UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS
MAESTRÍA EN INGENIERÍA DE SISTEMAS

SWAM: Un lenguaje para la programación de agentes móviles inteligentes en la Web Semántica

por
Cristian Maximiliano Mateos Diaz

Tesis sometida a evaluación como requisito parcial
para la obtención del grado de
Magister en Ingeniería de Sistemas

Dr. Marcelo Campo
Director
Dr. Alejandro Zunino
Co-director

Tandil, Septiembre de 2005

Resumen

La WWW está evolucionando hacia un medio abierto que no sólo es un enorme repositorio de contenido estático como páginas e imágenes, sino también un proveedor de servicios. Los Servicios Web posibilitan la construcción de nuevos tipos de aplicaciones que están caracterizadas por su habilidad de interactuar con funcionalidad publicada a lo largo de la Web a través de protocolos de comunicación estándares. Las recientes extensiones de los Servicios Web con descripciones semánticas apuntan a realizar una visión donde los programas utilicen la gran cantidad de recursos Web existentes de una forma completamente autónoma.

En este poderoso ambiente computacional, los agentes móviles inteligentes tendrán un rol fundamental dada su capacidad de inferir, aprender, moverse y actuar. Un agente móvil es una entidad de software capaz de controlar su propia ubicación dentro de una red de computadoras. Los agentes móviles constituyen un promisorio paradigma para el desarrollo de aplicaciones masivamente distribuidas, debido a que ofrecen ventajas en cuanto a escalabilidad, performance, tolerancia a fallas y flexibilidad con respecto a paradigmas existentes. En este sentido, muchos investigadores consideran que los agentes móviles cuentan con propiedades que los hacen adecuados para explotar el potencial de esta nueva Web.

En este trabajo se propone el lenguaje de programación SWAM (Semantic Web-Aware MoviLog). El objetivo de SWAM es proveer herramientas para la construcción de agentes móviles inteligentes que interactúan y viven dentro de la Web Semántica. A grandes rasgos, SWAM está materializado como una extensión de MoviLog, un lenguaje para la programación de agentes que ofrece soporte de movilidad reactiva y proactiva a través de un modelo de movilidad denominado MRF (Movilidad Reactiva por Fallas). Sin embargo, SWAM provee un modelo de ejecución de agentes que generaliza MRF, llamado MIG (Movilidad Inteligente Generalizada), con el fin de posibilitar la integración de agentes móviles con Servicios Web Semánticos.

La principal diferencia entre SWAM y otras plataformas para agentes móviles radica en su soporte de movilidad reactiva por fallas generalizado, que reduce el esfuerzo de desarrollo mediante la automatización de decisiones de movilidad y acceso a recursos, y la posibilidad de invocar Servicios Web Semánticos, lo que permite una efectiva integración de los agentes con la Web Semántica.

Palabras clave: Servicios Web, Web Semántica, Agentes Móviles, MIG, Programación Lógica, SWAM

Abstract

The WWW is evolving into an open medium that is not only a huge repository of static pages and data, but also a provider of Web Services, thus enabling the construction of new types of applications characterized by their ability to interact with functionality published across the Web through standard communication protocols. The recent extensions of Web Services with semantics aim at realizing a dream where programs use the vast amounts of Web-accessible resources in a fully automatized way.

In this rich computational environment, intelligent mobile agents will have a fundamental role due to their capacity to infer, learn, move and act. A mobile agent is a software entity able to control its location on a computer network. Mobile agents are a promissory paradigm for developing massively distributed applications since they provide advantages on scalability, performance, fault-tolerance and flexibility with respect to existing paradigms. In this sense, many researchers agree that mobile agents exhibit a number of properties that make them suitable for exploiting the potential of the forthcoming Web.

In this work, the SWAM (Semantic Web-Aware MoviLog) programming language is proposed. SWAM aims at providing tools for building intelligent mobile agents that interact and live within the Semantic Web. Basically, SWAM is materialized upon an extension of MoviLog, an agent programming language which offers support for reactive and proactive migration through a mobility model named RMF (Reactive Mobility by Failure). Nevertheless, SWAM provides an agent execution model called GIM (Generalized Intelligent Mobility) that generalizes RMF to allow the integration of mobile agents and Semantic Web Services.

The main difference between SWAM and other platforms for mobile agents is its generalized support for reactive mobility by failure, which reduces development effort by automatizing code mobility and resource access decisions, and its Semantic Web Services invocation and discovery support, which enables an effective integration of agents with the Semantic Web.

Keywords: Web Services, Semantic Web, Mobile Agents, GIM, Logic Programming, SWAM

Agradecimientos

A mis directores, el Dr. Marcelo Campo y el Dr. Alejandro Zunino, por su constante apoyo, valiosas sugerencias y paciente revisión del texto de este trabajo. A Marcelo debo agradecerle además por brindarme un lugar en la prestigiosa institución que dirige y apoyarme en más de una oportunidad para la obtención de una beca.

A mis compañeros del Instituto de Sistemas Tandil (ISISTAN), con quienes he compartido gran cantidad de charlas y muy buenos momentos.

A Andrea y a mi hija Bianca, por hacer cada día que finaliza mejor que el anterior.

A mis padres y mi hermana, por todo el afecto incondicional que me han sabido regalar a lo largo de todos estos años.

A la Universidad del Centro de la Provincia de Buenos Aires (UNCPBA) y la Comisión de Investigaciones Científicas (CIC) por haberme otorgado la Beca SAINI y la Beca de Estudio, respectivamente, las cuales proporcionaron el sustento económico que posibilitó la realización de esta tesis.

A todos, ¡muchas gracias!

Cristian Mateos Diaz

Índice general

Resumen iii

Abstract v

Agradecimientos vii

Índice de figuras xiv

Índice de tablas xv

Índice de algoritmos xvii

Glosario xix

1. Introducción 1

1.1. Motivación 2

1.2. La Tesis 4

1.3. Contribuciones 5

1.4. Organización de este trabajo 5

2. Conceptos Básicos 7

2.1. La Web Semántica 8

2.2. Representación de la información 9

2.2.1. XML 10

2.2.1.1. Parsers XML 12

2.2.1.2. Documentos bien formados y válidos 12

2.2.1.3. Espacios de nombres XML 13
2.2.2. RDF y RDFS 14
2.2.3. DAML+OIL 15
2.2.3.1. DAML-S 18
2.2.4. OWL 19
2.3. Servicios Web 20
2.3.1. El modelo de Servicios Web 22
2.3.2. SOAP 23
2.3.3. WSDL 25
2.3.4. UDDI 27
2.4. Agentes 29
2.4.1. Tipos de agentes 31
2.4.2. Agentes móviles 32
2.4.2.1. Ventajas 32
2.5. Agentes y Servicios Web 34
2.6. Conclusiones 36

3. Trabajos Relacionados **39**
3.1. Proyectos 39
3.1.1. Grid Computing 40
3.1.2. WSMF 42
3.1.3. Agentcities 43
3.2. Plataformas y lenguajes 44
3.2.1. ConGolog 44
3.2.2. IG-Jade-PKSlib 47
3.2.3. Jinni 50
3.2.4. MWS 52
3.2.5. MyCampus 55
3.2.6. CALMA 56
3.2.7. TuCSoN 58
3.3. Discusión 60
3.4. Conclusiones 62

4. Movilidad Reactiva por Fallas 65

4.1. MRF . . . 66

4.1.1. Modelo conceptual . . . 66

4.1.2. Brainlets . . . 67

4.1.3. Protocolos . . . 68

4.1.4. Soporte de ejecución de MRF . . . 69

4.1.4.1. PNS . . . 70

4.1.4.2. Agentes de Movilidad . . . 71

4.2. El lenguaje MoviLog . . . 71

4.3. La plataforma MoviLog . . . 72

4.3.1. El componente *Agent Manager* . . . 73

4.3.2. El componente *RMF Manager* . . . 73

4.3.3. El componente *Security Manager* . . . 74

4.3.4. El componente *Inter-Agent Communication Manager* . . . 74

4.3.5. El componente *Application Gateway* . . . 74

4.3.6. Brainlets . . . 74

4.4. Conclusiones . . . 75

5. SWAM (Semantic Web-Aware MoviLog) 77

5.1. MoviLog y MRF . . . 78

5.2. MIG: Movilidad Inteligente Generalizada . . . 79

5.2.1. Descripción de recursos . . . 80

5.2.2. Mecanismos para el acceso a recursos . . . 83

5.2.3. Clasificación de recursos . . . 85

5.3. Heurísticas para el acceso a recursos . . . 87

5.3.1. Profiling . . . 88

5.3.2. Programación de reglas . . . 90

5.4. Manejo de Servicios Web con información semántica . . . 91

5.4.1. Algoritmo de similitud semántica de Servicios Web . . . 95

5.4.2. Materialización en SWAM . . . 96

5.5. Conclusiones . . . 97

6. Ejemplos de aplicación 99

6.1. Agente para la creación de itinerarios turísticos . . . 99

6.2. Agente para la distribución de artículos científicos . . . 103

6.3. Conclusiones . . . 107

7. Diseño e implementación **109**

7.1. Arquitectura de SWAM 109

7.1.1. El componente *Protocol Container* 110

7.1.1.1. Protocolos 112

7.1.1.2. Clasificación de los recursos 115

7.1.2. El componente *Resource Access Manager* 117

7.1.3. El componente *Profiler Manager* 118

7.2. Arquitectura de invocación a Servicios Web 120

7.2.1. Interacción con WSIF 122

7.2.2. Traducción de tipos de datos 123

7.2.3. Deployment de Servicios Web 125

7.3. Conclusiones 128

8. Resultados Experimentales **129**

8.1. Descripción del dominio 130

8.2. Implementación utilizando IG-JADE-PKSlib 131

8.3. Implementación utilizando SWAM 134

8.4. Conclusiones 139

9. Conclusiones y Trabajos Futuros **143**

9.1. Contribuciones 144

9.2. Limitaciones 144

9.3. Trabajos futuros 145

9.3.1. Soporte para la construcción de procesos o *workflows* de Servicios Web . 145

9.3.2. Utilización de ASP para expresar y razonar a partir de las ontologías . . 146

9.3.3. Escalabilidad del registro de Servicios Web 147

9.3.4. Persistencia de protocolos 147

9.4. Consideraciones finales 148

Bibliografía **149**

Índice de figuras

2.1. Arquitectura conceptual de la Web Semántica 10
2.2. Un grafo dirigido rotulado para las triplas de la tabla 2.1 15
2.3. Principales clases de la ontología DAML-S 19
2.4. Componentes e interacciones de la arquitectura de Servicios Web 22
2.5. Vista del modelo de Servicios Web como una arquitectura de capas 23
2.6. Componentes y tipos de operaciones de una definición WSDL 26
2.7. Principales estructuras de datos de UDDI 28
2.8. Clasificación de agentes: Agencia, Inteligencia y Movilidad (Bradshaw, 1997) . 31
2.9. Agentes móviles: Reducción de la carga de red 33
2.10. Agentes móviles y desconexión de usuarios 34

3.1. Una vista general de una Grilla Computacional y la interacción entre sus entidades 40
3.2. Un ejemplo de un árbol de situaciones 45
3.3. Construcciones posibles en ConGolog 46
3.4. Un ejemplo de programa ConGolog 46
3.5. Componentes de un servicio móvil 53
3.6. Migración y clonación de un MWS 54
3.7. Vista general de la arquitectura de MyCampus 55
3.8. Vista general de la infraestructura de CALMA 58
3.9. Una representación del espacio de coordinación de TuCSoN integrado con Servicios Web 60

4.1. Movilidad Reactiva por Fallas: Vista general 67
4.2. Componentes de un Brainlet 68

4.3. La plataforma MoviLog 73
4.4. Ciclo de vida de un Brainlet 75

5.1. Acceso a recursos no locales 84
5.2. Clasificación de recursos accesibles por entidades móviles (Vigna, 1998) 86
5.3. Formas de interacción de un agente SWAM y un Servicio Web 86
5.4. Arquitectura del Matcher de Servicios Web de SWAM 93

6.1. Escenario de ejecución del agente planificador 100
6.2. Una ontología simple del concepto de *Documento* 104

7.1. Arquitectura de SWAM 110
7.2. Ejemplo de un árbol de recursos de un sitio SWAM 111
7.3. Vista general de la arquitectura de JNDI 112
7.4. La clase ResourceProtocol 115
7.5. Clasificación de recursos: clases principales 116
7.6. Diseño del componente *Resource Access Manager* 117
7.7. Diseño del componente *Profiler Manager* 119
7.8. Principales clases relacionadas con el diseño de los servicios de *profiling* 120
7.9. Vista general de la arquitectura de invocación a Servicios Web de SWAM . . . 121
7.10. La clase StubInvoker 123
7.11. Esquema de traducción de los argumentos de invocación a servicios 124
7.12. Las clases TypeMapperContainer, TypeMapper y TypeConverter 125
7.13. Diseño del componente *Package Manager* 127
7.14. Generación de *stubs*: la clase PackageCompiler 128

8.1. Gráfico de los resultados obtenidos para los problemas BPF, BMxF y BPMxF . 133
8.2. Gráficos de los tiempos obtenidos para la implementación SWAM 137
8.3. Mejora obtenida para el problema BBF ordenando las compañías en orden creciente según el costo de los vuelos 137
8.4. Tiempo empleado en la invocación de Servicios Web para el problema BPBF . 138

Índice de tablas

2.1. Una descripción RDF (triplas) acerca del creador de una página Web 14
2.2. Constructores de clases DAML+OIL 17
2.3. Axiomas provistos por DAML+OIL 18

3.1. Acciones *open* y *search* 49
3.2. Un ejemplo de una DSUR 50
3.3. Principales características de las plataformas analizadas 61

5.1. Ejemplos de recursos y métodos de acceso válidos asociados 84
5.2. Métricas del sistema soportadas por SWAM 88
5.4. Correspondencia entre OWL y Prolog 96

6.1. Ejemplos de instancias de conceptos y similaridad asociada con clases de la ontología 105

7.1. Correspondencia entre las estructuras Prolog y tipos de datos JAVA 124

8.1. Especificación de las acciones PKS para el dominio de viajes aéreos 132
8.2. Resultados para los problemas BPF, BMxF, BPMxF, BBF y BPBF de acuerdo a diferente cantidad de compañías conocidas por el agente ($t_{max} = 300$) 133
8.3. Performance de los agentes SWAM para los problemas BPF, BMxF, BPMxF, BBF y BPBF (segundos) 136
8.4. Mejoras obtenidas (en segundos y porcentaje del tiempo total) para el problema BBF 138

Índice de algoritmos

1. Similitud semántica entre Servicios Web 95
2. $listSimilarity(list1, list2)$: Similaridad entre dos listas de conceptos 96

Glosario

ACC Agent Coordination Context

AI Artificial Intelligence

API Application Program Interface

B2B Business-to-Business

BDI Beliefs-Desires-Intentions

BPEL4WS Business Process Execution Language for Web Services

CALMA Context-Aware Lightweight Mobile Agent

CGI Common Gateway Interface

CORBA Common Object Request Broker Arquitecture

DAML DARPA Agent Markup Language

DL Description Logics

DOM Document Object Model

DTD Document Type Definitions

DSUR Domain Specific Update Rules

FIPA Foundation for Intelligent Physical Agents

FTP File Transfer Protocol

GIS Geographical Information System

HTML Hyper Text Markup Language

HTN Hierarchical Task Networks

HTTP Hyper Text Transfer Protocol

IIOP Internet Inter-ORB Protocol

JADE Java Agent DEvelopment framework

JDBC Java Data Base Connectivity

Jinni Java INference engine and Networked Interactor

JNDI Java Naming Directory Interface

KB Knowledge Base

KQML Knowledge Query and Manipulation Language

LAN Local Area Network

MAS Multi-Agent System

MWS Mobile Web Services

NAICS North American Industry Classification System

OGSA Open Grid Services Architecture

OIL Ontology Inference Layer

OWL Ontology Web Language

P2P Peer-to-Peer

PC Program Committee

PCC Program Committee Chair

PDA Personal Digital Assistant

PKS Planning with Knowledge and Sensing

PNS Protocol Name Server

RDF Resource Description Framework

MRF Movilidad Reactiva por Fallas

RMI Remote Method Invocation

RPC Remote Procedure Call

SAX Simple API for XML

SMTP Simple Mail Transfer Protocol

SPI Service Provider Interface

SWAM Semantic Web-Aware MoviLog

SWWS Semantic Web enabled Web Services

SOAP Simple Object Access Protocol

UDDI Universal Description, Discovery and Integration

UML Unified Modeling Language

URL Uniform Resource Locator

VO Virtual Organizations

WAP Wireless Access Protocol

WSDL Web Service Definition Language

WSC Web Service Composition

WSFL Web Service Flow Language

WSMF Web Service Modelling Framework

WWG World Wide Grid

WWW World Wide Web

XML eXtensible Markup Language

XSD XML Schema Definition

Capítulo 1

Introducción

El desarrollo de la Web de nuestros días comenzó hace poco más de quince años atrás como una propuesta de proyecto (Berners-Lee, 1989). El primer resultado concreto de este proyecto lo constituyó la primer World Wide Web, un espacio de información con la mayor cantidad de documentos que jamás haya existido (Theilmann y Rothermel, 1998), capaz de ocultar la diversidad de software, hardware y arquitecturas de redes por entonces existentes. Este espacio de información fue diseñado para ser totalmente distribuido y sin un control o coordinación central. A diferencia de la mayoría de los sistemas de información que estaban basados en estructuras de árbol para organizar la información, la Web utilizó hipervínculos entre los diferentes documentos que podían ser localizados en cualquier lugar de la Internet (Berners-Lee, 1999). El mecanismo para navegar estos documentos fue ideado esencialmente para uso e interpretación por parte del humano (McIlraith et al., 2001), típicamente a través de la búsqueda y lectura de páginas mediante una aplicación especial, el *browser Web*.

Muchos investigadores imaginan a la Web del futuro como una comunidad global donde las personas y los agentes inteligentes interactúan y colaboran (Hendler, 2001). Los agentes inteligentes (o simplemente agentes) son entidades computacionales autónomas que están dirigidas por objetivos e inmersas en un entorno sobre el cual perciben y actúan. La visión mencionada corresponde a una Web que no sólo ofrece información, sino también *servicios* que llevan a cabo una tarea específica, y que son accedidos tanto por los usuarios a través de navegadores convencionales u otras interfaces, como también por parte de aplicaciones y agentes. En este sentido, los *Servicios Web* han surgido recientemente, y se han convertido en una pieza fundamental de la base tecnológica que materializará la Web que está por venir.

Básicamente, un Servicio Web (Curbera et al., 2001; Martin, 2001; Vaughan-Nichols, 2002; W3C Consortium, 2002) es cierta funcionalidad accesible vía Web, la cual puede ser vista como un conjunto de programas interactuando a través de una red sin la intervención humana. Los Servicios Web son una consecuencia natural de la evolución de la Web hacia un medio que facilita las interacciones complejas y sistemáticas entre aplicaciones (Curbera et al., 2001). Uno de los principales objetivos del modelo de Servicios Web es proveer una representación común de las interfaces de las aplicaciones que utilizan diferentes protocolos de comunicación y modelos de interacción. La tecnología actualmente propuesta para alcanzar este objetivo está basada en numerosos esfuerzos de estandarización centrados alrededor de XML (Extensible

Markup Language) (Bray et al., 1998). A grandes rasgos, XML es un lenguaje de *markup* para la representación de datos estructurados que extiende y formaliza al lenguaje HTML (Hyper Text Markup Language).

En los últimos años, la interacción entre aplicaciones a través de Servicios Web ha comenzado a tomar forma, mayormente en el contexto de las aplicaciones B2B (Business-to-Business) y de comercio electrónico. Por ejemplo, muchos sitios populares tales como Amazon[1], eBay[2] o Google[3] ofrecen Servicios Web que permiten a otras aplicaciones acceder a la misma información y funcionalidad que está disponible a los usuarios a través de páginas HTML. Sin embargo, para lograr una verdadera interoperabilidad automática entre los programas y los *recursos* - páginas y servicios - accesibles vía Web, es preciso que éstos últimos sean descritos de una manera no ambigua e interpretable por las computadoras. Así, la *Web Semántica* (Berners-Lee, 1999) constituye una extensión a la Web actual que propone asociar a cada recurso una descripción precisa que permite a las aplicaciones y agentes conocer el significado de la información incluida en las páginas y la funcionalidad de los Servicios Web ofrecidos. Consecuentemente, una gran variedad de lenguajes para describir la semántica de los recursos Web han sido propuestos, tales como RDF (Resource Description Framework) (Lassila y Swick, 1999), DAML+OIL (DARPA Agent Markup Language) (Horrocks, 2002) y OWL (Ontology Web Language) (Antoniou y van Harmelen, 2003).

1.1. Motivación

Existe un marcado consenso dentro de la comunidad científica que, en el contexto antes mencionado, los agentes inteligentes tendrán un rol fundamental (Jennings et al., 2000; Hendler, 2001; Huhns, 2003). Particularmente, en el escenario de la Web Semántica consistiendo de sitios que proveen contenido altamente dinámico y *servicios*, usuarios móviles, conexiones no confiables y pequeños dispositivos tales como PDAs (Personal Digital Assistant) y teléfonos celulares, los agentes *móviles* contarán con un papel protagónico. Un agente móvil es un programa que representa un usuario en una red de computadoras, y es capaz de migrar de forma autónoma entre los diferentes sitios para realizar alguna tarea en favor de dicho usuario (Tripathi et al., 2002). Esta característica es particularmente interesante cuando un agente hace uso esporádico de un recurso compartido valioso. Además, la eficiencia en el acceso a los recursos puede ser mejorada moviendo agentes a un sitio para consultar grandes repositorios en forma local y regresar con los resultados, evitando así múltiples interacciones con los datos a través de vínculos de red sujetos a demoras, interrupciones o caídas.

Los agentes móviles exhiben una serie de características que los hacen ideales para explotar el potencial de las redes actuales, debido a que cuentan con las propiedades de un agente convencional (reacción, percepción, deliberación, autonomía, etc.) más la *movilidad*, que es la capacidad de transportarse entre los diferentes sitios de una red (Fuggetta et al., 1998). Algunas de las ventajas más significativas que ofrece el uso de agentes móviles (Milojicic et al., 1999; Lange y Oshima, 1999) se listan a continuación:

[1] Amazon Bookstore, http://www.amazon.com
[2] eBay, http://developer.ebay.com/DevProgram/index.asp
[3] Google Search Engine, http://www.google.com

- **Soportan desconexión de los usuarios**: Muchos de los usuarios hoy en día son móviles, debido a que comienzan su trabajo en una oficina, para luego retomarlo en una *laptop* en una ubicación diferente. A menudo las tareas iniciadas por los usuarios deben continuar mientras el usuario está desconectado. En este sentido, puede delegarse esta responsabilidad a un agente móvil que se ocupará de las tareas de su dueño, incluso si éste está desconectado. Cuando el usuario reanude su conexión, el agente puede migrar desde el sitio actual hacia la ubicación del usuario para presentarle los resultados obtenidos.

- **Son de naturaleza heterogénea**: Las redes de computadoras son fundamentalmente heterogéneas, desde el punto de vista del software y el hardware que utilizan. Debido a que los agentes móviles son generalmente independientes del software y el hardware subyacente, y sólo dependen de las características de su ambiente de ejecución, proveen condiciones óptimas para la integración de sistemas distribuidos.

- **Son robustos y tolerantes a fallas**: La habilidad de reaccionar ante eventos y situaciones desfavorables que poseen los agentes móviles posibilita la construcción de sistemas robustos y tolerantes a fallas con mayor facilidad. Si un sitio está a punto de ser apagado, todos los agentes que están actualmente ejecutando en él pueden ser notificados y provistos de un cierto tiempo para abandonarlo. Los agentes pueden luego continuar con su ejecución en otro sitio.

A pesar de que hay un extenso conjunto de aplicaciones Web que pueden verse beneficiadas por el uso de agentes móviles (Kotz y Gray, 1999), el potencial que ha demostrado dicha tecnología se ha visto obstaculizado por las dificultades que presenta un agente móvil a la hora de interactuar con recursos Web (Hendler, 2001). Los agentes móviles son incapaces por sí solos de entender los conceptos inmersos en una página o Servicio Web; en consecuencia, el programador debe adicionar a cada agente el comportamiento necesario para extraer e interpretar la semántica asociada a cada recurso Web. Esta alternativa resulta poco flexible, y no promueve la reutilización de los agentes desarrollados, ya que el comportamiento destinado a interactuar con los recursos Web depende del formato en el cual la información está representada y almacenada (generalmente código HTML que contiene datos y presentación mezclados) y de las interfaces de acceso a los servicios, tales como CGI (Common Gateway Interface) o RMI (Remote Method Invocation). Bajo las condiciones expuestas, resulta clara la necesidad de crear una herramienta de desarrollo de agentes móviles inteligentes que de solución a los problemas mencionados, y que al mismo tiempo preserve los beneficios propios de la movilidad para la construcción de aplicaciones distribuidas.

Hasta ahora, los esfuerzos de investigación en el área de la Inteligencia Artificial han dado como resultado diversos modelos, arquitecturas de software y herramientas para el diseño e implementación de sistemas basados en agentes inteligentes. Es preciso notar, sin embargo, que aún existe una clara separación entre teoría y práctica. Particularmente, existe una falta de plataformas que aprovechen adecuadamente las características positivas ofrecidas por los agentes *móviles* inteligentes en la construcción de aplicaciones integradas a la Web Semántica (Dickinson y Wooldridge, 2003). Los enfoques que han sido propuestos hasta el momento para abordar este problema cuentan, a grandes rasgos, con dos limitaciones principales. Por un lado, presentan dificultades para procesar la información semántica asociada a los recursos Web, debido mayormente a la escasa adaptabilidad a la naturaleza evolutiva de los lenguajes

de descripción de metadatos, y a la falta de mecanismos de inferencia eficientes sobre éstos últimos. Por otra parte, los agentes considerados por dichos enfoques no poseen la propiedad de movilidad, lo que impacta negativamente tanto en la performance de las aplicaciones como en el aprovechamiento de los recursos de software y hardware presentes en la Web.

1.2. La Tesis

La hipótesis principal de este trabajo es que es posible lograr una efectiva integración entre los agentes móviles y las tecnologías de la Web Semántica que permita fundamentalmente aprovechar las características de los primeros para la construcción de aplicaciones Web automatizadas, capaces de procesar e interpretar el significado y las capacidades de los recursos. En esencia, no se trata de derivar un enfoque genérico que posibilite integrar cualquier modelo o plataforma de agentes móviles preexistente a la Web Semántica, sino que el estudio se acotará en principio a la plataforma MoviLog (Zunino et al., 2002).

MoviLog es una plataforma para la construcción de agentes móviles inteligentes para la WWW basados en Prolog. MoviLog cuenta con un novedoso mecanismo para administrar la movilidad llamado MRF (Movilidad Reactiva por Fallas). MRF se basa en la idea de que la movilidad es ortogonal al resto de los atributos o capacidades de un agente, tales como inteligencia (capacidades de razonamiento y aprendizaje) y agencia (autonomía y autoridad) (Bradshaw, 1997). Asumiendo esto, es posible pensar en una separación entre esas funcionalidades a nivel de implementación (Garcia et al., 2001). MRF explota esta separación permitiendo que el programador focalice su esfuerzo principalmente en la funcionalidad estacionaria propia de los agentes móviles y delegue aspectos relacionados con la movilidad en un sistema multiagente que forma parte de la plataforma de ejecución.

A pesar de las ventajas que ha demostrado MRF, la experiencia indica que la movilidad de código como única forma de acceder a los recursos de una red puede resultar poco eficiente (Zunino et al., 2005a). Considérese, por ejemplo, el caso de un agente que intenta acceder a un archivo remoto a través de un vínculo sujeto a fuertes limitaciones en cuanto a ancho de banda. Adicionalmente, supóngase que el tamaño del agente es mucho mayor que el asociado al archivo. Claramente, transferir el agente hacia la ubicación del recurso no es recomendable sino que, por el contrario, es conveniente migrar el recurso hacia el sitio actual donde el agente ejecuta. Más aún, en ciertas situaciones es preciso realizar una migración del flujo de control. Este es el caso, por ejemplo, de un agente de gran tamaño que intenta acceder a un recurso no transferible que acepta llamadas remotas, como una base de datos, una impresora o un Servicio Web.

Concretamente, esta tesis propone la generalización de MRF con el fin de adaptarlo a las necesidades de la Web Semántica. El resultado de esta adaptación es un modelo de ejecución de agentes móviles denominado MIG (Movilidad Inteligente Generalizada). MIG incorpora mecanismos extras a la movilidad de código propiamente dicha relacionados con la transferencia de recursos y de control, los cuales posibilitan llevar a cabo una utilización significativamente más eficiente de los recursos requeridos por los agentes. Por otra parte, MIG prescribe una infraestructura para la búsqueda e invocación de Servicios Web que contienen información semántica asociada. Para mostrar la utilidad práctica de esta infraestructura, se ha desarrollado el lenguaje SWAM (Semantic Web-Aware MoviLog) (Mateos et al., 2005b; Mateos

et al., 2005a), una herramienta que soporta el mecanismo MIG y que permite la construcción de aplicaciones basadas en agentes que interactúan con Servicios Web Semánticos.

Es preciso notar que SWAM ofrece una oportunidad excepcional para construir aplicaciones distribuidas basadas en agentes móviles inteligentes que acceden a información y recursos Web de forma autónoma. De esta forma, será posible automatizar las aplicaciones clásicas de comercio electrónico tales como tiendas, centros comerciales y subastas electrónicas, permitiendo la interacción autónoma y *eficiente* entre las entidades participantes en cada lado de una transacción, y con poco esfuerzo de desarrollo, debido a las facilidades dadas por MIG.

1.3. Contribuciones

En este trabajo se introduce una importante contribución al área de agentes móviles, por cuanto se propone una plataforma para la programación de agentes móviles inteligentes, cuyo modelo de ejecución ofrece:

- MIG: una generalización de MRF para lograr interacción automatizada de los agentes con Servicios Web Semánticos, aprovechando la movilidad, la transferencia de recursos y la invocación remota para hacer el mejor uso posible de los recursos de hardware y red al acceder a la información y los servicios que ofrece la Web. Como se expuso anteriormente, los Servicios Web permiten reutilizar funcionalidad accesible generalmente mediante páginas HTML; en consecuencia, es posible liberar a los usuarios de las tareas relacionadas con el uso de la Internet (búsqueda de vínculos útiles, negociación a través de formularios y páginas, etc.), generalmente tediosas y que consumen una gran cantidad de tiempo.
- El lenguaje SWAM, el cual permite el rápido prototipado y desarrollo de aplicaciones distribuidas que interactúan con información y recursos de una red computacional, gracias a las facilidades para la programación declarativa que ofrece Prolog y los beneficios propios de MIG en lo que a programación de código móvil se refiere. Estas características son fundamentales para facilitar la construcción de agentes móviles que utilizan recursos en el contexto de la Web Semántica.
- Uso eficiente de los recursos de software y hardware propios del sistema de agentes debido al soporte de políticas de SWAM que posibilita al programador especificar decisiones acerca de *cómo* y de *dónde* se obtienen los recursos Web. Adicionalmente, el hecho de utilizar Prolog como sintaxis base para la implementación de los agentes permite la especificación de políticas complejas.

1.4. Organización de este trabajo

El resto del trabajo está organizado como se describe a continuación. El capítulo 2 introduce los principales conceptos y tecnologías relacionados con la Web Semántica y los agentes inteligentes. Adicionalmente, se discute el papel de los agentes - en particular los agentes móviles inteligentes - en la realización de la visión de la Web Semántica.

En el capítulo 3 se describen y analizan los trabajos relacionados más representativos. El capítulo finaliza presentando una detallada comparación entre ellos y destacando sus principales problemas.

El capítulo 4 introduce MoviLog, un lenguaje para desarrollar agentes móviles que implementa el mecanismo de Movilidad Reactiva por Fallas. Este capítulo se centra principalmente en la descripción del modelo conceptual de MRF y los aspectos de diseño de MoviLog relacionados con el soporte de dicho mecanismo.

Luego, el capítulo 5 presenta SWAM (Semantic Web-Aware MoviLog), un lenguaje que soporta el mecanismo MIG (Movilidad Inteligente Generalizada) y que posibilita el desarrollo de agentes móviles basados en Prolog en el contexto de la Web Semántica.

El capítulo 6 analiza dos ejemplos de aplicaciones construidas con SWAM. Por un lado, se incluye una aplicación para elaborar paquetes turísticos entre ciudades de acuerdo a preferencias del usuario. Finalmente, se expone un agente móvil para la traducción de artículos científicos en base a Servicios Web Semánticos.

El capítulo 7 discute los aspectos más relevantes relacionados con el diseño e implementación de la plataforma SWAM. En particular, el capítulo se centra en la descripción detallada de los componentes y los mecanismos incluidos para permitir a los agentes interactuar con Servicios Web Semánticos.

En el capítulo 8 se detalla una aplicación concreta desarrollada con SWAM, y presenta una comparación experimental entre éste último y una plataforma para agentes que utilizan Servicios Web, denominada IG-Jade-PKSLib. Al término del capítulo, se describe la problemática en torno a la utilización de técnicas de planning en el contexto de la Web Semántica.

En el capítulo 9 se presenta un resumen de las contribuciones de la tesis, sus limitaciones y trabajos futuros.

Capítulo 2

Conceptos Básicos

Gran cantidad de investigadores imaginan a la Web en el futuro como una comunidad global donde las personas y los agentes inteligentes interactúan y colaboran (Hendler, 2001), compartiendo información y recursos. Desafortunadamente, la Web de hoy ha sido diseñada principalmente para uso e interpretación por parte del humano (McIlraith et al., 2001), generalmente a través de la lectura y navegación de páginas HTML (Hyper Text Markup Language) y el llenado de formularios en forma *online*. Sin embargo, existe una creciente necesidad de automatizar la interoperabilidad de las aplicaciones Web, especialmente el caso de las aplicaciones B2B (Business-to-Business) y de comercio electrónico. Hasta ahora, esta interoperación generalmente se ha realizado a través de programas específicos que localizan y extraen información Web e invocan servicios accesibles vía Web. Esta alternativa resulta poco flexible ya que depende del formato en el cual la información está representada (generalmente código HTML que contiene datos y presentación mezclados) y de las interfaces de acceso a los servicios, tales como CGI (Common Gateway Interface), RMI (Remote Method Invocation) o CORBA (Common Object Request Broker Architecture), entre otras. Con el propósito de lograr una verdadera interoperabilidad automática entre los programas y los recursos accesibles vía Web, las nuevas tecnologías relacionadas a la Web están apuntando a crear una *Web Semántica*, donde los recursos - información y servicios ofrecidos - sean descritos de una forma no ambigua e interpretable por las computadoras (Berners-Lee, 1999).

En los últimos años, la WWW ha ido evolucionando paulatinamente hacia la creación de una red global de *Servicios Web,* que permite a las aplicaciones publicar la funcionalidad que ofrecen a través de operaciones accesibles vía Web, y a su vez realizar la búsqueda e invocación de servicios implementados por otras aplicaciones. En el mundo ideal de los Servicios Web, cada aplicación puede razonar acerca de las capacidades y características ofrecidas por cada servicio, e incluso negociar con el proveedor del mismo. En este sentido, la tecnología de agentes, y en particular la de los agentes móviles, tendrá un rol fundamental (Hendler, 2001). Los agentes móviles ofrecen características que son cruciales para convertir los Servicios Web de una masa de funcionalidad e información que requiere la consulta y navegación manual por parte de los usuarios, hacia un conjunto dinámico de capacidades distribuidas destinadas a resolver automáticamente las tareas de dichos usuarios (Burg, 2002).

Este capítulo introducirá al lector en la problemática existente en torno a la Web Semántica. El mismo se estructura como se detalla a continuación. En la siguiente sección se introduce

el concepto de Web Semántica, es decir, una Web extendida capaz de manejar descripciones semánticas de los datos que almacena. Posteriormente, la sección 2.2 expone las alternativas existentes para la administración y representación de la semántica de los datos en la WWW. Luego, las secciones 2.3 y 2.4 describen los conceptos de Servicios Web y Agentes Inteligentes, respectivamente. Más tarde, la sección 2.5 discute el papel que juega la tecnología de agentes inteligentes en la automatización de tareas en la Web Semántica. Por último, la sección 2.6 contiene las conclusiones del capítulo.

2.1. La Web Semántica

La Web ha sido pensada como un espacio de información útil para la comunicación e interacción entre humanos, pero también para dar participación a las aplicaciones. Uno de los mayores obstáculos para lograr que las aplicaciones aprovechen la información Web ha sido el hecho de que la información está diseñada para uso humano. Incluso si la información mencionada ha sido derivada a partir de una base de datos con significado bien definido para las tablas y columnas, la estructura de los datos no es comprensible para una aplicación que utiliza recursos Web: a menudo, los datos están mezclados con código de presentación y visualización. En este sentido, la *Web Semántica* (Berners-Lee, 1999) está conformada por los lenguajes, modelos y tecnologías que permiten expresar la información en un formato procesable e interpretable por las computadoras.

La Web Semántica es una extensión de la Web actual en donde la información se encuentra almacenada en repositorios cuyo contenido está expresado de una forma accesible e interpretable en forma automática por las aplicaciones. La Web actual está mayormente conformada por documentos, páginas de contenido textual e imágenes diseñadas para ser interpretadas por el humano. La Web Semántica, por su parte, pretende expresar todo el contenido de acuerdo a la semántica precisa inherente a ellos.

Al igual que la WWW tiene servicios tales como motores de búsqueda, que retornan una colección de páginas Web, la Web Semántica también tendrá funcionalidad de búsqueda. La diferencia principal es que los motores actuales realizan búsquedas basadas en palabras claves (las páginas que contienen una palabra o expresión basada en palabras), lo que permite responder sólo a consultas del tipo “Qué páginas contienen el término X” o “Cuáles son las mejores páginas que incluyen el término X”. Los motores de búsqueda de la Web Semántica ofrecerán la posibilidad de llevar a cabo consultas estructuradas en base a *conceptos*, por ejemplo, “La temperatura actual de Miami es $?X$”. Adicionalmente, es preciso notar que, para el ejemplo señalado, una aplicación provista de un modelo conceptual preciso acerca de las definiciones propias de la meteorología puede obtener ventajas significativas en cuanto a la performance con que la consulta es resuelta. Finalmente, cabe destacar que el objetivo de la Web Semántica no es proveer resolución de consultas expresadas en lenguaje natural, sino adicionar semántica precisa a la información existente que permita navegar las relaciones entre los recursos y las propiedades de éstos.

2.2. Representación de la información

La Web Semántica es un lugar donde los datos pueden ser compartidos y procesados por los humanos, como así también por herramientas automáticas y aplicaciones para la recuperación de información. La clave para lograr esta automatización es describiendo la información de forma tal que sea comprendida por las aplicaciones, adicionando a la misma ciertos metadatos que expresen su significado, lo que se conoce como la *semántica* de la información.

Para que las aplicaciones puedan automáticamente procesar la información presente en la Web Semántica, es necesario acordar previamente una representación única y compartida, utilizando dicha representación para especificar el contenido de la información en base a su significado preciso. Una de las soluciones adecuadas para este problema lo constituye el uso de *ontologías*. Una ontología es una especificación formal de una conceptualización (Gruber, 1993), y provee el significado común de las definiciones de los términos o vocabulario dentro de un dominio particular (por ejemplo, medicina, informática, finanzas, etc.). En este sentido, las ontologías tendrán un rol crucial para lograr una interoperabilidad de aplicaciones a nivel semántico a través de la Web, ya que proporcionan una fuente compartida de términos minuciosamente definidos que puede ser usada para describir uniformemente la información almacenada en la Web (Horrocks, 2002).

A pesar de las ventajas que acarrea el uso de ontologías, la representación semántica de la información requiere contar con lenguajes de descripción de términos lo suficientemente expresivos como para describir la estructura de los conceptos, junto con las relaciones entre éstos. Debido a su flexibilidad, poder expresivo y habilidad de manipulación programática, XML (Extensible Markup Language) (Bray et al., 1998) es el lenguaje que se ha convertido en la base para la creación y evolución de los lenguajes de descripción semántica existentes en la actualidad. La primer contribución que XML hizo a la Web radicó en la provisión de un modelo para describir información separando el contenido de la presentación y, más importante, agregar significado a dicho contenido a través del uso de *metadatos* (Malik, 2002). Tales características positivas han motivado la extensión de XML para la creación de lenguajes de descripción de contenido Web, tales como RDF (Resource Description Framework) (Lassila y Swick, 1999). Básicamente, RDF permite describir recursos, asociarles propiedades y establecer relaciones entre ellos.

La arquitectura conceptual propuesta para la representación, procesamiento e intercambio de conocimiento en la Web Semántica (Berners-Lee et al., 2001) se presenta en la figura 2.1. Aquí, los lenguajes XML y RDF conforman los niveles de la arquitectura que permiten la descripción semántica de los datos en su forma más básica, a través del uso de tecnologías estándares para la representación de caracteres y la identificación de recursos Web mediante URIs (Uniform Resource Locator). Por encima de ellos, se encuentra el nivel de Ontologías, que provee un vocabulario común para cada dominio de aplicación.

Inmediatamente después del nivel de ontologías se sitúa el nivel de Lógica. Este es el punto donde las aserciones o reglas diseminadas a través de la Web (por ejemplo, reglas públicamente conocidas) pueden ser utilizadas para derivar nuevo conocimiento. Sin embargo, el problema aquí es que los sistemas deductivos actuales no son lo suficientemente interoperables. En este sentido, la arquitectura propone la existencia de un nivel de Inferencia o Pruebas, compuesto de un lenguaje universal para representar inferencias en un formato común. Las aplicaciones

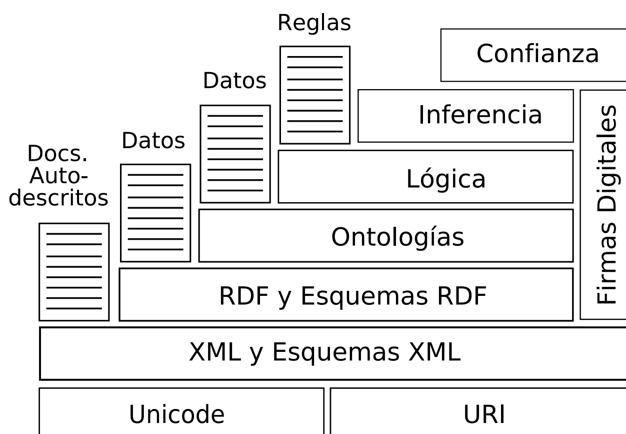


Figura 2.1: Arquitectura conceptual de la Web Semántica

y los sistemas pueden potencialmente autenticar digitalmente e intercambiar dichas pruebas a través del nivel de Confianza.

La siguiente subsección presenta XML, el lenguaje de descripción semántica que ha sido adoptado masivamente como formato común para la representación de información ofrecida por la Web Semántica.

2.2.1. XML

XML es un meta-lenguaje utilizado para crear documentos modulares y estructurados auto-descritos. Estos documentos son a menudo usados para el intercambio de datos entre sistemas heterogéneos. XML es increíblemente diverso y se relaciona con una gran cantidad de tecnologías incluyendo XPath y XQuery (direccionamiento y consulta sobre los elementos de un documento, respectivamente), y XSLT (transformación de documentos), por sólo nombrar algunas. Las primeras versiones de XML están fuertemente inspiradas en SGML (Standard Generalized Markup Language) (ANSI and ISO, 1985). Sin embargo, XML resultó ser más legible y, en varios sentidos, más simple que su predecesor. Actualmente, la importancia real de XML no está tanto en su innovatividad, sino en la aceptación masiva que ha tenido en el ambiente industrial como una forma común de describir e intercambiar datos y la generación de nuevos lenguajes utilizando los mecanismos de extensión que provee.

XML utiliza etiquetas para describir información. Una etiqueta encerrada entre “<” y “>” corresponde a un rótulo o descripción (por ejemplo, *street* en el código XML que sigue a continuación) del dato siguiente, que se denomina elemento. Por ejemplo, el elemento asociado a *street* en el código mencionado es “1111 AnyStreet”. Cada elemento se delimita con una etiqueta similar precedida por un caracter “/” para indicar el fin del elemento. En el ejemplo anterior, el elemento “1111 AnyStreet” está sucedido por una etiqueta de terminación “</street>”.

```
<?xml version="1.0" encoding="UTF-8"?>
<Order>
  <Customer>
    <name>John Doe</name>
```

```
    <street>1111 AnyStreet</street>
    <city>AnyTown</city>
    <state>GA</state>
    <zip>10000</zip>
  </Customer>
</Order>
```

La primera línea del ejemplo representa una convención utilizada para notificar al parser XML - es decir, el programa que procesa el documento XML - que la entrada es un documento XML con cierto formato para la codificación de sus caracteres. Por otra parte, el elemento *Customer* tiene varios elementos hijos: “John Doe”, “1111 AnyStreet”, “AnyTown”, “GA” y “10000”.

El lector habrá notado que una clara ventaja de XML es su naturaleza verborrágica - por consiguiente, legible por el humano - debido a que es un lenguaje meramente textual. Esta ventaja se transforma a veces en un inconveniente: debido a que XML es verborrágico, el tamaño de un documento puede ser excesivamente grande cuando describe una gran cantidad de datos. A pesar de este problema, XML cuenta con múltiples ventajas que es preciso mencionar:

- XML ES EXTENSIBLE. A diferencia de lenguajes de *markup* tales como HTML, que tienen un número fijo de etiquetas, XML permite al desarrollador definir un número arbitrario de etiquetas de acuerdo a las necesidades para resolver un problema. En el ejemplo mostrado, el documento representa una abstracción de un cliente e incluye los campos necesarios para describir sus datos personales.

- XML ES JERÁRQUICO. Los elementos de un documento XML pueden tener elementos subordinados dependientes de él. En el ejemplo anterior, el elemento *Customer* contiene varios elementos hijos que representan alguna característica o dato particular del cliente.

- XML ES MODULAR. Debido a que un documento puede referenciar a otros documentos, XML permite el diseño modular de documentos, facilitando así la reutilización de los mismos.

- XML NO INCLUYE TIPOS DE DATOS PREDEFINIDOS. Esta validación de datos es provista a través de documentos especiales, denominados DTD (Document Type Definitions) y XML Schemas, dos conceptos que serán discutidos más adelante con mayor detalle. A grandes rasgos, un DTD o un esquema XML define la estructura general y el formato de los datos presentes en un documento XML.

- XML ESTÁ SEPARADO COMPLETAMENTE DEL MECANISMO DE PRESENTACIÓN. Esta separación es producto de uno de los objetivos principales de XML, esto es, proveer facilidades para el intercambio de *datos*, lo que lo hace particularmente diferente del lenguaje HTML, que a menudo mezcla las definiciones de presentación con los datos propiamente dichos. De hecho, XML debe ser usado junto a otra tecnología (usualmente XSLT o Cascading Style Sheets) para presentar los datos incluidos en un documento.

- XML ES UN LENGUAJE DE PROGRAMACIÓN INDEPENDIENTE. Debido al hecho de que XML no es un lenguaje de programación en un sentido intrínseco, puede ser utilizado como un mecanismo común para en intercambio de datos entre lenguajes de programación existentes y, como se verá más adelante, como una forma común de conectar aplicaciones (por ejemplo, SOAP).

Algunos de los principales conceptos relacionados al lenguaje XML que son especialmente relevantes para la Web Semántica incluyen los parsers XML, los DTDs y los esquemas XML, y los espacios de nombres. A continuación se dará una breve explicación de cada uno de ellos.

2.2.1.1. Parsers XML

El procesamiento de un documento XML requiere el uso de algún parser XML, que descompone el documento XML en sus diferentes elementos individuales. Básicamente, hay dos categorías principales de parsers XML: Document Object Model (DOM) y Simple API for XML (SAX).

DOM es una API (Application Program Interface) independiente del lenguaje para el acceso y modificación de representaciones basadas en árboles n-arios de documentos HTML o XML. Los desarrolladores utilizan parsers DOM de un lenguaje específico para programáticamente construir y procesar documentos XML. Es preciso notar que los parsers DOM tienen dos defectos principales. Primero, el documento XML es almacenado por completo en memoria, lo que puede afectar la performance de la aplicación que lo procesa para los casos en que el documento XML es excesivamente grande. Segundo, dado que la API es independiente del lenguaje, DOM es bastante genérico, por lo que a menudo el procesamiento de un documento XML demanda más pasos que los requeridos al utilizar implementaciones de la API optimizadas para un lenguaje de programación particular.

SAX es un parser basado en eventos y puede ser utilizado sólo para la lectura del contenido de un documento XML. Un parser SAX opera en base a registración a eventos. El programador registra código o *handler*s que tratan determinados eventos, y que son invocados a medida que el documento XML es procesado. La principal ventaja de SAX sobre DOM es que no requiere que el documento se encuentre almacenado por completo en memoria, ya que el mismo es procesado como una sucesión de datos, invocando al mismo tiempo los *handlers* definidos por el desarrollador. Sin embargo, a pesar de que SAX es más simple y fácil de usar que DOM, presenta dos desventajas notorias. Por un lado, una vez que el documento ha sido leído, no existe una representación interna en memoria del mismo. Como consecuencia, cualquier procesamiento adicional requiere que el documento sea procesado nuevamente. Una alternativa para evitar este problema es que la aplicación mantenga en memoria estructuras *ad-hoc* de las partes del documento XML que le interesan, lo que requiere un esfuerzo de programación mayor. Por otra parte, SAX no permite la modificación de un documento XML, limitando su uso únicamente a su lectura.

En la siguiente subsección se discutirá brevemente las alternativas existentes para validación de documentos XML.

2.2.1.2. Documentos bien formados y válidos

Los documentos XML deben generarse conforme a ciertas reglas de creación antes de que puedan ser procesados y utilizados por las aplicaciones. Esto lleva a definir dos términos que son usualmente utilizados para describir el estado de un documento XML: bien formado y válido.

Un documento XML *bien formado* es aquel que sigue las convenciones sintácticas de XML y que puede ser procesado completamente por un parser XML. Si hay errores sintácticos en

alguna porción del documento (por ejemplo, una etiqueta que no contiene la correspondiente etiqueta de cierre), el parser rechaza entonces el documento completo.

Un documento XML *válido* es un documento bien formado que también puede ser verificado contra un DTD, que define las restricciones sobre cada elemento individual (orden establecido para los elementos, el rango de los valores, y así siguiendo). Un parser de validación es aquel que puede validar un documento XML contra un DTD o un esquema XML.

Básicamente, XML ofrece dos mecanismos para verificar la validez de un documento XML: los DTD y los esquemas XML. En primer lugar, un DTD es un documento externo que actúa como un *template* para la comparación de un documento XML. Un documento puede referenciar a un DTD, y el parser valida los elementos del documento basado en el DTD referenciado. Un DTD está capacitado para especificar el orden de los elementos, la frecuencia o cardinalidad con la cual dichos elementos pueden aparecer (por ejemplo, una factura contiene 0-n líneas), etc. A pesar de que es un concepto poderoso, los DTDs cuentan con algunas desventajas:

- Un DTD no posee sintaxis XML, lo que dificulta su aprendizaje y posterior utilización. A nivel aplicación, es necesaria la inclusión de parsers adicionales para el procesamiento de los documentos de tipo DTD.
- Un DTD no soporta tipos de datos, lo que trae como consecuencia la imposibilidad de restringir el valor de un elemento a un tipo de dato específico. Volviendo al ejemplo citado anteriormente, no es posible, por ejemplo, especificar que la cuenta de ítems de una factura debe ser un entero positivo.
- Un documento XML puede referenciar un único DTD, lo que limita la cantidad de validación posible y la reutilización de diferentes DTDs.
- Los DTDs surgieron antes de la estandarización de los *espacios de nombres* (ver sección 2.2.1.3) y, como consecuencia, no los soportan. Esto puede llevar a una gran cantidad de repeticiones en los nombres de las etiquetas, o en otras palabras, muchos DTD definidos para chequear las mismas estructuras XML. El concepto de espacio de nombre será expuesto en la siguiente subsección.

A raíz de estas limitaciones, las aplicaciones que procesan documentos XML incluyen una gran cantidad de funcionalidad de chequeo de errores. En este sentido, SOAP, una de las tecnologías más importantes relacionadas con los Servicios Web, prohíbe el uso de DTDs en las declaraciones de documentos. Por último, para solucionar los inconvenientes de los DTDs, el W3C (World Wide Web Consortium) ha desarrollado las especificaciones de los esquemas XML. Los esquemas XML tienen sintaxis XML, proveen soporte para los espacios de nombres, e incluyen un conjunto predefinido de tipos de datos (strings, binario, enteros positivos y negativos, fecha, hora, etc.) junto con diversos rangos y formatos para cada tipo. Adicionalmente, dichos esquemas permiten la creación de nuevos tipos simples o complejos siguiendo ciertas reglas de creación preestablecidas.

2.2.1.3. Espacios de nombres XML

Una aplicación real que utiliza XML consiste a menudo de docenas o cientos de documentos XML. A medida que varios documentos sean combinados o referenciados para crear nuevos

documentos, inevitablemente habrá etiquetas cuyos nombres estarán duplicados. Esto puede causar serios problemas de contenido o en la semántica de los datos, ya que cada elemento tiene que tener un nombre único. XML resuelve esta colisión de nombres a través del uso de *espacios de nombres*. Aquí, cada elemento es asociado con un espacio de nombre particular (prefijo), por lo que su nombre debe ser único sólo para el espacio dado y no en forma global. En la práctica, el prefijo usualmente representa el nombre de una compañía que se asocia con un URL (Uniform Resource Locator) particular. Así, el nombre de un elemento está compuesto de dos partes: el espacio de nombre y el nombre propiamente dicho, lo que reduce en gran forma la colisión de nombres de elementos. Considérese, por ejemplo, el caso de un sistema de archivos. Dentro de un directorio, el nombre de un archivo debe ser único. Sin embargo, pueden existir múltiples nombres idénticos a lo largo de todos los directorios. En este sentido, cada directorio provee un espacio de nombres para resolver los conflictos globales de nombres de archivos.

2.2.2. RDF y RDFS

Como se vio en el apartado anterior, XML es un lenguaje que provee una sintaxis para codificar y estructurar datos de diversa índole. En esta sección se describirá RDF (Resource Description Framework) (Lassila y Swick, 1999), un mecanismo basado en XML para expresar características semánticas acerca de los datos. Como su nombre lo indica, RDF no es un lenguaje sino un modelo para representar datos acerca de "cosas en la Web". A menudo, este tipo de datos que hablan acerca de los datos se conocen con el nombre de *metadatos*. En la terminología RDF, las "cosas" se corresponden con los *recursos* que pueden existir en la Web.

El modelo básico de RDF es extremadamente simple. Además de los recursos, el modelo incluye *propiedades* y *sentencias*. Una propiedad es un aspecto, característica o atributo específico que describe un recurso. Por su lado, una sentencia consiste de un recurso específico relacionado con una propiedad, más un valor particular de dicha propiedad para el recurso en cuestión. Este valor puede ser otro recurso o un valor *literal* (indivisible o atómico), usualmente de tipo texto plano. Adicionalmente, una descripción RDF es una lista de triplas, cada una consistiendo de un objeto (un recurso), un atributo (una propiedad), y un valor para dicho atributo. Por ejemplo, la tabla 2.1 muestra las triplas necesarias para especificar que una página Web específica fue creada por alguien cuyo nombre es "John" y cuyo número telefónico es "42-7782". Asimismo, puede graficarse un modelo RDF con un grafo dirigido rotulado. Para esto, cada recurso (nodo) se representa mediante un óvalo, y cada propiedad (arco) se representa mediante una flecha, graficando los valores literales como los rótulos de los arcos. La figura 2.2 muestra el grafo correspondiente a las triplas presentadas en la tabla 2.1.

Objeto	Atributo	Valor
http://www.w3c.org/	created_by	#anonymous_resource
#anonymous_resource	name	John
#anonymous_resource	phone	42-7782

Tabla 2.1: Una descripción RDF (triplas) acerca del creador de una página Web

Los ejemplos de notaciones anteriores revelan que RDF no se preocupa por la sintaxis de los datos, sino que solamente provee un modelo para la representación de metadatos. La lista

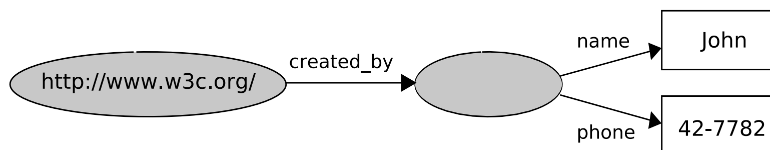


Figura 2.2: Un grafo dirigido rotulado para las triplas de la tabla 2.1

de triplas es una posible representación, tal como lo es el grafo rotulado, junto con otras representaciones sintácticas. Por supuesto, XML es un candidato obvio para una representación alternativa. En este sentido, la especificación del modelo de datos de RDF incluye una codificación basada en XML.

Al igual que XML, un modelo RDF no define (a priori) la semántica de un dominio de aplicación ni hace suposiciones acerca de un dominio de aplicación particular. RDF simplemente provee un mecanismo independiente del dominio para la descripción de metadatos. La definición de propiedades específicas de un dominio requiere de facilidades adicionales, tales como los esquemas RDF, o RDFS (Brickley y Guha, 2004). Básicamente, RDFS es un sistema simple de tipos para RDF. Provee un mecanismo para definir propiedades específicas de un dominio y clases de recursos descritas en base a dichas propiedades. Las primitivas de modelado básicas de RDFS son definiciones de uno de los siguientes tipos:

- *Clases* y *subclases*, que juntas permiten modelar jerarquías de clases de tamaño arbitrario,
- *Propiedades* y *subpropiedades*, para la construcción de jerarquía de propiedades,
- *Dominio* y *Rango*, que son sentencias destinadas a restringir las combinaciones posibles de clases y propiedades, y
- *Tipos*, que declaran un recurso como una instancia de una clase.

En base a las primitivas listadas se pueden construir esquemas RDF para un dominio específico. Para el ejemplo señalado más arriba, puede definirse un esquema que declara dos clases de recursos, Persona y Página Web, y dos propiedades, nombre y teléfono, ambas en el dominio Persona y el rango Literal. A su vez, este esquema podría ser utilizado para definir el recurso *http://www.w3c.org* como una instancia de Página Web, y un recurso anónimo como una instancia de Persona. Como resultado, estas declaraciones darían posibilidades de interpretación y validación de tipo semántica a los datos RDF.

RDFS es bastante simple comparado con otros lenguajes de representación de conocimiento que se analizarán más adelante. A su vez, RDFS aún no provee una semántica precisa o exacta. Sin embargo, esta omisión es parcialmente intencional, ya que el W3C se ha concentrado en trabajar en extensiones a RDFS un tanto más descriptivas en lo que a semántica se refiere.

2.2.3. DAML+OIL

El reconocimiento del rol clave que las ontologías juegan en la Web Semántica ha llevado al surgimiento de lenguajes para expresar la semántica de los datos Web, tales como RDF y

RDFS. Particularmente, RDFS es un lenguaje para la representación de ontologías y conocimiento, ya que habla acerca de *clases* y *propiedades* (relaciones binarias), restricciones de *rango* y *dominio* sobre dichas propiedades, y relaciones para definir *subclases* y *subpropiedades.* Sin embargo, RDFS es un lenguaje muy primitivo (lo descrito anteriormente representa prácticamente la totalidad de su funcionalidad), por lo que no ofrece el poder descriptivo que claramente se necesita para describir recursos con mayor detalle.

Con el propósito de contar con un lenguaje que permita la creación de ontologías más descriptivas, se ha desarrollado DAML+OIL (Horrocks, 2002), un lenguaje de ontologías Web muy expresivo. DAML+OIL es el resultado de la fusión entre DAML-ONT, un lenguaje desarrollado como parte del programa US DARPA Agent Markup Language (DAML) [1] y OIL (Ontology Inferece Layer) (Fensel et al., 2001), desarrollado en su mayoría por investigadores europeos [2]. DAML+OIL se materializa como una extensión a RDF, permitiendo definir relaciones más complejas entre las clases y propiedades modeladas en el dominio.

A grandes rasgos, DAML+OIL está diseñado para describir la *estructura* de un dominio mediante un enfoque similar al orientado a objetos, expresando dicha estructura en términos de *clases* y *propiedades*. Una ontología consiste de un conjunto de axiomas que establecen, por ejemplo, ciertas relaciones entre las clases o propiedades. Las aserciones del tipo "el recurso r es una instancia de la clase C" (o en terminología DAML+OIL, r es del tipo C) es delegado a las relaciones provistas por RDF, que son suficientes para dicha tarea.

Desde un punto de vista formal, DAML+OIL puede ser visto como equivalente a una Lógica de Descripción (DL) (ver (Baader et al., 2003)) muy descriptiva, cuyas ontologías se corresponden con terminologías DL (Tbox). Las Lógicas de Descripción son una familia de lenguajes de representación de conocimiento que ha sido extensamente estudiada en el área de Inteligencia Artificial durante las últimas dos décadas. Básicamente, dichos lenguajes se corresponden con formalismos para expresar conocimiento acerca de conceptos y jerarquías de éstos. En DAML+OIL, las clases pueden ser nombres - determinados por un URI - o diversas *expresiones*, creadas a partir de ciertos *constructores*, que pueden ser utilizados de manera anidada y de formas arbitrariamente complejas. En este caso, el poder expresivo del lenguaje o DL está determinado por el tipo de constructores de clases y el tipo de axiomas soportados.

La tabla 2.2 resume los constructores de clases provistos por DAML+OIL. Para expresar cada constructor, se ha utilizado la notación estándar de las Lógicas de Descripción, debido a que la sintaxis XML propia de DAML+OIL es significativamente más verborrágica y extensa. Por ejemplo, en sintaxis DAML+OIL, el conjunto expresado por $\geq 2 hasChild.Lawyer$ (todos los abogados que tienen al menos dos hijos) se escribiría como sigue:

```
<daml:Restriction daml:minCardinalityQ="2">
  <daml:onProperty rdf:resource="#hasChild"/>
  <daml:hasClass rdf:resource="#Lawyer"/>
</daml:Restriction>
```

El significado de los primeros tres constructores de la tabla anterior (*intersectionOf*, *unionOf* y *complementOf*) es relativamente claro: son simplemente operadores estándares que permiten que las clases sean creadas a partir de la intersección, unión o complementación de otras

[1] http://www.daml.org/
[2] http://www.ontoknowledge.org/oil

Constructor	Sintaxis DL	Ejemplo
intersectionOf	$Class_1 \sqcap ... \sqcap Class_n$	$Human \sqcap Male$
unionOf	$Class_1 \sqcup ... \sqcup Class_n$	$Doctor \sqcup Lawyer$
complementOf	$\neg Class$	$\neg Male$
oneOf	$x_1, ..., x_n$	$\{john, mary\}$
toClass	$\forall Property.Class$	$\forall hasChild.Doctor$
hasClass	$\exists Property.Class$	$\exists hasChild.Lawyer$
hasValue	$\exists Property.x$	$\exists citizenOf.\{USA\}$
minCardinalityQ	$\geq nProperty.Class$	$\geq 2hasChild.Lawyer$
maxCardinalityQ	$\leq nProperty.Class$	$\leq 1hasChild.Male$
cardinalityQ	$= nProperty.Class$	$= 1hasParent.Female$

Tabla 2.2: Constructores de clases DAML+OIL

clases. El constructor *oneOf* permite la definición de las clases de forma existencial, es decir, enumerando sus miembros.

Los constructores *toClass* y *hasClass* expresan restricciones sobre la relación entre clases y propiedades existentes. Más específicamente, $\forall P.C$ representa las clases cuyas instancias se relacionan a través de la propiedad P sólo con instancias de la clase C, mientras que $\exists P.C$ son las clases donde todas sus instancias están relacionadas a través de P con al menos una instancia de tipo C. El constructor *hasValue* es una combinación entre *hasClass* y *oneOf*.

Los constructores *minCardinalityQ*, *maxCardinalityQ* y *cardinalityQ* (conocidos en las DLs como restricciones de número) son generalizaciones de los constructores *toClass* y *hasClass*. La clase $\geq nP.C$ ($\leq nP.C, = nP.C$) es la clase cuyas instancias están relacionadas vía la propiedad P con al menos (a lo sumo, exactamente) n instancias *diferentes* de recursos de tipo C. El énfasis puesto aquí en la palabra "diferentes" radica en que no existe la suposición de que los nombres de recursos sean únicos, ya que, en DAML+OIL, es posible que varios URIs referencien al mismo recurso.

Otro aspecto de DAML+OIL que lo hace particularmente expresivo es el conjunto de axiomas que provee. La tabla 2.3 lista los axiomas soportados por el lenguaje. Estos axiomas permiten expresar relaciones complejas con respecto a clases o propiedades, tales como la disyunción de clases, la equivalencia o no de recursos, y varias propiedades acerca de las propiedades (unicidad, no ambigüedad, transitividad, etc.).

Una característica clave de DAML+OIL es que los axiomas *subClassOf* y *sameClassAs* pueden ser aplicados a expresiones de clases de complejidad arbitraria. Esto provee, en consecuencia, un poder expresivo mucho mayor al que ofrecen aquellos lenguajes cuyos axiomas están invariablemente restringidos al formato en el cual la parte izquierda es un nombre atómico, existe un sólo axioma por nombre, y no existen definiciones cíclicas (la clase ubicada en la parte derecha del axioma no puede referenciar, ya sea directa o indirectamente, el nombre de una clase que figura a la izquierda de algún axioma).

Una consecuencia del poder expresivo señalado es que todas los axiomas que hablan sobre clases o elementos específicos del dominio, al igual que los axiomas *uniqueProperty* y *unambiguousProperty*, pueden ser reducidos a los axiomas *subClassOf* y *sameClassAs*. De hecho, *sameClassAs* puede ser reducido a *subClassOf*, dado que un axioma de tipo $A \equiv B$ es equi-

Axioma	Sintaxis DL	Ejemplo
subClassOf	$Class_1 \sqsubseteq Class_2$	$Human \sqsubseteq Animal \sqcap Biped$
sameClassAs	$Class_1 \equiv Class_2$	$Man \equiv Human \sqcap Male$
subPropertyOf	$Property_1 \sqsubseteq Property_2$	$hasDaughter \equiv hasChild$
samePropertyAs	$Property_1 \equiv Property_2$	$cost \equiv price$
disjointWith	$Class_1 \sqsubseteq \neg Class_2$	$Male \sqsubseteq \neg Female$
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$	$\{President\, Bush\} \equiv \{G.W.Bush\}$
differentIndividualFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	$\{john\} \sqsubseteq \neg\{peter\}$
inverseOf	$Property_1 \equiv Property_2^-$	$hasChild \equiv hasParent^-$
transitiveProperty	$Property^+ \sqsubseteq Property$	$ancestor^+ \sqsubseteq ancestor$
uniqueProperty	$T \sqsubseteq \leq 1 Property$	$T \sqsubseteq \leq 1 hasMother$
unambigousProperty	$T \sqsubseteq \leq 1 Property^-$	$T \sqsubseteq \leq 1 isMotherOf^-$

Tabla 2.3: Axiomas provistos por DAML+OIL

valente a un par de axiomas de *subClassOf*, $A \sqsubseteq B$ y $A \sqsupseteq B$.

2.2.3.1. DAML-S

Un problema inherente del uso de lenguajes de descripción de ontologías es la necesidad de acordar previamente la base de conceptos particular que será utilizada. Por ejemplo, si dos aplicaciones cualesquiera pretenden interactuar en base a mensajes con contenido semántico, es necesario que dichas aplicaciones acuerden con anterioridad la base compartida que contiene las definiciones específicas para los conceptos involucrados en cada mensaje. Similarmente, la interacción entre aplicaciones en el contexto de la Web Semántica requiere que los conceptos involucrados en cada transacción estén definidos en una ontología común o compartida. Este problema ha motivado la creación de una ontología estándar de servicios para la Web Semántica denominada DAML-S (Burstein et al., 2002).

Como se vio en la sección previa, DAML+OIL define los conceptos y las relaciones entre éstos en términos de *clases* y *propiedades*. Por su parte, DAML-S define un conjunto estándar de clases y propiedades específicas para la descripción de Servicios Web. Particularmente, la clase SERVICE representa la clase raíz de la ontología DAML-S. Las propiedades de un servicio a este nivel son muy generales. Similarmente, las clases que conforman los conceptos básicos de la ontología son significativamente abstractas, sin establecer reglas en cuanto a cómo estructurar o categorizar un conjunto de servicios particular. Sin embargo, DAML-S espera que esta categorización se realice de acuerdo a la funcionalidad y dominio de cada servicio, y a las necesidades de las aplicaciones de comercio electrónico en general. Por ejemplo, podría pensarse en crear una clase genérica, denominada B2C-TRANSACTION, que abarque los servicios de compras de artículos en sitios Web de ventas al por menor, consultar el estado de una compra, establecimiento y mantenimiento de cuentas en dichos sitios, etc.

La ontología de servicios definida por DAML-S provee esencialmente tres tipos de conocimiento acerca de un servicio determinado, como muestra la figura 2.3. En primer lugar, los servicios se relacionan con una clase genérica llamada SERVICEPROFILE, que describe las capacidades y los parámetros del mismo. El SERVICEPROFILE provee información útil que puede ser utilizada por aplicaciones y agentes para determinar si el servicio asociado cumple

con sus necesidades, y si satisface ciertas restricciones tales como seguridad, de ubicación, requerimientos de calidad, etc. En segundo lugar, y contraria a ServiceProfile, la clase ServiceModel provee la información necesaria para hacer uso del servicio. Más específicamente, dicha clase describe la forma de operación del servicio, especificando el *workflow* o proceso asociado al mismo y sus posibles caminos de ejecución. Por último, todo servicio cuenta con cierta información asociada (*grounding*) que contiene los detalles relativos a cómo interoperar con el servicio, modelada través de la clase ServiceGrounding. Típicamente, la clase anterior especifica el protocolo de comunicación, el formato de los mensajes y demás detalles de bajo nivel (direcciones de red, números de puertos, etc.) para contactar e interactuar con el servicio en cuestión.

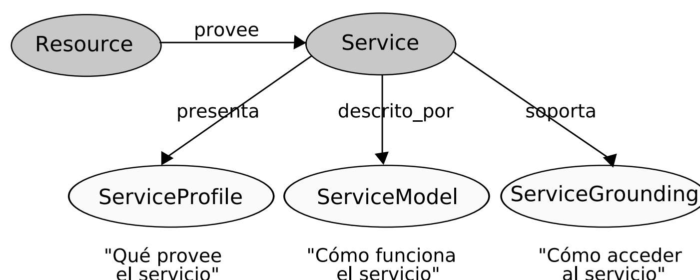


Figura 2.3: Principales clases de la ontología DAML-S

2.2.4. OWL

Recientemente, el W3C ha propuesto la especificación de un nuevo lenguaje de descripción de ontologías Web, denominado OWL (Ontology Web Language). OWL (Antoniou y van Harmelen, 2003) es utilizado para representar explícitamente el significado de los términos de un vocabulario dado y las relaciones entre dichos términos. OWL ofrece mayores facilidades para expresar semántica que RDF y RDFS, e incluso su antecesor, DAML+OIL, a través de la provisión de construcciones adicionales junto con una semántica formal bien definida. Básicamente, OWL agrega vocabulario adicional para la descripción de clases y propiedades, tales como relaciones de igualdad o desigualdad, mayores posibilidades de descripción de las características de las propiedades (por ejemplo, simetría), enumeración explícita de clases, por nombrar sólo algunas. Para citar un ejemplo concreto, OWL permite especificar - a diferencia de DAML+OIL - restricciones de número en el constructor de clases *hasClass;* considérese por ejemplo la definición de la clase de personas que tienen al menos dos amigos de origen italiano ($\exists^{\geq 2} hasFriend.Italian$).

OWL provee tres sublenguajes, cada uno con un diferente nivel de expresividad, diseñados para el uso por parte de comunidades de desarrolladores y usuarios específicas:

- *OWL Lite*: Está creado para aquellos usuarios que necesitan modelar taxonomías o jerarquías de clasificación sencillas con restricciones simples. Por ejemplo, el lenguaje permite la definición de restricciones de cardinalidad, pero limitando los valores de cardinalidad a 0 ó 1. OWL Lite tiene a su vez una menor complejidad formal que el sublenguaje OWL DL.

- *OWL DL*: Este lenguaje está destinado a los usuarios que pretenden el máximo poder expresivo de OWL, manteniendo al mismo tiempo la completitud computacional (todas las inferencias son computables) y decidibilidad (todas las inferencias finalizarán en una cantidad finita de tiempo). OWL DL incluye las construcciones del lenguaje OWL, pero éstas pueden ser usadas de acuerdo a ciertas restricciones (por ejemplo, a pesar de que una clase puede ser subclases de muchas clases, una clase no puede ser una instancia de más de una clase). El nombre OWL DL está dado debido a su correspondencia con las Lógicas de Descripción mencionadas en secciones previas.

- *OWL Full*: El sublenguaje OWL Full está pensado para usuarios que desean utilizar el máximo poder expresivo de OWL junto con la libertad sintáctica heredada de RDF, sin garantías computacionales de ningún tipo. Por ejemplo, una clase puede ser simultáneamente tratada como un conjunto de elementos (instancias) y como un elemento en sí. OWL Full permite a una ontología aumentar el significado del vocabulario predefinido de OWL.

Los desarrolladores que utilizan OWL deben adoptar el sublenguaje que mejor satisface sus necesidades. La elección entre OWL Lite y OWL DL depende del grado de expresividad requerido. La elección entre OWL DL y OWL Full depende mayormente de las facilidades de meta-modelamiento de RDFS que el usuario pretenda utilizar (por ejemplo, definir clases de clases, o asociar propiedades a las clases).

Al igual que DAML+OIL, la aplicación de OWL en el contexto de la Web Semántica está condicionada por la existencia de una ontología estándar compartida por las aplicaciones que interaccionan. En este sentido, una gran cantidad de investigadores en varias organizaciones han reunido sus esfuerzos para la creación de OWL-S (Martin et al., 2004). OWL-S es una ontología para la descripción de Servicios Web que permite a las aplicaciones y agentes inteligentes a automáticamente buscar, invocar, componer y monitorear la ejecución de servicios provistos a lo largo de la Web. Básicamente, OWL-S refina la ontología DAML-S, respetando en general el modelo de clases de ésta última.

2.3. Servicios Web

Las definiciones existentes del concepto de Servicio Web son tan amplias y abiertas como lo es el objetivo perseguido por la investigación relacionada a dicha tecnología. Por ejemplo, (Vaughan-Nichols, 2002) asocia a los Servicios Web con programas y dispositivos accesibles vía Web, que pueden ser vistos como programas que interactúan a través de una red de computadoras, sin intervención humana durante la transacción. Una definición similar puede encontrarse en (Curbera et al., 2001), donde define a un Servicio Web como una aplicación que es capaz de interactuar con otras aplicaciones utilizando protocolos Web estándares sobre interfaces bien definidas, y que es descrita utilizando un lenguaje de descripción funcional, también estándar. Por su parte, (Martin, 2001) establece que un Servicio Web es una aplicación modular, autocontenida y autodescrita, que puede ser publicada, encontrada y utilizada a través de la Web.

Una definición un tanto más formal del concepto de Servicio Web (W3C Consortium, 2002) es la publicada por el W3C, un organismo dedicado a la especificación y desarrollo de estándares para la Web Semántica. Aquí, se establece que un Servicio Web es una aplicación de

software identificada por un URI (Uniform Resource Identifier), cuyas interfaces y *bindings* pueden ser definidas, descritas y descubiertas por artefactos XML y soportan interacciones directas con otras aplicaciones mediante la utilización de mensajes XML vía protocolos de Internet existentes. En este caso, la definición evidencia un fuerte apoyo a la idea de basar la infraestructura de soporte de Servicios Web en el lenguaje XML. En este sentido, la W3C ha desarrollado diversas especificaciones de lenguajes que extienden el modelo formal de XML y que se han convertido en estándares para la utilización y descripción de Servicios Web.

En general, todas las definiciones anteriores coinciden en que los Servicios Web son aplicaciones modulares que proveen una funcionalidad determinada, y que pueden ser accedidas a través de una red (generalmente Internet) utilizando protocolos estándares preexistentes. De acuerdo a esta noción, son varias las actividades que están relacionadas al desarrollo y utilización de un Servicio Web, como se resumen a continuación (Pilioura y Tsalgatidou, 2001):

- *Desarrollo* de acuerdo a la funcionalidad que ofrece el servicio, lo que involucra la implementación del servicio en algún lenguaje de programación específico.

- *Descripción* abstracta de la funcionalidad (interfaz) provista por el servicio para posibilitar a los potenciales clientes, a través de dicha descripción, a interactuar con el mismo. Típicamente, el lenguaje utilizado para este fin es WSDL (Web Service Description Language), que posee una notación estándar basada en XML para la descripción de Servicios Web. WSDL permite expresar todos los detalles necesarios para interactuar con un servicio, incluyendo el formato de los mensajes intercambiados, los protocolos de transporte utilizados y la ubicación real del servicio.

- *Publicación* mediante la registración de las descripciones anteriores en un registro de público conocimiento. El mecanismo más difundido en la actualidad para la materialización de registros de servicios lo constituye UDDI (Universal Description, Discovery and Integration). Los servicios publicados pueden ser encontrados enviando solicitudes a ciertos registros UDDI, recibiendo los detalles de los servicios existentes que se corresponden con los argumentos de las consultas.

- *Invocación* a través de la red utilizando información de ubicación del servicio presente en su descripción abstracta. En la mayoría de los casos, la invocación de los servicios se efectúa valiéndose de protocolos de transporte estándares, como por ejemplo HTTP (Hyper Text Transfer Protocol). Uno de los protocolos de invocación a Servicios Web más utilizado actualmente es SOAP (Simple Object Access Protocol), un protocolo basado en el lenguaje XML.

- *Composición* del servicio con otros servicios existentes para formar nuevos servicios y aplicaciones más complejas. Un problema similar a esto es el que se conoce como el problema de la Composición de Servicios Web (Web Service Composition o WSC), que se encarga de componer automáticamente Servicios Web simples para proveer funcionalidad inexistente. Hasta ahora, este problema ha sido atacado mediante el desarrollo de lenguajes para la especificación de la forma en la cual determinados servicios pueden ser compuestos, tales como WSFL (Web Services Flow Language) (Leymann, 2001) y XLANG (Thatte, 2001). Ambos lenguajes usan un modelo de flujos como mecanismo básico de composición.

El modelo de Servicios Web está basado en diversos esfuerzos de estandarización alrededor del lenguaje de descripción XML. XML extiende y formaliza el modelo del lenguaje HTML, y provee medios para representar datos de una manera estructurada. Como tal, XML se ha convertido en el sucesor natural de HTML para la representación de información en Internet, y es un estándar sobre el cual se están construyendo las tecnologías Web asociadas al modelo que se describe a continuación.

2.3.1. El modelo de Servicios Web

La arquitectura básica de un ambiente orientado a Servicios Web (Kreger, 2001) está basada en la interacción de tres componentes: Proveedor de Servicios, Registro de Servicios y Cliente. Las interacciones, por su parte, involucran la publicación y búsqueda de servicios, y la invocación o *bind* para interactuar con un servicio ubicado en un punto específico de una red. La figura 2.4 ilustra la arquitectura mencionada.

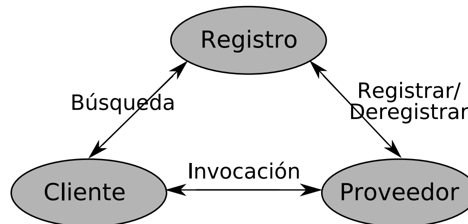


Figura 2.4: Componentes e interacciones de la arquitectura de Servicios Web

Los tres componentes principales de la arquitectura son:

- *Proveedor de Servicios*: Este componente ofrece servicios y los hace disponibles a los clientes a través de la Internet. Más específicamente, el Proveedor de Servicios es la parte de la arquitectura que provee aplicaciones o módulos de software que implementan la funcionalidad que los servicios que ofrecen. Desde una perspectiva de negocio, un Proveedor de Servicios es el dueño de un determinado servicio. Desde el punto de vista arquitectónico, es la plataforma que contiene la implementación de determinados Servicios Web.

- *Cliente*: Un Cliente es el componente que solicita servicios para satisfacer una tarea o funcionalidad determinada. El Cliente puede ser un usuario humano accediendo a servicios a través de un navegador, o también puede tratarse de una aplicación u otro Servicio Web. El Cliente encuentra los servicios necesarios a través del Registro de Servicios, e interactúa con cada uno de ellos a través del Proveedor de Servicios. Desde el punto de vista de un proceso de negocio, el Cliente es la entidad que requiere satisfacer una función para la que no está preparado. Desde una perspectiva arquitectónica, es una aplicación que busca e invoca un servicio.

- *Registro de Servicios*: El Registro de Servicios (o *Broker*) es un directorio de servicios centralizado en forma lógica. El Registro provee un lugar central donde los desarrolladores pueden publicar nuevos servicios o encontrar los ya existentes. Se comporta de forma similar a las páginas amarillas de una guía telefónica, es decir, un lugar centralizado donde las compañías pueden publicar sus servicios, y los clientes pueden encontrarlos.

Una segunda forma de ver el modelo descrito es como una arquitectura de capas (Shaw y Garlan, 1996), donde cada nivel incluye los distintos protocolos involucrados en la interacción con los Servicios Web, como se muestra en la figura 2.5. Dicha arquitectura está aún evolucionando, pero actualmente está compuesta de cuatro niveles principales:

- *Transporte*: Es el nivel responsable de transportar los mensajes entre las aplicaciones. Se compone de protocolos Web estándares existentes, tales como HTTP, FTP (File Transfer Protocol) o SMTP (Simple Mail Transfer Protocol), entre otros.
- *Mensajería XML*: Es el encargado de codificar los mensajes en un formato XML común entre las aplicaciones y los servicios. Actualmente, el protocolo de comunicación más utilizado para este fin es SOAP, que provee mecanismos para la invocación remota de métodos basados en mensajes estructurados expresados en XML, denominado XML-RPC (XML Remote Procedure Call).
- *Descripción*: Representa el nivel encargado de la descripción de la interfaz pública que posee un Servicio Web. El lenguaje más difundido para describir y representar la interfaz de un Servicio Web es WSDL, un lenguaje simple basado en XML que es usualmente utilizado en conjunto con el protocolo de comunicación SOAP.
- *Descubrimiento*: Es el nivel responsable de la centralización de los servicios mediante la utilización de un registro común, y de proveer funcionalidad básica para la publicación y búsqueda de los mismos. El mecanismo más difundido para este fin es UDDI.

Descubrimiento	UDDI
Descripción	WSDL
Mensajería XML	SOAP (XML-RPC)
Transporte	HTTP, FTP, SMTP, ...

Figura 2.5: Vista del modelo de Servicios Web como una arquitectura de capas

Paralelo a la evolución de la tecnología de los Servicios Web, han surgido diversas propuestas para agregar nueva funcionalidad en forma de capas a la arquitectura anterior, incluyendo nuevos lenguajes y tecnologías. Típicamente, estas propuestas apuntan a enriquecer el modelo anterior con manejo de ontologías y semántica, esquemas transaccionales, seguridad o composición de Servicios Web (Bussler et al., 2002; Sollazzo et al., 2002). Sin embargo, las siguientes secciones se concentrarán únicamente en las tecnologías estándares incluidas en la arquitectura básica expuesta anteriormente.

2.3.2. SOAP

El éxito de la Web de hoy en día se debe, en parte, a la estandarización de los mecanismos de intercambio de información, manifestada a través del uso masivo de ciertos protocolos tales como HTTP y FTP. El modelo de Servicios Web intenta replicar este patrón de éxito confiando el intercambio de información a un protocolo simple basado en XML, que en la mayoría de los casos es utilizado sobre protocolos Web de comunicación existentes. Esto minimiza la cantidad

de infraestructura adicional que necesita ser desarrollada, y asegura la interoperabilidad de aplicaciones a lo largo de Internet en el nivel de transporte.

Debido a que está basado en XML y a su masiva adopción, SOAP (Simple Object Access Protocol) (W3C Consortium, 2000) se está convirtiendo en el lenguaje común de las aplicaciones que interactúan vía Servicios Web. SOAP es un protocolo de mensajes XML cuyo núcleo es extremadamente simple, proporcionando sólo algunas convenciones en cuanto a cómo estructurar información en un mensaje XML. SOAP es independiente del protocolo de transporte subyacente, razón por la cual los mensajes SOAP pueden ser enviados a través de protocolos de transporte arbitrarios, tales como HTTP, IIOP (Internet Inter-ORB Protocol), WAP (Wireless Access Protocol), etc. Dado que aún se encuentra en etapa de desarrollo, existen muchos requerimientos que SOAP no tiene en cuenta todavía, tal es el caso de mecanismos generales de seguridad o el soporte de un modelo de transacciones unificado.

Cada mensaje SOAP se estructura como un documento XML, compuesto por una sección denominada sobre (*envelope*) que está compuesto por dos partes fundamentales: el encabezado (*header*) y el cuerpo (*body*). Por un lado, el encabezado provee mecanismos para extender características básicas de interoperación de los mensajes SOAP, tales como la codificación de caracteres utilizada. Por otra parte, el cuerpo permite la transmisión de información a un receptor determinado. Adicionalmente, ésta parte del mensaje puede ser utilizada para almacenar tanto la respuesta del receptor, como así también posibles fallas (*faults*) surgidas a raíz del envío del mensaje.

El siguiente ejemplo muestra un mensaje de notificación expresado en SOAP. El mensaje contiene dos estructuras definidas por la aplicación, es decir, agregadas a la especificación SOAP: un nombre local llamado “alertControl” en el encabezado, y un nombre local denominado “alert” en el cuerpo del mensaje. En general, los encabezados SOAP contienen información que puede ser de utilidad a los intermediarios y al destinatario del mensaje. En este ejemplo, un intermediario podría priorizar la entrega de un mensaje basado en la prioridad e información de expiración presente en el encabezado. Por otra parte, el cuerpo del mensaje contiene el mensaje de alerta propiamente dicho.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2 pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Como puede observarse, el documento XML declara tres espacios de nombres: *env*, *m* y *n*. El primero de ellos define la estructura de los nodos XML que representan los elementos básicos definidos por SOAP. Por otra parte, los espacios de nombres *m* y *n* incluyen las definiciones de los nodos del documento que representan los elementos propios de la aplicación.

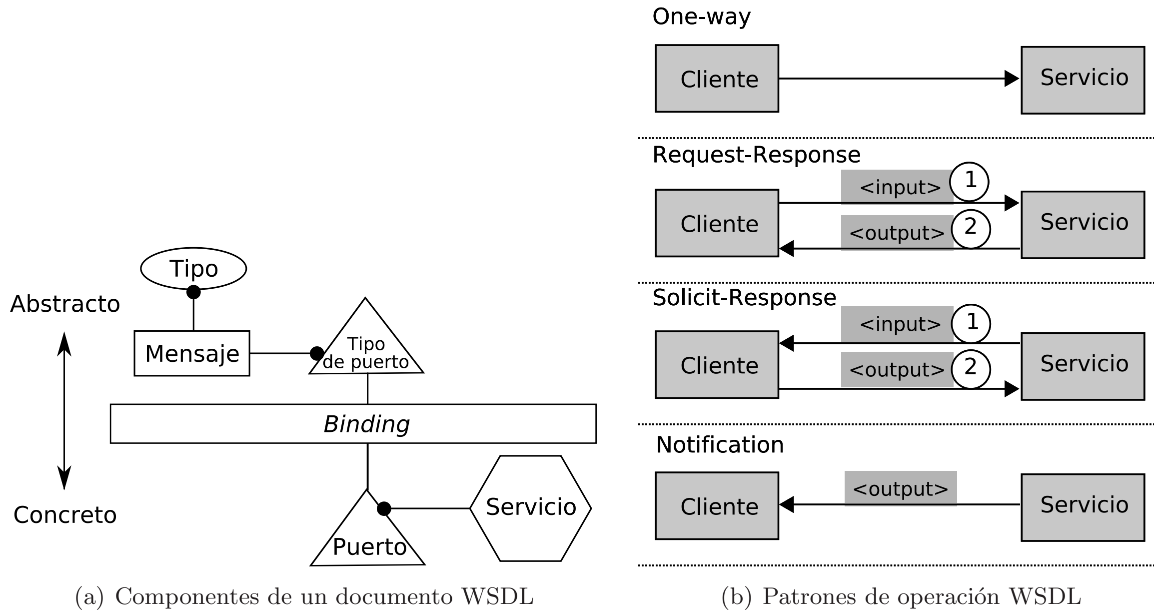
2.3.3. WSDL

El objetivo fundamental del modelo de Servicios Web es proveer una representación común de las aplicaciones que utilizan diversos protocolos de comunicación y modelos de interacción, posibilitando a su vez a los Servicios Web a aprovechar protocolos más eficientes cuando éstos se encuentran disponibles a ambos lados de una interacción. Una alterativa para alcanzar este objetivo es separar las descripciones abstractas de funcionalidad de una aplicación del modelo de interacción involucrado, y representando tales descripciones en un lenguaje común interpretable por cada aplicación.

Uno de los lenguajes más difundidos para la descripción y representación de Servicios Web es WSDL (Web Services Definition Language) (Christensen et al., 2001). WSDL es un lenguaje basado en XML para describir Servicios Web como un conjunto de puntos en una red que operan a través de mensajes genéricos. Un documento WSDL define *servicios* como una colección de *puertos* (los puntos de la red). En WSDL, las definiciones abstractas de los puertos y los mensajes se encuentran separadas de su correspondiente implementación y mapeo de datos a bajo nivel específicos del lenguaje de programación (figura 2.6 (a)). Esto permite la reutilización de las definiciones abstractas de los *mensajes*, que son descripciones acerca de los datos que se intercambian durante la interacción con el servicio, y los *tipos de puerto*, que son colecciones abstractas de *operaciones*. El protocolo de transporte concreto y las descripciones del formato de los tipos de datos utilizados por un *tipo de puerto* particular constituye un *binding*. En este sentido, un puerto se define asociando una dirección de red - típicamente un URL - con un *binding* determinado, y un conjunto de puertos definen un servicio. Así, un documento WSDL incluye los siguientes elementos para la definición de uno o más Servicios Web:

- *Tipos*: Es el conjunto de las definiciones de los tipos de datos utilizados para describir los mensajes intercambiados. En general, este conjunto es construido extendiendo algún sistema de tipos básicos existente tales como XSD (XML Schema Definition), que incluye tipos de datos básicos (caracteres, valores numéricos, cadenas de caracteres, etc.) y posibilita la definición de tipos de datos complejos (listas, uniones, arreglos, etc.).
- *Mensaje*: Representa una definición abstracta de un dato tipado a ser transmitido. Un mensaje consiste de diversas partes lógicas, cada una de las cuales está asociada con un tipo de dato específico dentro del sistema de tipos de datos utilizado.
- *Tipo de puerto*: Es un conjunto de operaciones abstractas. Cada operación puede incluir uno o más mensajes de entrada y salida. El modelo de interacción entre una aplicación y una operación dada dependerá de los mensajes definidos por ésta última. Por ejemplo, una operación que incluye mensajes de entrada y salida se corresponde con la interacción al estilo cliente-servidor. Por otra parte, una operación que sólo define mensajes de salida se corresponde con una notificación asincrónica que la aplicación puede recibir en un momento determinado por parte del servicio. Los patrones de operación soportados por WSDL se muestran en la figura 2.6 (b).
- *Binding*: Especifica el protocolo y el formato de datos concretos para las operaciones y los mensajes que han sido definidos por un *tipo de puerto*. En otras palabras, un *binding* asocia la descripción abstracta de un servicio con la implementación concreta de la funcionalidad del mismo, como muestra la figura 2.6 (a).

- *Puerto*: Especifica el punto de la red o URL específico para un *binding.*
- *Servicio*: Se utiliza como "contenedor" de un conjunto de puertos relacionados. Por ejemplo, pueden existir dos puertos que representan un servicio de compra de electrodomésticos, pero cada uno con *bindings* diferentes (por ejemplo, uno accesible a través de HTTP y otro a través de IIOP).



(a) Componentes de un documento WSDL (b) Patrones de operación WSDL

Figura 2.6: Componentes y tipos de operaciones de una definición WSDL

El siguiente ejemplo muestra la definición WSDL de un servicio simple destinado a proveer la cotización actual de una determinada acción bursátil. El servicio soporta una única operación llamada "GetLastTradePrice", accesible a través del protocolo SOAP. La solicitud de servicio toma como entrada la clave o nombre del título accionario (cadena de caracteres) y retorna su cotización como un valor de punto flotante.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="GetLastTradePriceInput">
    <part name="tickerSymbol" type="xsd:float"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="tradePrice" type="xsd:string"/>
  </message>
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
```

```
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>
  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
  </binding>
  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>
</definitions>
```

A partir de una especificación WSDL, un programa puede determinar los servicios que provee un servidor dado y cómo invocar estos servicios, independientemente del protocolo de transporte utilizado o el lenguaje de programación involucrado en la implementación del servicio. Esta característica, sumado a que está basado en XML, hacen de WSDL un lenguaje de descripción de Servicios Web de un alto grado de interoperabilidad.

En resumen, el propósito de WSDL es describir Servicios Web en un formato procesable e interpretable por las computadoras. Sin embargo, WSDL está mayormente enfocado en el nivel sintáctico de las descripciones de las entradas y salidas de un servicio (mensajes y tipos de datos requeridos), sin especificar nada sobre la semántica del mismo. Esto es adecuado en la medida que los clientes operen con el servicio mencionado de una forma relativamente fija, o en otras palabras, la semántica está implícita y es conocida por ambas partes. Pero, si se pretende utilizar WSDL para permitir la automatización de procesos de negocios de mayor nivel, es necesario expresar de forma no ambigua el significado de cada servicio, de modo que los procesos puedan realizar decisiones autónomas acerca de los Servicios Web que desean usar. Típicamente, estos problemas son resueltos agregando una descripción semántica a cada operación definida dentro del documento WSDL, como detalla (Sivashanmugam et al., 2003).

2.3.4. UDDI

Como complemento a WSDL, el WWW Consortium ha desarrollado la especificación UDDI (Universal Description, Discovery and Integration) (W3C Consortium, 2001). UDDI provee un mecanismo para la publicación y búsqueda de descripciones de servicios escritas en WSDL, y en general cualquier otro mecanismo de descripción de servicios. Mediante el uso de las facilidades de UDDI, los potenciales proveedores de servicios (por ejemplo empresas u organizaciones) registran individualmente la información acerca de los Servicios Web que ofrecen, para luego éstos ser usados por terceros. UDDI cuenta con nodos o registros que replican información basados en una política determinada. Una vez que un proveedor registra información

en un nodo UDDI, la misma es automáticamente compartida con otros nodos y queda disponible a cualquiera que necesite determinar qué Servicios Web son expuestos por un proveedor dado.

Como muestra la figura 2.7, existen cuatro estructuras de datos principales descritas en la especificación UDDI. Por un lado, las Entidades de Negocio (*Business Entities*) almacenan la información acerca de una empresa u organización, incluyendo su nombre, descripción, servicios ofrecidos e información de contacto. Los Servicios de Negocio (*Business Services*) proveen un mayor detalle de los servicios ofrecidos. A su vez, cada servicio puede tener múltiples *bindings*, cada uno describiendo un punto de entrada diferente para un servicio (mail, http, ftp, etc.). Finalmente, los *tModels* describen las especificaciones o estándares particulares utilizadas por cada servicio. Cada *tModel* puede estar asociado con estándares de descripción tales como WSDL, o taxonomías tales como NAICS (US Census Bureau, 2002). Con esta información, un cliente puede ubicar los servicios que son compatibles con las características técnicas de su propio sistema, como puede ser la plataforma utilizada.

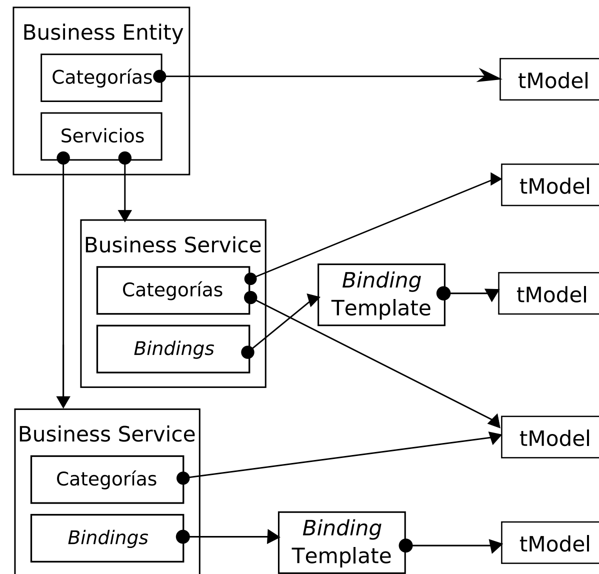


Figura 2.7: Principales estructuras de datos de UDDI

La implementación básica de la especificación UDDI se denomina UDDI Business Registry. Con este registro, una organización puede fácilmente publicar los servicios que ofrece y descubrir qué servicios proveen otras organizaciones. El registro en cuestión es creado como un grupo de múltiples sitios operadores. A pesar de que cada sitio operador es administrado separadamente, la información contenida en cada registro se sincroniza a través de todos los subregistros de cada uno de los sitios.

Una limitación de UDDI es que no hace uso de información semántica alguna. Por consiguiente, falla a la hora de resolver el problema de determinar similaridad semántica entre la funcionalidad provista por los servicios, imposibilitando entonces la búsqueda de servicios basada en las capacidades de los mismos. Una alternativa para la creación de registros UDDI compuestos de descripciones semánticas se describe en (Paolucci et al., 2002). Aquí, los autores proponen el mapeo de Perfiles DAML-S, que son descripciones semánticas de las capacidades de un servicio, a registros UDDI, permitiendo así almacenar en cada registro

información semántica acerca de la funcionalidad provista por los servicios asociados. Otros trabajos que merecen mención y que poseen un enfoque similar son (Akkiraju et al., 2003) y (Pokraev et al., 2003).

Hasta aquí se tiene una noción general acerca de qué consiste la Web Semántica, y de las principales tecnologías relacionadas a ella. Además, se exploró el concepto de Servicio Web, es decir, funcionalidad reutilizable y accesible a cualquier aplicación publicada a lo largo de la Web. En la siguiente sección se analizará en detalle el concepto de agente - entidades de software autónomas e inteligentes - para luego discutir cómo pueden ayudar los agentes en la selección, búsqueda e interacción con los Servicios Web.

2.4. Agentes

El término *agente* ha sido utilizado en los últimos tiempos para referirse a porciones de software autónomo e inteligente, con ciertos atributos adicionales tales como capacidad de percepción, reacción y deliberación. Sin embargo, no existe aún una definición universalmente aceptada por los autores de la Inteligencia Artificial del término agente, debido principalmente a la incertidumbre en cuanto a los límites y contextos donde éste puede ser aplicable. Se citan a continuación algunas definiciones del significado del término agente, según varios autores:

- Un agente es un componente de software y/o hardware que es capaz de actuar de acuerdo a los intereses del usuario (Nwana y Azarmi, 1997).

- Un agente es un componente computacional que habita en un ambiente complejo y dinámico. El agente puede percibir y actuar sobre su ambiente. Tiene un conjunto de objetivos o motivaciones que trata de satisfacer por medio de acciones preestablecidas (Maes, 1995).

- Un agente es un objeto activo con iniciativa (Parunak, 1996).

- Un agente es una entidad que reside en un ambiente donde interpreta datos que reflejan eventos en dicho ambiente y ejecuta comandos que producen efectos en el mismo ambiente. Un agente puede ser puramente software o hardware. En el último caso, una cantidad considerable de software se necesita para que ese hardware sea un agente (The Foundation for Intelligent Physical Agents, 1997).

- Un agente es un componente de software capaz de realizar ciertas tareas, que posee determinadas habilidades para la resolución de problemas, posiblemente conteniendo un estado interno y con diferentes propiedades. Un agente no es muy útil como componente aislado, sino que puede formar parte de un sistema multiagente, y desempeñar tareas de acuerdo a sus capacidades de resolución de problemas en colaboración con otros agentes (Moulin y Chaib-draa, 1996).

- Un agente es un componente de software activo y persistente que percibe, razona, actúa y se comunica (Huhns y Singh, 1997).

Una gran cantidad de investigadores de la Inteligencia Artificial asocian a los agentes con conceptos relacionados con los seres humanos, tales como conocimiento, creencia, intenciones y obligaciones (Minsky, 1985; Rao y Georgeff, 1995). Estos conceptos adicionan algunos atributos a los agentes que se mencionan a continuación:

- *Capacidad de acción*: Un agente es capaz de actuar, pudiendo modificar su medio ambiente. La capacidad de acción de un agente está determinada por las acciones que este puede realizar. Por ejemplo, las acciones básicas de un agente encargado de almacenar el contenido de un camión en un depósito podrían ser tomar un elemento del camión, desplazarse y dejar un elemento en el depósito.

- *Percepción*: El agente tiene la habilidad percibir los acontecimientos de su medio ambiente.

- *Reacción*: Los agentes pueden percibir su ambiente y responder de acuerdo a ciertas restricciones temporales a los cambios que ocurren en él. Por ejemplo, un agente podría estar interesado en los cambios de ubicación de otros agentes, y realizar alguna acción en consecuencia.

- *Autonomía*: Es la capacidad de actuar sin intervención humana o de otros sistemas con el fin de alcanzar sus objetivos. Una característica clave de los agentes autónomos es su habilidad de tomar la iniciativa de sus actos, en lugar de actuar sólo en respuesta a su medio ambiente.

- *Deliberación o racionalidad*: Es la capacidad de un agente de actuar determinando qué acciones realizar con el fin de alcanzar sus objetivos. La deliberación a menudo implica que el agente es capaz de *aprender*, modificando su conocimiento o comportamiento en base a la experiencia, con el objeto de cumplir con sus objetivos de una forma más eficaz y eficiente.

- *Comunicación a nivel de conocimiento*: Representa la capacidad de comunicarse con agentes y personas con lenguajes de alto nivel (análogos a los *speech acts* humanos, en lugar de protocolos entre programas), tales como KQML (Knowledge Query and Manipulation Language) (Finin et al., 1997).

- *Habilidad social o cooperación*: Los agentes pueden interactuar conjuntamente con otros agentes y/o con humanos para la resolución de un problema. Un ejemplo de esta situación lo constituye un grupo de agentes que simulan un equipo que participa en alguna competencia deportiva.

- *Negociación*: Es el proceso de mejora de acuerdos entre dos agentes con respecto a puntos de vista o planes comunes a ambos a través del intercambio de información.

- *Movilidad*: Capacidad de moverse autónomamente de una plataforma huésped a otra, posiblemente llevando consigo información acerca del estado de ejecución de las tareas. Pueden llevar directivas o instrucciones que pueden realizarse tanto en forma local como remota.

2.4.1. Tipos de agentes

El conjunto de atributos presentado en la sección anterior ha generado una gran cantidad de clasificaciones de los distintos tipos de agentes, de acuerdo a diferentes combinaciones de dichos atributos. Una clasificación clásica (Demazeau y Müller, 1991; Nwana, 1996) considera tres tipos de agentes:

- *Reactivos*: Los agentes reactivos reaccionan ante los cambios en su ambiente o mensajes de otros agentes. Dichos cambios y mensajes pueden provocar que las acciones o tareas que el agente reactivo posee se activen. Tal activación puede provocar a su vez un cambio en su conocimiento y/o el envío de mensajes a otros agentes. Por otra parte, un agente reactivo no es capaz de razonar acerca de sus intenciones u objetivos.

- *Deliberativos*: Son agentes capaces de razonar a partir de sus intenciones (objetivos perseguidos por el agente) y creencias (el conocimiento considerado por el agente como verdadero), y de crear y ejecutar planes de acción a seguir. Pueden seleccionar uno de sus posibles objetivos de acuerdo a sus motivaciones, detectar conflictos y coincidencias en la creación de los planes de acción, etc.

- *Híbridos*: Los agentes híbridos poseen las características de los agentes deliberativos y reactivos en forma simultánea.

Una clasificación alternativa muy difundida considera un espacio tridimensional, compuesto por los ejes denominados *agencia*, *inteligencia* y *movilidad* (Bradshaw, 1997). La agencia es el grado de autonomía del agente, y puede ser medido en función de la naturaleza de sus interacciones con otras entidades que incluyen agentes o incluso humanos. La inteligencia es el grado de razonamiento y aprendizaje que posee un agente. Como mínimo, un agente debe considerar las preferencias del usuario, y almacenar las mismas asociándoles alguna representación. Los mayores niveles de inteligencia incluyen un modelo del usuario y razonamiento, con capacidad de aprender y de adaptarse al ambiente. La movilidad se refiere a la capacidad del agente de transportarse entre diferentes sitios de una red (Fuggetta et al., 1998).

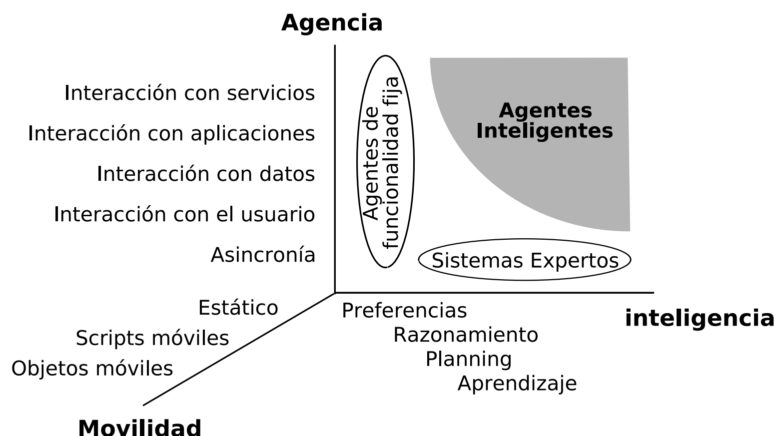


Figura 2.8: Clasificación de agentes: Agencia, Inteligencia y Movilidad (Bradshaw, 1997)

La figura 2.8 muestra un diagrama de la clasificación expuesta. Cuanto mayor es la agencia y la inteligencia del agente, más se acerca éste a ser un agente *inteligente*. Los agentes inteligentes poseen capacidades deliberativas y se caracterizan por poseer un modelo mental simbólico de su medio ambiente, de otros agentes y de sí mismo, a partir del cual decide qué acciones tomar. Dicho modelo mental es construido a partir de las percepciones realizadas por el agente, los mensajes recibidos y las acciones que puede llevar a cabo.

2.4.2. Agentes móviles

La movilidad es la capacidad de migrar una computación de un sitio de una red a otro durante su ejecución (White, 1996). Diversos autores del área de sistemas distribuidos (Johansen et al., 1995; White, 1996; Fuggetta et al., 1998) han dado el nombre de *agentes móviles* a los sistemas con la capacidad de migrar su computación durante su ejecución. Sin embargo, en el contexto de los sistemas distribuidos, el término agente no es utilizado con el mismo significado que el que se le da en el ámbito de los sistemas multi-agente. De aquí en más se considerará un agente móvil como un componente de software capaz de transportarse entre los diferentes sitios de una red.

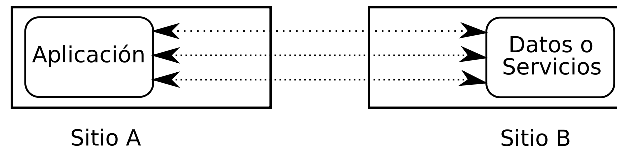
Es preciso notar que la movilidad es una propiedad ortogonal a las demás características de los agentes - reacción, percepción, deliberación, etc. - ya que no todos los agentes son móviles. Los agentes que no cuentan con la propiedad de movilidad se conocen con el nombre de *agentes estacionarios*. Estos agentes sólo ejecutan en el sitio en el cual fueron creados. Cuando un agente estacionario requiere recursos que no están presentes en el sitio origen, o desea interactuar con agentes localizados en sitios remotos, recurren al uso de mecanismos de comunicación - típicamente basados en mensajes o al estilo RPC (Remote Procedure Call) - para satisfacer sus necesidades.

En contraste con un agente estacionario, un agente móvil no está limitado a permanecer únicamente en el sitio donde comenzó su ejecución, sino que está posibilitado de viajar libremente a través de una red. Puede transportar su estado y código junto con él de un ambiente de ejecución a otro, para luego reanudar su ejecución. El *estado* incluye los datos (valores de las variables) y la información del estado de ejecución (contador de programa y pila) del agente, ambos necesarios para continuar la ejecución del mismo en el sitio remoto. El tipo de código transportado está íntimamente relacionado con el paradigma de programación en el que el agente está desarrollado. Por ejemplo, si se trata del caso de un lenguaje orientado a objetos, el código abarcará un determinado conjunto de clases, mientras que, para un lenguaje basado en lógica, el código se relacionará con las reglas o cláusulas lógicas que representan el conocimiento y el comportamiento del agente en cuestión.

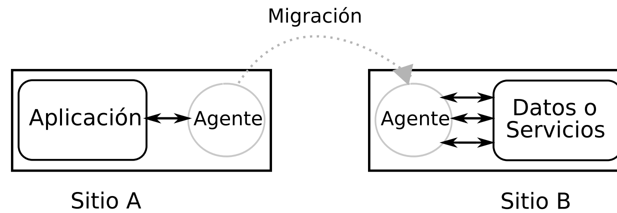
2.4.2.1. Ventajas

Los agentes móviles exhiben una serie de características que los hacen ideales para explotar el potencial de las redes actuales, lo que ha motivado su utilización en aplicaciones tan diversas como comercio electrónico, recuperación de información en Internet, asistencia personalizada y telecomunicaciones. Particularmente, a continuación se enumeran algunos de los potenciales beneficios que ofrece el uso de agentes móviles para la construcción de sistemas distribuidos (Lange y Oshima, 1998; Lange y Oshima, 1999):

1. *Reducen la carga de red*: Los sistemas distribuidos a menudo utilizan protocolos de comunicación que involucran múltiples interacciones para llevar a cabo una tarea dada, resultando en un elevado tráfico de red (figura 2.9 (a)). Los agentes móviles permiten migrar a otro sitio e interactuar con recursos o servicios en forma local, como se muestra en la figura 2.9 (b). Adicionalmente, son útiles para reducir el flujo de datos de la red. Cuando se almacenan grandes volúmenes de datos en sitios remotos, es posible interactuar localmente con ellos a través de un agente móvil, en vez de transferir dichos datos a través de la red hacia la ubicación del agente.



(a) Tráfico de red para una arquitectura estilo cliente/servidor



(b) Tráfico de red y uso de agentes móviles

Figura 2.9: Agentes móviles: Reducción de la carga de red

2. *Soportan desconexión de los usuarios*: Muchos de los usuarios hoy en día son móviles, comenzando su trabajo en una oficina, para luego quizá continuarlo en una *laptop* en una ubicación diferente. A menudo, las tareas iniciadas por los usuarios deben continuar mientras el usuario está desconectado. En este sentido, puede delegarse esta responsabilidad a un agente móvil que se ocupará de las tareas de su dueño, incluso si éste está desconectado, como se ilustra en la figura 2.10. Cuando el usuario reanuda su conexión, el agente puede ser traído desde el sitio actual hacia la ubicación del usuario.

3. *Son de naturaleza heterogénea*: Las redes de computadoras son fundamentalmente heterogéneas, desde el punto de vista del software y el hardware que utilizan. Debido a que los agentes móviles son generalmente independientes del software y hardware subyacente, proveen condiciones óptimas para la integración de sistemas de naturaleza heterogénea.

4. *Son robustos y tolerantes a fallas*: La habilidad de reaccionar antes eventos y situaciones desfavorables que poseen los agentes móviles posibilita la construcción de sistemas robustos y tolerantes a fallas con mayor facilidad. Si un sitio está a punto de ser apagado, todos los agentes que están actualmente ejecutando en él pueden ser notificados y provistos de un cierto tiempo para abandonar el sitio. Los agentes pueden luego continuar su ejecución en un sitio diferente.

5. *Facilidad de desarrollo*: El uso de agentes móviles facilita la programación de sistemas en donde existe una analogía con el mundo real. Por ejemplo, un comprador de artículos

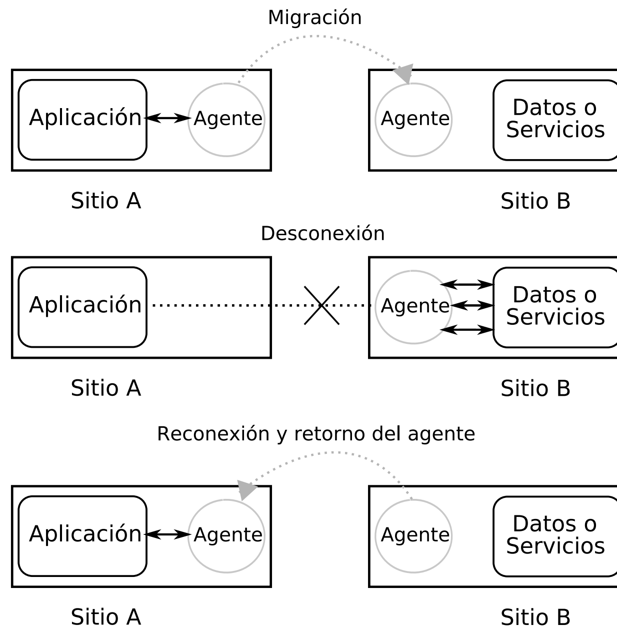


Figura 2.10: Agentes móviles y desconexión de usuarios

que realiza ofertas a diferentes vendedores resulta en una representación natural de un agente que viaja a través de la red y visita servidores.

Las ventajas señaladas anteriormente permiten que el uso de los agentes móviles beneficien a una gran gama de aplicaciones. Históricamente, la utilización de los agentes móviles se ha centrado mayormente en aplicaciones de asistencia a la búsqueda, recuperación y análisis de información en Internet. Con un agente móvil realizando estas tareas, el usuario no tiene la necesidad de lidiar con la vasta cantidad de sitios que produce hoy en día un motor de búsqueda en Internet. Esta idea se ha generalizado proponiendo la utilización de los agentes (móviles) para la búsqueda de servicios en la Web Semántica. Particularmente, las características ofrecidas por los agentes jugarán un rol fundamental en la realización de la visión de una Web proveedora de Servicios Web Semánticos (Jennings et al., 2000; Hendler, 2001).

2.5. Agentes y Servicios Web

La WWW se está convirtiendo más y más en un instrumento para la comunicación entre aplicaciones, y está siendo masivamente aceptado por los investigadores el hecho de que los Servicios Web jugarán un rol fundamental en este contexto (Jennings et al., 2000). A pesar de que muchas tecnologías de computación distribuida (por ejemplo CORBA) han abordado este problema antes, la naturaleza evolutiva de Internet junto con la especial atención que merecen los estándares de interoperabilidad implica la existencia de una necesidad mayor que antes de materializar los beneficios de la computación distribuida basada en tecnologías y modelos estándar.

Los diseñadores que están construyendo sistemas basados en tecnologías de Servicios Web se enfrentan con muchos de los problemas que los diseñadores de sistemas multiagente han

tratado en los últimos 10 o más años. La madurez de la tecnología de los agentes junto con los beneficios ofrecidos por las tecnologías relacionadas a los Servicios Web permiten un uso inteligente de los recursos de la Web Semántica. En general, los agentes poseen características que los hacen adecuados para resolver algunas de las dificultades inherentes del dominio complejo y dinámico que presenta el modelo de Servicios Web (Huhns, 2003), a saber:

- Un Servicio Web sólo conoce sobre sí mismo, y no acerca de sus usuarios o clientes. En este sentido, un agente inteligente es un componente de software ideal para representar y actuar en favor de un cliente (Jennings et al., 1998); de ser necesario, puede recordar las "conversaciones" o sesiones establecidas con cada servicio, manteniendo esta información en forma persistente para usos futuros.

- Los Servicios Web no están diseñados para utilizar o integrar ontologías entre sí o con sus clientes. Sin embargo, los repositorios que asocian cada servicio con su correspondiente descripción semántica pueden ser utilizados y/o administrados por agentes inteligentes capaces de comprender información ontológica.

- Un Servicio Web es pasivo hasta el momento en el cual es invocado; no pueden emitir alertas o notificaciones cuando hay nueva información disponible. En este caso, implementar un Servicio Web como un agente persistente a través del tiempo que se comunica periódicamente con la aplicación que hace uso del servicio permite materializar dicha funcionalidad.

- En general, los Servicios Web se materializan como aplicaciones independientes que no cooperan entre sí. Sin embargo, nueva funcionalidad compleja puede ser obtenida a partir de la composición de servicios más simples. En particular, si las características funcionales de éstos últimos son acompañadas de una apropiada descripción semántica, los agentes pueden utilizar dicha información para comprender la funcionalidad y estructura interna de cada Servicio Web, y generar servicios más complejos que ofrezcan funcionalidad inexistente (Hendler, 2001).

- Los Servicios Web no están pensados para ser utilizados en ambientes sujetos a fuertes limitaciones en el uso de recursos. Por ejemplo, la interacción con Servicios Web publicados en una red de dispositivos móviles requiere resolver el problema relativo al escaso ancho de banda e inestabilidad propios de las conexiones inalámbricas. Aquí, los agentes *móviles* son particularmente útiles, ya que ofrecen la posibilidad de interactuar en forma local con los servicios, sin necesidad de establecer conexiones remotas que consumen una cantidad mayor de recursos de red (Ishikawa y Tahara, 2004).

Hoy en día, existe un extenso conjunto de servicios publicados accesibles a través de la Web. La alternativa de descubrir y utilizar dichos servicios por parte del usuario no sólo supone un gran consumo de tiempo, sino que también requiere una gran cantidad de interacción entre la aplicación y el usuario (Palathingal y Chandra, 2004). En este sentido, resulta más adecuado estructurar la aplicación en cuestión como uno o más agentes que, en base a las preferencias y objetivos del usuario, descubran los servicios que mejor se adaptan a las condiciones deseadas, e interactúen autónomamente con ellos para retornar los resultados de la computación al usuario.

Considérese por ejemplo el siguiente escenario. Un analista, habituado al uso de Sistemas de Información Geográfica (GIS) está analizando un proyecto ambiental para una región geográfica determinada. Para esto, dicho analista necesitará información orográfica de la región, artículos o documentos que detallen las actividades desarrolladas en la zona (por ejemplo, construcción), y una herramienta gráfica o GIS para visualizar esta información. Para hallar tanto la información como el software requerido, el usuario típicamente deberá navegar la Web. El problema de esta alternativa radica en que la búsqueda manual de la información es una tarea extremadamente lenta y laboriosa por un lado, y poco eficaz por otro, debido a que los motores de búsqueda actuales que consultan páginas no tienen en cuenta la semántica de los datos. En contraste, delegar dichas tareas de búsqueda a un agente que navegue "semánticamente" la Web en busca de servicios apropiados que le brinden la información que el usuario ha solicitado representa una alternativa mucho más adecuada.

Hasta el momento, la investigación relacionada con la tecnología de agentes ha generado muchos resultados en cuanto a modelos y arquitecturas de software orientado a agentes. Adicionalmente, se han desarrollado una gran cantidad de aplicaciones y *toolkits* para la implementación de sistemas basados en agentes, pero existe aún una clara separación entre teoría y práctica. Particularmente, existe una falta de plataformas que aprovechen las características positivas ofrecidas por los agentes inteligentes para el desarrollo de aplicaciones en el contexto de la Web Semántica (Dickinson y Wooldridge, 2003). Como se verá en el siguiente capítulo, existen diversos esfuerzos orientados a la creación de plataformas y herramientas para la materialización de agentes integrados con Servicios Web. Sin embargo, es preciso notar que dichas herramientas están aún en su fase inicial de desarrollo, impidiendo su uso en un contexto real de aplicación.

2.6. Conclusiones

En contraste con la Web actual donde el contenido está formado por páginas, documentos y datos multimedia diseñados principalmente para uso y comprensión humana, la Web Semántica es un concepto recientemente surgido que denota una Web donde la totalidad del contenido puede ser utilizado y comprendido por las aplicaciones. La Web Semántica no apunta a crear una nueva Web separada de la actual, sino que pretende ser una extensión de ella, en donde la información almacenada tiene un significado preciso y definido entendible por las computadoras, permitiendo así que las aplicaciones y los usuarios cooperen para la resolución de las tareas.

Desde hace un tiempo, la idea original de una Web Semántica destinada a proveer un ambiente que agrupe información semánticamente representada y relacionada está evolucionando hacia un ambiente donde las computadoras resuelven tareas en favor del usuario utilizando diversos servicios accesibles vía Web. En este sentido, la importancia que tienen los Servicios Web ha sido ampliamente reconocida y aceptada. Los Servicios Web son aplicaciones modulares autodescritas y autocontenidas que pueden ser publicadas, ubicadas e invocadas a través de la Web. Los Servicios Web llevan a cabo funciones que pueden variar desde simples requerimientos o funcionalidad básica hasta complejos procesos y *workflows* de negocio.

Las características que exhiben los Servicios Web hacen de los agentes inteligentes una tecnología apropiada para la integración de dichos servicios a la Internet. Más específicamente,

los agentes inteligentes pueden simplificar significativamente el desarrollo de aplicaciones Web gracias a sus capacidades de autonomía y razonamiento, y además su adaptabilidad a ambientes abiertos y distribuidos cuando se trata de agentes inteligentes *móviles*. Por ejemplo, los usuarios pueden emplear agentes de software que los ayuden a identificar, localizar y obtener los productos o servicios que necesitan. Adicionalmente, los usuarios pueden conferir cierto grado de confianza a sus agentes para permitirles negociar electrónicamente con proveedores (o sus agentes asociados) con el fin de comprar o vender productos y servicios dentro de un sistema de comercialización distribuido.

En definitiva, la introducción de los agentes de software dentro de la infraestructura de soporte de los Servicios Web ofrece grandes ventajas en pro de la automatización y la coordinación de las tareas que los usuarios llevan a cabo sobre Internet. Por otra parte, la aceptación masiva de lenguajes aptos para la descripción de metadatos tales como XML permiten la comunicación semántica entre aplicaciones a un nivel de interoperabilidad impensado hace unos años atrás. En este sentido, las características cognitivas que poseen los agentes inteligentes pueden resultar beneficiosas para explotar y aprovechar la semántica de los recursos Web.

Capítulo 3

Trabajos Relacionados

Los Servicios Web son una consecuencia natural de la evolución de la WWW, y constituyen un modelo apropiado para la interacción sistemática entre aplicaciones basadas en la Web y la integración de ambientes y plataformas preexistentes. En los últimos años, el soporte para una verdadera interacción entre aplicaciones Web ha comenzado a tomar forma, primeramente con el desarrollo de transacciones B2B (Business-to-Business) automatizadas, y más recientemente, con la creación de infraestructura para compartir recursos computacionales a gran escala, lo que se conoce con el nombre *Grid Computing*.

Un solución alternativa al problema de la integración de sistemas heterogéneos y el acceso automatizado a recursos distribuidos lo constituyen los agentes inteligentes. En particular, los agentes inteligentes *móviles* son por naturaleza heterogéneos, por lo que ofrecen una solución flexible a la integración de sistemas y plataformas de diverso tipo (Lange y Oshima, 1999). Adicionalmente, los agentes móviles permiten hacer un uso eficiente de los recursos tanto de hardware como de software, debido principalmente a las características móviles y de conocimiento de ubicación que poseen (Chess et al., 1997; Gray et al., 2000). Más aún, la tecnología de agentes móviles constituye un paradigma poderoso para el desarrollo de aplicaciones que acceden a información y recursos Web, en particular si dichas aplicaciones se estructuran como un conjunto de agentes móviles que interactúan de forma inteligente y automatizada con la Web Semántica.

En este capítulo se presentarán los trabajos más relevantes relacionados con la problemática anteriormente presentada. El mismo se organiza como sigue: en la sección 3.1 se presentan los proyectos de gran envergadura relacionados con la Web Semántica existentes en la actualidad. A continuación, en la sección 3.2 se analizarán los lenguajes y plataformas que permiten la interacción entre agentes y Servicios Web. Por último, en las secciones 3.3 y 3.4 se comparan las propuestas y se analizan sus principales inconvenientes, respectivamente.

3.1. Proyectos

Para hacer realidad la visión de la Web Semántica, es necesario contar con lenguajes de representación de información semántica altamente descriptivos, algoritmos potentes de manipulación e inferencia de conocimiento, y plataformas y *frameworks* capaces de explotar

adecuadamente la información y servicios presentes en la nueva Web. Con este fin, una gran cantidad de proyectos han surgido a nivel mundial con el fin de incentivar y alentar la creación de tecnologías que ayuden a impulsar el estado del arte relacionado con la Web Semántica. A continuación se describen Grid Computing, WSMF y Agentcities.

3.1.1. Grid Computing

El concepto principal que gira en torno a la investigación en el área de Grid Computing (Foster et al., 2001) es la Grilla (Foster y Kesselman, 1999), término que denota a una infraestructura de computación distribuida cuyo objetivo es permitir compartir cualquier recurso computacional entre organizaciones de diversa índole (institutos de investigación, empresas, organizaciones gubernamentales, etc.), de forma segura y coordinada. Así, estas entidades se convierten en organizaciones virtuales (VO) que se asocian temporalmente para responder a requerimientos de aplicaciones de procesamiento a gran escala, cuya cooperación es soportada a través de redes de computadoras de gran tamaño (Camarinha-Matos y Afsarmanesh, 1999). Este nuevo enfoque para la computación distribuida es también conocido con nombres alternativos tales como “Metacomputing”, “Scalable Computing”, “Global Computing”, “Internet Computing”, y más recientemente, “Peer-to-Peer Computing” (P2P) (Oram, 2001).

Cada Grilla permite compartir, seleccionar y agrupar una amplia variedad de recursos que incluyen supercomputadoras, grandes sistemas de almacenamiento, repositorios de datos, servicios y dispositivos especializados que están geográficamente distribuidos y que son propiedad de diferentes organizaciones. Dichos recursos son utilizados en conjunto para resolver problemas de procesamiento computacional de datos intensivos de gran escala de la ciencia, la ingeniería y el comercio. El concepto de Grid Computing comenzó como un proyecto para vincular geográficamente supercomputadoras dispersadas alrededor del mundo, pero ha crecido vertiginosamente desde sus inicios. La infraestructura de una Grilla Computacional puede beneficiar a un gran número de aplicaciones, entre las que se encuentran la ingeniería colaborativa, la exploración de datos, y la computación a gran escala distribuida.

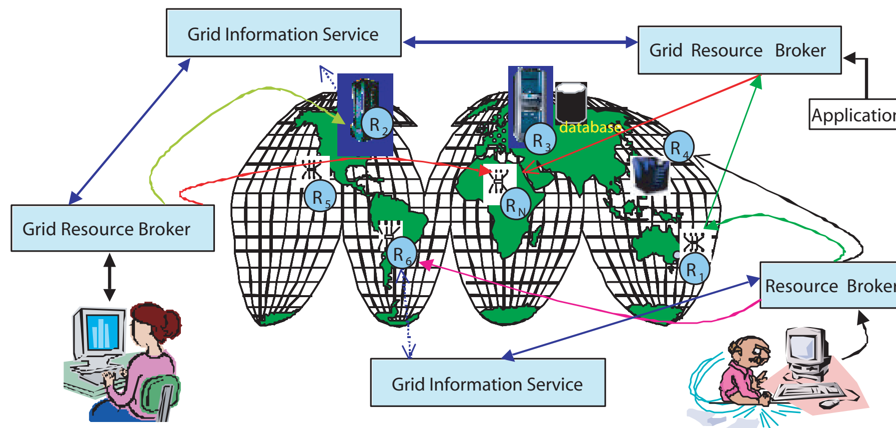


Figura 3.1: Una vista general de una Grilla Computacional y la interacción entre sus entidades

Una Grilla Computacional puede ser vista como un ambiente computacional integrado y colaborativo. Una vista de alto nivel de las actividades llevadas a cabo dentro de una Grilla

Computacional se muestra en la figura 3.1. El usuario interactúa con el Broker de Recursos para resolver problemas, que a su vez lleva a cabo la búsqueda y asignación planificada de los recursos, y el procesamiento de las aplicaciones encoladas para el acceso a los recursos distribuidos de la Grilla. Desde el punto de vista del usuario, las Grillas Computacionales pueden ser usadas para proveer los siguientes tipos de servicios:

- *Servicios computacionales*: Se ocupan de proveer servicios seguros para la ejecución de aplicaciones encoladas para acceder a recursos computacionales distribuidos. Los Brokers de Recursos proveen los servicios necesarios para el uso colectivo de los recursos distribuidos. Una Grilla que provee servicios computacionales es a menudo llamada Grilla Computacional. Algunos ejemplos de tales Grillas son NASA IPG (Johnston et al., 1999), la WWG (World Wide Grid) (Buyya, 2002) y NSF TeraGrid (National Sciences Foundation, 2002).

- *Servicios de datos*: Estos servicios están relacionados con el acceso y administración segura de datos distribuidos. Para proveer acceso y almacenamiento escalable a dichos datos, éstos pueden ser replicados, indexados, o incluso referenciados a través de la creación de vistas distribuidas sobre diferentes conjuntos de datos. El procesamiento de los datos es llevado a cabo mediante el uso de servicios especializados, y la combinación de ambos constituye la llamada Grillas de Datos. Ejemplos de aplicaciones que utilizan una Grilla de Datos son la física de alta energía (Hoschek et al., 2000) y el acceso de bases de datos distribuidas con información relativa al diseño de drogas (Buyya, 2001).

- *Servicios de aplicación*: Se relacionan mayormente con la provisión de acceso transparente a software y librerías remotos. Aquí es donde los Servicios Web juegan un rol protagónico, ya que proveen una solución satisfactoria al problema de integración de sistemas heterogéneos, en particular si se construyen sobre servicios computacionales y de datos provistos por una Grilla. Un ejemplo de sistema que puede ser utilizado para desarrollar tales servicios es NetSolve (Arnold et al., 2002).

- *Servicios de información*: Los servicios de este tipo se encargan de la extracción y presentación de datos que contienen un significado preciso. Para esto, los servicios de información utilizan los servicios computacionales, de datos y/o los servicios de aplicación.

- *Servicios de conocimiento*: Estos servicios tratan cuestiones relacionadas a la forma en que el conocimiento es adquirido, utilizado, recuperado, publicado y mantenido para asistir a los usuarios a alcanzar sus objetivos y metas particulares. Se entiende por "conocimiento" a toda información utilizada para alcanzar una meta, resolver un problema o ejecutar una decisión tomada. Ejemplos de servicios de conocimiento podrían constituir servicios que realizan Data Mining para inferir automáticamente nuevo conocimiento.

Existen actualmente muchos proyectos relacionados con Grid Computing en el mundo. En lo que a tecnología de Servicios Web respecta, el proyecto más importante lo constituye el *framework* Open Grid Services Architecture (OGSA) (Foster et al., 2002), ideado con el fin de integrar Servicios Web y Grid Computing. OGSA soporta la creación, mantenimiento y aplicación de conjuntos de servicios administrados por varias VO. Aquí, un servicio es definido

como una entidad accesible a través de una red que provee alguna capacidad, tales como recursos computacionales, recursos de almacenamiento y de red, programas y bases de datos.

Los estándares de Servicios Web utilizados en el proyecto OGSA incluyen SOAP (W3C Consortium, 2000) (para interacción con servicios a través de Internet), WSDL (Christensen et al., 2001) (para descripción de servicios ofrecidos por un proveedor determinado), y WS-Inspection (Ballinger et al., 2001), un lenguaje simple basado en XML (Bray et al., 1998) que establece convenciones para localizar descripciones de servicios publicadas por los diferentes proveedores. Los Servicios Web son esenciales en OGSA debido principalmente al gran nivel de interoperabilidad que dicha tecnología requiere para la materialización de aplicaciones distribuidas.

La adopción masiva de Grid Computing está sujeta a resolver en principio ciertas cuestiones técnicas y económicas, tal es el caso de un esquema general de seguridad global para las grillas, o la medición del consumo y uso de recursos dentro de ésta. Estos problemas limitan el alcance posible de las grillas a los límites de una organización, presentando problemas para su uso en el contexto de la Web. Adicionalmente, el enfoque de Grid Computing se encuentra más centrado en la creación de clústeres cooperativos de computadoras para resolver problemas a gran escala, característica que no puede ser asumida para la Web, dada la diversidad de capacidades del hardware que existe conectado a ella.

Lamentablemente, las grillas computacionales de hoy en día están tendiendo a ser un conjunto de plataformas con protocolos no interoperables e implementación propietaria y en general muy difícil de reutilizar. Quizás el problema más grave que presentan la mayoría de las implementaciones utilizadas para materializar aplicaciones basadas en grillas computacionales es la falta de interoperabilidad. En gran parte de los casos, la posibilidad de interoperación es nula, debido al hecho de que las aplicaciones están desarrolladas mediante plataformas cuya implementación es propietaria. Este es el caso, por ejemplo, de las plataformas provistas por IBM o Microsoft. En este sentido, OGSA propone la utilización de tecnología de Servicios Web (WSDL, SOAP, WS-Inspection) que permite abstraer las diferencias entre las plataformas, posibilitando así el acceso transparente a los servicios ofrecidos por las grillas computacionales. Sin embargo, sigue pendiente resolver el problema relacionado a la adopción de la arquitectura OGSA por parte de los desarrolladores de aplicaciones de Grid Computing.

3.1.2. WSMF

WSMF (Web Service Modeling Framework) (Fensel y Bussler, 2002) es una iniciativa europea que apunta a proveer una infraestructura de modelado y descripción de los aspectos funcionales y no funcionales relacionados a los Servicios Web. La iniciativa está siendo diseñada como parte del proyecto SWWS (Semantic Web enabled Web Services) fundado por la IST (Information Society Techonologies) perteneciente a la Unión Europea.

El objetivo principal de WSMF es la creación de un *framework* para la búsqueda, selección, interacción y composición de Servicios Web, esto es, permitir la programación de Servicios Web Semánticos y explotar sus beneficios. Con el fin de lograr esto, WSMF plantea dos principios complementarios:

- Fuerte desacoplamiento de los diferentes componentes que intervienen en una aplicación de comercio electrónico determinada. Este desacoplamiento incluye ocultamiento

de información basado en las diferencias entre la inteligencia interna de las partes intervinientes y las descripciones de las interfaces de los protocolos de intercambio público. El acoplamiento de los procesos se realiza a través de interfaces para mantener escalable el número posible de interacciones.

- Mediación en la utilización de servicios permitiendo la comunicación escalable entre diferentes entidades. Esto involucra mediación entre las diferentes terminologías empleadas (conceptos, lenguajes, etc.) como también entre los diferentes estilos de interacción (por ejemplo, los diferentes protocolos de nivel de transporte actuales).

A grandes rasgos, WSMF consiste de cuatro elementos fundamentales: las *ontologías*, que proveen terminología común utilizada por otros elementos; los *repositorios de capacidades*, que definen los problemas que deberían ser resueltos por los Servicios Web; las *descripciones* de los Servicios Web que definen los diferentes aspectos de cada servicio (precondiciones, postcondiciones, proveedor, costo, etc.); y los *mediadores*, que se ocupan de resolver los problemas de interoperabilidad.

Cabe destacar que WSMF es esencialmente un modelo conceptual que apunta a la definición de arquitecturas y plataformas interoperables para el desarrollo de aplicaciones basadas en Servicios Web Semánticos. Mucho trabajo queda por hacer para materializar esta visión. En este sentido, los componentes de la arquitectura conceptual están siendo mapeados a componentes de una arquitectura concreta basada en los estándares WSDL y BPEL4WS (Business Process Execution Language for Web Services) (Curbera et al., 2003). BPEL4WS es un lenguaje para la especificación de procesos o esquemas complejos de interacción (*workflows*) entre entidades de negocios. Sin embargo, las actividades de los procesos BPEL4WS son capaces de interactuar sólo con un conjunto preestablecido de Servicios Web, lo que le quita mucha flexibilidad. Por otra parte, BPEL4WS posee un manejo muy primitivo de la semántica de los servicios utilizados y del lenguaje "común" de las entidades intervinientes en el proceso de negocio.

3.1.3. Agentcities

Agentcities (Willmott et al., 2002) es una iniciativa a nivel mundial cuyo objetivo es proveer un ambiente para la investigación y el desarrollo basado en servicios a través de la Web. Básicamente, el proyecto apunta a la integración de las tecnologías de agentes y sistemas multiagentes junto con las tecnologías emergentes de la Web Semántica, con el fin de crear un modelo único que permita definir y describir ambientes de servicios complejos. La iniciativa cuenta actualmente con más de 100 organizaciones asociadas diseminadas a lo largo de 20 países del mundo.

Los nodos de la Red Agencities son plataformas de agentes distribuidas alrededor del mundo, que ejecutan en una o más computadoras cuyo dueño es una organización o una persona. Los agentes que ejecutan en una "ciudad" determinada pueden conectarse con otras ciudades disponibles públicamente y comunicarse directamente con sus agentes. Las aplicaciones que utilizan agentes ubicados en diferentes ciudades pueden ser creadas a través del uso de flexibles modelos de comunicación entre agentes y frameworks semánticos, ontologías compartidas, lenguajes de contenido y protocolos de interacción de soporte. En definitiva, este modelo consiste de los siguientes niveles:

- *Nivel del red*: Las diferentes plataformas de la Red Agentcities interoperan e intercambian mensajes básicos reutilizando la infraestructura de comunicación y transporte existente.
- *Nivel de composición de servicios*: Este nivel es el responsable de registrar y almacenar componentes de negocio tales como servicios provistos y las descripciones de éstos, como así también llevar a cabo la composición de servicios simples para satisfacer requerimientos de servicio más complejos por parte de los agentes de la red.
- *Nivel de interoperación semántica*: Este nivel convierte a la Red AgentCities en un ambiente abierto para la comunicación entre sistemas que es capaz de registrar, componer, buscar e invocar componentes de negocio sin necesidad de la intervención humana.

Las plataformas en la Red Agencities modelan los servicios disponibles en un pueblo o ciudad (de aquí el nombre de *Agentcities*). Existen una gran cantidad de aplicaciones basadas en agentes y Servicios Web que están siendo desarrolladas, típicamente en las áreas de servicios de turismo (planificación de viajes, visitas y vacaciones), eLearning (tutores inteligentes basados en agentes distribuidos), manufacturación (coordinación de procesos distribuidos de manufacturación e integración de cadenas de producción), personalización (composición dinámica de servicios basado en preferencias y gustos del usuario), entre otras.

La red Agentcities está completamente abierta a cualquier desarrollador de plataformas, agentes o servicios que quiera unirse o hacer uso de ella. La misma puede ser accedida mediante un conjunto de servicios[1] estándares que permiten publicar nuevos agentes y servicios, o descubrir y utilizar los ya existentes. A pesar de que Agentcities se encuentra en un nivel muy rudimentario, existen actualmente varios proyectos que están trabajando para aumentar la funcionalidad de la red.

3.2. Plataformas y lenguajes

Las características positivas que ofrece la Web Semántica, principalmente relacionadas con un mayor aprovechamiento de los recursos Web y la sistematización de las interacciones entre las aplicaciones que operan a través de la WWW, han motivado la creación de plataformas y lenguajes que permiten algún tipo de interacción con ella. A continuación se describen ConGolog, IG-Jade-PKSlib, Jinni, MWS, MyCampus, CALMA y TuCSoN.

3.2.1. ConGolog

ConGolog (Giacomo et al., 2000) es un lenguaje de programación de agentes desarrollado en la Universidad de Toronto, y está basado en una extensión a Golog (Levesque et al., 1997), un lenguaje de programación de alto nivel basado en lógica diseñado para la especificación y ejecución de acciones complejas en dominios dinámicos. Su uso principal es la programación de la planificación de tareas de robots.

ConGolog está construido por sobre el denominado Cálculo de Situación (McCarthy y Hayes, 1981), un lenguaje de primer orden que posee algunas características de los lenguajes de

[1] Disponibles a través de la dirección http://www.agentcities.net/

segundo orden. En este formalismo, el mundo es concebido como un árbol de situaciones, comenzando en la situación inicial, digamos S_0, y evolucionando hacia una nueva situación a través de la aplicación de una acción a. De esta manera, una situación s es una historia de las acciones llevadas a cabo partiendo de la acción inicial S_0. El estado del mundo está expresando en términos de relaciones y funciones (llamadas *fluentes*) que son verdaderas o falsas o tienen un valor particular en una situación s. La figura 3.2 muestra un ejemplo de un árbol de situaciones inducido por una teoría cuyas acciones posibles son $a_1, a_2, ..., a_n$ (ignorar las "X" por el momento). El árbol de la figura refleja un espacio de búsqueda posible derivado a partir de las acciones definidas y el conocimiento de un robot.

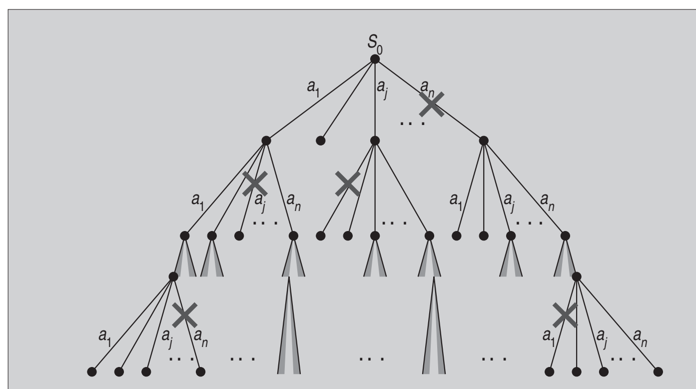


Figura 3.2: Un ejemplo de un árbol de situaciones

ConGolog ha sido extendido con éxito para la programación de agentes que interactúan con Servicios Web en el contexto de la Web Semántica (McIlraith y Son, 2002). El objetivo principal de esta extensión es proveer un lenguaje para la programación de agentes capaces de ejecutar tareas genéricas basadas en Servicios Web Semánticos, cuyos conceptos están descriptos en el lenguaje DAML (Horrocks et al., 2001; Horrocks, 2002).

Los agentes creados con ConGolog son en sí programas *basados en modelos*, compuestos de un *modelo* (en este caso la base de conocimiento del agente, o KB), y un *programa* (el procedimiento genérico que compone acciones atómicas que se desea ejecutar). Cuando el usuario invoca a un procedimiento genérico, como por ejemplo un procedimiento de construcción de itinerarios y reservas aéreas a través de ciudades, el agente busca en su base de conocimiento los Servicios Web basados en DAML que son relevantes para la ejecución del procedimiento.

ConGolog provee un conjunto de construcciones extra-lógicas para la programación procedural que permiten componer acciones primitivas y complejas del cálculo de situación para crear nuevas acciones complejas. Sean δ_1 y δ_2 acciones complejas, y sean ϕ y a los *pseudo-fluentes* y las *pseudo-acciones*, respectivamente; esto es, un fluente o una acción en el lenguaje del Cálculo de Situación con todos los argumentos de la situación no instanciados. La figura 3.4 muestra un subconjunto de las construcciones permitidas en el lenguaje ConGolog. Un usuario puede emplear las construcciones mostradas en la figura anterior para escribir procedimientos genéricos, que luego se traducen en acciones complejas. El conjunto de instrucciones para estas acciones complejas son simplemente Servicios Web generales (por ejemplo, **BookAirlineTicket(Origin, Destination, Date)**) u otras acciones complejas. En las figuras 3.3 y 3.4 se muestran las construcciones y algunos ejemplos de sentencias en ConGolog.

Primitive action: α
Test of truth: φ?
Sequence: (δ1;δ2)
Nondeterministic choice between action: (δ1 | δ2)
Nondeterministic choice of arguments: $\pi\chi\delta$
Conditional: if φ then δ1 else δ2 endIf
Loop: while φ do δ endWhile
Procedure: proc *P(v)* δ endProc

Figura 3.3: Construcciones posibles en ConGolog

while ∃x(hotel(x) & goodLoc(x,dest)) **do**
  checkAvailability(x,dDate,rDate)
endWhile
if ¬hotelAvailable(dest,dDate,rDate) **then**
  BookB&B(cust,dest,dDate,rDate)
endIf
proc Travel(cust,origin,dest,dDate,rDate,purpose);
  if registrationRequired **then** Register **endif**;
  BookTransport(cust,origin,dest,dDate,rDate);
  BookAccommodations(cust,dest,dDate,rDate);
  UpdateExpenseClaim(cust);
  Inform(cust);
endProc

Figura 3.4: Un ejemplo de programa ConGolog

Para instanciar un programa ConGolog en el contexto de una base de conocimiento, se define la abreviación $Do(\delta, s, s')$. Se dice que $Do(\delta, s, s')$ se cumple siempre y cuando s'es la situación final (es decir, una hoja del árbol de situaciones posibles) que sigue a la ejecución de una acción compleja δ, comenzando en la situación s. Dada la base de conocimiento de un agente (KB) y un procedimiento genérico δ, se puede instanciar δ con respecto a KB y a la situación actual S_0 encontrando una instanciación válida para las variables de la situación s. Debido a que las situaciones son simplemente la historia de las acciones a partir de S_0, la instanciación para s define una secuencia de acciones que lleva a una terminación exitosa del procedimiento δ: $KB \models (\exists s)Do(\delta, s, s')$.

Es importante observar que los programas escritos en ConGolog no son programas en el sentido convencional. A pesar de que poseen la estructura compleja de un programa incluyendo sentencias iterativas y del tipo “if-then-else”, tienen la característica de que no son necesariamente determinísticos. En lugar de retornar una única secuencia de acciones válidas, los programas ConGolog sirven para agregar restricciones temporales al árbol de situación de una KB, como se muestra en la figura 3.2. Asimismo, eliminan ciertas ramas del árbol de situación (indicadas con las “X”), reduciendo de esta manera el espacio de búsqueda para instanciar el procedimiento genérico. En el contexto de un agente para la construcción de itinerarios a través de la compra de pasajes aéreos, la reducción de una rama podría deberse a la imposibilidad de encontrar un vuelo disponible en una compañía determinada.

ConGolog provee un predicado *Desirable(a,s)* que permite incorporar restricciones y preferencias de usuario a los programas, lo que deriva en la reducción del árbol a aquellas situaciones que son deseables para el usuario. Ya que los procedimientos genéricos y las restricciones de personalización del usuario simplemente sirven para restringir la posible evolución de las acciones, depende de cómo sean definidas pueden jugar diferentes roles. En un extremo, el procedimiento genérico simplemente acota el espacio de búsqueda posible. En el otro extremo, un procedimiento genérico puede retornar una única secuencia de acciones, al igual que un programa convencional.

A pesar de las facilidades que ofrece para la programación de agentes en la Web, ConGolog cuenta con algunos inconvenientes que imposibilitan su aplicación en el contexto actual de la Web Semántica, destacándose los que se listan a continuación:

- Los Servicios Web que utiliza un agente deben estar referenciados en base a conceptos expresados en DAML. Si bien DAML es un lenguaje que posee muchas características beneficiosas para la definición de la semántica de los servicios, no es posible en ConGolog expresar semántica en un lenguaje diferente, lo que le quita flexibilidad.
- El espacio de búsqueda que el agente explora para hallar los Servicios Web que son relevantes a sus tareas es muy restringido, dado que sólo se limita a buscar en su base de conocimiento. ConGolog no considera la utilización de infraestructura estándar de búsqueda más poderosa, tal es el caso de UDDI.
- El modelo de ejecución de agentes de ConGolog combina e intercala la ejecución *online* de Servicios Web que reúnen información (no alteradores del mundo que conoce el agente) con la simulación *offline* de los servicios que sí producen cambios en el mundo. Este modelo opera bajo la suposición de que hay una persistencia razonable entre la información reunida y una completa independencia entre ambos tipos de servicios. Sin embargo, existen situaciones en que dichas suposiciones no se cumplen, tal es el caso de un sitio que requiere registración o ingreso (*login*) antes de acceder a sus recursos.

3.2.2. IG-Jade-PKSlib

IG-Jade-PKSlib (Martínez y Lespérance, 2004a) es un tookit para el desarrollo de sistemas multiagentes inteligentes e interoperables, principalmente destinados a proveer y componer Servicios Web. IG-Jade-PKSlib combina tecnologías de agentes y planning existentes tales como IndiGolog (Giacomo y Levesque, 1999) (un lenguaje de programación basado en modelos), JADE (Bellifemine et al., 1999) (una plataforma de agentes) y el sistema de planning PKS (Petrick y Bacchus, 2002; Petrick y Bacchus, 2003), construyendo así un tookit integrado que permite aprovechar las bondades de cada una de ellas.

IndiGolog es un lenguaje de programación de agentes que es parte de la familia de lenguajes de programación lógicos Golog desarrollados por el Cognitive Robotics Group de la Universidad de Toronto. IndiGolog provee facilidades para la programación de agentes autónomos mayormente gracias a su soporte para la creación y ejecución en dominios dinámicos de los cuáles se tiene poco conocimiento. También provee mecanismos para el monitoreo de ejecución y re-planning. Esta última capacidad es muy importante para la interacción con Servicios Web, ya que el agente puede registrar las prestaciones de cada servicio en particular y cuánto

control se delega a éstos. Por último, IndiGolog está creado principalmente para el diseño de agentes autónomos sin tener en cuenta la existencia o presencia de otros agentes.

JADE (Java Agent DEvelopment framework) es un *framework* basado en JAVA para el desarrollo de aplicaciones multiagentes. JADE provee interoperabilidad con otras plataformas y aplicaciones de agentes que son compatibles con las especificaciones de la FIPA (Foundation for Intelligent Physical Agents) (The Foundation for Intelligent Physical Agents, 1997) y soporte para varios tipos de ontologías. Sin embargo, JADE no provee mecanismos de razonamiento para los agentes, salvo una interfaz a JESS (Java Expert System Shell) (Friedman-Hill, 2003) que presenta limitaciones con respecto a razonamiento en dominios dinámicos. Por esta razón, IG-Jade-PKSlib utiliza JADE sólo como un componente para la búsqueda de los Servicios Web requeridos por los agentes.

El toolkit IG-Jade-PKSlib está esencialmente motivado por el problema de la composición de Servicios Web o WSC (Web Service Composition). El problema de WSC puede definirse como sigue: dado un conjunto de Servicios Web y alguna tarea genérica u objetivo a ser alcanzado definidos por un usuario, encontrar de forma automática una composición de los Servicios Web disponibles para satisfacer el objetivo (McIlraith y Son, 2002).

El problema de WSC es visto en el toolkit como un problema de ejecución de planes con conocimiento incompleto. Para la creación y ejecución de dichos planes, IG-Jade-PKSlib utiliza PKS (Planning with Knowledge and Sensing), un algoritmo de planning derivado a partir de una generalización de STRIPS (Fikes y Nilsson, 1990). En STRIPS, el estado del mundo está representado por una base de datos y las acciones disponibles están representadas como actualizaciones a dicha base de datos. En vez de una única base de datos, PKS utiliza un conjunto de bases de datos que representan el conocimiento del agente en vez del estado del mundo. Las acciones se modelan como modificaciones al conocimiento del agente, actualizando en consecuencia el conjunto de bases de datos mencionado. Las bases de datos en cuestión son cuatro, cada una conteniendo un tipo de conocimiento particular:

- K_f: Contiene cualquier hechos positivos o negativos conocidos por el agente. K_f también puede almacenar fórmulas que especifican el conocimiento acerca del valor de verdad de una función para determinados valores, por ejemplo, $p(a, b)$, $\neg q(c)$, $f(a) = b$.

- K_w: Almacena aquellas fórmulas cuyo valor de verdad es conocido por el agente. En particular, K_w puede contener una conjunción de fórmulas atómicas sin variables. Intuitivamente, en tiempo de construcción de un plan, $K(\alpha)$ significa que el agente conoce α o conoce $\neg\alpha$. El agente resolverá esta disyunción en tiempo de ejecución del plan. Esta base de datos se utiliza para modelar el efecto de acciones de percepción durante la creación de un plan. Las acciones de percepción son aquellas que no modifican el estado del mundo, por ejemplo, un Servicio Web que retorna la distancia entre dos ciudades determinadas.

- K_v: Contiene información acerca de los valores de las funciones. En particular, K_v puede almacenar términos de función no anidados cuyos valores son conocidos por el agente en tiempo de ejecución. K_v es usada para modelar los efectos de las acciones de percepción que retornan valores numéricos.

- K_x: Almacena información acerca de conocimiento disyuntivo (mediante el operador de

disyunción exclusiva) de literales de la forma $(l_1|l_2|...|l_n)$. Intuitivamente, esta fórmula representa el hecho de que el agente sabe que exactamente un literal l_i es verdadero.

Notar que la única forma de conocimiento incompleto que puede ser expresada es la falta completa de conocimiento acerca de un átomo, dejándolo fuera de K_f, y el conocimiento que sólo un elemento dentro de un conjunto finito de literales es verdadero utilizando K_x.

PKS permite la construcción de objetivos simples y complejos. Los objetivos simples se representan como consultas primitivas, que a su vez pueden tomar una de las siguientes formas:

1. $K(\alpha)$, ¿se sabe si α es verdadero?
2. $K(\neg\alpha)$, ¿se sabe si α es falso?
3. $K_w(\alpha)$, ¿el agente conoce el valor de verdad de α?
4. $K_v(t)$, ¿el agente conoce el valor de t?
5. La negación de alguna de las anteriores

En cualquiera de los casos, α representa alguna fórmula atómica y t representa un término sin variables. En este sentido, los objetivos complejos se representan como consultas que incluyen consultas primitivas, conjunción y disyunción de consultas, y cuantificación de consultas sobre un conjunto de objetos conocidos.

Las acciones en PKS son especificadas en términos de tres componentes: los parámetros, las precondiciones y los efectos. En la tabla 3.1 se muestra un ejemplo de especificación para las acciones *open(r)* y *search(r)*. La primera corresponde a una acción física para abrir la puerta de una habitación. La segunda es una acción (que produce conocimiento) para percibir la presencia de una persona en la habitación. Como se mencionó anteriormente, los efectos de las acciones se traducen en actualizaciones a alguna de las bases de datos. Existen a su vez otras reglas que permiten especificar efectos adicionales que corresponde a invariantes a nivel de conocimiento del agente, denominadas DSUR (Domain Specific Update Rules). En la tabla 3.2 se muestra un ejemplo de una DSUR. Esta regla captura el efecto adicional de la acción de "marcar" la búsqueda como finalizada cuando alguna persona es encontrada en alguna habitación. Por último, el antecedente de una DSUR puede ser cualquier fórmula objetivo, pero el consecuente debe ser siempre una actualización a alguna base de datos.

Acción	Precondición	Efectos
open(r)	*K(room(r)) K(¬opened(r))*	*add(*K_f*, opened(r))*
search(r)	*K(room(r)) ¬K(found(r)) K(opened(r))*	*add(*K_w*, found(r))*

Tabla 3.1: Acciones *open* y *search*

Domain Specific Update Rules
$K(room(r)) \wedge K(found(r)) \Rightarrow add(K_f,\ done)$

Tabla 3.2: Un ejemplo de una DSUR

Una de las características que hacen a PKS atractivo para resolver el problema de composición automática de servicios es su habilidad de generar planes condicionales parametrizables (conteniendo variables) en presencia de conocimiento incompleto. Adicionalmente, los planes son generados a nivel de conocimiento sin tener en cuenta todas las formas posibles en las que el mundo puede ser configurado o modificado por los efectos de las acciones. Finalmente, estos planes se componen de acciones primitivas que se corresponden unívocamente con cada Servicio Web disponible. Los planes son creados, ejecutados y monitoreados por agentes programados en IndiGolog a través de una interfaz denominada IndiGolog-PKS.

Aunque el enfoque de IG-Jade-PKSlib es novedoso, resulta poco flexible, pues obliga al programador a utilizar el algoritmo de planeamiento provisto y, como se verá en el capítulo 8, presenta graves problemas de performance que imposibilitan su uso en aplicaciones Web donde el tiempo de ejecución es un requerimiento prioritario. Por otra parte, IG-Jade-PKSlib no provee manejo de semántica de los Servicios Web en absoluto, lo que impacta negativamente en la autonomía de los agentes desarrollados.

Por último, y similar a ConGolog, los Servicios Web que un agente utiliza son sólo el conjunto de acciones primitivas que han sido declaradas en la especificación de dominio asociada. Como se vio anteriormente, la creación de una especificación de dominio (acciones y DSUR) es una tarea difícil y engorrosa. Debido a esto, adicionar un conjunto nuevo de Servicios Web como acciones primitivas de un agente requiere un esfuerzo de programación considerable.

3.2.3. Jinni

Jinni (Java INference engine and Networked Interactor) (Tarau, 1998) es un lenguaje interpretado basado en Prolog para la programación de agentes móviles. El intérprete está desarrollado para ser usado como una herramienta de unificación de componentes de procesamiento de conocimiento y objetos JAVA, principalmente en aplicaciones cliente/servidor en redes heterogéneas. Comercialmente, se utiliza como estándar para conectividad e interoperabilidad de electrodomésticos y computadoras.

La terminología de Jinni está basada en tres conceptos principales: cosas, lugares y agentes. Las *cosas* están representadas por términos Prolog conteniendo constantes y variables utilizadas durante la ejecución de los programas. Los *lugares* son procesos que ejecutan en varias computadoras. Están constituidos por un servidor que recibe requerimientos en un determinado puerto, y un componente *blackboard* (Shaw y Garlan, 1996) para escritura sincronizada de información a través de operaciones tipo Linda (Gelernter, 1985), y transacciones remotas (consultas a predicados y ejecución remota de código). Por último, los agentes son una colección de *threads* que ejecutan un conjunto de objetivos Prolog, posiblemente diseminados a través de varios *lugares* y generalmente ejecutando transacciones locales y remotas en coordinación con otros agentes.

Un agente puede iniciar múltiples motores de inferencia, cada uno encargado de construir una parte de la solución total. Cuando un motor finaliza su computación, copia sus resultados -

de forma sincronizada - al motor padre que le dio origen. Cada uno de éstos motores ejecutan en *threads* separados, y tienen la posibilidad de comunicarse mediante el intercambio de mensajes, espacios de memoria compartida o a través del *blackboard* local (la base de datos de cada intérprete Jinni). Las principales operaciones que un agente puede realizar sobre el *blackboard* incluye la publicación de términos Prolog, la espera por la escritura de un determinado término, y la obtención de la lista de los términos que están presentes actualmente en el blackboard y que unifican con una cláusula Prolog dada.

La migración de agentes en Jinni está soportada a través de movilidad indicada explícitamente por el programador en el código fuente del agente, mediante un modelo de movilidad fuerte (Fuggetta et al., 1998). Todos los *threads* correspondientes a los motores internos creados durante la ejecución de un agente pueden ser enviados a un servidor mediante la operación *move*, lo que causa la interrupción de la ejecución del motor, la migración de su estado de ejecución (variables, pila, etc.) y la reanudación de la ejecución en el sitio destino. El *thread* cliente que envió la computación al servidor espera a que ésta finalice. Cuando esto sucede, la computación retorna con los resultados obtenidos mediante la operación *return*, posibilitando al thread cliente continuar con su ejecución normal.

Además de la movilidad de computaciones, Jinni permite la ejecución remota de código Prolog. Un *thread* puede enviar código Prolog a un servidor para su ejecución sin necesidad de mover su computación, invocando a un predicado predefinido llamado *run*, pasándole como parámetro el objetivo particular que quiere resolverse.

Jinni ha ido evolucionando paulatinamente desde sus inicios, convirtiéndose en una herramienta flexible con extensiones poderosas de orientación a objetos y agentes para permitir la integración de componentes JAVA y .NET en aplicaciones distribuidas (BinNet Corporation, 2004). Para ello, Jinni permite transacciones multiusuario sincronizadas e interopera con BinProlog (Tarau, 1992), un sistema Prolog de alta performance utilizado para operaciones de gran volumen en servidores Web. Cada requerimiento de un cliente al servidor es tratado por un Agente Servidor que es capaz de invocar Servicios Web. En general, los agentes destinados a interactuar con la Web se construyen como una combinación de Servicios Web específicos, y un conjunto de *wrappers* que permiten interactuar y aislar los detalles de implementación de cada servicio. Por ejemplo, Jinni cuenta con un *wrapper* de la API de Google de Servicios Web, lo que permite la utilización de los servicios de búsqueda basados en *keywords* a los agentes Jinni. Por último, ha sido desarrollado un novedoso sistema de chat (Tarau y Figa, 2004) basado en agentes conversacionales Jinni que utilizan ontologías y Servicios Web, éstos últimos encargados de proveer facilidades de inferencia sobre bases de datos de conocimiento de gran tamaño expresado en RDF (Resource Description Framework) (Lassila y Swick, 1999).

Un aspecto que dificulta considerablemente la utilización de Jinni lo constituye su limitado esquema de migración de agentes. Un agente Jinni está formado por un *thread* más el código de la cláusula en ejecución actual. Para clarificar el mecanismo de migración mencionado, considérese el siguiente ejemplo:

```
..., there, move,
println(on_server), length([1,2,3], X),
return,
println(back), ...
```

Al ejecutarse los predicados *there* y *move*, el lenguaje migra el estado del *thread* más el código de la cláusula en ejecución a otro servidor Jinni. Después, evalúa length([1,2,3],X) y retorna. Notar que el código asume que length se encuentra definido en el servidor. Si esto no es así, surge un gran problema, pues Jinni no provee otros mecanismos para definir la *clausura* de código de un agente; esto es, no permite seleccionar el conjunto de reglas Prolog que el agente lleva consigo al servidor remoto.

Las alternativas que Jinni ofrece para solucionar estas deficiencias adolecen de dos problemas principales: no son transparentes para el programador, y en ciertos casos generan problemas extras a los que intentan resolver. Por ejemplo, un esquema muy utilizado para migrar la totalidad del código de un agente consiste en definir un predicado moveAgent, que coloca dicho código en una lista Prolog, transfiere el *thread* al servidor, restaura el código contenido en la lista en el lado servidor, y finalmente continúa con la ejecución del *thread*. El problema de esta solución consiste en determinar qué código pertenece al agente. Adicionalmente, puede ocurrir que parte del código de un agente se encuentre ya definido en el servidor remoto, produciéndose duplicaciones de código, o incluso fallas en la ejecución, por tratarse de cláusulas con el mismo nombre y argumentos, pero con diferente significado.

Sumado a las desventajas que posee para la programación de agentes móviles, Jinni presenta algunos inconvenientes y limitaciones para el desarrollo de aplicaciones en la Web Semántica. Por un lado, para *cada* Servicio Web utilizado, se debe proveer un *wrapper* específico que aisla los detalles de implementación (tipos de datos) y de comunicación con el servicio. Una alternativa un tanto más apropiada para administrar la ejecución de Servicios Web la constituye el uso de bibliotecas de invocación tales como WSIF (Web Service Invocation Framework) (Duftler et al., 2001), que aprovechan las tecnologías estándares de la Web Semántica (SOAP, WSDL, etc.). Por otra parte, Jinni no cuenta con mecanismos para expresar y manipular la semántica de los recursos y servicios. Como consecuencia, las aplicaciones basadas en Jinni que utilizan *metadatos* deben realizar por sí mismas las inferencias semánticas, generalmente a través de algoritmos *ad-hoc* complejos que poseen un escaso nivel de reutilización.

3.2.4. MWS

MWS (Mobile Web Services) (Ishikawa et al., 2004b) es un *framework* genérico para la implementación de Servicios Web móviles, esto es, Servicios Web con capacidades de migración brindadas por agentes móviles. En MWS, un servicio móvil se compone de una combinación de descripciones del proceso de interacción con otros servicios, componentes o recursos necesarios para su ejecución, y políticas de migración. De esta manera, es posible implementar una amplia gama de servicios móviles variando la combinación de los tres elementos mencionados. Adicionalmente, permite que el comportamiento migratorio de cada servicio sea configurable de acuerdo al ambiente en el cual el servicio es provisto. MWS representa un avance hacia el desarrollo de Servicios Web *inteligentes* (Preece y Decker, 2002; Paolucci y Sycara, 2003).

La motivación principal de MWS descansa en que los Servicios Web y los agentes móviles se complementan entre sí. Los Servicios Web complementan los agentes móviles con la habilidad de componer dinámicamente sistemas heterogéneos poco acoplados, mientras que, a la inversa, los agentes móviles complementan a los Servicios Web con la posibilidad de acceder inteligentemente a diversos recursos o servicios. La figura 3.5 muestra el modelo general de MWS donde un servicio móvil se compone de una combinación de los siguientes tres elementos:

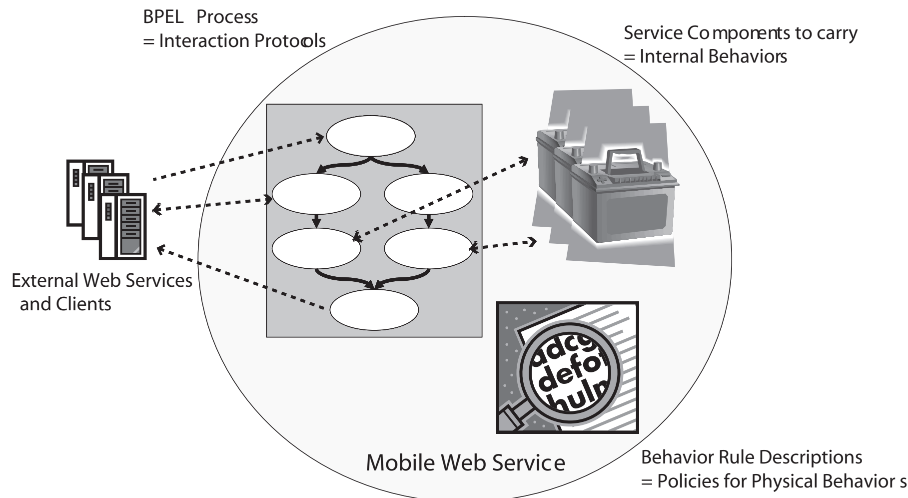


Figura 3.5: Componentes de un servicio móvil

- *Descripciones de las interacciones del proceso*: Expresan en qué orden y bajo qué condiciones el servicio móvil interactúa con otros servicios. En otras palabras, estas descripciones denotan protocolos de interacción del servicio móvil. Para expresar las descripciones mencionadas, se utiliza BPEL4WS. BPEL4WS es un lenguaje basado en XML para describir procesos ejecutables que proveen servicios generales integrando servicios existentes más simples, desde una perspectiva de *workflow*.

  Los proceso en BPEL4WS puede ser descritos de dos formas diferentes. Los procesos *ejecutables* modelan el comportamiento real de un participante en una interacción o negocio. Los *protocolos* o procesos *abstractos*, en contraste, utilizan descripciones de proceso que especifican el comportamiento relativo al intercambio de mensajes de cada una de las partes involucradas en una transacción, sin revelar el comportamiento interno. Valiéndose de estas dos formas de descripción, BPEL4WS permite describir protocolos de transacciones comerciales complejos cuyas acciones de bajo nivel están representadas por Servicios Web reales.

- *Servicios empaquetados*: Existen ciertos servicios que son parte de un servicio móvil, transportados e invocados por él, y se corresponden al comportamiento interno del servicio móvil. La funcionalidad de llevar un componente de servicio permite las interacciones locales entre el servicio y sus pares, aprovechando la movilidad para su ejecución en el sitio que actualmente tenga los recursos computacionales adecuados. Por el momento, MWS restringe los servicios de componentes a comportamiento implementado con clases JAVA, utilizando la extensión WSDL a JAVA (Apache Software Foundation, 2003).

- *Políticas de comportamiento físico*: Sumado a los dos ítems anteriores, existen políticas para decidir el comportamiento físico de un servicio móvil, por ejemplo, comportamiento de migración o clonación. Actualmente, MWS propone para este fin el uso de reglas simples para asociar comportamientos físicos con actividades del proceso BPEL4WS, por ejemplo, hacer que el servicio móvil ejecute una serie de actividades en un sitio cuando el servicio migre a éste.

Existen básicamente dos reglas primitivas para expresar el comportamiento físico de los servicios móviles. Por un lado, la regla *migrate* indica que el servicio móvil se mueve a un cierto sitio y ejecuta un cierto servicio empaquetado (bloque) remotamente si la migración resulta exitosa (figura 3.6(a)), excluyendo otra posibilidad de migración asociada con bloques concurrentes. Esto permite asociar una tarea determinada a un bloque para luego enviarlo a interactuar localmente con los recursos de un sitio remoto, mientras que otros bloques permanecen ejecutando localmente con comportamiento diferente. Por otro lado, por medio de la regla *launch* el servicio móvil crea un nuevo servicio "hijo" y le asocia el bloque de ejecución actual (figura 3.6(b)). Dado que BPEL4WS es capaz de describir controles de concurrencia capaces de esperar la finalización de dos actividades diferentes, la regla sólo especifica las actividades a ser ejecutadas por el servicio móvil recientemente creado. En la figura, la regla muestra que el servicio móvil crea un sub-servicio que luego accede a otro servicio de información localizado en el sitio remoto.

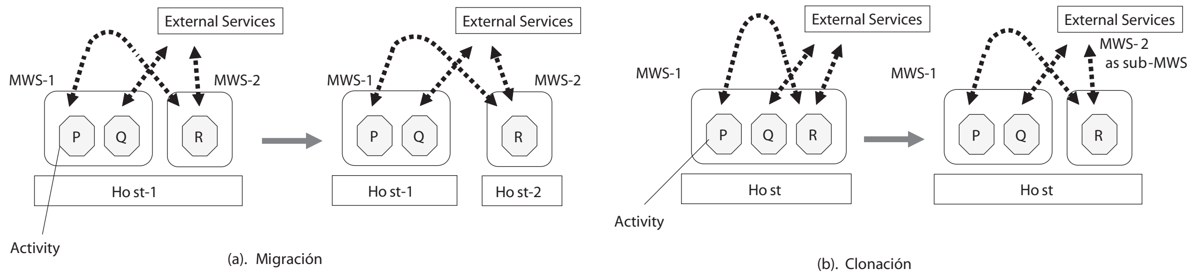


Figura 3.6: Migración y clonación de un MWS

El prototipo del sistema MWS se encuentra implementado utilizando Bee-Gent (Kawamura et al., 1999), un *framework* basado en JAVA para la programación de sistemas de agentes móviles. En este prototipo, un servicio móvil se materializa como un agente móvil JAVA que interpreta descripciones registradas por uno o varios proveedores de servicios. Cada vez que recibe una solicitud de servicio, la plataforma crea una instancia de un agente intérprete que resuelve - de acuerdo a su comportamiento, servicios internos y reglas asociadas - el requerimiento o tarea en cuestión. Para más detalles de implementación del prototipo descrito ver (Ishikawa et al., 2004a).

De forma similar a WSMF, MWS confía en el lenguaje BPEL4WS para la representación del proceso o *workflow* asociado a cada servicio que determina la interacción de éste último con otros servicios. En consecuencia, MWS adolece de problemas parecidos a los que presenta WSMF: poca flexibilidad, ya que los Servicios Web incluidos en las descripciones BPEL4WS son fijos y determinados al momento de la creación de dichas descripciones, y falta de manejo de la semántica de los Servicios Web utilizados. Adicionalmente, MWS está implementado utilizando Bee-Gent, una plataforma de agentes móviles basada en JAVA, lenguaje no apto para manipular las actitudes mentales de los agentes ni proveer comportamiento inteligente. Así, el programador debe desarrollar complejos algoritmos de inferencia que en general son difíciles de implementar, y que una vez implementados ofrecen una flexibilidad muy pobre (Amandi et al., 1999).

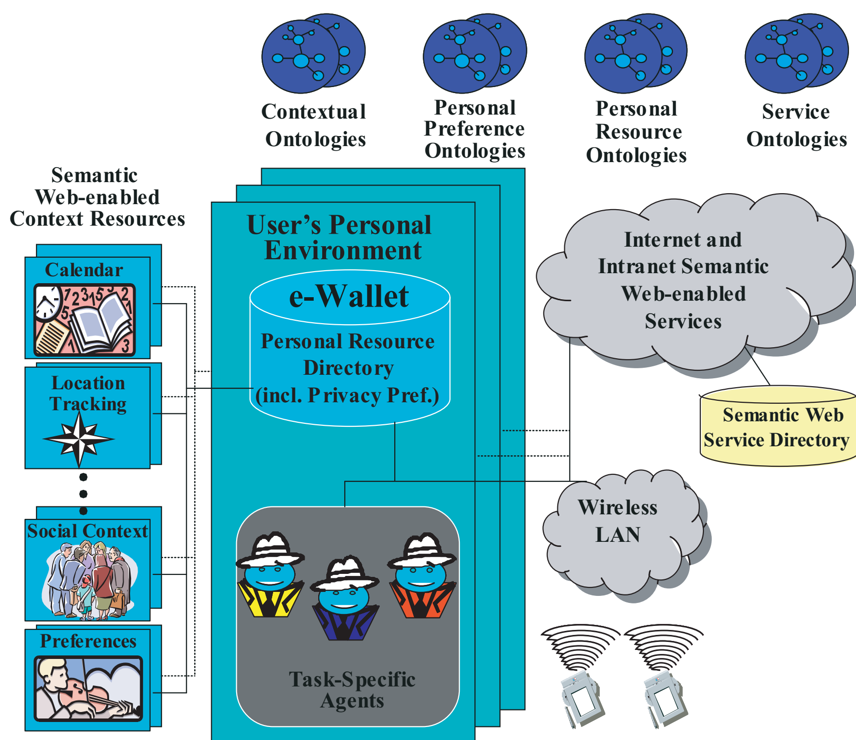


Figura 3.7: Vista general de la arquitectura de MyCampus

3.2.5. MyCampus

MyCampus (Sadeh et al., 2003) es un ambiente basado en agentes para la implementación de servicios móviles semánticos en el contexto de un campus, desarrollado en la Universidad de Carnegie Mellon. El ambiente está materializado a través de una colección de agentes configurables, capaces de encontrar y acceder automáticamente a recursos de los usuarios (agendas, ubicación actual, preferencias gastronómicas, etc.) y Servicios Web, a medida que asisten a los usuarios en tareas tales como la planificación de salidas y esparcimiento, la organización de grupos de estudios o el filtrado de correo y mensajería. Los agentes realizan la búsqueda de Servicios Web de interés basados en reglas específicas del dominio de aplicación, información de contexto y preferencias específicas del usuario.

Para permitir una mejor automatización de las tareas, MyCampus define un conjunto de ontologías para la descripción de recursos personales, un gran número de atributos contextuales, preferencias de usuario y funcionalidad de Servicios Web. La información contextual y otros detalles personales acerca de un usuario son accedidos a través de una *Billetera Electrónica Semántica*, de acuerdo a los privilegios de acceso establecidos por el dueño de la misma. A través del tiempo, el usuario puede crear e ingresar nuevos agentes a su contexto para descubrir y acceder a información de contexto relevante y detalles personales. Similarmente, las ontologías de servicio definidas facilitan la introducción de nuevos Servicios Web descubiertos por los agentes que ejecutan en el contexto propio de un usuario.

La arquitectura de MyCampus se muestra en la figura 3.7. Cada usuario tiene una billetera electrónica que actúa como un directorio semántico de recursos personales (agenda, funciona-

lidad de ubicación, repositorios de preferencias, suscripciones a servicios, etc.) posibilitando a los agentes descubrir y acceder automáticamente a los recursos personales de un usuario. Como se mencionó anteriormente, los usuarios pueden agregar a lo largo del tiempo nuevos agentes a su ambiente o contexto personal, que llevarán a cabo sus tareas teniendo en cuenta la información presente de la billetera electrónica. Adicionalmente, los agentes externos pueden acceder a esta información de acuerdo a restricciones y permisos impuestos por el dueño de la billetera electrónica. En general, los accesos a un recurso personal de un usuario puede involucrar:

- Solicitar el valor de un atributo de contexto en un momento dado (por ejemplo, ¿dónde está el usuario ahora? o ¿el usuario tiene alguna reunión programada entre la tarde de hoy y las 13:00 hs. de mañana?).
- Solicitar actualizaciones periódicas acerca del valor de un atributo de contexto (por ejemplo, se necesita saber la ubicación del usuario cada 5 minutos).
- Solicitar actualizaciones a medida que un atributo de contexto particular cambia (por ejemplo, se requiere una notificación cada vez que el usuario ingresa o se retira de un edificio).
- Modificar el valor de un atributo de contexto (por ejemplo, agregar una nueva reunión a la agenda).

Los atributos de contextos soportados actualmente son la ubicación del usuario dentro del campus, su agenda, sus relaciones sociales (por ejemplo amigos y compañeros de clases), y ciertas preferencias (por ejemplo, preferencias gastronómicas y de filtrado de mensajes). Por otra parte, los Servicios Web actualmente implementados incluyen sólo recomendación de películas e información del estado del tiempo. No es posible obtener y utilizar Servicios Web ofrecidos por sitios externos (por ejemplo, sitios de la Internet), sino que la búsqueda de servicios relevantes se limita a los nodos en los cuales la plataforma MyCampus se encuentre instalada. Por último, es preciso notar que MyCampus no provee soporte inteligente para la interacción entre diferentes usuarios, o lo que es lo mismo, cada agente actúa independientemente en favor de su dueño sin tener conocimiento de los demás.

3.2.6. CALMA

CALMA (Context-Aware Lightweight Mobile Agent) (Chuah et al., 2004) es un *framework* para la construcción de agentes móviles basados en el modelo BDI (Beliefs-Desires-Intentions) (Rao y Georgeff, 1995). Los planes de los agentes son expresados en un lenguaje basado en XML que facilita la especificación de planes basados en condiciones de contexto. El *framework* permite la construcción de agentes "livianos" mediante la utilización de infraestructura que permite la obtención de planes bajo demanda basados en los cambios de condiciones contextuales, y la utilización de Servicios Web para delegar computaciones costosas a servidores externos. Otra característica interesante es que CALMA ha sido implementado como un *add-on* a la plataforma de agentes móviles Grasshopper (Breugst et al., 1999).

El objetivo principal de CALMA es la provisión de infraestructura adecuada para la creación de agentes móviles basados en el modelo BDI, capaces de procesar información de contexto y

ejecutar en dispositivos móviles con fuertes restricciones de hardware. Como se muestra en la figura 3.8, la infraestructura de CALMA cuenta con tres componentes principales, a saber:

- *Agente de Tareas*: Extiende el modelo de agentes BDI convencional para proveer la funcionalidad necesaria para la implementación de agentes con tareas específicas. Un agente BDI CALMA está representado por un objeto que puede ser manipulado por los agentes móviles construidos mediante el *toolkit* subyacente (Grasshopper en este caso). Para esto, CALMA utiliza un motor de ejecución de agentes especial construido por sobre el motor de agentes de Grasshopper.

- *Servidor*: Este componente incluye al CalmaAgentManager, el MatchMakerAgent y el PlanManager. Los servicios provistos por los dos últimos se implementan como Servicios Web, que se comunican con los demás componentes utilizando los mecanismos de comunicación externos provistos por Grasshopper.

  El componente CalmaAgentManager procesa solicitudes de servicio provenientes del componente de Dispositivo Móvil. Para esto, provee una lista de servicios disponibles, contacta al MatchMakerAgent para encontrar un agente que provea el servicio solicitado, y finalmente contacta a éste último para obtener una respuesta al servicio. Por otra parte, la imposibilidad de encontrar un agente que resuelva el servicio solicitado es informada al dispositivo móvil a través de un agente mensajero.

  El componente MatchMakerAgent permite que los Agentes de Tareas puedan publicar los servicios que ofrecen. Los servicios publicados son mantenidos en una lista que asocia la descripción del servicio con el agente particular que lo provee. Naturalmente, los Agentes de Tareas pueden eliminar un servicio de esta lista mientras ejecutan un plan de acción.

  Los planes de los Agentes de Tareas se almacenan en un repositorio global de planes. Basado en el nombre de un servicio solicitado, el componente PlanManager recupera el plan desde el repositorio. Finalmente, pueden ser agregados nuevos planes en forma dinámica, y ser utilizados en tiempo de ejecución por los Agentes de Tareas.

- *Dispositivo Móvil*: Contiene un componente llamado MDAAgentManager que provee una interfaz entre el usuario y los servicios y agentes CALMA en un dispositivo móvil particular. Cuando un usuario realiza una solicitud de servicio, el componente de Dispositivo Móvil determina en primer lugar si existe un Agente de Tareas local capaz de atender la solicitud en cuestión, antes de contactar al componente Servidor. Si no es así, se contacta al componente Servidor para encontrar un servicio similar a través del MatchMakerAgent. Luego, el componente Servidor contacta al Agente de Tareas asociado al servicio hallado. Este agente migra al dispositivo móvil, reúne información acerca del contexto, selecciona un plan de acción y comienza a ejecutar las acciones definidas en el plan escogido.

Como se mencionó anteriormente, el prototipo del sistema está implementado utilizando la plataforma de agentes Grasshopper. Los componentes encargados de la interacción con Servicios Web corren en el servidor Web Apache Tomcat, y utilizan Apache Axis - una implementación eficiente del protocolo SOAP - para la invocación de los Servicios Web. En lo que a semántica respecta, no se evidencia a partir de la documentación existente de CALMA que el componente MatchMakerAgent lleve a cabo el matching de Servicios Web basado en la información semántica asociada a éstos.

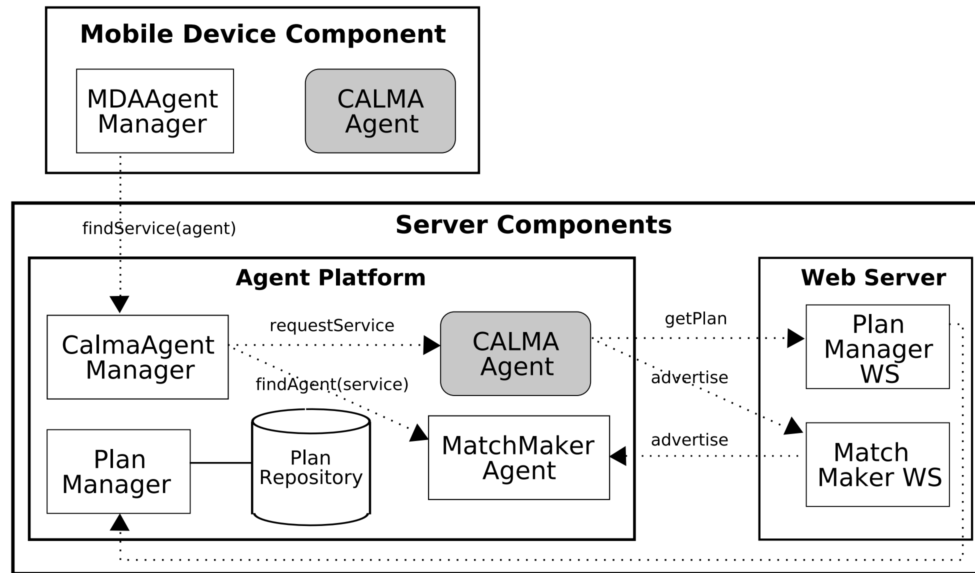


Figura 3.8: Vista general de la infraestructura de CALMA

3.2.7. TuCSoN

TuCSoN (Omicini y Zambonelli, 1998; Ricci et al., 2001) es una infraestructura que provee servicios de coordinación en el contexto de un MAS (Multi-Agent System) (Wooldridge, 1997; Weiss, 1999). A grandes rasgos, un MAS es un conjunto de agentes inteligentes que interactúan en un ambiente común, focalizados en la resolución de problemas en forma colaborativa y coordinada. Cada agente intenta satisfacer objetivos particulares como así también objetivos globales o comunes de todos los agentes involucrados en el MAS.

Los servicios de coordinación en TuCSoN son materializados a través de *artefactos de coordinación*, que son entidades persistentes en tiempo de ejecución que cada agente del MAS utiliza para llevar a cabo sus actividades de forma coordinada. Básicamente, la infraestructura general de TuCSoN puede ser resumida describiendo cuatro modelos fuertemente relacionados, vale decir, el modelo de *coordinación*, el modelo de *topología*, el modelo de *organización* y el modelo de *seguridad*:

- El modelo de coordinación de TuCSoN está basado en artefactos de coordinación denominados *centros de tuplas*, que son *blackboards* reactivos basados en lógica que actúan como mediadores de la interacción entre los agentes, de acuerdo a las reglas o leyes de coordinación que definen su comportamiento reactivo. Los centros de tuplas se utilizan para controlar el acceso de los agentes a ciertos recursos del sistema (por ejemplo, una impresora), actuando como *proxies* a los recursos y a las políticas de coordinación embebidas que gobiernan el acceso a éstos.

- Desde el punto de vista topológico, los artefactos de coordinación están organizados mediante nodos distribuidos a través de una red, organizados en dominios articulados (Cremonini et al., 1999). Un dominio está caracterizado por un nodo *gateway* y un conjunto de nodos denominados *places*. Resumidamente, un *place* contiene centros de

tuplas específicos para las aplicaciones, mientras que un *gateway* almacena centros de tuplas utilizados para la administración, manteniendo información relativa a cada *place*.

- Desde el punto de vista organizacional, TuCSoN adopta un modelo organizacional basado en roles. De esta manera, cada aplicación está estructurada como un conjunto de sociedades que definen roles específicos. Un agente participa en la organización jugando uno o varios roles, en la misma o en varias sociedades. La participación es completamente dinámica: los agentes pueden dinámicamente - y físicamente, en el caso de agentes móviles - entrar y salir de la organización, activando diferentes conjuntos de roles. La principal abstracción para lograr esta flexibilidad es el Agent Coordination Context (ACC), que puede ser visto como una entidad de tiempo de ejecución creada por la infraestructura que actúa como una interfaz utilizada por los agentes para interactuar con los recursos y demás agentes del ambiente. En particular, un ACC define los patrones de acciones/operaciones que el agente puede ejecutar sobre los centros de tuplas de la organización, de acuerdo a sus roles. Así, un agente que pretende integrarse a la organización debe obtener primero un ACC de un *gateway* configurado apropiadamente a través del *Servicio de Bienvenida* provisto por el *gateway*. Luego de esto, el agente puede usar el ACC ejecutando primitivas de coordinación para acceder a los centros de tuplas. Cuando el agente quiere retirarse de la organización, se quita del ACC, cerrando así su sesión.

- El modelo de seguridad está estrictamente relacionado con el modelo de organización y el modelo de topología. La noción de rol permite el control de acceso basado en roles, donde cada rol determina el conjunto de acciones y protocolos de interacción que el agente puede ejecutar y utilizar, respectivamente.

La infraestructura de TuCSoN ha sido integrada con tecnología de Servicios Web, como se describe en (Morini et al., 2004). Esta integración apunta principalmente a la provisión de Servicios Web de coordinación a los agentes TuCSoN. Un rol clave en la materialización de la integración de TuCSoN y Servicios Web es el desempeñado por la abstracción ACC. Debido a que es la interfaz que los agentes adquieren y utilizan para interactuar con los centros de tuplas, convertir cada ACC en un Servicio Web permite a los agentes explotar los beneficios de los protocolos estándares para Servicios Web sin necesidad de APIs de comunicación propietarias.

La figura 3.9 muestra una vista general del esquema de coordinación de TuCSoN implementado mediante Servicios Web. Un agente interactúa con el *Servicio Web de Bienvenida* para negociar la adquisición de un ACC. Una vez obtenido hecho esto, el agente utiliza la descripción WSDL asociada al ACC que describe las operaciones disponibles en el mismo.

El Servicio Web de Bienvenida está provisto a modo de *wrapper* al Servicio de Bienvenida convencional de TuCSoN, en un nodo *gateway* de la organización. Básicamente, representa un punto de acceso para los agentes Web que desean entrar a la organización TuCSoN. El WSDL del servicio en cuestión debe ser conocido por los agentes, o puede ser descubierto a partir de registros UDDI (Universal Description, Discovery and Integration) (W3C Consortium, 2001) donde la organización TuCSoN lo publica. Este WSDL contiene operaciones básicas para unirse a la organización, cuyos argumentos son el identificador del agente solicitante y los roles particulares que deben ser activados junto con el ingreso. El Servicio Web de Bienvenida interactúa con el Servicio de Bienvenida convencional de TuCSoN para establecer el resultado

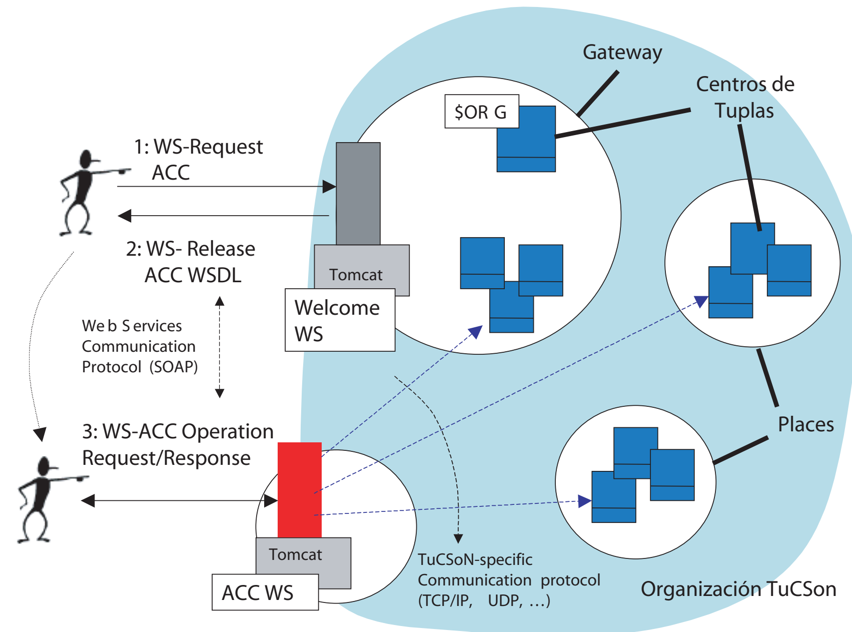


Figura 3.9: Una representación del espacio de coordinación de TuCSoN integrado con Servicios Web

de la negociación: en caso exitoso, el primero retorna al agente solicitante un WSDL que es utilizado para acceder, mediante mensajes SOAP, las operaciones provistas por el ACC adquirido.

TuCSoN es una de las pocas herramientas que permite implementar Servicios Web basados en agentes, y al mismo tiempo posibilita desarrollar agentes que explotan los Servicios Web existentes en la plataforma. Sin embargo, un aspecto importante que no ha sido considerado aún está relacionado con la semántica de los servicios de coordinación, en particular en términos de razonadores y ontologías asociadas a los Servicios Web asociados. Una investigación preliminar en este sentido lo constituye (Viroli y Ricci, 2004).

3.3. Discusión

La tabla 3.3 resume las principales características de los sistemas analizados. Es preciso destacar la importancia que ha adquirido la movilidad de código en el contexto de los lenguajes y plataformas de agentes. Así, la mayor parte de las herramientas analizadas proveen algún tipo de soporte para la programación de agentes *móviles*, ya sea en base a modelos de movilidad fuerte o débil.

La provisión de mecanismos de representación y manipulación de la información semántica asociada a un Servicio Web por parte de una plataforma es un requisito fundamental para asegurar un nivel adecuado de automatización en la interacción entre los agentes y servicios. De no cumplirse, cada agente debe ser instruido en forma explícita por el programador acerca de los servicios a utilizar y cómo interactuar con ellos, afectando negativamente el grado de autonomía de las aplicaciones construidas.

	Movilidad	Manejo de semántica	Servicios Web inteligentes	Búsqueda de servicios
ConGolog		$\surd$		a nivel agente
IG-Jade-PKSlib				a nivel agente
Jinni	$\surd$	$\surd$		no posee
MWS	$\surd$		$\surd$	no posee
MyCampus	$\surd$	$\surd$	$\surd$	a nivel plataforma
CALMA	$\surd$			a nivel plataforma
TuCSoN	$\surd$		$\surd$	a nivel Web

Tabla 3.3: Principales características de las plataformas analizadas

La mayoría de las plataformas expuestas emplean mecanismos muy rudimentarios para el manejo de la semántica de los Servicios Web o, en el caso extremo, no proveen mecanismos en absoluto. ConGolog utiliza DAML para describir las ontologías asociadas a los Servicios Web, pero no permite el uso de otros lenguajes de representación de información semántica más elaborados, tal es el caso de OWL. Un problema similar presenta Jinni, dado que sólo permite el uso de RDF como lenguaje para la descripción de conceptos. La única herramienta que en principio presenta un manejo adecuado de la semántica de los Servicios Web es MyCampus.

Una característica de suma importancia en toda plataforma que interactúa con Servicios Web es qué facilidades provee para hallar los servicios disponibles. Es claro que los mecanismos de descubrimiento de servicios que exploran en un espacio de búsqueda amplio permiten a las aplicaciones y agentes efectuar un mejor aprovechamiento de los servicios existentes. Para el caso de las herramientas analizadas, dos de ellas (Jinni y MWS) no poseen un mecanismo de búsqueda de servicios, de modo que los Servicios Web que un agente utiliza deben ser determinados en forma estática, es decir, al momento de programar dicho agente. Por su parte, ConGolog e IG-Jade-PKSlib limitan el espacio de búsqueda a los servicios que inicialmente conoce un agente, o en otras palabras, los servicios con los cuales un agente interactúa son los que han sido declarados en su base de conocimiento al momento de su implementación. Un mecanismo un tanto más adecuado es el provisto por MyCampus y CALMA, mediante el cual los agentes son capaces de hallar cualquier servicio que se encuentre publicado en algún nodo de la red donde la aplicación ha sido instalada. Finalmente, el esquema de búsqueda que resulta más flexible es el provisto por TuCSoN, ya que permite descubrir Servicios Web tanto en los nodos de una organización TuCSoN, y al mismo tiempo consultar y procesar el contenido de registros UDDI ubicados en nodos externos diseminados a través la Web.

Un concepto interesante que está cobrando importancia en el contexto de la Web Semántica es el de *servicio inteligente*, es decir, un servicio accesible a través de la Web cuya funcionalidad está implementada en base a uno o más agentes inteligentes. En este sentido, tres de las herramientas analizadas (MWS, MyCampus y TuCSoN) ofrecen la posibilidad de desarrollar Servicios Web *inteligentes*. Particularmente, TuCSoN permite desarrollar aplicaciones basadas en agentes que interactúan con Servicios Web, o viceversa.

Finalmente, un aspecto importante que no se desprende de la tabla anterior es la influencia que ha tenido JAVA en la programación de sistemas distribuidos basados en agentes móviles que proveen acceso a recursos Web. Así, una variedad de plataformas han sido desarrolladas en este lenguaje. Sin embargo, a pesar que JAVA posee numerosas ventajas respecto de otros

lenguajes para soportar el desarrollo de tales sistemas, no permite transferir el estado de ejecución de un *thread* debido a problemas de portabilidad. En consecuencia, la mayoría de las plataformas que proveen soporte de movilidad de código basadas en JAVA sólo soportan movilidad débil, lo que dificulta enormemente la programación de las aplicaciones móviles.

3.4. Conclusiones

En este capítulo se describieron en detalle algunos de los proyectos, plataformas y lenguajes relacionados a la Web Semántica más importantes. En particular, se expusieron las iniciativas más transcendentes para la integración de sistemas heterogéneos a gran escala a través de la tecnología de Servicios Web, tales como Grid Computing, WSMF y Agentcities, y los lenguajes y herramientas concretos que han sido desarrollados para la permitir la interacción de agentes y Servicios Web en el contexto de una Web inteligente, soportada mayormente a través del uso de ontologías.

Es preciso notar que las propuestas y herramientas analizadas presentan algunas falencias que dificultan la construcción de agentes conscientes de la semántica tanto de los datos como de los Servicios Web existentes. En consecuencia, dichas herramientas fallan a la hora de ofrecer un soporte para el desarrollo de aplicaciones autónomas que aprovechen los recursos presentes en la Web, lo que impacta negativamente en la flexibilidad, autonomía y adaptabilidad de las mismas. A grandes rasgos, los problemas que presentan los trabajos expuestos a lo largo del capítulo son los siguientes:

- **Los modelos de ejecución de los agentes están basados en técnicas de planning**. Históricamente, y debido a su complejidad computacional inherente, los algoritmos de planning han sido utilizados mayormente en la construcción de soluciones en forma *offline*, o en su defecto, en los casos en que la cantidad de información a procesar es escasa. En este sentido, la utilización de este enfoque en el contexto de la Web Semántica resulta en la materialización de aplicaciones que ofrecen una performance pobre y poco aceptable, dada la gran cantidad de recursos presentes en la Web, y poco flexible, debido al alto grado de dinamismo tanto de la información como de los servicios. En el capítulo 8 se verificará empíricamente esta afirmación a través de algunos ejemplos concretos.

- **Hacen caso omiso de la semántica de los Servicios Web**. Paradójicamente, gran parte de las herramientas que permiten construir agentes integrados a la Web Semántica no proveen mecanismos para manejar la información semántica asociada a cada Servicio Web. En consecuencia, el programador debe indicar los servicios específicos (generalmente las URLs de los documentos WSDL correspondientes) con los cuales un agente interactúa a lo largo del tiempo. Esto proviene del hecho de que el agente no es capaz de comprender los conceptos involucrados en un servicio, limitación que le imposibilita llevar a cabo una búsqueda de servicios según la funcionalidad requerida, por ejemplo. Claramente, este enfoque resulta poco flexible, y al mismo tiempo no permite un adecuado aprovechamiento de los servicios publicados a través de la Web.

  Por otro lado, muchos de los enfoques existentes obligan a que el modelo de ejecución de las aplicaciones y los agentes esté dado por uno o más *workflows* descritos en

BPEL4WS, lenguaje que se ha impuesto rápidamente como estándar para la descripción de procesos de negocios e industriales. Sin embargo, BPEL4WS no es más expresivo que otros lenguajes similares ya propuestos para la descripción de *workflows* (van der Aalst et al., 2003), y posee un manejo muy rudimentario de la semántica de los servicios referenciados en cada *workflow* (Staab et al., 2003).

- **Poseen mecanismos de descubrimiento de servicios poco adecuados en términos del tamaño del espacio de búsqueda**. Típicamente, las plataformas que adolecen de este problema limitan el descubrimiento de Servicios Web a los nodos de la red LAN (Local Area Network) donde la misma ha sido instalada. Más aún, en ciertos casos, el espacio de búsqueda explorado por una aplicación o agente se reduce a un determinado conjunto de servicios especificados por el desarrollador, y cuyo contenido no puede ser modificado ni actualizado. Sin embargo, para lograr una mayor utilidad práctica de las aplicaciones construidas, como así también un mejor aprovechamiento de los recursos Web existentes, es deseable contar en principio con un mecanismo eficiente de descubrimiento de Servicios Web cuyo espacio de búsqueda sea lo más amplio posible, idealmente pensado para operar en el contexto de la Internet.

En el siguiente capítulo se presentará una plataforma de agentes móviles denominada MoviLog (Zunino et al., 2002), una herramienta para el desarrollo de agentes inteligentes basados en lógica, que implementa un novedoso modelo de movilidad denominado Movilidad Reactiva por Fallas (MRF) (Zunino et al., 2005b). MRF y MoviLog han sido generalizados para permitir la programación de agentes móviles en el contexto de la Web Semántica, utilizando un enfoque que da solución a los problemas anteriores. En el siguiente capítulo se describirán MRF y MoviLog y, más tarde, el capítulo 5 presentará la generalización mencionada, lo que representa la propuesta central de la presente tesis.

Capítulo 4

Movilidad Reactiva por Fallas

MRF (Movilidad Reactiva por Fallas) (Zunino et al., 2005b) es un novedoso modelo de movilidad no explotado aún por las herramientas de programación de agentes móviles existentes. MRF es un mecanismo basado en el concepto de falla de Prolog. En Prolog, una falla ocurre cuando el objetivo en evaluación no puede ser deducido a partir de las cláusulas contenidas en la base de datos. MRF aprovecha este concepto para automatizar la movilidad, migrando un agente a un sitio remoto cuando se produce una falla en un objetivo especialmente declarado por el programador. De esta manera, la evaluación del objetivo que falló puede reintentarse en el sitio remoto, haciendo uso de los recursos allí disponibles. Este mecanismo facilita enormemente el desarrollo de agentes móviles, ya que permite al programador concentrarse en la implementación del comportamiento propio de cada agente, delegando ciertas decisiones de movilidad a MRF.

MoviLog (Zunino et al., 2002) es una plataforma para la construcción de agentes móviles inteligentes, siguiendo un modelo de movilidad *fuerte* (Fuggetta et al., 1998), donde el estado de ejecución de un agente es transferido junto a éste durante la migración, de forma transparente al programador. MoviLog es una extensión de JavaLog (Amandi et al., 2005), un *framework* para la programación orientada a agentes. Además de proveer primitivas básicas para la movilidad fuerte a los agentes JavaLog, el aspecto más importante de MoviLog radica en que implementa el modelo MRF, facilitando la programación de agentes móviles *inteligentes*.

MoviLog aprovecha los beneficios ofrecidos por los lenguajes Java y Prolog, ya que está construido como una extensión a JavaLog. Por un lado, el uso de Prolog permite al programador representar adecuadamente el estado mental de los agentes, haciéndolo fácilmente interpretable por los algoritmos de toma de decisiones (Amandi et al., 2005). Por otra parte, Java posee características positivas que posibilitan la migración de código a bajo nivel, tales como independencia de la plataforma, soporte multi-thread y serialización de objetos (Wong et al., 1999).

El objetivo de este capítulo es introducir las características más relevantes de MRF y MoviLog. El mismo está organizado de la siguiente forma: en la sección 4.1 se describe el modelo conceptual de MRF, detallando las particularidades de sus agentes, las facilidades ofrecidas para delegar decisiones de movilidad, y su soporte de ejecución. Más tarde, las secciones

4.2 y 4.3 presentan el lenguaje y la plataforma MoviLog, respectivamente. Finalmente, en la sección 4.4 se presentan las conclusiones del capítulo.

4.1. MRF

Los agentes móviles son entidades que se caracterizan por tener la habilidad de migrar su ejecución de sitio en sitio buscando la mejor forma de interactuar con la red y sus recursos. Así, por ejemplo, un agente móvil puede decidir si accede mediante interacciones no locales a un recurso localizado en un sitio remoto, o migra su ejecución a la ubicación en la que el recurso se encuentra, dependiendo tanto del costo de las interacciones no locales como del costo de la migración.

Incluso cuando el paradigma de agentes móviles cuenta con conocidas ventajas respecto de otros paradigmas de sistemas distribuidos (Lange y Oshima, 1999; Gray et al., 2000), su utilización está aún limitada a aplicaciones de poco tamaño (Marques et al., 2001; Kotz et al., 2002). Entre otros factores, esto es causado por el hecho de que los agentes móviles son inherentemente más complejos que los sistemas estacionarios tradicionales. Claramente, los desarrolladores de agentes móviles deben proveer mecanismos para decidir cuándo un agente migra y hacia dónde lo hace, lo que agrega un nivel de complejidad extra al desarrollo de los mismos. En este sentido, MRF es un modelo de movilidad que ha surgido con el objeto de reducir el esfuerzo de programación de agentes móviles, automatizando ciertas decisiones sobre movilidad.

MRF se basa en la idea de que la movilidad es una características *ortogonal* al resto de atributos o capacidades que un agente puede poseer, tales como inteligencia (capacidades de razonamiento y aprendizaje) y agencia (autonomía y autoridad) (Bradshaw, 1997). MRF explota esta ortogonalidad permitiendo que el programador focalice su esfuerzo principalmente en la funcionalidad estacionaria (comportamiento) propia de los agentes móviles, y delegue aspectos relacionados con la movilidad en un sistema multi-agente que forma parte de la plataforma de ejecución.

A continuación se describe el modelo conceptual de MRF.

4.1.1. Modelo conceptual

MRF es un mecanismo de migración que actúa en forma *reactiva*. Se entiende por migración reactiva a la movilidad que es disparada por agentes o entidades diferentes del agente que migra. A diferencia de la migración reactiva, la migración *proactiva* es originada por el propio agente. Esto implica que el programador debe indicar explícitamente, en el código que implementa dicho agente, los puntos de ejecución en los cuales éste último debe migrar y, adicionalmente, hacia dónde.

En el caso de MRF, las entidades externas que disparan la migración de los agentes móviles son agentes estacionarios, es decir, agentes que no poseen movilidad y que forman parte de la plataforma de ejecución (figura 4.1). Dichos agentes estacionarios intervienen en el curso normal de ejecución de un agente móvil cuando se produce una *falla*.

Conceptualmente, una falla se produce cuando el acceso a un recurso no local no puede ser realizado. En términos de Prolog, una falla ocurre cuando un agente móvil evalúa un objetivo

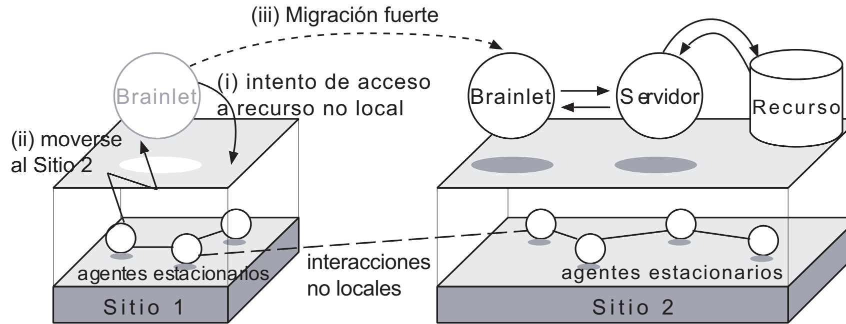


Figura 4.1: Movilidad Reactiva por Fallas: Vista general

especialmente declarado que no puede ser deducido a partir de las cláusulas disponibles en el sitio local ni en su base de conocimiento ((i) de la figura 4.1). Nótese que la evaluación de un objetivo puede fallar al no encontrarse recursos tales como datos, código, páginas, aplicaciones Web, entre otros. Por ejemplo, supóngase el caso de un agente situado en un sitio s_1 que intenta acceder a datos que se encuentran disponible en otros sitios s_2, s_3 y s_4. En ese momento actúa MRF y los agentes estacionarios escogen el sitio destino para el agente móvil ((ii) en la figura 4.1).

En resumen, MRF tiene por objetivo automatizar decisiones relacionadas con movilidad tales como:

- *dónde migrar*: en el ejemplo anterior se analizaría a cuál de los sitios migrar. Esta decisión podría tomarse en función del tráfico de red, la carga de CPU en los sitios, la velocidad de procesamiento, etc. En el ejemplo anterior se asumió que los datos están disponibles en tres sitios. El problema es que un agente puede no saber en qué sitios se ofrece un determinado recurso. En consecuencia, MRF también debe poseer conocimiento acerca de qué recursos son provistos en cada sitio que acepta la ejecución de agentes móviles.
- *cuándo migrar*: cuando se produce una falla, MRF podría transferir la ejecución del agente móvil a s_2, s_3 o s_4 indistintamente. Sin embargo, también podría ser conveniente acceder a los datos mediante interacciones remotas o incluso copiar parte de ellos al sitio donde se encuentra el agente. Estas decisiones deberían tomarse analizando los costos de tiempo y tráfico de red involucrados.

Como puede observarse a partir de la figura 4.1, el modelo conceptual de MRF consta de unidades de ejecución o agentes (*Brainlets*), mecanismos utilizados por los agentes para permitir o no actuar a MRF, y una plataforma de soporte de movilidad compuesta por agentes estacionarios. Estos tres componentes se describen en las secciones 4.1.2, 4.1.3 y 4.1.4, respectivamente.

4.1.2. Brainlets

La unidades de ejecución o agentes considerados por el modelo MRF se denominan *Brainlets*. Un Brainlet es un agente móvil conformado por los componentes que se listan a continuación (ver figura 4.2):

- *código*: está compuesto por cláusulas Prolog que implementan el comportamiento propio del Brainlet.
- *datos*: son cláusulas Prolog que representan los datos o conocimiento que posee un Brainlet.
- *estado de ejecución*: consiste de uno o más *threads* de ejecución.
- *protocolos*: indican qué cláusulas deben ser tratadas por el mecanismo de movilidad reactiva por fallas y, consecuentemente, cuáles generan la migración del Brainlet a un sitio diferente del sitio actual en el cual se encuentra ejecutando.

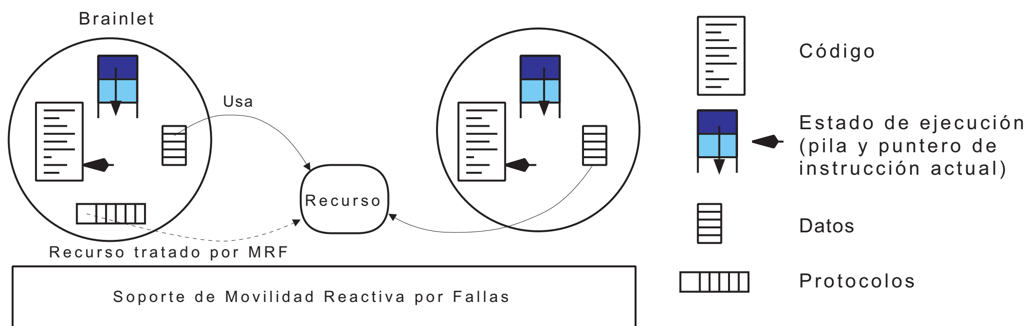


Figura 4.2: Componentes de un Brainlet

Un Brainlet es capaz de migrar proactivamente, o en otras palabras, por decisión propia; o reactivamente, es decir, por algún evento externo al agente. El mecanismo de MRF se basa en este último tipo de migración, por lo que el resto del capítulo tratará mayormente sobre migración reactiva. La siguiente sección describe cómo se utilizan los protocolos para delegar decisiones de movilidad en MRF.

4.1.3. Protocolos

Los protocolos son utilizados para describir los predicados que deben ser tratados por MRF cuando la evaluación de éstos falla. Genéricamente, los protocolos representan un medio para describir la interfaz de acceso de, potencialmente, cualquier tipo de recurso presente en una red, e indicar a MRF que actúe ante la falla en el acceso a tales recursos.

En su forma más básica, un protocolo posee la forma *protocol(functor,aridad)*. Tal declaración de protocolo en un Brainlet indica al mecanismo de movilidad reactiva que debe tratar las fallas de los predicados con la forma $functor(arg_1, ..., arg_{aridad})$ y buscar otros sitios que ofrezcan cláusulas con el mismo protocolo, para poder así migrar el Brainlet cuando se produzca una falla en un objetivo con el mismo functor y aridad. Por ejemplo, el protocolo *protocol(cd,6)* denota cláusulas de la forma *cd(Id, Author, Title, Style, #Songs, Price)*, que representan datos sobre discos compactos de música.

A continuación se presentará un ejemplo práctico con el fin de aclarar el concepto de protocolo y su relación con MRF. Cabe destacar que MRF no prescribe ninguna notación sintáctica

particular para la programación de agentes, sino que la sintaxis que se presentará corresponde a la materialización de MRF dada por MoviLog, que será expuesta en la sección 4.2.

El ejemplo en cuestión consisten de un Brainlet cuyo objetivo es recolectar los valores de temperaturas generados en diferentes sitios de medición, y luego calcular el promedio de dichos valores. Cada punto de medición está representado por un sitio MRF, y compuesto por un proceso que periódicamente almacena su medición - digamos T - en la base de datos Prolog local, en la forma de un predicado *temperature(T)*. El código que implementa al Brainlet es como sigue:

```
PROTOCOLS
  protocol(temperature, 1)
CLAUSES
  average(List, Avg):- ...
  getTemp(Curr, List):-
    temperature(T),
    thisSite(S),
    M = measure(T, S),
    not(member(M, Curr)), !,
    getTemp([M|Curr], List).
  getTemp(Curr, Curr).
  average(Avg):-
    getTemp([], List),
    average(List, Avg).
  ?-average(Avg).
```

La idea del programa es forzar al Brainlet a visitar todos los sitios, solicitando en cada caso la temperatura medida por el sitio actual, para luego calcular el promedio de esos valores (proceso no relevante aquí). El punto potencial de activación de MRF está localizado en el predicado *temperature(T)*, como se ha subrayado en el código. Como el lector podrá observar, la sección PROTOCOLS define que dicho predicado debe ser tratado por MRF. Como consecuencia, si la evaluación de *temperature(T)* falla en un sitio S - por ejemplo, el agente ya posee registrada la medición de ese sitio - la plataforma transferirá al Brainlet a un sitio que contenga definiciones para tal predicado. La evaluación de *getTemp* finalizará exitosamente una vez que todos los sitios que ofrecen la cláusula *temperature/1* han sido visitados.

En la próxima sección se describe el soporte de ejecución de Brainlets con MRF.

4.1.4. Soporte de ejecución de MRF

MRF se basa en la utilización de un sistema multiagente formado por agentes estacionarios responsables de tomar las decisiones sobre movilidad cuando ocurre una falla en un predicado Prolog declarado como protocolo. En dicho sistema se distinguen dos tipos de agentes estacionarios: Protocol Name Server (PNS) y de movilidad. Los PNS son responsables de mantener información sobre los protocolos ofrecidos en los sitios de una red. Cada vez que un Brainlet necesita cierto recurso no presente en el sitio local, se consulta al PNS de ese sitio para obtener sitios alternativos a los cuales migrar para interactuar con el recurso buscado.

Los agentes estacionarios de movilidad (de aquí en más, agentes de movilidad) actúan conjuntamente con los PNSs para determinar el próximo destino de un Brainlet, y construir un itinerario en caso de que el recurso requerido se encuentre en varios lugares, tomando en

consideración el tráfico de red entre los sitios, la velocidad de los vínculos, u otro mecanismo provisto por el desarrollador.

A continuación se describen los PNSs y agentes de movilidad.

4.1.4.1. PNS

Cada sitio que pertenece a una red MoviLog posee un componente denominado Protocol Name Server (PNS). Las principales responsabilidades de dicho componente son mantener y proveer información sobre los protocolos ofrecidos por cada sitio capaz de ejecutar Brainlets. Esta información es utilizada por los agentes de movilidad para construir y mantener itinerarios de Brainlets que utilizan MRF. La funcionalidad provista por los PNSs relacionada con mantenimiento de información sobre protocolos incluye operaciones de:

- *altas*: para que un recurso de un sitio pueda ser utilizado por Brainlets, se debe registrar el protocolo de acceso de dicho recurso en el PNS del sitio. Posteriormente, el PNS anuncia a otros sitios de la red, bajo una política preestablecida, la disponibilidad de dicho protocolo. Así, por ejemplo, en la aplicación de búsqueda de mediciones de temperatura, cada uno de los sitios debería registrar *protocol(temperature,1)*. De esta forma, las cláusulas Prolog presentes en los sitios con esos protocolos podrían ser utilizadas por Brainlets provenientes de otros sitios.
- *bajas*: de forma similar, cuando un sitio deja de ofrecer un determinado tipo de recurso, debe informarle al PNS, el cual actuará en consecuencia informando a otros PNS de la red sobre la baja. Para el ejemplo analizado, este sería el caso de un sitio que ya no oficia como punto de medición para la temperatura actual.

Por otro lado, otro tipo de funcionalidad que proveen los PNSs está relacionada con consultas que realizan los agentes de movilidad localizados en el mismo sitio que el PNS, como también PNSs pertenecientes a otros sitios de red:

- *consultas de agentes de movilidad*: cuando los agentes de movilidad determinan el itinerario de un Brainlet, consultan al PNS del sitio local qué sitios ofrecen recursos con el protocolo del objetivo cuya evaluación falló. Una vez obtenida esta información, construyen un itinerario y transfieren el Brainlet al siguiente destino. En caso de producirse fallas adicionales sobre ese mismo objetivo, el itinerario es actualizado consultando a los PNSs con el fin de considerar otros sitios que hayan dejado de ofrecer recursos, problemas de comunicación a través de la red, etc.
- *consultas de PNS*: cuando un sitio se incorpora a una red anuncia que es capaz de ejecutar Brainlets. En consecuencia, otros sitios de la red responden, a través de su PNS, con el fin de que el nuevo sitio conozca a sus pares. De esta forma se establece una *red lógica*, formada por los sitios de una red física capaces de ejecutar Brainlets que se conocen entre sí. Finalmente, el nuevo miembro de la red lógica consulta a los demás sitios los protocolos que cada uno de ellos ofrece.

En la sección siguiente se expone la función que tienen los agentes de movilidad dentro del soporte de MRF.

4.1.4.2. Agentes de Movilidad

Los agentes de movilidad son agentes estacionarios responsables de tomar decisiones sobre movilidad cuando ocurre una falla. Las decisiones que estos agentes son capaces de tomar pueden clasificarse en dos tipos:

- *acceso a recursos*: cuando se produce una falla, MRF intenta migrar la ejecución del Brainlet a un sitio donde se ofrecen recursos descritos por el protocolo del objetivo que causó dicha falla. Sin embargo, podría ser conveniente - por cuestiones de performance, por ejemplo - acceder al recurso mediante interacciones remotas o incluso copiar el recurso desde otro sitio al sitio donde se encuentra el Brainlet. A pesar de que el modelo conceptual de MRF tiene en cuenta métodos diferentes a la movilidad para el acceso a los recursos, MoviLog no los implementa. En el capítulo 5 se analizará en detalle cómo se extendió MRF y MoviLog para efectivamente proveer los métodos de acceso mencionados.

- *construcción de itinerarios*: cuando un agente de movilidad decide que un Brainlet debe migrar a otro sitio, solicita al agente PNS local la lista de sitios que ofrecen un protocolo determinado. En este punto, MRF envía el Brainlet a cada uno de los sitios de a uno por vez, asegurándose de no visitar sitios caídos o ya visitados, y de visitar nuevos sitios que ingresen a la red lógica. Los agentes de movilidad son responsables de construir un itinerario a través de determinados sitios conociendo previamente los sitios que ofrecen recursos con el protocolo buscado. Los agentes determinan el orden en el cual se visitarán los sitios basándose en métricas tales el como tráfico de red, la carga de cada sitio, etc., con el fin de hacer el mejor uso posible de los recursos de hardware y software del sistema.

Hasta aquí, se ha descrito el modelo de movilidad MRF. En la siguiente sección se describirá MoviLog, una plataforma de agentes móviles basados en Prolog que implementa MRF. Como se verá en el capítulo 5, MRF y MoviLog han sido extendidos para soportar interacción con recursos Web, en especial Servicios Web Semánticos.

4.2. El lenguaje MoviLog

MoviLog extiende el lenguaje multi-paradigma JavaLog con mecanismos para movilidad de código. El principal concepto que introduce JavaLog es el de *módulo lógico*. Un módulo lógico es una secuencia de cláusulas Prolog que guardan una relación determinada, por ejemplo, el conocimiento que un agente cree verdadero, o la implementación de un algoritmo de ordenamiento. MoviLog define un tipo especial de módulo lógico denominado *Brainlet*. Un Brainlet consta de tres secciones de código, donde las dos primeras son opcionales, a saber:

- PROTOCOLS: Define todas las cláusulas que activarán el mecanismo de MRF. Si el objetivo actual en evaluación falla, y si éste se corresponde con un protocolo declarado en esta sección, se mueve el Brainlet para que continúe su evaluación en otro sitio que contenga las definiciones necesarias.

La definición de cada protocolo se lleva a cabo mediante hechos Prolog de la forma protocol(functor,aridad), donde functor y aridad corresponden a la cláusula que en caso de falla disparará la migración del Brainlet.

- FETCH: contiene declaraciones de protocolos con la forma protocol(functor,aridad). Cuando falla un objetivo declarado en esta sección, se transfiere una cláusula con ese protocolo desde un sitio remoto. La definición de cada protocolo se realiza utilizando hechos Prolog de la forma protocol(functor,aridad,[$S_1, S_2, ..., S_n$]), donde functor y aridad poseen el mismo significado que en el caso de la sección anterior, mientras que el tercer argumento representa la lista de sitios donde se buscarán definiciones de la cláusula.

- CLAUSES: contiene cláusulas Prolog con el código y datos de un Brainlet.

Las siguientes secciones describen los aspectos más relevantes relacionados con el diseño e implementación de la plataforma MoviLog. Estos detalles serán de crucial importancia para comprender los conceptos que se introducirán en el siguiente capítulo.

4.3. La plataforma MoviLog

MoviLog es una extensión del *framework* JavaLog para soportar el mecanismo de Movilidad Reactiva por Fallas en la WWW. Las unidades de ejecución provistas por MoviLog son agentes móviles denominados *Brainlets*. El ambiente de ejecución de los Brainlets se denomina *Mobile Agent Resource servlet* (MARlet). Básicamente, un MARlet provee servicios de inferencia, razonamiento, comunicación, seguridad e identificación global a los Brainlets, como así también a aplicaciones legadas. Adicionalmente, y aún más importante, los MARlets ofrecen servicios de movilidad reactiva por fallas.

Los MARlets presentes en una red se organizan en *redes lógicas* (ver sección 4.1.4.1). Una red lógica es un conjunto de al menos dos MARlets distribuidos en varios sitios tal que todos los MARlets utilizan el mismo puerto TCP. Así, un sitio podría alojar uno o más MARlets, cada uno escuchando en un puerto diferente. En consecuencia, en una red física podría existir más de una red lógica, cada una perteneciendo a una aplicación de agentes móviles diferente. Los Brainlets sólo pueden moverse entre MARlets de la red lógica en la que fueron creados, por lo que no son capaces de interactuar - en principio - con Brainlets que residen en otras redes lógicas.

Un MARlet es un *servlet* Java (Hunter y Crawford, 1998) que ejecuta como extensión de un servidor de Web estándar. De esta forma, la plataforma MoviLog puede ser integrada en la WWW, permitiendo aprovechar tanto los recursos de información (páginas, bases de datos y programas) como el hardware disponible. En la figura 4.3 se muestran los principales componentes de los MARlets. Como se observa en el diagrama, un MARlet está constituido por los siguientes componentes: *Agent Manager*, el *RMF Manager*, *Security Manager*, *Inter-Agent Communication Manager* y *Application Gateway*.

En las secciones que siguen a continuación se describen cada uno de los componentes mencionados.

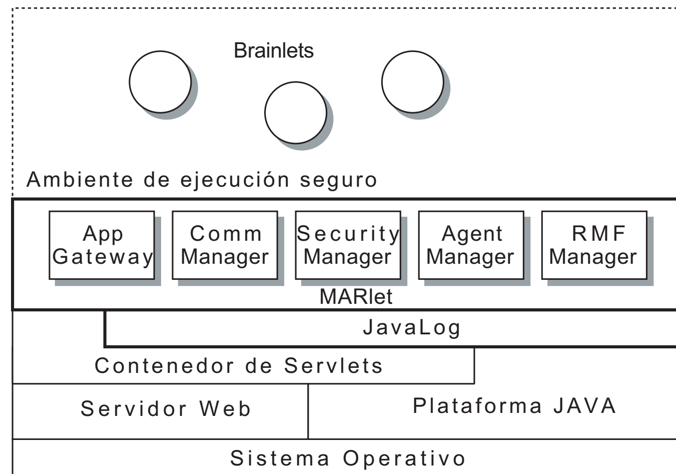


Figura 4.3: La plataforma MoviLog

4.3.1. El componente *Agent Manager*

El componente *Agent Manager* es el responsable de convertir el estado de un Brainlet en una representación transferible a través de la red, y a partir de esta representación, reconstruir el estado del Brainlet luego de haber sido recibido en el destino. Un MARlet requiere de dichos servicios para enviar y recibir Brainlets mediante el protocolo HTTP (Hyper Text Transfer Protocol).

El *Agent Manager* es clave tanto para la movilidad proactiva como para la movilidad reactiva por fallas, pues es quien posibilita la transferencia del *estado de ejecución* de los agentes y, por consiguiente, provee las bases para soportar migración fuerte. Nótese que transferir el estado de ejecución de un agente implica que la plataforma debe ser capaz de manipular el contador de programa, la pila, los puntos de *backtracking* y el estado de las variables, lo cual, al menos en Java estándar, no es posible sin modificar la máquina virtual. En este sentido, el hecho de que MoviLog extiende a JavaLog permite proveer esta funcionalidad, pues éste último representa el estado de ejecución de los programas mediante objetos Java que pueden ser manipulados y transferidos por la red sin mayores inconvenientes.

4.3.2. El componente *RMF Manager*

El componente *RMF Manager* es el responsable de implementar la funcionalidad para soportar el mecanismo de MRF. Este componente, a su vez, se subdivide en dos subcomponentes principales: *Protocol Name Server (PNS)* y *Mobility Manager (MM)*. El *Protocol Name Server* es el componente que mantiene la información de identificación de los demás MARlets de la red lógica, y de los protocolos que cada uno ofrece. Un protocolo describe la interfaz de acceso a un recurso representada mediante un predicado con cierto functor y aridad. Cada vez que un Brainlet necesita un recurso no presente en el MARlet local, se consulta al PNS para obtener los potenciales sitios que contienen el recurso requerido. Por ejemplo, si un Brainlet falla evaluando un objetivo *hardDisk(Size,RPM,Price)*, el PNS buscará los sitios que ofrecen recursos con el protocolo *hardDisk/3*. Por otro lado, el componente *Mobility Manager* consulta al PNS y construye un itinerario para el agente en cuestión que mantiene actualizado.

Para que MRF se adapte al dinamismo de los recursos de una red, cada vez que un MARlet ofrece un nuevo recurso, el soporte de MRF debe ser capaz de conocer dicho recurso. Para lograr esta funcionalidad, cada vez que un MARlet publica un recurso (por ejemplo, una base de datos o una aplicación), notifica al PNS local. Posteriormente, y de acuerdo a una política configurable en tiempo de ejecución, el PNS comunica a sus pares en los demás sitios de la red lógica acerca de la existencia de nuevos recursos en el sitio local.

4.3.3. El componente *Security Manager*

El *Security Manager* es responsable de la autenticación de un Brainlet previo a su ejecución. Sus servicios son utilizados por el componente *Agent Manager*, ya que es necesaria alguna forma de validación de los Brainlets que arriban a un MARlet antes de permitirle reanudar su ejecución o realizar alguna operación que involucre acceso a recursos.

Debido a que MoviLog es aún un prototipo académico, parte de la funcionalidad necesaria para construir aplicaciones reales basadas en Brainlets no está aún provista por la plataforma. En particular, el componente *Security Manager* no está aún diseñado ni implementado.

4.3.4. El componente *Inter-Agent Communication Manager*

El *Inter-Agent Communication Manager* tiene como finalidad principal facilitar la comunicación entre agentes diseminados en la red, haciendo transparente la ubicación de los mismos mediante un componente denominado *proxy*. El *proxy* provee transparencia en la ubicación de los Brainlets; esto es, oculta la ubicación física real de los Brainlets. Si un Brainlet desea comunicarse con otro lo hace a través del *proxy* local del mismo, que se encargará de redireccionar todos los mensajes hacia la ubicación actual del Brainlet receptor, para luego retornar la respuesta correspondiente al emisor del mensaje.

4.3.5. El componente *Application Gateway*

El *Application Gateway* provee un punto de entrada seguro a través del cual los agentes pueden interactuar con servidores de aplicación residentes en el sitio local. Un agente móvil que arriba a un sitio accede a servidores de aplicación locales (por ejemplo un servidor de base de datos o una lista de mail) a través del *Application Gateway* asociado al sitio. Adicionalmente, el *Application Gateway* provee servicios de inferencia y representación de actitudes mentales a aplicaciones de terceros que ejecutan remotamente en navegadores Web y PDAs (Personal Digital Assistant).

4.3.6. Brainlets

Las unidades de ejecución o agentes móviles de la plataforma MoviLog se denominan *Brainlets*. Los Brainlets son módulos lógicos que extienden el soporte básico de JavaLog para permitir movilidad fuerte de tipo proactiva y reactiva por fallas.

La figura 4.4 muestra el ciclo de vida correspondiente a cada Brainlet. El mismo comienza cuando el componente *Agent Manager* crea un Brainlet. Luego de la creación, el *Agent Manager* inicia su ejecución (estado *Ejecutando*). Un agente en ejecución puede ser suspendido por

diversos motivos, tales es el caso de la espera por un determinado evento externo, ejecutar la primitiva de migración (*move*) o haber fallado un predicado declarado como protocolo que activa MRF. En todos estos casos, la ejecución del Brainlet se detiene (estado *Suspendido*). En caso de haber ejecutado la instrucción *move* o si el soporte de MRF determina que es necesario migrar el Brainlet, éste es transferido a otro MARlet. Cuando la migración se completa y se restaura el estado de ejecución interno, el Brainlet se encuentra suspendido hasta que el *Agent Manager* reinicie su ejecución. Finalmente, la ejecución de un Brainlet finaliza cuando no tiene más objetivos para resolver o algún componente externo lo destruye.

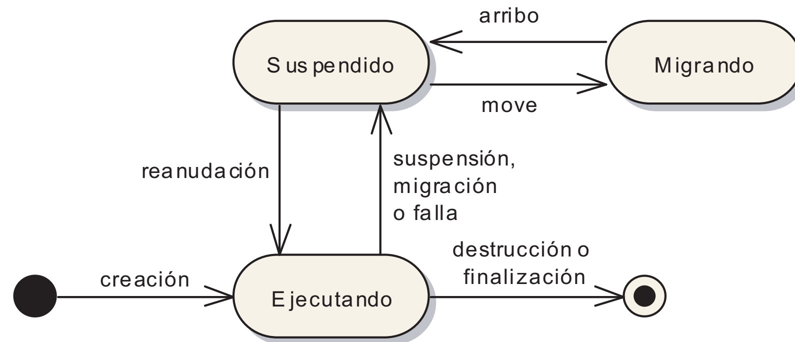


Figura 4.4: Ciclo de vida de un Brainlet

Un Brainlet está implementado como un intérprete Prolog convencional extendido, incorporando funcionalidad para la movilidad de código. Cada vez que se inicia la ejecución de un nuevo agente o arriba el estado de ejecución de un Brainlet a un MARlet, se crea una instancia de este intérprete. Adicionalmente, cada Brainlet se compone de código, datos y estado de ejecución, éste último tanto local como distribuido. El estado de ejecución *local* está constituido por las variables y estructuras de datos básicas que representan el estado del intérprete en el sitio actual. Por otra parte, el estado de ejecución *distribuido* contiene la información relacionada a la evaluación de la consulta entre los diferentes sitios, tal como MARlets visitados o puntos de *backtracking* a considerar situados en diferentes sitios.

4.4. Conclusiones

En este capítulo se describió MRF (Movilidad Reactiva por Fallas), un modelo de movilidad que permite concentrarse mayormente en el desarrollo de la funcionalidad estacionaria de un agente móvil, dado que automatiza ciertas cuestiones que determinan el comportamiento migratorio del mismo. Adicionalmente, se introdujo MoviLog, una plataforma que implementa MRF y cuyo objetivo es facilitar la programación de agentes móviles inteligentes en la WWW.

MoviLog está implementado como una extensión al *framework* JavaLog, lenguaje que está diseñado para ser extendido, en contraposición con la mayoría de las implementaciones de Prolog. JavaLog provee facilidades para la representación y la manipulación del conocimiento de los agentes, dado que es en sí la implementación de un intérprete Prolog *extensible*. Adicionalmente, el hecho de que JavaLog haya sido desarrollado en JAVA lo hace ideal para su utilización en la materialización de una plataforma de agentes móviles, pues es totalmente portable a todos los sistemas operativos que soportan JAVA.

Una limitación notoria de MoviLog - heredada de MRF - lo constituye la sintaxis utilizada para representar los protocolos, dado que sobresimplica aspectos que podrían ser importantes en ciertos contextos, tales como los valores y tipos de los argumentos para acceder a un recurso, los mecanismos empleados para interactuar con los mismos (RPC, CORBA, HTTP), cómo se codifican los argumentos, cuestiones de secuencialidad, etc. Dicha sintaxis está mayormente relacionada con la descripción de cláusulas Prolog, lo que imposibilita su uso para referirse a *recursos* en el sentido más general, tal es el caso de una base de datos, un archivo, o en el caso particular de este trabajo, un Servicio Web.

Con el fin de resolver los problemas anteriores, y al mismo tiempo reutilizar MRF como modelo de movilidad básico, se ha extendido MoviLog con soporte para la invocación de Servicios Web Semánticos. El resultado de esta integración es un lenguaje de programación denominado SWAM (Semantic Web-Aware MoviLog) que, utilizando un modelo que extiende y generaliza MRF denominado MIG (Movilidad Inteligente Generalizada), facilita el desarrollo de agentes móviles en la Web Semántica. SWAM hereda las facilidades de MoviLog para la programación de agentes móviles en la WWW, aumentando su utilidad práctica y proporcionando una implementación de MRF mucho más general y completa.

En el siguiente capítulo se describirá el lenguaje SWAM y el modelo de ejecución MIG.

Capítulo 5

SWAM (Semantic Web-Aware MoviLog)

En el capítulo 3 se exploraron los principales enfoques existentes relacionados con la integración de agentes y la Web Semántica. Sobre el final del capítulo, se enumeraron y expusieron los inconvenientes que presentan dichos enfoques para la materialización de agentes que interactúen con contenido Web de una forma completamente autónoma e inteligente.

Con el fin de dar solución a estos problemas, y al mismo tiempo aprovechar las ventajas ofrecidas por los agentes móviles para la construcción de aplicaciones Web, se ha extendido MoviLog con soporte para la invocación y búsqueda de Servicios Web Semánticos (Mateos et al., 2005b; Mateos et al., 2005a). El resultado de esta integración es un lenguaje de programación denominado SWAM (Semantic Web-Aware MoviLog) que, utilizando un modelo de MRF (Movilidad Reactiva por Fallas) generalizado, facilita el desarrollo de agentes móviles en el contexto de la Web Semántica. Adicionalmente, SWAM hereda los beneficios de JavaLog, lenguaje que ha demostrado ser apto para la programación de agentes *inteligentes* en varias experiencias.

Básicamente, el modelo de ejecución MRF ha sido generalizado para proveer integración con las tecnologías asociadas a la Web Semántica, dando origen a un nuevo modelo llamado MIG (Movilidad Inteligente Generalizada) que, además de la migración de agentes, permite la movilidad de recursos y de control. Esta extensión permite que los agentes SWAM interactúen eficientemente con recursos Web para llevar a cabo sus tareas, lo que influye positivamente en la utilidad del lenguaje como herramienta para la programación de aplicaciones Web basadas en agentes. Para lograr esta adaptación, MIG refina MRF en relación a los siguientes aspectos:

- *Movilidad generalizada*: MIG incorpora nuevas formas extras a la movilidad de agentes para el acceso a recursos Web, tales como invocación remota para el caso de Servicios Web, y replicación o copia de recursos entre sitios para el caso de información y páginas Web. En este sentido, fue necesaria la extensión del mecanismo de descripción de recursos utilizado por MRF con el objeto de describir recursos Web y la forma en la cual un agente accede a una determinada instancia de un recurso.
- *Acceso eficiente a los recursos*: Por cuestiones de eficiencia, es deseable que una plataforma de agentes móviles provea un ambiente de ejecución donde las decisiones relacionadas con la forma de interacción con los recursos sean tomadas de forma inteligente.

Por ejemplo, sería interesante dar prioridad a la copia de un determinado recurso de escaso tamaño por sobre la migración del agente solicitante, para los casos en que el ancho de banda de la red sea pobre. Por este motivo, MIG incluye un soporte predefinido de reglas de acceso que permiten recuperar eficientemente los recursos de acuerdo a condiciones de ejecución del sistema tales como tráfico de red, distancia entre sitios, carga de CPU o memoria RAM libre en un sitio. A su vez, SWAM incluye facilidades sintácticas que permiten al programador adaptar este soporte, definiendo heurísticas de acceso de acuerdo a las particularidades de su aplicación.

- *Semántica de los Servicios Web*: Para lograr una verdadera interacción entre los agentes y los Servicios Web Semánticos, es necesario que cada agente entienda el *significado* preciso de cada servicio. Para resolver este problema, MIG prescribe una arquitectura de publicación y búsqueda de Servicios Web Semánticos, compuesta de repositorios de ontologías que almacenan los conceptos asociados a cada Servicio Web. De esta manera, los agentes son capaces de descubrir y posteriormente utilizar Servicios Web en base a su significado.

En este capítulo se analizará en detalle la forma en la cual MIG y SWAM dan solución a las cuestiones anteriores. Primeramente, la sección 5.1 incluye una síntesis de los principales conceptos discutidos en el capítulo anterior. Por su parte, las secciones 5.2, 5.3 y 5.4 exponen en detalle los enfoques utilizados para resolver cada uno de los problemas mencionados, respectivamente. Finalmente, la sección 5.5 presenta las conclusiones del capítulo.

5.1. MoviLog y MRF

Antes de entrar en detalles propios del modelo de ejecución MIG y el lenguaje SWAM, se expondrá una síntesis de los conceptos más importantes en torno al lenguaje MoviLog y su modelo de movilidad MRF. Particularmente, los agentes móviles de MoviLog se denominan *Brainlets*. La plataforma residente en un sitio que provee soporte de ejecución y recursos a los Brainlets se denomina *MARlet*.

Un Brainlet consiste de una secuencia de cláusulas Prolog y una secuencia (posiblemente vacía) de *protocolos*. Una *falla* entonces se define como la evaluación no satisfactoria de un objetivo Prolog que ha sido declarado como protocolo. El modelo de ejecución de un Brainlet está basado en el mecanismo MRF, que actúa cuando ocurre una *falla*. En otras palabras, los protocolos describen aquellos predicados Prolog cuya evaluación a falso puede ocasionar la migración del Brainlet.

Cuando un Brainlet produce una *falla*, MRF reactivamente migra el Brainlet a un MARlet que contenga cláusulas con el mismo protocolo del objetivo Prolog que falló. Si más de un MARlet cumplen con esta condición, MRF construye un itinerario para visitar dichos MARlets que es incrementalmente actualizado. Una vez que un Brainlet ha migrado, su ejecución se reanuda inmediatamente después de arribar al sitio destino.

En resumen, un Brainlet es un agente móvil compuesto de las siguientes partes:

- *Código*: Es una secuencia de cláusulas Prolog que implementa el comportamiento del agente.

- *Datos*: Consiste de una secuencia de cláusulas representando el conocimiento del agente.
- *Estado de ejecución*: Está compuesto de uno o más *threads* Prolog. Cada *thread* incluye un contador de programa, diversas ligaduras de variables y puntos de *backtracking*. El estado de ejecución es preservado durante la migración.
- *Protocolos*: Describen todos aquellos predicados Prolog cuya evaluación no exitosa pueden disparar la movilidad.

Un protocolo es una declaración con la sintaxis $protocol(functor, arity)$ que indica a MRF que debe tratar la falla de objetivos Prolog con la forma $functor(arg[1], ..., arg[arity])$. Desde el punto de vista del Brainlet, los protocolos permiten al programador controlar los puntos del código del agente que potencialmente disparará la movilidad reactiva. Desde el punto de vista de un MARlet, los protocolos posibilitan describir cláusulas o potencialmente la interfaz de acceso a determinados recursos disponibles en una red.

5.2. MIG: Movilidad Inteligente Generalizada

MIG generaliza el mecanismo de definición de protocolos descrito anteriormente para permitir describir cualquier recurso computacional que un agente pudiera necesitar para finalizar sus tareas. Particularmente, una cláusula Prolog puede ser considerada como un recurso que los agentes acceden en un cierto punto de su ejecución. De hecho, (Zunino et al., 2005b) define el concepto de *falla* como la imposibilidad de la obtención de un recurso por parte de un agente en el sitio actual, sin limitar esta idea sólo a cláusulas Prolog.

Un recurso en MIG se compone de un nombre y un conjunto de propiedades que lo caracterizan. La forma en que el programador declara la necesidad de obtener un recurso - en cierto punto en la ejecución del agente - es mediante un protocolo, que está ubicado en una sección especial del código que implementa al agente. Este protocolo se compone del nombre del recurso en cuestión y las propiedades que lo identifican. Por ejemplo, un recurso de tipo "Servicio Web" está caracterizado por el nombre del servicio, los argumentos de entrada necesarios para su ejecución y la respuesta específica que retorna al agente que lo invoca.

Por ejemplo, el siguiente código SWAM declara un protocolo que describe un Servicio Web (*webService*) de ingreso o *login* a un servidor que requiere validación a través de un usuario y password. El protocolo establece que el Servicio Web cuenta con dos argumentos de entrada, vale decir, el nombre de usuario y la clave. Sin embargo, no establece ninguna restricción sobre el formato de la salida del servicio, a pesar de que ésta puede tratarse de un valor booleano, un string o una estructura compleja representando el resultado del intento de ingreso:

```
protocol(webService,
         [name(doLogin),
          input([userName(U), password(P)]),
          output(X)], wsPolicy).
```

Nótese que algunos atributos del servicio tales como la dirección del servidor o el protocolo de transferencia no han sido especificados. Esta información es encapsulada en cada Servicio Web existente en su documento WSDL asociado, y accedida por MIG en tiempo de ejecución

cuando una instancia específica de servicio es seleccionada por el agente. Tales atributos son conocidos como las propiedades *ocultas* del recurso, esto es, información sólo accesible por la plataforma de ejecución y por ende no accesible a los agentes. Tanto las propiedades ocultas como las *públicas* (name, input y output en el ejemplo) deben ser especificadas por los proveedores al momento de publicar un servicio o recurso.

Como puede observarse, además del nombre del recurso y la lista de propiedades asociadas a éste, el protocolo contiene un tercer argumento. Aquí, *wsPolicy* representa una política provista por el desarrollador compuesta por reglas que determinan el sitio específico (a partir de una lista de candidatos) al cual se accederá para interactuar con el servicio en cuestión, y la forma de acceso preferida (migración al sitio remoto o RPC, por ejemplo). La sección 5.3.2 explica con mayor detalle cómo se especifican e implementan las reglas de acceso a recursos en SWAM y cómo es su funcionamiento.

MRF propone mover un Brainlet hacia el sitio fuente de un recurso dado cada vez que el agente necesita interactuar con éste último (Zunino et al., 2002). Si bien la movilidad es, en muchos casos, una forma efectiva de obtener un recurso, la performance en la ejecución de los agentes puede verse negativamente afectada si la migración no se lleva a cabo de una forma cuidadosa. Por ejemplo, si el tamaño de un agente es mayor que el tamaño del recurso a acceder, claramente conviene copiar el recurso desde el sitio remoto hasta el sitio donde el agente está ejecutando, en vez de migrar el agente al sitio remoto, en particular si la velocidad de transferencia de datos a través de la red es pobre. En el caso que el recurso se trate de un Servicio Web, la forma de acceso más conveniente es invocándolo en forma remota, transfiriendo a través de la red sólo los argumentos de invocación y los resultados del mismo. Por último, si un agente necesita interactuar con registros de una base de datos de gran tamaño, la forma más apropiada de accederlos es migrando al agente hacia el sitio donde se encuentra la base de datos, y así interactuar localmente. En este último caso, acceder a los registros copiándolos desde el sitio remoto resulta inaceptable, dados los costos asociados a la copia en términos de transferencia de datos a través de la red.

Lo expuesto en el párrafo anterior conduce a la necesidad de, por un lado, generalizar MRF para acceder a un recurso determinado de otras formas. Como consecuencia, la performance en la ejecución de agentes puede incrementarse escogiendo el mecanismo que mejor se adapta a las condiciones actuales de ejecución, problema que se abordará en la sección 5.3. Por otro lado, el mecanismo de acceso que puede elegirse para obtener un recurso puede depender de las características del mismo. Para solucionar este problema, cada recurso considerado por MIG tiene un tipo asociado que permite determinar los mecanismos de acceso posibles a aplicar para su obtención.

5.2.1. Descripción de recursos

Todo recurso en MIG está compuesto por un nombre que determina su clase (una base de datos, un Servicio Web, una cláusula, etc.), y un conjunto de propiedades, cuyos valores varían según cada instancia particular del mismo. Por ejemplo, el directorio personal de un usuario dentro de un sitio corriendo Linux estará determinado por la categoría del recurso (*file*) y diversas propiedades de interés asociadas al directorio, tales como su nombre atómico, ubicación dentro del sistema de archivos, tamaño, permisos, entre otras. En general, las propiedades específicas que se incluyen en la publicación de un recurso dependen, en gran

medida, del grado de descripción que el sitio propietario del mismo desea dar a conocer. Sin embargo, cuanto más extensa sea la lista de propiedades publicadas acerca de un recurso, más eficiente y fiel será su selección de acuerdo a las necesidades particulares de cada aplicación. Por ejemplo, un agente podría desear acceder a un archivo de información siempre y cuando éste último confiera determinados permisos de acceso. Si esta información no está incluida en la descripción del recurso, no es posible conocer los permisos asociados al archivo hasta el momento en que es accedido en forma física.

Para declarar la necesidad de obtener un recurso, los agentes SWAM utilizan un mecanismo de descripción de recursos basado en *protocolos*. Básicamente, un protocolo es una estructura Prolog cuya finalidad es identificar el conjunto de instancias de recursos que potencialmente provocarán la falla a nivel MIG durante la ejecución del agente. Dicha estructura presenta el siguiente formato:

$$protocol(resourceKind,\ [property_1, property_2, ..., property_n],\ accessPolicy)$$

donde:

- *resourceKind* es un literal (átomo) que representa la categoría general a la que pertenecen las instancias de los recursos descritas a partir del protocolo. Básicamente, una categoría engloba un conjunto dado de recursos accesibles a los Brainlets, tales como archivos, bases de datos, código Prolog, bibliotecas, Servicios Web, etc.

- El segundo argumento de la estructura corresponde a la lista de propiedades que definen el subconjunto de instancias de la clase de recursos *resourceKind* que el agente pretende identificar. Cada propiedad debe estar instanciada con una estructura compuesta de un functor (el nombre de la propiedad) y un argumento (el valor de la propiedad).

- *accessPolicy* identifica la política empleada por el agente para escoger la instancia específica del recurso y la forma de recuperación apropiada a la hora de efectivizar el acceso a éste. Esta política debe estar declarada en una sección especial del código del Brainlet denominada POLICIES, como se explicará más adelante. Un valor "none" en este argumento indicará que el agente delega las decisiones mencionadas - para la clase de instancias de recursos descritas - a la plataforma de agentes.

Los protocolos constituyen definiciones estáticas acerca de los recursos que los agentes SWAM potencialmente necesitarán a lo largo de su tiempo de vida. El acceso particular a las instancias de un recurso se hace efectivo desde las reglas que determinan el comportamiento del agente. Cada vez que se evalúa una regla Prolog de comportamiento, el mecanismo de MIG determina si su formato se corresponde con algún protocolo declarado por el agente. De ser así, la etapa de búsqueda de instancias candidatas de los recursos descritos por el protocolo es iniciada, para finalmente seleccionar y posteriormente acceder a una de ellas. Se entiende por "acceso" a la acción de recuperar la instancia particular de un recurso para su posterior uso o procesamiento. Por ejemplo, el acceso a un archivo tendrá como resultado la obtención de un *handler* o *proxy* a sus datos físicos. A su vez, el acceso a un Servicio Web se corresponde con la invocación del mismo.

Con el fin de aclarar las ideas expuestas, a continuación se presenta un breve ejemplo consistente de un agente móvil SWAM para la búsqueda distribuida de los archivos llamados *FileName* y que a su vez contienen el texto *Text*. Particularmente, se le ha indicado al agente la tarea de encontrar los archivos denominados "ContactList.txt" que contengan el texto "John Smith". Por simplicidad, no se considera el uso de *wildcards* para realizar la búsqueda. El código que implementa al agente es como sigue:

```
PROTOCOLS
  protocol(file, [name(X)], none)
POLICIES
  % empty section
CLAUSES
  searchForFiles(Text, FileName, Files):-
    assert(filesFound([])).
    findFiles(Text, FileName),
    filesFound(Files).
  findFiles(Text, FileName):-
    file([name(FileName)], FileProxy),
    analizeFile(Text, FileProxy), !, fail.
  findFiles(_, _).
  analizeFile(Text, FileProxy):-
    searchText(Text, FileProxy),
    send(FileProxy, 'getURL', [], FileURL),
    addNewFile(FileURL).
  addNewFile(FileURL):-
    filesFound(TempFiles),
    retract(filesFound(_)),
    assert(filesFound([FileURL|TempFiles])).
  searchText(Text, FileProxy):- ...
  ?-searchForFiles("ContactList.txt","John␣Smith", Result).
```

En este caso, el agente declara un único protocolo, que establece la necesidad de obtener (en algún punto de su ejecución) una o más instancias de un recurso de tipo archivo (*file*). Adicionalmente, el protocolo indica que el nombre del archivo debe ser una propiedad pública, sin establecer una restricción en cuanto al valor de la misma. En resumen, el protocolo indica que el mecanismo de MIG se activará siempre y cuando el agente intente ejecutar una regla que indica el acceso a un archivo llamado *X*. Notar que el programador no especifica ninguna política particular para la recuperación de los archivos. Sin embargo, sería interesante condicionar el método de acceso a cada archivo de acuerdo a su tamaño: si excede una cantidad determinada de bytes, puede decidirse migrar al agente hacia la ubicación del archivo en vez de transferir el contenido de éste último hacia la ubicación actual del agente, efectuando de esta manera un uso más eficiente del ancho de banda.

El Brainlet comienza su ejecución evaluando la cláusula cuyo functor comienza con "?-". Cuando un predicado solicita acceso a un *archivo*, MIG obtiene la lista de instancias tanto locales como remotas que satisfacen el protocolo solicitado. Dado que el agente no declara ninguna política para administrar el acceso al archivo, MIG escoge una instancia de dicha lista, un método de acceso apropiado, y luego recupera el archivo. Cada vez que el predicado en cuestión es reevaluado, MIG extrae el siguiente elemento de la lista, para luego repetir el paso anterior. Una vez que todos los elementos han sido consumidos, el predicado solicitante no puede ser reevaluado, por lo que el proceso de búsqueda codificado por *findFiles* finaliza.

Luego de esto, el Brainlet retorna al sitio que lo lanzó para presentar los resultados.

El acceso efectivo a cada archivo particular se lleva a cabo a través de la regla *file([name(FileName)],FileProxy)*, filtrando al mismo tiempo las instancias de los archivos de acuerdo al nombre deseado (*FileName*). El resultado de dicho acceso es un objeto JAVA, ligado al valor de la variable *FileProxy*. Este objeto oculta la ubicación del archivo físico y provee medios para la lectura del contenido del mismo. La interacción desde Prolog con los métodos de los objetos JAVA es posible en SWAM a partir del predicado predefinido *send(Instance, Method, Args, Result)*. En el ejemplo expuesto, el método *getURL()* del objeto *proxy* es invocado con el fin de recuperar la ubicación real del archivo procesado.

5.2.2. Mecanismos para el acceso a recursos

El modelo de ejecución de agentes móviles en MoviLog se basa en el mecanismo de MRF (Zunino et al., 2002; Zunino et al., 2005b). Cuando cierto recurso requerido por un agente no se encuentra en el sitio donde ejecuta actualmente, MRF transfiere al agente hacia un sitio que contenga alguna instancia de tal recurso, para luego accederlo en forma local. Este proceso es llevado a cabo reiteradamente hasta que el agente finaliza su ejecución.

La movilidad como única forma de acceder a un recurso no local puede ocasionar problemas de performance en la ejecución de agentes. En este sentido, en ciertas ocasiones resulta más conveniente acceder de determinada manera a un recurso dado con el objetivo de lograr la mejor performance y aprovechamiento de recursos de hardware y red posibles. Para dar solución a estos problemas, MIG soporta diversas formas de acceder a un recurso, basándose en tres formas básicas de acceso (figura 5.1):

- *Fetch*: Transfiere el recurso desde un sitio remoto al sitio local, copiándolo a algún repositorio compartido o privado accesible por el agente. Por ejemplo, si un agente necesita un algoritmo de ordenamiento de datos en algún punto de su ejecución, pueden transferirse el código Prolog que implementa dicho algoritmo desde el sitio remoto hacia la ubicación del agente.

- *Invoke*: Invoca el recurso en forma remota, enviando una petición a un agente específico ubicado en el sitio remoto, bloqueándose hasta recibir los resultados, y continuando luego con el flujo de ejecución normal. Aquí, el agente envía una petición - compuesta del nombre del servicio requerido y los argumentos de entrada - a un agente servidor remoto, encargado de invocar efectivamente el servicio y devolver los resultados obtenidos al agente solicitante.

- *Move*: Mueve el agente al sitio donde se encuentra el recurso. Esto implica serializar el estado de ejecución del agente y enviarlo al sitio remoto. Desde allí, el agente continuará con la ejecución normal accediendo localmente al recurso. Por ejemplo, si un agente necesita interactuar repetidas veces con una base de datos remota de gran tamaño, es conveniente migrar el agente hacia el sitio en cuestión para interactuar localmente con los datos, evitando así malgastar una gran cantidad de ancho de banda producto de las numerosas interacciones remotas.

Recurso	Métodos de acceso
Servicio Web	invoke, move
Código Prolog	fetch, invoke, move
Página Web o archivo público	fetch, move
Bases de datos grandes o archivos privados	move
Servicio Web externo	invoke
Código Prolog externo	fetch, invoke
Página Web o archivo externo	fetch

Tabla 5.1: Ejemplos de recursos y métodos de acceso válidos asociados

La tabla 5.1 muestra algunos ejemplos de recursos y los métodos de acceso aplicables para llevar a cabo su recuperación. En la tabla, un recurso *externo* se refiere a uno ubicado en un sitio que no ofrece soporte de ejecución de Brainlets, como lo son la mayoría de los servidores Web públicos existentes.

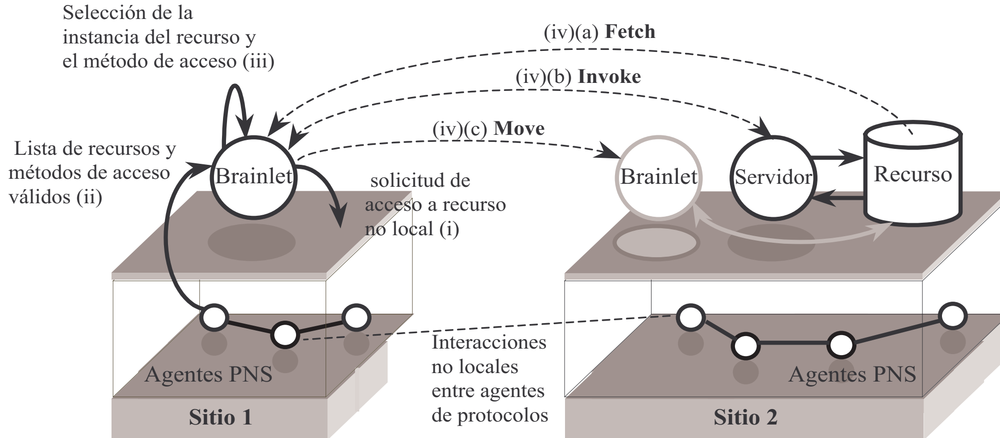


Figura 5.1: Acceso a recursos no locales

La figura 5.1 muestra los pasos llevados a cabo por MIG para el acceso a los recursos. En primer lugar, basado en un protocolo que describe al recurso, se solicita a la plataforma la lista de sitios que lo ofrecen. Luego, se realiza el acceso efectivo al recurso, escogiendo alguno de los sitios obtenidos en la etapa anterior, y realizando la operación de recuperación particular (copia, solicitud remota o migración del agente). Por último, existe una etapa de actualización del estado del agente en función de las nuevas condiciones de ejecución luego de haber obtenido el recurso. Basándose en estas tres etapas, se ha derivado una jerarquía flexible y extensible de mecanismos de acceso a recursos, con una interfaz común que posibilita a MIG el acceso de forma transparente a los recursos, independientemente del mecanismo involucrado. Esta jerarquía se explicará con mejor detalle en el capítulo 7.

El algoritmo de evaluación de un Brainlet SWAM opera como sigue: dada la falla en la obtención de un recurso declarado como protocolo, el agente solicita a MIG los sitios que lo ofrecen, obteniendo así la información asociada a cada instancia particular del recurso (tipo,

disponibilidad, tamaño, etc.). A partir de esta información, el algoritmo construye una lista de pares $L =< \alpha, \beta >$, donde α representa la información asociada a una instancia del recurso, y β son las posibles formas de acceder a la instancia mencionada, eliminando alternativas de recuperación no válidas (por ejemplo, la plataforma no considera el acceso a una base de datos de gran tamaño vía *fetching*). Basado en la lista L, se llevan a cabo los siguientes pasos:

1. *Se elige a qué instancia del recurso se va a acceder*: El algoritmo selecciona de la lista de entrada el sitio particular desde donde se recuperará el recurso en cuestión. Esto corresponde a elegir, en base a alguna política, un único elemento de la lista disponible de instancias, dejando los ítems restantes como futuros puntos de *backtracking* para la reevaluación del algoritmo.

2. *Seleccionar el método de acceso*: El algoritmo selecciona la estrategia de acceso que mejor se adapta a las condiciones actuales de ejecución del sistema (carga de CPU remota, tráfico de red, etc.), particularidades de la aplicación (muchas o pocas interacciones remotas con el mismo recurso) o incluso políticas de preferencia especificadas por el desarrollador del agente (ver sección 5.3).

Es preciso notar que las extensiones introducidas al algoritmo de evaluación original de MoviLog trae considerables ventajas. En primer lugar, permiten mejorar la performance en la ejecución de agentes, ya que se accede a un recurso utilizando la forma más adecuada de acuerdo a las condiciones de ejecución del sistema de agentes. En segundo lugar, otorgan al desarrollador de agentes la posibilidad de emplear políticas de acceso específicas, lo que introduce una gran flexibilidad en la programación de la interacción de los agentes con los recursos. Por último y relacionado al punto anterior, la plataforma de agentes puede recurrir al uso de políticas de acceso eficientes por defecto cuando el usuario no especifica decisiones propias.

5.2.3. Clasificación de recursos

En la sección anterior se describió la forma general en la cual opera el mecanismo de acceso a recursos no locales. Cuando un agente solicita el acceso a un recurso no presente en el sitio local, la plataforma subyacente construye una lista de instancias de recursos que coincidan con la solicitud del agente. Dicha lista está formada por pares $L =< \alpha, \beta >$, donde el primer elemento (α) es la información relativa a una instancia del recurso, y el segundo elemento (β) son las formas de acceso para recuperar el recurso denotado por α. Es en este punto en que la plataforma filtra, de acuerdo al tipo de recurso, las formas de acceso válidas. Con este fin, MIG asocia a cada recurso un tipo único que está definido a partir de una clasificación estándar compartida a través de los sitios. Esta clasificación se ilustra en la figura 5.2.

Básicamente, MIG clasifica los recursos en transferibles o no transferibles. Un recurso transferible es aquel que puede ser copiado de un sitio a otro (un archivo o una variable de ambiente, por ejemplo), mientras que un recurso no transferible permanece fijo dentro un sitio y no puede ser transferido (una impresora o un scanner, por ejemplo). Los recursos transferibles se subdividen en libres o fijos: un recurso libre puede migrarse entre dos sitios, mientras que un recurso fijo representa datos o dispositivos cuya transferencia resulta inviable (una base

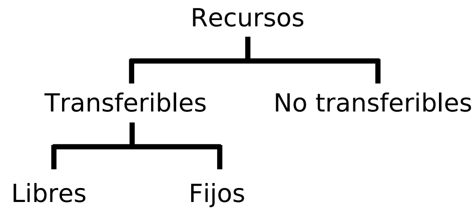
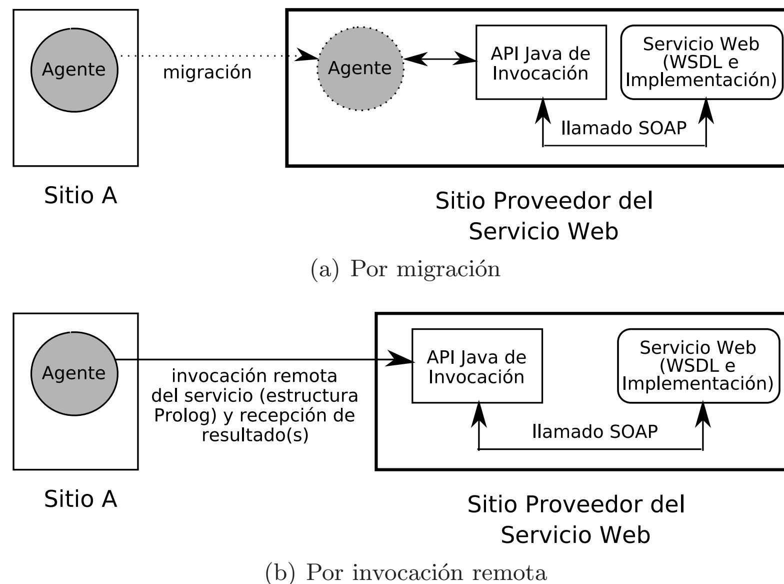


Figura 5.2: Clasificación de recursos accesibles por entidades móviles (Vigna, 1998)

de datos de gran volumen) o indeseada (archivos con información privada). Esta última subdivisión la establece cada aplicación en particular, y no depende del tipo de recurso como en el caso anterior.

En SWAM, un Servicio Web es un recurso que representa una funcionalidad específica ofrecida por la plataforma de agentes, accesible a través de la Web. Dicha funcionalidad se materializa como código JAVA encargado de asociar una cláusula Prolog en ejecución que representa una invocación a un Servicio Web con la invocación SOAP de bajo nivel correspondiente. En otras palabras, dicho código JAVA es el encargado de mapear el nombre del servicio y los argumentos, representados como tipos de datos Prolog, a los tipos de datos SOAP asociados, para luego invocar el servicio solicitado. Cabe destacar que el código que implementa el servicio y lleva a cabo las conversiones de argumentos es local al sitio que provee el servicio. Consecuentemente, un Servicio Web en SWAM es un recurso no transferible, dado que el código de soporte de invocación e implementación del servicio no es transferido entre los sitios. Este hecho limita la forma en que un agente puede acceder a un Servicio Web a los mecanismos de migración o invocación remota únicamente, como muestra la figura 5.3.



(a) Por migración

(b) Por invocación remota

Figura 5.3: Formas de interacción de un agente SWAM y un Servicio Web

Las cláusulas Prolog representan recursos transferibles libres de un sitio, ya que son en sí código Prolog que en general puede ser copiado sin mayores inconvenientes. Sin embargo, un sitio puede considerar oportuno la denegación de la transferencia de un conjunto de cláusulas; en

este caso, el conjunto de cláusulas es visto como un recurso transferible fijo. La imposibilidad de la transferencia podría ser producto de una de las siguientes situaciones:

- El tamaño estimado a partir del conjunto de cláusulas es muy grande, por lo que su transferencia resulta ineficiente. Por ejemplo, la transferencia de un grupo de cláusulas que representan un extenso algoritmo Prolog puede no ser permitida por limitaciones en el ancho de banda asignado para el tráfico por la red.
- Alguna de las cláusulas del conjunto poseen variables instanciadas con diversas estructuras u objetos no serializables, es decir, no representables en un formato transferible a través de la red. En este caso, la transferencia del recurso resulta imposible, a diferencia del caso anterior, en donde la misma no es deseable.
- Existen cuestiones de seguridad establecidas en el sitio remoto que limitan la transferencia de ciertos recursos. A pesar que las cláusulas Prolog estén libre de los problemas anteriores, existen limitaciones de seguridad que impiden su copia a otros sitios de la red.

SWAM delega la responsabilidad de clasificación de los recursos presentes en un sitio a ciertos agentes estacionarios extendidos con diversas políticas de categorización de recursos, cuya función es administrar la clasificación de recursos en forma dinámica. Mediante este soporte, es posible especializar tanto como se quiera la política de clasificación de un recurso particular, pudiendo implementar políticas de clasificación fija, donde el tipo de recurso es determinado estáticamente, o variable, estableciendo el tipo de recurso algorítmicamente y en forma dinámica. En este sentido, el protocolo que describe un Servicio Web tendrá asociado una política de clasificación fija (no transferible), mientras el correspondiente asociado a un conjunto de cláusulas Prolog referenciará a una política de clasificación variable que implemente, por ejemplo, un algoritmo de estimación de tamaño de código Prolog.

5.3. Heurísticas para el acceso a recursos

La introducción de políticas de acceso a recursos por parte de MIG mejora la eficiencia de MRF (Zunino et al., 2005a). En primer lugar, la plataforma de ejecución de Brainlets puede establecer políticas por defecto para el acceso eficiente a un recurso con el fin de recuperarlo de la forma más apropiada de acuerdo a las condiciones actuales del sistema. Además, un soporte flexible de especificación de políticas permite al programador establecer condiciones y mecanismos de acceso propios, mayormente en base a los requerimientos y características de su aplicación. La forma en que la plataforma define las políticas de acceso es similar a la manera en que lo hace el programador, es decir, tomando la instancia de un recurso más adecuada a partir de un conjunto de posibilidades. La diferencia clave es que el desarrollador debe tener a su disposición determinadas construcciones y directivas para indicar a la plataforma sus propias decisiones de acceso. En otras palabras, el desarrollador debe poder especificar, dado un número de fuentes y formas de acceso posibles para un mismo recurso, un único sitio y mecanismo de acceso para recuperarlo.

El programador SWAM expresa sus políticas de acceso mediante reglas Prolog que están basadas en métricas del sistema. El programador tiene a su disposición diversos predicados

Prolog que se encargan de retornar el valor actual de una característica particular del sistema (por ejemplo la carga de CPU, el tráfico de red, la proximidad entre sitios, etc.). Por ejemplo, la evaluación de *freeMemory(Site, R)* instanciará la variable R con la cantidad de la memoria RAM libre en el sitio *Site*. A partir de este predicado, el programador puede crear reglas que le permitan implementar políticas de acceso más complejas. Este sería el caso, por ejemplo, de una regla que acceda a la instancia del recurso que se encuentra en el sitio con menor cantidad de memoria RAM ocupada actualmente.

La forma escogida para la especificación de políticas de acceso presenta dos ventajas importantes. Por un lado, la naturaleza declarativa y las características de inferencia que ofrece Prolog otorgan una gran flexibilidad a la hora de especificar reglas. Adicionalmente, dado que cada agente SWAM cuenta con la posibilidad de modificar su base privada de reglas, es posible variar dinámica e inteligentemente las políticas de acceso establecidas para cada recurso particular.

5.3.1. Profiling

La introducción de métricas sobre diferentes aspectos de la performance del sistema sugiere la necesidad de que cada sitio que conforma una red de agentes MIG incluya algún componente que permita administrar este tipo de información. En este sentido, MIG incluye un componente especial responsable de obtener y mantener valores actualizados para ciertas métricas del sistema, y a su vez proveer una interfaz para servicios de *profiling* acorde a las necesidades de los agentes. La lista de métricas consideradas por MIG se resumen en la tabla 5.2.

Tipo	Métrica
Procesador y memoria	Carga de CPU local
	Carga de CPU remota
	RAM libre local
	RAM libre remota
Red	Tasa de transferencia
	Proximidad a un sitio
	Fiabilidad de la comunicación
Misceláneas	# de agentes en ejecución
	Tamaño de un agente

Tabla 5.2: Métricas del sistema soportadas por SWAM

El significado de cada métrica mostrada en la tabla es como sigue:

- *Carga de CPU*: Se interpreta como el porcentaje de utilización de CPU en el sitio local. Es una métrica de utilidad en ciertas ocasiones. Por ejemplo, si un recurso dado puede accederse mediante copia o migración, y la carga de CPU local es mucho mayor que la del sitio remoto, puede utilizarse la migración como forma de acceso, a modo de técnica sencilla para balancear la carga de los sitios.
- *Carga de CPU remota*: Representa la carga de CPU de un sitio remoto determinado. En principio, resulta difícil contar con mediciones actualizadas de utilización de CPU de

otros sitios, dada la naturaleza altamente dinámica de dicha métrica y las potenciales demoras en el intercambio de la información entre sitios. Para resolver este problema, cada sitio envía periódicamente a sus pares información actualizada del promedio de utilización de CPU local. Luego, los sitios estiman el uso de CPU del sitio remoto a partir de la historia de utilización de CPU.

- *RAM libre local*: Representa el porcentaje de memoria disponible en un el sitio local. Por otra parte, la memoria RAM libre de un sitio remoto puede estimarse de forma similar a la carga de CPU remota.

- *Tasa de transferencia*: Esta métrica está asociada a la velocidad actual en los vínculos de comunicación salientes (en Kbps), y puede ser medida mediante algún comando típico provisto por el sistema operativo (*ping* de Linux, por ejemplo). Contar con medidas de velocidad de transferencia a nivel de vínculos de red es vital para decidir entre migrar agentes y copiar recursos desde sitios remotos, o bien invocar remotamente. En la mayoría de los casos, ésta última alternativa generará una menor cantidad de datos a transferir que la migración o la copia, mejorando en consecuencia la performance en la ejecución de agentes.

- *Fiabilidad de comunicación*: Es el porcentaje de pérdida en la transferencia de información a través de la red. Notar que esta medida reviste una singular utilidad para categorizar la calidad de la conectividad con nodos de una red tales como dispositivos móviles (PDA, celulares, etc.) o equipos conectados a través de redes inalámbricas.

- *Proximidad al sitio remoto*: Es una métrica mayormente relacionada con la topología de la red, y calcula la cantidad de sitios intermedios que es necesario atravesar para enviar un paquete de datos desde el sitio local a un determinado nodo de la red. En general, puede ser útil para dar una idea de cuan grande puede ser el retardo del intercambio de información entre dos sitios.

- *Cantidad de agentes en ejecución*: Tiene una estrecha relación con la carga de CPU, principalmente si cada agente del sitio está ejecutando tareas en forma activa, es decir, no bloqueado a la espera de algún suceso. Este valor puede obtenerse a partir del motor de ejecución de agentes local, y provee una idea cualitativa de la utilización de CPU y memoria. Puede estimarse también la cantidad de agentes en ejecución de un sitio remoto de forma similar a la carga de CPU remota mencionada más arriba.

- *Tamaño de un agente*: Representa la estimación medida en bytes del espacio en memoria asignado a un agente en ejecución, es decir, la cantidad de memoria RAM ocupada por su código y estado de ejecución. Puede ser una métrica útil para tomar decisiones en cuanto a la migración de agentes, comparando su tamaño con las condiciones actuales velocidad de transferencia de la red. Similarmente, puede considerarse aplicar esta métrica a los recursos, permitiendo así establecer reglas de recuperación en función del tamaño del recurso a utilizar. De esta manera, es posible crear reglas de recuperación de recursos estableciendo la conveniencia de migrar el agente hacia el recurso, o viceversa.

A nivel SWAM, cada una de las métricas anteriores son accesibles mediante un predicado Prolog que provee la interfaz entre el agente y el administrador de servicios de *profiling* de un sitio. Finalmente, pueden ser agregadas con facilidad nuevas métricas, implementando

el servicio de cálculo correspondiente e incorporándolo en el administrador de servicios de *profiling*, y haciéndolo visible a los agentes mediante un nuevo predicado Prolog.

5.3.2. Programación de reglas

El programador puede utilizar los predicados que calculan cada una de las métricas listadas en la sección anterior para construir reglas de acceso a recursos más complejas. Dichas reglas son especificadas en una sección especial del código de un Brainlet, cada una con un nombre que la identifica y la lógica decisoria correspondiente. De esta manera, un programador puede utilizar una regla de acceso a un recurso determinado agregando el nombre de la regla en cuestión al protocolo asociado a dicho recurso. Así, cuando la obtención del recurso descrito por el protocolo falle, MIG buscará instancias posibles del mismo, para luego seleccionar la instancia particular y el método de acceso según la regla de decisión indicada en el protocolo. Para materializar este concepto, SWAM permite al programador la declaración de dos reglas diferentes, una para realizar la selección de la instancia del recurso, y la otra para escoger el método de acceso. El formato de ambas reglas es:

```
sourceFrom(Name, [ID_1, Site_1], [ID_2, Site_2], Result):- ...
accessWith(Name, [IDResource, Site], MethodA, MethodB, Result):- ...
```

respectivamente. La primera regla, cuyo nombre se corresponde con el valor al que está ligada la variable *Name*, incluye en su cuerpo la política empleada para decidir qué recurso acceder (*Result*) a partir de dos instancias cualesquiera. Similarmente, la segunda regla contiene la lógica necesaria para decidir qué método de acceso aplicar dados dos métodos de acceso válidos cualesquiera (*MethodA* y *MethodB*) para acceder a una instancia del recurso. Se entiende por *válido* a aquellos métodos que MIG considera apropiados para acceder a un recurso específico. Por ejemplo, MIG no considera la operación de *fetch* para acceder a una base de datos de gran tamaño.

Volviendo al ejemplo del Brainlet para búsqueda de archivos expuesto en secciones previas, supóngase que se desea indicar al agente la siguiente política para acceder a los archivos: "en cualquier caso, acceder a la instancia del archivo que se encuentra en el sitio remoto más próximo al sitio local. Por otra parte, si el tamaño del Brainlet es menor o igual que el tamaño de dicho archivo, recuperarlo siempre mediante migración". Las reglas que el programador debe incluir en el código del agente para reflejar esta política son:

```
sourceFrom('file-policy', [_, Site_1], [_, Site_2], S):-
  proximity(Site_1, P_1),
  proximity(Site_2, P_2),
  min(P_1, P_2, Site_1, Site_2, S).
accessWith('file-policy', [ID_Res, Site], move, _, move):-
  agentSize(BrSize),
  resourceSize(ID_Res, FileSize),
  BrSize <= FileSize.
```

Como puede observarse, la regla *sourceFrom* calcula la proximidad entre la ubicación actual del agente y cada sitio remoto, ligando la variable S con la dirección del sitio más cercano. Por otra parte, la regla *accessWith* selecciona el método *move* para efectivizar el acceso a un archivo siempre y cuando el tamaño del agente sea menor que el tamaño del archivo en cuestión. Cabe

destacar que *proximity*, *agentSize* y *resourceSize* son tres predicados predefinidos provistos por el soporte básico de programación de políticas de SWAM.

5.4. Manejo de Servicios Web con información semántica

El uso efectivo de la gran cantidad de funcionalidad ofrecida en la Internet requiere que tanto las aplicaciones como los agentes estén integrados con infraestructura y Servicios Web existentes. Sin embargo, materializar adecuadamente esta integración será imposible, a menos que dichos servicios sean representados en un lenguaje semántico interpretable por los agentes inteligentes (Kagal et al., 2001).

Una alternativa para resolver el problema de la interoperación entre agentes inteligentes y Servicios Web es la utilización de descripciones estructuradas de los conceptos presentes en cada servicio, describiendo *qué* funcionalidad proveen, y no *cómo* la llevan a cabo. De esta manera, los agentes interactúan con los Servicios Web a nivel de aplicación, interpretando descripciones abstractas de los servicios que son suficientes para conocer su propósito y su funcionalidad.

El uso de lenguajes de descripción de contenido Web ofrece ventajas para la localización de Servicios Web basado en la funcionalidad que proveen, característica que permite la interacción automática entre agentes y servicios que implementen una funcionalidad requerida. La localización de Servicios Web es un problema inherentemente semántico, ya que debe abstraer las diferencias superficiales entre las representaciones de los servicios provistos y los servicios solicitados (por ejemplo, el formato de los parámetros) a fin de reconocer similitudes semánticas entre ambos. Así, por ejemplo, (Paolucci et al., 2002) ha propuesto la utilización de DAML por encima de UDDI, un mecanismo estándar para la publicación y registro de servicios, posibilitando la creación de una poderosa infraestructura de búsqueda de Servicios Web basado en las tareas que éstos llevan a cabo. Cabe destacar que esta integración se basa en DAML-S (Burstein et al., 2002), una ontología de servicios Web estándar construida en base al lenguaje DAML.

Con el fin de alcanzar una verdadera automatización en la interacción con la Web, los agentes deben entender el significado preciso de cada servicio Web. Para hacer esto posible, MIG prescribe una arquitectura que incluye componentes para el manejo de ontologías haciendo uso del lenguaje DAML+OIL, un lenguaje de descripción de recursos que extiende a RDF. El aspecto más interesante de RDF y DAML+OIL para este trabajo es que ambos tienen una semántica bien definida que es fácilmente traducible a Prolog (Fikes y McGuinness, 2001). De esta forma, es posible realizar inferencias automáticamente a partir de una ontología DAML+OIL traducida, utilizando algoritmos tradicionales de prueba de teoremas.

Lo anteriormente mencionado tiene consecuentes ventajas en la implementación del matching de servicios a nivel SWAM. Recordar que los agentes SWAM identifican recursos en base a protocolos, que son en sí reglas Prolog, lo que permite llevar a cabo inferencias con respecto a una ontología DAML+OIL expresada como fórmulas Prolog. Así, un protocolo de descripción lo suficientemente expresivo permitirá identificar Servicios Web en base a los conceptos involucrados en éste último, tales como la entrada requerida, la funcionalidad provista o la salida retornada.

Existen en la actualidad varias propuestas para atacar el problema de encontrar, dada una solicitud de servicio determinada, un Servicio Web semánticamente compatible. Los enfoques propuestos para dar solución a este problema, conocido como *semantic matching*, pueden resumirse en:

- *Matching semántico estructural*: Dada una solicitud de servicio expresada como una lista de conceptos de entrada (argumentos del servicio) y una lista de conceptos de salida (la salida esperada del servicio), se asumen como compatibles todos los servicios cuyos conceptos de entrada y salida coincidan semánticamente con la entrada y salida de la solicitud, respectivamente. El problema de esta alternativa es que no es posible discriminar los servicios cuyos conceptos de entrada y salida coinciden, pero la funcionalidad es diferente. Algunos trabajos que utilizan este enfoque son (Sycara et al., 1999; Pokraev et al., 2003; Li y Horrocks, 2004).

- *Matching semántico funcional*: En este enfoque (Payne et al., 2001; Sivashanmugam et al., 2003), el cliente incluye en su solicitud de servicio un concepto que representa la funcionalidad del mismo (por ejemplo, *GetCreditCardBalance*). Luego, todos los Servicios Web cuya acción asociada coincida semánticamente con el concepto de entrada son considerados compatibles con la solicitud del cliente. El principal problema del enfoque radica en la dificultad de expresar adecuadamente la funcionalidad que provee un servicio valiéndose sólo de un concepto que la describe.

- *PRE y POST condiciones*: Aquí, cada operación o servicio es descripta mediante ciertas precondiciones y efectos. En general, las precondiciones pueden ser ciertas condiciones lógicas que deben ser verdaderas para ejecutar la operación. Por otra parte, los efectos o postcondiciones son cambios producidos en el "mundo" (el ambiente donde se ejecuta el servicio) luego de la ejecución de la operación. El punto débil del enfoque radica en que los proveedores de servicios deben acordar previamente un lenguaje común para representar las precondiciones y efectos, y definir consensuadamente el alcance de los mismos. Algunos trabajos relacionados con este enfoque son (Akkiraju et al., 2003; Sivashanmugam et al., 2003; Sirin et al., 2003).

Adicionalmente, existen enfoques híbridos que proponen el *matching* semántico basado en una combinación de las alternativas señaladas anteriormente. Por ejemplo, la estrategia de *matching* estructural-funcional-contextual determina la compatibilidad de una solicitud y un Servicio Web basada en las similitudes semánticas entre los argumentos de entrada y salida, la funcionalidad y el contexto de provisión de dicho servicio (proveedor, área geográfica, etc.).

La alternativa escogida por MIG para llevar a cabo el *matching* semántico de Servicios Web es un enfoque híbrido compuesto por *matching* estructural y funcional. En este sentido, se ha desarrollado un componente de *matching* que, a partir de un repositorio de ontologías expresadas como reglas Prolog y una base de datos de descripciones semánticas de servicios, calcula el grado de similitud semántica entre dos Servicios Web. El repositorio de ontologías es actualizado a partir de un componente encargado de la traducción de ontologías expresadas en RDF, DAML+OIL u OWL a su representación Prolog asociada. La arquitectura general de *matching* se ilustra en la figura 5.4.

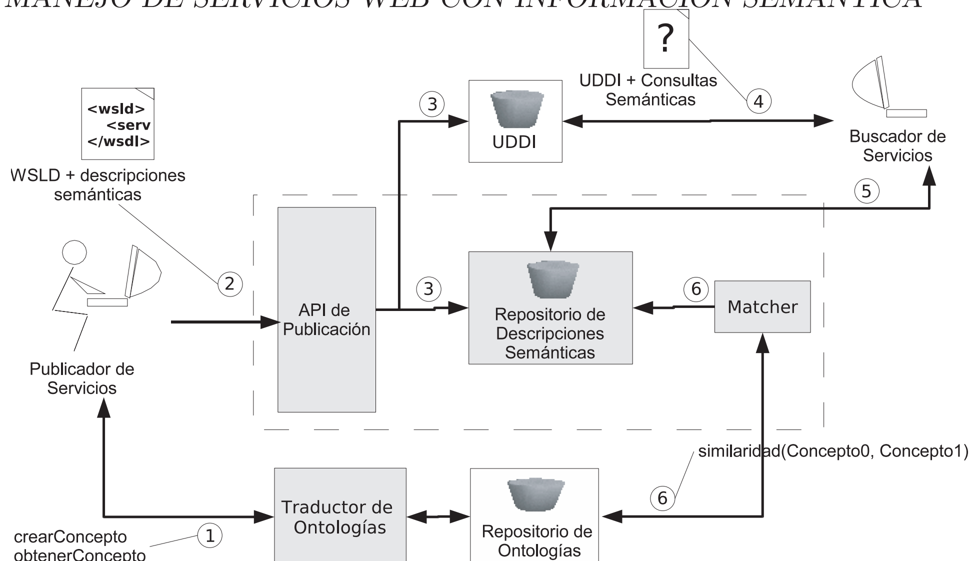


Figura 5.4: Arquitectura del Matcher de Servicios Web de SWAM

La línea discontinua delimita los componentes encargados de la representación y matching de la semántica asociada a cada Servicio Web. En primer lugar, la API de Publicación permite definir y agregar nuevos servicios a partir de un documento WSDL provisto de descripciones semánticas para cada mensaje declarado. Dicha información es almacenada en el Repositorio de Descripciones Semánticas, que provee un repositorio para guardar los conceptos que definen la acción llevada a cabo por un Servicio Web, y los conceptos asociados a las entrada(s) y salida(s) del mismo. El Matcher utiliza esta base de datos para determinar la posible semejanza entre una petición de servicio por parte de un agente y alguna de las descripciones de servicio almacenadas.

Los principales componentes que se encuentran fuera del subsistema mencionado son el Repositorio de Ontologías y los Registros UDDI. Por un lado, el Repositorio de Ontologías se encarga de almacenar las ontologías modeladas por los usuarios relacionadas con conceptos inmersos en las definiciones de los Servicios Web. Cada ontología agrupa el conjunto de conceptos existente en un dominio determinado, y están representadas como módulos lógicos compuestos de reglas Prolog que describen la estructura de los conceptos y las relaciones entre éstos. En general, las ontologías Web se modelan utilizando lenguajes verborrágicos basados en XML. Por este motivo, la arquitectura incluye un componente denominado Traductor de Ontologías, encargado del mapeo de ontologías escritas en RDF, DAML y OWL a su correspondiente representación Prolog, y viceversa. Esta traducción permite representar la información semántica mediante un lenguaje común que posibilita la realización de un único motor de inferencias semánticas, evitando así la construcción de un motor específico para cada lenguaje, tarea que resulta más costosa que implementar un traductor a y desde Prolog para cada lenguaje.

Los Registros UDDI son otro componente de particular importancia en la arquitectura, por cuanto proveen una forma de publicación de servicios en registros accesibles a través de la Web. Como se ha visto con anterioridad, UDDI es un mecanismo estándar para la búsqueda y publicación de Servicios Web que ha tenido una gran aceptación entre los desarrolladores de aplicaciones en la Web Semántica. Por este motivo, se ha decidido proveer una conexión con

UDDI, en el sentido de permitir publicar las definiciones WSDL de servicios tanto en un sitio que acepta agentes MIG como en registros UDDI externos, lo que permite la reutilización de la implementación de los servicios a aplicaciones externas.

Básicamente, las posibles interacciones entre las entidades y componentes mencionados pueden resumirse como sigue:

1. A través de una aplicación *front-end*, los usuarios publican nuevos Servicios Web (un nuevo recurso, en terminología MIG). Esto requiere modelar la estructura de los conceptos que representan la acción o funcionalidad de cada servicio, la entrada necesaria para su invocación y la salida que retorna. Los cambios concernientes a nuevos conceptos creados o conceptos existentes que han sido refinados son reflejados en el Repositorio de Ontologías, previa traducción a Prolog a partir del lenguaje de representación origen.

2. El documento WSDL que define un Servicio Web, que además contiene las descripciones semánticas para cada mensaje, es enviado por la aplicación *front-end* a la API de Publicación, que a su vez interactúa con el Repositorio de Descripciones Semánticas y registros UDDI diseminados a lo largo de la Web.

3. La API de publicación interactúa con el Repositorio de Descripciones Semánticas para almacenar los documentos WSDL que contienen las definiciones de los Servicios Web, y la información semántica asociada. Similarmente, los WSDL son incorporados a registros UDDI para su posterior acceso por parte de plataformas y aplicaciones externas. Los registros particulares escogidos para este fin pueden depender, por ejemplo, de una lista configurada en forma estática.

4. El Buscador de Servicios es una entidad (un agente o aplicación) que periódicamente consulta registros UDDI externos (previamente conocidos) con el fin de hallar documentos WSDL que representen servicios no incluidos en el Repositorio de Descripciones Semánticas. Naturalmente, los registros UDDI consultados deben proveer cierta información de contexto con respecto a cada servicio; de lo contrario, la base de descripciones semánticas no contará con la información necesaria para establecer similitudes con los servicios solicitados por los agentes.

5. Basado en las nuevas definiciones WSDL encontradas a través de la Web, el Buscador de Servicios actualiza el Repositorio de Descripciones Semánticas. Esta información será utilizada por el Matcher para resolver futuras solicitudes de servicios.

6. Al recibir una petición de servicio, el Matcher consulta el Repositorio de Descripciones Semánticas para encontrar los Servicios Web que son semánticamente compatibles con la misma. Para esto, se comparan los conceptos asociados a la solicitud de servicio y a las descripciones almacenadas, usando un algoritmo de similaridad que debe ser provisto por la plataforma que materializa MIG. Dicho algoritmo utiliza como base las definiciones de los conceptos de las ontologías presentes en el Repositorio de Ontologías. Cabe destacar que el Matcher sólo retorna todos los servicios que superan un cierto umbral de similitud preestablecido.

En la siguiente subsección se detallará el funcionamiento del algoritmo que computa el grado de similitud semántica existente entre dos Servicios Web. Dicho algoritmo implementa la responsabilidad principal del componente Matcher previamente visto.

5.4.1. Algoritmo de similitud semántica de Servicios Web

Cada vez que un agente móvil solicita a la plataforma de agentes un Servicio Web para resolver alguna tarea, el Matcher del sitio local determina qué Servicios Web - del conjunto registrado localmente - satisfacen semánticamente la petición del agente. Para esto, el Matcher posee un algoritmo que determina la similitud semántica existente entre dos servicios dados. Este algoritmo toma como base los conceptos relativos a la acción que los servicios llevan a cabo, y los conceptos asociados a los argumentos de entrada y salida de cada servicio, y retorna un valor entero que representa el grado de similitud entre ambos. El siguiente pseudocódigo ilustra la forma general de operación del algoritmo mencionado.

$reqAction \Leftarrow getActionConcept(service1)$
$offAction \Leftarrow getActionConcept(service2)$
$actionSim \Leftarrow similarity(reqAction, offAction)$
if $actionSim \neq 0$ **then**
  $inputs1 \Leftarrow getInputConcepts(service1)$
  $inputs2 \Leftarrow getInputConcepts(service2)$
  $outputs1 \Leftarrow getOutputConcepts(service1)$
  $outputs2 \Leftarrow getOutputConcepts(service2)$
  if size(inputs1) = size(inputs2) and size(outputs1) = size(outputs2) **then**
    $inputSim \Leftarrow listSimilarity(inputs1, inputs2)$
    $outputSim \Leftarrow listSimilarity(outputs1, outputs2)$
    return overallSimilarity(actionSim, inputSim, outputSim)
  else
    return 0
  end if
else
  return 0
end if

Algoritmo 1: Similitud semántica entre Servicios Web

Primeramente, el algoritmo examina la similitud existente entre los conceptos que representan la funcionalidad provista por el servicio. En caso de no existir semejanza (funcionalidad disjunta o diferente) se retorna un valor de similitud cero. Caso contrario, la lista de conceptos que representan las entradas y salidas de los servicios son comparadas entre sí (ver algoritmo 2). El valor de similitud final es calculado por medio de la función *overallSimilarity* en base a los valores de semejanza semántica para la acción, la entrada requerida y la salida retornada de los servicios.

La comparación semántica de más bajo nivel realizada por los algoritmos mencionados es el chequeo de similitud entre dos conceptos cualesquiera, representado por los llamados a *similarity(Concept0,Concept1)*. La función *similarity* calcula el grado de semejanza semántica (valor entero) existente entre dos conceptos, tomando ambos como entrada, y computando las similaridades estructurales basándose en las estructuras conceptuales que se encuentran almacenadas en el Repositorio de Ontologías. Las estructuras conceptuales son representaciones que describen el significado de un concepto específico, es decir, la estructura interna que posee y las relaciones que presenta con otros conceptos (por ejemplo, un *paper* se compone de *secciones*, y puede referenciar otros papers).

$accumulated \Leftarrow 0$
$listSize \Leftarrow size(conceptList1)$
for $i = 1$ to $listSize$ **do**
  $concept1 \Leftarrow item(conceptList1, i)$
  $concept2 \Leftarrow item(conceptList2, i)$
  $accumulated \Leftarrow accumulated + similarity(concept1, concept2)$
end for
return $similarity/listSize$

Algoritmo 2: $listSimilarity(list1, list2)$: Similaridad entre dos listas de conceptos

5.4.2. Materialización en SWAM

Para permitir una interacción real entre agentes y Servicios Web Semánticos, SWAM materializa la arquitectura anterior proveyendo una implementación para cada uno de los componentes descritos. Concretamente, el Repositorio de Ontologías es una base de datos Prolog compuesta de cláusulas que representan la estructura de los conceptos. Cada ontología particular agrupa en un modulo lógico los conceptos propios del dominio que ésta modela, por ejemplo *autos*, *deportes*, *medicina*, etc. Las asociaciones existentes entre los WSDL que describen servicios y los conceptos del repositorio mencionado son almacenadas en una base de datos a través de tablas simples.

El componente Matcher se implementó como un razonador que opera en base a ontologías descritas en el lenguaje OWL Lite (Antoniou y van Harmelen, 2003). OWL Lite es un sublenguaje de OWL que soporta jerarquías de clasificación (subclases) y restricciones simples, pero asegura la completitud y decidibilidad de las inferencias. Por otra parte, OWL es una de las tecnologías para representar ontologías que se espera se convierta en estándar en un futuro próximo.

Una característica particularmente interesante de OWL Lite es que su modelo formal es fácilmente traducible a lógica de primer orden (de Bruijn et al., 2004). En otras palabras, las ontologías escritas en OWL Lite pueden traducirse a una representación Prolog equivalente, para luego realizar inferencias sobre ésta última.

Sentencia OWL Lite	Prolog	Descripción
Class	class(X)	X es una clase.
rdfs:subClassOf	subClassOf(X,Y)	X es una subclase de Y.
rdf:Property	property(X)	X es una propiedad.
rdfs:subPropertyOf	subPropertyOf(X,Y)	X es una subpropiedad Y.
Individual	individualOf(X,Y).	X es una instancia Y.
inverseOf	inverseOf(X,Y)	X es inversa a la propiedad Y.
equivalentProperty	equivalentProperty(X,Y)	X es equivalente a la propiedad Y.
equivalentClass	equivalentClass(X,Y)	X es equivalente a la clase Y.
Properties	triple(X,Y,Z).	X se relaciona con Z mediante Y.

Tabla 5.4: Correspondencia entre OWL y Prolog

La tabla 5.4 muestra la contrapartida Prolog correspondiente a algunas de las sentencias OWL Lite soportadas por el Matcher. Los constructores de clases y propiedades se representan mediante hechos simples, mientras que las relaciones se almacenan como triplas RDF. Una tripla RDF es una estructura de la forma *triple(subject, property, object)* que indica que *subject* esta relacionado con *property* a través del valor *object*. Asimismo, las restricciones de cardinalidad, rango y dominio de OWL Lite son también representadas como "triplas" Prolog. Por ejemplo, *triple(author, range, person)* establece que la propiedad *author* debe ser una instancia de la clase *person*.

Basado en la representación discutida anteriormente, el Matcher SWAM implementa un algoritmo Prolog que retorna un valor de similitud dados dos conceptos cualesquiera. El cálculo del valor de similitud final entre dos Servicios Web (*overallSimilarity*) se obtiene a partir de la fórmula:

$$OS = \frac{w_f * S_f + w_i * S_i + w_o * S_o}{S_f + S_i + S_o}$$

donde:

S_f, S_i *y* S_o son los valores de similitud entre la funcionalidad, la entrada y la salida de los servicios, respectivamente

w_f, w_i *y* w_o son los pesos que otorga SWAM a los valores de similitud S_f, S_i *y* S_o, los cuales cumplen con la restricción $w_f + w_i + w_o = 1$

Los pesos son parametrizables al momento de realizar el *matching* semántico entre dos servicios. Los agentes pueden utilizar diversas configuraciones de valores para ajustar la comparación de servicios de acuerdo a sus necesidades. Por defecto, SWAM considera la misma prioridad para cada peso, esto es, $w_f = w_i = w_o = \frac{1}{3}$.

5.5. Conclusiones

La movilidad fuerte y los mecanismos de inferencia son, sin dudas, dos características de suma importancia que una plataforma de agentes móviles debiera proveer. MoviLog representa un avance en esta dirección. La mayor contribución de MoviLog es el concepto de movilidad reactiva por fallas, que junto a Prolog facilita enormemente el desarrollo de agentes móviles inteligentes.

Debido a la creciente evolución de la Web hacia un ambiente donde las aplicaciones y los agentes inteligentes interactúan de forma automatizada, compartiendo recursos y servicios, la integración de MoviLog con Servicios Web Semánticos representa una poderosa herramienta para explotar las ventajas de la movilidad de agentes y de la gran cantidad de información y servicios existentes en la Web actual. Así, se ha extendido MoviLog dando origen a SWAM, lenguaje que implementa un mecanismo MRF generalizado llamado MIG. SWAM posibilita la interacción con Servicios Web Semánticos, lo que permite alcanzar una efectiva integración de los agentes a la Web Semántica.

En el siguiente capítulo se expondrán ejemplos de aplicaciones basadas en agentes SWAM con el objeto clarificar los conceptos e ideas expuestos en este capítulo. Particularmente, se

analizarán dos ejemplos de aplicaciones, la primera incluyendo un agente encargado de llevar a cabo de reservas hoteleras y de pasajes aéreos dado un itinerario determinado, y la segunda constituida por un agente para la distribución de artículos científicos en la fase de revisión de una conferencia. Esta última ejemplifica el uso de los mecanismos de *matching* semántico considerados por MIG para la búsqueda semántica de Servicios Web presentados hacia el final del presente capítulo.

Capítulo 6

Ejemplos de aplicación

Para construir agentes utilizando SWAM, es necesario escribir un programa compuesto de tres secciones, cada una conteniendo un conjunto de cláusulas Prolog con una determinada estructura. A grandes rasgos, la primer sección del código (*PROTOCOLS*) incluye las descripciones genéricas de los recursos que activarán el mecanismo de MIG; la segunda sección (*POLICIES*) contiene las políticas personalizadas definidas por el programador para el acceso a dichos recursos; y la tercera y última sección (*CLAUSES*) se compone de reglas que determinan el comportamiento básico del agente. Así, las aplicaciones construidas con SWAM se componen de agentes móviles con un comportamiento y objetivos determinados, y que interactúan con los recursos de una red de manera transparente al desarrollador de la aplicación.

En el presente capítulo se describirá en detalle la implementación de dos aplicaciones codificadas en SWAM. Básicamente, dichas aplicaciones están constituidas por agentes móviles que interactúan con Servicios Web para la resolución de sus tareas. En particular, se expondrá una aplicación que consiste de un agente encargado de la construcción de itinerarios turísticos a través de varias ciudades, llevando a cabo las reservas hoteleras y de pasajes aéreos correspondientes interactuando con Servicios Web ofrecidos por empresas hoteleras y aerolíneas. Posteriormente, se expondrá una aplicación conformada por un agente móvil cuyo objetivo es la distribución de artículos científicos a los evaluadores asignados a éste último en el contexto de una conferencia científica. Adicionalmente, esta aplicación ejemplifica el uso de las facilidades de SWAM para el aprovechamiento de la semántica de los Servicios Web involucrados.

A continuación, se describe la organización del capítulo. La sección 6.1 expone la implementación de un agente SWAM para la construcción de itinerarios turísticos. Más tarde, la sección 6.2 describe la implementación de un agente móvil para la administración de la fase de revisión de artículos de una conferencia científica. Por último, en la sección 6.3 se presentan las conclusiones del capítulo.

6.1. Agente para la creación de itinerarios turísticos

En este apartado se expone en detalle una aplicación programada en SWAM, que consiste de un agente encargado de construir itinerarios entre diversas ciudades, automatizando el

proceso de reservas hoteleras y aéreas para completar cada escala del recorrido. El contexto de ejecución se sitúa en una empresa que ofrece diversos paquetes turísticos, administrando su venta mediante un sistema implementado mediante agentes móviles. Cada vez que un cliente desea adquirir un paquete turístico, se delega la solicitud a un agente encargado de planificar el itinerario entre el origen y el destino solicitados, reservando adecuadamente los pasajes aéreos y las habitaciones en los hoteles con el fin de completar las escalas propuestas en el itinerario. Se supone que las empresas involucradas en el proceso proveen soporte de reservación y venta vía Servicios Web.

En la figura 6.1 se muestra un diagrama de alto nivel de las tareas llevadas a cabo por cada uno de los actores que participan en el proceso de venta de un paquete turístico. Al recibir una nueva solicitud de un cliente, el agente de viaje construye un plan adecuado en función de preferencias del cliente, tales como las ciudades intermedias a recorrer y los días de permanencia en cada una de ellas. Por cada tramo del recorrido generado, el agente solicita un vuelo en alguna aerolínea para completar la escala en función de la fecha planeada para ésta última. Adicionalmente, el agente realiza las reservas hoteleras necesarias. Finalmente, el *schedule* resultante y las reservas efectivizadas son retornadas al cliente, registrando la transacción en una base de datos privada de la empresa.

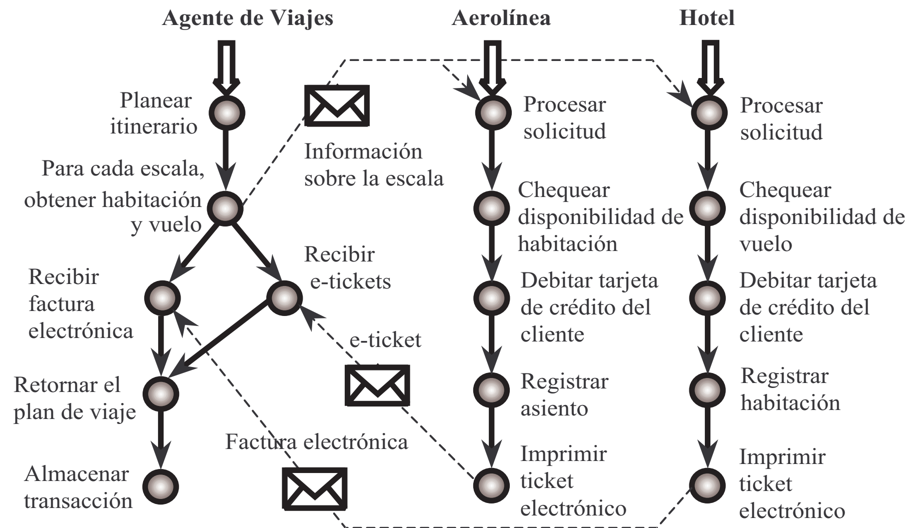


Figura 6.1: Escenario de ejecución del agente planificador

Observando la lista de tareas llevadas a cabo para procesar una nueva solicitud, puede concluirse que el agente interactúa con dos tipos de recursos a lo largo de su tiempo de vida: Servicios Web y conexiones a bases de datos. Por un lado, el agente efectiviza las reservas de los pasajes de avión y las habitaciones en los diferentes hoteles invocando Servicios Web publicados por las aerolíneas y las empresas hoteleras. Una vez que la solicitud ha sido procesada exitosamente, el agente debe registrar la transacción en una base de datos de la empresa, por lo que necesitará una conexión a ésta con el fin de almacenar la información resultante de procesar la solicitud. Consecuentemente, la falla en la obtención de un Servicio Web o una conexión a la base de datos de ventas en el sitio actual donde el agente ejecuta provocará la activación del mecanismo de movilidad inteligente generalizada. MIG obtendrá los sitios de

la red SWAM que ofrecen el recurso faltante, para luego intentar acceder a alguna instancia particular del mismo.

El código SWAM que implementa al agente de viaje se analiza a continuación. Por cuestiones de simplicidad, no se han tenido en cuenta las fechas involucradas en el itinerario a la hora de realizar las reservaciones. Por razones similares, se ha omitido el tratamiento de posibles errores y fallas en la ejecución de los Servicios Web, causados por situaciones tales como saldo insuficiente en la tarjeta de crédito del cliente, no disponibilidad de pasajes aéreos y/o habitaciones, etc.

La primer parte del código mencionado es la sección PROTOCOLS. Esta sección declara todos los protocolos del agente, o en otras palabras, las descripciones de los recursos cuya falla provocará la activación del mecanismo de MIG. Como puede verse, se describen tres recursos: una conexión a la base de datos de ventas, cuyo nombre se supone ser *sellings*, un servicio Web de reserva de pasajes aéreos (*bookFlight*) y un Servicio Web para reservas hoteleras (*bookHotelRoom*). Con respecto a los dos últimos, nótese que la descripción es lo suficientemente general como para abstraerse de la empresa u organización que provee los servicios en cuestión. Adicionalmente, los datos de contacto de un servicio (dirección del servidor que lo ofrece, protocolo de transferencia, codificación de argumentos, etc.) permanece encapsulada en su documento WSDL asociado, el cual es utilizado por SWAM al momento de invocar dicho servicio.

```
PROTOCOLS
  protocol(dataBaseConnection, [dbName(sellings)], dbPolicy).
  protocol(webService, [name(bookFlight), input(X)], wsPolicy).
  protocol(webService, [name(bookHotelRoom, input(X))], wsPolicy).
```

La segunda sección del código (POLICIES) está constituida por las políticas de acceso a los recursos en caso de una falla en su obtención. En este caso, se han declarados dos políticas, denominadas *dbPolicy* y *wsPolicy*, asociadas al protocolo que describe la conexión a la base de datos de ventas y a los servicios de reservas, respectivamente. La primera de ellas establece que el agente siempre se moverá al sitio donde se provean conexiones a la base de datos si la carga de CPU en éste es menor o igual al 50 %, ignorando otra forma de acceso alternativa para tal fin. En otras palabras, el agente sólo accederá al recurso bajo el mecanismo de movilidad propiamente dicho si el uso de CPU en el sitio que provee el recurso en cuestión no supera el 50 %. La segunda política, por su parte, establece restricciones sobre la fuente desde la cual se accederá a un Servicio Web. En este caso, dados dos sitios que ofrecen invocación a los Servicios Web requeridos, el acceso efectivo se realizará desde el sitio con menor carga de CPU. Es preciso notar que, para ambas políticas, *CPULoad* es un predicado de profiling de la plataforma SWAM que retorna, dado un sitio, su porcentaje actual de utilización de CPU. De esta manera, las políticas de recuperación y acceso a los recursos mencionados podrían haber sido construidas utilizando otros predicados de profiling, relacionados por ejemplo con la performance actual de la transferencia de datos a través de los vínculos de comunicación.

```
POLICIES
  accessWith(dbPolicy, [ResourceID, Site], move, _, move):-
    CPULoad(Site, Load), Load <= 50.
  sourceFrom(wsPolicy, [ID1, Site1], [ID2, Site2], Result):-
    leastCPUUsage(Site1, Site2, Result).
```

La tercera y última sección del código, denominada CLAUSES, implementa mediante reglas Prolog el comportamiento del agente planificador. Aquí, mediante la regla *?-doService*, el algoritmo de planificación construye un itinerario a través de ciertas ciudades (proceso no relevante para el ejemplo), efectúa las reservaciones correspondientes, y finalmente almacena la transacción realizada. Las cláusulas *webService* y *dataBaseConnection* de esta sección constituyen los puntos potenciales de activación del mecanismo de movilidad MIG de SWAM. Cuando el algoritmo de ejecución del agente evalúa una cláusula cuyo functor corresponde al nombre de un recurso declarado como protocolo en la sección PROTOCOLS, y a su vez el primer argumento de dicha cláusula corresponde a la lista de propiedades del recurso especificadas para el protocolo, dicha cláusula no se evalúa bajo el algoritmo de evaluación estándar de Prolog. En este caso, la cláusula se interpreta como un punto en el cuál es necesario recuperar el recurso descrito por el protocolo asociado para que el agente continúe con su ejecución normal.

```
CLAUSES
  leastCPUUsage(Site1, Site2, Site1):-
    CPULoad(Site1, Load1),
    CPULoad(Site2, Load2),
    Load1 <= Load2, !.
  leastCPUUsage(_, Site2, Site2).
  scheduleCircuit(Origin, Destination, Cities, Schedule):-
    bookFlightsAndHotelRooms([Destination], [], 0).
  bookFlightsAndHotelRooms([C1Info, C2Info|Cities], [SchInfo|Sch], Cost):-
    C1Info = city-info(CityFrom, DaysAtCityFrom),
    C2Info = city-info(CityTo, DaysAtCityTo),
    webService([name(bookFlight),
               input([CityFrom, CityTo])], FlightInfo),
    webService([name(bookHotelRoom),
               input([CityTo, DaysAtCityTo])], HotelInfo),
    FlightInfo = [cost(TicketCost), ticket(Ticket), airline(Airline)],
    HotelInfo = [cost(RoomCost), number(RoomNumber), hotel(Hotel)],
    ScheduleInfo = [FlightInfo, HotelInfo],
    bookFlightsAndHotelRooms([C2Info|Cities], Sch, SubCost),
    TempCost is SubCost + TicketCost,
    Cost is TempCost + RoomCost.
  storeTransaction(Schedule, Cost):-
    dataBaseConnection([dbName(sellings)], Connection),
    storeTransaction(Schedule, Cost, Connection),
    closeDBConnection(Connection).
  storeTransaction(Schedule, Cost, Connection):- ...
  closeDBConnection(Connection):- ...
  ?-doService(Origin, Destination, Cities, Schedule, Cost):-
    scheduleCircuit(Origin, Destination, Cities, Circuit),
    bookFlightsAndHotelRooms(Circuit, Schedule, Cost),
    storeTransaction(Schedule, Cost).
```

Posiblemente el lector se pregunte en este punto cómo se hace efectiva la recuperación de un recurso a partir de un simple llamado Prolog, o en otras palabras, cómo asocia la plataforma SWAM un llamado del estilo $resourceName([prop_1, prop_2, ..., prop_n], arg_1, arg_2, ..., arg_m)$ con el recurso requerido propiamente dicho. En primer lugar, todo protocolo publicado por un sitio SWAM se compone de un nombre, un identificador único, una lista de propiedades, y una

cláusula Prolog, presente en el sitio local, encargada de las tareas de creación y retorno de la instancia del recurso al agente solicitante. Para un Servicio Web, dicha cláusula se encargará de llevar a cabo la invocación al servicio y retornar los resultados correspondientes. En el caso de una conexión a una base de datos, la cláusula en cuestión tendrá como tarea crear, por ejemplo, un objeto que representa una conexión JDBC (Java Data Base Connectivity), chequear cuestiones de seguridad, y finalmente retornar el objeto creado. En otras palabras, la cláusula de recuperación asociada a cada protocolo implementa la funcionalidad de acceso que depende del tipo de recurso descrito de forma abstracta por el protocolo.

Por último, cada protocolo publicado define la forma en que la cláusula en ejecución a interpretar mencionada más arriba se relaciona con la cláusula que efectiviza la recuperación del recurso. De esta manera, la recuperación de los recursos es totalmente transparente al programador del agente, dado que la plataforma SWAM se encargará de ligar adecuadamente - en tiempo de ejecución - las variables arg_i de la cláusula actual en ejecución con los resultados obtenidos a partir de la evaluación de la cláusula de recuperación particular definida por el protocolo correspondiente.

6.2. Agente para la distribución de artículos científicos

La fase de revisión de contribuciones técnicas de una conferencia científica es un proceso largo y tedioso. En general, para comenzar el proceso de revisión, los autores envían sus artículos al Program Comittee Chair (PCC). Cuando se ha llegado a la fecha máxima fijada para la recepción de artículos, se seleccionan los miembros adecuados del Program Comittee (PC) que efectuarán la revisión de cada uno de los artículos. Para esto, el PCC envía cada artículo junto con un formulario de evaluación a los revisores correspondientes, quienes a su vez completan el formulario con los resultados de la evaluación de los artículos en cuestión, y los envían al PCC. Típicamente, el proceso de revisión finaliza mediante una reunión, organizada por el PCC, para determinar la aceptación o rechazo de cada artículo. A su vez, los autores de los artículos aceptados pueden ser notificados con sugerencias o comentarios para mejorar los mismos. Por último, la versión final de cada artículo es enviada por el PCC, junto con el resumen y la tabla de contenidos de la conferencia, a la entidad encargada de la publicación de los *proceedings*.

Este proceso es perfectamente factible de ser automatizado por una aplicación. Así, los autores enviarían los artículos en forma electrónica a la aplicación, que los organizaría según los tópicos que cubre cada uno. En función de las áreas en las que trabaja cada miembro del PC, los artículos podrían ser enviados a los evaluadores adecuados para su posterior revisión. De forma similar, la aplicación mediaría entre cada revisor y los autores de los artículos para el envío de las sugerencias, y también en la recepción de las copias finales de los artículos que estarán incluidos en la versión impresa de los *proceedings* de la conferencia.

Actualmente, existen diversas aplicaciones que automatizan la fase de revisión de artículos de una conferencia, tales como (Zakon Group, 1992; van de Stadt, 2001; Rigaux, 2004), que han sido utilizadas exitosamente en diversas conferencias alrededor del mundo. Sin embargo, una limitación de dichas herramientas es que, en general, no tienen en cuenta el lenguaje de los evaluadores a la hora de distribuir los artículos para su revisión. Sería interesante que la aplicación traduzca cada artículo de acuerdo al lenguaje preferido del evaluador asignado,

como así también el formulario de evaluación correspondiente. En este sentido, se presentará a continuación un agente SWAM encargado de llevar a cabo la funcionalidad descrita mediante la utilización de Servicios Web Semánticos de traducción de artículos y formularios.

En primer lugar, se asume la existencia de Servicios Web para traducción de documentos. En otras palabras, se supone que existen Servicios Web Semánticos, publicados por determinados sitios de la red, cuya entrada requerida es una instancia particular del concepto de *Documento*, el *Lenguage* fuente y el *Lenguage* al cual el documento quiere traducirse. La salida de dichos servicios será una instancia de *Documento* que representa la entrada traducida al lenguaje especificado. Adicionalmente, pueden existir Servicios Web que son capaces de traducir tipos de documentos más específicos. Por ejemplo, un servicio especializado en la traducción de artículos podría aprovechar las palabras clave del mismo para llevar a cabo una traducción basada en contexto, lo que resulta más adecuado. En este caso, la entrada y la salida del Servicio Web Semántico sería una instancia del concepto de *Artículo*.

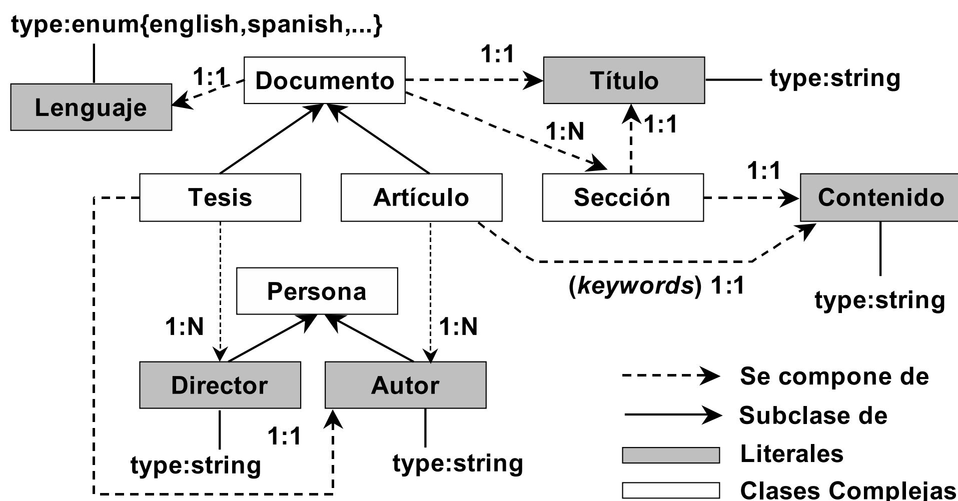


Figura 6.2: Una ontología simple del concepto de *Documento*

En el contexto planteado, se ha definido una ontología de los conceptos involucrados en los Servicios Web de traducción, como muestra la figura 6.2. En particular, el concepto de *Documento* está relacionado con el lenguaje en el cuál está escrito, el título y la lista de secciones del documento, que agrupa la información del documento. Por otra parte, existen clases que especializan el concepto de *Documento*, como pueden ser un artículo científico, compuesto además de autores y palabras claves, o una tesis, que posee relaciones extras que indican su autor y sus directores. Similarmente, los conceptos de *Autor* y *Director* especializan en la ontología definida al concepto de *Persona*. Cabe destacar que se ha simplificado significativamente la definición de la estructura de cada concepto. Sin embargo, la ontología expuesta será suficiente a fin de comprender el ejemplo planteado.

A partir de la figura puede observarse la existencia de dos tipos de conceptos. Por un lado, existen conceptos complejos con una estructura definida, tal es el caso de los documentos. Por otra parte, los conceptos que se encuentran resaltados corresponden a *literales*, esto es, conceptos que están estrechamente relacionados con un tipo de dato básico particular, pero que no poseen una estructura compleja. El algoritmo de *matching* de SWAM que establece

la similaridad entre dos conceptos A y B opera básicamente teniendo en cuenta estructuras complejas y literales. En el primer caso, el *matching* exitoso sucede cuando la estructura de A se compone de clases o subclases de los conceptos incluidos en la estructura de B. En el segundo caso, el *matching* tiene éxito si A y B tienen el mismo tipo de dato. Para el caso de la ontología planteada, la limitación del algoritmo radica en que un valor de tipo *string* hace *match* con más de un concepto literal de la ontología (por ejemplo, "Tim Berners-Lee" puede ser un *Autor*, un *Director*, un *Título* o *Contenido*). Este tipo de ambigüedades afectan negativamente la eficacia del algoritmo de *matching* mencionado, y deben ser solucionadas refinando cuidadosamente la estructura de los conceptos modelados en la ontología.

La tabla 6.1 muestra algunos ejemplos de instancias particulares de conceptos, la clase dentro de la ontología con la cual se corresponden de acuerdo al algoritmo de SWAM, y con qué grado de similaridad. El grado de similaridad entre una instancia particular de un concepto y una clase se define como la distancia entre la clase más específica (menos general) con la que la instancia hace *match* y la segunda. Por ejemplo, una instancia de un artículo científico hace *matching* directo con el concepto de *Artículo* (distancia mínima o cero en el árbol), pero indirecto o menos similar con el concepto de *Documento* (distancia uno en el árbol). Como puede apreciarse, a mayor índice de distancia en el árbol, menor es el grado de similaridad entre los conceptos.

Instancia	Clase	Similaridad
"spanish"	*Lenguaje*	exacta; coincidencia con el tipo de dato del literal
"Tim Berners-Lee"	*Lenguaje*	inexistente; no hay coincidencia en el tipo de dato
"Semantic Web"	*Título*	exacta; coincidencia con el tipo de dato
Instancia de artículo	*Artículo*	exacta; coincidencia con la estructura del concepto
Instancia de artículo	*Documento*	inexacta; *Artículo* especializa la estructura de *Documento*

Tabla 6.1: Ejemplos de instancias de conceptos y similaridad asociada con clases de la ontología

Basado en la ontología expuesta más arriba, el comportamiento básico del agente de distribución de artículos ha sido programado en SWAM, como se muestra a continuación. Básicamente, el agente declara un único protocolo, que representa la necesidad de activar el mecanismo de MIG ante la falla en la obtención de un Servicio Web cuyo nombre sea *translate*, sin describir restricciones sobre la entrada del mismo. En base al formulario de evaluación y al artículo a distribuir, ambos representados mediante estructuras Prolog, el agente busca Servicios Web de traducción disponibles para realizar la traducción de los documentos mencionados al lenguaje particular de cada evaluador. El artículo y el formulario de evaluación están representados mediante estructuras Prolog que forman parte de la base de conocimiento del agente, y el proceso de distribución asociado a éstos comienza cuando la regla *distribute(Reviewers)* es evaluada.

```
PROTOCOLS
  protocol(webService, [name(translate), input(X)], none).
POLICIES
  % empty section
CLAUSES
  review-form(document(title('Review␣Form'),
                       language(english),
                       sections([...]))).
  article(document(title('Jena:␣A␣Semantic␣Web␣Toolkit'),
                   author('Brian␣Mc␣Bride'),
                   language(english),
                   keywords('Semantic␣Web,␣Ontology,␣Databases'),
                   sections([...]))).
  language(document(Data), Lang):- member(language(Lang), Data).
  translate(Doc, To, Doc):- language(Doc, To), !.
  translate(Doc, To, Res):-
    language(Doc, From),
    Input=[Doc, From, To],
    webService([name(translate), input(Input)], Res).
  distribute([]).
  distribute([Rew|Rews]):-
    article(A),
    review-form(F),
    Rew=reviewer(Name, Mail, Lang),
    translate(A, Lang, At),
    translate(F, Lang, Ft),
    send-mail(Name, Mail, attachs([At, Ft])),
    distribute(Rews).
  send-mail(Name, Mail, Attachs):- ...
```

La parte interesante del código del agente está representada por el predicado *webService* que se encuentra en el cuerpo de la regla *translate(Doc, To, Res)*. Dado que el predicado coincide con la definición de un protocolo, la evaluación del mismo activará el mecanismo de MIG. Como consecuencia, el agente solicitará a la plataforma las instancias de Servicios Web publicadas en la red SWAM cuyo nombre sea "translate", y cuya entrada requerida sea una instancia del concepto de *Documento* y dos instancias del concepto de *Lenguaje*. De esta manera, se inicia la búsqueda *semántica* de los servicios que cumplen con las restricciones de entrada mencionadas. Para esto, se toma como entrada la lista de servicios cuyo nombre coincide con el solicitado por el agente, y posteriormente se establece la similaridad entre las instancias de conceptos de entrada (*Input*) con los conceptos de entrada declarados por cada servicio. Esta tarea es responsabilidad de la estrategia de correspondencia definida para cada servicio (recurso) publicado, que en este caso establecerá la necesidad de una compatibilidad sintáctica para la propiedad "name", y una compatibilidad semántica para la propiedad "input". La compatibilidad sintáctica se cumplirá siempre y cuando el servicio haya sido publicado bajo el nombre "translate". Por otra parte, la compatibilidad semántica será exitosa si cada elemento de los argumentos de entrada proporcionados por el agente se corresponde con cada concepto de entrada (uno a uno) definido por el servicio. En otras palabras, el algoritmo de *matching* de conceptos debe retornar una similaridad positiva entre la lista de entrada para invocar al servicio y la lista de conceptos requerida como entrada por éste último. Nótese que la estrategia de correspondencia también podría requerir una compatibilidad semántica para el nombre

del servicio; en este caso, el agente debiera indicar como nombre del servicio un concepto - perteneciente posiblemente a una ontología de acciones - que denote la funcionalidad que éste último debe ofrecer.

Ahora, considérese la existencia de servicios de traducción de documentos (traducción literal) y de artículos (traducción contextual). De acuerdo a la ontología anteriormente expuesta, un servicio de traducción de documentos requerirá como primer argumento de entrada una instancia del concepto *Documento*, mientras que un servicio específico de traducción de artículos científicos demandará, para el mismo argumento, una instancia del concepto de *Artículo*. En este sentido, cuando el agente solicite los servicios disponibles para traducir el formulario de evaluación, la plataforma SWAM le retornará la lista correspondiente al primer conjunto de servicios, ya que la estructura de un formulario no coincide con la estructura del concepto de *Artículo*. Sin embargo, ante la solicitud de servicios para traducir el artículo, la lista resultado estará compuesta de los servicios de traducción de documentos y artículos, ya que la estructura del artículo en cuestión se corresponde semánticamente con la estructura del concepto de *Documento* y *Artículo*, con mayor similaridad con respecto a éste último. Para priorizar la utilización de los servicios con mayor similaridad semántica, la plataforma ordena la lista de los servicios resultantes en orden creciente de acuerdo al grado de similaridad existente entre la solicitud del agente y la entrada conceptual requerida por cada uno de los servicios. Para el caso del ejemplo expuesto, esto evita la posibilidad de realizar una traducción literal para el caso en que existan servicios que ofrecen funcionalidad de traducción contextual - más adecuada, por cierto - para un artículo.

6.3. Conclusiones

Los agentes móviles resultan ideales para explotar las características de la Web actual (Hendler, 2001). Las propiedades de autonomía e inteligencia que poseen pueden beneficiar la automatización de la interacción de las aplicaciones con los recursos Web y, en particular, con los Servicios Web. Por un lado, la autonomía permite la materialización de aplicaciones que no necesitan interacción con el usuario para resolver una tarea específica. Sin embargo, para alcanzar una verdadera interacción automatizada con los Servicios Web, es necesario que los programas entiendan la *semántica* de cada servicio. Esto implica que la aplicación comprenda las descripciones semánticas de los servicios para seleccionar aquellos que ofrecen una funcionalidad requerida. En este sentido, la facilidades de inferencia y manejo de conocimiento conceptual que poseen los agentes pueden simplificar enormemente esta tarea.

Con el fin de apreciar las ventajas señaladas, en este capítulo se describieron en detalle dos aplicaciones que ejemplifican la forma en que los agentes móviles SWAM pueden interactuar con los Servicios Web. En particular, se expuso una aplicación para la reserva de hospedaje y movilidad aérea de un itinerario turístico, materializada a través de un agente móvil que interactúa con Servicios Web sin tener en cuenta la semántica o significado de cada uno de ellos. Posteriormente, se mostró una aplicación para la distribución de artículos en el contexto de la fase de revisión de una conferencia científica. En este caso, la aplicación se implementó a través de un agente móvil capaz de interactuar con Servicios Web semánticos. Como pudo observarse, el agente representa mediante estructuras Prolog la estructura de los conceptos que son utilizados para hallar Servicios Web cuya funcionalidad es compatible con lo que dicho agente necesita. De esta manera, la lista de servicios que la plataforma SWAM retorna

ante la solicitud de un servicio está formada por aquellas instancias cuya entrada (o salida) conceptual es similar a los conceptos de entrada (o salida) indicados por el agente.

El siguiente capítulo se concentrará en describir en detalle el diseño e implementación de la plataforma SWAM, poniendo énfasis en el modelo de ejecución que permite la búsqueda y utilización de recursos Web de forma transparente al programador. Asimismo, se mostrará el soporte que ofrece la plataforma para la administración de la semántica de los Servicios Web.

Capítulo 7

Diseño e implementación

En este capítulo se presentarán los principales aspectos relacionados con el diseño e implementación de la plataforma SWAM expuesta en el capítulo 5. En particular, se analizarán los componentes extras adicionados a la arquitectura básica de MoviLog, y se describirán en detalle los mecanismos soportados para permitir a los agentes interactuar con Servicios Web Semánticos. En ambos casos, se detallará la funcionalidad ofrecida, junto con los detalles de implementación más relevantes. El capítulo incluye algunos diagramas en notación Unified Modeling Language (UML) para facilitar la comprensión del diseño de los componentes y la forma de interacción y colaboración existente entre éstos.

El capítulo se organiza como se resume a continuación. La sección 7.1 detalla la implementación de la arquitectura de SWAM, haciendo hincapié en los componentes adicionados para proveer la funcionalidad extendida que fue introducida en el capítulo 5. Más tarde, la sección 7.2 expone los detalles considerados para la materialización del soporte de invocación e interacción con Servicios Web desde la plataforma SWAM. Por último, la sección 7.3 presenta las conclusiones del capítulo.

7.1. Arquitectura de SWAM

Como se explicó a lo largo del capítulo 5, la plataforma MoviLog no resulta apta para la programación de agentes móviles en el contexto de la Web Semántica. Esto radica principalmente en el hecho de que el mecanismo de protocolos provisto por MoviLog no es lo suficientemente expresivo para poder describir cualquier tipo de recurso, o en este caso, Servicios Web. Más aún, el soporte de tiempo de ejecución de MoviLog no ofrece a los agentes la posibilidad de describir e interactuar con recursos diferentes de predicados Prolog. En función de estos nuevos requerimientos, se han dotado a los MARlets de cada sitio con los componentes necesarios para materializar la funcionalidad mencionada.

La figura 7.1 muestra la arquitectura de SWAM. Los componentes más importantes que fueron creados para ofrecer soporte de protocolos genéricos lo constituyen el *Resource Access Manager*, el *Profiler Manager* y el *Protocol Container*. El primero de ellos se encarga de administrar las estrategias de acceso a los recursos de acuerdo a las características particulares

de éstos. Por su parte, el *Profiler Manager* incluye funcionalidad de cálculo y actualización de las métricas del sistema (carga CPU, memoria libre, velocidad de transmisión, etc.), además de proveer una interfaz a los agentes para utilizar tales métricas. El diseño asociado a estos dos componentes se explica con mayor detalle en las secciones 7.1.2 y 7.1.3, respectivamente.

El componente *Protocol Container* es el encargado de almacenar los protocolos extendidos que describen los recursos ofrecidos por el sitio local. Adicionalmente, resuelve las solicitudes que el componente *RMF Manager* realiza cuando se producen *fallas* a nivel de MIG, fundamentalmente consultas cuyo objetivo es conocer si el sitio local implementa los protocolos asociados a las fallas suscitadas. A grandes rasgos, este componente materializa un *Service Provider JNDI*, es decir, un servidor que responde a la interfaz de directorios remotos JNDI (Java Naming Directory Interface) (Lee y Seligman, 2000). Esta interfaz permite construir y administrar una estructura jerárquica de objetos remotos, similar a la de un sistema de archivos convencional, asociando propiedades a los objetos y realizando consultas en base a ellas. En la siguiente sección se expone en detalle la implementación del componente *Protocol Container*.

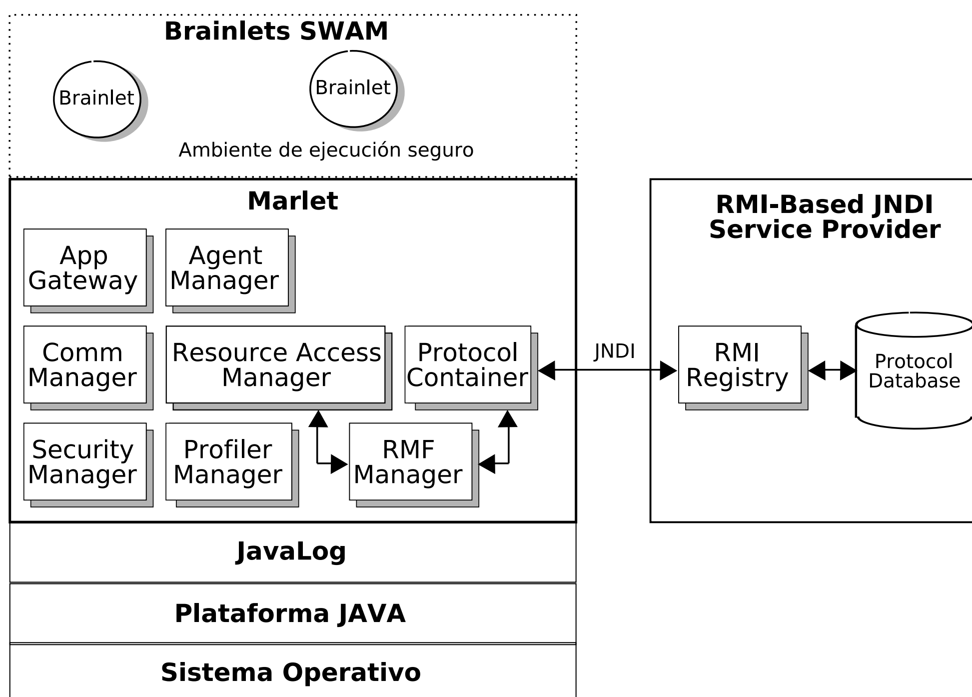


Figura 7.1: Arquitectura de SWAM

7.1.1. El componente *Protocol Container*

El componente encargado de mantener la información sobre los recursos ofrecidos por un sitio SWAM se denomina *Protocol Container*. Para esto, el *Protocol Container* administra una estructura de tipo árbol n-ario que incluye los protocolos ofrecidos tanto por el sitio local como los demás sitios. Por cada sitio (incluido el local) existe un nodo en el árbol cuyo rótulo es la dirección IP del mismo, y donde los hijos son nodos que representan la información de

los recursos compartidos por el sitio, agrupados según una categoría determinada (cláusulas, archivos, Servicios Web, bases de datos, etc.). Adicionalmente, cada nodo de una categoría posee tantos hijos como instancias particulares de ese recurso incluya; por ejemplo, un sitio determinado puede desear compartir dos o más *archivos*. A modo de ejemplo, la figura 7.2 muestra el árbol de recursos correspondiente a un sitio cuya dirección IP es A, y que integra una red lógica compuesta por otros dos sitios B y C. En este caso, los sitios han publicado a la plataforma información sobre diferentes archivos físicos y servicios ofrecidos por cada uno de ellos.

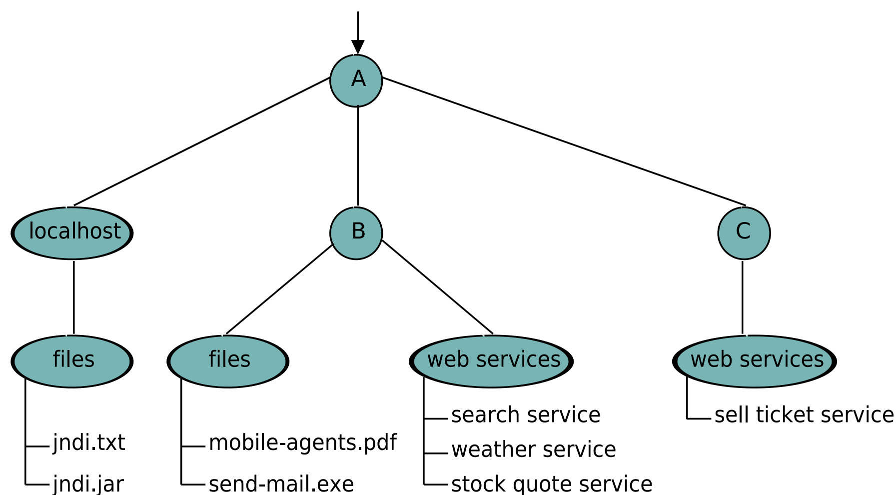


Figura 7.2: Ejemplo de un árbol de recursos de un sitio SWAM

El *Protocol Container* de cada sitio está implementado como un *Service Provider* para el protocolo[1] RMI (Remote Method Invocation), de acuerdo a la interfaz JNDI. JNDI es una API que provee funcionalidad de nombrado y organización de objetos, sin importar su ubicación real. En particular, los objetos se estructuran en forma de directorios, pudiendo contener referencias a otros directorios y objetos tanto locales como remotos. La figura 7.3 muestra la arquitectura de JNDI.

Básicamente, la arquitectura de JNDI consiste de una API para el lenguaje JAVA, y una interfaz llamada Service Provider Interface (SPI). Las aplicaciones JAVA utilizan la API para acceder, consultar y/o modificar la estructura de directorios. Por otra parte, el SPI permite aislar el mecanismo particular empleado para acceder a los directorios y objetos remotos; por ejemplo, es posible crear un árbol de directorios accesibles a través de RMI, cada uno conteniendo nodos que referencian objetos CORBA remotos.

El SPI provisto en la implementación de SWAM está basado en RMI. Esta elección se debe en gran parte a la simplicidad de dicho mecanismo y a la facilidad de configuración que lo caracteriza. Sin embargo, el hecho de haber utilizado JNDI permite cambiar fácilmente el SPI (por ejemplo, uno basado en CORBA) sin afectar la organización del árbol de recursos ni la forma en que la plataforma lleva a cabo actualizaciones sobre el mismo.

Claramente, la estructura del árbol de recursos de un sitio debe adaptarse a los cambios

[1] Aquí se ha utilizado el término *protocolo* en el sentido de un mecanismo de comunicación a través de una red

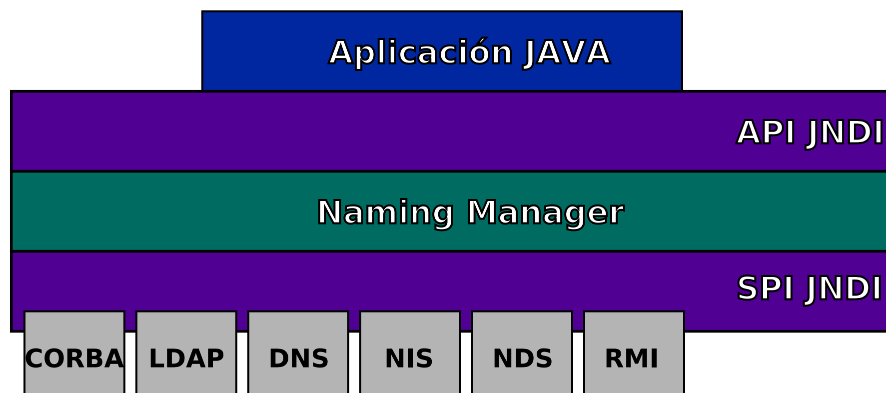


Figura 7.3: Vista general de la arquitectura de JNDI

producidos en la red SWAM a medida que los sitios publican nuevos recursos, o bien cuando dejan de ofrecer un determinado protocolo. Los componentes que se encargan de proveer esta funcionalidad dentro de un sitio son los agentes PNS. Estos agentes se comunican con sus pares remotos para intercambiar información relacionada a la registración o deregistración de protocolos, y actualizan la información del árbol local de recursos en base a dicha información. Adicionalmente, y ante la falla en el acceso a un recurso por parte de un agente móvil, MIG solicita a los agentes PNS qué sitios de la red proveen el recurso requerido, quienes a su vez consultan al árbol de recursos para resolver la solicitud en cuestión.

7.1.1.1. Protocolos

Como se vio en la sección anterior, cada sitio posee la información relacionada a los protocolos ofrecidos tanto localmente como en el resto de los sitios de la red SWAM. Dicha información se organiza en forma jerárquica de acuerdo a una estructura arborescente, donde los nodos hoja contienen un objeto que representa la información que describe un recurso ofrecido por un sitio, es decir, un *protocolo*. Cada protocolo se compone de los siguientes elementos:

- *Nombre*: Identifica la categoría de recursos a la que pertenece la instancia descrita por el protocolo. Básicamente, es un literal que toma valores dentro de un conjunto de categorías preestablecidas, entre las que figuran *clauses*, *web-services*, *files*, *data-bases*, etc., o en otras palabras, todo recurso o servicio que puede ser compartido. Notar que este elemento determina la ubicación del protocolo en el árbol de recursos local.

- *Aridad*: Es un número entero que define la aridad de las cláusulas Prolog encargadas de la recuperación y retorno de la instancia del recurso, conocidas como *cláusulas interfaz*. Para el caso de un recurso de la categoría *clauses*, esta propiedad se refiere a la aridad del conjunto de cláusulas Prolog descritas por el protocolo. En el caso de un recurso de tipo *Servicio Web*, la aridad corresponderá a la cantidad de argumentos del predicado predefinido provisto por la plataforma para la invocación a Servicios Web.

- *Atributos*: Representa el conjunto de propiedades que caracterizan a una instancia particular de un recurso. Por ejemplo, los atributos relacionados a una base de datos pueden

incluir el nombre, el puerto y la información de acceso (usuario y contraseña). Similarmente, los atributos correspondientes a un Servicio Web incluyen el archivo WSDL asociado, y la cantidad y tipos de los argumentos de entrada y salida. El soporte de tiempo de ejecución de MIG utiliza los *atributos* para determinar las instancias de los recursos que satisfacen el intento de acceso no exitoso de un agente a un recurso del sitio local.

- *Entradas (Input Mappings)*: Es una tabla que relaciona los atributos del protocolo con los argumentos de la(s) cláusula(s) interfaz asociadas al recurso. Al acceder a un recurso, los argumentos de la cláusula interfaz son instanciados, de acuerdo a esta tabla, con los valores de la cláusula que provocó la falla a nivel MIG. Supóngase el caso de un agente que intenta acceder a un Servicio Web. La cláusula que dispara el mecanismo de MIG se compondrá de variables Prolog ligadas con el nombre del servicio requerido, y los datos de entrada deseados. En consecuencia, MIG buscará una instancia de un Servicio Web que satisfaga la petición del agente y, posteriormente, accederá a la cláusula interfaz encargada en este caso de invocar al servicio. Aquí, el argumento que representa la entrada al servicio de la cláusula interfaz es ligado con los datos de entrada mencionados previamente.

- *Salidas (Output Mappings)*: De forma similar a las *entradas*, las *salidas* de un protocolo relacionan los argumentos de la(s) cláusula(s) interfaz con los atributos del mismo. En el ejemplo anterior, dicha tabla contendrá típicamente una única entrada, que relacionará el argumento que contiene el resultado de la invocación del Servicio Web con una cierta variable de la cláusula en ejecución que provocó la falla MIG.

Un ejemplo ayudará a comprender mejor los conceptos expuestos. Supóngase un agente que declara un protocolo a una conexión a una base de datos, cuyo nombre es *dbName*, y con información de acceso dada por un usuario *default* (sin password). A continuación se muestra el código que implementa dicho agente:

```
PROTOCOLS
  protocol(data-base, [name(dbName), user(default), password('')], none).
POLICIES
  % empty section
CLAUSES
  ...
  sqlQuery(Query, Res):-
    data-base([name(dbName), user(default), password('')], Conn),
    doQuery(Conn, Query, Res),
    closeConnection(Conn).
  ...
```

El código anterior declara, en la sección PROTOCOLS, el protocolo correspondiente a la instancia de la base de datos. Los predicados con el formato *functor([*$p_1(V_1)$*,...,*$p_n(V_n)$*],* *OutputArg*$_1$*,...,OutputArg*$_m$*)* son puntos potenciales de habilitación de MIG. En particular, el predicado dado por *data-base([name(dbName),user(default),password(")],Conn)* representa uno de esos puntos, ya que su functor coincide con el nombre del protocolo declarado, y la lista de estructuras (primer argumento) unifica con los atributos especificados para el protocolo.

Nótese que, para satisfacer el acceso a la base de datos, el predicado en cuestión limita el campo de búsqueda a las instancias cuyos protocolos declaren una tabla de *salidas* de longitud uno. La única entrada de la tabla relacionará la variable que almacena el resultado de la ejecución del predicado predefinido que establece la conexión con la variable *Conn*.

Considérese ahora la existencia de una cláusula interfaz capaz de establecer una conexión con una base de datos cualquiera, con el formato *data-base(User, Pass, Name, R)*, ubicado en el sitio que ofrece la base de datos con las características mencionadas. En consecuencia, el protocolo que describe la base de datos constará de:

- Un *nombre* que categoriza el recurso (*data-base*).
- La aridad de la cláusula interfaz encargada de recuperar una instancia (cuatro en este caso).
- Los atributos, dados por el nombre de la base de datos (*dbName*), y la información de acceso (usuario *default* y sin contraseña). Notar que un conjunto distinto de atributos, por ejemplo haciendo alusión a otro usuario, debe ser declarado como un protocolo diferente por el sitio proveedor.
- Las *entradas*, dadas por un conjunto de pares $< X, Y >$, donde X representa un atributo del recurso, e Y el número del argumento de la cláusula interfaz con el cual se asocia dicho valor. En este caso, el conjunto de pares es {*(user,1),(password,2),(name,3)*}.
- Las *salidas*, o en este caso, el conjunto con un único par {*(1,4)*}. Dicho par mapea el índice del argumento de la cláusula interfaz que se instancia con la conexión con el índice i del argumento del predicado en ejecución donde se espera el resultado del acceso al recurso ($OutputArg_i$).

Así, el acceso a un recurso se divide básicamente en dos partes. En primer lugar, la cláusula en ejecución actual se compara con los protocolos declarados por el agente. En caso de tratarse de un predicado que habilita MIG, se escoge una instancia de un recurso cuyo protocolo coincida con el que ha declarado el agente. Posteriormente, la plataforma se encarga de ligar la lista de propiedades y variables de salida incluidas en el predicado en ejecución con las variables de la cláusula interfaz asociada al recurso elegido, de acuerdo a las *entradas* y *salidas* especificadas para éste último.

El comportamiento relacionado a cada protocolo se implementó mediante la clase ResourceProtocol (ver figura 7.4), que hereda de la clase GenericProtocol. La clase GenericProtocol representa un protocolo general, compuesto de una categoría y un conjunto de atributos que lo identifican. Por su parte, la clase ResourceProtocol modela los protocolos que confían en cláusulas que ofician de interfaz para el acceso a los recursos de los sitios. Como puede apreciarse, la clase ResourceProtocol no posee métodos para obtener las cláusulas interfaz asociadas; en la siguiente sección se observará que este comportamiento está provisto por las políticas de acceso a recursos.

El conjunto de *atributos* que posee un protocolo se diseñó mediante la clase AttributeMap. Esta clase representa una estructura asociativa, formada por pares clave-valor, donde las claves corresponden a los nombres de los atributos definidos para el protocolo. Adicionalmente, por

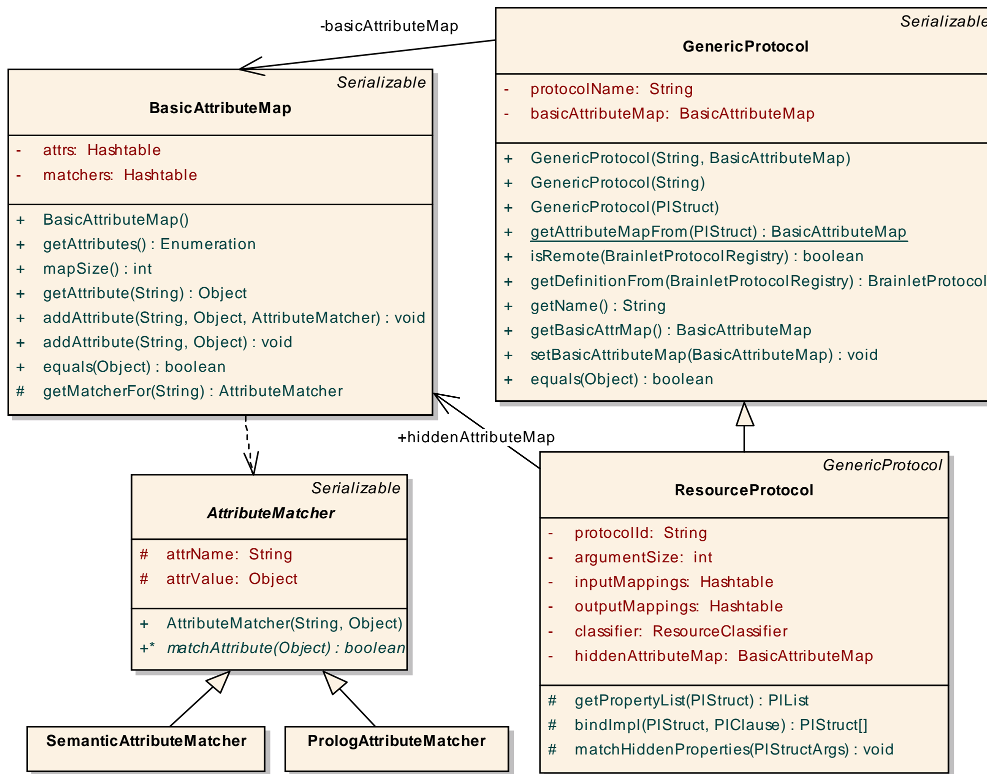


Figura 7.4: La clase ResourceProtocol

cada atributo definido, debe crearse una instancia de la clase AttributeMatcher, diseñada de acuerdo al patrón *Strategy* (Gamma et al., 1995). La clase AttributeMatcher representa todas las clases que implementan una política para decidir si el valor de un atributo coincide con el valor de otro. De esta manera, un protocolo puede definir, para cada atributo declarado, diferentes estrategias de *matching*. Por ejemplo, el protocolo que describe la conexión a una base de datos puede requerir un *match* literal (coincidencia de mayúsculas y minúsculas) para los atributos *user* y *password*, pero sin este tipo de restricciones para el nombre de la base de datos. Finalmente, el protocolo que describe un Servicio Web podría definir un *matching* de tipo semántico para los datos de entrada y salida del servicio, es decir, establecer coincidencia basándose en la posible correspondencia de los conceptos representados por los valores de dichos atributos.

7.1.1.2. Clasificación de los recursos

Al momento de efectivizar el acceso a un recurso, más de un método puede ser aplicable. Por ejemplo, la recuperación de cláusulas Prolog localizadas en un sitio remoto puede llevarse a cabo moviendo los agentes a éste último, copiando el código desde dicho sitio, o incluso enviar el predicado Prolog que provoca la falla para evaluarlo en forma remota. De esta forma, cada vez que un agente intenta recuperar un recurso, SWAM debe determinar, de acuerdo a la naturaleza del recurso, los métodos de acceso que es posible utilizar.

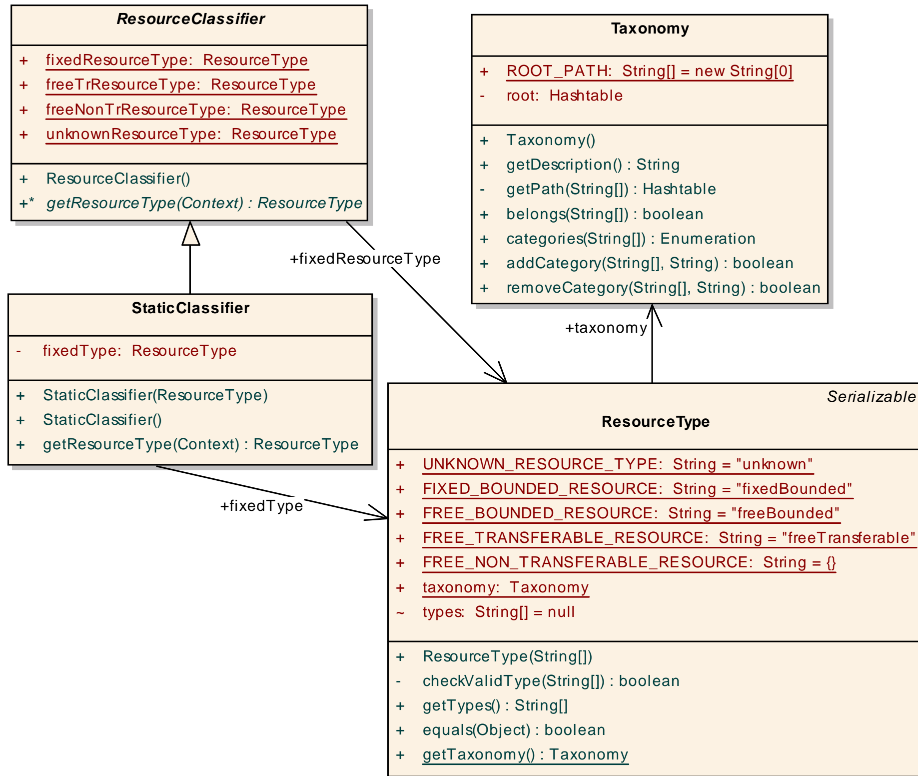


Figura 7.5: Clasificación de recursos: clases principales

SWAM establece las estrategias de recuperación válidas para un recurso basada en su *tipo* (ver capítulo 5). El tipo de un recurso es un valor perteneciente a una taxonomía fija que es conocida y compartida por los sitios de una red lógica SWAM. Dicho valor es calculado en forma dinámica; consecuentemente, el tipo de un recurso puede variar a lo largo del tiempo. Por ejemplo, el tamaño (dinámico) de una base de datos es una característica que podría influir directamente en decidir si el recurso en transferible o no. La figura 7.5 muestra las principales clases que diseñan la funcionalidad de clasificación de recursos en SWAM.

En primer lugar, la clase Taxonomy modela una taxonomía genérica de definiciones representada como un árbol n-ario de tamaño arbitrario y modificable dinámicamente. En particular, SWAM instancia dicha clase con el árbol que categoriza los posibles tipos de recursos ofrecidos. La clase ResourceType representa un tipo de recurso específico, o en otras palabras, un camino completo desde la raíz hasta cualquier hoja en el árbol anterior. Por último, todas las instancias de la clase ResourceProtocol poseen una referencia a una instancia de la clase ResourceClassifier, que representa el algoritmo de clasificación particular de un recurso. Las subclases deben implementar el método abstracto *getResourceType*, que retorna una instancia de la clase ResourceType. Por ejemplo, la clase StaticClassifier implementa un algoritmo de clasificación estático, utilizado para los recursos cuyo tipo es invariable.

7.1.2. El componente *Resource Access Manager*

El componente encargado de administrar las estrategias para el acceso a los recursos en un sitio SWAM se denomina *Resource Access Manager*. Básicamente, este componente registra los métodos válidos para recuperar un recurso de acuerdo a los tipos de recursos definidos en la taxonomía compartida. Adicionalmente, retorna instancias concretas de cada estrategia bajo demanda, a medida que la plataforma le solicita los métodos permitidos para interactuar con un recurso dado.

La figura 7.6 visualiza el diseño de clases del componente *Resource Access Manager*. La clase ResourceAccessManager materializa la funcionalidad básica mencionada más arriba. Dicha clase se diseñó mediante el patrón *Singleton* (Gamma et al., 1995), de modo limitar la cantidad de instancias de la clase a una, asegurando así una única forma de asociar estrategias de acceso a cada tipo de recurso dentro una red lógica.

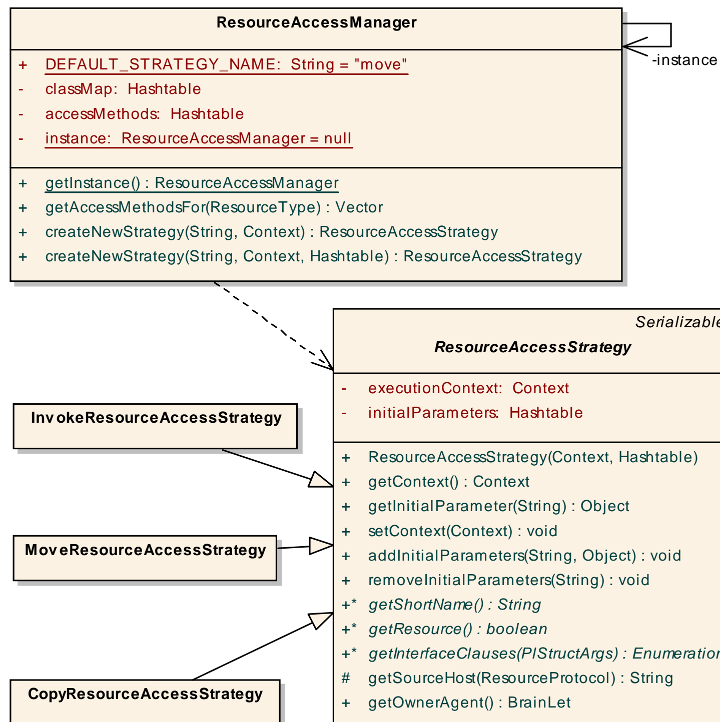


Figura 7.6: Diseño del componente *Resource Access Manager*

La clase ResourceAccessManager mantiene una tabla de *hashing* que almacena pares de la forma $< t, L >$, donde L representa la lista de métodos de acceso permitidos para obtener los recursos del tipo t. Esta lista se compone de strings que identifican unívocamente cada estrategia de acceso, los cuales son utilizados a modo de clave para acceder a una tabla de *hashing* adicional que mapea cada identificador con la clase JAVA que implementa la estrategia correspondiente.

Los diferentes métodos provistos para interactuar con los recursos fueron diseñados mediante la clase abstracta ResourceAccessStrategy. A grandes rasgos, las subclases de ResourceAccessStrategy deben implementar los métodos *getResource*, que retorna un booleano que indica si

es posible acceder a un cierto recurso, y *getInterfaceClauses*, que obtiene las cláusulas interfaz asociadas a dicho recurso para interactuar con él. Actualmente, se han implementado las subclases correspondientes a los métodos de acceso analizados en el capítulo 5. Sin embargo, es posible adicionar nuevos métodos, simplemente creando las entradas necesarias en las estructuras administradas por la clase ResourceAccessManager, e implementando subclases de ResourceAccessStrategy con el comportamiento deseado.

7.1.3. El componente *Profiler Manager*

La idea del componente *Profiler Manager* es proveer un soporte encargado de medir las condiciones de ejecución del servidor de agentes local en un momento dado. Básicamente, el *Profiler Manager* de un sitio SWAM se compone de servicios que pueden agregarse dinámicamente, y cuyo objetivo es dar valor a ciertas métricas de tiempo de ejecución, tales como la cantidad de agentes en el motor local, la carga de CPU, la tasa de transferencia a través de la red, etc. Basándose en estas métricas, los agentes pueden tomar decisiones acerca de la forma más conveniente - por ejemplo, en términos de performance - de acceder a los recursos.

De la misma manera que proporciona medición para las condiciones locales de ejecución, el *Profiler Manager* permite obtener el valor de una métrica en otros sitios de la red. Para esto, los servicios que implementan las métricas pueden ser *inmediatos*, es decir, el valor que retornan ya sea para el sitio local o uno remoto es calculado *on-the-fly*, o *estadístico*, donde el valor se determina promediando una cierta cantidad de valores recibidos previamente desde el sitio remoto. La principal razón por la cual se realizó esta división radica en que la naturaleza fuertemente dinámica de determinadas métricas impide obtener un valor actualizado cuando el mismo es solicitado bajo demanda desde sitios remotos. Por ejemplo, la carga de CPU es una métrica que varía fuertemente a lo largo del tiempo. Las demoras que potencialmente existen al enviar datos a través de la red no permiten que un sitio obtenga una medida lo suficientemente actualizada sobre la carga de CPU de otro sitio. En consecuencia, los agentes que utilizan el valor medido pueden tomar decisiones en base a datos erróneos o no acordes a la realidad. Para aproximar una solución a este problema, el *Profiler Manager* de cada sitio envía periódicamente al resto información sobre los valores actuales de las métricas que varían significativamente con el tiempo.

El componente *Profiler Manager* se implementó mediante la clase ProfilerManager, diseñada mediante el patrón *Singleton* (figura 7.7). Esta clase implementa la interfaz ProfileEventListener, que declara los métodos necesarios para tratar la recepción de mediciones de otros sitios, y la actualización de las estructuras temporales utilizadas para almacenar dichos datos ante la registración o deregistración de sitios de la red. La clase ProfilerManager permite agregar o borrar servicios que implementan el comportamiento de medición de las diversas condiciones de ejecución, materializados a través de instancias de la clase ProfileService. Por último, el comportamiento de obtención y distribución de las métricas estadísticas está implementado mediante la clase ProfilingMonitorThread. Esta clase es en sí un *thread* JAVA que cada cierto tiempo ejecuta los servicios de medición locales de interés, empaqueta los resultados, y comunica la información obtenida al resto de los sitios de la red, haciendo uso de una política de distribución (clase ProfilingBroadCastPolicy).

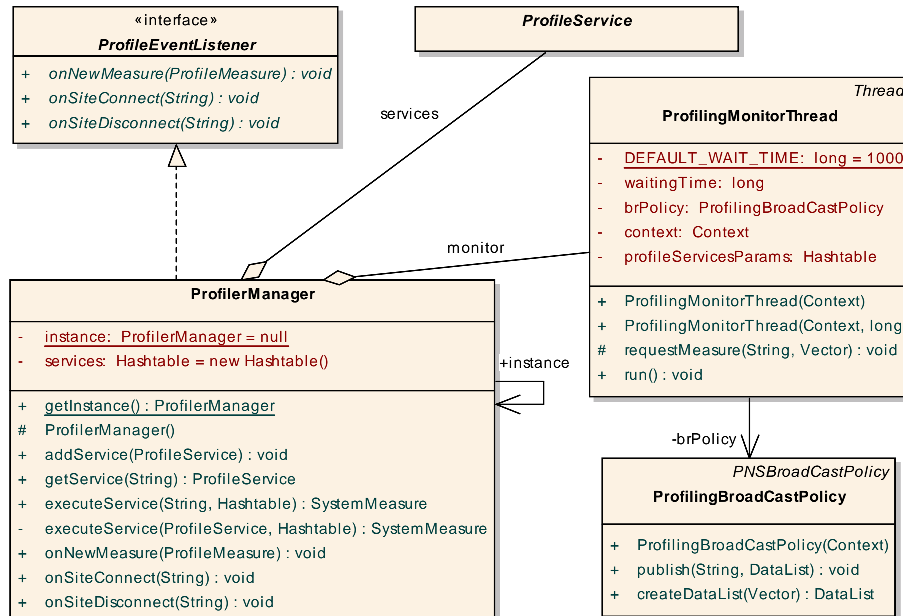


Figura 7.7: Diseño del componente *Profiler Manager*

Los servicios de *profiling* están representados por la clase abstracta ProfileService, que provee los métodos *measureLocal* y *measureRemote* para obtener el valor de una métrica a nivel local y en un sitio determinado, respectivamente (ver figura 7.8). Ambos métodos retornan instancias de la clase SystemMeasure, la cual está compuesta por el valor medido y la unidad de representación del mismo. Como puede observarse a partir de la figura, ciertos servicios están implementados a través de clases que heredan directamente de la clase ProfileService. Por su parte, el resto de los servicios implementados redefinen el comportamiento a DefaultRemoteProfileService, clase que representa la funcionalidad de un servicio estadístico. En este sentido, la clase DefaultRemoteProfileService implementa la interfaz ProfileEventListener, almacenando las mediciones enviadas por los demás sitios, y aplicando un operador estadístico para derivar un valor único en cada caso. El cálculo del *promedio* es responsabilidad de las subclases de DefaultRemoteProfileService a través del método *calculateAverage*, lo que permite computar el mismo a través de diferentes operadores tales como media o mediana.

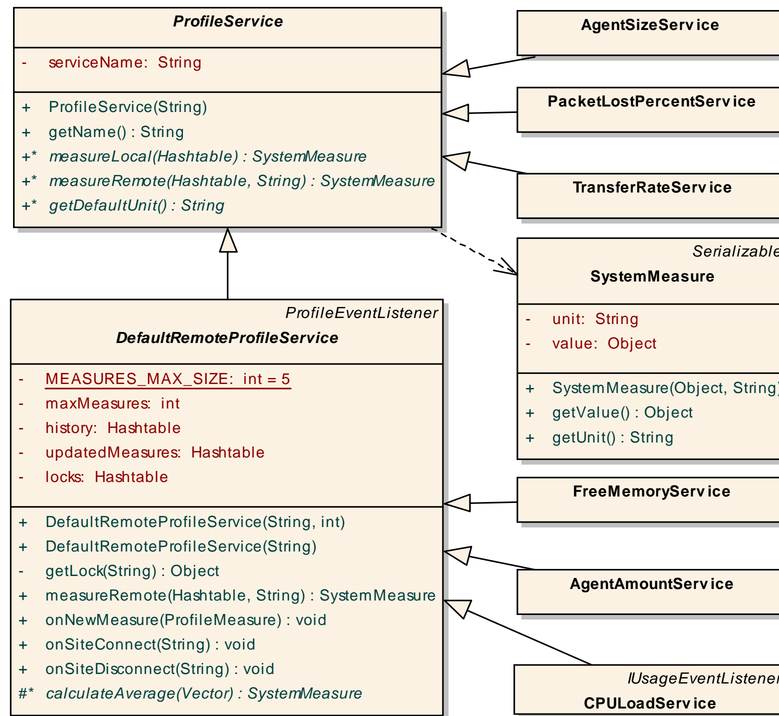


Figura 7.8: Principales clases relacionadas con el diseño de los servicios de *profiling*

Los servicios de medición mencionados son accesibles a los agentes a través del mecanismo de extensión de predicados predefinidos que provee JavaLog, lenguaje al cual MoviLog extiende. Este mecanismo permite incorporar en el motor de agentes de un sitio código Prolog capaz de interactuar con clases JAVA con cierta funcionalidad. Así, fueron implementados seis predicados predefinidos, uno por cada servicio de *profiling* publicado en el *Profiler Manager*.

7.2. Arquitectura de invocación a Servicios Web

Al principio del capítulo se analizaron los elementos que componen los protocolos ofrecidos por un sitio SWAM. Particularmente, se observó que cada protocolo cuenta con una o varias *cláusulas interfaz*, que son cláusulas Prolog encargadas de la recuperación del recurso descrito por el protocolo. En lo que respecta a los Servicios Web, los protocolos asociados se componen de una única cláusula interfaz. Básicamente, la función de esta cláusula es llevar a cabo la invocación de un Servicio Web en base al archivo WSDL que lo describe, cuyo URL es encapsulado por el protocolo asociado al servicio.

El soporte de invocación a Servicios Web se ilustra en la figura 7.9. Desde el punto de vista del programador de un Brainlet, este soporte es accesible vía un predicado predefinido Prolog, denominado *webServiceCall*, que representa la cláusula interfaz que efectiviza el acceso a cualquier Servicio Web. Internamente, dicho predicado utiliza cierta funcionalidad JAVA (*Built-In*) encargada de la traducción de los argumentos del servicio desde Prolog a JAVA, y viceversa. A su vez, el *Built-In* accede a una API provista por un *framework* JAVA denominado WSIF (Web Services Invocation Framework) (Duftler et al., 2001), que se encarga de

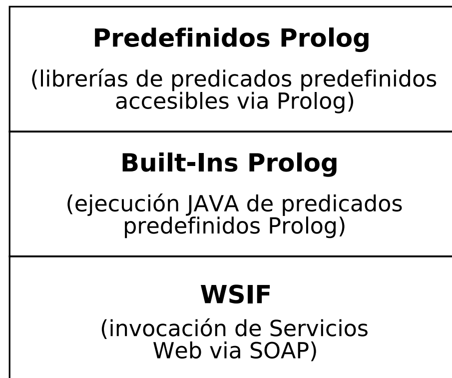


Figura 7.9: Vista general de la arquitectura de invocación a Servicios Web de SWAM

interactuar con el sitio remoto que provee el servicio, enviando los parámetros de invocación provistos por el programador, y retornando la respuesta correspondiente. En este sentido, podemos resumir la secuencia completa de invocación a un servicio Web como sigue:

1. Llamado al predicado predefinido: La *falla* en la obtención de un Servicio Web durante la ejecución de un Brainlet provocará el llamado al predicado *webServiceCall(Name, Params, Response)*, que bloqueará la ejecución del mismo hasta obtener la respuesta al servicio en cuestión. Una invocación favorable al Servicio Web resultará en una evaluación exitosa del predicado, instanciando la variable *Response* con la información que representa la respuesta del servicio. Por otro lado, una invocación que resulte desfavorable derivará en la reevaluación del predicado *webServiceCall*.
2. Acceso al entorno JAVA: Todo predicado predefinido accesible vía Prolog posee una referencia a un método JAVA que se invoca cada vez que el primero sea evaluado. La funcionalidad del método asociado al predicado *webServiceCall* es traducir argumentos de entrada a tipos de datos JAVA, y posteriormente invocar al servicio, utilizando el soporte de acceso provisto por la capa inferior. Similarmente, la respuesta retornada por el servicio es mapeada a estructuras Prolog interpretables por el Brainlet.
3. Invocación efectiva del servicio Web: Este es el último paso en el acceso a un Servicio Web, e involucra todas las operaciones de bajo nivel (conexión con el sitio remoto, serialización de los parámetros según el formato SOAP, parseo de la respuesta, etc.) llevadas a cabo por el *framework* WSIF.

El diseño previamente presentado puede verse como una arquitectura de capas (Shaw y Garlan, 1996). Cada Brainlet accede a los Servicios Web utilizando la capa superior, y la capa inferior representa la funcionalidad de bajo nivel para el acceso a un servicio particular. La capa central provee una interfaz entre el soporte Prolog de invocación a Servicios Web y el código JAVA de acceso real a cada servicio.

Naturalmente, el mayor esfuerzo en cuanto a diseño e implementación fue requerido por la capa intermedia de la arquitectura, cuestiones que se describen en las secciones 7.2.1 y 7.2.2. La primer capa sólo requirió implementación de ciertos predicados, tarea típicamente sencilla debido a las características declarativas del lenguaje Prolog. Finalmente, la capa inferior

no insumió esfuerzo de implementación alguno ya que, como se mencionó más arriba, su funcionalidad está materializada a través de una API ya desarrollada.

7.2.1. Interacción con WSIF

La capa intermedia de la arquitectura de invocación anterior tiene como finalidad principal establecer un "puente" entre el llamado Prolog a un Servicio Web y el acceso de bajo nivel a éste último. Para esto, el nivel superior le comunica el nombre del servicio, los argumentos de entrada, e información encapsulada por el protocolo SWAM asociado a la cláusula interfaz que generó la invocación, por ejemplo, el URL al archivo WSDL que define el servicio. El *Built-In* asociado a dicha cláusula utiliza todos estos datos para conectarse con el servicio.

La clase que implementa la funcionalidad de invocación se denomina StubInvoker (ver figura 7.10). Las instancias de esta clase son en sí un *wrapper* de las clases JAVA que actúan como un *stub* del servicio remoto, similar a los utilizados para acceder a objetos remotos RMI. Básicamente, el *stub* asociado a un Servicio Web es un paquete de clases JAVA generado a partir del WSDL del servicio, mediante la herramienta WSDL2JAVA de WSIF. Este paquete contiene una interfaz por cada *tipo de puerto* y una clase por cada puerto (*binding*) definido para el servicio, además de las clases correspondientes a los tipos de datos complejos declarados. Por cuestiones de performance, SWAM mantiene un repositorio asociando el URL de cada WSDL procesado con el *stub* correspondiente, para futuros accesos. Dicho repositorio es administrado por un componente llamado *Package Manager*, que será descrito en la sección 7.2.3.

La información relacionada a la invocación de un servicio está representada por la clase ServiceInvokeInfo. En particular, la variable *operation* contiene el nombre de la operación deseada, que se corresponde (potencialmente) con un método JAVA de alguna de las clases del *stub* que implementa un tipo de puerto asociado a algún *binding*. Por otra parte, la variable *args* contiene la lista de argumentos del servicio, previamente traducidos desde Prolog a JAVA. Finalmente, el *binding* específico a ser usado durante la invocación está representado como una instancia de la clase PortInfo.

De acuerdo a la especificación WSDL (Christensen et al., 2001), los puertos o *bindings* que implementan un tipo de puerto particular pueden ser varios. Por ejemplo, es común encontrar Servicios Web que hacen accesible una operación ya sea a través SOAP, HTTP GET o HTTP POST. Por esta razón, cada instancia de StubInvoker posee una referencia a una clase denominada PortSelectionPolicy que se ocupa de escoger el *binding* específico a utilizar a partir de un conjunto de opciones. Particularmente, la clase DefaultSelectionPolicy implementa una política de elección de un *binding* al azar. Si bien la implementación actual de SWAM no permite a los agentes tomar esta decisión, se espera desarrollar predefinidos parametrizables con este tipo de políticas.

El método *invokeService* definido en la clase StubInvoker implementa la funcionalidad de invocación de un servicio, en base a los argumentos configurados a cada instancia. Para esto, solicita al *Package Manager* el *stub* asociado al servicio, y valiéndose del mecanismo de introspección de clases que provee JAVA, ejecuta el método de la clase que implementa la operación y *binding* solicitados. El resultado que devuelve el método anterior es traducido al objeto Prolog que corresponda, y retornado al *Built-In*.

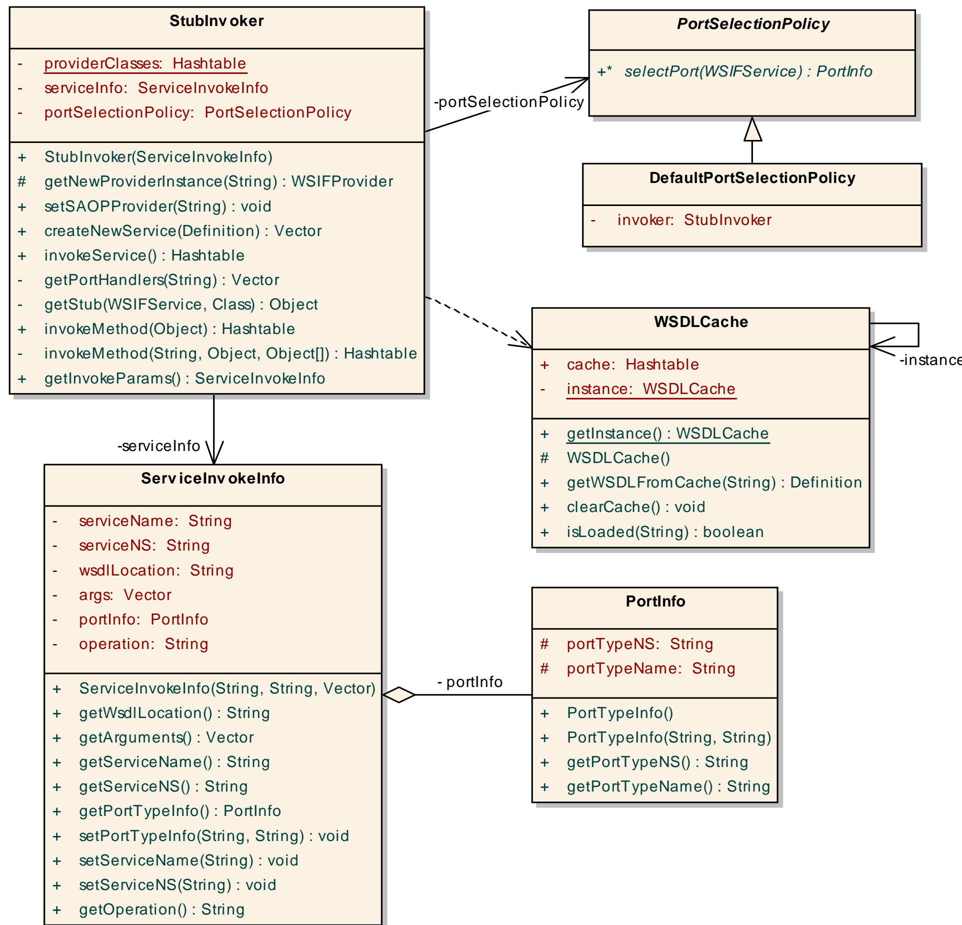


Figura 7.10: La clase StubInvoker

7.2.2. Traducción de tipos de datos

Antes de la ejecución de un servicio, el *Built-In* asociado al predicado de invocación a Servicios Web traduce los argumentos recibidos a tipos de datos JAVA. Para esto, se utiliza el criterio de traducción que se muestra en la tabla 7.1. Aquí, para cada estructura atómica o compleja de Prolog se define un único tipo de dato JAVA. Ciertos tipos de datos básicos, tales como los átomos o valores numéricos, se mapean a un objeto o tipo de dato predefinido JAVA en forma directa. Por otra parte, las listas o las estructuras Prolog propiamente dichas requieren de un proceso de traducción recursivo más elaborado. En el primer caso, un arreglo JAVA es creado, cuyo tipo básico y contenido están determinados por los elementos traducidos de la lista. En el segundo caso, se crea una instancia de la clase ComplexPrologObject, que consiste de un objeto encargado de almacenar los argumentos traducidos correspondientes a la estructura.

Este esquema simple de traducción de tipos de datos adolece de un problema originado a partir del mecanismo provisto por JAVA para la introspección de clases. Como se explicó en la sección anterior, la invocación a un Servicio Web se realiza localizando la clase - dentro de un *stub* - que incluye el método particular que implementa la funcionalidad de conexión con el servicio. Esta localización requiere que los tipos de datos suministrados para invocar

Estructura Prolog	Tipo de dato JAVA
átomo	String
número entero	int
número real	double
lista	array
$functor(arg_1, ..., arg_n)$	ComplexPrologObject

Tabla 7.1: Correspondencia entre las estructuras Prolog y tipos de datos JAVA

dicho método se correspondan uno a uno con los parámetros del mismo, sin excepción. De lo contrario, la ejecución no podrá realizarse. Supóngase por ejemplo un método que recibe un único argumento de tipo *String*, y un agente que pretende parametrizar el servicio asociado con un valor numérico. El parser de Prolog interpretará el argumento como un objeto de tipo entero o real, el cual será oportunamente traducido a un valor JAVA de tipo *int* o *double*, según corresponda. Sin embargo, este tipo de dato no es compatible con la signatura del método, por lo que el servicio no puede ser ejecutado.

El problema anteriormente descrito se solucionó adicionando un paso extra a la traducción de los argumentos de invocación a los servicios. El esquema final se muestra en la figura 7.11. En primer lugar, los argumentos recibidos por el *Built-In* son traducidos a JAVA, según el mapeo definido por la tabla 7.1. Posteriormente, la salida de esta etapa se intenta adaptar a la signatura del método que implementa la comunicación con el servicio. Para el caso del ejemplo anterior, esto equivale a convertir el valor numérico JAVA a su correspondiente representación en formato String. Finalmente, el método en cuestión es cargado, ejecutado, y su respuesta retornada al *Built-In*, previa conversión a su equivalente Prolog.

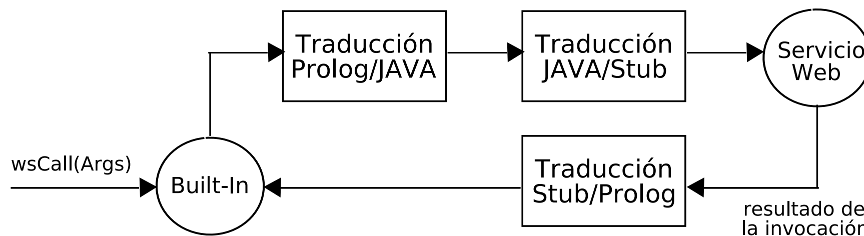


Figura 7.11: Esquema de traducción de los argumentos de invocación a servicios

El diseño de la funcionalidad de traducción expuesta se ilustra en la figura 7.12. Cabe destacar que, por cuestiones de simplicidad, gran parte de las clases no han sido incluídas en el diagrama, ya que el código de conversión de tipo de datos de SWAM actualmente consta de más de 40 clases.

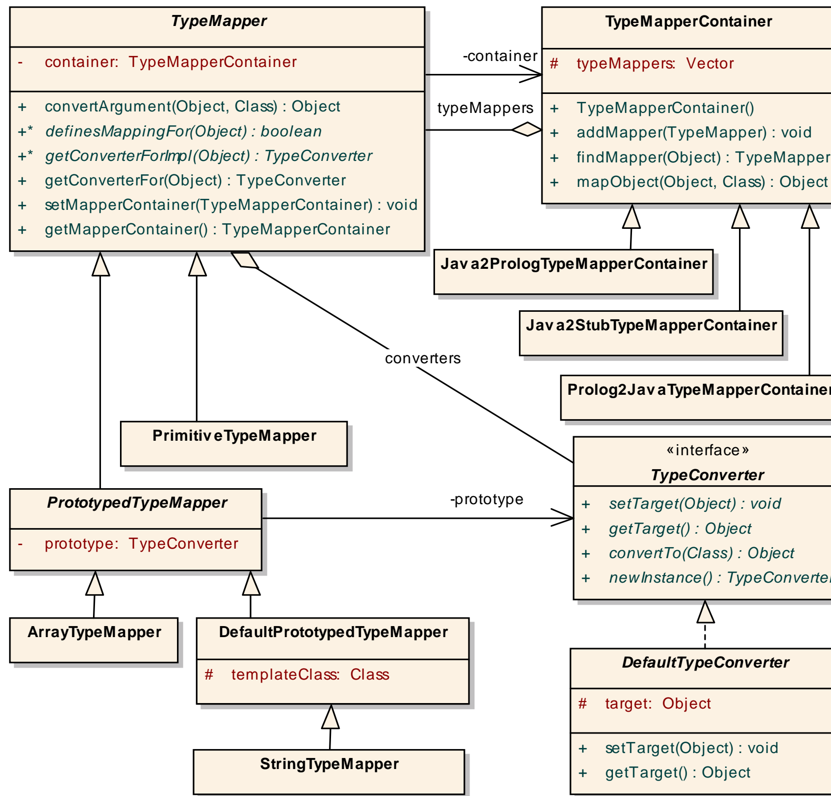


Figura 7.12: Las clases TypeMapperContainer, TypeMapper y TypeConverter

La clase TypeMapperContainer modela el comportamiento de un contenedor de objetos (*mapers*) útil para la conversión de argumentos desde un conjunto de tipos de datos A a otro B. Cada *maper* está implementado a través de la clase TypeMapper, cuyas subclases definen la lógica necesaria para filtrar el subconjunto de tipos de datos de A que están capacitados para traducir, y para proveer el objeto conversor apropiado en cada caso. Cada conversor se encuentra implementado mediante la clase abstracta TypeConverter; las subclases definen el comportamiento específico de conversión a través del método *convertTo*.

Para ejemplificar el diseño presentado, considérese la configuración del componente de traducción de objetos JAVA a estructuras Prolog. Dicho componente es esencialmente un contenedor implementado por la clase Java2PrologTypeMapperContainer, que contiene *mapers* para la traducción de strings, predefinidos, arreglos y objetos JAVA. Finalmente, el *maper* encargado de la traducción de predefinidos JAVA está compuesto de un conversor por cada tipo de dato, vale decir, *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* y *char*.

7.2.3. Deployment de Servicios Web

El componente *Package Manager* tiene como propósito la administración del repositorio de *stubs* local al sitio. Este componente mantiene una estructura que asocia los archivos WSDL

de los servicios que han sido ejecutados con la ubicación (directorio) del paquete de clases que implementa el *stub* del servicio. Cuando un agente solicita la ejecución de un servicio que aún no existe en el repositorio, la atención de la misma es bloqueada hasta que el *Package Manager* lleve a cabo el *deployment* físico de las clases JAVA relacionadas con el servicio.

La clase PackageManager implementa el comportamiento del componente *Package Manager*, implementada como un *Singleton* (figura 7.13). Esta clase administra en el sistema de archivos local el *deployment* de los *stubs* correspondientes a cada WSDL. Cada vez que el *deployment* de un WSDL es iniciado, se chequea que el mismo no se haya hecho con anterioridad. Si esto no se cumple, se crea e inicia un *thread* encargado de generar las clases JAVA asociadas al WSDL, compilarlas, y hacerlas disponibles a los potenciales clientes de los servicios que el WSDL declara. Los *threads* creados son instancias de la clase DeployerThread, la cual provee solución a los problemas de sincronización que el *deployment* acarrea. Por ejemplo, es posible que varios clientes invoquen al mismo tiempo servicios de un mismo WSDL cuyas clases aún no han sido generadas. Para esto, la clase DeployerThread proporciona complejos mecanismos de espera que permiten bloquear tales clientes hasta que la construcción del *stub* en cuestión finalice y, mientras esto ocurre, otros clientes puedan utilizar los *stubs* generados con anterioridad a partir de WSDL diferentes.

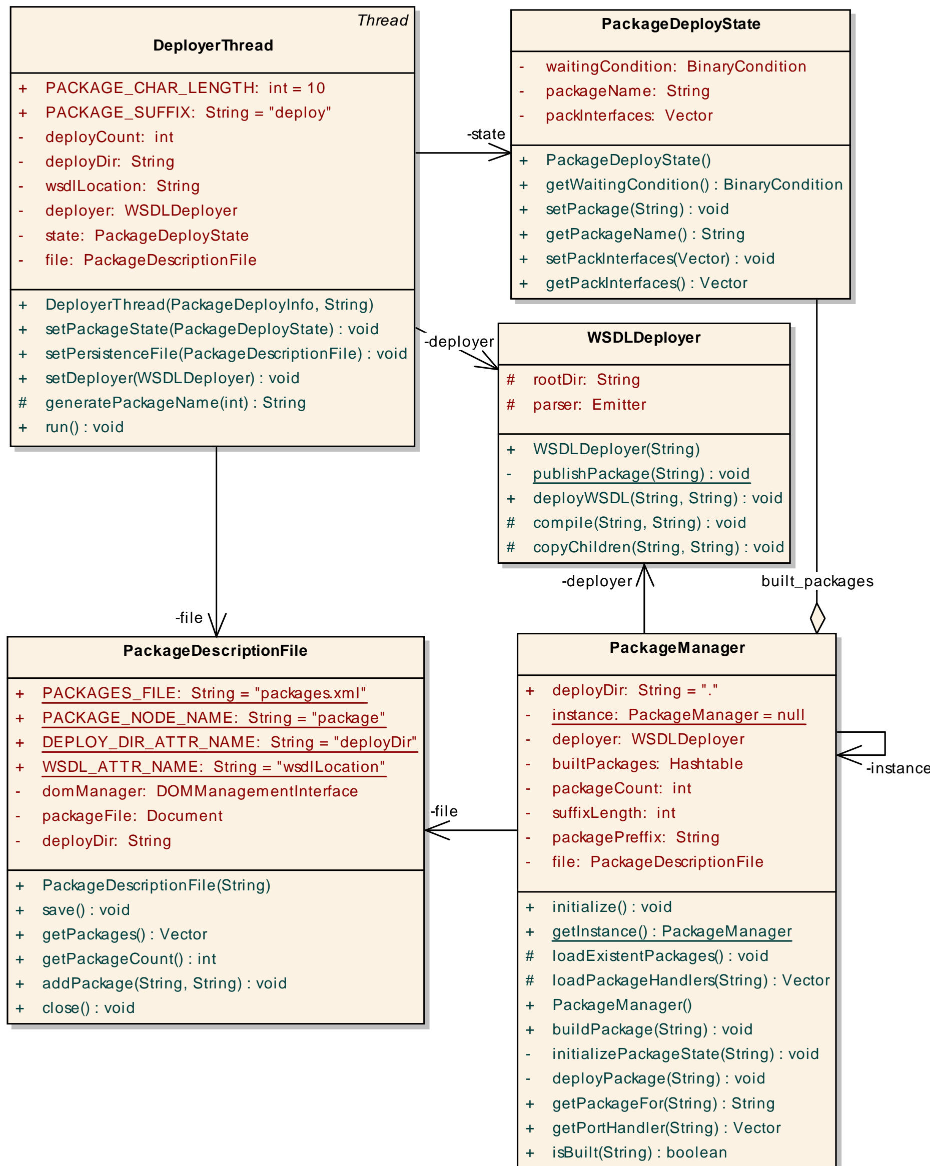


Figura 7.13: Diseño del componente *Package Manager*

La materialización del componente *Package Manager* dio como resultado algunas clases adicionales, cada una con una responsabilidad diferente. La clase PackageDescriptionFile administra la información de los archivos WSDL que han sido procesados y el directorio dentro del sistema de archivos donde reside el *stub* asociado. La clase PackageDeployState almacena información sobre la lista de clases que corresponden a puertos y *tipos de puertos* de un WSDL cuyo *deployment* ha sido efectuado. Finalmente, la clase WSDLDeployer provee el comportamiento necesario para generar y posteriormente compilar las clases JAVA correspondientes a un WSDL, en el directorio local que se le indique.

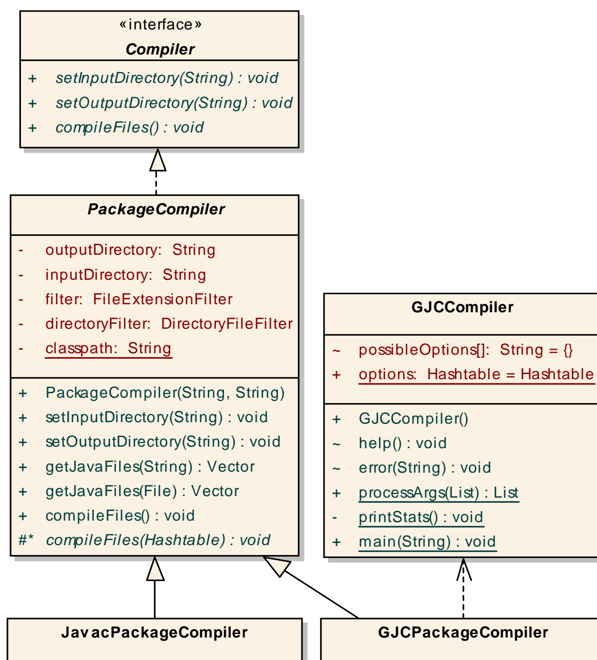


Figura 7.14: Generación de *stubs*: la clase PackageCompiler

Para realizar la compilación de los archivos JAVA obtenidos a partir de un WSDL, se derivó un pequeño diseño de clases que permite independizarse de la API o programa específico que efectúa la traducción del código fuente JAVA a formato de *bytecode*. La figura 7.14 visualiza la jerarquía mencionada, donde la clase padre de la misma es PackageCompiler. Como puede observarse, dos subclases han sido implementadas, una basada en el programa de compilación *javac* (incluido en la distribución de la máquina virtual de JAVA), y la restante basada en la biblioteca GJ[2].

7.3. Conclusiones

Para permitir la interacción de Brainlets con recursos presentes en la Web Semántica, el soporte de MRF y posteriormente de MoviLog fue extendido, dando origen a MIG y SWAM, respectivamente. En este capítulo se describió el diseño de SWAM, focalizándose en la forma en que se materializaron a clases los principales componentes de la arquitectura de dicha plataforma. Adicionalmente, fueron expuestos los detalles de implementación más significativos.

En el siguiente capítulo se describen los resultados experimentales.

[2]http://homepages.inf.ed.ac.uk/wadler/gj/

Capítulo 8

Resultados Experimentales

Las aplicaciones que fueron presentadas en el capítulo 6 fueron útiles para validar la utilidad práctica de la materialización de agentes móviles SWAM que interactúan con la Web Semántica. Particularmente, dichas aplicaciones ejemplificaron la bondades del lenguaje SWAM para el prototipado de agentes móviles que buscan e invocan Servicios Web para resolver las tareas que les han sido asignadas. Adicionalmente, pudo observarse cómo pueden ser aprovechadas las facilidades del lenguaje en cuanto a manejo de la semántica de los Servicios Web, con el fin de crear agentes móviles capaces de comprender los conceptos involucrados en cada servicio, posibilitando así una verdadera automatización de la interacción entre los agentes y los recursos Web.

En este capítulo se presentará un nuevo ejemplo de aplicación en el dominio de la compra de pasajes aéreos, similar al que se incluye en el capítulo 6. En esta oportunidad, el problema ha sido restringido a la reserva de pasajes aéreos entre dos ciudades determinadas, sin tener en cuenta itinerarios de mayor tamaño ni las consecuentes reservas en las empresas hoteleras. A grandes rasgos, la tarea a automatizar aquí es la compra de un pasaje aéreo entre dos ciudades dadas, teniendo en cuenta restricciones propias del dominio, tal es el caso de la existencia de viajes entre un origen y un destino dados o la disponibilidad de asientos, o restricciones impuestas por el usuario, como pueden ser el monto máximo a pagar por el pasaje, la preferencia de viajar en una compañía aérea por sobre las demás, entre otras.

El problema expuesto ha sido originalmente presentado en (Martínez y Lespérance, 2004b). En dicho trabajo, se han implementado cinco variantes del problema utilizando la herramienta IG-JADE-PKSlib (ver capítulo 6, sección 3.2.2), cada una considerando diferentes conjuntos de restricciones del usuario para la adquisición de un pasaje. Adicionalmente, esta implementación asume la existencia de Servicios Web para hallar un vuelo existente entre dos ciudades dadas, determinar la disponibilidad de asientos y costos asociados, y la compra del pasaje propiamente dicha.

De forma similar, las variantes del problema mencionadas anteriormente han sido implementadas en SWAM, y se han realizado experimentos comparativos en cuanto a performance y facilidad de desarrollo con respecto a la implementación mediante IG-JADE-PKSlib. Como se verá hacia finales del capítulo, los resultados arrojados por los experimentos evidencian los

graves problemas de performance que presentan los enfoques basados técnicas de planning para la composición *online* de Servicios Web.

Los contenidos del presente capítulo están organizados como se detalla a continuación. En la siguiente sección se describe el dominio de aplicación de la compra de pasajes, que ha sido el objeto de estudio para los resultados experimentales documentados en este capítulo. Luego, en las secciones 8.2 y 8.3 se presenta la implementación de la aplicación mencionada utilizando IG-JADE-PKSlib y SWAM, respectivamente, y los resultados experimentales obtenidos a partir de ambas implementaciones. Finalmente, la sección 8.4 presenta las conclusiones del capítulo.

8.1. Descripción del dominio

Los experimentos que se expondrán en este capítulo han sido llevados a cabo a partir de la implementación de una aplicación en el dominio de la compra de pasajes aéreos, la cual utiliza tecnología de Servicios Web. El problema consiste en componer ciertos Servicios Web existentes para la adquisición de un pasaje en un vuelo entre dos ciudades, de acuerdo a diversas restricciones impuestas por el usuario. Como entrada, el usuario provee la ciudad origen y destino, y las fechas deseadas de salida y de llegada. A partir de esta información, y de preferencias tales como el precio máximo que el usuario desea pagar por un pasaje, se debe encontrar una composición válida de los Servicios Web en cuestión para adquirir un pasaje satisfaciendo al mismo tiempo las restricciones indicadas por el usuario.

Se asume la disponibilidad de cuatro Servicios Web básicos: (i) *findRFlight*, que chequea la existencia de un vuelo dado en una compañía determinada, (ii) *checkFSpace*, que determina si hay asientos disponibles en un vuelo, (iii) *checkFCost*, que retorna el costo de un vuelo, y (iv) *bookFlight*, que adquiere un pasaje para un vuelo determinado. Adicionalmente, se asume la existencia de un conjunto fijo de compañías aéreas que es utilizado para la resolución del problema. En un futuro, podría adicionarse un nuevo Servicio Web encargado de informar el conjunto de compañías aéreas existentes.

Los tres primeros Servicios Web se conocen como servicios de "información", mientras que el último corresponde a un servicio de "alteración del mundo". Para el caso (i), se acepta como entrada la información del vuelo y la compañía aérea, y se *informa* el identificador del vuelo encontrado, o un valor discernible si el vuelo solicitado es inexistente. Similarmente, los casos (ii) y (iii) toman como entrada el identificador de vuelo y retornan la disponibilidad de asientos y el costo asociado al mismo, respectivamente. Por otra parte, el servicio (iv) produce cambios en el mundo o ambiente, como por ejemplo la actualización de los asientos disponibles para el vuelo en cuestión, la emisión electrónica del *ticket* correspondiente y la modificación del saldo de la tarjeta de crédito del cliente.

Por cuestiones de simplicidad, las restricciones que el usuario puede manifestar a la hora de adquirir un pasaje aéreo han sido restringidas y agrupadas en cinco conjuntos diferentes de preferencias. Cada uno de éstos conjuntos se ha generado una variante particular del problema a resolver, como se lista a continuación:

- *BPF (Book Preferred Flight)*: Este problema involucra una restricción simple: el usuario tiene una compañía aérea preferida para viajar. El objetivo entonces es adquirir un

pasaje en un vuelo en la compañía preferida; de no ser posible, comprar un pasaje en cualquiera de las demás compañías.

- *BMxF (Book Maximum Flight)*: En esta variante, el usuario manifiesta un precio máximo a pagar por el pasaje. El objetivo es encontrar un vuelo cuyo costo sea menor o igual al presupuesto indicado por el usuario.
- *BPMxF (Book Preferred Maximum Flight)*: Este problema considera dos restricciones diferentes al mismo tiempo, esto es, el usuario indica una compañía preferida para viajar, como así también un precio máximo a pagar por el pasaje. La restricción más importante es no exceder el presupuesto máximo indicado por el usuario. Adicionalmente, el pasaje debe ser adquirido a través de la compañía preferida del usuario.
- *BBF (Book Best Flight)*: La variante BBF del problema propone una tarea de optimización, que consiste en adquirir un pasaje en el vuelo de menor costo.
- *BBPF (Book Best Preferred Flight)*: Este problema es una versión refinada del anterior. El objetivo general consiste en adquirir el vuelo menos costoso de todos, pero si dos vuelos tienen el mismo precio, se debe optar por el ofrecido por la compañía preferida del usuario.

Las variante expuestas resultan interesantes debido a que abarcan propiedades muy generales encontradas en los problemas actuales de composición de Servicios Web. En particular, representan preferencias típicas de los usuarios, ya sea mediante restricciones fuertes tales como presupuesto limitado, o de optimización. En las siguientes dos secciones se describe la implementación de las variantes mencionadas utilizando IG-JADE-PKSlib y SWAM.

8.2. Implementación utilizando IG-JADE-PKSlib

IG-JADE-PKSlib es un lenguaje de programación de agentes deliberativos, que ataca el problema de la composición de Servicios Web mediante la utilización de técnicas de planning capaces de operar con conocimiento incompleto. En particular, el soporte de planning de IG-JADE-PKSlib se basa en PKS (Planning with Knowledge and Sensing) (Petrick y Bacchus, 2002; Petrick y Bacchus, 2003), un algoritmo de planning basado en conocimiento incompleto.

En PKS, el conocimiento del agente se almacena en cuatro bases de datos diferentes. Por un lado, K_f contiene hechos positivos y negativos conocidos por el agente, como así también fórmulas que especifican el valor de una función de argumentos instanciados. La base K_w almacena las fórmulas cuyo valor de verdad es conocido por el agente. A su vez, K_v puede almacenar términos de función cuyos valores son conocidos por el agente en tiempo de ejecución. Finalmente, la base K_x se compone de fórmulas formadas por la disyunción exclusiva de un conjunto de fórmulas. Para más detalles sobre el contenido de cada base de conocimiento, remitirse al capítulo 3.

Las acciones en IG-JADE-PKSlib son especificadas en base a tres componentes principales: los parámetros, precondiciones y efectos. Los parámetros son valores que permiten la ejecución de un plan existente de acuerdo a determinados argumentos de instanciación. Las precondiciones, por su parte, son fórmulas lógicas construidas en base a fórmulas incluídas en

Acción	Precondición	Efectos
findRFlight(x)	$K(airCo(x))$ $\neg K_w(flightExists(x))$ $K(desFindRFlight(x))$	$add(K_w, flightExists(x))$ $add(K_f, \neg desFindRFlight(x))$
checkFSpace(x)	$K(airCo(x))$ $\neg K_w(availFlight(x))$ $K_v(flightNum(x))$ $K(desCheckFSpace(x))$	$add(K_w, availFlight(x))$ $add(K_f, \neg desCheckFSpace(x))$
checkFCost(x)	$K(airCo(x))$ $\neg K_v(flightCost(x))$ $K(flightExists(x))$ $K(desCheckFCost(x))$	$add(K_v, flightCost(x))$ $add(K_f, \neg desCheckFCost(x))$
bookFlight(x)	$K(airCo(x))$ $\neg K(bookedFlight(x))$ $K(availFlight(x))$ $K(desBookFlight(x))$	$add(K_f, bookedFlight(x))$ $del(K_f, availFlight(x))$ $add(K_f, \neg desBookFlight(x))$
DSUR (Domain Specific Update Rules)		
$K(airCo(x)) \wedge \neg K_v(flightNum(x)) \wedge K(flightExists(x)) \Rightarrow$ $add(K_v, flightNum(x))$ (1) $K(airCo(x)) \wedge \neg K(\neg availFlight(x)) \wedge K(\neg flightExists(x)) \Rightarrow$ $add(K_f, \neg availFlight(x))$ (2) $K(airCo(x)) \wedge \neg K_w(desCheckFCost(x)) \wedge K(availFlight(x)) \Rightarrow$ $add(K_f, desCheckFCost(x))$ (3) $K(airCo(x)) \wedge \neg K_w(desCheckFCost(x)) \wedge K(\neg availFlight(x)) \Rightarrow$ $add(K_f, \neg desCheckFCost(x))$ (4)		

Tabla 8.1: Especificación de las acciones PKS para el dominio de viajes aéreos

las diferentes bases de conocimiento del agente. Por otra parte, los efectos de una acción se modelan como actualizaciones (altas o bajas) de los términos y fórmulas almacenados en las bases de conocimiento del agente. A su vez, pueden ser especificados efectos adicionales que se desprenden del conjunto de acciones del agente a través de reglas denominadas DSUR. En otras palabras, las reglas DSUR se corresponden con invariantes en un estado dado.

Las acciones básicas incluidas por el agente para la compra de pasajes aéreos se muestra en la tabla 8.1. Como puede verse, cada una de las acciones se corresponde con uno de los Servicios Web definido en la sección anterior. Adicionalmente, la tabla muestra las precondiciones y efectos asociados a cada acción. Por ejemplo, la precondición de la acción *findRFlight* establece que el agente debe conocer alguna compañía aérea x ($K(airCo(x))$, no haber determinado aún la existencia del vuelo solicitado en x ($\neg K_w(flightExists(x))$), y por último tener como pendiente la realización de dicho chequeo ($K(desFindRFlight(x))$. Luego de ejecutar la acción, el agente habrá determinado la existencia de un vuelo en la compañía en cuestión ($add(K_w, flightExists(x))$), eliminando este chequeo de su lista de tareas pendientes ($add(K_f, \neg desFindRFlight(x))$).

En el estado inicial, el agente conoce el conjunto de compañías existentes, representadas por fórmulas del tipo *airCo(x)*, y los datos del viaje en cuestión están implícitos en su base de conocimiento. Los efectos adicionales de las acciones se representan mediante DSUR, como

muestra la tabla. En particular, las reglas (1) y (2) capturan algunos efectos de las acciones informativas acerca de los vuelos. Si el agente encuentra el vuelo solicitado, acto seguido averigua el número del mismo. Por otra parte, si el agente determina que dicho vuelo no existe, entonces concluye que el vuelo no está disponible. Las reglas (3) y (4) representan una restricción simple en el control de la búsqueda: el agente debiera chequear el precio de los vuelos para los que ha establecido la existencia de asientos disponibles. Cabe destacar que todas las variantes del problema presentadas con anterioridad se implementan mediante reglas DSUR específicas adicionadas a la definición de cada acción.

Los resultados experimentales de la implementación de las diferentes variantes del problema han sido extraídos de (Martínez y Lespérance, 2004b), y se muestran en la tabla 8.2. Los experimentos fueron realizados sobre una computadora Intel XEON 3.0 GHz con 4 Gb de RAM, sobre Linux versión 2.4.22. Las pruebas fueron realizadas ejecutando el planificador PKS cinco veces por cada problema, y tomando el tiempo promedio de ejecución en cada caso. Todos los tiempos de la tabla están expresados en segundos.

# de compañías	BPF	BMxF	BPMxF	BBF	BPBF
2	0.17	0.21	0.37	1.27	8.42
3	0.58	0.72	1.20	24.02	109.33
4	1.45	2.20	3.71	$> t_{max}$	$> t_{max}$
5	3.76	4.33	4.65	$> t_{max}$	$> t_{max}$
10	80.60	96.45	105.49	$> t_{max}$	$> t_{max}$

Tabla 8.2: Resultados para los problemas BPF, BMxF, BPMxF, BBF y BPBF de acuerdo a diferente cantidad de compañías conocidas por el agente ($t_{max} = 300$)

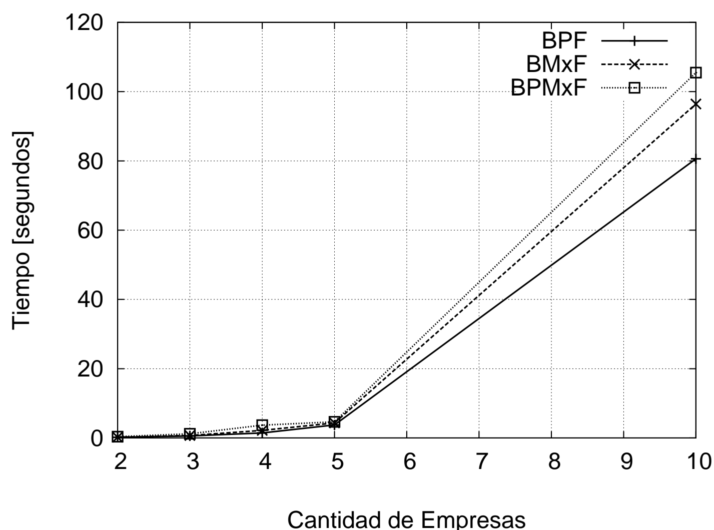


Figura 8.1: Gráfico de los resultados obtenidos para los problemas BPF, BMxF y BPMxF

En términos de performance, IG-JADE-PKSlib se comporta razonablemente bien para los problemas BPF, BMxF y BPMxF, para los que pueden generarse planes en menos de 5 segundos considerando hasta 5 compañías. Sin embargo, dichos problemas no escalan adecuadamente al considerar un número mayor de compañías, como muestra el gráfico de la figura 8.1. Notar

además que los problemas de optimización BBF y BPBF no escalan bien en ningún caso. Ambas versiones requieren un ordenamiento completo de las compañías por costo, por lo que el planificador PKS tiene que llevar a cabo una gran cantidad de razonamiento combinatorio complejo.

8.3. Implementación utilizando SWAM

En esta sección se describe la implementación de la solución al problema anterior mediante la utilización de agentes SWAM. Para ello, se codificó cada una de las variantes vistas como un agente diferente que interactúa con los Servicios Web en cuestión implementados en JAVA. Las operaciones que llevan cabo cada uno de los servicios son como siguen:

- *findRFlight(company, origin, destination, departureDate, arrivalDate)*: El servicio inspecciona una estructura de datos asociativa creada en memoria a partir de los argumentos recibidos, y retorna el identificador del vuelo correspondiente, o un valor por defecto (-1) si el vuelo no es hallado.
- *checkFSpace(company, flightID)*: Por cuestiones de simplicidad, el valor retornado por este servicio se ha simulado utilizando una función de probabilidad en función del identificador de la compañía. Para esto, las compañías son ordenadas (estáticamente) de acuerdo a la probabilidad de encontrar asientos disponibles en cualquiera de sus vuelos. Basado en este ordenamiento y en los valores de probabilidad, el servicio simula un valor probabilístico al azar. Si este valor supera el umbral definido para la compañía pasada como argumento, el servicio retorna que efectivamente existen lugares disponibles en el vuelo, de lo contrario retorna falso. En un futuro, podría reimplementarse el servicio para retornar el resultado en función de una consulta a una base de datos conteniendo datos reales.
- *checkFCost(company, flightID)*: De forma similar al caso anterior, este servicio simula el precio de un viaje basado en un costo promedio de los vuelos para cada compañía. En definitiva, el costo de un vuelo es una función lineal que depende del identificador de la compañía. Nuevamente, el parámetro *flightID* es ignorado por el servicio, pero futuras implementaciones podrían tenerlo en cuenta para recuperar el costo a partir de una base de datos.
- *bookFlight(company, flightID, userID)*: Simula la registración de la venta de un ticket para un vuelo, agregando una nueva entrada a una lista que es mantenida en memoria por parte del servicio. Como resultado, retorna el número de ticket impreso producto de la transacción simulada.

Para realizar los experimentos, los Servicios Web listados anteriormente se publicaron en un servidor SWAM, construyendo e instalando el archivo WSDL correspondiente que contiene la definición relacionada al servidor particular que los implementa, los argumentos de entrada necesarios para la invocación, y la salida que cada servicio retorna. Adicionalmente, el *protocolo* que describe cada servicio particular fue publicado en dicho servidor. Como consecuencia, la descripción asociada a cada Servicio Web es comunicada a los demás sitios SWAM, lo que permite a los agentes que ejecuten en sitios remotos encontrar los servicios e invocarlos.

El siguiente programa corresponde a la implementación SWAM del agente que resuelve el problema BPBF, que consiste en encontrar el vuelo menos costoso, seleccionando el que pertenece a la compañía preferida si dos vuelos tienen el mismo costo. Básicamente, el agente declara un protocolo que habilita el mecanismo de MIG ante la falla en la obtención de un Servicio Web (no importa el nombre ni la entrada) en el sitio local. Dicho protocolo no define una política de acceso específica, lo que se indica mediante el átomo "none". Por su parte, la sección CLAUSES incluye las cláusulas que implementan el algoritmo para la adquisición del vuelo bajo las restricciones impuestas por la variante del problema. Dicho algoritmo opera como sigue:

1. Obtiene la compañía que ofrece el viaje de menor costo.
2. Si la compañía obtenida en el paso anterior coincide con la preferida por el usuario, el agente adquiere un pasaje para el vuelo en dicha compañía. Finalizado esto, el algoritmo termina.
3. Si la compañía obtenida en el paso (1) ofrece el mismo costo que la compañía preferida, se adquiere un pasaje para ésta última, y posteriormente el algoritmo termina.
4. En otro caso, se adquiere un vuelo en la compañía que ofrece el viaje de menor costo.

Inicialmente, el agente conoce las compañías aéreas existentes a partir de las cuáles se determinará la disponibilidad o no del vuelo solicitado por el usuario. Este conocimiento es expresado mediante hechos de la forma *company(C)* que forman parte del conocimiento privado del agente móvil. Similarmente, los datos del vuelo a adquirir y la identificación del usuario solicitante son representados mediante hechos Prolog almacenados en la base de conocimiento del agente.

```
PROTOCOLS
  protocol(webService, [name(X), input(X)], none).
CLAUSES
  company('United␣Airlines').
  company('Air␣France').
  ...
  travel-info(['New␣York', 'Rome', 01/01/2004, 03/12/2004]).
  user-id(agent-owner-id).
  getCompaniesAndCost(I, CList):-
    getCompaniesAndCost(I, [], CList).
  getCompaniesAndCost(I, Temp, Result):-
    company(C),
    retract(company(C)),
    webService([name(findRFlight), input([C|I])], FlightID),
    FlightID \= -1,
    webService([name(checkFSpace), input([C|FlightID])], true),
    webService([name(checkFCost), input([C|FlightID])], Cost),
    getCompaniesAndCost(I, [cost(C,FlightID,Cost)|Temp], Result).
  getCompaniesAndCost(_, Temp, Temp).
  minCost(cost(Co1,FID,Cost1), cost(_,_,Cost2), cost(Co1,FID,Cost1)):-
    Cost1 <= Cost2, !.
  minCost(_, CostInfo2, CostInfo2).
```

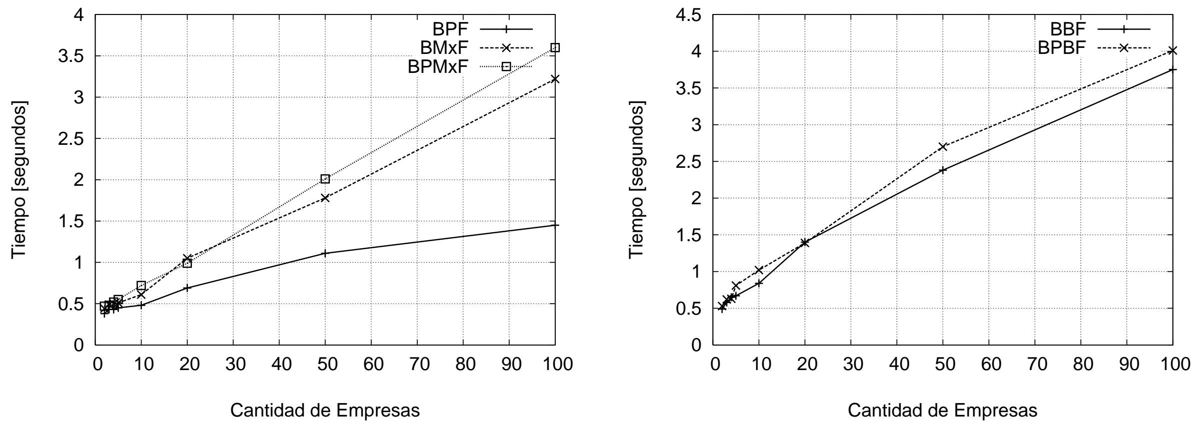
```
findCheapest([CostInfo|Tail], Ch):-
  findCheapest(Tail, CostInfo, Ch).
findCheapest([CostInfo|Tail], TempCost, Ch):-
  minCost(CostInfo, TempCost, MinCost),
  findCheapest(Tail, MinCost, Ch).
findCheapest([], MinCost, MinCost).
selectMinCostCompany(CList, Cheapest, Pref, Cheapest):-
  not(member(cost(Pref,_,_), CList)).
selectMinCostCompany(_, PrefInfo, Pref, PrefInfo):-
  PrefInfo = cost(Pref, _, _).
selectMinCostCompany(CList, Cheapest, Pref, PrefInfo):-
  Cheapest = cost(_, _, Cost),
  member(cost(Pref,FID,Cost), CList),
  PrefInfo = cost(Pref, FID, Cost).
?-arrangeTravel(Pref, TicketID):-
  travelInfo(I),
  getCompaniesAndCost(I, CList),
  findCheapest(CList, Cheapest),
  selectMinCostCompany(CList, Cheapest, Pref, Result),
  Result = cost(Co, FID, _),
  user-id(UID),
  webService([name(bookFlight), input([Co,FID,UID])], TicketID).
```

Los resultados obtenidos a partir de la experimentación sobre las diferentes variantes del problema de adquisición de pasajes se muestran en la tabla 8.3. La figura 8.2 presenta los gráficos comparativos correspondientes a los tiempos de resolución de cada problema. En esta oportunidad, las pruebas se realizaron sobre un servidor con un procesador Intel Pentium 4 de 2.2 GHz, con 1 GHz de memoria RAM, sobre Linux 2.6.8. En un servidor extra de similares características se instaló la implementación correspondiente a los Servicios Web, provista a través de un servidor Apache Tomcat 4.1 corriendo sobre la máquina virtual de JAVA versión 1.4.2 (build 05).

# de compañías	BPF	BMxF	BPMxF	BBF	BPBF
2	0.38	0.44	0.47	0.49	0.53
3	0.43	0.49	0.48	0.58	0.62
4	0.43	0.50	0.52	0.65	0.63
5	0.45	0.51	0.55	0.67	0.81
10	0.48	0.61	0.72	0.84	1.02
20	0.69	1.05	0.99	1.40	1.39
50	1.11	1.78	2.01	2.38	2.70
100	1.45	3.22	3.60	3.75	4.01

Tabla 8.3: Performance de los agentes SWAM para los problemas BPF, BMxF, BPMxF, BBF y BPBF (segundos)

Para obtener los tiempos que figuran en la tabla, se inició el agente encargado de resolver cada problema veinte veces, y se tomó el tiempo promedio de ejecución en cada caso. Como puede observarse, la ejecución de cada agente muestra una muy buena performance. A su vez, puede apreciarse que las soluciones escalan de manera adecuada, incluso para una gran cantidad de compañías aéreas. Como era de esperar, los mayores tiempos promedio se registraron para las



(a) Performance para los problemas BPF, BMxF y BPMxF

(b) Performance para los problemas BBF y BPBF

Figura 8.2: Gráficos de los tiempos obtenidos para la implementación SWAM

variantes BBF y BPBF, que son los problemas de optimización que llevan a cabo la mayor cantidad de procesamiento. En este caso, los agentes asociados deben determinar cuál es la compañía aérea que ofrece el vuelo de menor costo, por lo que se ven obligados a tener en cuenta la totalidad de las compañías conocidas. Por otra parte, si se conoce de antemano el orden (creciente) de las compañías relativo al costo de los vuelos ofrecidos, se obtiene una significativa mejora en la ejecución del agente, como se ejemplifica en la figura 8.3 para el caso del problema BBF.

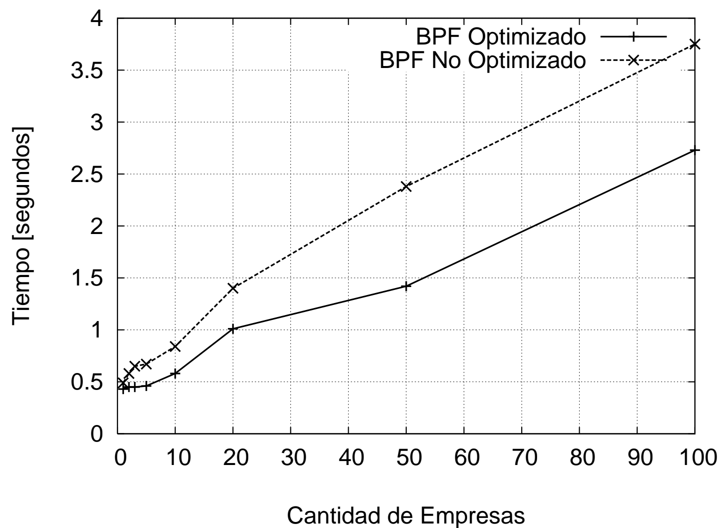


Figura 8.3: Mejora obtenida para el problema BBF ordenando las compañías en orden creciente según el costo de los vuelos

La tabla 8.4 muestra la mejora en segundos y el porcentaje que esta mejora representa en los tiempos de ejecución de la tabla 8.3, para el problema BBF. Como puede observarse, la nueva solución ofrece una mejora que oscila aproximadamente entre el 12 y el 40 por ciento. Esto

# de Co	Mejora (seg)	Mejora (%)
2	0.06	12.24
3	0.13	22.41
4	0.20	30.77
5	0.21	31.34
10	0.26	30.95
20	0.39	27.86
50	0.96	40.34
100	1.02	27.20

Tabla 8.4: Mejoras obtenidas (en segundos y porcentaje del tiempo total) para el problema BBF

se debe mayormente a que conocer el orden de las compañías de acuerdo al costo implica la invocación de una menor cantidad de Servicios Web para determinar la existencia de vuelos, asientos disponibles y costos, debido a que muchas de ellas no son tenidas en cuenta para la resolución del problema. En general, la mejora obtenida cuando se ejecuta al agente con una cantidad escasa de compañías es muy inferior a la registrada con un número mayor de ellas, fundamentalmente porque la cantidad de invocaciones a Servicios Web que se evitan es mucho menor.

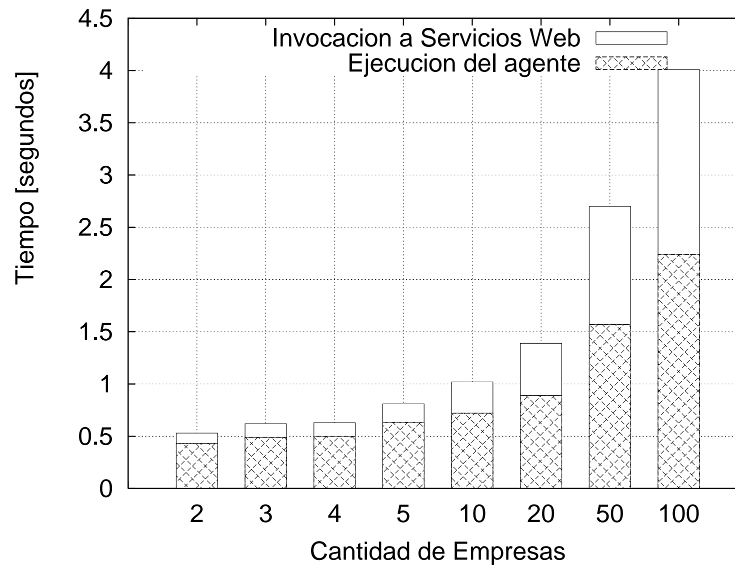


Figura 8.4: Tiempo empleado en la invocación de Servicios Web para el problema BPBF

Un detalle que puede resultarle curioso al lector es la diferencia significativa en el porcentaje de mejora para ciertos casos puntuales. Por ejemplo, cuando el agente utiliza una base de 50 compañías, el porcentaje de mejora es alrededor del 40 %, mientras que con una base de 100 compañías (el doble del valor anterior) la mejora es apenas del 27 %. La explicación a este fenómeno se relaciona con la implementación del servicio *checkFSpace*. Al estar implementado mediante una función probabilística, pueden darse casos de ejecuciones desafortunadas en las cuáles el agente intenta en varias compañías antes de lograr ubicar un asiento disponible. Así,

la cantidad de compañías procesadas y el número de Servicios Web invocados a tal efecto aumenta, degradando la performance del agente. Cabe destacar que, con una implementación más "realista" del servicio *checkFSpace*, las situaciones mencionadas no ocurrirían.

Un aspecto que merece particular atención es el tiempo promedio que emplea cada agente en la ejecución de los Servicios Web a lo largo de su tiempo de vida. En la figura 8.4 se muestra un gráfico de barras que visualiza el tiempo total de ejecución del agente discriminado según sea tiempo empleado para invocación a Servicios Web o no, para el problema BPBF. Para un número de compañías menor o igual a 10, el porcentaje del tiempo total utilizado en invocación a servicios va del 20 al 25 %, lo que significa que el agente emplea alrededor de la cuarta parte del tiempo en interactuar con Servicios Web. Sin embargo, para un número mayor de compañías, este porcentaje crece significativamente, llegando a valores del 45 % para 100 compañías aéreas. Estos resultados no son sorpresivos, ya que el agente permanentemente basa sus decisiones en información obtenida a partir de la interacción con Servicios Web. Sin embargo, estos resultados han motivado algunas tareas de eficientización del soporte provisto por SWAM para invocación de Servicios Web.

En primer lugar, se cambió la biblioteca que implementa la invocación de servicios a bajo nivel vía SOAP por la biblioteca Axis (Apache Software Foundation, 2004). Axis es una implementación del protocolo SOAP provista por Apache que utiliza un parser eficiente de datos XML denominado SAX (Simple API for XML), en vez del costoso parser DOM (Document Object Model) utilizado por otras implementaciones. En general, esta modificación redujo notablemente los tiempos empleados en la invocación a cada servicio. Sin embargo, los componentes encargados de la traducción de tipos datos Prolog a tipos de datos SOAP y viceversa no están aún optimizados, tarea que introducirá nuevas mejoras en la performance de SWAM en cuanto al llamado a Servicios Web.

8.4. Conclusiones

La composición de Servicios Web (WSC) puede verse como el problema de componer o agrupar servicios de funcionalidad relativamente simple para producir nuevos servicios arbitrariamente complejos. En general, se habla de dos estrategias diferentes de composición de servicios, denominadas *proactiva* y *reactiva*. Por un lado, la estrategia proactiva se basa en la composición *offline* de los servicios disponibles. Esta estrategia es utilizada generalmente cuando los servicios son estables (su estructura no cambia significativamente con el tiempo) y se encuentran siempre en ejecución, como puede ser, por ejemplo, un servicio de adquisición de tickets similar al incluido en el ejemplo discutido en este capítulo. Por otra parte, la estrategia reactiva apunta a crear dinámicamente servicios compuestos. En este caso, la estrategia es aplicable mayormente cuando el servicio resultado no es usado a menudo, y a su vez cuando los servicios simples que lo conforman no son estables. Este es el caso, por ejemplo, de un servicio para realizar las reservas hoteleras y de movilidad asociadas a un paquete turístico cuando el itinerario de éste último no es fijo.

El problema de generar un servicio complejo basado en servicios más simples está actualmente siendo estudiado fuertemente por la Inteligencia Artificial (AI). La composición de servicios ha sido asociada en el área de AI a un problema de *planning*, donde el servicio compuesto está representado por el objetivo a alcanzar, y las tareas que conforman el plan final se

corresponden con los servicios simples. En este sentido, varias son las técnicas de planning que han sido propuestas para la composición de Servicios Web:

- *Cálculo de Situación* (McCarthy y Hayes, 1981): Esta técnica asume que todos los cambios dinámicos en el mundo son producto de la ejecución de alguna acción. Una situación se define como una historia del mundo, es decir, una secuencia de acciones llevadas a cabo. El estado del mundo está descrito mediante funciones y relaciones (llamados fluentes) asociados a una situación *s*. La constante s_0 describe la situación inicial, es decir, donde no se ha aplicado aún ninguna acción. Por ejemplo, el estado $do(putDown(A), do(walk(L), do(pickUp(A), s_0)))$ describe la situación que surge a partir de la ejecución de la secuencia de acciones [*pickUp(A),walk(L),putDown(A)*].

  ConGolog (Giacomo et al., 2000) es un lenguaje de programación lógica de alto nivel para la especificación y ejecución de acciones complejas en dominios dinámicos. ConGolog está construido por encima del Cálculo de Situación y ha sido propuesto para su aplicación en el área de Servicios Web (McIlraith y Son, 2001).

- *Hierarchical Task Networks (HTNs)* (Nau et al., 2001): HTN es un método de planning basado en descomposición de tareas. Contrario a otras técnicas de planning, el concepto central de HTN no son los estados, sino las tareas. Un sistema de planning HTN descompone una tarea objetivo en un conjunto de subtareas, y cada una de éstas en nuevas subtareas, y así sucesivamente hasta que el conjunto de tareas actual sean tareas primitivas que puedan ser ejecutadas mediante la invocación de operaciones atómicas. Cada vez que se realiza la descomposición de una tarea, es necesario chequear que ciertas condiciones no sean violadas (por ejemplo, exceder la utilización de determinados recursos). HTN ha sido propuesto para su utilización en la composición de Servicios Web como se detalla en (Wu et al., 2003).

- Graphplan, introducido primeramente en (Blum y Furst, 1995), es un algoritmo de planning que ofrece una buena performance. Aquí, el problema de planning se modela como un grafo consistente de nodos de tipo *acción* y *condición*. Estos nodos son agrupados en diferentes niveles, que incluyen un nivel de nodos de acción seguido de un nivel de nodos de condición, y así siguiendo. Los arcos del grafo conectan nodos de acciones y condiciones basados en las precondiciones y efectos de cada acción. Una alternativa para la utilización de Graphplan para la composición de Servicios Web se presenta en (Chatterjee y Mitra, 2002).

En general, la aplicación de las técnicas mencionadas en el contexto de la Web Semántica está sujeta a que los servicios disponibles estén descritos en forma de acciones aplicables en determinados estados. Las transiciones se definen en base a las precondiciones y postcondiciones de las acciones que representan el comportamiento de un Servicio Web. Ejecutar una acción (servicio) dispara una transición que lleva a un nuevo estado donde los efectos definidos para la acción son válidos. Sin embargo, a pesar de las ventajas que las técnicas de planning de AI ofrecen para la creación de planes para satisfacer objetivos complejos, existen algunos obstáculos para la aplicación de dichas técnicas al problema de WSC (Srivastava y Koehler, 2004):

- La especificación de las acciones tradicionalmente consistieron de precondiciones y efectos. Sin embargo, un Servicio Web modelado como una acción puede tener estructuras

de control complejas en su especificación, consistiendo por ejemplo de ciclos y elecciones no determinísticas.

- Los objetos manipulados por los Servicios Web (acciones) son mensajes compuestos de datos que pueden tener estructuras arbitrariamente complejas. Tradicionalmente, los métodos de planning han trabajado con tipos de datos simples.

- Los procesos que utilizan Servicios Web pueden crear nuevos objetos en tiempo de ejecución. En planning, se asume que todos los objetos que intervienen en la planificación son conocidos e identificables en el estado inicial del algoritmo.

- Las técnicas de planning han sido fundamentalmente diseñadas para la creación de planes en forma *offline*, por lo que, en general, su aplicación en forma *online* produce soluciones que insumen un tiempo de construcción muy grande, y excesivo en algunos casos. Por otra parte, no se han estudiado aún algoritmos adecuados basados en planning para la creación y ejecución de planes en forma intercalada para WSC.

Con el fin de comprobar empíricamente algunas de las afirmaciones anteriores, en este capítulo se han expuesto algunos experimentos realizados en torno a una aplicación simple en el dominio de la compra de pasajes aéreos. Dicha aplicación ha sido codificada mediante las facilidades de inferencia lógica provistas por SWAM. Adicionalmente, se ha discutido la implementación de la aplicación anterior mediante la herramienta IG-JADE-PKSlib, la cual permite la composición de servicios basada en un algoritmo de planning a nivel de conocimiento denominado PKS.

Los experimentos realizados arrojaron una diferencia notoria de la solución ofrecida por SWAM por sobre la primera implementación expuesta. En particular, la implementación de la aplicación mediante agentes SWAM registró una performance y escalabilidad muy superior a la solución codificada mediante IG-JADE-PKSlib. La escasa performance y escalabilidad demostrada por IG-JADE-PKSlib en la resolución de problemas clásicos de WSC con diversas restricciones limita su uso a aplicaciones donde se acepta una composición de tipo proactiva. En contrapartida, los resultados obtenidos en la experimentación a partir del uso de SWAM demuestra que el problema de WSC puede ser efectivamente atacado utilizando un enfoque basado en lógica. En otras palabras, los experimentos muestran que dicho enfoque resulta adecuado para composición reactiva de Servicios Web, solucionando una parte de los problemas que presentan las técnicas de planning al ser aplicadas en este contexto.

Capítulo 9

Conclusiones y Trabajos Futuros

En los capítulos previos se describió un lenguaje denominado SWAM (Semantic Web-Aware MoviLog) que posibilita la programación de agentes móviles capaces de interactuar automatizadamente con Servicios Web Semánticos (Mateos et al., 2005b; Mateos et al., 2005a). De esta manera, SWAM permite desarrollar aplicaciones en el contexto de la promisoria Web Semántica, aprovechando al mismo tiempo las ventajas que poseen los agentes móviles para la construcción de aplicaciones masivamente distribuidas.

SWAM está basado en MoviLog, lenguaje que provee un modelo de movilidad llamado Movilidad Reactiva por Fallas (MRF) cuyo objetivo es reducir el esfuerzo de desarrollo de agentes móviles. En esencia, MRF posibilita al programador indicar qué solicitudes de recursos de un agente móvil podrían generar la migración del mismo. Las decisiones de movilidad sobre cuándo y dónde migrar un agente cuando la obtención de uno de esos recursos falla son tomadas por los mecanismos de soporte de ejecución de MRF. No obstante, SWAM incluye un modelo de ejecución de agentes que generaliza y extiende el modelo conceptual de MRF, denominado MIG (Movilidad Inteligente Generalizada). MIG soporta, además de movilidad de código, movilidad e invocación remota de recursos. Asimismo, MIG ofrece un soporte que el programador puede utilizar para especificar decisiones que permiten personalizar, entre otros aspectos, el método específico a aplicar para interactuar con un recurso dado y el sitio fuente a partir del cual se obtiene el mismo.

Para validar experimentalmente SWAM se realizaron algunas aplicaciones y comparaciones con otros enfoques para la implementación de agentes en la Web Semántica. De las experiencias surge que SWAM, además de reducir el esfuerzo de programación de las aplicaciones, posee una mejor performance en términos de tiempo de ejecución con respecto a enfoques basados en técnicas de planning, debido a la gran complejidad computacional inherente a éstas últimas.

El presente capítulo extrae algunas conclusiones a partir de todo el material que ha sido presentado a lo largo de esta tesis. La siguiente sección presenta las principales contribuciones aportadas por este trabajo. Luego, la sección 9.2 discute las limitaciones de los enfoques propuestos. Finalmente, en las secciones 9.3 y 9.4 se presentan posibles trabajos futuros y las consideraciones finales, respectivamente.

9.1. Contribuciones

Esta tesis aporta varias contribuciones al área de agentes móviles, a saber:

- Introduce el lenguaje SWAM, que posibilita la implementación de aplicaciones en la Web capaces de interactuar y utilizar los recursos en forma autónoma. Esta característica resulta sumamente útil para facilitar el desarrollo de aplicaciones que aprovechan el contenido de la Web actual sin la intervención de los usuarios.
- Proporciona el modelo de ejecución MIG, el cual generaliza el modelo conceptual de MRF, combinando los beneficios de la migración de código, en lo referido a la facilidad y transparencia para el programador, con las ventajas de la copia e invocación remota de recursos, lo que permite aprovechar mejor los recursos de una red. Como complemento, SWAM ofrece mecanismos sintácticos flexibles para automatizar decisiones relacionadas con el acceso a los recursos que permiten adaptar los agentes a las necesidades particulares de cada aplicación.
- Define una arquitectura que permite la publicación y búsqueda de Servicios Web, tanto en forma local como en la Internet. Adicionalmente, la arquitectura provee funcionalidad para almacenar ontologías, registrar las descripciones semánticas correspondientes a cada servicio, y determinar la similitud semántica entre dos Servicios Web. Las ontologías pueden ser escritas en cualquiera de los lenguajes de descripción de contenido semántico más difundidos (RDF u OWL por ejemplo) ya que existen componentes de traducción encargados de representar dicho contenido en un formato Prolog estándar.

9.2. Limitaciones

Una limitación de SWAM lo constituye la sintaxis utilizada para representar los protocolos de un agente móvil. La declaración de un protocolo, como se observó en capítulos previos, obedece al formato *protocol(resourceKind,propertyList,accessPolicy)*, donde *resourceKind* es un átomo que representa la categoría particular de recursos descrita por el protocolo, *propertyList* es la lista de propiedades que identifica las instancias específicas de éstos, y *accessPolicy* referencia a la política empleada por el agente para efectivizar el acceso a tales recursos. Claramente, el hecho de basar la especificación de protocolos en estructuras Prolog permite aprovechar las ventajas que ofrece dicho lenguaje en cuanto a programación declarativa se refiere. Sin embargo, es preciso notar que los elementos de la lista *propertyList*, por tratarse de propiedades potencialmente complejas (argumentos de entrada de un Servicio Web, por ejemplo), pueden presentar una estructura recursiva arbitrariamente anidada. Ciertamente, este problema dificulta en gran medida la legibilidad de los programas implementados en SWAM.

Con el fin de administrar adecuadamente la semántica de los Servicios Web, MIG prescribe una arquitectura que incluye funcionalidad para búsqueda y publicación de servicios, como así también funcionalidad para determinar la similitud semántica entre dos o más Servicios Web. Esta última tarea es responsabilidad de un componente denominado *Matcher*, el cual implementa un algoritmo que establece el grado de correspondencia semántica entre dos conceptos cualesquiera a partir de una base de ontologías local. Lamentablemente, esta base de

ontologías no es capaz de manejar conflictos producto de diferentes definiciones para un mismo concepto; por ejemplo, un nuevo servicio es descubierto, pero la definición de un concepto asociado a su semántica contradice una definición previamente almacenada en la base local.

Si bien SWAM ha mejorado sustancialmente como lenguaje con respecto a su predecesor, principalmente en cuanto a la flexibilidad otorgada al programador para la especificación de decisiones para el acceso a los recursos, aún cuenta con ciertos aspectos que sería interesante mejorar. Uno de estos aspectos es definir claramente la secuencia de acciones que se realizan desde que se produce la falla a nivel MIG hasta que el recurso en cuestión es recuperado. Por ejemplo, la falla en la obtención de un Servicio Web por parte de un agente tiene como contrapartida la activación de MIG, que busca un servicio semánticamente similar al solicitado por el agente, y posteriormente lo ejecuta. Sin embargo, ante la existencia de servicios con idéntico grado de similitud, el agente no cuenta con la posibilidad de seleccionar qué servicio será finalmente invocado, cuando quizás existe un mayor nivel de preferencia por un proveedor u otro. En este sentido, es necesario que la plataforma delegue al programador la decisión de cuándo hacer uso de un recurso luego de que MIG lo ha obtenido o instanciado.

A continuación se describen los trabajos futuros.

9.3. Trabajos futuros

SWAM es un lenguaje que apunta a convertirse en una herramienta que permita la completa automatización de las aplicaciones Web a través de los agentes móviles. Sin embargo, para materializar esta visión, existen algunos aspectos que serán motivo de investigación futura. En las siguientes subsecciones se exponen algunos de ellos.

9.3.1. Soporte para la construcción de procesos o *workflows* de Servicios Web

A medida que las interacciones entre aplicaciones en la Web Semántica aumentan en complejidad, la necesidad de automatizar los procesos de negocio se hace cada vez más evidente. Los procesos de negocio constituyen especificaciones no ambiguas acerca de la secuencia de tareas que deben llevar a cabo dos partes cualesquiera durante una transacción. En la actualidad, muchas compañías publican los servicios ofrecidos a través de Servicios Web, y al mismo tiempo definen varios procesos estándares que contienen la lógica concerniente a la forma en que un cliente debe interactuar con tales servicios con el fin de completar una transacción determinada.

Una tecnología útil para materializar el comportamiento anterior son los *workflows*. Los *workflows* son sistemas de software capaces de administrar y monitorear un proceso de negocio, compuestos de *tareas* que realizan un conjunto específico de acciones, y *vínculos*, que definen los flujos permitidos entre éstas de acuerdo a ciertas condiciones preestablecidas (Workflow Management Coalition, 1999). En el contexto de la Web Semántica, se han producido algunos avances de importancia, mayormente relacionados con la creación de lenguajes que extienden XML para la especificación de procesos basados en Servicios Web, tales como BPEL4WS. Sin embargo, como se observó en capítulos anteriores, dichos lenguajes no poseen las características requeridas para una efectiva automatización de las aplicaciones Web.

Una línea de investigación interesante que apunta a resolver tal limitación es la incorporación de mecanismos para la definición y ejecución de *workflows* dentro del lenguaje SWAM. De esta manera, las ventajas que poseen los agentes móviles SWAM en términos de manipulación de Servicios Web Semánticos resultarán beneficiosas para la especificación de *workflows* cuyas tareas referencian servicios en términos de la *funcionalidad* requerida, y no en base URLs específicos. Una investigación preliminar que está siendo realizada está relacionada con la creación de un motor de *workflows* simple basado en agentes móviles SWAM; un paso posterior consistirá en la integración de dicho motor con funcionalidad para la ejecución de *workflows* que interactúan con Servicios Web Semánticos.

9.3.2. Utilización de ASP para expresar y razonar a partir de las ontologías

Una paradigma que ha sido extensamente utilizado para realizar inferencias sobre bases de conocimiento es la *programación lógica*. La programación lógica consiste en expresar conocimiento a través de reglas y realizar inferencias a partir de éstas. Una ventaja importante de este paradigma es que es capaz de tratar con conocimiento negativo, y expresar suposiciones de *mundo cerrado*, en donde lo que no puede ser inferido a partir de un conjunto de reglas conocido se supone falso. Sin embargo, este paradigma no es completamente apropiado como un formalismo general para la representación de conocimiento en la Web Semántica, debido a que la lógica clásica no permite crear modelos a partir de teorías inconsistentes (Alferes et al., 2003). La habilidad de manejar información contradictoria y/o incompleta es una característica particularmente deseable para los sistemas de inferencias que operan sobre ontologías, dado que es muy natural obtener definiciones diferentes para un mismo concepto desde varios sitios fuente.

Un formalismo bien establecido que resulta apropiado para la manipulación de conocimiento incompleto y/o contradictorio es ASP (Answer Set Programming) (Gelfond y Lifschitz, 1988). ASP es un paradigma declarativo para la representación y razonamiento a partir de conocimiento. Para realizar inferencias, el programador diseña un programa lógico cuyos "modelos estables" constituyen las soluciones al problema. Las ventajas más significativas del paradigma son su simplicidad, su habilidad para modelar efectivamente conocimiento incompleto y restricciones de clausura, y su utilidad para satisfacer problemas de restricciones.

La mayor parte de los lenguajes que han sido actualmente propuestos para representar ontologías en el contexto de la Web Semántica (DAML+OIL u OWL, por ejemplo) constan de un modelo formal basado en una Lógica de Descripción (LD) (Baader et al., 2003). Las Lógicas de Descripción son una familia de lenguajes de representación de conocimiento que ha sido extensamente estudiada en el área de Inteligencia Artificial durante las últimas dos décadas, y constan de clases de lógicas que poseen propiedades interesantes tales como completitud computacional y decidibilidad. Estas propiedades son particularmente interesantes para la búsqueda de información en la Web Semántica a partir de un motor que realiza inferencias sobre ontologías, ya que se asegura una respuesta para cada solicitud procesada en una cantidad finita de tiempo.

En definitiva, la utilización de ASP o un formalismo similar constituye una alternativa viable para tratar con conocimiento inconsistente en SWAM. Así, ASP podría adoptarse para expresar los conceptos conocidos por un sitio, y de esta manera determinar las similitudes entre ellos a partir de un intérprete especial tal como smodels (Niemelä et al., 2000) o DLV (Leone

et al., 2002). Adicionalmente, esto permitirá dar solución a las limitaciones del algoritmo de *matching* de conceptos mencionadas en la sección anterior.

9.3.3. Escalabilidad del registro de Servicios Web

Un aspecto que merece especial atención y que será objeto de investigación futura se relaciona con la escalabilidad del *Registro de Descripciones Semánticas*. Básicamente, SWAM materializa este componente en forma centralizada, almacenando las relaciones existentes entre las descripciones WSDL de cada Servicio Web publicado con los conceptos ontológicos involucrados en éstos. Notar que, sin embargo, la utilización de esta infraestructura en el contexto de la WWW resultará en un registro excesivamente grande que condiciona en forma inaceptable los tiempos de respuesta producto de determinar la similitud semántica entre servicios. Un problema similar ocurre con el *Repositorio de Ontologías*, ya que debe almacenar la estructura de cada uno de los conceptos asociados a *cada* servicio publicado.

Intuitivamente, una forma eficaz de solucionar la situación anterior es distribuir el contenido de ambos repositorios entre varios sitios, y utilizar un algoritmo de similitud semántica capaz de operar en función de la información distribuida. La distribución podría ser o bien arbitraria o en base a alguna disposición determinada (por ejemplo, balanceando el tamaño de cada repositorio o agrupando los servicios de ciertos proveedores en un sitio). Una solución adecuada a este problema lo constituye la tecnología de P2P Computing (Oram, 2001). P2P Computing ofrece un modelo apto para resolver problemas computacionalmente complejos haciendo uso de recursos de hardware y software localizados en nodos dispersos a través de una red. En este sentido, se estudiarán las alternativas que ofrece el área de P2P Computing para soportar el requerimiento de escalabilidad discutido.

9.3.4. Persistencia de protocolos

Uno de los problemas que presenta la implementación actual de la plataforma de SWAM radica en la falta de un registro persistente de los protocolos ofrecidos por un sitio. A grandes rasgos, el *deployment* de un nuevo recurso en un sitio involucra dos etapas básicas: por un lado, existe un *deployment lógico*, en el cual se adiciona el protocolo que describe el recurso al servidor local, el cual lo publica al resto de los sitios, y un *deployment físico*, donde los componentes físicos asociados al recurso (archivos, bibliotecas, cláusulas, etc.) son copiados e instalados localmente. Por ejemplo, el *deployment* de un Servicio Web involucra la publicación local del protocolo asociado, que describe aspectos del servicio tales como la entrada requerida y la salida retornada, más las actualizaciones físicas, dadas por la instalación del código y bibliotecas necesarias para ejecutar el servicio, la copia del documento WSDL que lo describe, y el registro del servicio en la configuración del servidor SOAP local.

Actualmente, SWAM no provee un mecanismo para hacer persistente la información relacionada al *deployment* lógico de un recurso; en consecuencia, el *shutdown* de un sitio ocasiona la pérdida de los protocolos ofrecidos. Notar que, sin embargo, los cambios correspondientes al *deployment* físico seguirán vigentes. Asimismo, el mecanismo de persistencia debe ser diseñado de modo que permita las altas, bajas y modificaciones tanto de la información lógica como física asociada a un recurso. Finalmente, dichas operaciones debieran tener en cuenta los posibles problemas derivados de los conflictos surgidos por versiones incompatibles de los componentes involucrados en el *deployment* físico de recursos.

9.4. Consideraciones finales

La World Wide Web es la colección más grande de documentos que alguna vez haya existido. Sin lugar a dudas, esta invención ha cambiado el mundo de la computación y las comunicaciones mucho más que cualquier otro avance tecnológico. Fiel a su naturaleza cambiante, la Web está evolucionando ahora de un conjunto estático de páginas hacia una colección de servicios dinámicos accesibles a las personas y aplicaciones en formas variadas.

Con el fin de crear una herramienta que facilite la programación de aplicaciones en esta nueva versión de la WWW, se ha desarrollado el lenguaje SWAM, que combina los beneficios de los agentes móviles y los Servicios Web Semánticos por medio la materialización del modelo de ejecución MIG. Este modelo aprovecha las ventajas en cuanto a escalabilidad, performance, tolerancia a fallas y flexibilidad que ofrece la promisoria tecnología de agentes móviles para la construcción de aplicaciones en ambientes distribuidos y complejos, haciendo uso al mismo tiempo de los estándares más significativos relacionados con la tecnología de los Servicios Web Semánticos.

Ciertamente, SWAM y MIG constituyen un avance hacia la provisión de una infraestructura masivamente distribuida para la creación y ejecución de agentes móviles inteligentes que aprovechan la vasta cantidad de recursos Web existentes. En este sentido, ambos generan nuevas posibilidades de investigación futura, tal es el caso de la adopción de ASP para administrar y razonar a partir de información Web incompleta o contradictoria, y la materialización de mecanismos de especificación de procesos de negocios y composición de servicios a fin de facilitar aún más el desarrollo de aplicaciones basadas en Servicios Web.

Bibliography

R. Akkiraju, R. Goodwin, P. Doshi, y S. Roeder. A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI. En S. Kambhampati y C. A. Knoblock, editores, *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, páginas 87–92. Acapulco, Mexico, 2003.

J. J. Alferes, C. V. Damásio, y L. M. Pereira. Semantic Web Logic Programming Tools. En *International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2003), Navi Mumbai, India*, tomo 2901 de *Lecture Notes in Computer Science*, páginas 16–32. Springer-Verlag, 2003. ISBN 3-540-20582-9.

A. Amandi, M. Campo, y A. Zunino. JavaLog: A Framework-Based Integration of Java and Prolog for Agent-Oriented Programming. *Computer Languages, Systems & Structures*, 31(1):17–33, 2005. Elsevier Science. ISSN 0096-0551.

A. Amandi, A. Zunino, y R. Iturregui. Multi-paradigm Languages Supporting Multi-Agent Development. En F. J. Garijo y M. Boman, editores, *Multi-Agent System Engineering*, tomo 1647 de *Lecture Notes in Artificial Intelligence*, páginas 128–139. Springer-Verlag, Valencia, Spain, 1999.

ANSI and ISO. *Information Processing: Text and Office Systems: Standard Generalized Markup Language (SGML)*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1985.

G. Antoniou y F. van Harmelen. Web Ontology Language: OWL. En S. Staab y R. Studer, editores, *Handbook on Ontologies in Information Systems*. Springer-Verlag, 2003.

Apache Software Foundation. WSDL Java Extension. `http://ws.apache.org/wsif/providers/wsdl_extensions/java_extension.html`, 2003.

Apache Software Foundation. Apache Axis implementation. `http://http://ws.apache.org/axis/`, 2004.

D. C. Arnold, H. Casanova, y J. Dongarra. Innovations of the NetSolve Grid Computing System. *Concurrency and Computation: Practice and Experience*, 14(13–15):1457–1479, 2002. John Wiley and Sons. ISSN 1532-0626 (print), 1532-0634 (electronic).

F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, y P. F. Patel-Schneider, editores. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003. ISBN 0-521-78176-0.

K. Ballinger, P. Brittenham, A. Malhotra, W. A. Nagy, y S. Pharies. Web Services Inspection Language (WS-Inspection) 1.0 Specification. `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`, 2001.

F. Bellifemine, A. Poggi, y G. Rimassa. JADE: A FIPA-Compliant Agent Framework. En *Proceedings of the Practical Applications of Intelligent Agents (PAAM'99)*, páginas 97–108. London, UK, 1999.

T. Berners-Lee. Information Management: A Proposal. `http://www.w3.org/History/1989/proposal.html`, 1989.

T. Berners-Lee. *Weaving the Web*. Harpur, San Francisco, CA, 1999.

T. Berners-Lee, J. Hendler, y O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001. Macmillan Publishers, Brunel Road, Houndmills, Basingstoke Hampshire RG21 6XS, England.

BinNet Corporation. Jinni 2004 Prolog Compiler: A High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming. `http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html`, 2004.

A. L. Blum y M. L. Furst. Fast Planning Through Planning Graph Analysis. En *Proceedings of the Forteenth International Joint Conference on Artificial Intelligence (IJCAI95), Montreal, Canada*, páginas 1636–1642. 1995.

J. M. Bradshaw. *Software Agents*. AAAI Press, Menlo Park, USA, 1997. ISBN 0-262-52234-9.

T. Bray, J. Paoli, y C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, `http://www.w3.org/TR/REC-xml`, 1998.

M. Breugst, L. Choy, M. Hoft, y T. Magedanz. Grasshopper - An Agent Platform for Mobile Agent-based Services in Fixed and Mobile Telecommunications Environments. En A. L. G. Hayzelden y J. Bigham, editores, *Software Agents for Future Communication Systems*, páginas 326–357. Springer-Verlag, 1999. ISBN 3-540-65578-6.

D. Brickley y R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, `http://www.w3.org/TR/rdf-schema/`, 2004.

B. Burg. Agents in the World of Active Web-Services. En *Lecture Notes In Computer Science. Revised Papers from the Second Kyoto Workshop on Digital Cities II, Computational and Sociological Approaches*, páginas 343–356. Springer-Verlag, 2002. ISBN 3-540-43963-3.

M. H. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, y K. P. Sycara. DAML-S: Web Service Description for the Semantic Web. En I. Horrocks y J. A. Hendler, editores, *Proceedings of the First International Semantic Web Conference (ISWC 02)*, tomo 2342 de *Lecture Notes in Computer Science*, páginas 348–363. Springer-Verlag, London, UK, 2002. ISBN 3-540-43760-6.

C. Bussler, D. Fensel, y A. Maedche. A Conceptual Architecture for Semantic Web Enabled Web Services. *ACM Special Interest Group on Management of Data*, 31(4):24–29, 2002. ACM Press. ISSN 0163-5808.

R. Buyya. The Virtual Laboratory Project: Molecular Modeling for Drug Design on Grid. *IEEE Distributed Systems Online*, 2(5), 2001. IEEE Computer Society.

R. Buyya. The World-Wide Grid. `http://www.buyya.com/ecogrid/wwg/`, 2002.

L. Camarinha-Matos y H. Afsarmanesh. *Infrastructure For Virtual Enterprises: Networking Industrial Enterprises*. Kluwer Academic Press: Norwell, 1999. ISBN 0-792-38639-6.

N. Chatterjee y R. Mitra. MUPPA: An Improvement of Graphplan Algorithm for Planning in Real World Applications. En *Proceedings of the International Conference on Knowledge Based Computer Systems (KBCS 2002), Navi Mumbai, India*. 2002.

D. M. Chess, C. G. Harrison, y A. Kershenbaum. Mobile Agents: Are they a good idea? *Mobile Object Systems - Towards the Programmable Internet*, páginas 25–47, 1997. Springer-Verlag.

E. Christensen, F. Curbera, G. Meredith, y S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, `http://www.w3.org/TR/wsdl`, 2001.

S. H. Chuah, S. W. Loke, S. Krishnaswamy, y A. Sumartono. CALMA: Context-Aware Lightweight Mobile BDI Agents for Ubiquitous Computing. En *Proceedings of the Workshop on Agents for Ubiquitous Computing (UbiAgents 2004)*. New York, NY, USA, 2004. Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), New York City, USA.

M. Cremonini, A. Omicini, y F. Zambonelli. Multi-Agent Systems on the Internet: Extending the Scope of Coordination towards Security and Topology. En F. J. Garijo y M. Boman, editores, *Multi-Agent Systems Engineering*, tomo 1647 de *LNAI*, páginas 77–88. Springer-Verlag, 1999. ISBN 3-540-66281-2. ISSN 0302-9743. Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'99), Valencia, Spain, 30 junio – 2 julio 1999.

F. Curbera, Y. Goland, Y. Klein, F. Leymann, D. Roller, y S. Weerawarana. Business Process Execution Language for Web Services Specification Version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel/`, 2003.

F. Curbera, W. A. Nagy, y S. Weerawarana. Web Services: Why and How. En *Workshop on Object-Oriented Web Services, in (OOPSLA 2001), Tampa, Florida, USA*. ACM Press, 2001.

J. de Bruijn, A. Polleres, y D. Fensel. Deliverable D20v0.1 OWL Lite, WSML Working Draft. `http://www.wsmo.org/2004/d20/v0.1/`, 2004.

Y. Demazeau y J. Müller. From Reactive to Intentional Agents. En Y. Demazeau y J. Müller, editores, *Decentralized A.I. 2 — Proceedings of the Second European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-90)*, páginas 3–10. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1991.

I. Dickinson y M. Wooldridge. Towards Practical Reasoning Agents for the Semantic Web. En *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, páginas 827–834. ACM Press, 2003. ISBN 1-58113-683-8.

M. J. Duftler, N. K. Mukhi, A. Slominski, y S. Weerawarana. Web Services Invocation Framework (WSIF). `www.research.ibm.com/people/b/bth/OOWS2001/duftler.pdf`, 2001.

D. Fensel y C. Bussler. The Web Service Modelling Framework WSMF. En *Proceeding of the NSF-EU Workshop on Database and Information Systems Research for Semantic Web and Enterprises*, páginas 15–20. Georgia, USA, 2002.

D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, y P. F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001. IEEE Computer Society.

R. Fikes y D. L. McGuinness. An Axiomatic Semantics for RDF, RDF Schema and DAML+OIL. Informe Técnico KSL-01-01, Stanford University, 2001.

R. E. Fikes y N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. En J. Allen, J. Hendler, y A. Tate, editores, *Readings in Planning*, páginas 88–97. Morgan-Kaufmann Publishers, 1990.

T. Finin, Y. Labrou, y J. Mayfield. KQML as an Agent Communication Language. En (Bradshaw, 1997).

I. Foster y C. Kesselman, editores. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann Publishers, 1999. ISBN 1-558-60475-8.

I. Foster, C. Kesselman, J. Nick, y S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

I. Foster, C. Kesselman, y S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001. SAGE Publications Ltd., 1 Oliver's Yard, 55 City Road, London, UK. ISSN 1094-3420.

E. Friedman-Hill. Jess, The Expert System Shell for Java Platform. Sandia National Laboratories, `http://herzberg.ca.sandia.gov/jess/docs/manual.pdf`, 2003.

A. Fuggetta, G. P. Picco, y G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998. IEEE Press.

E. Gamma, R. Helm, R. Johnson, y J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

A. Garcia, C. Chavez, O. Silva, V. Silva, y C. Lucena. Promoting Advanced Separation of Concerns in Intra-Agent and Inter-Agent Software Engineering. En *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA'2001*. 2001.

D. Gelernter. Generative Communications in Linda. *ACM Computing Surveys*, 7(1):80–112, 1985. ACM Press, New York, NY, USA.

M. Gelfond y V. Lifschitz. The Stable Model Semantics for Logic Programming. En R. A. Kowalski y K. Bowen, editores, *Proceedings of the Fifth International Conference on Logic Programming*, páginas 1070–1080. The MIT Press, Cambridge, Massachusetts, 1988.

G. D. Giacomo, Y. Lesperance, y H. J. Levesque. ConGolog: A Concurrent Programming Language based on Situation Calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000. Elsevier Science.

G. D. Giacomo y H. Levesque. An incremental interpreter for high-level programs with sensing. En H. F. P. H. Levesque, editor, *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, páginas 86–102. Springer-Verlag, 1999.

R. S. Gray, D. Kotz, G. Cybenko, y D. Rus. Mobile Agents: Motivations and State-of-the-Art Systems. Informe Técnico TR2000-365, Dartmouth College, Computer Science, Hanover, NH, 2000. URL `ftp://ftp.cs.dartmouth.edu/TR/TR2000-365.ps.Z`.

T. R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993. Academic Press.

J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–36, 2001. IEEE Computer Society.

I. Horrocks. DAML+OIL: a Description Logic for the Semantic Web. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, 2002. IEEE Computer Society.

I. Horrocks, F. van Harmelen, y P. Patel-Schneider. The DAML+OIL Language Specification. DARPA Agent Markup Language (DAML) Web Site, `http://www.daml.org`, 2001.

W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, y K. Stockinger. Data Management in an International Data Grid Project. En *First IEEE/ACM International Workshop on Grid Computing (Grid'2000), Bangalore, India*, páginas 482–496. Springer-Verlag, Berlin, Germany, 2000.

M. N. Huhns. Software Agents: The Future of Web Services. En *Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, tomo 2592 de *Lecture Notes in Artificial Intelligence*, páginas 1–18. Springer-Verlag, 2003. Agent Technologies, Infrastructures, Tools, and Applications for E-Services, NODe 2002 Agent-Related Workshops, Erfurt, Germany, 2002. Revised Papers.

M. N. Huhns y M. P. Singh, editores. *Readings in Agents*. Morgan Kaufmann Publishers, 1997. ISBN 1-55860-495-2.

J. Hunter y W. Crawford. *Java Servlet Programming*. O'Reilly and Associates Inc., 1998.

F. Ishikawa y Y. Tahara. Behavior Descriptions of Mobile Agents for Web Services Integration. En *Proceedings of the IEEE International Conference on Web Services (ICWS 2004)*, páginas 342–349. San Diego, California, USA, 2004. ISBN 0-7695-2167-3.

F. Ishikawa, N. Yoshioka, Y. Tahara, y S. Honiden. Mobile Agent System for Web Services Integration in Pervasive Networks. En C. Bussler, R. Hull, S. McIlraith, M. Orlowska, B. Pernici, y J. Yang, editores, *International Workshop on Ubiquitous Computing (IWUC 2004), Sixth International Conference on Enterprise Information Systems (ICEIS 2004)*, páginas 38–47. Porto, Portugal, 2004a.

F. Ishikawa, N. Yoshioka, Y. Tahara, y S. Honiden. Towards Synthesis of Web Services and Mobile Agents. En Z. Maamar, C. Lawrence, D. Martin, B. Benatallah, K. Sycara, y T. Finin, editores, *Workshop on Web Services and Agent-Based Engineering (WSABE), in (AAMAS'2004)*. New York, NY, USA, 2004b.

N. Jennings, K. Sycara, y M. Wooldridge. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998. Kluwer Academic Publishers.

N. R. Jennings, K. Woghiren, y S. Osborn. Interacting Agents: The Way Forward for Agent-Mediated Electronic Commerce. Informe técnico, Lost Wax, 2000. URL http://www.lostwax.com/lostwax1/opinions/opinions1.html.

D. Johansen, R. van Renesse, y F. B. Schneider. An Introduction to the TACOMA Distributed System. Informe Técnico 95-23, Department of Computer Science, University of Tromsø, Tromsø, Norway, 1995. URL http://www.cs.uit.no/Lokalt/Rapporter/Reports/9523.html.

W. E. Johnston, D. Gannon, y B. Nitzberg. Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid. En *Proceedings of the Eighth International Symposium on High Performance Distributed Computing*, páginas 197–204. IEEE Press, 1999.

L. Kagal, F. Perich, H. Chen, S. Tolia, Y. Zou, T. Finin, A. Joshi, Y. Peng, R. Cost, y C. Nicholas. Agents Making Sense of the Semantic Web. En *First GSFC/JPL Workshop on Radical Agent Concepts (WRAC)*. NASA Goddard Space Flight Center, Maryland, USA, 2001.

T. Kawamura, T. Hasegawa, A. Ohsuga, y S. Honiden. Bee-gent: Bonding and Encapsulation Enhancement Agent Framework for Development of Distributed Systems. En *Proceedings of the Sixth Asia Pacific Software Engineering Conference*, página 260. IEEE Computer Society, 1999. ISBN 0-7695-0509-0. http://www.toshiba.co.jp/beegent/.

D. Kotz, R. Gray, y D. Rus. Future Directions for Mobile Agent Research. *IEEE Distributed Systems Online*, 3(8), 2002. IEEE Computer Society. URL http://dsonline.computer.org/0208/f/kot_print.htm.

D. Kotz y R. S. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, 1999. ACM Press. URL http://www.cs.dartmouth.edu/~dfk/papers/kotz:future2/.

H. Kreger. Web Services Conceptual Arquitecture (WSCA 1.0). Informe técnico, IBM Corporation, 2001. http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.

D. Lange y M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

D. B. Lange y M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999. ACM Press. ISSN 0001-0782. `http://www.acm.org:80/pubs/citations/journals/cacm/1999-42-3/p88-lange/`.

O. Lassila y R. Swick. Resource Description Framework (RDF) Model And Syntax Specification. W3C Recommendation, `http://www.w3.org/TR/REC-rdf-syntax`, 1999.

R. Lee y S. Seligman. *JNDI API Tutorial and Reference: Building Directory-Enabled Java(TM) Applications*. Addison-Wesley, 2000. ISBN 0-201-70502-8.

N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, y A. Polleres. The DLV System. En S. Flesca, S. Greco, N. Leone, y G. Ianni, editores, *European Conference on Logics in Artificial Intelligence (JELIA 2002), Cosenza, Italy*, tomo 2424 de *Lecture Notes in Computer Science*, páginas 537–540. Springer-Verlag, 2002. ISBN 3-540-44190-5.

H. J. Levesque, R. Reiter, I. Lespérance, F. Lin, y R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997. Elsevier Science.

F. Leymann. Web Services Flow Language (WSFL 1.0). Informe técnico, IBM Corporation, 2001. `http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

L. Li y I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. *International Journal of Electronic Commerce*, 8(4):39–60, 2004. ACM Press. ISSN 1086-4415.

P. Maes. Artificial Life Meets Entertainment: Lifelike Autonomous Agents. *Communications of the ACM*, 38(11):108–114, 1995. ACM Press. ISSN 0001-0782.

A. Malik. XML, Ontologies and the Semantic Web. *XML Journal*, 4(2), 2002. SYS-CON Media, 135 Chestnut Ridge Road Montvale, NJ 07645, USA.

P. Marques, P. Simes, L. Silva, F. Boavida, y J. Silva. Providing Applications with Mobile Agent Technology. En *The Fourth IEEE Conference on Open Architectures and Network Programming*. 2001.

D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, y K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. En *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, páginas 482–496. San Diego, CA, USA, 2004.

J. Martin. Web Services: The Next Big Thing. *XML Journal*, 2, 2001. SYS-CON Media, 135 Chestnut Ridge Road Montvale, NJ 07645, USA.

E. Martínez y Y. Lespérance. IG-JADE-PKSlib: An Agent-Based Framework for Advanced Web Service Composition and Provisioning. En *Proceedings of the AAMAS-2004 Workshop on Web Services and Agent-Based Engineering*, páginas 2–10. Morgan-Kaufmann Publishers, New York, NY, USA, 2004a.

E. Martínez y Y. Lespérance. Web Service Composition as a Planning Task: Experiments using Knowledge-Based Planning. En *Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services*, páginas 62–69. Morgan-Kaufmann Publishers, Whistler, British Columbia, Canada, 2004b.

C. Mateos, A. Zunino, y M. Campo. Extending MoviLog for Supporting Web Services. *Computer Languages, Systems & Structures*, 2005a. Elsevier Science. ISSN 1477-8424. To appear.

C. Mateos, A. Zunino, y M. Campo. Integrating Intelligent Mobile Agents with Web Services. *International Journal of Web Services Research*, 2(2):85–103, 2005b. Idea Group Publishing, 701 E. Chocolate Avenue, Hershey, PA 17033, USA.

J. McCarthy y P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. En B. L. Webber y N. J. Nilsson, editores, *Readings in Artificial Intelligence*, páginas 431–450. Morgan-Kaufmann Publishers, Los Altos, CA, 1981.

S. McIlraith, T. Son, y H. Zeng. Semantic Web Services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 16(2):46–53, 2001. IEEE Computer Society. ISSN 1094-7167.

S. A. McIlraith y T. C. Son. Adapting Golog for Programming the Semantic Web. En *Proceedings of the Fifth Symposium on Logical Formalizations of Commonsense Reasoning (CommonSense-01)*, páginas 195–202. New York, NY, USA, 2001.

S. A. McIlraith y T. C. Son. Adapting Golog for Composition of Semantic Web Services. En D. Fensel, F. Giunchiglia, D. McGuinness, y M.-A. Williams, editores, *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, páginas 482–496. Morgan-Kaufmann Publishers, San Francisco, CA, USA, 2002.

D. Milojicic, F. Douglis, y R. Wheeler, editores. *Mobility - Processes, Computers and Agents*. Addison-Wesley, 1999.

M. Minsky. *The Society of Mind*. Simon and Schuster, New York, 1985.

S. Morini, A. Ricci, y M. Viroli. Integrating a MAS coordination infrastructure with Web Services. En Z. Maamar, C. Lawrence, D. Martin, B. Benatallah, K. Sycara, y T. Finin, editores, *Workshop on Web Services and Agent-Based Engineering (WSABE) (AAMAS'2004), New York, NY, USA*. 2004.

B. Moulin y B. Chaib-draa. An Overview of Distributed Artificial Intelligence. En G. O'Hare y N. Jennings, editores, *Foundations of Distributed Artificial Intelligence*, capítulo 1. John Wiley and Sons, 1996.

National Sciences Foundation. NFS Tera-Grid. `http://www.teraGrid.org`, 2002.

D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, y S. Mitchell. Total-Order Planning with Partially Ordered Subtasks. En *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI01)*, páginas 425–430. Morgan-Kaufmann Publishers, 2001.

I. Niemelä, P. Simons, y T. Syrjänen. Smodels: A System for Answer Set Programming. En *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (cs.AI/0003073)*. Breckenridge, Colorado, USA, 2000.

H. Nwana. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3):205–244, 1996. Cambridge University Press.

H. S. Nwana y N. Azarmi. *Software agents and soft computing: towards enhancing machine intelligence: concepts and applications*, tomo 1198 de *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*. Springer-Verlag, New York, NY, USA, 1997. ISBN 3-540-62560-7.

A. Omicini y F. Zambonelli. Coordination of Mobile Information Agents in TuCSoN. *Internet Research: Electronic Networking Applications and Policy*, 8(5):400–413, 1998. MCB University Press. ISSN 1066-2243.

A. Oram. *Peer-To-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly and Associates Inc., primera edición, 2001. ISBN 0-596-00110-X.

P. Palathingal y S. Chandra. Agent Approach for Service Discovery and Utilization. En *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), Big Island, HI, USA*. IEEE Computer Society, 2004. ISBN 0-7695-2056-1.

M. Paolucci, T. Kawamura, T. R. Payne, y K. Sycara. Importing the Semantic Web in UDDI. En C. Bussler, R. Hull, S. McIlraith, M. Orlowska, B. Pernici, y J. Yang, editores, *CAiSE '02/ WES '02: Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web*, páginas 225–236. Springer-Verlag, London, UK, 2002. ISBN 3-540-00198-0.

M. Paolucci y K. Sycara. Autonomous Semantic Web Services. *IEEE Internet Computing*, 7(5):34–41, 2003. IEEE Educational Activities Department. ISSN 1089-7801.

H. Parunak. Go to the ant: Engineering principles from Natural MultiAgents Systems. En *Annals of Operations Research*. 1996.

T. R. Payne, M. Paolucci, y K. Sycara. Advertising and Matching DAML-S Service Descriptions. En *International Semantic Web Working Symposium (SWWS)*. California, CA, 2001. Position Paper.

R. P. Petrick y F. Bacchus. A Knowledge-Based Approach to Planning with Incomplete Information and Sensing. En *Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, páginas 212–221. AAAI Press, Menlo Park, CA, USA, 2002.

R. P. Petrick y F. Bacchus. Reasoning with Conditional Plans in the Presence of Incomplete Knowledge. En *Workshop on Planning Under Uncertainty and Incomplete Information (ICAPS-03)*, páginas 96–102. Trento, Italy, 2003.

T. Pilioura y A. Tsalgatidou. E-Services: Current Technology and Open Issues. En F. Casati, D. Georgakopoulos, y M. C. Shan, editores, *Technologies for E-Services, Second International Workshop, TES 2001*, tomo 2193 de *Lecture Notes in Computer Science*, páginas 1–15. Springer-Verlag, 2001.

S. Pokraev, J. Koolwaaij, y M. Wibbels. Extending UDDI with Context-Aware Features Based on Semantic Service Descriptions. En L. Zhang, editor, *Proceedings of the IEEE International Conference on Web Services (ICWS 2003)*, páginas 184–190. 2003.

A. Preece y S. Decker. Intelligent Web Services. *IEEE Intelligent Systems*, 17(1):12–15, 2002. IEEE Computer Society.

A. S. Rao y M. P. Georgeff. BDI Agents: from theory to practice. En V. Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, páginas 312–319. MIT Press, San Francisco, CA, 1995.

A. Ricci, A. Omicini, y E. Denti. The TuCSoN Coordination Infrastructure for Virtual Enterprises. En *IEEE 10th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2001), MIT, Cambridge, MA, USA*, páginas 348–353. IEEE Computer Society, 2001. ISBN 0-7695-1269-0. ISSN 1080-1383.

P. Rigaux. The MyReview System. `http://www.lri.fr/~rigaux/myreview.pdf`, 2004.

N. M. Sadeh, T. Chan, L. Van, O. Kwon, y K. Takizawa. Creating an Open Agent Environment for Context-Aware m-Commerce. *Agentcities: Challenges in Open Agent Environments*, páginas 152–158, 2003. Springer-Verlag.

M. Shaw y D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs , NJ , USA, 1996.

E. Sirin, J. Hendler, y B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. En *Workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS 2003)*. Angers, France, 2003.

K. Sivashanmugam, K. Verma, A. P. Sheth, y J. A. Miller. Adding Semantics to Web Services Standards. En L. Zhang, editor, *Proceedings of the IEEE International Conference on Web Services (ICWS 2003)*, páginas 395–401. 2003.

T. Sollazzo, S. Handschuh, S. Staab, y M. Frank. Semantic Web Service Architecture – Evolving Web Service Standards toward the Semantic Web. En *Proceedings of the 15th International FLAIRS Conference, Pensacola, Florida, USA*. ACM Press, 2002.

B. Srivastava y J. Koehler. Planning with Workflows - An Emerging Paradigm for Web Service Composition. En *Proceedings of the Workshop on Planning and Scheduling for Web and Grid Services (ICAPS 2004)*. Morgan-Kaufmann Publishers, Whistler, British Columbia, Canada, 2004.

S. Staab, W. M. P. van der Aalst, V. R. Benjamins, A. P. Sheth, J. A. Miller, C. Bussler, A. Maedche, D. Fensel, y D. Gannon. Web Services: Been There, Done That? *IEEE Intelligent Systems*, 18(1):72–85, 2003. IEEE Computer Society.

K. Sycara, M. Klusch, S. Widoff, y J. Lu. Dynamic Service Matchmaking Among Agents in Open Information Environments. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28:47–53, 1999. ACM Press.

P. Tarau. BinProlog: A Continuation Passing Style Prolog Engine. En M. Bruynooghe y M. Wirsing, editores, *Programming Language Implementation and Logic Programming: Proc. of the Fourth International Symposium PLILP'92*, páginas 479–480. Springer, Berlin, Heidelberg, 1992.

P. Tarau. Jinni: a Lightweight Java-based Logic Engine for Internet Programming. En K. Sagonas, editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*. Manchester, U.K., 1998. Invited talk.

P. Tarau y E. Figa. Knowledge-Based Conversational Agents and Virtual Storytelling. En *Proceedings of the 2004 ACM symposium on Applied computing (SAC-04)*, páginas 39–44. ACM Press, New York, NY, USA, 2004.

S. Thatte. XLANG: Web Services for Business Process Design. Informe técnico, Microsoft, 2001. URL `www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

The Foundation for Intelligent Physical Agents. The FIPA'97 Specification. `http://drogo.cselt.it/fipa/spec/fipa97/fipa97.htm`, 1997.

W. Theilmann y K. Rothermel. Domain Experts for Information Retrieval in the World Wide Web. En G. Wei y M. Klusch, editores, *Cooperative Information Agents CIA'98*, páginas 216–227. 1998.

A. R. Tripathi, N. M. Karnik, T. Ahmed, R. D. Singh, A. Prakash, V. Kakani, M. K. Vora, y M. Pathak. Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 62(2):123–140, 2002. Elsevier Science.

US Census Bureau. North American Industry Classification System (NAICS). `http://www.census.gov/epcd/www/naics.html`, 2002.

R. van de Stadt. CyberChair: a Web-based Groupware Application to Facilitate the paper Reviewing Process. `http://www.cyberchair.org`, 2001.

W. M. P. van der Aalst, M. Dumas, y A. H. M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? En *Proceedings of the 29th Conference on EUROMICRO*, tomo 18, páginas 298–305. IEEE Computer Society, 2003. ISBN 0-7695-1996-2.

S. J. Vaughan-Nichols. Web Services: Beyond the Hype. *Computer*, 35(2):18–21, 2002. IEEE Computer Society. ISSN 0018-9162.

G. Vigna. Mobile Code Technologies, Paradigms, and Applications. PhD Thesis, Politecnico di Milano, Italia, 1998.

M. Viroli y A. Ricci. Instructions-Based Semantics of Agent Mediated Interaction. En *International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2004), New York, NY, USA*, tomo 1, páginas 102–109. 2004.

W3C Consortium. Simple Object Access Protocol (SOAP) 1.1 Specification. http://www.w3.org/TR/SOAP/, 2000.

W3C Consortium. UDDI Technical White Paper. UDDI Home Site, http://www.uddi.org, 2001.

W3C Consortium. Web Services Architecture Requirements. W3C Working Draft, http://www.w3.org/TR/2002/WD-wsa-reqs-20020429, 2002.

G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

J. E. White. Telescript Technology: Mobile Agents. En (Bradshaw, 1997).

S. Willmott, J. Dale, y B. Burg. Agentcities: Connecting Agents across the World. En W. Truszkowski, C. Rouff, y M. G. Hinchey, editores, *Innovative Concepts for Agent-Based Systems, First International Workshop on Radical Agent Concepts, WRAC 2002*, tomo 2564 de *Lecture Notes in Computer Science*, páginas 453–457. Springer-Verlag, 2002. ISBN 3-540-40725-1.

D. Wong, N. Paciorek, y D. Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102, 1999. ACM Press. URL http://www.acm.org/pubs/articles/journals/cacm/1999-42-3/p92-wong/p92-wong.pdf.

M. Wooldridge. Agent-based software engineering. *IEE Proceedings of Software Engineering*, 144(1):26–37, 1997. URL http://www.elec.qmw.ac.uk/dai/people/mikew/pubs/iee-se.ps.

Workflow Management Coalition. The Workflow Management Coalition: Terminology and Glossary. Document Number WFMC-TC-1011, 1999.

D. Wu, E. Sirin, B. Parsia, J. Hendler, y D. Nau. Automatic Web Services Composition Using SHOP2. En *Proceedings of then Planning for Web Services Workshop (ICAPS 2003)*. Trento, Italy, 2003.

Zakon Group. OpenConf Documentation. http://www.zakongroup.com/technology/openconf-documentation.shtml, 1992.

A. Zunino, M. Campo, y C. Mateos. Simplifying Mobile Agent Development Through Reactive Mobility by Failure. En G. Bittencourt y G. Ramalho, editores, *Advances in Artificial Intelligente*, tomo 2507 de *Lecture Notes in Computer Science*. Springer-Verlag, 2002. ISBN 3-540-00124-7.

A. Zunino, C. Mateos, y M. Campo. Enhancing Agent Mobility Through Resource Access Policies and Mobility Policies. En *V Encontro Nacional de Inteligência Artificial (ENIA), XXV Congresso da Sociedade Brasileira de Computacão (SBC)*. San Leopoldo, RS, Brasil, 2005a.

A. Zunino, C. Mateos, y M. Campo. Reactive Mobility by Failure: When Fail Means Move. *Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications*, 7(2):141–154, 2005b. Springer Science + Business Media B.V. ISSN 1387-3326.