Magentix2: Una Nueva Plataforma para Sistemas Multiagente Abiertos

Ricard López Fogués

Departamento de Computación y Sistemas Informáticos

Universitat Politécnica de València

Dirigido por:

Agustín Espinosa, Ana García-Fornes

Septiembre 2010

Agradecimientos

A mis directores, Ana y Agustín, por su ayuda y apoyo en la realización de este proyecto y durante el tiempo que llevo trabajando con ellos.

Índice general

Índice de figuras V

1. Introducción 1

2. Motivación y Objetivos 5

3. Plataforma Magentix2 9

3.1. Plataforma Previa . . . 9

3.2. Nivel de Interacción . . . 10

3.2.1. AMQP . . . 10

3.2.1.1. El Modelo AMQP . . . 11

3.2.1.2. El Protocolo de Red AMQP . . . 13

3.2.2. Implementación AMQP Apache Qpid . . . 14

3.2.3. Arquitectura de la Comunicación . . . 15

3.2.3.1. FIPA-ACL sobre AMQP . . . 16

3.2.3.2. FIPA-ACL sobre FIPA-HTTP . . . 17

4. Conversaciones de los Agentes en Magentix2 19

4.1. Interacción y Agentes . . . 20

4.2. Procesadores de Conversaciones . . . 22

4.3. Fábricas de Conversaciones . . . 26

4.4. Conversaciones Anidadas . . . 27

4.5. Visión Global de un Agente Conversacional . . . 29

4.6. Ejemplo de Implementación de Agentes Magentix2 . . . 31

4.6.1. Ejemplo de un Agente Conversacional Básico . . . 31

4.6.2. Ejemplo de uso de las Fabricas de Conversaciones . . . 32

4.6.3. Ejemplo de uso de las Plantillas de Protocolos 35

5. Agentes BDI en Magentix2 **41**

5.1. Jason 43

5.2. Jason sobre Magentix2 47

6. Organizaciones de Agentes en Magentix2 **49**

6.1. Moise$^+$ 49

6.1.1. Dimensión Estructural 50

6.1.2. Dimensión Funcional 50

6.1.3. Dimensión Deóntica 52

6.2. S-Moise$^+$ y J-Moise$^+$ 52

6.3. Moise$^+$ sobre Magentix2 53

6.4. THOMAS 54

6.4.1. Agente Intermediario SF 55

6.4.2. Agente intermediario OMS 56

6.5. THOMAS sobre Magentix2 57

7. Conclusiones y Trabajo Futuro **59**

Bibliografía **63**

Índice de figuras

3.1. Modelo AMQP 12
3.2. AMQP Stack 13
3.3. Servidor Qpid y aplicaciones heterógeneas 15
3.4. Arquitectura de la comunicación en Magentix2 16
3.5. Pila de estándares de la comunicación en Magentix2 17

4.1. Grafo para el rol de taxista (participante) en el *Protocolo de Interacción del Taxi* 23
4.2. Grafo para el rol de cliente (iniciador) en el *Protocolo de Interacción del Taxi* . 23
4.3. Protocolo del taxista modificado dinámicamente 26
4.4. Protocolo del taxista usando subprotcolos 28
4.5. Protocolo de interacción FIPA Request para el rol iniciador 29
4.6. Visión global de un agente 30
4.7. Funcionamiento de la *CFactory* por defecto 1 31
4.8. Funcionamiento de la *CFactory* por defecto 2 31

5.1. Ciclo de razonamiento de Jason 45
5.2. Estructura de JasonAgent 48

6.1. Estructura de un equipo de fútbol usando Mosie$^+$ 51
6.2. Esquema de ataque para equipo de fútbol usando Mosie$^+$ 52
6.3. Integración de Moise$^+$ en Magentix2 54
6.4. Ejemplo de organizaciones THOMAS Magentix2 58

ÍNDICE DE FIGURAS

Capítulo 1

Introducción

Los sistemas basados en agentes son una de las áreas más vibrantes e importantes de investigación y desarrollo que han aparecido en el campo de la tecnología de la información. El concepto de agente se usa en muchas subdisciplinas de la tecnología de la información, incluyendo redes de computadores, ingeniería del software, inteligencia artificial, interacción humano-máquina, sistemas concurrentes, sistemas móviles y un largo etcétera.

Como se afirma en [32], los Sistemas Multiagente (SMA) permiten el desarrollo de sistemas computacionales complejos en entornos abiertos. La tecnología de agentes está relacionada con un amplio abanico de campos, especialmente aquellos identificados como entornos abiertos y dinámicos, donde los agentes puedan actuar en representación de sus controladores, buscando servicios, negociando contratos y realizando decisiones proactivas mientras responden a circustancias variables. Esta visión de los sistemas demanda el desarrollo e integración de infrastructuras en las cuales los agentes con diferentes capacidades sean capaces de colaborar y crear coaliciones.

Aplicaciones prácticas basadas en tecnología SMA serán mayormente realizadas en el futuro para sistemas abiertos (mercados electrónicos, negocios virtuales, organizaciones virtuales, etc.) donde los miembros sean desarrollados por diferentes grupos y tengan objetivos confrontados [9]. En los últimos años, muchos investigadores han centrado sus esfuerzo en este tipo de sistemas: instituciones electrónicas [19], organizaciones virtuales [5], etc.

Los sistemas multiagente pueden ser abiertos, en este caso concreto de sistema multiagente, los agentes no se conocen previamente. Los agentes del sistema pueden haber sido desarrollados por grupos diferentes, con diferentes tecnologías y metodologías, puede que incluso se ejecuten en diferentes lugares geográficos. Por lo tanto estos sistemas han de ser capaces

de asimilar esta heterogeneidad entre los agentes y facilitar mecanismos de comunicación y organización que todos los agentes puedan comprender y utilizar. A este respecto existen ya diferentes iniciativas cuyo objetivo es la estandarización de los procedimientos de interacción y comunicación entre los agentes, una de las más conocidas es FIPA [20] aceptada por IEEE. FIPA define unos estándares en protocolos de comunicación entre agentes los cuales son de gran utilidad para facilitar la interacción entre agentes heterogéneos en un SMA abierto.

Uno de los ámbitos de estudio en el campo de los agentes es el de las plataformas para SMA. Son muchos los grupos de investigación y desarrollo dedicados a esta tarea, también son muchas las plataformas que actualmente se encuentran disponibles para los desarolladores de SMA [1; 3; 4; 10]. Cada una de ellas presenta ciertas ventajas, funcionalidades concretas y hacen uso de diferentes tecnologías.

Según [32] las infraestructuras computacionales que soportan los SMA se centran en tres niveles. Una plataforma multiagente debe ofrecer soporte y herramientas para el desarrollo de SMA en cada uno de estos niveles, los cuales son:

1. Nivel de organización: En este nivel encontramos tecnologías y técnicas que permiten a los agentes adquirir roles, así como especificar aquello que pueden, deben y tienen prohibido realizar. Todas estas especificaciones se encuentran definidas en este nivel y constituyen las normas de la organización

2. Nivel de interacción: Este nivel ofrece tecnologías y técnicas relacionadas con la comunicación entre agentes. Por ejemplo, el estándar FIPA antes comentado quedaría enmarcado en este nivel.

3. Nivel de agente: Por último en este nivel encontramos tecnologías y técnicas relacionadas con agentes individiaules, como son el razonamiento y el aprendizaje entre otras.

En esta tesina se presenta Magentix2, una nueva plataforma de agentes. Esta plataforma ofrece un conjunto de características propias que la hacen diferente respecto a otras plataformas actuales. Algunas de estas características son el uso de un moderno estándar de comunicación para ofrecer soporte al nivel de interacción, posibilidad de que agentes heterogéneos funcionen sobre la misma plataforma, soporte a protocolos de interacción dinámicos, un sistema de trazas para facilitar la supervisión de las tareas, implantación de técnicas de seguridad en la comunicación e identificación de los agentes, posibilidad de programar agentes inteligentes y

capacidad de definir organizaciones de agentes. El objetivo principal de la plataforma Magentix2 es ofrecer soporte al desarrolador de SMA en los tres niveles comentados anteriormente, es decir, organizativo, de interacción y de agente.

La plataforma Magentix2 se enmarca en el proyecto de investigación “MAGENTIX II: Una plataforma para sistemas multiagente abiertos” a cargo del Ministerio de Ciencia e Innovación cuya referencia es TIN2008-04446/TIN y en el proyecto “Agreement Technologies” perteneciente al programa Consolider Ingenio con referencia CSD2007-00022.

1. INTRODUCCIÓN

Capítulo 2

Motivación y Objetivos

Tal y como se ha comentado en la introducción, el objetivo de Magentix2 es ofrecer soporte en todos los niveles de un SMA. Este objetivo permitirá que Magentix2 sea una plataforma válida sobre la que basar futuras investigaciones en el campo de los agentes inteligentes.

Teniendo en mente el objetivo principal de Magentix2 se ha definido una lista de objetivos, los cuales abarca este trabajo. Los objetivos son los siguientes:

- Los agentes que funcionen sobre Magentix2 pueden estar programados en cualquier lenguaje de programación. Es decir, la plataforma ha de ser capaz de trabajar con componentes heterógeneos.

- Los agentes de la plataforma pueden ser desarrollados sin seguir ninguna arquitectura de agente concreta y aún así aprovechar las ventajas y herramientas de Magentix2.

- Ofrecer un método de interacción entre agentes que sea transparente al desarrollador de SMA y a los agentes.

- La plataforma debe facilitar la definición y el uso de protocolos de interacción dinámicos entre agentes.

- Facilitar un soporte para el desarrollo de agentes dentro de la plataforma según el modelo BDI (*belief-desire-intention*).

- Los agentes desarrollados en la plataforma han de ser capaces de formar organizaciones según diferentes modelos de organización.

Como se ha explicado anteriormente los SMA abiertos están formados por agentes que no se conocen previamente, han sido desarrollados por grupos diferentes e incluso pueden no estar ejecutándose en el mismo lugar. Por tanto una plataforma que pretenda dar soporte a estos sistemas ha de ser capaz de asimilar esa heterogeneidad. Como se mostrará en el desarrollo de esta tesina, Magentix2 se basa en el estándar AMQP y su implementación QPid. Esta implementación dispone de APIs para diferentes lenguajes, como C++, Java, C#, .Net, Ruby y Python, por lo tanto agentes programados en alguno de estos lenguajes pueden coexistir en Magentix2 y ser capaces de interactuar entre ellos.

Magentix2 pretende ser la plataforma utilizada por los miembros del programa Consolider en cuanto a investigación en el campo de agentes inteligentes se refiere. Cada investigación, y por tanto cada grupo, tiene unas necesidades específicas respecto a sus agentes, por lo que es de esperar que se requieran diferentes arquitecturas de agentes para cada una de estas investigaciones. En vista a esta necesidad, Magentix2 no impone ninguna arquitectura de agente concreta y deja en manos del desarrollador esta decisión, lo que ofrece Magentix2 es una serie de herramientas y utilidades que facilitan la creación de SMA.

Un requisito que se definió cuando se concibió Magentix2 era que fuese posible integrar trabajo ya realizado dentro del grupo de investigación GTI-IA en la plataforma, los dos objetivos anteriormente expuestos permiten cumplir este requisito. Durante el desarrollo de este trabajo se puso a prueba la capacidad de Magentix2 de trabajar con componentes heterogéneos y de combinar diferentes arquitecturas de agente en un mismo SMA. Concretamente se ha implementado el modelo de organización THOMAS [37] en Magentix2. Este modelo de organización gestado en el grupo de investigación GTI-IA ya se encontraba implementado en la plataforma Jade, su implementación en Magentix2 fue relativamente sencilla y no afectó a la arquitectura de los agentes utilizados en THOMAS. Además, se han realizado pruebas usando el modelo THOMAS donde algunos agentes estaban desarrollados en Magentix2 y otros en Jade. Estas pruebas han servido como demostración de las posibilidades que ofrece Magentix2 para asimiliar la heterogeneidad y por tanto demostrar que cumple los requisitos exigidos.

Un elemento identificativo de los agentes inteligentes son sus capacidades sociales, las cuales se refieren a la comunicación. En este trabajo se realiza una propuesta para el tratamiento conversaciones de agentes. Debido a que los SMA abiertos son entornos muy variables, estos deben ser capaces de adaptar dinámicamente su estructura y comportamiento por medio de la adición, eliminación y substitución de componentes del sistema mientras este está en funcionamiento y sin tener que detenerlo [18; 38]. En este sentido, tal y como se afirma en [7], la

especificación estática de las interacciones no es válida para ciertos entornos abiertos ya que condiciones sociales o del entorno pueden favorecer, o incluso requerir, que las especificaciones sean modificadas durante la ejecución. La propuesta presentada permite la especificación dinámica de protocolos de interacción de manera que las reglas asociadas a un protocolo de interacción específico pueden ser modificadas mientras este se encuentra en ejecución.

Además de la propuesta para modelar conversaciones de agentes, en el presente trabajo, se muestra la herramienta fruto de la implementación de dicha propuesta. Esta herramienta se proporciona en forma de API a nivel de agente que sirve para definir de una manera intuitiva los protocolos de interacción en los que participará el agente.

Actualmente podemos encontrar varias plataformas de agentes que ofrecen herramientas para la comunicación entre agentes. La plataforma SACI [28] ofrece manejadores de mensajes, estos manejadores se ejecutan automáticamente cuando el agente recibe un mensaje adecuado que actúa como disparador, pero no ofrece un tratamiento automático para protocolos de interacción. La plataforma Jade [10] sí ofrece tratamiento automatizado de protocolos de interacción, concretamente ofrece tratamiento para todos aquellos definidos por el estándar FIPA [20], pero no ofrece posibilidad de modificar estos protocolos, ni de definir nuevos protocolos diferentes a los especificados por FIPA.

Según Wooldridge [40] los agentes deben ser: autónomos, proactivos, reactivos y tener capacidades sociales. La autonomía implica que el agente tenga sus propios objetivos y que estos son verdaderamente suyos, es decir, nada ni nadie le dicta esos objetivos explícitamente. Que una agente sea proactivo significa que éste sea capaz de mostrar un comportamiento dirigido por objetivos. Si se ha delegado un objetivo a un agente es de esperar que éste trate de conseguirlo. El agente también ha de ser reactivo y capaz de adaptarse a los cambios que se produzcan en su entorno, no es complicado desarrollar software reactivo pero encontrar un equilibrio entre reactividad y comportamiento dirigido por objetivos sí lo es. Por último, el agente ha de ser social, es decir, que tenga capacidad de comunicación con otros agentes. Esta comunicación permite que los agentes puedan cooperar y coordinarse ente ellos. Es imposible pensar en que un objeto, como concepo Java, sea un agente, pues carece de características que son íntrinsecas al concepto de agente. Por lo tanto una plataforma de agentes ha de ser capaz de aportar medios con los que programar agentes que cumplan estos requerimientos. Magentix2 ofrece la posibilidad de la creación de agentes racionales usando un elegante lenguaje de agentes como es AgentSpeak(L) [34].

Se ha demostrado que las organizaciones de agentes permiten a un SMA llevar a cabo tareas complejas, por medio de una estructuración, cooperación y coordinación de los agentes de la organización. Para estructurar la organización se hace uso de roles y relaciones entre ellos, además se identifican objetivos que los agentes que adquieran un rol concreto deben llevar a cabo para conseguir los objetivos de la organización. Por último se definen unas normas que los agentes deben cumplir si forman parte de la organización.

Conceptos relacionados de una u otra forma con las organizaciones de agentes como sistemas abiertos, negociación, argumentación, contratación, reputación, confianza, conocimiento distribuído, etc. están siendo aplicados al área de los SMA y están en contínuo proceso de innovación e investigación. Es por tanto, un concepto vivo y que está siendo tratado y estudiado por grupos de investigación del área [29; 31; 37].

Es, por tanto, importante que una plataforma actual que pretenda ser útil ofrezca soporte para organizaciones. Magentix2 integra dos contrastados modelos de organización y sus implementaciones en la plataforma y los ofrece como herramientas para el desarrollador de SMA.

Capítulo 3

Plataforma Magentix2

Los sistemas multiagentes donde los miembros han sido desarrollados por grupos con diferentes intereses y donde no existe el acceso al estado interno de los miembros, son clasificados como abiertos. La plataforma Magentix2 es una plataforma orientada a dar soporte a estos SMA. El uso de Magentix2 no implica el uso de una metodología de programación de agentes concreta, esta decisión queda en manos del desarrollador. En definitiva es un conjunto de APIs y utilidades que facilitan la creación de estos sistemas.

3.1. Plataforma Previa

Magentix2 cuenta con un antecesor, Magentix [6]. Esta plataforma anterior fue concebida de manera que fuese lo más eficiente posible ya que estudios previos con otras plataformas demostraron que el rendimiento de estas decrecía muy rápidamente conforme aumentaba el número de agentes en la plataforma. Con esta idea en mente, la plataforma fue desarrollada en lenguaje C y usando funciones próximas al sistema operativo. Los agentes se representan en Magentix como procesos del sistema operativo, y realizan las comunicaciones entre cada par de agentes mediante conexiones punto a punto, utilizando sockets TCP.

Magentix2 abandona C como lenguaje de programación para pasar a dar libertad al desarrollador para elegir el lenguaje de programación que más le convenga. Actualmente la mayor parte de Magentix2 se encuentra orientada a dar soporte a Java, aunque en un futuro, gracias al *middleware* sobre el que se basa, el soporte al resto de lenguajes de programación será un proceso trivial. Se ha elegido Java como primer lenguaje al que brindar soporte debido a que tiene un uso muy extenso en la actualidad en el campo de la programación de agentes [10; 14; 15; 33].

Por otra parte, existe una gran diferencia en el nivel de interacción entre las dos plataformas. Magentix2 usa AMQP *(Advanced Message Queuing Protocol)* [2] un nuevo estándar de comunicación para el intercambio de mensajes entre los agentes. En cambio, Magentix fue diseñado e implementado de manera que los agentes tuviesen que comunicarse utilizando un protocolo de comunicación propio de la plataforma. En la próxima sección se explicará con mayor profundidad este importante aspecto de la nueva plataforma.

3.2. Nivel de Interacción

El soporte que Magentix2 provee para la interacción de agentes se compone de dos partes: comunicación de agentes y, sobre ésta, conversaciones de agentes.

Magentix 2 usa AMQP como base para la comunicación entre agentes. Este estándar facilita la interoperabilidad entre entidades heterogéneas. Por consiguiente, Magentix2 permite que agentes heterógeneos interaccionen entre ellos mediante mensajes estructurados de acuerdo al estándar FIPA-ACL e intercambiados usando el estándar AMQP.

Las interacciones entre los agentes están orientadas a las conversaciones. Cada conversación es identificada por un identificador de conversación específico *conversation_id*, este identificador está asociado a cada mensaje producido durante esa conversación. Una conversación en Magentix2 se representa mediante una secuencia de mensajes que siguen un protocolo de interacción específico. Magentix2 facilita la especificación, ejecución automática y gestión de cada protocolo de interacción que un agente esté manteniendo. Tal y como se ha dicho anteriormente estos protocolos pueden ser modificados dinámicamente para adaptarse a las necesidades del sistema, más adelante en este documento se profundizará en esta capacidad. A continuación se detalla como funciona la comunicación de agentes en la plataforma.

3.2.1. AMQP

AMQP es un estandar para softwares intermedios orientados a mensajes (MOM) [27]. AMQP garantiza plena interoperabilidad entre clientes y servidores (de ahora en adelante brokers) que cumplan con el estándar. Las características principales de AMQP son: interoperabilidad entre aplicaciones heterógeneas, esquemas de comunicación múltiples, definición de interficies de aplicaciones de servicios sin que esto afecte al software intermedio, su fiabilidad y escalabilidad. Además, tal y como afirman Singh y Chopra en [16; 36], AMQP cumple las

propiedades para un sistema de mensajes interoperable, esto es, AMQP es asíncrono y confiable. AMQP es adaptable y capaz de ofrecer multiples arquitecturas de mensajería, incluyendo, aunque no limitandose a estas:

1. *Store-and-forward* con muchos escritores y un solo lector.
2. Distribución de transacciones con muchos escritores y muchos lectores.
3. Productor-consumidor con muchos escritores y muchos lectores.
4. Encaminamiento basado en el contenido con muchos lectores y muchos consumidores.
5. Transferencia de fichero encolado con muchos lectores y muchos escritores.
6. Conexión punto a punto entre dos pares.

AMQP consigue toda esta funcionalidad definiendo las semánticas de los servicios del broker (el modelo AMQP) y el protocolo de red a nivel de cableado (el protocolo de red AMQP). En las próximas secciones se describirán ambos elementos.

3.2.1.1. El Modelo AMQP

El modelo AMQP define la semantica de los servicios del broker (el servidor MOM). Un broker AMQP es un servidor de datos que acepta mensajes de los clientes y realiza básicamente dos acciones con ellos: (i) los encamina a los diferentes consumidores dependiendo de un criterio arbitrario; (ii) y los almacena en memoria o disco cuando los consumidores no son capaces de aceptarlos con suficiente prontitud.

El modelo AMQP define una serie de componentes dentro del broker (como se muestra en la imagen 3.1). Estos componentes se encargan de enrutar y almacenar los mensajes. Además, estos componentes pueden ser conectados entre ellos siguiendo una serie de reglas. La manera en que estos componentes se conecten definirá el modelo de comunicación.

Los componentes básicos de un broker AMQP son:

1. *Message* Un mensaje es la unidad atómica de encaminamiento y encolamiento. Los mensajes tienen una serie de cabeceras que el broker puede leer y usar para encaminar el propio mensaje. Los mensajes, además, también poseen contenido (datos binarios), el cual es inmutable, opaco y esencialmente ilimitado en tamaño.

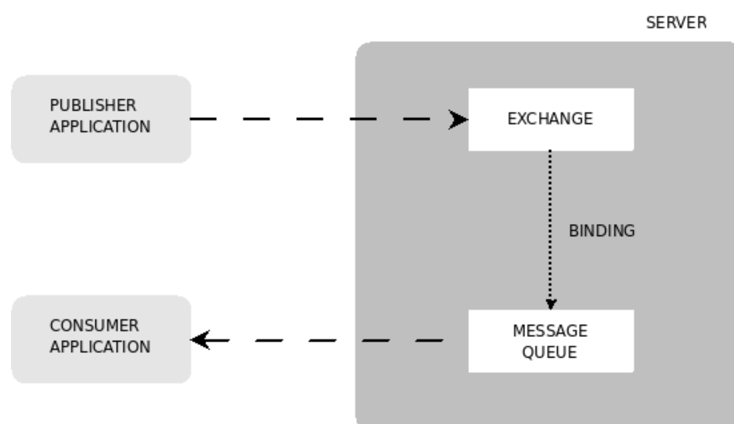

Figura 3.1: Modelo AMQP

2. *Message Queue* Las colas de mensajes son el elemento central del modelo AMQP. Las colas mantienen los mensajes en representación de un conjunto de aplicaciones consumidoras de mensajes. Una cola puede mantener los mensajes en memoria, disco o en ambos medios. Las aplicaciones puede crear, compartir, usar y destruir libremente las colas, siempre dentro de los límites de su autoridad. Las colas de mensajes ofrecen una garantía FIFO limitada, esto es, para mensajes con una prioridad equivalente que se originen de un productor dado, su entrega al consumidor siempre se intentará que sea en el orden en que los mensajes llegaron a la cola. Las colas pueden ser durables, pasajeras o auto-borrables. Las colas durables permanecen en el broker hasta que son borradas, las colas pasajeras duran hasta que el servidor se apaga, por último, las auto-borrables duran hasta que dejan de ser utilizadas.

3. *Exchange* Los intercambiadores son el servicio de entrega para los mensajes. Un intercambiador acepta mensajes y los enruta - normalmente usando una clave de enrutamiento - a una o varias colas de mensajes. Las aplicaciones pueden crear, compartir, usar y destruir libremente intercambiadores, aunque siempre dentro de los límites de su autoridad. Los intercambiadores, de forma semejante a las colas, pueden ser durables, pasajeros o auto-borrables. La duración de los intercambiadores según su tipo es el mismo que para las colas. Un servidor AMQP ofrece diferentes tipos de intercambiadores. Cada uno de estos tipos implementa un algoritmo específico de enrutamiento:

    - *Direct* Estos intercambiadores enrutan basándose en una coincidencia exacta entre

la clave de enrutamiento y la clave de enlace.

- *Topic* Estos intercambiadores enruta basandose en una coincidencia de patrón entre la clave de enrutamiento y la clave de enlace.
- *Fanout* En este caso los intercambiadores enrutan los mensajes a todas las colas enlazadas sin importar la clave de enrutamiento.
- *Headers* Este tipo de intercambiadores enrutan basándose en una expresión compleja y multiparte.

4. *Binding* Un enlace es una relación entre una cola de mensajes y un intercambiador. El enlace especifica los argumentos de enrutamiento que dictan al intercambiador que mensajes debería recibir la cola. Las aplicaciones crean y destruyen los enlaces, según sea necesario, para controlar el flujo de mensajes a sus colas. El tiempo de vida de un enlace depende de la cola y el intercambiador sobre el cual está definido, cuando una cola o un enlace es destruido, también lo es el enlace.

3.2.1.2. El Protocolo de Red AMQP

El estándar AMQP también define un protocolo de red, el cual permite a los clientes hablar con el broker. Por consiguiente, aplicaciones cliente pueden interactuar con el Modelo AMQP que el broker implementa. La figura 3.2 muestra la pila del protocolo de red AMQP.

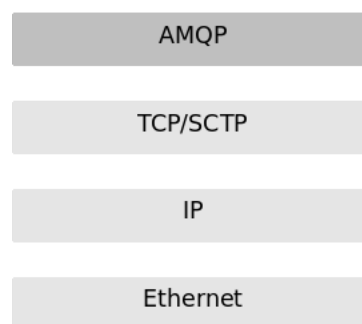

Figura 3.2: AMQP Stack

Los elementos centrales del protocolo de red AMQP son:

1. *Conexión* El concepto de conexión en AMQP consiste en una conexión de red, en otras palabras una conexión TCP o una STCP. El protocolo de red AMQP permite que múltiples diálogos independientes comparta una sola conexión. Cada marco del protocolo de

red AMQP incluye un número que identifica inequívocamente el diálogo al cual el marco pertenece. Este número divide una conexión en diferentes canales.

2. *Sesión* Las sesiones son las interacciones entre puntos AMQP a través de conexiones. Las sesiones pueden tener un estado asociado a ellas, en uno o en los dos puntos que participan en la interacción. Cada comando que publique un mensaje, cree una cola o selecione un modo de transacción, debe ser realizado dentro del contexto de una sesión. Las sesiones son la base sobre las que se basan el resto de componentes de AMQP.

3.2.2. Implementación AMQP Apache Qpid

Apache Qpid[1] es un MOM de código abierto, licenciado bajo los términos de Apache License Version 2.0. Apache Qpid (de ahora en adelante Qpid) implementa el estándar AMQP. Proveé de dos servidores AMQP, uno implementado en C++ (alta eficiencia, baja latencia y soporte RDMA) y otro en Java (completamente conforme con JMS, se ejecuta sobre cualquier plataforma Java).

Qpid también ofrece APIs para cliente de AMQP. Concretamente, para los siguientes lenguajes: C++, Java, C#, .Net, Ruby y Python. En este sentido, Qpid permite que las diferentes partes, escritas en cualquiera de estos lenguajes, que forman una aplicación distribuida, sean capaces de comunicarse entre ellas.

No está de más mencionar que cualquier cliente desarrollado usando una de las APIs de Qpid es capaz de comunicarse con otro cliente desarrollado usando otra API ofrecida por cualquier otra implementación de AMQP. Este cliente también es capaz de hablar con cualquier implementación de servidor de AMQP.

La figura 3.3 muestra un ejemplo de una aplicación distribuida heterógenea. En este ejemplo algunas aplicaciones, subscriptores, están escuchando ciertas colas, mientras otras aplicaciones, productores, producen mensajes y los envian a los intercambiadores para que sean enrutados a las colas. Cuando un mensaje es enrutado a una cola será, en algun momento, consumido por un subscriptor, hasta que esto ocurra el mensaje será almacenado en la cola del servidor. Un mensaje puede ser enviado por una aplicación escrita en cualquier lenguaje de programación y ser leído por otra aplicación escrita en un lenguaje diferente. Para que las comunicaciones sean posibles, solo es necesario que ambas aplicaciones compartan las mismas semánticas, esta es la gran ventaja que AMQP aporta a cualquier aplicación heterogénea.

[1] http://qpid.apache.org/

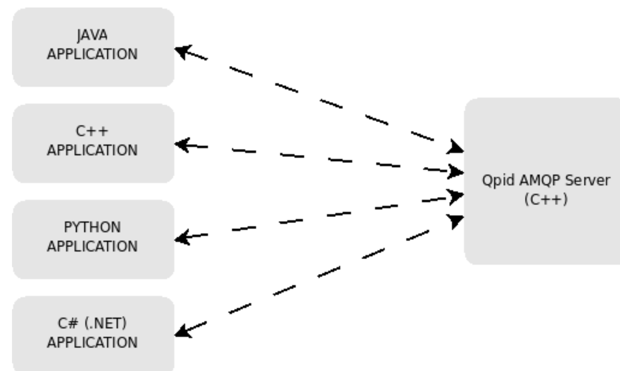

Figura 3.3: Servidor Qpid y aplicaciones heterógeneas

3.2.3. Arquitectura de la Comunicación

La figura 3.4 muestra una vista general de la arquitectura de la comunicación en Magentix2. Magentix2 está compuesto por un servidor AMQP (broker Qpid) o múltiples servidores federados. Los agentes Magentix2 actúan como clientes AMQP, se conectan al broker usando los diferentes APIs para clientes de Qpid, una vez conectados al broker son capaces de comunicarse unos con otros. Los agentes Magentix2 pueden estar localizados en cualquier lugar de Internet, para formar parte de la plataforma solo es necesario que conozcan el servidor donde está alojado el broker Qpid (o alguno de los brokers federados).

Magentix2 ofrece un API Java para facilitar el desarrollo de agentes. Este API permite a los programadores centrarse en la interacción entre agentes, sin tener que enfrentarse con el API Java de Qpid. El API ofrecido por Magentix2 no especifica ningún modelo o arquitectura específica de agente, por lo tanto, múltiples modelos y arquitecturas pueden coexistir e interaccionar entre ellos en el mismo sistema multiagente funcionando sobre Magentix2. Hasta este momento el API Magentix2 solo se encuentra escrito en Java, pero la existencia de múltiples APIs para cliente de Qpid en diversos lenguajes abre las puertas para que agentes escritos en múltiples lenguajes de programación formen parte de un mismo SMA. Además, cualquier implementación que siga los estándares AMQP y FIPA-ACL es interoperable con los agentes Magentix2. Los mensajes intercambiados entre los agentes siguen el estándar FIPA-ACL y son enviados a través de los estándares AMQP y FIPA-HTTP [21].

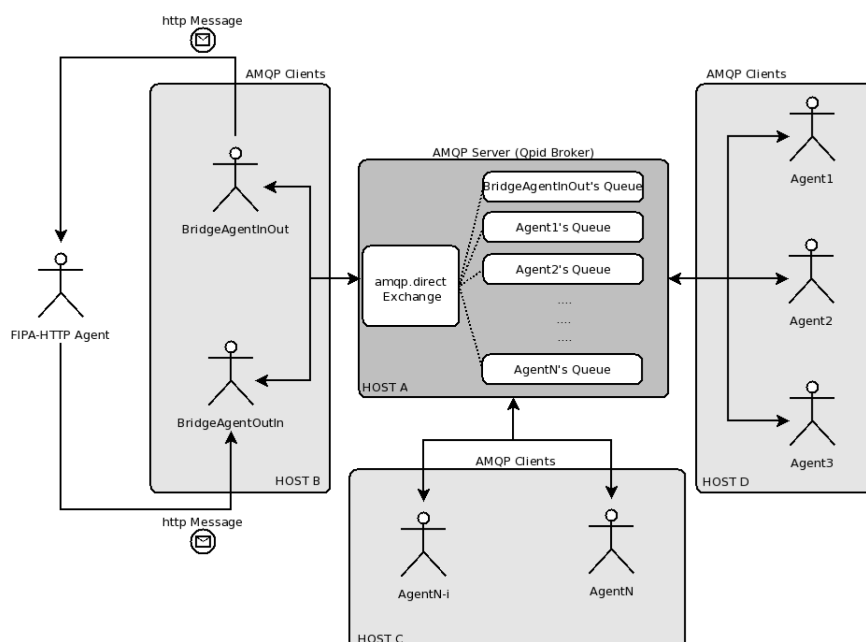

Figura 3.4: Arquitectura de la comunicación en Magentix2

3.2.3.1. FIPA-ACL sobre AMQP

Esta es la opción por defecto de envío de mensajes para los agentes Magentix2. La figura 3.5 muestra la superposición de estándares en la comunicación de Magentix2. Los agentes se comunican mediante mensajes FIPA-ACL encapsulados en mensajes AMQP. Intérnamente, cuando un mensaje FIPA-ACL está preparado para ser enviado, antes de enviarlo, el mensaje es serializado e incluido como contenido en un mensaje AMQP, acto seguido el mensaje es enviado al broker. Finalmente, cuando el mensaje es recibido por el otro agente, el mensaje es deserializado y formado otra vez como un mensaje FIPA-ACL. Todo este proceso es transparente al programador.

Cuando un agente arranca, el API ofrecido por Magentix2 se encarga de conectar el agente al broker Qpid. Posteriormente, se crea una sesión para el agente, a través de esta sesión, se crea una cola y esta se enlaza con el intercambiador por defecto *amqp.direct*, este intercambiador es creado por el propio broker cuando este arranca. El intercambiador *amqp.direct* es del tipo *direct*, como se ha explicado con anterioridad este tipo de intercambiadores enruta los mensajes basándose en una coincidencia exacta entre la clave de enrutamiento y la clave de enlace. Tanto el nombre de la cola como la clave de enlace toman el nombre del agente, de esta manera

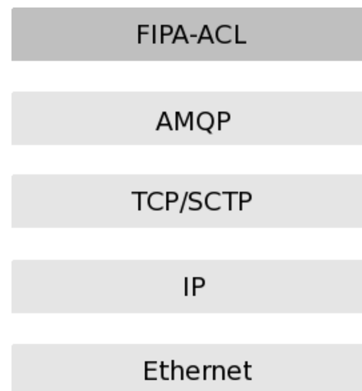

Figura 3.5: Pila de estándares de la comunicación en Magentix2

para comunicarse con un agente dado, solo es necesario enviar un mensaje al intercambiador *amqp.direct* usando como clave de enrutamiento el nombre del agente, como existirá una coincidencia exacta entre la clave de enrutamiento y la clave de enlace (ambas serán el nombre del agente), el mensaje será enrutado a la cola del agente.

Todos los agentes están subscritos a su cola de mensajes. Una subscripción es un elemento del estándar AMQP que permite al cliente recibir de forma asíncrona los mensajes que lleguen a una cola a la cual se encuentra subscrito. Por tanto, una vez el agente está subscrito a su cola, cada vez que un mensaje llegue a ella, el agente es notificado y se ejecutará una función de recepción de mensaje. Las colas de los agentes son privadas, esto es, solo el agente que creó la cola puede leer los mensajes que lleguen a la misma.

3.2.3.2. FIPA-ACL sobre FIPA-HTTP

Como se aprecia en la figure 3.4, Magentix2 implementa el protocolo de transporte de mensajes FIPA-HTTP [21] por medio de dos agentes especiales internos y propios de la plataforma: el agente BridgeAgentInOut y BridgeAgentOutIn. La función de estos agentes es permitir al resto de agentes de la plataforma comunicarse con agentes en plataformas diferentes que cumplan con los estándares de comunicación FIPA. El uso de un protoclo u otro (HTTP o AMQP) es transparente al agente, solo es necesario especificar como destinatario el nombre del agente, si el destinatario se encuentra dentro de la plataforma Magentix2 los agentes Bridge no entrarán en juego, en caso contrario el agente BridgeAgentInOut se encargará de enviar el mensaje sobre FIPA-HTTP al agente externo. Los mensajes entrantes serán tratados por el agente BridgeA-

gentOutIn y su funcion es la contraria a la del BridgeAgentInOut, es decir, transforma mensajes FIPA-HTTP en mensajes FIPA sobre AMQP.

Capítulo 4

Conversaciones de los Agentes en Magentix2

Tal y como se afirma en [24], las conversaciones en curso entre agentes normalmente siguen patrones típicos. En estos casos, ciertas secuencias de mensajes son predecibles y, en cualquier momento de la conversación, se espera la recepción de ciertos mensajes. Estos patrones típicos de mensajes intercambiados son llamados protocolos de interacción. En Magentix2 cada conversación entre agentes representa una secuencia de mensajes, los cuales corresponden a un protocolo de interacción específico.

Los protocolos de interacción permiten a los agentes seguir secuencias predeterminadas de mensajes. Aún así, en sistemas abiertos estas secuencias pueden requerir algunas modificaciones mientras el protocolo está en funcionamiento. Se presenta en este documento un soporte a nivel de plataforma multiagente para definir y ejecutar protocolos de interacción que pueden modificarse en tiempo de ejecución.

De acuerdo con las especificaciones de FIPA, un protocolo de interacción es generalmente representado por dos roles de agente, cada uno con su propio comportamiento: *initiator*, el cual corresponde al agente que inicia el protocolo de interacción, y *participant*, el cual corresponde al agente (o agentes) que participan en la conversación, pero no la iniciaron. Por tanto, los mensajes que correspondan a un protocolo de interacción siempre son intercambiados entre agentes jugando dichos roles.

Como ejemplo de protocolo de conversación podemos referirnos a un situación de la vida real entre un taxista y un cliente. Cuando el cliente entra al taxi no necesita preocuparse sobre todas las posibles afirmaciones sobre direcciones que se pueden hacer. En vez de esto, el taxista

espera a que el cliente le indique una dirección de destino, después conducirá hasta ella y por último pedirá el dinero el cliente por la carrera. La secuencia esperada de mensajes en este escenario representa un protocolo de interacción específico, podríamos llamarlo *Protocolo de Interacción del Taxi*. En este protocolo el cliente y el taxista juegan roles diferentes, el primero sería el iniciador y el taxista el participante.

Desde el punto de vista del paradigma de agentes, el taxista podría atender más de una petición simultáneamente. Por tanto existirían varias conversaciones ejecutando el Protocolo de Interacción del Taxi entre el taxista y varios clientes.

Se ha elaborado una propuesta [25] para ayudar a los programadores a desarrollar protocolos de interacción, este API permite a los programadores crear conversaciones siguiendo protocolos de conversación y a modificar estos de forma dinámica. En las próximas secciones se muestran los diferentes componentes de la propuesta y su utilidad. La explicación de cada uno de los componentes se apoya en un ejemplo práctico basado en el protocolo de interacción explicado anteriormente entre el taxista y el cliente.

4.1. Interacción y Agentes

Los protocolos de interacción (IP) son muy usados para especificar patrones de comunicación entre agentes. Algunas MAPs [1; 3; 4; 10] ofrecen soporte para ejecución de IPs. Estas implementaciones normalmente usan la aproximación de FIPA en el cual una interacción se representa como una secuencia de mensajes intercambiados entre un agente con el rol inicador y uno o más con el rol de participante. Todas estas funcionalidades solo consideran IPs predefinidos y por tanto sus especificaciones no pueden se modificadas en tiempo de ejecución. La aproximación presentada en este trabajo se basa en permitir cambios dinámicos en estos IPs.

Existen actualmente diferentes técnicas y metodologías para el tratamiento de las conversaciones por parte de los agentes. Una de las técnicas más populares son las Instituciones Electrónicas (IE) [35]. El concepto básico de las IE es que las interacciones humanas siempre están guiadas por convenciones formales o informales. Las IE son un medio para representar esas convenciones y tratar las interacciones entre agentes como interacciones humanas. La definición formal de las IE es similar a la de una producción teatral, donde los agentes son *actores* que interpretan uno o más *roles* en la institución. Las interacciones entre agentes se estructuran mediante el uso de *escenas* en las cuales grupos de agentes interactúan. Dentro de una escena todos los agentes participantes siguen un guión el cual guía sus interacciones.

Esta aproximación al tratamiento de las interacciones entre agentes presenta algunos problemas, el primero de ellos es que los agentes deben conocer los estados internos de la institución, es decir, deben conocer de antemano todas las escenas en las que participarán. Otro problema es que la definición de la IE ha de ser estática, asumiendo que se cubren todos los espacios de la conversación. El último problema atañe a la sincronización de agentes dentro de la IE, para garantizar esta sincronización es necesario recurrir al uso de agentes administrativos, esta solución es contraria al principio por el cual los agentes autónomos deben ser autocontenidos.

Otro enfoque de la interacción entre agentes se basa en protocolos flexibles [39]. En esta propuesta los protocolos de conversación se crean conforme avanza la conversación entre los agentes, para esto los agentes comunican el protocolo que se va a usar a los otros agentes por medio de mensajes, el protocolo es especificado mediante un lenguaje especial creado para este propósito. Este lenguaje permite la definición de protocolos como máquinas de estados similares a los modelos usados para las IE. Esta propuesta presenta algunos problemas, el primero de ellos es que el lenguaje usado para la definición no sigue ningún estándar por lo que limita su uso en un SMA abierto. Por otra parte, esta propuesta no contempla la posibilidad de que el agente mantenga conversaciones concurrentes, en entornos con muchos agentes y donde se busca la eficiencia es importante que el agente sea capaz de comunicarse con múltiples agentes simultáneamente.

Artikies y otros [8] presentan un armazón para especificar sistemas abiertos desde la perspectiva de la organización y no desde el agente. Representan los SMA abiertos como sistemas normativos especificando que está permitido, prohibido y que es obligatorio. En este armazón la especificación de protocolos es llevada a cabo en tiempo de diseño aunque puede ser modificada durante la ejecución, debido a que las reglas que gobiernan el protocolo pueden cambiar. De cualquier manera esta visión restringe el rango de aplicación a los sistemas normativos.

De acuerdo al trabajo previo, en este trabajo se propone un soporte a nivel de plataforma multiagente que es independiente del SMA desarrollado sobre él. La propuesta aquí mostrada intenta ofrecer un soporte a nivel de plataforma para gestionar interacciones complejas entre agentes que pueden cambiar dinámicamente. A continuación se detallan los elementos de nuestra propuesta y su función.

4.2. Procesadores de Conversaciones

Tal y como se ha explicado al inicio del capítulo, una conversación representa un a secuencia de mensajes entre un iniciador y uno o más participantes. Dependiendo del rol que un agente esté jugando en la conversación, ciertas acciones específicas son llevadas a cabo en cada paso de la conversación. Un Procesador de Conversaciones (*CProcessor*) está a cargo de llevar a cabo las acciones y manejar los mensajes enviados y recibidos en cada paso de la conversación. Por lo tanto es este componente el que decide en cualquier punto de la conversación cual es el próximo paso de acuerdo al protocolo de interacción que la conversación está siguiendo.

Dependiendo del rol que un agente tenga en una conversación y del momento en el que se encuentre según el transcurso de ésta, existen ciertas acciones específicas permitidas. Además, ciertas acciones específicas sólo pueden seguir a otras. Para representar toda esta información un *CProcessor* usa un grafo dirigido compuesto por nodos y arcos. Por una parte, los nodos representan todos los posibles estados en los que un rol puede hayarse durante una conversación, por otra parte los arcos representan una función de transición entre nodos, esto es, que estados pueden ser alcanzados desde cualquier estado dado de la conversación.

En el escenario del taxista un nodo podría representar el momento durante el cual el taxista espera que el cliente le indique la dirección de destino. Por otra parte, un arco podría representar la transición desde el anterior estado, donde el taxista espera a la dirección, al momento en el que puede pasar a conducir hacia la dirección de destino.

Se definen varios tipos de estados para representar las diferentes acciones que pueden ser llevadas a cabo durante una conversación. Los tipos son los siguientes:

- *Begin*: Este estado representa que un rol empieza la conversación.
- *Final*: Este estado representa que un rol termina la conversación.
- *Action*: Este estado representa cualquier acción realizada por el rol diferente a un acto del habla. Un ejemplo podría ser una acción realizada por un agente a raíz de una petición por parte de otro agente (conducir hasta la dirección de destino).
- *Send*: En este estado el rol envía un mensaje.
- *Wait*: Cuando un rol alcanza a este estado, la conversación se detiene hasta que un nuevo mensaje perteneciente a la conversación llega al agente, entonces, dependiendo del tipo

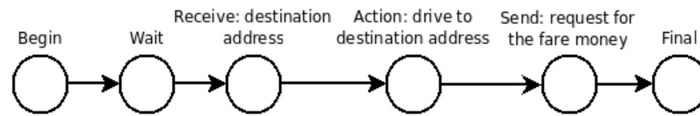

Figura 4.1: Grafo para el rol de taxista (participante) en el *Protocolo de Interacción del Taxi*

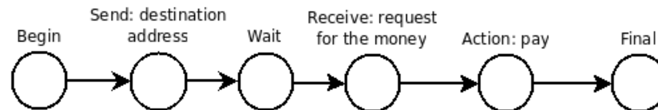

Figura 4.2: Grafo para el rol de cliente (iniciador) en el *Protocolo de Interacción del Taxi*

de mensaje que ha llegado, el estado del rol en la conversación pasa a ser un cierto estado siguiente del tipo *Receive*. El tipo de un mensaje es definido por su cabecera.

- *Receive*: Un estado de este tipo debe estar precedido por uno del tipo *Wait*. En este estado el rol recibe un mensaje. Cada estado *Receive* acepta un tipo específico de mensajes con unas cabeceras específicas.

- *Initiate*: En este estado el rol inicia una nueva subconversación. En la sección 4.4 se profundizará en este tipo de estado y las conversaciones anidadas.

- *Participate*: Este tipo de estado es un tipo especial de estado *Receive* donde el rol empieza una nueva subconversación cuando recibe un mensaje apropiado. Al igual que el estado *Initialize* se explicará con más profundidad en la sección 4.4.

Con todo esto, podemos resumir que los protocolos de interacción pueden ser definidos en términos de estados y transiciones entre ellos. Para el caso del *Protocolo de Interacción del Taxi*, el grafo asociado al rol de conductor se muestra en la figura 4.1, mientras que el grafo asociado al cliente se muestra en la figura 4.2.

También se definen tipos de estado para excepciones, estos estados no son añadidos al grafo por el programador si no que están presentes en todo grafo. Todo estado normal está conectado a cada uno de los estados de excepción, no es necesario declarar estas transiciones en el momento de crear el grafo, esto se hace tácitamente por el propio *CProcessor*. De cada tipo de estado de excepción solo hay uno en cada grafo. Cada tipo ofrece un comportamiento por defecto, aunque este comportamiento puede ser modificado y adaptado por el programador. Los diferentes tipos de estado de excepción son:

- *Cancel*: Cuando la conversación recibe un mensaje para que termine de forma inesperada este estado trata esa terminación.
- *Not Accepted Messages*: Cuando un *CProcessor* que se encuentra en un estado *Wait*, recibe un mensaje con su identificador de conversación y no existe ningún estado *Receive* capaz de tratar ese tipo de mensaje se produce una excepción que es tratada por este estado.
- *Sending Errors*: Si se produce un error de envío de mensajes en un estado *Send* este estado trata la excepción.

Todos los tipos de estado poseen un método que será ejecutado por el *CProcessor*. Este método es implementado por el programador para cada una de las instancias de estado. Dependiendo del tipo este método recibe y devuelve unos parámetros diferentes, por ejemplo, en el caso de un estado del tipo *Send* el método ha de devolver un objeto del tipo mensaje que será enviado por el agente, en la sección 4.6 se explica con más profundidad como se implementan y funcionan los métodos de los estados.

Ciertos protocolos de interacción necesitan asignar plazos de espera para que un agente involucrado en la conversación conteste con un mensaje. Un ejemplo típico de esta situación sería el protocolo FIPA Contract-Net[22], en este protocolo el agente iniciador envía a cada participante un mensaje pidiéndole que le envíe una propuesta, el agente iniciador tendrá un límite de tiempo máximo para esperar a la respuesta de todos los agentes participantes, si algún agente falla en cumplir este plazo, es de esperar que el agente participante no tenga en cuenta su propuesta. Para este própósito existen diferentes opciones a la hora de crear un estado del tipo *Wait*, las cuales son:

- Espera ilimitada: El rol que alcance este estado esperará ilimitadamente hasta que un mensaje sea asignado a la conversación. El uso de este tipo de estado *Wait* solo es recomendable en casos donde se esté seguro de que el agente recibirá una respuesta.
- Espera simple: En este caso el agente espera en este estado hasta que se alcanza un plazo o un mensaje sea asignado al *CProcessor*, lo que ocurra antes. Posteriormente, si en el transcurso de la conversación, el agente vuelve a este estado, el plazo se reinicia, por ejemplo si el estado tiene un plazo de 3 segundos, este se tendrá que cumplir cada vez que el agente vuelva a este estado.

- Espera absoluta: En este tipo, el agente espera en este estado hasta que se alcanza un plazo o un mensaje sea asignado al *CProcessor*, lo que ocurra antes. Posteriormente, si en el transcurso de la conversación el agente vuelve a este estado el plazo no se reinicia. Por ejemplo, si un estado tiene un plazo de 3 segundos, el agente, una vez alcanza ese estado, espera como máximo 3 segundos, si pongamos que en 2 segundos recibe un mensaje, entonces el agente abandona el estado de espera, si más tarde vuelve a este estado, el plazo será de 1 segundo sólo.

- Espera cíclica: En este caso el agente espera en este estado hasta que se alcanza un plazo o un mensaje sea asignado al *CProcessor*, lo que ocurra antes. Posteriormente, si en el transcurso de la conversación el agente vuelve a este estado, el plazo no se reinicia, si no que se calculará el menor plazo posterior al tiempo actual.

En todos los estados tipo *Wait* que tengan un plazo es necesario añadir un estado enlazado al estado *Wait* y que sea el estado destino en caso de que el plazo se cumpla. Es decir, este estado se ejecutará cuando se cumpla el plazo del estado del tipo *Wait* que lo precede.

Cada ejecución de un *CProcessor* se realiza en un hilo diferente al de los demás *CProcessor* y del propio agente. Este hilo es extraído de un *pool* de hilos que el agente posee en su creación. Cada vez que un *CProcessor* alcanza un estado *Wait* el hilo es devuelto al *pool*, de manera que no se consumen recursos mientras la conversación espera a que llegue un mensaje para poder continuar, una vez llega un nuevo mensaje y es asignado a una conversación en espera, se le asigna un nuevo hilo del *pool*. Usar hilos concurrentes posibilita que el agente mantenga múltiples conversaciones simultaneas, incluso mientras el propio agente realiza otras acciones diferentes a conversar.

El grafo asociado a un *CProcessor* puede ser modificado dinámicamente durante el tiempo de ejecución. Las modificaciones pueden afectar indistintamente a los estados o arcos. Por tanto, un *CProcessor* puede adaptarse a cualquier cambio mientras el protocolo de interacción estás siendo ejecutado. Por ejemplo, siguiendo con el ejemplo del taxi supongamos que a partir de ahora, las carreras de los taxis serán calculadas y abonadas por el cliente antes del viaje. Este cambio en el protocolo puede ser realizado dinámicamente. Concrétamente, el estado *Send* pidiendo el dinero será movido justo después del estado donde se recibe la petición con la dirección de destino. El nuevo grafo que dirijirá el protocolo de conversación del rol taxista se muestra en la figura 4.3. Hay que tener en cuenta que un cambio en el protocolo de interacción de uno de los roles de la conversación puede implicar cambios en el protocolo de interacción

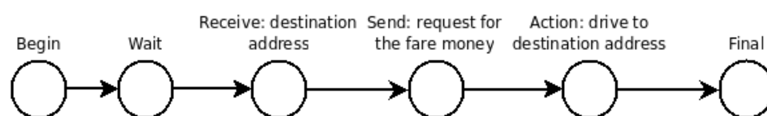

Figura 4.3: Protocolo del taxista modificado dinámicamente

del otro rol. En nuestro ejemplo el protocolo asociado al cliente debería también ser modificado de acuerdo a los cambios realizados en el protocolo asociado al taxista.

4.3. Fábricas de Conversaciones

Las fábricas de conversación (*CFactories*) están a cargo de empezar las conversaciones. Toda *CFactory* tiene asociado un *CProcessor* el cual puede gestionar un protocolo de interacción concreto. Cuando una nueva conversación necesita ser creada la *CFactory* específica inicia un *CProcessor* el cual será responsable de ejecutar la conversación.

Existen dos tipos de *CFactory*, el uso de uno u otro depende del rol que el agente tenga en las conversaciones que la *CFactory* creará. Si el agente tiene el rol iniciador en la conversación, tendrá que usar una *CFactory* del tipo iniciador. Por el contrario, si el rol del agente es el de participante, entonces deberá usar una *CFactory* del tipo participante. Una *CFactory* del tipo iniciador, empieza conversaciones sin necesidad de eventos o estímulos externos. En cambio, una *CFactory* participante empieza una nueva conversación cuando un mensaje apropiado llega al agente. Para decidir si un mensaje es apropiado para empezar o no una nueva conversación, y por tanto un *CProcessor*, las *CFactories* usan un filtro de mensajes. Este filtro especifíca unos ciertos valores para los campos de la cabecera de los mensajes. Cuando un agente recibe un mensaje, su cabecera es comparada con los valores que dicta el filtro de cada una de las *CFactories*, aquella cuyo filtro coincida con los valores del mensaje será la encargada de inicializar un nuevo *Cprocessor*.

Los filtros se definen mediante expresiones lógicas. Los operadores lógicos que estas expresiones aceptan son AND, OR y NOT, también aceptan parentización. Estas expresiones sirven para comprar cualquier cabecera definida por FIPA con cualquier valor, además es posible comparar cualquier cabecera no estándar añadida al mensaje. Un ejemplo de filtro sería el siguiente:

```
performative= INFORM OR (performative= REQUEST AND purpose= buy)
```

Este filtro solo aceptaría mensajes con performativa *Inform* o con performativa *Request* y *purpose* (cabecera no perteneciente al estándar FIPA) igual a *buy*. El filtro de los estados del tipo *Receive* funciona exactamente de la misma manera.

Cada conversación tiene un identificador único el cual es asignado por el *CProcessor* en el caso de que el rol del agente en la conversación sea el de iniciador, en el caso de que sea participante, el identificador de conversación vendrá dado por el primer mensaje que llegó al agente y inició la conversación, por lo que, tanto el *CProcessor* del agente iniciador como el del participante tendrán el mismo identificador de conversación. Cuando un mensaje llega al agente antes de compararlo con los filtros de las *CFactories* participantes se comprueba si el identificador de conversación del mensaje (es un campo de la cabecera del mensaje) es el mismo que el de alguna conversación en curso, en caso afirmativo el mensaje es asignado diréctamente a esa conversación, en caso contrario seguirá el proceso anteriormente explicado y su cabecerá se comparará con todos los filtros de las *CFactories* participantes del agente.

Cuando se recibe un mensaje cabe la posibilidad de que ninguna *CFactory* sea capaz de tratar el mensaje, en otras palabras, el agente recibe un mensaje cuya cabecera no encaja con ningún filtro de ninguna *CFactory* participante y el identificador de conversación no es de ninguna conversación que el agente está manteniendo actualmente. En ese caso el mensaje es asignado a un *CFactory* llamada *DefaultCFactory*, esta fábrica es proporcionada por el API y se encarga del tratamiento de todos los mensajes que son intratables por el resto de fábricas del agente. Aunque esta fábrica tiene un comportamiento por defecto, es posible para el programador modificarlo y adaptarlo a sus necesidades.

En nuestro ejemplo, el taxista podría ser visto como un agente con una *CFactory* participante. El filtro de esta fábrica solo aceptaría mensajes cuya performativa fuese *Request*. Por otro lado el cliente puede ser visto como un agente con una *CFactory* iniciadora con un *CProcessor* cuyo primer estado sería uno de envío y el mensaje enviado por este estado tendría performativa *Request* (de esta manera sería aceptado por la *CFactory* del taxista) y como contenido la dirección de destino.

4.4. Conversaciones Anidadas

Algunas conversaciones pueden necesitar crear otras conversaciones llamadas subconversaciones. De esta manera, un protocolo de interacción puede estar compuesto de subprotocolos.

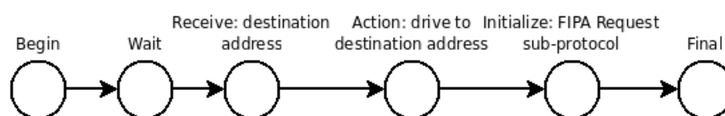

Figura 4.4: Protocolo del taxista usando subprotcolos

El API propuesto ofrece dos tipos diferentes de subprotocolo, síncrono y asíncrono. En el caso de subprotocolos síncronos, la conversación madre espera a que la subconversación acabe para continuar con su ejecución. Por el contrario, en el caso de un subprotocolo asíncrono, la conversación madre no espera que la subconversación termine para continuar con su ejecución.

Para poder iniciar subconversaciones existen dos tipos especiales de estado: *Initiate* y *Participate*. El primero inicia una subconversación tan pronto la conversación llega a ese estado. El segundo, a diferencia del primero, inicia una subconversación tras recibir un mensaje apropiado. De la misma manera que las fábricas participantes esperan a mensajes que pasen su filtro para iniciar nuevos *CProcessors*, un estado *Participate*, que siempre ha de estar precedido por uno del tipo *Wait*, inicia una subconversación cuando la conversación recibe un mensaje y este pasa el filtro del estado.

Cuando una subconversación síncrona arranca se le asigna el mismo identificador de conversación que el de la conversación que la creó. Si la subconversación es asíncrona, se le asigna un nuevo identificador de conversación.

La figura 4.4 muestra el ejemplo del taxi usando subprotocolos. El estado ”Send: request for the fare money“ ha sido substituido por uno del tipo *Initiate* el cual inicia una subprotocolo síncrono. Este subprotocolo es el protocolo de interacción FIPA Request [23]. En la figura 4.5 se puede observar este protocolo modelado según esta propuesta para el tratamiento de conversaciones. En este subprotocolo el taxista tiene el rol de iniciador (solicita el dinero) y el cliente el de participante (recibe la petición del dinero). Hay que tener en cuenta que el *CProcessor* del cliente también necesitará ser modificado para responder al subprotocolo FIPA Request iniciado por el taxista.

Cada *CProcessor* tiene capacidad para almacenar información interna, esta información puede ser accedida en cada estado del grafo y por cada conversación anidada a la conversación que el *CProcessor* está tratando. Esto es útil en casos en los que una subconversación necesita conocer algún dato obtenido en la conversación del nivel superior. La información almacenada en un *CProcessor* debería ser aquella que esté intímamente relacionada con la conversación.

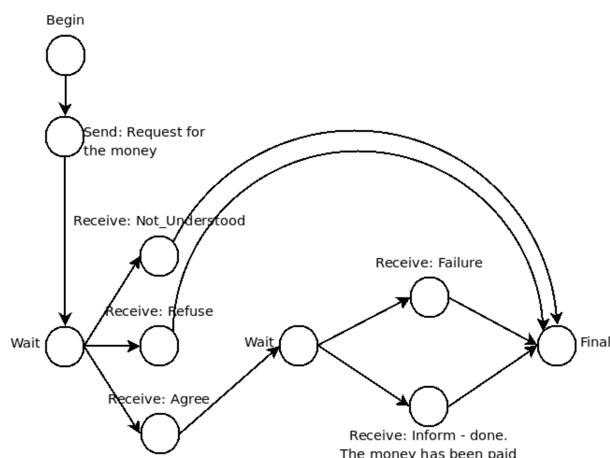

Figura 4.5: Protocolo de interacción FIPA Request para el rol iniciador

4.5. Visión Global de un Agente Conversacional

Una vez todos los componentes del modelo propuesto han sido explicados es posible mostrar una visión global de un agente Magentix2 y como este crea y destruye nuevas conversaciones. La figura 4.6 (subdividida en cuatro figuras) muestra todo el proceso de creación de un nuevo *CProcessor* y como el agente mantiene varias conversaciones simultáneas por medio de las *CFactories* y los *CProcessors*.

En la figura *A* de la visión global del agente vemos al agente con tres *CFactories*, dos de ellas son del tipo participante mientras la tercera es del tipo iniciador. En el momento mostrado por la figura, *Initiator Factory 1* ya ha creado un *CProcessor* cuyo identificador de conversación es *conv1*, este identificador de conversación le ha sido dado por el propio *CProcessor*. Esta conversación se mantiene en ejecución enviando y recibiendo mensajes (las flechas representan los mensajes entrantes y salientes) en paralelo con cualquier otra conversación que se cree dentro del agente en el futuro. En la siguiente figura (*B*), el agente recibe un mensaje cuya performativa es *Inform* y cuyo identificador de conversación es *conv2*, como no existe ninguna conversación activa con dicho identificador, las cabeceras del mensaje son comparadas con los filtros de las fábricas participantes, en este ejemplo son *Participant Factory 1* con un filtro que sólo acepta mensajes con performativa *Inform* y *Participant Factory 2* capaz de tratar sólo mensajes con performativa *Request*. Es por tanto *Participant Factory 1* la encargada de crear un nuevo procesador de conversaciones capaz de tratar el mensaje recibido, este hecho se puede observar en la figura *C*. Este nuevo procesador es independiente de cualquier otro que ya tenga

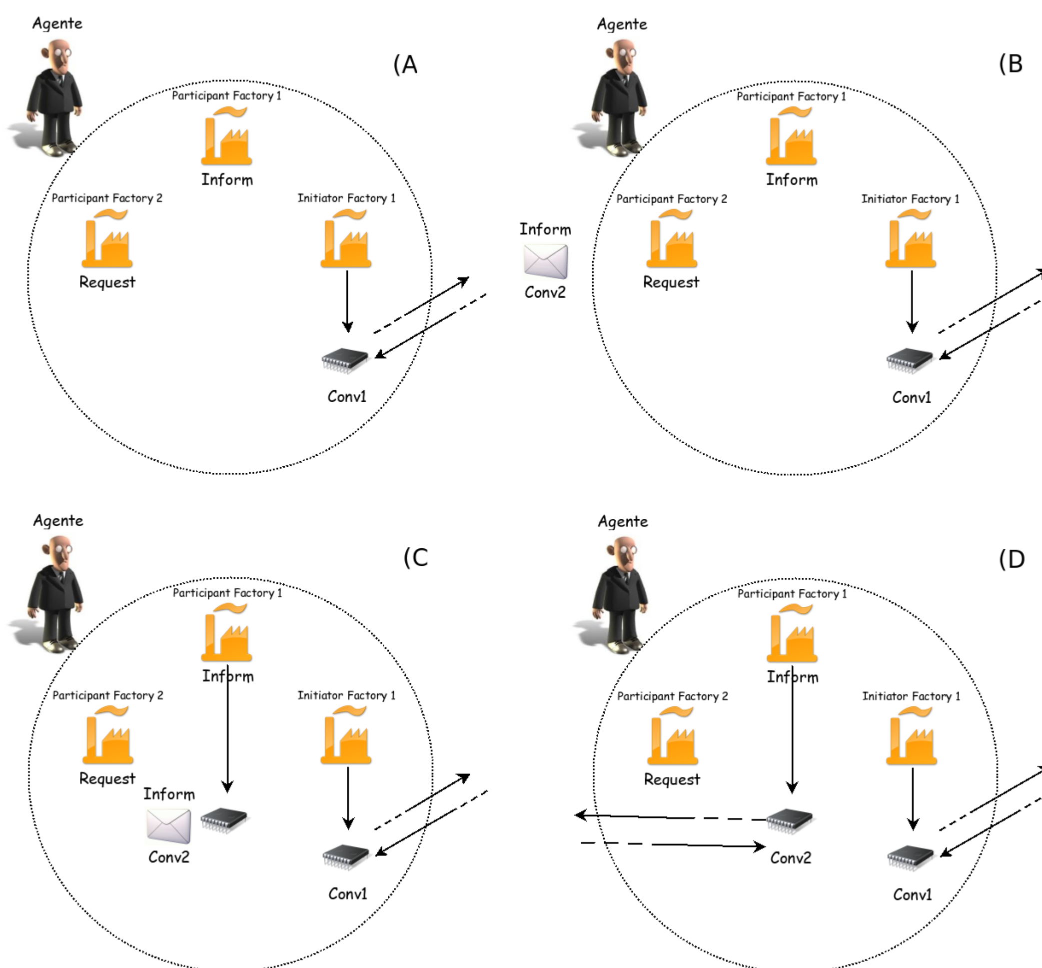

Figura 4.6: Visión global de un agente

el agente y será el encargado de tratar todos los futuros mensajes con identificador de conversación *conv2*. En la figura *D* podemos observar el agente como maneja dos conversaciones simultáneamente por medio de dos procesadores de conversación.

En el caso en el que el agente reciba un mensaje para el cual no tiene ninguna *CFactory* con un filtro que case con la cabecera del mensaje o una conversación activa con el mismo identificador de conversación que el del mensaje entrante, este mensaje será tratado por la *CFactory* por defecto. Esta *CFactory* se encuentra definida tácitamente en todo agente conversacional. El comportamiento por defecto del *CProcessor* creado por esta *CFactory* es desechar el mensaje. Este comportamiento puede ser modificado por el programador del agente para cumplir sus

necesidades. En la figura 4.7 y figura 4.8 se puede observar esta funcionalidad.

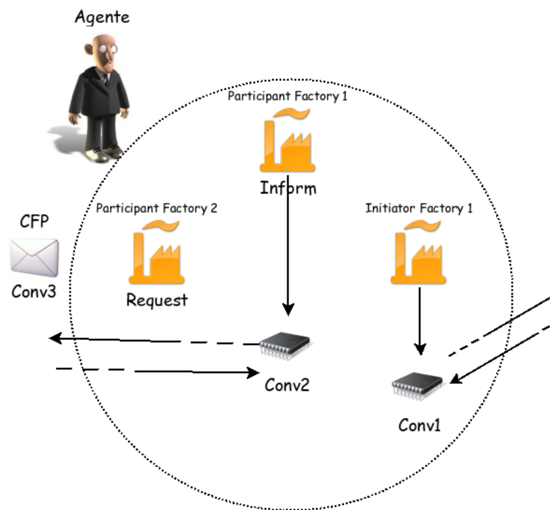

Figura 4.7: Funcionamiento de la *CFactory* por defecto 1

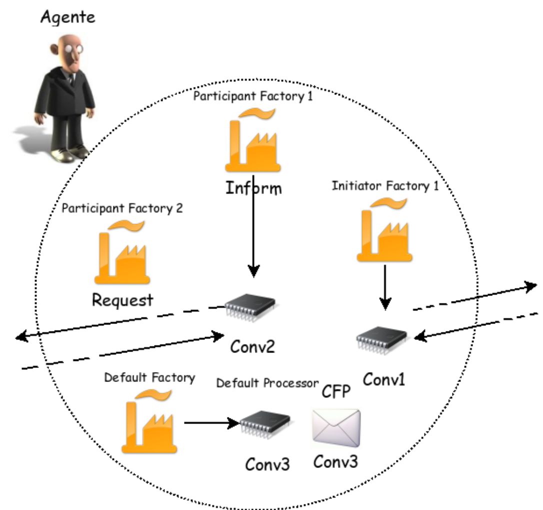

Figura 4.8: Funcionamiento de la *CFactory* por defecto 2

4.6. Ejemplo de Implementación de Agentes Magentix2

En base al modelo presentado se ha desarrollado una API que permite a los agentes crear y participar conversaciones siguiendo este modelo. A continuación se muestran diferentes ejemplos de implementación de agentes que hacen uso de este API. En las siguientes secciones se empieza por lo más básico, crear un agente conversacional muy sencillo, acabando con el uso de plantillas de protocolos, las cuales ayudan a usar protocolos complejos de una manera sencilla.

4.6.1. Ejemplo de un Agente Conversacional Básico

Para crear un *CAgent* es necesario crear una nueva clase que herede de la clase *CAgent*, esta clase tiene dos métodos abstráctos que el programador ha de implementar. En el siguiente código podemos ver el esqueleto de un nuevo *CAgent* con el que vamos a implementar un agente *HelloWorld* que nos ayudará a entender los siguientes ejemplos más complejos.

```
1 class HelloWorldAgentClass extends CAgent {
```

```
2
3   public HelloWorldAgentClass(AgentID aid) throws Exception {
4    super(aid);
5   }
6
7   protected void Initialize(CProcessor myProcessor, ACLMessage
       welcomeMessage) {
8    System.out.println(myProcessor.getMyAgent().getName() + ": the
        welcome message is " + welcomeMessage.getContent());
9    System.out.println(myProcessor.getMyAgent().getName() + ":
        inevitably I have to say hello world");
10   myProcessor.ShutdownAgent();
11  }
12
13  protected void Finalize(CProcessor myProcessor, ACLMessage
       finalizeMessage) {
14   System.out.println(myProcessor.getMyAgent().getName() + ": the
        finalize message is " + finalizeMessage.getContent());
15  }
16 }
```

Todo *CAgent* empieza su existencia conversando con la plataforma. Esta conversación es tratada por un *CProcessor*, el cual es pasado como parámetro al método *Initialize* y *Finalize*. Por lo tanto todos los agentes empiezan ya con un *CProcessor* el cual ejecuta el método *Initialize* y el *Finalize*. Además la plataforma asigna un mensaje de bienvenida y otro de despedida al *CProcessor* inicial, estos mensajes son pasados como parámetros en los métodos *Initialize* y *Finalize* respectivamente.

Este agente solo muestra por pantalla el mensaje de bienvenida que la plataforma envía al agente y por último muestra el mensaje de despedida enviado también por la plataforma.

4.6.2. Ejemplo de uso de las Fabricas de Conversaciones

Una vez mostrado el esqueleto básico de un *CAgent* podemos pasar a explicar el funcionamiento de las *CFactories*. Las *CFactories* de un agente se definen en el método *Initialize*. En este ejemplo crearemos una *CFactory* con nombre *talk*, su filtro de mensajes solo aceptará aquellos que tengan como performativa *Request*, por último, la *CFactory* solo podrá mantener en activo un único *CProcessor* simultáneamente. En el siguiente código se muestra como realizar estas acciones para implementar el agente taxista de los ejemplos anteriores, la *CFactory*

que iniciará las conversaciones con los clientes y la plantilla a partir de la cual se crearán los *CProcessors* que manejarán estas conversaciones.

```
1  protected void Initialize(CProcessor myProcessor, ACLMessage
       welcomeMessage) {
2    ACLMessage template;
3    template = new ACLMessage(ACLMessage.REQUEST);
4    CFactory talk = new CFactory("TALK", template, 1, this);
5    ......
6  }
```

Una vez se ha creado la *CFactory* pasamos a crear la plantilla de *CProcessor* a partir de la cual la *CFactory* creará futuros *CProcessors*. Primero, tenemos que especificar el grafo y cada uno de sus estados, en el próximo código podemos ver como se realiza:

```
1  protected void Initialize(CProcessor myProcessor, ACLMessage
       welcomeMessage) {
2    ......
3    //BEGIN state
4    BeginState BEGIN = (BeginState) talk.cProcessorTemplate().getState
         ("BEGIN");
5    class BEGIN_Method implements BeginStateMethod {
6      public String run(CProcessor myProcessor, ACLMessage msg) {
7        return "WAIT";
8      };
9    }
10   BEGIN.setMethod(new BEGIN_Method());
11
12   //WAIT state
13   talk.cProcessorTemplate().registerState(new WaitState("WAIT", -1))
         ;
14   talk.cProcessorTemplate().addTransition(BEGIN, WAIT);
15
16   //RECEIVE state
17   ReceiveState RECEIVE = new ReceiveState("RECEIVE");
18   class RECEIVE_Method implements ReceiveStateMethod {
19     public String run(CProcessor myProcessor, ACLMessage
           messageReceived) {
20       CabDriver myAgent = (CabDriver) myProcessor.getMyAgent();
21       myAgent.destination = messageReceived.getContent();
```

```
22          return "ACTION";
23        }
24      }
25      RECEIVE.setAcceptFilter(new ACLMessage(ACLMessage.REQUEST));
26      RECEIVE.setMethod(new RECEIVE_Method());
27      talk.cProcessorTemplate().registerState(RECEIVE);
28      talk.cProcessorTemplate().addTransition(WAIT, RECEIVE);
29
30      //ACTION state
31      ActionState ACTION = new ActionState("ACTION");
32      class ACTION_Method implements ActionStateMethod {
33        public String run(CProcessor myProcessor) {
34          CabDriver myAgent = (CabDriver) myProcessor.getMyAgent();
35          myProcessor.myAgent.driveToDestination(myAgent.destination);
36          return "SEND";
37        }
38      }
39      talk.cProcessorTemplate().registerState(ACTION);
40      talk.cProcessorTemplate().addTransition(RECEIVE, ACTION);
41
42      //SEND state
43      SendState SEND = new SendState("SEND");
44      class SEND_Method implements SendStateMethod {
45        public String run(CProcessor myProcessor, ACLMessage
             messageToSend) {
46          CabDriver myAgent = (CabDriver) myProcessor.getMyAgent();
47          messageToSend.setPerformative(ACLMessage.REQUEST);
48          messageToSend.setContent(myAgent.calculateFare());
49          messageToSend.setReceiver("customer");
50          return "FINAL";
51        }
52      }
53      talk.cProcessorTemplate().registerState(SEND);
54      talk.cProcessorTemplate().addTransition(ACTION, SEND);
55
56      // FINAL state
57      FinalState FINAL = new FinalState("FINAL");
58      class FINAL_Method implements FinalStateMethod {
59        public void run(CProcessor myProcessor, ACLMessage messageToSend
             ) {
60          messageToSend.setContent("Done");
```

```
        }
    }
    FINAL.setMethod(new FINAL_Method());
    talk.cProcessorTemplate().registerState(FINAL);
    talk.cProcessorTemplate().addTransition(SEND, FINAL);
    ........
}
```

Básicamente, para cada uno de los estados, especificamos un método que se ejecutará cuando el protocolo de interacción llegue a ese estado. Una excepción a esto es el estado *Wait* el cual no tiene método y como ya se ha explicado en la sección 4.2 espera durante un plazo o hasta que llegue un mensaje. En este caso el estado *Wait* esperará indefinidamente a que llegue un mensaje (el plazo pasado es -1 lo cual es interpretado como infinito). Cuando un estado es creado es necesario registrarlo en el *CProcessor* e indicar todas las transiciones del resto de estados al nuevo estado.

Por último, tenemos que añadir la *CFactory* a la lista de *CFactories* del agente, al añadir una *CFactory* hay que especificar si el agente actuará como iniciador o participante en las conversaciones tratadas por la nueva factoría, en este ejemplo la factoría será del tipo participante por lo que usamos el método *addFactoryAsParticipant* como se puede apreciar en el código mostrado a continuación:

```
protected void Initialize(CProcessor myProcessor, ACLMessage
     welcomeMessage) {
    ........
    this.addFactoryAsParticipant(talk);
}
```

4.6.3. Ejemplo de uso de las Plantillas de Protocolos

Muchas veces cuando creamos un SMA diseñamos las interacciones entre los agentes del sistema de tal manera que sigan protocolos de interacción estándar. Un conjunto muy utilizado de estos protocolos de interacción lo encontramos en el estándar FIPA. Para facilitar la implementación de estas interacciones el API de CAgents ofrece plantillas de protocolo, de tal manera que solo es necesario implementar métodos de unos pocos estados de un protocolo definido por el estándar FIPA para poder usarlo.

Actualmente el API Cagents ofrece dos protocolos del estandar FIPA, el protocolo Request y Contract-Net. En el código que se encuentra a continuación se muestra un ejemplo de uso de la plantilla para el protocolo FIPA Contract-Net. En este ejemplo el agente iniciador Harry desea arreglar su ordenador, envía una solicitud de propuestas de costes a los agentes participantes Sally y Charlie. Posteriormente los agentes participantes comunicarán a Harry el coste de la reparación, Harry aceptara la propuesta con menor coste.

Para usar una de esta plantillas hemos de crear nuestra propia clase hija de la plantilla del protocolo Contract-Net. La clase madre tiene ciertos métodos declarados como abstractos por lo que en nuestra clase hija deberemos implementarlos. Estos métodos son los que se ejecutaraán en ciertos estados clave del protocolo, por ejemplo, en el caso de FIPA Contract-Net, en el rol de iniciador, es necesario implementar el método que se ejecutará en el estado del tipo *Action* donde una vez recibidas las propuestas se decide cual de ellas se acepta y cual se rechaza.

A continuación se muestra el código del método *Initialize* del agente Harry.

```
1  protected void Initialize(CProcessor myProcessor, ACLMessage
       welcomeMessage) {
2      //Creamos nuestro protocolo a partir de la plantilla
3      class myFIPA_CONTRACTNET extends FIPA_CONTRACTNET_Initiator {
4         @Override
5         protected void doEvaluateProposals(CProcessor myProcessor,
6            ArrayList<ACLMessage> proposes,
7            ArrayList<ACLMessage> acceptances,
8            ArrayList<ACLMessage> rejections) {
9            int min = 1000;
10           int index = -1;
11           for(int i=0; i < proposes.size(); i++){
12              if(Integer.valueOf(proposes.get(i).getContent()) < min){
13                  min = Integer.valueOf(proposes.get(i).getContent());
14                  index = i;
15              }
16           }
17           for(int i=0; i < proposes.size(); i++){
18              if(i == index){
19                 System.out.println("I accept "+proposes.get(i).
                       getSender()+"'s proposal");
20                 ACLMessage accept = new ACLMessage(ACLMessage.
                       ACCEPT_PROPOSAL);
```

```
21                  accept.setContent("I accept your proposal");
22                  accept.setReceiver(proposes.get(i).getSender());
23                  accept.setSender(getAid());
24                  accept.setProtocol("fipa-contract-net");
25                  acceptances.add(accept);
26              }
27              else{ // reject the rest
28                  System.out.println("I reject "+proposes.get(i).
                        getSender()+"'s proposal");
29                  ACLMessage reject = new ACLMessage(ACLMessage.
                        REJECT_PROPOSAL);
30                  reject.setContent("I don't like your proposal, I
                        reject it");
31                  reject.setReceiver(proposes.get(1).getSender());
32                  reject.setSender(getAid());
33                  reject.setProtocol("fipa-contract-net");
34                  rejections.add(reject);
35              }
36          }
37      }
38
39      @Override
40      protected void doReceiveInform(CProcessor myProcessor,
41          ACLMessage msg) {
42          // receive accepted proposal result
43          System.out.println("Result: "+msg.getContent());
44      }
45  }
46  ACLMessage msg;
47  msg = new ACLMessage(ACLMessage.CFP);
48  msg.addReceiver(new AgentID("Sally"));
49  msg.addReceiver(new AgentID("Charlie"));
50  msg.setContent("How much would it cost to fix my computer?");
51  CProcessorFactory talk = new myFIPA_CONTRACTNET().newFactory("
        TALK", null, msg, 1, myProcessor.getMyAgent(), 2, 2000, 2000);
52
53  this.addFactoryAsInitiator(talk);
54  myProcessor.createSyncConversation(msg);
55  myProcessor.ShutdownAgent();
56 }
```

Como se puede observar en la línea 2 se crea una *CFactory* a partir de la plantilla para FIPA Contract-Net, esta clase hija se llama *myFIPA_CONTRACTNET*. En el interior de la clase es donde se definen los métodos declarados como abstractos en la plantilla, estos son *doEvaluateProposals* y *doReceiveInform*. El primero sirve para, una vez recibidas las propuestas, elegir cual se acepta y cuales se rechazan. El segundo método sirve para tratar el mensaje que el participante cuya propuesta se aceptó envía informando del resultado de la propuesta aceptada.

Cada método de los estados del grafo de la plantilla puede ser modificado sobreescribiendo el método de la clase padre. Cada uno de estos métodos tiene un comportamiento por defecto, por lo que no es necesario modificarlos, solo es estríctamente necesario programar los métodos declarados como abstractos en la clase madre.

A continuación se muestra el código de uno de los agentes participantes (Harry o Charlie).

```
1  protected void Initialize(CProcessor myProcessor, ACLMessage
       welcomeMessage) {
2     class myFIPA_CONTRACTNET extends FIPA_CONTRACTNET_Participant {
3        @Override
4        protected String doReceiveSolicit(CProcessor myProcessor,
            ACLMessage msg) {
5          // accept all the solicits
6          return "SEND_PROPOSAL";
7        }
8        @Override
9        protected void doSendInfo(CProcessor myProcessor, ACLMessage
            messageToSend) {
10          messageToSend.setSender(getAid());
11          messageToSend.setReceiver(myProcessor.
               getLastReceivedMessage().getSender());
12          messageToSend.setContent("I'm "+getAid()+". Ok. Your
               computer has been fixed");
13          messageToSend.setPerformative(ACLMessage.INFORM);
14          messageToSend.setProtocol("fipa-contract-net");
15       }
16       @Override
17       protected void doSendProposal(CProcessor myProcessor,
            ACLMessage messageToSend) {
18          Random rand = new Random(System.currentTimeMillis());
19          int x = rand.nextInt(100);
20          messageToSend.setSender(getAid());
```

```
21              messageToSend.setReceiver(myProcessor.
                    getLastReceivedMessage().getSender());
22              messageToSend.setContent(String.valueOf(x));
23              messageToSend.setPerformative(ACLMessage.PROPOSE);
24              messageToSend.setProtocol("fipa-contract-net");
25          }
26          @Override
27          protected String doTask(CProcessor myProcessor, ACLMessage
                solicitMessage) {
28              // actions to fix the computer
29          }
30      }
31      CProcessorFactory talk = new myFIPA_CONTRACTNET().newFactory("
            TALK", null, null, 1, myProcessor.getMyAgent(), 0);
32      this.addFactoryAsParticipant(talk);
33  }
```

Como se puede observar en el método *doReceiveSolicit* (el cual selecciona que solicitudes de propuestas son aceptadas) el agente acepta todas las solicitudes. La propuesta que se envía se decide en el método *doSendProposal*, en aras de la simplicidad del ejemplo la propuesta del coste de la reparación enviada se calcula aleatoriamente. Posteriormente si la propuesta ha sido aceptada en el método *doTask* el agente realizará la tarea solicitada por el participante. Finalmente en el método *doSendInfo* el agente crea el mensaje para informar sobre el resultado de la tarea realizada (en este caso como ha ido la reparación del ordenador).

Capítulo 5

Agentes BDI en Magentix2

Uno de los modelos de agente más extendidos es el llamado *belief-desire-intention* (BDI). Las arquitecturas BDI se originaron en el trabajo del proyecto Rational Agency en el Stanford Research Institute a mediados de los 80. Los orígenes de este modelo se basan en la teoría del razonamiento práctico humano, teoría desarrollada por el filósofo Michael Bratman [12]. El armazón conceptual del modelo BDI se describe en [13].

Para definir de una manera rápida los componentes del modelo BDI podemos resumirlos de la siguiente manera:

- Beliefs: son la información que el agente posee del mundo. Esta información puede estar desfasada o incluso ser inexacta.

- Desires: son todos los posibles estados del mundo que el agente pude querer conseguir. De todas maneras, tener un deseo no implica que el agente actúe según ese deseo, es una influencia potencial en las acciones del agente.

- Intentions: son los estados del mundo que el agente ha decidido conseguir. Las intenciones pueden ser objetivos que han sido delegados al agente, o pueden ser resultado de una consideración de las opciones para conseguir un objetivo.

En el contexto de agentes racionales, el modelo BDI se muestra muy atractivo por varias razones. En primer lugar, las abstracciones usadas en el modelo son realmente intuitivas, es sencillo comprender la distinción entre el proceso de decidir que hacer y el de como hacerlo, de manera similar, las nociones de creencia, deseo e intención son muy familiares para el razonamiento humano.

Existen múltiples implementaciones del modelo BDI, vale la pena destacar JACK [15], el cual es un lenguaje de programación comercial que extiende el lenguaje Java con características BDI. En este lenguaje las creencias son tratadas como estructuras de datos arbitrarias, los deseos como eventos y las intenciones son realizadas como ejecución de planes. Con todo esto, la implementación de los estados mentales, deliberación y el razonamiento práctico difieren del planteamiento filosófico.

Por otra parte en JADEX [14], el cual es un complemento de la plataforma JADE, se solucionan algunos de los problemas de la propuesta de JACK, aunque mantiene ciertas semejanzas. Como principal diferencia tenemos que JADEX soporta una representación explícita y declarativa de los objetivos.

Otra propuesta es 2APL [17], el cual es un lenguaje de programación de agentes. Permite implementar creencias, objetivos, actualizaciones de creencias, acciones externas, acciones comunicativas y un conjunto de reglas de razonamiento práctico mediante las cuales los objetivos de los agentes pueden ser actualizados.

Otra aproximación al modelo BDI es PRACTIONIST [33], como diferencias destacables con otras propuestas encontramos que los agentes son capaces de razonar sobre sus creencias y sobre las creencias de los otros agentes en el SMA. Además los agentes tienen capacidad de planificación y son capaces de decidir si dos objetivos son dependientes, independientes o incompatibles. Esta capacidad de planificación permite la ejecución en paralelo de varias intenciones del agente.

De entre todas las opciones disponibles de lenguajes de programación de agentes racionales se escogió Jason [11], el cual es un intérprete para una versión extendida de AgentSpeak(L) [34]. AgentSpeak(L) es particularmente interesante, en comparación a otros lenguajes orientados a agentes, en que retiene los aspectos más importantes de los sistemas de planificación reactiva basados en BDI. Por otra parte la implementación de Jason es muy flexible y permite funcionar sobre cualquier infraestructura de agente, a diferencia de todas las demás opciones comentadas, las cuales han sido creadas para funcionar sobre agentes Jade. En la siguiente sección se exponen con cierto detalle las características más importantes de la versión extendida del lenguaje AgentSpeak(L) del cual hace uso Jason, así como del ciclo de razonamiento de los agentes implementados en Jason. En la última sección de este capítulo se muestra como se ha integrado el intérprete Jason sobre la infraestructura de agente Magentix2.

5.1. Jason

Los componentes más importantes del lenguaje de programación Jason son los siguientes:

- *Beliefs*: Cada agente tiene una base de creencias, la cual, en su forma más simple, es una colección de literales. Se representan de forma simbólica del siguiente modo `tall(john)`, esta creencia expresa una propiedad particular de un individuo u objeto, en este caso *John*. Las creencias de un agente provienen de tres fuentes: la percepción del entorno, por la interacción con otro agente o notas mentales, que son creencias que provienen del propio agente.

- *Goals*: Representan los estados del mundo que el agente desea conseguir. Existen dos tipos de objetivos en Jason:

  - *Achievement Goals*: se denotan con el operador `!`. Por ejemplo si escribimos la sentencia `!own(house)` indicamos que el agente tiene el objetivo de alcanzar un cierto estado del mundo en el cual el agente crea que posee la casa.

  - *Test Goals*: se denotan con el operador `?`. Normalmente se usan para recuparar información de la base de creencias. Por lo tanto el objetivo `?bank_balance(BB)` intenta instanciar la variable lógica `BB` con la cantidad específica de dinero que el agente actualmente cree que tiene en su balance bancario.

- Plans: Los planes son las "recetas" o pasos que el agente ha de realizar para cumplir sus objetivos. Un plan en Jason tiene tres partes:

  - *Triggering event*: Ciertos eventos o cambios del entorno pueden comportar nuevas oportunidades para el agente para realizar acciones y tal vez, oportunidades para considerar adoptar nuevos objetivos o descartar objetivos existentes. Existen dos tipos de cambios en la actitud mental del agente que son importantes en un programa de agente: cambios en las creencias y cambios en los objetivos del agente. Cambios en ambos tipos de actitudes crean los eventos sobre los cuales el agente actuará. Además estos cambios pueden ser de dos tipos: adición y borrado. Por lo tanto, en resumen tenemos que los eventos pueden ser: adición de objetivos o creencias y borrado de objetivos y creencias. Cuando se produzca el evento que dispara un cierto plan diremos que el plan pasa a ser *relevante*.

- *Context*: El contexto de un plan sirve para comprobar si la situación actual del entorno del agente hace que el plan tenga posibilidades de tener éxito. Un contexto es una conjunción de literales y expresiones relacionales que se evalúa cierto o falso. Si el contexto de un plan es cierto diremos que ese plan es *aplicable*.
- *Body*: El cuerpo del plan es el curso de acción que el agente ha de seguir. Dentro del cuerpo de un plan pueden aparecer los siguientes elementos:
  - *Actions*: Son actos que el agente realiza en su entorno. Un ejemplo de acción que interactuase con el entorno sería `rotate(left_arm, 45)`, esta acción haría que un agente robótico rotase su brazo izquierdo 45 grados.
  - *Achievement Goals*: Dentro de un plan pueden añadirse nuevos objetivos. Estos nuevos objetivos se han de cumplir para que el plan tenga éxito y con ello el objetivo inicial.
  - *Test Goals*: Estos objetivos son usados normalmente para recuperar información de la base de creencias del agente. Podría pensarse que los test goals no son necesarios ya que todas las variables necesarias para ejecutar el plan pueden ser instanciadas en el contexto, esto no es siempre cierto, ya que en entornos muy variables puede ser que el contexto del agente varíe durante la ejecución del plan y se necesite instanciar una variable lo más tarde posible para realizar una acción con la información actualizada.
  - *Mental notes*: Durante el transcurso de la ejecución de un plan, un agente puede necesitar añadir nuevas creencias a su base de creencias. Estas nuevas creencias son consideradas notas mentales ya que el origen de la creencia es el propio agente. Son muy útiles para recordar resultados de acciones y tenerlos en cuenta para la realización de acciones futuras.
  - *Internal actions*: Este tipo de acciones son aquellas que no tienen un impacto en el entorno del agente. Por ejemplo mostrar un texto por pantalla.
  - *Expressions*: Tienen una utilidad y siguen una sintaxis muy similar a Prolog.

Un agente opera según un ciclo de razonamiento, el cual, en el caso de Jason, podemos dividir en 10 pasos. En la figura 5.1 se muestra de una manera gráfica el ciclo de razonamiento y como afecta a las distintas partes del agente.

Los pasos del ciclo de razonamiento son los siguientes:

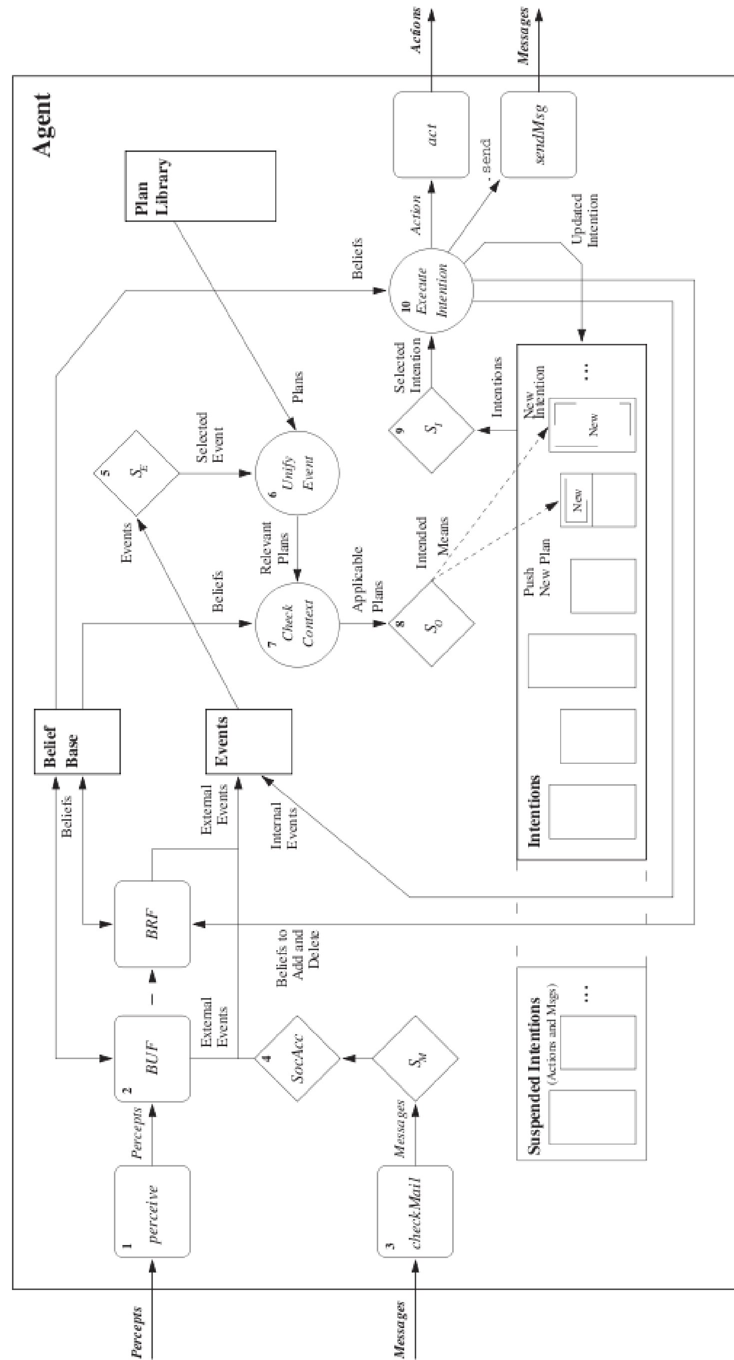

Figura 5.1: Ciclo de razonamiento de Jason

1. Percepción del entorno: El agente obtiene una lista de percepciones de su entorno.

2. Actualización de la base de creencias: Una vez el agente ha obtenido la lista de percepciones, modifica su base de creencias de acuerdo a estas nuevas percepciones.

3. Recepción de comunicación desde otros agentes: Cada agente tiene un buzón donde se almacenan los mensajes que va recibiendo. En este paso un mensaje del buzón es extraído. Existe una función de selección de mensaje que se encarga de este cometido, el usuario puede modificarla para variar su comportamiento por defecto.

4. Seleccionar mensajes socialmente aceptables: Al seleccionar un mensaje, el mensaje es pasado a una función que determina si un mensaje es socialmente aceptable. Por ejemplo, un agente puede solicitar a otro que realice una acción, esta petición sólo es socialmente aceptable si el agente que realizó la petición tiene autoridad sobre el agente al que se le pide que realice la acción.

5. Seleccionar un evento: Cada vez que se produce un evento este es almacenado en una lista de eventos pendientes del agente, en cada ciclo de razonamiento un solo evento es elegido. La función de selección se puede modifiar por el desarrollador del agente.

6. Recuperar todos los planes relevantes: Una vez elegido un evento se comprueba si algún plan tiene como disparador dicho evento. Si es así ese plan es considerado relevante y se añade al conjunto de planes relevantes.

7. Determinar los planes aplicables: Una vez conformado el conjunto de planes relevantes, para cada uno de ellos, se evalúa su contexto. Aquellos cuyo contexto sea evaluado a cierto formarán el conjunto de planes aplicables.

8. Seleccionar un plan aplicable: Del conjunto de planes aplicables el agente debe seleccionar uno que llevará a cabo, esta decisión es realizada por la función de selección del plan aplicable, esta función también puede ser personalizada. El plan seleccionado pasa a ser una intención del agente y se añade al conjunto de intenciones del agente.

9. Seleccionar una intención para ejecutar: Del conjunto de intenciones se escoge una para proseguir su ejecución. Esta selección viene dada por una función de selección de intenciones, que al igual que las demás funciones, también puede ser personalizada. Hay que tener en cuenta que pueden haber intenciones suspendidas a la espera de que un evento

las desbloquee, un ejemplo sería el caso de un agente que envía un mensaje preguntando cierta información a otro agente, hasta que el agente no reciba respuesta, la intención que envío el mensaje queda bloqueada. Estas intenciones bloqueadas no son elegibles para proseguir su ejecución hasta que queden desbloqueadas.

10. Ejecutar un paso de una intención: Por último se ejecuta un nuevo paso de la intención escogida en el paso número 9 del ciclo de razonamiento del agente.

Jason usa para la comunicación Knowledge Query and Manipulation Language (KQML) [30]. Este lenguaje define una serie de performativas, las cuales hacen explícitas las intenciones del agente al enviar el mensaje. La lista de performativas disponibles son: tell, untell, achive, unachive, askOne, askAll, tellHow, untellHow y askHow. El envío de mensajes desde un agente Jason se trata como una acción interna del agente, por ejemplo `send(r, tell, open(left_door));` sería un mensaje cuyo receptor es `r`, la performativa `tell` y el contenido `open(left_door)`. El tratamiento de los mensajes es automático en función de la performativa, por ejemplo, para el mensaje anterior, si el mensaje es socialmente aceptable por el agente receptor, la creencia `open(left_door)` sería añadida automáticamente a su base de creencias.

5.2. Jason sobre Magentix2

Como se ha comentado al principio de este capítulo Jason puede funcionar sobre cualquier infraestructura de agente que ofrezca una serie de servicios mínimos. La versión actual de Jason funciona sobre tres infraestructuras: una propia, SACI y Jade. Esta fácilidad para funcionar sobre diversas infraestructuras de agente ha sido clave a la hora de escoger esta opción para proveer de agentes BDI a Magentix2.

Para poder hacer funcionar el intérprete Jason como un agente Magentix2 ha sido necesario crear una nueva clase de agente el cual pueda ejecutar el intérprete Jason, a este tipo de agente de ahora en adelante lo llamaremos *JasonAgent*. Este agente requiere de dos elementos para funcionar, el primero de ellos es un programa en el lenguaje de programación AgentSpeak extendido explicado en la sección anterior. Este programa es el que dictaminará el comportamiento del agente. El segundo componente es, lo que llamamos, una arquitectura de agente. Esta arquitectura de agente permite definir acciones del agente sobre el entorno y acciones internas. También sirve para modificar algunos comportamientos del agente, por ejemplo, pode-

mos redefinir el método por el cual recibe mensajes y que realice ciertas acciones automáticas cuando reciba un cierto tipo de mensaje. Existe una arquitectura por defecto en la plataforma Magentix2, esta arquitectura puede ser modificada (mediante herencia). En el siguiente capítulo, donde se muestra la integración de Moise$^+$ en Magentix2, se aprecia la versatilidad de esta arquitectura de clases. En la figura 5.2 se muestra la estructura de un *JasonAgent*.

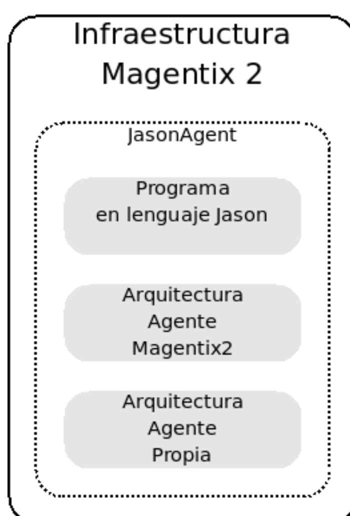

Figura 5.2: Estructura de JasonAgent

Hay que tener en cuenta que Jason hace uso de KQML para la comunicación, mientras que en Magentix2 se usa FIPA, por lo tanto ha sido necesario crear un método de traducción de mensajes FIPA a KQML y viceversa. Este método de traducción se encuentra integrado en la arquitectura de agente por defecto ofrecida por Magentix2 y es completamente transparente al programador del agente.

Es importante destacar que gracias a que el *JasonAgent* es un agente Magentix2, éste puede hacer uso todas las herramientas que Magentix2 ofrece, en otras palabras, es posible usar las herramientas de seguridad, la de trazas y la de protocolos de interacción dinámicos con *JasonAgents*. Además este tipo de agente es capaz de formar parte de cualquier SMA funcionando sobre Magentix2.

Capítulo 6

Organizaciones de Agentes en Magentix2

La autonomía de los agentes es una de las características más importantes del concepto de agente [40]. Aún así, esta autonomía puede llevar a que el sistema tenga un comportamiento indeseado, ya que cada agente actúa individualmente. Este problema se puede solucionar creando una organización en el sistema. La organización puede ser vista como un conjunto de restricciones al comportamiento que el agente adopta cuando entra en el sistema. Esta aproximación al problema es muy útil en un SMA abierto ya que no sabemos que tipo de agente entrará en el sistema y por tanto es necesario marcar unos límites al comportamiento del agente que ha entrado en él.

Ya que la aproximación de organización es muy útil para SMA abiertos y Magentix2 es una plataforma cuyo objetivo es brindar apoyo a este tipo de sistemas, es necesario que Magentix2 ofrezca soporte en el nivel organizativo. Con el fin de ofrecer diversidad de opciones a los futuros desarrolladores de SMA en Magentix2, la plataforma incluye dos modelos de organización, Moise$^+$ y THOMAS.

6.1. Moise$^+$

Moise$^+$ es un modelo de organización el cual contempla tres dimensiones en la organización: estructural, funcional y deóntica. A continuación se exponen con detalle cada una de estas dimensiones.

6.1.1. Dimensión Estructural

La dimensión estructural se divide en tres niveles: (i) los comportamientos de los cuales un agente es responsable cuando adopta un determinado rol *(nivel individual)*; (ii) los vínculos de conocimiento, comunicación y autoridad entre los roles *(nivel social)*; (iii) la agrupación de roles en grupos *(nivel social)*.

A nivel individual existe la herencia entre roles, un rol que herede de otro, hereda también todos sus vínculos sociales con los demás roles. En el modelo Moise^{+}, la adopción de roles viene restringida por la relación de compatibilidad entre roles. Un agente puede adquirir dos o más roles sólo si son compatibles entre ellos.

En el nivel colectivo, los roles son dividos en grupos, estos grupos pueden tener subgrupos. La relación entre grupo y rol viene definida por su cardinalidad, existe un mínimo y un máximo de agentes con un determinado rol que pueden formar parte de un grupo. Cuando un grupo cumple todos los requisitos de máximos y mínimos se dice que está bien formado.

En el nivel social, los roles se encuentran vinculados. Cada vínculo tiene un rol fuente y otro objetivo. Existen tres tipos de vínculo:

- Conocimiento: un agente con el rol fuente conoce a los agentes que juegan el rol objetivo.
- Comunicación: los agentes con el rol fuente pueden enviar mensajes a los agentes que jueguen el rol objetivo.
- Autoridad: los agentes que jueguen el rol fuente tienen autoridad sobre los agentes que interpreten el rol objetivo.

Un vínculo de autoridad implica uno de comunicación y a su vez, uno de comunicación uno de conocimiento. Los vínculos pueden ser intragrupo o intergrupo. Los vínculos intragrupo indican que un agente jugando el rol fuente en un grupo *gr* está vinculado a todos los agentes jugando el rol objetivo en el mismo grupo *gr* o en un subgrupo de *gr*. El vínculo intergrupo indica que un agente jugando el rol fuente está vinculado a todos los agentes jugando el rol objetivo sin importar a que grupos los agentes pertenezcan. En la figura 6.1 se muestra de forma gráfica la especificación estructural de un equipo de fútbol.

6.1.2. Dimensión Funcional

La especificación funcional se compone de un conjunto de esquemas que representan como el SMA consigue sus objetivos, descomponiendo estos objetivos en planes y distribuyendo

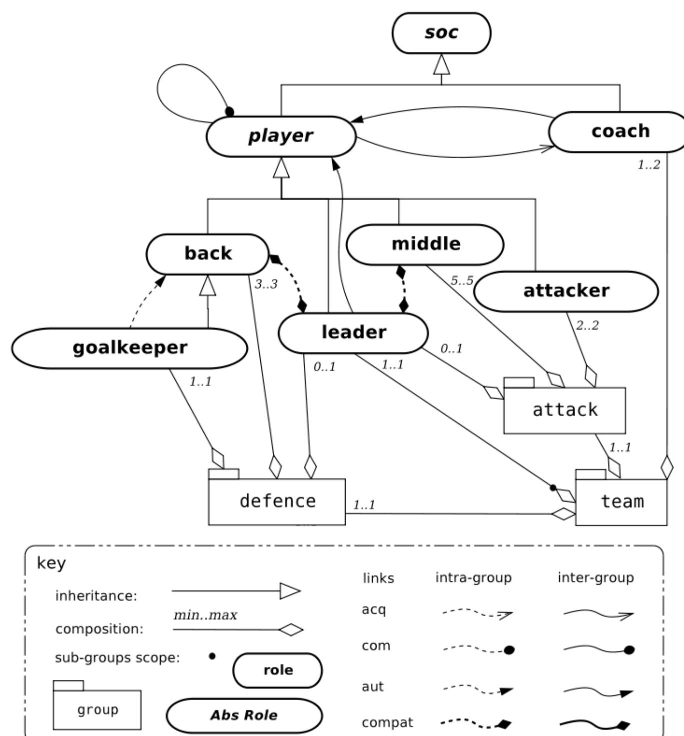

Figura 6.1: Estructura de un equipo de fútbol usando Mosie$^+$

estos planes entre los agentes mediante misiones. Un esquema puede ser visto como un árbol de descomposición de objetivos donde la raíz es el objetivo global y las hojas son objetivos que pueden ser alcanzados por los agentes. Para relacionar los objetivos de un esquema con los roles tenemos las misiones, una misión es un conjunto de objetivos coherentes que un agente puede alcanzar. Cuando un agente se compromete a una misión, el agente es responsable de todos los objetivos de la misión.

En un esquema, cada nodo no hoja se descompone en subobjetivos mediante planes usando los siguientes operadores:

- secuencia ",": el plan "$g_1 = g_2, g_3$" indica que el objetivo g_1 será alcanzado sólo si se consigue el objetivo g_2 y a continuación el objetivo g_3.
- selección "|": el plan "$g_1 = g_2 \mid g_3$" indica que el objetivo g_1 será alcanzado si uno, y sólo uno de los objetivos g_2 o g_3 es conseguido.
- paralelismo "‖": el plan "$g_1 = g_2 \parallel g_3$" indica que el objetivo g_1 será conseguido si se consiguen g_2 y g_3, aunque éstos pueden ser conseguidos en paralelo.

Como ejemplo de una especificación funcional la figura 6.2 muestra un esquema de ataque para la organización del equipo de fútbol.

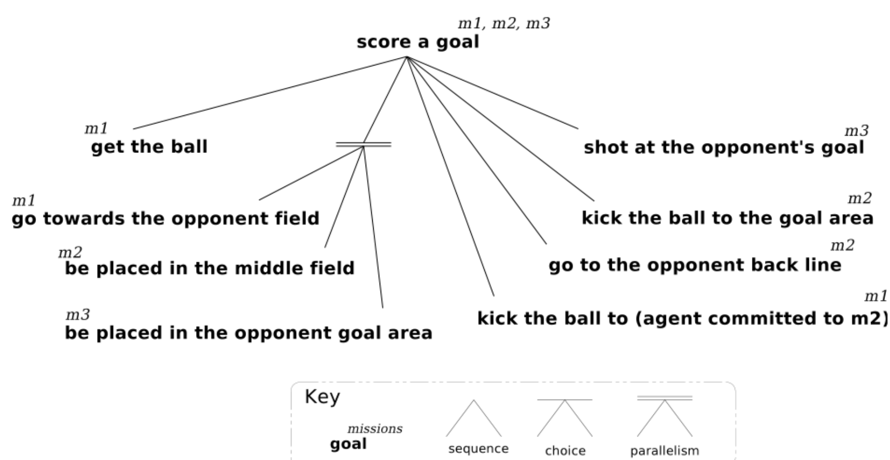

Figura 6.2: Esquema de ataque para equipo de fútbol usando Mosie$^+$

6.1.3. Dimensión Deóntica

La dimensión deóntica trata la autonomía de los agentes, especificando explícitamente que está permitido y obligado en la organización. La correspondiente especificación describe los permisos y obligaciones de los roles para con las misiones. Un permiso *permission(p, m)* indica que un agente con el rol *p* puede comprometerse a cumplir la mision *m*. Por otra parte, una obligación *obligation(p, m)* indica que un agente interpretando el rol *p* debe comprometerse a realizar la misión *m*.

6.2. S-Moise$^+$ y J-Moise$^+$

S-Moise$^+$ [29] es un software intermedio para organizaciones basado en el modelo Moise$^+$. Este software intermedio ofrece acceso al agente al estado actual de la organización (grupos creados, esquemas, roles asignados, etc.) y permite a los agentes modificar la entidad de la organización y su especificación. Estos cambios están delimitados para asegurar que los agentes respetan la especificación de la organización.

S-Moise$^+$ tiene dos componentes principales: un API llamado *OrgBox* que los agentes usan para acceder a la capa de organización y un agente especial llamado *OrgManager*. Este último

almacena el estado actual de la instancia de organización y mantiene su consistencia durante su ciclo de vida. Los agentes envian peticiones de cambios en la organización al *OrgManager* a través de sus *OrgBox*. El *OrgManager* realizará estos cambios sólo si las peticiones no violan las restricciones de la organización.

J-Moise$^+$ se ha construido sobre S-Moise$^+$. J-Moise$^+$ añade al lenguaje de agentes Jason capacidades organizativas. Básicamente J-Moise$^+$ ofrece al programador de agentes Jason un conjunto de acciones organizativas, además de producir eventos organizativos a los que el agente Jason puede reaccionar. Por ejemplo, un agente *a* puede crear un esquema mediante la acción `jmoise.create_scheme(Sch, GId)`, además un agente que forme parte de la organización recibirá eventos cada vez que se produzca un cambio en la organización, estos eventos son semejantes a los vistos en la sección 5.1. Siguiendo el ejemplo anterior donde el agente ha creado un esquema, los demás agentes de la organización recibirían el evento `+scheme(Sch, Gid)[owner(a)]`. En otras palabras, los eventos organizacionales son eventos que se perciben del entorno, solo que en este caso el entorno es la organización.

6.3. Moise$^+$ sobre Magentix2

J-Moise$^+$ permite usar el modelo Moise$^+$ con agentes Jason, teniendo ya integrados los agentes Jason en la plataforma Magentix2, la integración de J-Moise$^+$ en Magentix2 es una opción obvia para disponer de un modelo de organización en Magentix2.

Para usar J-Moise$^+$ en Magentix2 se ha creado una nueva arquitectura de agente para los *JasonAgents* comentados en la sección 5.2. Se han creado dos arquitecturas diferenciadas, una para agentes organizativos y otra para el agente *OrgManager*. Por una parte, la arquitectura para los agentes organizativos funciona como una interfaz entre el agente y el *OrgBox* implementado en S-Moise$^+$. La arquitectura define como nuevas acciones internas del agente todas las funciones que ofrece el *OrgBox* de S-Moise$^+$. Por otra parte, la arquitectura *OrgManager* está creada para que el agente que funcione con esta arquitectura únicamente realice las actividades de mantenimiento y modificación de la instancia de organización sobre la que esté funcionando. De esta manera las acciones que el agente puede realizar sobre la organización no se ofrecen como acciones internas del agente, si no que son ejecutadas a petición del resto de agentes organizativos. Estas peticiones llegan al agente *OrgManager* en forma de mensajes, estas peticiones son tratadas de forma automática por el agente, aceptando aquellas permitidas al agente solicitante y rechazando las que no lo sean.

Al igual que en el caso de la integración de Jason sobre Magentix2, los agentes que hagan uso de las herramientas organizativas que ofrece J-Moise$^+$ siguen pudiendo usar todas las demás herramientas de la plataforma, como son la seguridad, las trazas y soporte a conversaciones.

En la imagen 6.3 se puede observar la estructura de la integración de J-Moise$^+$ sobre Magentix2.

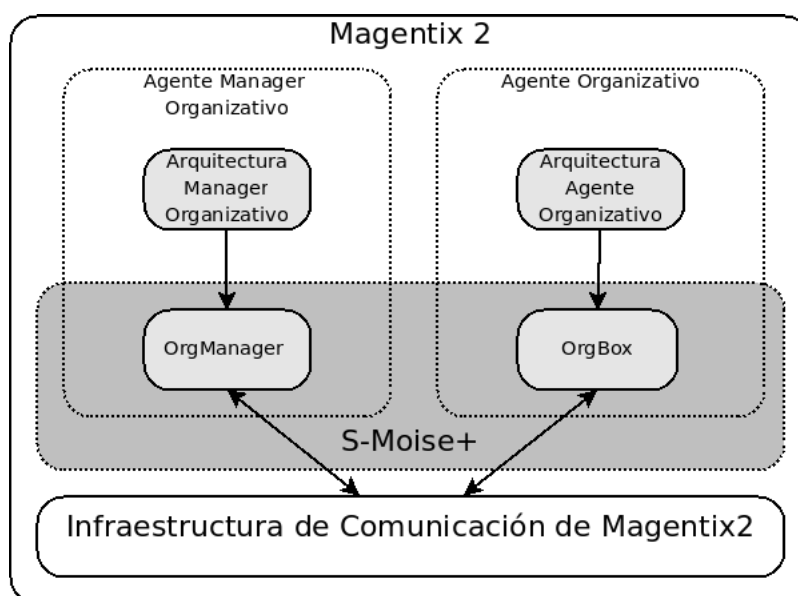

Figura 6.3: Integración de Moise$^+$ en Magentix2

6.4. THOMAS

THOMAS [37] (MeTHods, Techniques and Tools for Open Multi-Agent Systems) es un modelo de organizaciones de agentes y servicios web. Ha sido desarrollado por el Grupo de Tecnología Informática - Inteligencia Artificial del Departamento de Sistemas Informáticos y Computación, de la Universidad Politécnica de Valencia.

La arquitectura de THOMAS consiste básicamente en un conjunto de servicios modulares. Aunque está basada en la arquitectura FIPA, THOMAS expande las capacidades de dicha arquitectura para gestionar organizaciones. Por lo tanto, se ha incluido un nuevo módulo para conseguir este objetivo, con la redefinición del *Directory Facilitator* (DF) de FIPA. Con lo cual,

puede manejar los servicios de una manera más elaborada siguiendo las guías para las arquitecturas orientadas a servicios (*Service Oriented Architectures*, SOA). Los servicios son lo más importante del modelo THOMAS, los agentes disponen de un conjunto de servicios incluídos en diferentes módulos o componentes. Los principales componentes son:

- Agente intermediario SF (*Service Facilitator*). Ofrece servicios simples y complejos para los agentes activos y las organizaciones. Su funcionalidad básica es ofrecer un servicio de páginas amarillas para búsqueda y otro de páginas verdes como descriptor de servicios.

- Agente intermediario OMS (*Organization Management System*). Es el responsable de la gestión de las organizaciones y de sus entidades. Con lo cual, permite crear y gestionar cualquier organización.

6.4.1. Agente Intermediario SF

El SF es un mecanismo mediante el cual los agentes y las organizaciones pueden ofrecer y encontrar servicios. Provee un soporte en el que las entidades autónomas pueden registrar descripciones de servicios como entradas de directorio. También actúan como pasarela de acceso a la plataforma THOMAS. Esto es manejado de forma transparente pero teniendo en cuenta la seguridad y la gestión de derechos de acceso.

Este agente intermediario puede encontrar servicios mediante la búqueda de un perfil dado o de un objetivo que puede ser conseguido ejecutando el servicio. Además, también actúa como gestor de páginas amarillas, con lo que puede encontrar qué entidades proveen un servicio dado.

Un servicio representa la interacción entre dos entidades que se comunican entre sí. Éste ofrece las capacidades para satisfacer un objetivo dado y suele tener algunas precondiciones que necesitan cumplirse para su ejecución. A parte de los parámetros funcionales del servicio, también pueden existir algunos en la descripción de éste que no lo sean, como por ejemplo: calidad del servicio y protocolos de seguridad.

En el caso del modelo THOMAS, se utilizan los protocolos de comunicación FIPA. Por lo tanto, cada servicio tiene asociado un protocolo y en los casos en los que es necesario la ejecución de una cadena de protocolos quedan marcados como “complejos”. Por otra parte, teniendo en cuenta que THOMAS trabaja con servicios semánticos, la ontología usada en el servicio es muy importante. Así pues, cuando se accede a la descripción de un servicio, cualquier entidad tiene la información necesaria para interactuar con el mismo y hacer una aplicación que lo utilice. Además, la descripción se puede utilizar para servicios precompilados, donde el modelo

de proceso se compone de una secuencia de servicios elementales que serán ejecutados, en vez de los procesos internos de ese servicio.

En general, un servicio puede ser ofrecido por más de un proveedor en el sistema, por lo tanto, cada servicio tiene asignada una lista de proveedores. Todos los proveedores pueden ofrecer copias exactas del servicio, es decir, comparten una implementación común. Por otra parte, pueden compartir sólo la interfaz y cada uno implementar el servicio de una forma. Esto se consigue en THOMAS teniendo el perfil del servicio separado del proceso.

El SF dispone de un conjunto de servicios estándar (meta-servicios) para gestionar los servicios proveídos por las organizaciones o por agentes individuales. Estos metaservicios también tienen que ser usados por el resto de los componentes de THOMAS (OMS) para publicitar los suyos. Los meta-servicios pueden ser clasificados en tres tipos:

- Registro: permiten añadir, modificar y eliminar servicios del directorio SF.
- Alcance: se utilizan para gestionar la asociación entre los proveedores y sus servicios.
- Descubrimiento: su funcionalidad consiste en la búsqueda y composición de servicios como respuesta a las peticiones del usuario. Pueden ser tan complejos que pueden ser delegados a componentes especializados.

6.4.2. Agente intermediario OMS

El agente intermediario OMS se encarga de la gestión de las organizaciones. Esto incluye la especificación y la administración de los componentes estructurales (roles, unidades y normas) y de los componentes de ejecución (agentes participantes, roles que estos juegan y unidades organizativas activas). Las organizaciones están estructuradas mediante unidades organizativas que representan grupos de entidades (agentes u otras unidades). Los componentes que forman una unidad persiguen un objetivo común. Las unidades organizativas tienen una topología interna que impone control y restricciones a las relaciones entre los agentes.

Existe una unidad llamada “virtual” en la plataforma THOMAS que ha sido definida para representar el “mundo” del sistema en el que los agentes participan por defecto. Las organizaciones son creadas dentro de esta unidad y éstas, a su vez, pueden estar compuestas de más unidades. Cabe destacar que los roles se definen dentro de cada unidad y representan la funcionalidad requerida para alcanzar el objetivo de la unidad. Además pueden tener normas asociadas para controlar las acciones de los roles (por ejemplo, qué servicios pueden solicitar

u ofrecer los agentes que están jugando un rol determinado; permisos para acceder a determinados recursos). En conclusión, los agentes pueden adoptar roles dinámicamente dentro de las unidades, con lo cual el OMS controla todo este proceso y cuáles son las entidades que juegan un rol en cada instante

Los servicios que ofrece el OMS se clasifican como estructurales y dinámicos. Los servicios estructurales son los que modifican la estructura y la normativa de la organización. Por su parte, los dinámicos permiten a los agentes entrar o abandonar la organización de forma dinámica, así como la adopción de roles.

6.5. THOMAS sobre Magentix2

Para la integración de THOMAS sobre Magentix2 ha sido necesario modificar la implementación previa de los agentes intermediarios SF y OMS. En un principio estos agentes fueron implementados como agentes Jade y sólo funcionaban sobre esta plataforma de agentes. Tras la integración ambos agentes han pasado a ser agentes Magentix2 y siguen ofreciendo las mismas funcionalidades, por lo que siguen el modelo de organización de agentes propuesto por THOMAS.

Gracias a los agentes pasarela que posibilitan la interacción de agentes en la plataforma Magentix2 con agentes externos, en este caso agentes Jade, es posible formar una organización gestionada por los agentes intermediarios dentro de Magentix2 que incluya tanto agentes Jade como agentes Magentix2. Los mensajes emitidos por los agentes externos hacia los agentes intermediarios o los del resto de su organización son redirigidos por el agente BridgeAgentOutIn, por otra parte, los mensajes enviados por los agentes internos de la organización o de los agentes intermediarios a los agentes externos son tratados por el agente BridgeAgentInOut.

En la figura 6.4 podemos ver un ejemplo de dos organizaciones THOMAS en Magentix2. En el dibujo podemos ver como los agentes intermediarios están en Magentix2. Una de las organizaciones *Org1* está formada exclusivamente por agentes Magentix2, en cambio la organización *Org2* contiene tanto agentes Magentix2 como Jade.

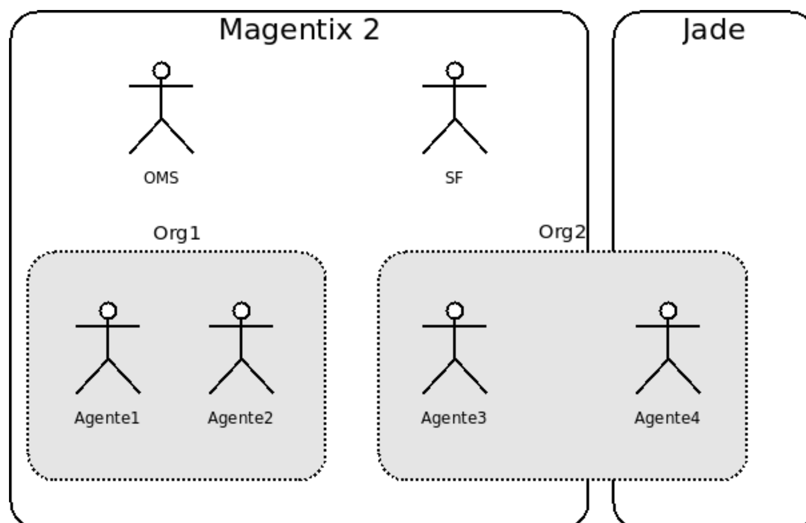

Figura 6.4: Ejemplo de organizaciones THOMAS Magentix2

Capítulo 7

Conclusiones y Trabajo Futuro

Se ha presentado la plataforma de agentes Magentix2, esta nueva plataforma ofrece herramientas novedosas dentro del entorno de plataformas multiagente y SMA. La interacción entre agentes de la plataforma funciona sobre el nuevo y potente estándar de comunicación AMQP. Este estándar y su implementación QPid permiten facilitar la comunicación entre agentes y grupos de estos, además, garantiza la compatibilidad con otros sistemas externos a la plataforma que utilicen el mismo estándar.

Sobre este nivel de comunicación interoperable ofrecido por Magentix2 se ha presentado una propuesta para el manejo de las conversaciones de agentes. Este propouesta se basa en la especificación y ejecución de protocolos de interacción dinámicos. Los protocolos de interacción se especifican como grafos y las conversaciones que siguen estos protocolos son creadas y gestionadas por los *CProcessors* y las *CFactories*. Esta propuesta constituye un primer intento hacia el soporte de interacciones dinámicas entre agentes, las cuales son totalmente necesarias en SMA abiertos y variables.

Además de la propuesta para el tratamiento de conversaciones de agentes, se ha presentado su implementación en forma de herramienta en la plataforma Magentix2. Esta herramienta puede ser utilizada de forma muy sencilla por cualquier agente Magentix2 ya que se presenta en forma de API. Esta API ofrece todas las funcionalidades desarrolladas en la propuesta. Además se han mostrado varios ejemplos para entender el funcionamiento del API así como para reforzar la exposición del modelo de conversaciones de agentes explicado.

Como plataforma de agentes que es, Magentix2 permite la inclusión de agentes racionales a los SMA desarrollados sobre la misma. Para la programación de estos agentes racionales se ha elegido un lenguaje elegante y aceptado por la comunidad científica como es AgentSpeak(L).

Estos agentes racionales además pueden hacer uso de las herramientas que la plataforma Magentix2 ofrece.

Se han presentado dos soportes a organizaciones que han sido integrados en Magentix2. Ambos modelos de organización seleccionados han tenido buena acogida en el entorno científico. Por una parte THOMAS, es un modelo gestado en el propio grupo de investigación. Actualmente nuevas líneas de investigación están trabajando sobre este modelo. Por otra parte, se ha integrado Moise^{Inst}, el cual es un modelo externo al grupo pero que presenta ciertas características que lo hacen interesante, una de ellas es su flexibilidad, la cual ha permitido que ya se hayan realizado modificaciones por otros grupos de investigación, tal y como demuestra la variación Moise^{Inst} [26]. Esta flexibilidad va en consonancia con la filosofía de Magentix2, ya que la plataforma trata de facilitar en la medida de lo posible la adaptación de la misma a las necesidades de los diferentes grupos de investigación que se encuentran en el programa Consolider.

Existen múltiples direcciones en las cuales seguir y aumentar el trabajo presentado. Una de ellas sería la utilización de la propuesta presentada para el tratamiento de protocolos de interacción dinámicos en SMA adaptativos. Este tipo de sistemas tienen la capacidad de tener en cuenta eventos impredecibles que puedan ocurrir en entornos muy variables, de tal manera, que son capaces de adaptarse para seguir cumpliendo sus objetivos. La propuesta presentada para manejar las conversaciones podría ser útil en este tipo de sistemas ya que la posibilidad de modificar los protocolos de interacción de forma dinámica está en sintonía con el concepto de adaptabilidad.

Otra opción para un trabajo futuro sería la creación de una aplicación de alto nivel que facilitase el uso de la herramienta para el tratamiento de protocolos de interacción dinámicos. Con esta aplicación sería posible definir de forma gráfica un protocolo de interacción, posteriormente, a partir de esta representación visual, la aplicación generaría código de forma automática. Más tarde, este código podría ser utilizado por cualquier agente que necesite hacer uso del protocolo generado, ya sea como participante o iniciador.

Por otra parte, una dirección obvia para un futuro trabajo es unir la propuesta para el tratamiento de conversaciones con el de agente BDI. En la plataforma Magentix2 sería posible realizar este trabajo integrando el lenguaje de agentes *Jason* ya incluido en la plataforma y la herramienta para el manejo de conversaciones dinámicas. Este posible trabajo futuro abre la posibilidad de que agentes *Jason* puedan tratar varias conversaciones simultáneamente y por tanto razonar sobre cada una de ellas de manera también simultánea. Por ejemplo, el agente

OrgManager de S-Moise$^+$ podría atender concurrentemente múltiples peticiones de agentes de la organización. Además, esta nueva capacidad de los agentes racionales aumentaría su adaptabilidad, ya que podrían modificar sus protocolos de interacción de forma dinámica.

Bibliografía

[1] Agentbuilder. Acronymics, Inc. http://www.agentbuilder.com.

[2] AMQP: Advanced message queuing protocol, version 0.10. AMQP working group protocol specification http://jira.amqp.org/confluence/display/AMQP/AMQP+Specification.

[3] Madkit. http://www.madkit.org.

[4] Zeus agent toolkit. http://labs.bt.com/projects/agents/zeus/.

[5] E. Argente, V. Botti, and V. Julian. Gormas: a methodological guideline for open multi-agent systems. In *EUMAS'09*, 2009.

[6] G. Artificial. MAGENTIX: Una Plataforma de Sistema Multiagente Integrada en Linux.

[7] A. Artikis. Dynamic protocols for open agent systems. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 97–104, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[8] A. Artikis and M. Sergot. Executable specification of open multi-agent systems. *Logic Journal of the IGPL, doi: 10.1093/jigpal/jzp071*, 2010.

[9] A. Artikis, M. Sergot, and J. Pitt. Specifying norm-governed computational societies. *ACM Trans. Comput. Logic*, 10(1):1–42, 2009.

[10] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa. *JADE PROGRAMMER'S GUIDE*, 2008.

[11] R. H. Bordini, J. F. Hübner, R. Vieira, P. O. Naso, M. (ed. Arthur Golding, and B. Vii. Chapter 1 jason and the golden fleece of agent-oriented programming.

[12] M. E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, March 1999.

[13] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning, 1988.

[14] L. Braubach, W. Lamersdorf, and A. Pokahr. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP–in search of innovation*, 3:76–85, 2003.

[15] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents-components for intelligent agents in java. *AgentLink News Letter*, 2:2–5, 1999.

[16] A. K. Chopra and M. P. Singh. An architecture for multiagent systems: An approach based on commitments. In *Seventh international Workshop on Programming Multi-Agent Systems.PROMAS 2009*, 2009.

[17] M. Dastani. 2APL: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248, 2008.

[18] V. Dignum, F. Dignum, and L. Sonenberg. Towards dynamic reorganization of agent societies. In *In Proceedings of Workshop on Coordination in Emergent Agent Societies*, pages 22–27, 2004.

[19] M. Esteva, J. A. Rodríguez-Aguilar, J. L. Arcos, C. Sierra, and P. García. Institutionalising open multi-agent systems. a formal approach. In *Fourth International Conference on MultiAgent Systems (ICMAS'00)*, pages 381–382. IEEE Computer Society, 2000.

[20] FIPA. *FIPA ACL Message Structure Specification*. FIPA, 2001.

[21] FIPA. *FIPA Agent Message Transport Protocol for HTTP Specification*. FIPA, 2002.

[22] FIPA. *FIPA Contract Net Interaction Protocol Specification*. FIPA, 2002.

[23] FIPA. *FIPA Request Interaction Protocol Specification*. FIPA, 2002.

[24] D. T. Fipa. *Fipa interaction protocol library specification.*

[25] R. L. Fogues, J. M. Alaberola, J. M. Such, A. Espinosa, and A. Garcia-Fornes. Towards dynamic agent interaction support in open multiagent systems. *In press*, 2010.

[26] B. Gâteau. Using a normative organisational model to specify and manage an institution for multi-agent systems.

[27] S. Goel, H. Sharda, and D. Taniar. Message-oriented-middleware in a distributed environment. In *IICS*, pages 93–103, 2003.

[28] J. F. Hübner and J. S. Sichman. *Saci Programming Guide*. Universidade de Sao Paulo, July 2003.

[29] J. Hübner, J. Sichman, and O. Boissier. : A Middleware for Developing Organised Multi-agent Systems. *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, pages 64–78, 2006.

[30] Y. Labrou and T. Finin. Semantics and conversations for an agent communication language. *Readings in Agents*, pages 235–242, 1998.

[31] C. Lemaître and C. Excelente. Multi-agent organization approach. In *Proceedings of II Iberoamerican Workshop on DAI and MAS*, 1998.

[32] M. Luck, McBurney, S. P., O., and S. Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.

[33] V. Morreale, S. Bonura, G. Francaviglia, M. Cossentino, and S. Gaglio. PRACTIONIST: a new framework for BDI agents. In *Proceedings of the Third European Workshop on Multi-Agent Systems (EUMAS'05*, page 236, 2005.

[34] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. pages 42–55. Springer-Verlag, 1996.

[35] J. Rodríguez-Aguilar, F. Martín, P. Garcia, P.Ñoriega, and C. Sierra. Towards a formal specification of complex social structures in multi-agent systems. *Collaboration between Human and Artificial Societies*, pages 284–300, 1999.

[36] M. P. Singh and A. K. Chopra. Correctness properties for multiagent systems. In *DALT*, pages 192–207, 2009.

[37] E. D. Val, N. Criado, M. Rebollo, E. Argente, and V. Julian. Service-oriented framework for virtual organizations. In *International Conference on Artificial Intelligence (ICAI)*, volume 1, pages 108–114. CSREA Press, 2009.

[38] G. Valetto, G. E. Kaiser, and G. S. Kc. A mobile agent approach to process-based dynamic adaptation of complex software systems. In *EWSPT '01: Proceedings of the 8th European Workshop on Software Process Technology*, pages 102–116, London, UK, 2001. Springer-Verlag.

[39] C. Walton and D. Robertson. Flexible multi-agent protocols. *Proceedings of UKMAS*, 2002.

[40] M. Wooldridge. *An introduction to multiagent systems*. Wiley, 2009.