contar con un modelo de arquitectura software que sea consistente con las primitivas formales del cálculo descrito, definiendo al mismo tiempo las tecnologías y principales componentes con los que debe contar un sistema software que pretenda transformar dicho modelo formal en código ejecutable.
En esta sección describiremos este modelo arquitectónico, mediante la presentación de una ontología *RDF Schema* (*RDFS*) que describe un modelo de servicios web que pueden dar soporte a las operaciones teóricas del cálculo de recursos *REST* semánticos discutido anteriormente y que sigue, al mismo tiempo, los principios arquitectónicos propuestos por la comunidad de Datos Enlazados Abiertos. Cualquier implementación de la arquitectura de servicios web *REST* semánticos que proponemos debería ser capaz de procesar automáticamente una descripción de una *API* en los términos de la ontología *RDF* que proponemos y trasformarla en una serie de procesos software en tiempo de ejecución que ofrezcan los servicios especificados en dicha descripción.
Los únicos componentes software que se presuponen es algún tipo de adaptador *HTTP* capaz de recibir en una interfaz de red peticiones en este protocolo, y un repositorio de grafos *RDF* que soporte el lenguaje de consultas *SPARQL 1.1 Update*. Esta versión del lenguaje *SPARQL* incluye no solo operaciones para recuperar información desde el grafo *RDF*, sino también para modificar y eliminar tripletes del grafo. El modelo arquitectónico de servicios web *REST* semánticos que vamos a describir, cumple con los siguientes requisitos fundamentales:

- Debe ser una alternativa viable para el desarrollo de *APIs* de servicios web para el tipo de aplicaciones web que se desarrollan en la actualidad. A la hora de introducir nuevas tecnologías o conceptos, debe intentar hacerlo de una manera que resulte lo más familiar posible para los desarrolladores web no familiarizados con las tecnologías estandarizadas por la comunidad Web Semántica y Datos Enlazados Abiertos.

- Debe permitir llevar a cabo una implementación software del cálculo formal de recursos *REST* semántico mostrado en la sección previa de este documento.

- Debe cumplir con los principios arquitecturales *REST* y las recomendaciones de diseño originadas en la comunidad de Datos Enlazados Abiertos.

- Debe dar soporte, no sólo a la recuperación de datos, sino también a la creación, actualización y eliminación de recursos.

- Debe intentar, dentro lo posible, reutilizar el trabajo y los vocabularios ya existentes dentro de la comunidad de Datos Enlazados Abiertos.

Dentro de las posible alternativas para la construcción de una *API* de servicios web para el acceso a grafos *RDF* que mencionamos en apartados previos de este documento, la *API* que vamos a describir se basa en el intercambio de grafos *RDF* a través del protocolo *HTTP*, tal y como establece la propuesta de recomendación de la *W3C* para un Protocolo *SPARQL HTTP* Uniforme para la Gestión de Grafos *RDF* [97]. Usando este protocolo, los recursos de información en el servicio pueden ser descritos como tripletes almacenados en un *grafo con nombre*. Esta concepción de los recursos expuestos en la *API* también coincide con el concepto de recurso *REST* semántico descrito en el cálculo formal de recursos *REST* semánticos descrito anteriormente en este documento, donde los tripletes de cada *grafo con nombre* serían equivalentes al espacio de tripletes que respalda al proceso del cálculo que ofrece la funcionalidad *REST* básica y la *URI* del *grafo con nombre* haría las veces del canal con nombre por el que los procesos agentes tendrían acceso a los tripletes del recurso. El uso de grafos con nombre como base para la definición de recurso en la *API* ofrece un nivel

de granularidad similar al de otras *APIs REST* [132].
De forma adicional, introduciremos algunas características convenientes que normalmente pueden encontrarse en *APIs* basadas en entidades atributo-valor, como la propuesta LD-API [108] con el fin de facilitar el uso de la *API* a clientes web con restricciones, como navegadores web.

3.2.1. Declaración de recursos enlazados

En el cuadro 3.1 se encuentra la descripción *RDF* de un ejemplo de servicio web *REST* semántico, que se almacenará en una base de datos relacional, de acuerdo con el modelo de servicio web que proponemos.

```
@prefix testblog: <http://example.org/blog#> .
@prefix lda: <http://restful_linked_data_api.org#> .
@prefix api: <http://purl.org/linked-data/api/vocab#> .
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix sioct: <http://rdfs.org/sioc/types#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

testblog:
  a lda:API ;
  lda:exposes testblog:blogs, testblog:blog .

testblog:blogs
  a lda:Resource ;
  api:uriTemplate "http://testblog.org/lodapi/blogs" ;
  lda:endpoint
  [ a lda:R2RMLSparqlEndpoint ;
    lda:has_r2rml_mapping testblog:bogsMapping ;
    lda:has_r2rml_graph rr:columnGraphIRI ];
```

```
  lda:has_operation lda:GET, lda:POST ;
  lda:named_graph_creation_mechanism testblog:blogsMappingUriMinter .

testblog:blogsMappingUriMinter
  a lda:NamedGraphCreationMechanism ;
  lda:uri_template "http://testblog.org/lodapi/blogs/{id}" ;
  lda:mapped_uri_parts
  [ lda:mapped_component_value "id" ;
    lda:uri_generator lda:UniqueIdInt ] .

testblog:blog
  a lda:Resource ;
  api:uriTemplate "http://testblog.org/lodapi/blogs/{id}" ;
  lda:endpoint
  [ a lda:R2RMLSparqlEndpoint ;
    lda:hasR2RMLMapping testblog:blogsMapping ;
    lda:hasR2RMLGraph rr:columnGraph ];
  lda:has_operation lda:GET, lda:PUT, lda:DELETE .

testblog:blogsMapping
  a rr:TriplesMap ;
  rr:logicalTable "blogs" ;
  rr:class sioct:Weblog ;
  rr:subjectMap [ a rr:IRIMap;
                  rr:column "id" ];
  rr:propertyObjectMap [ rr:property dc:creator;
                         rr:column "author" ];
  rr:propertyObjectMap [ rr:property dc:title;
                         rr:column "title" ];
  rr:propertyObjectMap [ rr:property dcterms:created;
                         rr:column "created_at" ;
```

```
        rr:datatype xsd:dateTime ].
```

Listing 3.1: Descripción *RDF* de un servicio *REST* semántico.

Esta descripción describe dos aspectos básicos del modelo de servicio:

- Cómo se expone un conjunto de tripletes *RDF* como un recurso *REST* a través de la interfaz uniforme *HTTP*.
- Cómo se transforman las peticiones *HTTP* en consultas *SPARQL* que cumplen con la semántica que los principios arquitectónicos *REST* asignan a las operaciones *GET*, *POST*, *PUT* y *DELETE* del protocolo *HTTP*.

Distintos elementos del vocabulario propuesto permite especificar los componentes de estos dos aspectos básicos del modelo de servicio.
En el caso de la asociación entre el servicio y la interfaz *HTTP*, el vocabulario ofrece las siguientes propiedades:

- ***uriTemplate*:** Un patrón para un *URI* que designa a un conjunto de grafos con nombre que podrán ser accedidos mediante operaciones *HTTP*. Esta propiedad de la ontología se ha reutilizado de la especificación de LD-API, así como la semántica del proceso de validación de un *URI* con un patrón *URI* como el identifica esta propiedad.
- ***has_operation*:** Asocia al recurso una collección de operaciones *HTTP* que serán válidas a la hora de acceder a los datos almacenados en el *grafo con nombre* expuesto como un recurso. *lda:GET*, *lda:PUT*, *lda:POST* y *lda:DELETE* son posible operaciones válidas para esta propiedad. Estas operaciones deberán ser interpretadas por cualquier implementación de la arquitectura de acuerdo con lo acordado en la propuesta de recomendación sobre el Protocolo *SPARQL HTTP* Uniforme para la Gestión de Grafos *RDF*. En el siguiente apartado de este documento definiremos la semántica para cada una de las operaciones anteriores como determinadas consultas *SPARQL* 1.1/Update, compatibles con la semántica de la interfaz uniforme *HTTP*, tal y como se estableció en el cálculo

formal para recursos *REST* semánticos descrito en el capítulo anterior de este documento.

El segundo aspecto del modelo servicio, la forma en que los datos semánticos del recurso son almacenados en un repositorio de grafos *RDF*/*SPARQL*, viene dado por las siguientes propiedades:

- ***endpoint*:** El repositorio *RDF* en el que se almacenarán los conjuntos de tripletes *RDF* que dan soporte a la información de los diferentes recursos expuestos por la *API*. Este repositorio debe ser capaz de soportar el conjunto de especificaciones *SPARQL 1.1 Update*.

- ***named_graph_creation_mechanism*:** Esta propiedad describe como nuevos grafos con nombre serán generados cuando nuevos recursos necesiten ser creados. El mecanismo debe especificar un patrón de *URI* para el nuevo *grafo con nombre* así como las partes del patrón que serán generados. Dos mecanismos están definidos en nuestro modelo de servicio: *lda:UniqueIdInt* y *lda:UUID*. El primero de ellos genera un nuevo entero único mientras que el segundo genera un nuevo identificador universal único *UUID*.

3.2.2. Modelo de Procesamiento del Servicio

Las implementaciones software del modelo aquí propuesto deben aceptar peticiones *HTTP* y procesarlas en un esquema de tres etapas como el siguiente:

- Traducción de la petición *HTTP* a una petición *SPARQL*.

- Generación de nuevos identificadores para grafos con nombre, si es necesario, y ejecución de la consulta *SPARQL* en el repositorio de grafos *RDF*.

- Serialización del grafo *RDF* resultante como en la representación acordada entre cliente y servidor mediante el proceso de negociación de contenido descrito en el protocolo *HTTP*.

```
CONSTRUCT { ?s ?p ?o }
WHERE
{
  GRAPH <graph_uri> { ?s ?p ?o }
}
```

Cuadro 3.5: Consulta *SPARQL* para una peticion *HTTP GET*.

En las siguientes secciones analizaremos como se puede transformar cada tipo de operación *HTTP* en consultas *SPARQL* sobre un repositorio de grafos *RDF* de manera que se conserve la semántica del protocolo *HTTP*.

Peticiones *GET*

Las peticiones *HTTP GET* sobre un determinado *URI* que identifica a un *grafo con nombre* en el repositorio de grafos *RDF*, se transforman en peticiones *CONSTRUCT SPARQL* que recuperan todos los tripletes almacenados en dicho *grafo con nombre*.
La sintaxis de la consulta *CONSTRUCT SPARQL* se muestra en la tabla 3.5.

La semántica de esta consulta *SPARQL* coincide con la semántica descrita en la propuesta de estándar *Protocolo SPARQL HTTP Uniforme para la Gestión de Grafos RDF* de la *W3C*.
Adicionalmente, las peticiones *GET HTTP* pueden especificar dos parámetros *_pageSize* y textit_page para modificar el número de tripletes en el grafo de destino que serán recuperados por la consulta *SPARQL* mediante el uso de las cláusulas *LIMIT* y *OFFSET* del lenguaje *SPARQL* al mismo tiempo que se ordena el conjunto de tripletes recuperados por sujeto, mediante el uso de la claúsula *ORDER BY ?s*. Estos parámetros han sido propuestos en la propuesta *LD-API* y permiten implementar paginación en el cliente *HTTP* realizando la petición.

```
INSERT DATA
{
  GRAPH <graph_uri> {.. RDF payload .. }
}
```

Cuadro 3.6: Consulta *SPARQL* para una petición *HTTP POST*.

Peticiones *POST*

Las peticiones *HTTP POST* para la creación de un nuevo recurso, son traducidas como peticiones *INSERT DATA* del lenguaje *SPARQL UPDATE* que insertarán los datos enviados en el cuerpo de la petición *HTTP* en un nuevo *grafo con nombre* cuya *URI* deberá generar el sistema, de acuerdo con el mecanismo de generación de identificadores elegido en la descripción de la *API*.
La sintaxis de la consulta *SPARQL* UPDATE se muestra en el cuadro 3.6.

Una consideración importante es la distinción que se debe hacer entre el *URI* generado para identificar el *grafo con nombre* que se almacenará en el repositorio de grafos *RDF* y el *URI* que identifica el recurso *RDF* en los tripletes almacenados en el *grafo con nombre* de acuerdo con la práctica estándar a la que ha llegado la comunidad de Datos Enlazados Abiertos durante la discusión de la relación entre recursos de información y no recursos de información [39].
La siguiente tabla 3.2.2 muestra las diferencias entre ambas *URIs*.

Grafo con nombre	***Recurso RDF***
Identificado por un *URI*.	Identificado por un *URI*.
Recurso de información.	No recurso de información.
Identifica el *grafo con nombre*.	Aparece dentro del *grafo con nombre*.
Desreferenciable usando *HTTP*	No desreferenciable usando *HTTP*

Cuadro 3.7: *Grafos con nombre* frente a recursos *RDF*.

Sin embargo, ambas *URIs* están relacionadas, debiendo ser posible para un cliente web que se encuentre con la *URI* del recurso formando parte de un triplete *RDF* en otro grafo, seguir un enlace que desencadene una petición *HTTP* sobre la *URI* del

grafo con nombre que almacena los datos del recurso.
Además, según los principios arquitectónicos *REST*, la responsabilidad de generar la *URI* del recurso *HTTP*, lo que en nuestro caso significa generar un *URI* para el *grafo con nombre* y para el recurso *RDF* almacenado en dicho grafo, depende del proveedor de la *API*, no del cliente, por lo que la *URI* del recurso *RDF* no puede venir dada en los tripletes *RDF* enviados por el cliente en la petición *HTTP*.
Nuestra solución a nuestro problema viene dada por un mecanismo que permite al cliente especificar un patrón de tripletes *RDF* que se transformará en el grafo *RDF* final que se almacenará en el repositorio *RDF*, una vez que el servicio ha generado *URIs* para el *grafo con nombre* y el recurso *RDF* mediante el mecanismo especificado en la descripción de la *API*.
El proceso es el siguiente:

- El cliente envía una petición *HTTP POST* a la *URI* base de la colección de recursos, incluyendo un grafo *RDF* en el cuerpo de la petición, donde la *URI* del recurso *RDF* descrito en el grafo ha sido sustituido por un identificador de nodo anónimo *RDF*. Adicionalmente, un parámetro *_self* se enviará en la *URI* de la petición con el identificador del grafo anónimo usado para designar al recurso *RDF* en el cuerpo de la petición.

- Si el grafo *RDF* enviado en el cuerpo de la petición incluye algún triplete cuyo sujeto no es un identificador de nodo *RDF* anónimo, el servicio debe devolver una respuesta de error *HTTP 403 forbidden*.

- El servicio genera una nueva *URI graph_uri* para identificar al *grafo con nombre* de acuerdo con el mecanismo especificado en la descripción de la *API*.

- El servicio reemplaza el nodo anónimo especificada por el cliente en el patrón de tripletes *RDF* para identificar el recurso *RDF* por el *hash URI* [39] *graph_uri#self*.

- El *grafo con nombre* resultante, identificado por el *URI graph_uri*, incluyendo la descripción del recurso *RDF* identificado por el *URI graph_uri#self* es almace-

nado en el repositorio *RDF* mediante la consulta *SPARQL Update* especificada en el cuadro 3.6.

- Si la cración del *grafo con nombre* resulta exitosa, el servicio debe devolver una respuesta *HTTP 201 created* al cliente con una cabecera *HTTP location* con la *URI* del nuevo *grafo con nombre* creado.

El uso de un *hash URI* para identificar al recurso *RDF*, que usa como base el *URI* utilizado para identificar al recurso *HTTP* y al *grafo con nombre* asociado, permite que clientes web que encuentran el *URI* del recurso web enlazado en otro grafo *RDF*, descubran el contenido del recurso *RDF* sin desreferenciarlo directamente. Los *hash URIs* no pueden ser directamente desereferenciadas. Cualquier intento de hacer una petición *HTTP GET* a un *hash URI* será interpretada por el servicio web como una petición a la *URI* base del *hash URI*, en nuestro caso la *URI* del *grafo con nombre*, iniciándose de esta manera el proceso de negociación de contenido *HTTP* que finalmente llevará al cliente a obtener una serialización concreta de la información *RDF* del recurso asociada al *grafo con nombre*.

Peticiones *PUT*

Las peticiones *HTTP PUT* son interpretadas por el servicio como un intento de reemplazar el contenido del recurso asociado por un nuevo contenido, en nuestro caso, un nuevo grafo *RDF* que se almacenará en el mismo *grafo con nombre* asociado al recurso *HTTP*.
Esta semántica queda recogida en la combinación de consultas *SPARQL Update DROP SILENT* y *INSERT DATA* sobre el repositorio de grafos *RDF* en las que se traduce la petición *HTTP*. La tabla 3.8 muestra la sintaxis de dicha consulta.

El cuerpo de la petición *PUT* debe contener sólo tripletes que usan como sujeto el identificador del recurso *RDF graph_uri#self* o identificadores de nodos anónimos *RDF*. El servicio debe cancelar el procesamiento de la petición y devolver un código de error *HTTP 403 forbidden* si esta condición no se cumple.

```
DROP SILENT GRAPH <graph_uri>;
INSERT DATA {
  GRAPH <graph_uri> { .. RDF payload .. }
}.
```

Cuadro 3.8: Consulta *SPARQL* para una petición *HTTP PUT*.

Peticiones *DELETE*

Por último, las peticiones *HTTP DELETE* se traducen como peticiones *DROP GRAPH SPARQL*, que suponen la eliminación del *grafo con nombre* y toda la información asociada a dicho grafo. El cuadro 3.9 muestra la sintaxis de la petición.

```
DROP GRAPH <graph_uri>.
```

Cuadro 3.9: Consulta *SPARQL* para una petición *HTTP DELETE*.

El procesamiento de la petición *HTTP DELETE* supone la destrucción del recurso *HTTP* identificado por el *URI* del *grafo con nombre*. Cualquier petición adicional a la *URI* del recurso *HTTP* debe ser resuelta por el servicio con una respuesta *HTTP 404 not found*.

3.2.3. Serialización de resultados

Tras obtener una respuesta exitosa desde el repositorio de grafos *RDF* mediante una consulta *SPARQL* la implementación de la *API* debe devolver la información recuperada al cliente mediante la representación acordada entre cliente y servidor mediante el proceso de negociación de contenido especificado en el protocolo *HTTP*. Los tipos de contenido soportados por la especificación LD-API deben ser también soportados como tipos válidos por implementaciones de esta *API*.
Un tipo de datos de especial interés es el tipo de datos *JSON*. *JSON* es el formato de datos elegido por la mayoría de *APIs* de datos en uso hoy en día. Con el fin de asegurar que la *API* va a resultar útil para desarrolladores web no familiarizados con la pila de tecnologías semánticas, que simplemente están recuperando información a

través de la *API*, es importante ofrecer una representación de los grafos *RDF* como objetos *JSON* que pueda ser utilizada con facilidad por parte de estos desarrolladores. *JSON-LD* [120] es una representación que cumple estos criterios, gracias especialmente a una característica denominada coerción de tipos que permite codificar los grafos *RDF* como objetos *JSON* sencillos, con una propiedad adicional en la que se incluye la traducción entre propiedades simples del objeto *JSON* y los *URIs* de las propiedades *RDF* y tipos de los literales. Aquellos desarrolladores quen consuman la *API* y no estén interesados en estas características semánticas de los resultados obtenidos, pueden ignorar esta propiedad adicional y tratar los resultados obtenidos como objetos *JSON* sencillos. *JSON-LD* es la codificación elegida para la implementación de nuestro modelo de servicio para cualquier petición *HTTP* seleccionando *JSON* como el tipo de representación elegido para el recurso que se quiere recuperar, o enviando información codificada como *JSON*.
Por último, la *API* también soporta un par de parámetros adicionales en las peticiones que permiten modificar la representación del recurso solicitado fuera del mecanismo de negociación de contenido *HTTP*:

- *_callback*: Parámetro que puede ser pasado en la petición y que forzará al servicio a devolver una representación *JSON-LD* del grafo *RDF* asociado al recurso *HTTP*, como el único argumento de una invocación a una función *JavaScript* cuyo nombre se pasa como valor del parámetro. Esta técnica, conocida como *JSONP* [98], permite a aplicaciones cliente *JavaScript* que han sido iniciadas dentro de un navegador web, sobrepasar la limitación impuesta por la política de seguridad de dominio único activa en todos los navegadores web.

- *_format*: Especifica un tipo de representación, tomando precedencia sobre el valor especificado en la cabecera accept de la petición *HTTP*.

La presencia de estos parámetros adicionales que imponen una semántica adicional fuera de lo especificado por el protocolo *HTTP* se justifica por la existencia de clientes

web con un control limitado sobre el uso del protocolo, como aplicaciones *JavaScript* siendo ejecutadas dentro de un navegador web.

3.3. Components Software

En las dos secciones anteriores de este capítulo hemos descrito un modelo formal para la descripción de la computación basada en servicios *REST* semánticos y un modelo arquitectónico y de servicio que permite transformar el modelo formal en una arquitectura web en el que un conjunto de peticiones *HTTP* recibidas desde una interfaz de red se transforman en peticiones *SPARQL* sobre un repositorio de grafos *RDF*.

Sin embargo, el modelo arquitectónico descrito no establece la naturaleza de los componentes software concretos que se deben emplear para implementar dicha arquitectura. En particular no se especifica ningún detalle sobre el repositorio de grafos *RDF*, más allá de que debe soportar consultas *SPARQL 1.1 Update*. Tampoco se ha mencionado cómo un cliente web puede gestionar la información *RDF* recuperada por el servicio para llevar a cabo una computación como la descrita en nuestro cálculo formal.

Siguiendo el espíritu de la comunidad de Datos Enlazados Abiertos, en esta sección examinaremos la implementación que hemos llevado a cabo de dos componentes software que ofrecen alternativas para implementar esos componentes de la arquitectura propuesta usando las tecnologías más comunes hoy en día en el desarrollo web: los sistemas gestores de bases de datos relaciones como repositiorios de grafos *RDF* y aplicaciones *JavaScript* ejecutándose dentro de un navegador web como clientes.

3.3.1. Ejecución de consultas *SPARQL 1.1 UPDATE* sobre datos relacionales.

Como vimos en el capítulo dedicado al estado del arte en este documento, una de las contribuciones de la comunidad de Datos Enlazados Abiertos ha sido el desarrollo

```
<R2RMLMapping> ::= { <TableMapping> } ;

<TableMapping> ::= ( table:String,
                     <subject:TermMapping>,
                     <graph:TermMapping>,
                     <propertyObj:{ TripleMapping }>) ;

<TripleMapping>  ::= { (<property:TermMapping>,
                        <column:TermMapping>,
                        [rr:datatype],
                        [rr:laguage]) } ;

<TermMapping> ::= <VariableMapper> | <ConstantMapper> ;

<ConstantMapper> ::=  rr:property | rr:constantValue
                    | rr:columnGraphIRI ;

<VariableMapper> ::=  rr:propertyColumn | rr:column
                    | rr:columnGraph ;
```

Cuadro 3.10: Sintaxis *EBNF* de *R2RML*.

de la propuesta de recomendación *W3C R2RML* [32] para la traducción de datos relaciones en tripletes *RDF*.

R2RML consiste en un vocabulario genérico que puede usarse para transformar un esquema relacional en tripletes *RDF*. Una descripción formal simplificada de la sintaxis de *R2RML* se muestra en el cuadro 3.10 como una gramática *EBNF*.

Este modelo formaliza *R2RML* como una traducción de una colección de *TableMappings* por cada tabla del esquema relacional que se intenta transformar en tripletes *RDF*. Cada *TableMapping* describe como los datos almacenados en la tabla se deben transformar en *quads RDF*, tripletes aumentados consistentes en sujeto, predica, objeto y un *grafo con nombre*. Los componentes de un *quad* son generados usando un *TermMapping* definido en el *TableMapping*.

Cada *TermMapping* puede ser constante o variable. Los *TermMappings* constantes asignan un *URI* o literal *RDF* para el valor de ese componente en el quad *RDF* que se va a generar. Por su parte los *TermMappings* variables pueden hacer referencia a una columna en el esquema relacional de donde el valor para el componente del *quad* será extraído. Por su parte, el estándar *SPARQL 1.1 Update* describe un lenguaje común para la recuperación y modificación de grafos *RDF* con una sintaxis similar a la del lenguaje de consulta relacional *SQL*. La principal unidad usada para construir consultas *SPARQL* son los *QuadPatterns* que pueden ser aplicados sobre los tripletes

```
<QuadPattern> ::= (<subject:Term>,
                   <property:Term>,
                   <object:Term>,
                   <graph:Term>) ;

<Term> ::= <VariableTerm> | <ConstantTerm>;

<ConstantTerm> ::= URI | RDF Literal ;

<VariableTerm> ::= {?a, $a, ?b, $b...} ;
```

Cuadro 3.11: Sintaxis de los *QuadPatterns*.

almacenados en un grafo *RDF*. El cuadro 3.11 formaliza la noción de un *QuadPattern* con sus posible componentes, variables y constantes.

R2RML describe una transformación desde el modelo relacional al modelo de datos de *RDF*. Con el fin de usar un documento *R2RML* para construir una transformación genérica de consultas *SPARQL 1.1 Update* sobre un grafo *RDF* en consultas *SQL* sobre un esquema relacional, es necesario describir una transformación inversa a la especificada por dicho documento *R2RML*. La figura 3-2 muestra el proceso propuesto para llevar a cabo esa transformación, así como el papel que el documento *R2RML* juega en dicho proceso.

El algoritmo descrito en el cuadro 3.12, especifica en la función *buildQuadMatchers* cómo llevar a cabo esa transformación inversa, generando un conjunto de *QuadMatchers* para un conjunto de *TripleMappers* de una descripción *R2RML* tomada como datos de entrada.

El listado del algoritmo mostrado en el cuadro 3.13 describe a su vez un procedimiento para comprobar si un *QuadMatcher* es compatible con un *QuadPattern* en una consulta *SPARQL*.

En las siguientes secciones describiremos algoritmos que permiten usar la compatibilidad entre *QuadMatchers* y *QuadPatterns* para ejecutar consultas *SPARQL SE-*

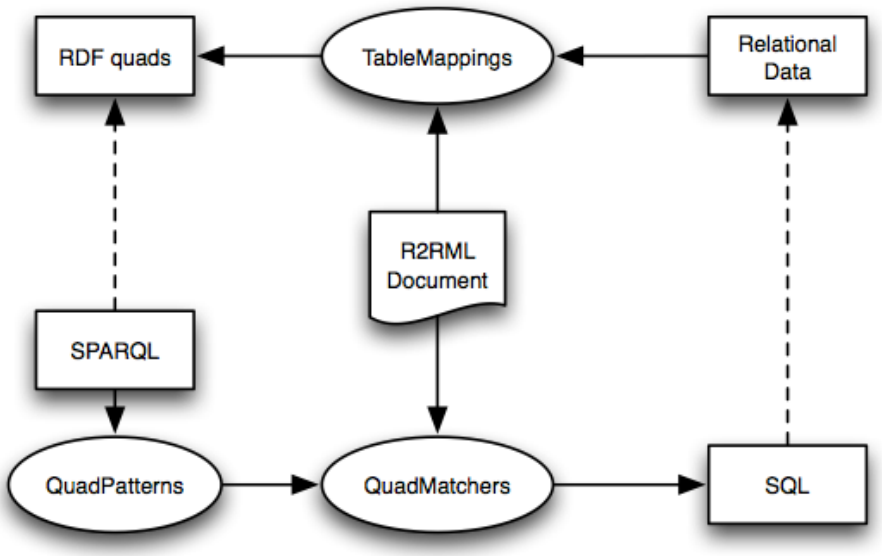


Figura 3-2: Diferentes transformaciones codificadas en un documento *R2RML*

```
Function: buildQuadMatchers
Input: mapping : R2RMLMapping
Output: Collection of QuadMatchers
Begin
   quadMatchers ← {}
   for tableMapping in mapping do
      table ← tableMapping.table
      subjetTerm ← tableMapping.subject
      for tripleMapping in tableMapping.tripleMapping do
         property ← tripleMapping.property
         object ← tripleMapping.object
         graph ← tripleMapping.graph
         quadMatchers ∪ (table, graph, subject, property, object)
      end for
   end for
   return quadMatchers
```

Cuadro 3.12: Algoritmo 1: Construcción de *QuadMatcher* para una tranformación *R2RML*.

```
Function: compatible?
Input: p : quadPattern, m : QuadMatcher
Output: True or False
Begin
   compatible ← True
   for patternComp in p, matcherComp in m do
      if var?(patternComp) ∨ var?(matcherComp) then
         compatible ∧ True
      else
         compatible ∧ (patternComp = matcherComp)
      end if
   end for
   return compatible
```

Cuadro 3.13: Algoritmo 2: Procedimiento para comprobar si un *QuadPattern* y un *QuadMatcher* son compatibles.

LECT, *INSERT DATA* y *DELETE* sobre datos relacionales con una transformación *R2RML* asociada. Estas operaciones son suficientes para implementar las consultas sobre el repositorio de grafos *RDF* descritas en la sección anterior de este capítulo como transformaciones de las operaciones *HTTP* sobre recursos *REST* semánticos en el modelo arquitectónico de servicios propuesto. Las consultas *DROP GRAPH* usadas en dicho modelo pueden ser transformadas en operaciones *DELETE* equivalentes donde todos los tripletes del grafo son eliminados.

Consultas *SPARQL SELECT*

La transformación de consultas *SPARQL SELECT* que proponemos se basa en el trabajo de [25], pero modificado para adaptarlo a la sintaxis y semántica de las transformaciones *R2RML*, donde algunas de las restricciones que impone [25] son demasiado restrictivas. Un jemplo de estas restricciones es el hecho de que la tabla de donde se extraen los componentes de un quad *RDF* debe ser única, supuesto que no tiene por que ser válido en una transformación *R2RML*.
En el algoritmo del cuadro 3.14 se muestra un procedimiento para transformar una consulta *SPARQL SELECT* en un conjunto de *QuadPatterns* que pueden usarase como la base de una consulta *SQL SELECT* equivalente. Esto se lleva a cabo mediante la búsqueda de los *QuadMatchers* compatibles con los *QuadPatterns* que luego son unidos mediante una cláusula *UNION* de *SQL*.

Consultas *SPARQL INSERT DATA*

El el cuadro 3.15 contiene un algoritmo, que muestra un procedimiento para la inserción de un *QuadPattern* con todos sus términos constantes en un esquema relacional con una transformación *R2RML* asociada.

El principal problema que se debe solucionar para llevar a cabo la inserción es que es posible encontrar más de un único *QuadMatcher* compatible con el componente del QuadPattern donde llevar a cabo la inserción.
Como muestra la figura 3-3, el algoritmo propuesto funciona construyendo un árbol

```
Function: select
Input: quad : QuadPattern, matchers : QuadMatcher
Output: SQL query or FAIL
Begin
   compatibleMatchers ← mapCompatibleQuadMatchers(quad, matchers)
   query ← FAIL
   if compatibleMatchers ≠ ∅ then
      subselects ← mapSubselects(quad, compatibleMatchers)
      query ← join("UNION", subselects)
   end if
   return query

Function: mapSubselects
Input: quad : QuadPattern, matchers : QuadMatcher
Output: SQL subquery
Begin
   subselects ← {}
   for matcher in matchers do
      table ← matcher.table
      projections ← genProjections(quad, matcher)
      conditions ← genConditions(quad, matcher)
      sql = "SELECT DISTINCT" + join(",", projections) + "FROM" +
      table
      if conditions ≠ ∅ then
         sql + "WHERE" + join(",", conditions)
      end if
      subselects ∪ sql
   end for
   return subselects
```

Cuadro 3.14: Algoritmo 3: Composición de una consulta *SELECT* para un *QuadPattern* y un conjunto de *QuadMatchers*.

```
Function: insert
Input: quads : QuadPattern, matchers : QuadMatcher
Output: SQL DML query or FAIL
Begin
   sortedQuads ← sortBySubject(quads)
   contexts ← initialContext(quads)
   for quad in sortedQuads do
      compMatchers ← mapCompatibleQuadMatchers(quad, matchers)
      if compMatchers = ∅ then
         return FAIL
      end if
      contexts′ ← nextLevel(quad, compMatchers, contexts)
      contexts ← minSchemaContexts(contexts′)
   end for
   sql ← generateInsertionSQL(first(contexts))
   return sql
```

Cuadro 3.15: Algoritmo 4: Inserción de un *QuadPattern* para un conjunto de *QuadMatchers*.

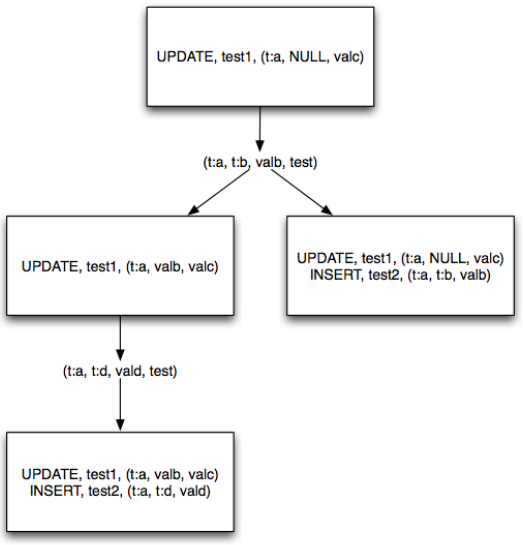


Figura 3-3: Inserción de dos *quads*

Function: insertionCost
Input: $context : SchemaUpdateContext$
Output: $cost : integer$
Begin
  $columns \leftarrow 0$
  for $rowMatch$ in $context$ **do**
    $columns \leftarrow columns + count(rowMatch.columns)$
  end for
  return $((1 + count(context.rows)) * columns)$

Cuadro 3.16: Algoritmo 5: Métrica de coste.

con las posibles opciones de inserción encontradas y después selecciona aquella potencial inserción que minimiza una métrica de coste descrita en el algoritmo del cuadro 3.16.

Consultas *SPARQL DELETE WHERE*

Para traducir consultas *DELETE SPARQL* en consultas *SQL* equivalentes dada una transformación *R2RML* de un esquema relacional, el primer paso consiste en transformar la cláusula *WHERE* de la consulta *SPARQL* en una consulta *SELECT SQL* y una vez ejecutada usando el algoritmo previamente descrito 3.14, usar los resultados obtenidos para aplicarlos sobre el patrón de la cláusula *SELECT* para obtener el conjunto final de quads que deben ser eliminados del grafo *RDF*.

```
Function: delete
Input: quads : {QuadPattern}, matchers : {QuadMatcher}
Output: SQL DELETE DML query
Begin
   sql ← ""
   for quad in quads do
      compatibleMatchers ← mapCompatibleQuadMatchers(quad, matchers)
      for matcher in compatibleMatchers do
         columnMatches ← genColumnMatches(quad, matcher)
         if count(columnMatches) > 0 then
            table ← matcher.table
            sql ← sql + "UPDATE" + nameSQL(table) + "SET"
            conds = {}
            values = {}
            for columnMatch in columnMatches do
               if isNotSubject?(columnMatch) then
                  values ∪ nameSQL(columnMatch.column) + " = NULL"
               end if
               conds ∪ nameSQL(columnMatch.column) + " = "
                  +nameSQL(columnMatch.value)
            end for
            sql ∪ join(values, ",") + "WHERE" + join(conds, "AND") + ";"
         end if
      end for
   end for
   return sql
```

Cuadro 3.17: Algoritmo 6: Composición de una consulta para eliminar un *QuadPattern* para un conjunto de *QuadMatchers*.

El algoritmo del cuadro 3.17 describe un procedimiento para eliminar los quads almacenados en un esquema relacional con una transformación *R2RML* asociada. El algoritmo actualiza las columnas de las tablas asociadas por la transformación *R2RML* a las propiedades *RDF* y objetos con valores nulos, en vez de eliminar todo el quad, ya que la columna sujeto puede ser compartida por otros tripletes almacenados en la misma fila.

Tras eliminar todos los quads, la función removeEmptyRows elimina todas aquellas filas en las tablas de la transformación *R2RML* donde todos los valores para las columnas propiedad y objeto tienen valores nulos.

3.3.2. Un repositorio *RDF SPARQL 1.1 Update* para aplicaciones *JavaScript*

En secciones anteriores de este capítulo hemos expuesto cómo, desde el punto de vista formal, una computación basada en recursos *REST* semánticos involucra tanto a procesos recurso, que exponen datos semánticos a través de canales identificados con *URIs* como a procesos agente, que aunque no pueden ser accedidos directamente a través de *URIs*, son capaces de consumir los datos semánticos expuestos por los

servicios y hacer avanzar la computación basándose en los datos recuperados.
En la descripción de la arquitectura que puede usarse para implementar el modelo teórico propuesto hemos hecho hincapié en la parte servidor de la computación, sin entrar en los requisitos arquitecturales necesarios para implementar un proceso agente capaz de consumir y tomar decisiones basadas en la información semántica recuperada.
En principio, los únicos requisitos necesarios para implementar un proceso agente vienen dadas por los siguientes elementos básicos:

- Soporte para el lado cliente del protocolo *HTTP*.
- Un mecanismo para almacenar grafos *RDF* y ejecutar consultas *SPARQL* sobre ellos.
- Capacidad para reaccionar ante modificaciones en el grafo *RDF* recuperado.

El soporte para el almacenamiento de grafos *RDF* puede venir dado por el uso de un repositorio de grafos *RDF* o un mecanismo que transforme un sistema gestor de bases de datos relacional en un repositorio *RDF* con soporte para el lenguaje de consultas *SPARQL*, como el que hemos expuesto anteriormente. Sin embargo, ningún tipo de repositorio *RDF* de uso común está disponible para el que quizás sea el más común de los entornos posibles para la ejecución de un proceso agente en una computación basada en recursos *REST* semánticos: el navegador web.
En esta sección describiremos nuestra implementación de un repositorio *RDF* con soporte para el lenguaje de consultas *SPARQL 1.1 Update* así como para notificaciones asíncronas asociadas a modificaciones en el grafo *RDF* almacenado, que puede usarse como un componente de aplicaciones *JavaScript* que son ejecutadas dentro del navegador web en general y en particular para aplicaciones clientes consumiendo la *API* descrita en la sección dedicada a la descripción de una arquitectura de servicios web expuesta en este documento.
El repositorio está contenido en una biblioteca *JavaScript* que puede usarse como la base de la capa de datos en aplicaciones *JavaScript* complejas ejecutándose en el navegador. La biblioteca soporta no sólo el almacenamiento de grafos *RDF* y la consulta

de estos datos usando el lenguaje *SPARQL 1.1 Update*, también incluye funcionalidades para la deserialización de datos *RDF* codificados usando formatos *JSON*-LD, *Turtle* y *N3*.

La biblioteca se ha diseñado para tomar ventaja de las características modernas que incluyen las últimas versiones de los navegadores web, si se encuentran disponibles, como la ejecución multi-hilo mediante el soporte para *WebWorkers* [55] y el almacenamiento de datos persistentes usando la funcionalidad de Web Storage [57].

Otra interesante capacidad de la biblioteca consiste en la posibilidad de registrar funciones asociadas a eventos sobre el grafo, que serán invocadas por el repositorio cuando una modificación en el grafo *RDF* satisfaga las condiciones asociadas al evento.

Por último, la implementación *JavaScript* del repositorio permite su ejecución no sólo dentro del navegador web, sino también en el lado servidor mediante una plataforma para la ejecución de aplicaciones *JavaScript* conocida como *Node.JS* que ha ganado una gran popularidad recientemente entre los desarrolladores de aplicaciones web. La figura 3-4 muestra los principales componentes de la biblioteca que implementa el repositorio *RDF*.

Almacenamiento persistente y no persistente

El almacenamiento de datos en memoria no persistente de la biblioteca se basa en un modelo conceptual en el que todo los quads *RDF* se almacenan en una única tabla y son indexados usando un conjunto de estructuras de datos *b-tree* que aumentan la eficiencia de las consultas de recuperación de datos.

Estos índices están formados por el mínimo número de índices necesarios para cubrir los diferentes patrones presentes en las consultas *SPARQL* formadas por patrones básicos de grafo (*BGP*) [50].

En el almacenamiento persistente, hemos hecho uso de la *API* de Web Storage [57]. Esta *API* define un objeto *JavaScript localStorage* que recubre un mapa persistente donde se pueden almacenar pares clave-valor como objetos *String JavaScript*. La capacidad de almacenamiento de este mapa es limitado, siendo 15 MB el límite por

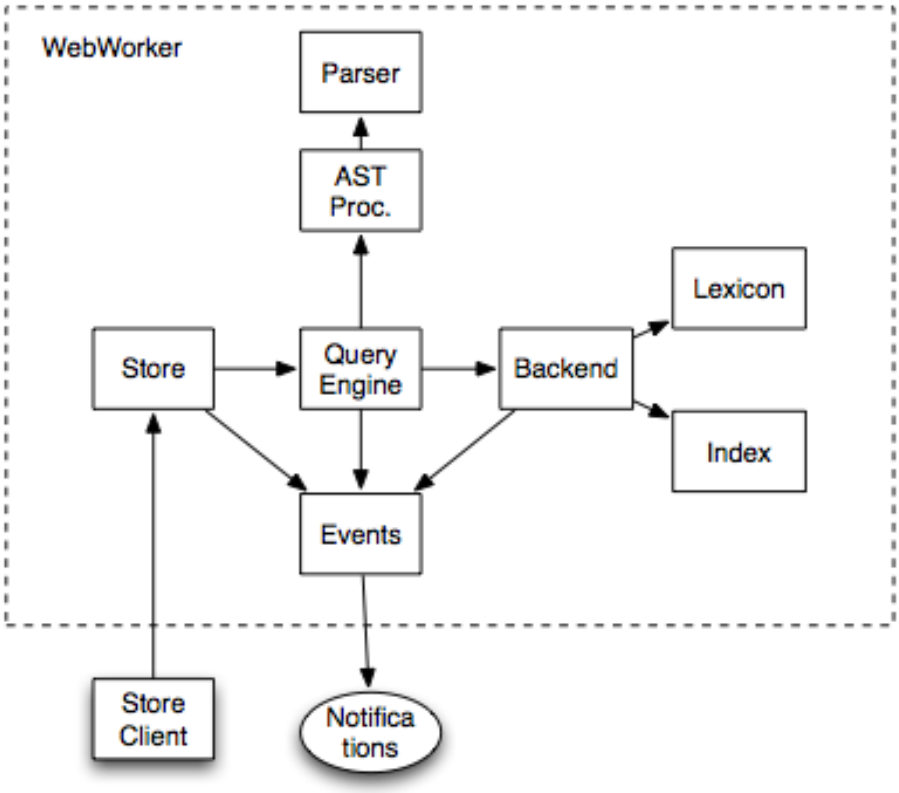


Figura 3-4: Principales componentes del repositorio *RDF*.

defecto presente en la mayoría de los navegadores web modernos. La tabla de datos y los índices son almacenados en esta capa de persistencia usando un mecanismo propio que permite la serialización eficiente de los componentes de los *quads RDF* como objetos *JSON*. Al mismo tiempo, un componente de *caché* permite un acceso eficiente a los datos almacenados reduciendo el número de búsquedas necesarios en la capa de persistencia.

Procesamiento de consultas *SPARQL*

La ejecución de una consulta *SPARQL* sobre un grafo *RDF* almacenado en el repositorio se realiza en dos pasos diferenciados por la biblioteca.
En primer lugar se lleva a cabo el análisis del texto de la consulta *SPARQL* que es transformado en un objeto *JSON* complejo que representa el árbol de sintaxis abstracta para la consulta. Este paso se ha implementado usando una *Parsing Grammar Expression* (*PEG*) [41] con capacidad para analizar léxicamente consultas *SPARQ 1.1 Update* y documentos *Turtle*.
A continuación, el árbol de sintaxis abstracta es transformado en un objeto *JSON* diferente que contiene la representación de una expresión en el álgebra *SPARQL cyganiak2005relational* equivalente de acuerdo con la semántica para *SPARQL* estandarizada por la *W3C*.
Esta expresión algebraica será el objeto de procesamiento por el motor de ejecución *SPARQL* del repositorio, que realizará las consultas adecuadas sobre la tabla léxica y los índices del repositorio para generar los resulados finales de la consulta y serializarlos en el formato deseado por el cliente.

Ejecución multi-hilo

Las aplicaciones *JavaScript* que se ejecutan en el navegador se encuentran limitadas a un único hilo de ejecución. Un posible mecanismo para sobrellevar esta limitación se encuentra en la *API* de *WebWorkers* [55] presente en las últimas versiones de la mayoría de navegadores web. Esta *API* hace posible ejecutar código *JavaScript* en un hilo de ejecución diferente con severas restricciones en cuanto a las capacidades del

navegador que ese hilo puede utilizar. El hilo principal de la aplicación y estos hilos especiales y restringidos se comunican mediante un protocolo de paso de mensajes.
La biblioteca que implementa el repositorio se beneficia del soporte para *WebWorkers*, si está disponible, para ejecutar consultas *SPARQL* de una manera concurrente en un conjunto de hilos restringidos. Desde el punto de vista de la arquitectura de la biblioteca, la interfaz entre cliente y repositorio está implementada en un componente que sirve de fachada al sistema. Si capacidad para usar *WebWorkers* es detectada en el navegador, un conjunto de hilos restringidos son creados y las consultas se distribuyen entre ellos. Cuando el procesamiento de una consulta ha finalizado, el resultado es recibido por el componente fachada que finalmente envía los resultados al cliente. Si el soporte para *WebWorkers* no está presente, la lógica presente en los hilos restringidos es cargada por el componente fachada en el hilo principal y se procede a ejecutar las consultas *SPARQL* de una manera secuencial.
Como ya hemos comentado, los hilos creados a través de la *API WebWorkers* se encuentran muy limitados, no pudiendo, por ejemplo, realizar peticiones web o acceder a la *API WebStorage*. Cuando un hilo restringido necesita llevar a cabo cualquiera de estas operaciones, un mensaje debe ser enviado al hilo principal para que la lleve a cabo y envíe el resultado al hilo restringido de modo que pueda continuar con su ejecución.

API de eventos

Las aplicaciones *JavaScript* presentan un modelo de ejecución reactivo, donde el código escrito de forma asíncrona, es ejecutado como respuesta a eventos que tienen lugar en la aplicación.
Nuestra implementación de un repositorio *RDF* para *JavaScript* hace posible usar este modelo de ejecución asíncrono, permitiendo que la aplicación registre funciones que serán ejecutadas cuando un determinado evento suceda en el grafo *RDF* almacenado en él.
Este modelo de ejecución también se encuentra próximo al modelo teórico que hemos presentado donde diferentes procesos se comunican a través de un espacio de tripletes

a través de operaciones de lectura y escritura sobre él.

El mecanismo de eventos funciona a dos niveles de abstracción diferentes: al nivel de las consultas *SPARQL* y al nivel de los nodos *RDF*.

Desde el punto de vista puramente *SPARQL*, el cliente puede subscribirse a una consulta *SPARQL* que será evaluada continuamente por el repositorio. Cada vez que una modificación del grafo *RDF* modifique los resultados de la consulta, la función subscrita a la consulta será invocada de nuevo con el nuevo resultado para la misma.

Al nivel de los nodos *RDF*, un cliente puede subscribirse a un nodo *RDF* particular, identificado por un *URI*, siendo notificado con el nuevo estado del nodo cada vez que una modificación del grafo *RDF* haga que los contenidos o la misma presencia del nodo en el grafo cambie.

Capítulo 4

Validación de la Propuesta y Ejemplos de aplicación

En los capítulos anteriores de este documento hemos abordado el problema de la construcción de *APIs* de datos enlazados semánticos desde la definición de un modelo formal y un modelo arquitectónico hasta el diseño de componentes software que permitan implementar dicha arquitectura en una solución real.

En este capítulo trataremos de mostrar la aplicabilidad de dicho modelo teórico y arquitectónico, así como de los componentes software que hemos desarrollado para poder llevarlo a la práctica. Para ello mostraremos los resultados obetnidos en la evaluación de rendimiento del componente clave par la construcción de aplicaciones clientes que hemos desarrollado, el repositorio de tripletes RDF escrito en *JavaScript* y describiremos algunos ejemplos concretos que muestran cómo pueden ser usados para construir aplicaciones software que hacen uso de una *API* de datos enlazados semánticos para ofrecer nuevas soluciones a dominios de aplicación reales, como la construcción de aplicaciones web sociales o la implementación de sistemas para visualización de la información en el navegador.

En todos los casos haremos un breve repaso de los problemas que el actual estado de implementación de este tipo de aplicaciones presenta para después centrarnos en qué ventajas aporta el uso de datos enlazados semánticos así como en la implementación de dicha solución en el marco tecnológico propuesto anteriormente en este

Query	Chrome 16	Safari 5	Firefox 11	MSIE 9
0	0.552	1.176	0.834	0.771
1	0.005	0.033	0.043	0.016
2	0.018	0.149	0.046	0.111
3	0.005	0.022	0.026	0.023
4	0.155	0.502	0.311	0.603
5	0.043	0.091	0.109	0.131
6	0.023	0.039	0.045	0.057
7	0.324	0.573	0.73	1.678
8	0.828	1.581	1.789	2.548
10	0.008	0.022	0.024	0.027
11	0.001	0.003	0.006	0.003
12	0.003	0.007	0.011	0.006
13	0.042	0.103	0.098	0.119
14	0.009	0.028	0.024	0.035

Cuadro 4.1: Pruebas de rendimiento *LUBM* para el repositorio *RDF*

documento.

4.1. Evaluación de rendimiento del Repositorio RDF para aplicaciones JavaScript

En el capítulo anterior (3.3.2) se describió la implementación de un repositorio RDF *SPARQL 1.1 Update* para aplicaciones *JavaScript*. Como parte final del proceso de desarrollo del repositorio, hemos realizado una evaluación de rendimiento de la biblioteca utilizando el modelo de evaluación *LUBM* [46] ajustado sus parámetros para usar un conjunto de datos pequeño, más adecuado al volumen de datos con el que debe tratar una aplicación web *JavaScript*. La evaluación se ha realizado para diferentes navegadores web en un equipo portátil medio.

Los casos de prueba se han construido automáticamente mediante el software generador de datos incluido dentro de *LUBM* y a continuación se han transformado en *JSON-LD* antes de ser cargados en el navegador para su ejecución. La cantidad final de tripletes almacenados en un único grafo en el repositorio ha sido de 100.545 tripletes. El cuadro 4.1 muestra los resultados obtenidos en milisegundos para las diferentes consultas *LUBM*.

Dado que el repositorio no soporta inferencia lógica, algunas consultas en *LUBM* han debido ser re-escritas para utilizar cláusulas *UNION* con el fin de cubrir todos los casos y obtener los resultados correctos esperados. Una consulta adicional que simplemente devuelve todos los tripletes ha sido también añadida. El texto de las consultas así como el código para ejecutar las consultas forma parte de la distribución de la biblioteca. En los resultados se puede apreciar como la mayoría de las consultas son evaluadas en un tiempo inferior al segundo. Siendo este el umbral máximo de tiempo que consideramos aceptable para construir aplicaciones cliente que puedan ser utilizadas por el usuario final de una forma aceptable, además, el volumen de datos consultados y almacenados en el repositorio para las pruebas, es muy superior al volumen normal de tripletes que almacena una aplicación cliente de escritorio típica. Realizando pruebas de carga con diferentes volumenes de datos, nos permite comprobar como el tiempo de las consultas se incrementa linealmente con el número de tripletes almacenados en el repositorio. Por último, cabe destacar que un elemento decisivo para el rendimiento de la biblioteca es el motor *JavaScript* del navegador web en el que se ejecuta la prueba. Diferentes navegadores con diferentes implementaciones de *JavaScript* ofrecen rendimientos muy dispares para los mismos datos e implementaciones de la biblioteca. Esto puede suponer un problema a la hora de construir una aplicacion con un comportamiento predecible en diferentes plataformas. Por otro lado, el incremento en el rendimiento de los motores *JavaScript* presentes en los actuales navegadores web es una de las principales preocupaciones de los fabricantes de dichos navegadores, con lo que es sensato esperar mejoras importantes en el rendimiento de estos motores y por lo tanto de la biblioteca y aplicaciones construidas sobre ella en el futuro cercano.

4.2. Un servidor personal de datos enlazados semánticos para la Web Social

La Web Social está compuesta hoy en día por un número ingente de diferentes servicios donde los usuarios mantienen fragmentos de su grafo social. La interoperabilidad entre diferentes redes sociales es un importante problema sin resolver en el diseño de la Web Social vigente hoy en día. Desde el punto de vista del acceso a los datos, la mayoría de los sitios web sociales pueden considerarse silos de información [47] donde la posibilidad de compartir datos a través aplicaciones se ve dificultada por mecanismos de autenticación, *APIs* y modelos de datos incompatibles.

La misma identidad de los usuarios también se encuentra fragmentada entre aplicaciones. Diferentes mecanismos de autenticación y sistemas de credenciales son usadas para identificar al usuario en diferentes redes sociales. Como resultado, funcionalidades como la importación del grafo social de contactos de una red social dónde ya se es miembro a un nuevo servicio son tareas complejas que los usuarios deben completar antes de poder empezar a utilizar el nuevo servicio de una forma satisfactoria, convirtiéndose de esta manera en un importante barrera de entrada para la adopción de nuevos servicios.

A pesar de los esfuerzos de estandarización a los niveles de *API* e identidad, como OAuth [49] y OpenID [107], la tendencia reciente consiste en concentrar la autenticación de los usuarios en un pequeño conjunto de proveedores de identidad como *Google*, *Twitter* o *Facebook*. Una consecuencia de esta tendencia es que no sólo los datos sociales de los usuarios del servicio sino también su mismas identidades son retenidos por un pequeño conjunto de proveedores de servicios web, con el riesgo que esto supone de perdida de esa identidad si el servicio desaparece o la subordinación de los intereses del usuario a las políticas corporativas de dichas compañías.

En esta sección describiremos cómo un servicio basado en una *API* de datos enlazados semánticos que hace uso de estándares desarrollados por las comunidades Web Semántica y Datos Enlazados Abiertos en conjunción con otras propuestas recientes aparecidas en el seno de dichas comunidades, como *WebID*, puede ser una alternati-

va viable para la construcción de redes sociales descentralizadas donde los usuarios tengan un mayor control sobre su identidad en la web y grafo social.
En concreto, el sistema descrito a continuación es capaz de recoger datos sociales de diferentes servicios en los que el usuario está registrado, transformando todos estos datos diversos en un único grafos social haciendo uso de un modelo de datos y una semántica unificadas. El sistema asocia también dicho grafo a una identidad *WebID* para el usuario que puede ser generada y mantenida por el servicio o estar almacenada en cualquier otro proveedor *WebID*.
El uso de tecnologías estandarizadas y abiertas hacen posible la integración el sistema como un nodo más en una red social semántica distribuida y desacoplada, introduciendo en dicha red los datos sociales ya existentes en otros servicios y ofreciendo al mismo tiempo la posibilidad de que las interacciones entre usuarios dentro de esta red puedan ser enviados de vuelta a redes sociales externas.

4.2.1. Principios de diseño

En el diseño del sistema, los siguientes principios básicos fueron seguidos para intentar conseguir una aplicación que cumpliese con los objetivos de funcionalidad anteriormente propuestos:

- **Construcción de un sistema extensible**, capaz de integrar datos de diferentes redes sociales ya existentes. Esto es posible gracias al diseño del sistema como un conjunto de extensiones para las diferentes redes sociales, capaces de captar información proveniente de estas redes y traducirlas a datos *RDF* usando una ontología común. Vocabularios dentro de la comunidad Web Semántica, especialmente diseñados para tratar con información personal y social como SIOC [18] y FOAF [20] fueron usados con este fin.

- **Exposición de los datos a través de una *API REST*** construida siguiendo los principios formales y la arquitectura propuesta en este documento. Esto

incluye la distinción entre recurso *HTTP* y recurso *RDF* asociado, los mecanismos anteriormente explicados para generar *URI* identificadores de los grafos asociados a los recursos o el uso de parámetros especiales para características como la paginación.

- **Uso de una identidad y autenticación distribuida basada en *WebID*.** *WebID* es un mecanismo de autenticación basado en el uso de *URIs* y un empleo particular de la criptografía de clave pública. El resultado de desreferenciar una *URI* asociada a una identidad *WebID* no es sólo la información de la clave pública necesario para validar la identidad del certificado autofirmado enviado por el cliente, sino un documento *RDF* descrito usando el vocabulario *FOAF* que incluye toda la información de perfil (o enlaces a esa información) que el dueño de dicha identidad esté dispuesto a compartir públicamente. Esta información es utilizada por el sistema para validar los derechos de acceso a los recursos expuestos examinando las relaciones de confianza establecidas entre el firmante de la petición y el dueño del recurso expuesto. El diseño de *WebID* sigue los principios arquitectónicos *REST*, con lo que ha sido muy fácil integrarlo en la arquitectura y modelo de servicios propuestos anteriormente en este documento.

- **Capacidad para exponer datos sociales como objetos sociales** [35], [36], que pueden ser definidos como entidades discretas de información generadas por los usuarios de un servicio web social y que sirven como enlaces entre los usuarios de dicho servicio, proveyendo al mismo tiempo a esas interacciones de un contexto. Posts en un servicio de blogging o fotos en un servicio de fotografía social como *Flickr*, son ejemplos de objetos sociales. Los objetos sociales son consumidos mediante un mecanismo de *pull* donde los clientes usan el protocolo *HTTP* para extraer una representación del objeto social expuesto como un recurso *REST* semántico [71] y almacenado en el servicio como un pequeño grafo *RDF* denominado *molécula RDF* [34]. *Capacidad para exponer datos sociales como un flujo social*, que puede ser definido como una colección ilimitada de datos sociales con una dimensión temporal y una identidad estable asociadas.

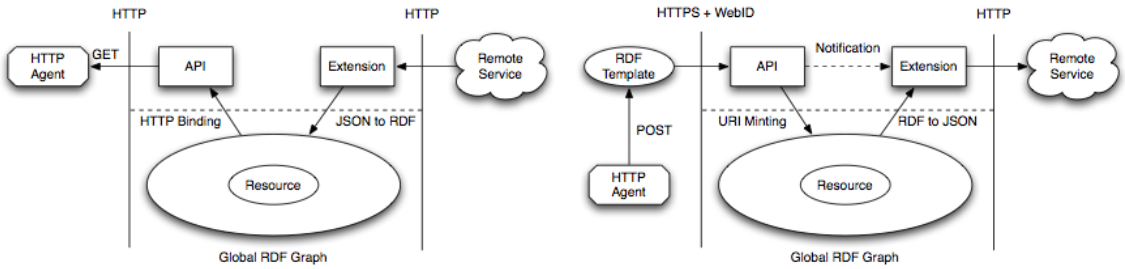


Figura 4-1: Flujo de información en el sistema.

Servicios de estado social como *Twitter* son ejemplos de este tipo de datos sociales. El uso que de este tipo de datos sociales hacen los usuarios es típicamente *push* donde es el servicio el que notifica al cliente sobre nuevos eventos en el flujo.

4.2.2. Arquitectura del sistema

La arquitectura del sistema se basa en el modelo arquitectónico expuesto en secciones anteriores de este documento. En este apartado expondremos las principales extensiones que de dicho modelo se han llevado a cabo con el fin de publicar los diferentes tipos de datos sociales que se pretendían compartir. La figura 4-1 muestra el flujo de información a través de los principales componentes del sistema.

- **Concepción de la aplicación como un sistema modular.** Cada módulo gestiona la interacción con una red social diferente y se ejecutan de una forma completamente autónoma cooperando a través del uso de un mecanismos de mensajes y de una ontología *RDF* común donde se exponen los tipos de recursos que la extensión va a generar. Gracias a este vocabulario común, el núcleo de la aplicación es capaz de integrar esos recurso en la *API HTTP*, recuperarlos del repositorio *RDF* y darles el formato adecuado si es necesario.

- **Integración de la autenticación *WebID*.** En la definición de la arquitectura de servicios *REST* semánticos expuesta anteriormente no se mencionaron

los aspectos de seguridad relativos a la autenticación del acceso a los recursos expuestos. La aplicación construida soluciona este problema mediante el uso de *WebID*. Cualquier recurso expuesto a través de la *API*, puede ser marcado como publico, privado o accesible sólo por un determinado número de identidades *WebID*. El sistema se encarga de autenticar la identidad de los usuarios accediendo a los recursos remotos con información sobre la identidad del cliente en cada petición *HTTP* realizando el proceso de autenticación *WebID* y comprobando el grafo *RDF* resultante de añadir la información del perfil remoto obtenida con los datos de perfil de usuario mantenidos localmente. Este proceso es completamente *REST* involucrando simplemente la desrefernciación de la *URI* asociada a la identidad del cliente y la comprobación criptográfica de su veracidad.

- **Exposición de un flujo social de datos.** El concepto de flujo social se traduce mal en términos *REST* tanto por su naturaleza no discreta y dinámica, como por el hecho de estar pensado para ser consumido mediante un mecanismo *push* en el que el servicio notifica a los clientes de la presencia de nuevos datos. En nuestra implementación hemos optado por una opción intermedia en el que el consumo del flujo es todavía *pull* siendo los clientes los encargados de realizar algún tipo de petición periódica para obtener los nuevos eventos expuestos. Para ello, el flujo se asocia a una *URI* estable y bien conocida, donde los clientes pueden realizar peticiones de solo lectura mediante operaciones *GET HTTP*. Las diferentes extensiones autónomas de la aplicación pueden publicar en cualquier momento información en el flujo insertando en el grafo recursos *RDF* con un tipo *SIOC MicroBlogPost*.

- **Implementación de *SPARQL* WebHooks**. Para mitigar el problema del consumo del flujo de datos mediante peticiones periódicas, el sistema también emplea un sistema que permite a un cliente con una *URI* desreferenciable e identificado por una identidad *WebID* crear una petición de notificación web (*WebHook*) con una consulta *SPARQL* asociada. Si una modificación del grafo

de la aplicación resulta en una respuesta satisfactoria de la consulta *SPARQL*, los resultados serán enviadas a la *URI* provista por la identidad *WebID* que creo la petición de notificación en el sistema. Las peticiones de notificación se exponen en el sistema como otro recurso semántico más con la excepción de que puedenser manipuladas, no solo por el administrador del servicio, sino por los clientes remotos que crearon dichas peticiones de notificación.

4.2.3. Detalles de implementación

El prototipo del sistema aquí detallado fue implementado usando la plataforma de desarrollo de aplicaciones *JavaScript Node.JS*. La capa de persistencia *RDF* se implementó usando la base de datos no relacional *MongoDB*, dónde se almacenaron las *moléculas RDF* directamente como objetos *JSON-LD*. Los componentes para llevar a cabo la generación de identidades *WebID* o llevar a cabo la autenticación también fueron desarrollados como bibliotecas para *Node.JS* y liberados posteriormente con licencia de Código Abierto.

En aquellos momentos en el que la ejecución de consultas *SPARQL* resultaba indispensable, nuestra propia librería que implementa un repositorio *RDF JavaScript* con soporte para *SPARQL 1.1 Update* fue utilizada, cargando en el repositorio los documentos *JSON-LD* recuperados desde *MongoDB* que son necesarios. Como parte de la implementación del sistema, componentes modulares para los servicios de web sociales *Twitter*, *Github* y un componente genérico para feeds *RSS/Atom* fueron desarrollados.

La *API HTTP* expuesta soporta peticiones procedentes de clientes *JavaScript* ejecutándose en el navegador desde dominios diferentes a los del servicio gracias a la implementación del estándar de peticiones *cross domain* (*CORS*) [127] de la *W3C*.

Por último, algunas aplicaciones clientes *JavaScript* han sido desarrollados e incluidas en el sistema para consumir la *API* que el servicio expone. Una primera aplicación administrativa accede a los recursos de perfil y configuración de la *API* para configurar la información de perfil del usuario del servicio así, como las políticas de acceso a los recursos expuestos y de gestión de la identidad *WebID* del usuario.

Figura 4-2: Aplicación *JavaScript* mostrando el flujo de actividad de un usuario.

La figura 4-2 muestra el aspecto de otra de las aplicaciones incluidas capaz de visualizar el flujo social de una identidad *WebID* alojada en el servicio. Se puede observar como notificaciones generadas por diferentes componentes (Twitter, Github) se mezclan en el flujo social del usuario.

Ambas aplicaciones usan nuestro repositorio *RDF JavaScript* ejecutándose en el navegador para agregar los recursos *REST* semánticos expuestos a través de la *API* de datos enlazados semánticos y construir con ellos una página final *HTML* que también incorpora la información *RDF* extraída mediante el uso del estándar *RDFa*.

4.3. Visualización de datos *RDF* en aplicaciones *JavaScript*

En el caso de aplicación anterior, hemos visto como el uso de una *API* de datos enlazados semánticos hace posible transformar el diseño y las características de un dominio de aplicación especifico, como es el caso de la construcción de redes sociales

de usuarios. En esta sección nos centraremos en un caso de aplicación más concreto, como es el de la construcción de visualizaciones de datos en un cliente *JavaScript* ejecutándose en el navegador, para intentar mostrar como el uso de información semántica disponible en una *API* enlazada semántica y el uso de librerías específicas que hagan su tratamiento sencillo, como el repositorio *RDF JavaScript* que hemos descrito en secciones anteriores, permite ofrecer soluciones alternativas más simples en este dominio de aplicación.

Generar visualizaciones de datos a partir de grafos *RDF*, especialmente usando *JavaScript* en el navegador es un proceso complejo y tedioso. La dificultad radica en el hecho de que las bibliotecas de visualización de datos disponibles, como *D3* [16], no están preparadas para trabajar directamente con el modelo de datos *RDF*, necesitándose por tanto, un proceso de limpieza y transformación de los datos en el modelo de datos que la biblioteca demanda para cada tipo de visualización en concreto, problema que se vuelve todavía más compleja si se intenta combinar en la visualización diferentes fuentes de datos de varios proveedores.

En esta sección describmos una biblioteca *JavaScript* para la construcción de visualizaciones de datos, que tiene en cuenta las características del modelo de datos *RDF* para definir la visualización. El sistema se ha construido sobre la biblioteca que implementa un repositorio *RDF* para *JavaScript* con soporte para consultas *SPARQL 1.1 Update* que describimos en secciones anteriores.

4.3.1. Gramática de gráficos para *RDF*

La biblioteca se ha construido sobre las ideas de una gramática de gráficos [133], un conjunto finito de operaciones de alto nivel que se pueden combinar de diferentes maneras para generar un numero determinado visualizaciones gráficas, dados unos datos iniciales de entrada.

La biblioteca construida extiende las operaciones clásicas de este tipo de gramáticas de gráficos, con dos operaciones específicas para manipular datos *RDF*:

- **Selección de datos *RDF***, mediante consultas *SPARQL* y la selección de

```
new LinkedVis({width: 500, height: 500})
  .from(data, 'text/n3')
  .struct(LinkedVis.List("{ ?node a gg:Diamond }"))
  .stackLayer(function(l) {
    l.bind({'shape':        'circle',
            'x':            {'rdfProperty': 'gg:x',
                             'scale':       'continous'},
            'y':            {'rdfProperty': 'gg:y',
                             'scale':       'continous'},
            'radius':       {'rdfProperty': 'gg:carat',
                             'scale':       'continous',
                             'rangeMin':     1,
                             'rangeMax':     10},
            'fill':         {'rdfProperty': 'gg:cut',
                             'scale':       'hue'},
            'stroke':       'black',
            'stroke-width': 1})
     .layout(LinkedVis.Cartesian())
     .theme({'title':'ggplot diamonds x,y,carat,cut',
             'axis':{'y':'gg:y', 'x':'gg:x'}})
   })
  .render("#canvas");
```

Cuadro 4.2: Definición de una visualización usando la gramática de gráficos.

una estructura de datos de destino: lista, árbol o grafo, donde los nodos *RDF* seleccionados serán automáticamente coercionados.

- ***Join* de datos generalizado**. Mediante el cual los nodos *RDF* recuperados e introducidos en la estructura de datos seleccionada, son asociados a marcas visuales en la visualización, traduciéndose durante el proceso las propiedades *RDF* del recurso en características visuales de la marca visual asociada, como la posición, el radio, la altura y anchura, el color, etc.

La tabla 4.2 muestra la sintaxis final de la biblioteca, mostrando como el vocabulario de la gramática puede usarse para construir una visualización. La figura 4-3 muestra el resultado final de esa misma descripción.

Se puede observar como la función *struct* introduce una estructura de datos de destino para los nodos *RDF* recuperados por una consulta *SPARQL*. A su vez, la función *bind* realiza el *join* entre datos y marcas visuales, usando para ello las propiedades *RDF* de los nodos recuperados que son asignadas a diferentes valores de las propiedades de la marca.

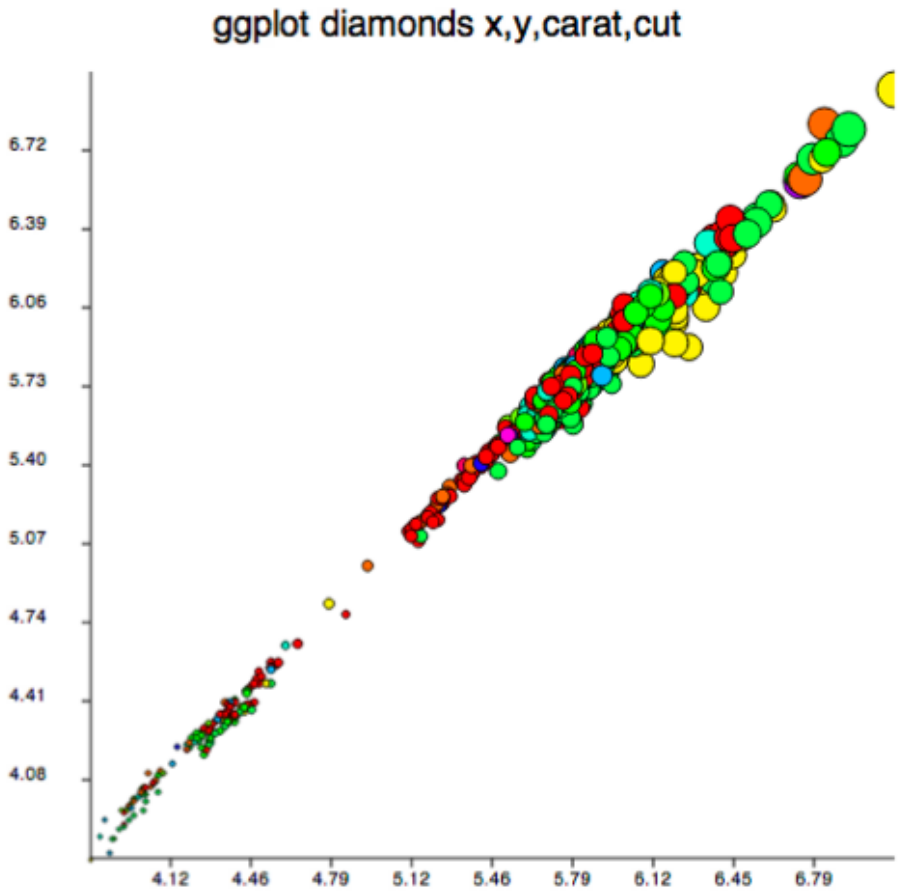


Figura 4-3: Visualización generada por la biblioteca a partir del código del cuadro 4.2

4.3.2. Diseño e implementación

El diseño de la biblioteca se basa en la ejecución de cuatro etapas que conducen a la generación de la visualización final como un documento *SVG* insertado en el árbol *DOM* de la página *HTML* de destino.

- **Selección de datos**. Es el proceso por el que el usuario de la gramática selecciona el origen de los datos *RDF*, un recurso *REST* semántico remoto, el texto de un documento *RDF* o una instancia del objeto repositorio *RDF*, y lo transforma en una estructura de destina mediante una consulta *SPARQL*. La biblioteca soporta tres diferentes estructuras de datos de desitno, listas, árboles y grafos. Cada una de estas estructuras requieren que el usuario especifique consultas *SPARQL* con cierta estructura, usando valores específicos de las variables *SPARQL* que luego serán utilizados por la lógica de la estructura de datos de destino para recuperar los datos.

- **Composición de capas**. La visualización es construida por la biblioteca como un conjunto de capas que pueden ser superpuestas, compartiendo entonces los mismos límites que la capa inferior, o anidadas dentro de otra, con lo que la nueva capa queda restringida a un subárea de la capa padre. La capa recibe la selección de datos actual que será usada para generar las marcas visuales necesarias dentro de la capa.

- ***join* de marcas visuales**. Dentro de una capa, es posible realizar una operación de *join* entre los datos de la selección anterior y un tipo de marca visual con determinadas propiedades estéticas. La biblioteca soporta diferentes tipos de marcas, entre las que algunas coinciden con elementos *SVG*, como *rect*, *circle* o *text*. Las propiedades estéticas de cada marca se asocian a las propiedades *RDF* del nodo *RDF* que ha sido asignado a la marca mediante la operación de *join*. Las propiedades estéticas se pueden clasificar en propiedades dimensión, como la posición x, y, el radio, etc, propiedades de estilo, modeladas de acuerdo a las propiedades *CSS* como el color de relleno o el color de trazo y por

último, propiedades específicas para el tipo de layout que se ha asignado a la capa, como veremos a continuación. Los valores de estas propiedades estéticas se pueden especificar como un valor constante, el valor de la propiedad *RDF* asociada o una función de escala que toma como valor de entrada el valor de la propiedad *RDF*.

- **Procesamiento del layout**. Cada capa de la visualización debe declarar una propiedad de *layout*, usada por la biblioteca para computar la posición de cada marca visual en la visualización. Para ello el *layout* utilizará el valor de las propiedades estéticas de cada marca visual. Los *layouts* soportados por la biblioteca se organizan de forma jerárquica, siendo los *layouts cartesiano* y *polar* los más genéricos. Distintos *layouts* generan diferentes visualizaciones y exigen diferentes propiedades en las marcas visuales.

4.3.3. Visualizaciones enlazadas

Una de las funcionalidades de la biblioteca construida que no suele estar presente en otras soluciones para la generación de visualizaciones de datos es el de la inclusión de abundantes meta-datos *RDF* en la propia visualización. Estos es posible gracias a que el formato último de la visualización generada consiste en un documento SVG con formato *XML*. Dos grafos están presentes en la visualización final generada.

- **Grafo de datos**. Es un grafo *RDF* conteniendo la agregación de todos los nodos *RDF* seleccionados en todas las capas de la visualización.

- **Meta-grafo**. Conteniendo una descripción de la visualización usando una ontología específica. Este grafo incluye detalles como el número de capas, las estructuras de datos usadas o qué nodos del grafo de datos están asociados a qué marcas visuales.

La presencia de ambos tipos de grafos abre la puerta a la construcción de bibliotecas capaz de manipular la visualización en función de la descripción de la misma que se encuentra insertada en ella para añadir un elemento de interactividad al uso

que de la visualización harán los usuarios finales. La generación de ambos grafos de meta-datos es opcional y puede ser desactivada.

Capítulo 5

Conclusiones y trabajo futuro

A lo largo de los capítulos de este documento se ha realizado una completa caracterización de las APIs de datos semánticos.
Para ello se ha tomado como punto de inicio la discusión de los problemas actuales que se encuentran en el desarrollo web y que motivaron nuestro trabajo en primer lugar, así como el repaso al estado actual de las distintas tecnologías y áreas de investigación que se han tomado como base. A continuación se ha realizado un repaso de nuestra propuesta de solución desde distintos niveles: desde el punto de vista puramente formal hasta su implementación software y distintos ejemplos de aplicación de esta solución tecnológica a problemas reales de desarrollo web.
Por último y a modo de conclusiones sobre el trabajo realizado, se pueden destacar los siguientes aspectos:

- La web orientada a documentos se encuentra en pleno proceso de transformación en una red de datos y APIs que son consumidas por agentes software. El desarrollo de las plataformas móviles y su ecosistema de aplicaciones, de servicios web sociales e incluso la incipiente *Internet of Things* [5] son los principales motores detrás de esta transformación.

- Esta nueva web de datos presenta importantes desafíos desde el punto de vista del desarrollo web que todavía no han sido resueltos, incluyendo problemas como la privacidad de los datos, el control de la información, la interoperabilidad entre

aplicaciones o la automatización del acceso a los datos.

- Diversas tecnologías y areas de investigación desarrolladas en los últimos años, tienen el potencial de solucionar diferentes aspectos relevantes a estos problemas. Dichas tencologías incluyen las arquitecturas *REST*, ciertos desarrollos en el área de la Web Semántica y la iniciativa *Open Linked Data*, así como la investigación en sistemas distribuidos.

- El concepto de API de Datos Semánticos es un intento por combinar los avances en todos estos campos para ofrecer una interfaz práctica, respetuosa con los conceptos arquitectónicos de la web, tal y como están recogidos en los principios *REST* y basada en estándares que facilite la interconexión de datos entre agentes software y garantice la privcacidad y el control sobre sus datos a los usuarios.

- Desde el punto de vista puramente formal, las APIs de datos semánticos y los agentes y servicios que interaccionan a través de ellos, se pueden modelar como un sistema distribuido descrito usando un cálculo de proceso similar al Cálculo Pi con tipos y en el modelo de *Espacios de Tripletes*. Esto permite razonar sobre la corrección, requisitos y el diseño de las aplicaciones web que se van a implementar usando APIs semánticas, además de abrir la puerta al tratamiento formal de problemas como la corrección del diseño.

- Este modelo teórico se puede transformar en una arquitectura software para el desarrollo de APIs de datos basadas en los principios arquitecturales *REST* y en las consideraciones fundamentales de la iniciativa *Open Linked Data*, usando como modelo el estándar básico de la Web Semántica: *RDF*.

- El uso de tecnologías semánticas como *RDF* no es incompatible con la utilización de herramientas y tecnologías básicas en el desarrollo web, con una gran base de usuarios y años de refinamiento en cuanto a rendimiento y estabilidad, como las bases de datos relacionales. Con el desarrollo de bibliotecas para la traducción de consultas *SPARQL* a *SQL* y del modelo de datos *RDF* al modelo de datos relacional, hemos visto como ambas tecnologías pueden hacerse compatibles.

- Con el desarrollo de una biblioteca *JavaScript* que implementa un completo repositorio de tripletes con soporte para responder a consultas *SPARQL* que se puede ejecutar dentro del navegador web, hemos mostrado como los agentes software de nuestro modelo formal pueden ser implementados dentro del cliente web por excelencia, el navegador web, además de demostrar la viabilidad del uso de tecnologías semánticas fuera del lado servidor de la arquitectura web.

- A través del desarrollo de una aplicación agregadora de datos sociales que usa nuestra arquitectura de APIs de datos semánticos y componentes software, hemos mostrado como dicho modelo puede usarse para solucionar algunos de los problemas de las redes sociales actuales, incluyendo la privacidad y autenticación, la integración de datos de diferentes servicios así como la capacidad de dotar de control sobre sus propios datos a los usuarios de dichas aplicaciones.

- El accesso a datos semánticos que garantiza nuestra arquitectura de APIs semánticas, así como la flexibilidad y expresividad del modelo de datos semántico *RDF* permite encontrar nuevas soluciones que simplifiquen problemas complejos como la construcción de visualizaciones de datos interactivas, que incorporen datos provenientes d ediferentes servicios, tal y como hemos mostrado con el desarrollo de una biblioteca de visualización de datos para el navegador, construida sobre nuestro repositorio *RDF JavaScript*.

Lejos de agotar la investigación en el área de la construcción de interfaces de para datos semánticos, el trabajo recogido en este documento abre la puerta a posibles desarrollos futuros que continúen y expandan el trabajo aquí realizado.
Como posibles líneas de investigación abiertas podríamos mencionar las siguientes:

- La integración del trabajo realizado con tecnologías móviles. En este documento se ha mostrado como los clientes web basados en navegadores web pueden ser asimilados a los agentes software de nuestro modelo formal, a través de bibliotecas software *JavaScript* que les permiten interaccionar con los datos semánticos. De una manera similar, los diferentes dispositivos móviles se podrían integrar como

clientes de APIs semánticas desarrollando los diferentes componentes software necesarios para que puedan interaccionar con los datos semánticos recuperados desde dichas APIs.

- Estos componentes semánticos desarrollados para diferentes plataformas móviles se podrían utilizar no solo para comunicar aplicaciones con APIs semánticas sino que se podrían integrar como una *capa semántica* dentro de dicha plataforma, de forma tal que diferentes aplicaciones ejecutándose dentro del mismo terminal tengan acceso a los datos expuestos por otras aplicaciones, así como los servicios de datos nativos del terminal usando estándares semánticos. Esto supone una particularización del modelo formal de interacción entre agentes semánticos que hemos descrito a un conjunto de procesos ejecutándose dentro del mismo dispositivo.

- El trabajo que se ha iniciado en la construcción de componentes software para la implementación de nuestra arquitectura de APIs semánticos podría expandirse para llevar a cabo el desarrollo de un verdadero *framework* para el desarrollo de APIs semánticas, que sea capaz de tomar como entrada la descripción de una API descrita en *RDF* y algunos parámetros de configuración y transformarla en una versión ejecutable de dicha API lista para ser puesta en producción.

- La arquitectura propuesta en este documento cubre el desarrollo de servicios de datos basados en el modelo tradicional *REST* web, donde las peticiones son iniciadas siempre por los clientes. Nuevos desarrollos en tecnologías web, como los *Web Sockets* [58], están transformando este modelo *pull* en un modelo *push* donde el servicio es capaz de iniciar la comunicación directamente con el cliente. Esta nuevas tecnologías web son compatible con nuestro modelo formal de APIs semánticas, pero para poder integrarlas dentro de soluciones reales es necesario expandir nuestra propuesta arquitectural y resolver problemas técnicos como el envío de datos semánticos a través de *streams* de datos en tiempo real.

- Nuestro trabajo de adaptación de un sistema de almacenamiento de informa-

ción y consulta relacional a los estándares semánticos para integrarlo dentro de nuestra arquitectura web, puede emularse dentro de otros sistemas de almacenamiento y recuperación de información donde la integración de dichos datos con otros sistemas software resulta complicada. Un caso especialmente interesante donde se podría construir una capa de integración semántica es el de los sistemas *Big Data* como *Hadoop* o *Hive*.

- Un dominio de aplicación especialmente adecuado para estudiar la posible aplicación de APIs semánticas lo constituye la configuración de sistemas de desarrollo, denominada *dev ops* [119]. Diversas fuentes de datos, bibliotecas, máquinas, repositorios de código fuente, requisitos, etc se encuentran almacenados en sistemas aislados y desconectados mutuamente. La introducción de tecnologías de *Cloud Computing* en los escenarios de desarrollo han hecho el problema todavía más acuciante. El uso de tecnologías semánticas y APIs de datos semánticos, pueden ofrecer una solución para integrar todas estas fuentes de datos presentes en los entornos de desarrollo software actuales.

- Por último, otro campo interesante para la aplicación de interfaces semánticas entre sistemas es el de la denominada *Internet of Things* donde la necesidad de intercambio de datos entre dispositivos con capacidades de computo muchas veces limitadas, podrían permitir una propuesta estándar basada en nuestro modelo formal y arquitectónico para el intercambio de datos usando tecnología semántica.

Bibliografía

[1] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. Rdfa in xhtml: Syntax and processing. *Recommendation, W3C*, 2008.

[2] Rosa Alarcon, Erik Wilde, and Jesus Bellido. Hypermedia-driven restful service composition. In *Service-Oriented Computing*, pages 111–120. Springer, 2011.

[3] Keith Alexander. Rdf in json: a specification for serialising rdf in json. *SFSW 2008*, 2008.

[4] Mark Allman. An evaluation of xml-rpc. *ACM sigmetrics performance evaluation review*, 30(4):2–11, 2003.

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[6] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A Nagy, and Stefan Pharies. Web services inspection language (ws-inspection) 1.0. *IBM, Microsoft*, 2001.

[7] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10, 2004.

[8] David Beckett and Tim Berners-Lee. Turtle-terse rdf triple language. *W3C Team Submission*, 14, 2008.

[9] Djamal Benslimane, Schahram Dustdar, and Amit Sheth. Services mashups: The new generation of web applications. *Internet Computing, IEEE*, 12(5):13–15, 2008.

[10] Tim Berners-Lee. Axioms of web architecture. Technical report, Technical report, W3C, December 1996. http://www. w3. org/DesignIssues/Axioms. html, 1997.

[11] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable rdf syntax. *W3C Team Submission (January 2008) http://www. w3. org/TeamSubmission*, (3), 1998.

[12] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 163–176. ACM, 2008.

[13] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. Verifying policy-based security for web services. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 268–277. ACM, 2004.

[14] Chris Bizer, Richard Cyganiak, and Tom Heath. How to publish linked data on the web. *Retrieved June*, 20:2008, 2007.

[15] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.

[16] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[17] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[18] John G Breslin, Stefan Decker, Andreas Harth, and Uldis Bojars. Sioc: an approach to connect web-based communities. *International Journal of Web Based Communities*, 2(2):133–142, 2006.

[19] Dan Brickley and Ramanathan V Guha. {RDF vocabulary description language 1.0: RDF schema}. 2004.

[20] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.98. *Namespace document*, 9, 2010.

[21] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the expressiveness of linda coordination primitives. *Information and Computation*, 156(1):90–121, 2000.

[22] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Process calculi for coordination: From linda to javaspaces. In *Algebraic Methodology and Software Technology*, pages 198–212. Springer, 2000.

[23] Christoph Bussler. A minimal triple space computing architecture. In *Proceedings of WIW*, volume 134, 2005.

[24] Luca Cardelli and Andrew D Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, pages 140–155. Springer, 1998.

[25] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving sparql-to-sql translation. *Data & Knowledge Engineering*, 68(10):973–1000, 2009.

[26] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[27] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. Sparql protocol for rdf. *World Wide Web Consortium (W3C) Recommendation*, 2008.

[28] UDDI Consortium et al. Uddi: Universal description, discovery and integration.

[29] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[30] Richard Cyganiak and Chistian Bizer. Pubby-a linked data frontend for sparql endpoints. *Retrieved fro m http://www4. wiwiss. fu-berlin. de/pubby/at May*, 28:2011, 2008.

[31] Florian Daniel and Barbara Pernici. Insights into web service orchestration and choreography. *International Journal of E-Business Research (IJEBR)*, 2(1):58–77, 2006.

[32] Souripriya Das, Seema Sundara, and Richard Cyganiak. {R2RML: RDB to RDF Mapping Language}. 2012.

[33] Christine Denney, Colin Batchelor, Olivier Bodenreider, Sam Cheng, John Hart, John Hill, John Madden, Mark Musen, Elgar Pichler, Matthias Samwald, et al. Creating a translational medicine ontology. In *International Conference on Biomedical Ontology*, volume 24, 2009.

[34] Li Ding, Tim Finin, Yun Peng, Paulo Pinheiro Da Silva, and Deborah L McGuinness. Tracking rdf graph provenance using rdf molecules. In *Proc. of the 4th International Semantic Web Conference (Poster)*, 2005.

[35] Jyri Engeström. Why some social network services work and others dont-or: the case for object-centered sociality. *entrada de blog]. En: Zengestrom.¡http://www. zengestrom. com/blog/2005/04/why_some_social. html*, 2005.

[36] Ritva Engeström. Voice as communicative action. *Mind, culture, and activity*, 2(3):192–215, 1995.

[37] Dennis R FATLAND. Open data protocol (odata): A practical web protocol for data query and retrieval. In *2011 GSA Annual Meeting in Minneapolis*, 2011.

[38] Dieter Fensel. Triple-space computing: Semantic web services based on persistent publication of information. In *Intelligence in Communication Systems*, pages 43–53. Springer, 2004.

[39] R Fielding. Httprange-14 resolved, 2005.

[40] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[41] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[42] Paul Gearon, Alexandre Passant, and Axel Polleres. Sparql 1.1 update. *Working draft WD-sparql11-update-20110512, W3C (May 2011)*, 2011.

[43] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[44] Birte Glimm and Bijan Parsia. Sparql. entailment regimes. wc working draft. *World Wide Web Consortium (WC), June,.: http://www. w3. org/TR/2009/WD-sparql11-entailment-20100601/.(Cit. on p.)*.

[45] Alexander Graf. Rdfa vs. microformats. *Digital Enterprise Research Institute, Innsbruck*, 2007.

[46] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[47] Harry Halpin. Beyond walled gardens: Open standards for the social web. *SDOW2008, Karlsruhe*, 2008.

[48] Eran Hammer-Lahav. Link-based resource descriptor discovery. *draft-hammer-discovery-03 (work in progress)*, 2009.

[49] Eran Hammer-Lahav. The oauth 1.0 protocol. 2010.

[50] Andreas Harth and Stefan Decker. Yet another rdf store: Perfect index structures for storing semantic web data with contexts. Technical report, DERI Technical Report, 2004.

[51] Michael Hausenblas. Exploiting linked data to build web applications. *Internet Computing, IEEE*, 13(4):68–73, 2009.

[52] Patrick Hayes and Brian McBride. Rdf semantics, 2004.

[53] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[54] Martin Hepp. Goodrelations: An ontology for describing products and services offers on the web. In *Knowledge Engineering: Practice and Patterns*, pages 329–346. Springer, 2008.

[55] I Hickson. Web workers. w3c draft (accessed on sept. 8, 2010).

[56] Ian Hickson. Html microdata. *W3C Working Draft*, 24, 2011.

[57] Ian Hickson. Web storage. *Draft, W3C, October 4th 2011http://dev. w3. org/html5/webstorage/accessed on*, 4(11), 2011.

[58] Ian Hickson. The websocket api. *W3C Working Draft WD-websockets-20110929, September*, 2011.

[59] Ian Hickson, Gregg Kellogg, Jeni Tennison, and Iván Herman. Microdata to rdf: Transformation from html+ microdata to rdf. *W3C Recommendation*, 2012.

[60] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[61] Aidan Hogan, Andreas Harth, Alexandre Passant, Stefan Decker, and Axel Polleres. Weaving the pedantic web. 2010.

[62] James Hollenbach, Joe Presbrey, and Tim Berners-Lee. Using rdf metadata to enable access control on the social semantic web. In *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, volume 514, 2009.

[63] Koen Holtman and Andrew Mutz. Transparent content negotiation in http. Technical report, RFC 2295, March, 1998.

[64] Ian Horrocks and Peter F Patel-Schneider. Reducing owl entailment to description logic satisfiability. In *The Semantic Web-ISWC 2003*, pages 17–29. Springer, 2003.

[65] Ian Horrocks, Peter F Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Web semantics: science, services and agents on the World Wide Web*, 1(1):7–26, 2003.

[66] Wolfgang Hoschek. The web service discovery architecture. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 38–38. IEEE, 2002.

[67] Antonio J Jara, Pedro Martinez-Julia, and Antonio Skarmeta. Light-weight multicast dns and dns-sd (lmdns-sd): Ipv6-based resource and service discovery for the web of things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 731–738. IEEE, 2012.

[68] FENG Jian-wen. Design and implement of erp system based on soa and oss. *Modern Computer*, 5:034, 2008.

[69] N Kavantzas and D Burdett. et all,web services choreography description language version 1.0 (ws-cdl), w3c, 2005.

[70] Rohit Khare and Tantek Çelik. Microformats: a pragmatic path to the semantic web. In *Proceedings of the 15th international conference on World Wide Web*, pages 865–866. ACM, 2006.

[71] Sheila Kinsella, Andreas Harth, Alexander Troussov, Mikhail Sogrin, John Judge, Conor Hayes, and John G Breslin. Navigating and annotating semantically-enabled networks of people and associated objects. In *Why Context Matters*, pages 79–96. Springer, 2008.

[72] Graham Klyne, Jeremy J Carroll, and Brian McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

[73] Jacek Kopecky, Karthik Gomadam, and Tomas Vitvar. hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, volume 1, pages 619–625. IEEE, 2008.

[74] Jacek Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *Internet Computing, IEEE*, 11(6):60–67, 2007.

[75] Jacek Kopecky, Tomas Vitvar, and Dieter Fensel. Microwsmo: Semantic description of restful services. *Online at http://www. wsmo. org/TR/d38/v0*, 1, 2008.

[76] Markus Lanthaler, Michael Granitzer, and Christian Gütl. Semantic web services: state of the art. In *Proc. of the IADIS Int. Conf. on Internet Technologies & Society (ITS 2010), Perth, Australia*, 2010.

[77] Markus Lanthaler and C Gutl. Towards a restful service ecosystem. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 209–214. IEEE, 2010.

[78] Holger Lausen, Axel Polleres, and Dumitru Roman. Web service modeling ontology (wsmo). *W3C Member Submission*, 3, 2005.

[79] Baoan Li and Mingxing Li. Research and design on the refinery erp and eerp based on soa and the component oriented technology. In *Networking and Digital Society, 2009. ICNDS'09. International Conference on*, volume 1, pages 85–88. IEEE, 2009.

[80] Olga Liskin, Leif Singer, and Kurt Schneider. Teaching old services new tricks: adding hateoas support as an afterthought. In *Proceedings of the Second International Workshop on RESTful Design*, pages 3–10. ACM, 2011.

[81] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *The Journal of logic and algebraic programming*, 70(1):96–118, 2007.

[82] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[83] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.

[84] Marino Miculan and Caterina Urban. Formal analysis of facebook connect single sign-on authentication protocol. In *SOFSEM*, volume 11, pages 22–28, 2011.

[85] Robin Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[86] Robin Milner. Functions as processes. *Mathematical structures in computer science*, 2(02):119–141, 1992.

[87] Robin Milner. *The polyadic π-calculus: a tutorial*. Springer, 1993.

[88] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[89] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. Owl 2 web ontology language: Profiles. *W3C recommendation*, 27:61, 2009.

[90] Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can owl and logic programming live together happily ever after? In *The Semantic Web-ISWC 2006*, pages 501–514. Springer, 2006.

[91] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between owl and relational databases. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):74–89, 2009.

[92] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. Owl 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27:17, 2009.

[93] San Murugesan. Understanding web 2.0. *IT professional*, 9(4):34–41, 2007.

[94] Srini Narayanan and Sheila A McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88. ACM, 2002.

[95] H Penny Nii. Blackboard application systems, blackboard systems and a knowledge engineering perspective. *AI magazine*, 7(3):82, 1986.

[96] BPEL OASIS. Web services business process execution language, 2007.

[97] Chimezie Ogbuji. Sparql 1.1 graph store http protocol. *W3C Working Draft*, 12, 2011.

[98] S Ozses and S Ergul. Cross-domain communications with jsonp, 2009.

[99] Kevin R Page, David C De Roure, and Kirk Martinez. Rest and linked data: a match made for domain driven development? In *Proceedings of the Second International Workshop on RESTful Design*, pages 22–25. ACM, 2011.

[100] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[101] Cesare Pautasso. Restful business process management. 2010.

[102] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[103] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *The Semantic Web-ISWC 2006*, pages 30–43. Springer, 2006.

[104] Eric PrudHommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[105] Mark Pruett. *Yahoo! pipes*. O'Reilly, 2007.

[106] David Recordon and Brad Fitzpatrick. Openid authentication 1.1. *Finalized OpenID Specification, May*, 2006.

[107] David Recordon and Drummond Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, pages 11–16. ACM, 2006.

[108] Dave E. Reynolds. Linked-data api: Api and formats to simplify use of linked data by web developers, 2011.

[109] Kurt Rohloff and Richard E Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, page 4. ACM, 2010.

[110] Jason Ronallo. Html5 microdata and schema. org. *Code4Lib Journal*, (16), 2012.

[111] Daniel Rubio. Wadl: The rest answer to wsdl, 2007.

[112] Marwan Sabbouh, Jeff Higginson, Salim Semy, and Danny Gagne. Web mashup scripting language. In *Proceedings of the 16th international conference on World Wide Web*, pages 1305–1306. ACM, 2007.

[113] Leo Sauermann, Richard Cyganiak, and Max Völkel. Cool uris for the semantic web. 2011.

[114] M-T Schmidt, Beth Hutchison, Peter Lambros, and Rob Phippen. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.

[115] Amit P Sheth, Karthik Gomadam, and Jon Lathem. Sa-rest: semantically interoperable and easier-to-use services and mashups. *Internet Computing, IEEE*, 11(6):91–94, 2007.

[116] Elena Simperl, Reto Krummenacher, and Lyndon Nixon. A coordination model for triplespace computing. In *Coordination Models and Languages*, pages 1–18. Springer, 2007.

[117] Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In *OWLED*, volume 258, 2007.

[118] Evren Sirin, Michael Smith, and Evan Wallace. Opening, closing worlds-on integrity constraints. In *OWLED*, 2008.

[119] David Mitchell Smith. Hype cycle for cloud computing 2011. *Gartner Inc., Stamford*, 2011.

[120] M Sporny, G Kellogg, and M Lanthaler. Json-ld 1.0-a json-based serialization for linked data. *W3C Working Draft*, 2013.

[121] Manu Sporny, S Corlosquet, T Inkster, H Story, B Harbulot, and R Bachmann-Gmür. Webid 1.0: Web identification and discovery. unofficial draft (august 2010).

[122] San-Tsai Sun, Eric Pospisil, Ildar Muslukhov, Nuray Dindar, Kirstie Hawkey, and Konstantin Beznosov. What makes users refuse web single sign-on?: an empirical investigation of openid. In *Proceedings of the Seventh Symposium on Usable Privacy and Security*, page 4. ACM, 2011.

[123] CURIE Syntax. Curie syntax 1.0. *Policy*, 3, 2004.

[124] Toshiro Takase, Satoshi Makino, Shinya Kawanaka, Ken Ueno, Christopher Ferris, and Arthur Ryman. Definition languages for restful web services: Wadl vs. wsdl 2.0. *IBM Reasearch*, 2008.

[125] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L McGuinness. Integrity constraints in owl. In *AAAI*, 2010.

[126] Jürgen Umbrich, Michael Hausenblas, Phil Archer, Eran Hammer-Lahav, and Erik Wilde. Discovering resources on the web. Technical report, Technical Report, DERI, 2009.

[127] Anne van Kesteren. Cross-origin resource sharing. w3c working draft, version wd-cors-20100727, 2010.

[128] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Jos De Roo, Rik Van de Walle, and Joaquim Gabarro Vallés. Description and interaction of restful services for automatic discovery and execution. In *2011 FTRA International workshop on Advanced Future Multimedia Services (AFMS 2011)*. Future Technology Research Association International (FTRA), 2011.

[129] Goetz Viering, Christine Legner, and Frederik Ahlemann. The (lacking) business perspective on soa-critical themes in soa research. In *Wirtschaftsinformatik (1)*, pages 45–54. Citeseer, 2009.

[130] Tomas Vitvar, Jacek Kopeckỳ, Jana Viskova, and Dieter Fensel. Wsmo-lite annotations for web services. In *The semantic web: Research and applications*, pages 674–689. Springer, 2008.

[131] Stuart Weibel. The dublin core: a simple content description model for electronic resources. *Bulletin of the American Society for Information Science and Technology*, 24(1):9–11, 1997.

[132] Erik Wilde and Michael Hausenblas. Restful sparql? you name it!: aligning sparql with rest and resource orientation. In *Proceedings of the 4th Workshop on Emerging Web Services Technology*, pages 39–43. ACM, 2009.

[133] Leland Wilkinson. *The grammar of graphics*. Springer, 2012.

[134] Michael Zur Muehlen, Jeffrey V Nickerson, and Keith D Swenson. Developing web services choreography standardsthe case of rest vs. soap. *Decision Support Systems*, 40(1):9–29, 2005.