PROYECTO FIN DE CARRERA

Título:	Diseño e Implementación de una Arquitectura de Agentes Orientada a las Comunicaciones basada en JAIN SLEE
Autor:	Felipe Echanique Torres
Tutor:	Carlos Ángel Iglesias Fernández
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Gregorio Fernández Fernández
Vocal:	Mercedes Garijo Ayestarán
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	Marifeli Sedano Ruiz

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

PROYECTO FIN DE CARRERA

DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA DE AGENTES ORIENTADA A LAS COMUNICACIONES BASADA EN JAIN SLEE

Alumno: Felipe Echanique Torres

Tutor: Carlos Ángel Iglesias Fernández

2013

Dedicado a todos.

Resumen

Esta memoria presenta la arquitectura de agentes SLAA para el desarrollo de agentes en la Web orientados a las comunicaciones. La implementación se ha realizado mediante el estándar Java JAIN SLEE sobre el que se ha añadido el motor de reglas Drools para la programación de agentes basados en reglas. El comportamiento de un agente queda definido por las acciones realizadas tras el disparo de alguna de sus reglas. Las reglas se disparan ante la llegada de eventos. Estos eventos son los que se reciben en el servidor JAIN SLEE a través de los adaptadores de recursos y son generados normalmente por elementos externos aunque también se pueden generar de forma interna. Un agente tiene conocimiento de lo que sucede en su entorno a través de los eventos que recibe y puede interactuar con este mediante las acciones disponibles que se definen ante la llegada de eventos.

Palabras clave: Agentes, Arquitectura de agentes, Drools, HTML, JAIN SLEE, Java, JBoss Rules, Knockout.js, Mobicents, Motor de reglas.

Abstract

This Project describes the SLAA agent architecture for Web-based developing of communication oriented agents. The implementation has been done with the Java JAIN SLEE standard and the Drools rule engine for rule-based programming agents. The behavior of an agent is defined by the actions done after a rule is fired. The rules are triggered by the arrival of events. These events are received at the JAIN SLEE server through resource adapters and are typically generated by external elements but can also be generated internally. An agent is aware of what is happening in its environment through the events received, and it can interact with the environment via the available actions defined by the arrival of events.

Keywords: Agents, Agent architecture, Drools, HTML, JAIN SLEE, Java, JBoss Rules, Knockout.js, Mobicents, Rule engine.

Agradecimientos

Escribir estas líneas es señal inequívoca de que lo que comencé hace varios años está terminando. Han sido buenos años de carrera para lograr una de las metas más difíciles que me he propuesto hasta este momento. Empecé sin saber muy bien qué me iba a encontrar y ahora puedo asegurar que ha sido una de las mejores decisiones que he tomado nunca.

Tengo que agradecer sin duda especialmente a todo el Grupo de Sistemas Inteligentes de la Universidad Politécnica de Madrid la oportunidad que me ofreció de colaborar con ellos desde poco tiempo después de entrar en la carrera hasta el momento de finalizarla y por el apoyo y consejos que he recibido durante el desarrollo la carrera. La mayor parte de todo lo que he aprendido y de mi currículum lo he hecho allí. Atrás quedan años de grandes recuerdos y de gente que he conocido con la que espero mantener el contacto.

También agradezco a todas las personas que en algún momento de alguna forma me han ayudado a conseguir que haya sido posible el gran reto de ser ingeniero. Sin dar nombres para no dejarme a nadie, pero cualquiera que esté leyendo estas líneas sabe si tengo que estar agradecido.

Esto termina, pero no es el final. Es el comienzo de una nueva etapa y nuevos retos.

Índice general

Resumen vii

Abstract ix

Agradecimientos xi

Índice de Tablas xix

Índice de Figuras xxi

Índice de Códigos xxiii

1. Introducción 1

1.1. Estructura de la memoria . . . 3

1.2. Motivación . . . 3

1.3. Objetivos . . . 4

2. Tecnologías habilitadoras 5

2.1. Especificación JAIN SLEE . . . 7

2.1.1. Arquitectura SLEE . . . 8

2.1.2. Modelo de componentes . . . 8

2.1.2.1. Eventos y tipos de Evento . . . 8

2.1.2.2. Aplicaciones orientadas a eventos . . . 10

2.1.2.3. Componentes de las aplicaciones . . . 11

2.1.3. Recursos . . . 11

2.1.3.1. Adaptadores de recursos . . . 11

2.1.4. Bloques de construcción de servicios . . . 12
2.1.4.1. Componente SBB . . . 12
2.1.4.2. Árboles de SBBs y SBB raíz . . . 13
2.1.4.3. Entidad de un componente SBB . . . 14
2.1.4.4. Árboles de entidades SBB y entidad SBB raíz . . . 14
2.1.4.5. Prioridad de entrega de eventos . . . 15
2.1.5. Servicios . . . 16
2.1.5.1. Estados del servicio . . . 16
2.1.6. Actividades . . . 18
2.1.6.1. Objeto de Actividad . . . 18
2.1.6.2. Contexto de Actividad . . . 19
2.1.6.3. Interfaz del Contexto de Actividad . . . 20
2.1.7. Perfiles de datos . . . 21
2.1.8. Herramientas SLEE . . . 21
2.1.9. Plataformas JAIN SLEE . . . 22
2.1.9.1. Rhino . . . 22
2.1.9.2. Mobicents . . . 22
2.1.9.3. Elección de la plataforma . . . 23
2.2. Drools . . . 23
2.2.1. Beneficios de un motor de reglas . . . 23
2.2.2. Razonamiento y representación de conocimiento . . . 24
2.2.3. Motor de reglas y sistema de reglas de producción . . . 25
2.2.4. Sistemas de razonamiento híbrido . . . 25
2.2.5. Algoritmo Rete . . . 29
2.3. Conceptos de agentes . . . 34
2.3.1. Agentes . . . 34
2.3.2. Arquitectura de Agentes . . . 34
2.3.3. Arquitectura de Agentes basada en SLEE . . . 34

3. Análisis funcional **35**
3.1. Casos de uso . . . 37
3.1.1. Diccionario de actores . . . 37
3.1.2. Descripción de los casos de uso . . . 37
3.2. Análisis y captura de requisitos . . . 43

4. Diseño arquitectónico y detallado **45**
4.1. Descripción general del sistema . . . 47

4.2. Adaptadores de recursos . . . 49
4.2.1. Adaptador de recursos HTTP servidor . . . 49
4.2.2. Adaptador de recursos HTTP cliente . . . 51
4.2.3. Adaptador de recursos XMPP . . . 52
4.2.4. Adaptador de recursos SocialNetwork . . . 54
4.2.5. Adaptador de recursos Drools . . . 55
4.2.6. Otros adaptadores de recursos . . . 56
4.3. Actividades . . . 57
4.3.1. Actividad entre gestor y agentes . . . 57
4.3.2. Actividad entre adaptador de recursos y gestor o agentes . . . 58
4.4. Eventos . . . 58
4.4.1. Eventos internos de la arquitectura . . . 58
4.4.2. Eventos externos a la arquitectura . . . 59
4.4.3. Mapeado de eventos entre SLEE y Drools . . . 60
4.5. Herramientas SLEE . . . 60
4.5.1. Temporizadores . . . 60
4.5.2. Tablas de datos . . . 60
4.6. Servicios . . . 62
4.6.1. Servicio gestor de agentes . . . 62
4.6.1.1. AgentManagerSbb . . . 63
4.6.1.2. Descriptor de despliegue . . . 63
4.6.1.3. AgentManagerSBB.class . . . 66
4.6.1.4. Diagrama funcional . . . 71
4.6.2. Servicio agente . . . 73
4.6.2.1. SlaaAgentSbb . . . 73
4.6.2.2. Descriptor de despliegue . . . 73
4.6.2.3. SlaaAgentSBB.class . . . 75
4.6.2.4. Diagrama funcional . . . 78

5. Interfaz web de administración **81**
5.1. Introducción . . . 83
5.2. Vista de administración . . . 83
5.3. Vista de usuarios . . . 84

6. Caso de estudio: Agente enrutador de conversaciones **89**
6.1. Introducción . . . 91
6.2. Escenario . . . 91

6.3. Estados del agente . . . 92
6.3.1. Iniciar conversación . . . 92
6.3.2. Conversación . . . 94
6.3.3. Finalizar conversación . . . 95
6.4. Código del agente . . . 95
6.4.1. Clases importadas . . . 95
6.4.2. Regla de agente iniciado . . . 96
6.4.3. Regla de mensaje HTTP de inicio de conversación . . . 97
6.4.4. Regla de tarea temporizada . . . 98
6.4.5. Regla de mensaje HTTP de conversación . . . 99
6.4.6. Regla de mensaje XMPP de conversación . . . 100
6.4.7. Regla de mensaje XMPP de presencia . . . 100
6.4.8. Regla de petición de historial de conversación . . . 101

7. Conclusiones y trabajos futuros **103**
7.1. Conclusiones . . . 105
7.2. Trabajos futuros . . . 106

Bibliografía **107**

A. Configuración del servidor y entorno de desarrollo **109**
A.1. Repositorio del proyecto . . . 111
A.2. Configuración del sistema operativo . . . 111
A.3. Puesta en marcha del servidor . . . 112
A.4. Desarrollo en Eclipse . . . 112
A.5. Desplegar y compilar componentes . . . 112
A.5.1. Componentes propios de Mobicents . . . 113
A.5.2. Componentes desarrollados para SLAA . . . 113

B. Programación de un agente **115**
B.1. Esquema general de un agente . . . 117
B.2. Ciclo de vida . . . 117
B.3. Configuración . . . 118
B.4. Eventos recibidos por el agente . . . 120
B.4.1. OnAgentStarted . . . 120
B.4.2. OnAgentStopped . . . 120
B.4.3. OnXmppMessage . . . 120
B.4.4. OnXmppPresence . . . 120

B.4.5. OnHttpGet . . . 121
B.4.6. OnHttpResponse . . . 121
B.4.7. OnTask . . . 122
B.5. Interfaz del agente . . . 122
B.5.1. Perfiles de datos . . . 122
B.5.1.1. setProfile(profileName) . . . 122
B.5.1.2. delProfile(profileName) . . . 122
B.5.1.3. delAllProfiles() . . . 122
B.5.2. Parámetros y valores . . . 122
B.5.2.1. setParam(profileName, paramName, paramValue) . . . 122
B.5.2.2. delParam(profileName, paramName) . . . 123
B.5.2.3. getStringParam(tableName, paramName) . . . 123
B.5.2.4. getIntParam(tableName, paramName) . . . 123
B.5.2.5. getLongParam(tableName, paramName) . . . 123
B.5.3. Servicios . . . 123
B.5.3.1. sendXmppMessage(user, message) . . . 123
B.5.3.2. sendHttpGet(url) . . . 123
B.5.3.3. sendTweet(message) . . . 123
B.5.4. Temporizadores . . . 124
B.5.4.1. setTask(taskID, repetitions, duration) . . . 124
B.5.4.2. delTask(taskID) . . . 124
B.5.4.3. delAllTasks() . . . 124
B.5.5. Trazas . . . 124
B.5.5.1. info(message) . . . 124
B.5.5.2. error(message) . . . 124
B.6. Programación de un agente . . . 125

Glosario de términos **127**

Índice de tablas

3.1. Diccionario de actores 37
3.2. Caso de uso gestionar la plataforma 39
3.3. Caso de uso crear un usuario 40
3.4. Caso de uso eliminar un usuario 40
3.5. Caso de uso crear un agente 41
3.6. Caso de uso eliminar un agente 41
3.7. Caso de uso programar un agente 42
3.8. Caso de uso iniciar/parar un agente 42
3.9. Caso de uso programar SLAA 43
3.10. Requisitos de la arquitectura SLAA 44

Índice de figuras

2.1. Arquitectura SLEE 9
2.2. Ejemplo de árbol de SBBs 14
2.3. Ejemplo de árbol de entidades SBB 15
2.4. Estados operacionales de un Servicio 17
2.5. Ejemplo de Actividad 18
2.6. Ejemplo de Contexto Actividad 20
2.7. Ejemplo de Perfiles de datos 21
2.8. Vista de alto nivel de un Sistema de Reglas de Producción 26
2.9. Encadenamiento hacia delante 27
2.10. Encadenamiento hacia atrás 28
2.11. Nodos Rete 29
2.12. Nodos ObjectTypeNode 30
2.13. Nodos AlphaNode 30
2.14. Nodos JoinNode 31
2.15. Compartición de nodos 33

3.1. Diagrama de casos de uso 38

4.1. Arquitectura SLAA 48
4.2. Diagrama de secuencias HTTP server RA 50
4.3. Diagrama de secuencias HTTP client RA 52
4.4. Diagrama de secuencias XMPP RA 53
4.5. Diagrama de secuencias SocialNetwork RA 55
4.6. Diagrama de secuencias Drools RA 56
4.7. Ejemplo de tablas de datos 61

4.8. Diagrama funcional del gestor de agentes 72
4.9. Diagrama funcional de un agente 79

5.1. Vista de usuarios 83
5.2. Vista de agentes 84
5.3. Vista de agentes del usuario 85
5.4. Vista de configuración del agente 85
5.5. Vista de reglas del agente 86
5.6. Vista de trazas del agente 87

6.1. Escenario del caso de estudio 91
6.2. Secuencia de inicio de conversación 92
6.3. Ventana de conversación iniciada 93
6.4. Ventana de conversación fallida 93
6.5. Secuencia de mensaje de conversación 94
6.6. Ventana de conversación en curso 94
6.7. Secuencia de fin de conversación 95
6.8. Ventana de conversación finalizada 96

B.1. Esquema general de un agente 117
B.2. Ciclo de vida de un agente 118

Índice de códigos

2.1. Las dos partes de un sistema de reglas de producción 25
2.2. Compartición de nodos: Regla 1 32
2.3. Compartición de nodos: Regla 2 32
4.1. agent-manager-service.xml 62
4.2. agent-manager-description 63
4.3. agent-manager-classes 63
4.4. agent-manager-ra-binding 64
4.5. agent-manager-events 65
4.6. agent-manager-profile 66
4.7. AgentManagerSBB setSbbContext 66
4.8. AgentManagerSBB onEvents 67
4.9. AgentManagerSBB onGET 68
4.10. AgentManagerSBB onInitialEventSelect 69
4.11. AgentManagerSBB startAgent 69
4.12. AgentManagerSBB CMP 70
4.13. AgentManagerSBB profiles 70
4.14. slaa-agent-service.xml 73
4.15. slaa-agent-ra-binding 73
4.16. slaa-agent-events 74
4.17. SlaaAgentSBB setSbbContext 75
4.18. SlaaAgentSBB onStartAgentEvent 76
4.19. SlaaAgentSBB onMessage 77
4.20. SlaaAgentSBB fireRules 77
4.21. SlaaAgentSBB sendXmppMessage 78

6.1. Clases importadas del agente enrutador 96
6.2. Regla de agente iniciado 96
6.3. Regla de mensaje HTTP de inicio de chat 97
6.4. Regla de tarea temporizada 98
6.5. Regla de mensaje HTTP de chat 99
6.6. Regla de mensaje XMPP de chat 100
6.7. Regla de mensaje XMPP de presencia 101
6.8. Regla de petición de historial de conversación 102
A.1. Árbol de directorios del proyecto 111
A.2. Variables de entorno 111
B.1. JSON de configuración de un agente 119
B.2. agentRulesExample.drl 125

Capítulo 1

Introducción

Este primer capítulo es una introducción al proyecto fin de carrera. En él se hará una descripción de la estructura del documento para dar una vista del contenido de cada capítulo, la motivación que impulsó el diseño e implementación del proyecto y los objetivos propuestos.

1.1. Estructura de la memoria

Para dar a conocer el contenido de cada capítulo, en esta sección se va a introducir cada uno de ellos con una breve explicación de su contenido. El documento completo consta de siete capítulos:

- Primer capítulo: se trata de una introducción al proyecto fin de carrera para dar a conocer una primera visión del proyecto, cuales fueron las motivaciones, objetivos y cómo está estructurado.
- Segundo capítulo: en este capítulo se presentan las principales tecnologías que han hecho posible la realización de este proyecto.
- Tercer capítulo: es un análisis funcional del proyecto que muestra las acciones que podrán realizar los usuarios con el sistema desarrollado y los requisitos definidos.
- Cuarto capítulo: presenta la arquitectura completa del sistema comenzando por una breve descripción de todos los elementos y posteriormente un análisis en mayor profundidad.
- Quinto capítulo: uno de los elementos clave del desarrollo es la interfaz web a través de la que se interactúa con el sistema. En este capítulo se presentan sus características.
- Sexto capítulo: para dar a conocer qué se puede hacer con el desarrollo fruto del proyecto, se describe un ejemplo real de funcionamiento.
- Séptimo capítulo: finalmente el proyecto culmina con el análisis de las conclusiones a las que se ha llegado y las posibles mejoras que se proponen sobre el trabajo realizado.

1.2. Motivación

Tras haber trabajado con diferentes tecnologías de agentes (especialmente JADE [5]), uno de los problemas encontrados es la dificultad añadida en el desarrollo de agentes cuando se tiene como objetivo la comunicación con otros sistemas o servicios.
Esto es debido a que es responsabilidad del desarrollador gestionar todos los elementos que hacen posible que un agente pueda comunicarse con otros recursos. Estos elementos consisten en la creación de hilos, establecimiento de comunicaciones, generación de mensajes de protocolos, etc...
Esta dificultad añadida hace que el desarrollo de un agente se vea obstaculizado por problemas que se alejan de la programación en sí de un agente y alargan en coste y tiempo su finalización.

Es por ello que tras conocer la tecnología JAIN SLEE surge la idea de implementar una arquitectura de agentes siguiendo la filosofía de desarrollo establecida en la especificación SLEE. JAIN SLEE permite crear servicios capaces de utilizar múltiples protocolos para comunicarse sin necesidad de tener que elaborar el mecanismo que lo permite por parte del desarrollador del servicio. Al seguir el estándar, un servicio desarrollado en SLEE solo tiene que especificar qué recursos va a necesitar y de qué manera, y el desarrollador se preocupa únicamente de las funcionalidades del servicio que quiere desarrollar.
Con todo esto se decide diseñar e implementar una arquitectura de agentes basada en JAIN SLEE con la ventaja añadida de que el desarrollador no tiene que conocer cómo programar siguiendo la especificación SLEE. Para obtener este objetivo se decide añadir una capa de abstracción sobre la programación en SLEE incluyendo el motor de reglas Drools como única vía para la programación de los agentes.
Finalmente lo que se pretende es que se puedan desarrollar agentes conociendo únicamente la programación usando el sistema Drools y el lenguaje Java, abstrayendo al programador completamente de lo que sucede en JAIN SLEE y de esta forma tener un sistema para programar agentes capaces de comunicarse con múltiples recursos conectados a Internet de una forma ágil.

1.3. Objetivos

El objetivo principal de este proyecto ha sido el diseño y la implementación de una arquitectura de agentes orientada a las comunicaciones basada en JAIN SLEE.
Para conseguir este objetivo principal ha sido necesario completar otros objetivos:

- Estudio de las principales tecnologías que permiten su funcionamiento.
- Análisis de los principales requisitos presentes una arquitectura de agentes.
- Diseño e implementación de un prototipo sobre JAIN SLEE sin motor de reglas.
- Estudio de la viabilidad del uso de un motor de reglas para programar agentes.
- Implementación funcional del sistema sobre JAIN SLEE usando el motor de reglas Drools.
- Desarrollo de un caso de estudio real utilizando el sistema desarrollado.
- Obtención de conclusiones y propuesta de mejoras para el futuro.

Capítulo 2

Tecnologías habilitadoras

En este capítulo se describen las tecnologías clave que se han empleado en la elaboración del proyecto. En primer lugar se hará la introducción a los elementos más importantes de la especificación JAIN SLEE y el motor de reglas Drools y finalmente una descripción de los conceptos básicos de una arquitectura de agentes.

2.1. Especificación JAIN SLEE

Qué es JAIN SLEE

JAIN (Java APIs for Integrated Networks) SLEE [1] es el acrónimo de Entorno de Ejecución de Lógica de Servicios. Es un servidor de aplicaciones o un contenedor de componentes software. La especificación SLEE está diseñada y optimizada para aplicaciones orientadas a eventos que requieren baja latencia (<100 ms), alta disponibilidad para el proceso de eventos (1000 eventos/s) y permitir implementaciones que soporten escalabilidad y disponibilidad a través de *clustering*.

Principales beneficios

- Plataforma de alto rendimiento para aplicaciones simples o complejas orientadas a eventos y desarrollo centrado únicamente en la lógica del servicio quedando delegado al contenedor SLEE las tareas relacionadas con la creación de hilos, transacciones, etc...
- Estructura de aplicaciones estándar. Modelo de programación Java definido, orientado a objetos, asíncrono, robusto y distribuible.
- Independiente de las redes subyacentes.
- Soporte Asíncrono. Mecanismo de distribución de eventos robusto. Mapeo de eventos a invocación de métodos en los componentes desarrollados. Creación de componentes por parte de SLEE en respuesta a eventos iniciales.
- Componentes reutilizables.
- Disponibilidad y escalabilidad a través de *clustering*.
- Fácil desarrollo de componentes robustos.
- El contenedor SLEE se encarga de manejar el ciclo de vida de los componentes.
- Perfiles de datos de fácil definición para registrar y acceder a información persistente por parte de los componentes desarrollados.
- Punto de integración de múltiples protocolos y recursos. Un sólo contenedor SLEE es capaz de manejar diversos protocolos y recursos. Facilidad para integrar nuevas tecnologías.

2.1.1. Arquitectura SLEE

En la figura 2.1 se puede ver un esquema completo de todos los elementos que componen la arquitectura SLEE. En las siguientes secciones se detallarán las características principales de todos los elementos reflejados en el esquema para comprender el funcionamiento de JAIN SLEE.
De forma introductoria se presentan los siguientes elementos:

- Aplicación de gestión (*Management application*).
  Es un entorno que permite administrar y gestionar todos los componentes ejecutándose en la plataforma SLEE.

- Herramientas SLEE (*SLEE facilities*).
  Son componentes internos de SLEE que que ofrecen diferentes funcionalidades que se pueden usar dependiendo del tipo de aplicación que se esté desarrollando. Por ejemplo, los temporizadores permiten que una aplicación reciba un evento cuando expira un tiempo programado.

- Adaptadores de Recursos (*Resource adaptors*).
  Adaptan los recursos de red a SLEE. Los recursos de red son los elementos externos a SLEE con los que interactúan las aplicaciones.

- Instancias de Componentes.
  Son las instancias de los bloques que forman y permiten el funcionamiento de las aplicaciones desarrolladas y ejecutándose en SLEE.

2.1.2. Modelo de componentes

El modelo de componentes de SLEE está enfocado a las aplicaciones orientadas a eventos, también conocidas como aplicaciones asíncronas. Estas aplicaciones se caracterizan por recibir peticiones en forma de eventos.

2.1.2.1. Eventos y tipos de Evento

Un evento representa un suceso que requiere el procesado por parte de una aplicación. El evento transporta la información que describe el suceso, como por ejemplo la fuente de origen del evento. Un evento puede ser originado por:

- Un recurso externo como la pila de un protocolo de comunicaciones. Un ejemplo puede ser un mensaje GET o POST de HTTP.

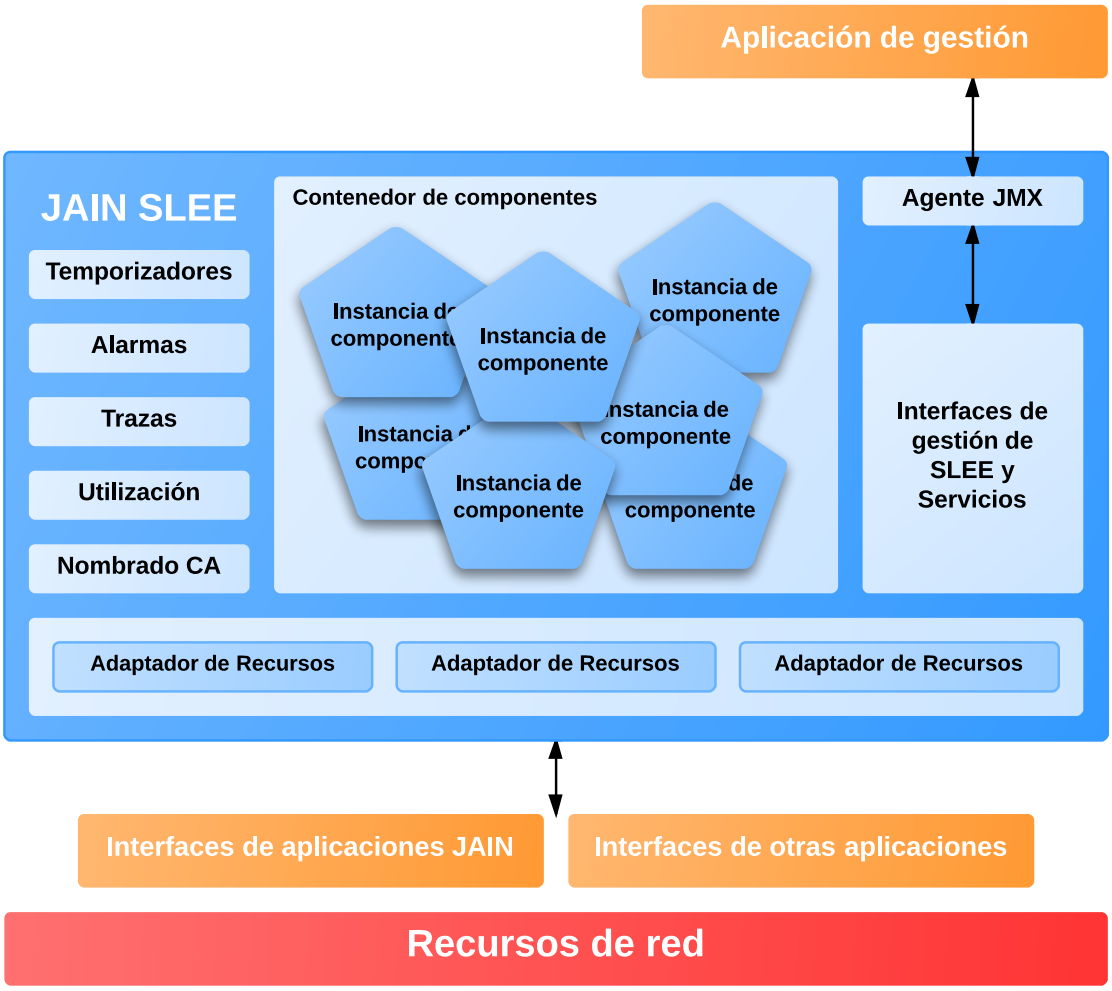

Figura 2.1: Arquitectura SLEE

- Dentro del contenedor SLEE.
  SLEE usa eventos para comunicar cambios que pueden ser de interés para aplicaciones ejecutándose en SLEE. Por ejemplo, la herramienta de temporización (*Timer Facility*) emite un evento cuando un temporizador configurado llega a cero para avisar a la aplicación que lo solicitó.

- Una aplicación ejecutándose en SLEE.
  Una aplicación ejecutándose en SLEE puede hacer uso de los eventos para señalizar, invocar o comunicarse con otras aplicaciones ejecutándose en SLEE.

Todo evento en SLEE está definido por un tipo de evento (*event type*). El tipo de evento de un evento determina cómo el evento es enrutado a diferentes componentes de aplicación.

2.1.2.2. Aplicaciones orientadas a eventos

Una aplicación orientada a eventos no suele tener un hilo de ejecución. Normalmente, una aplicación orientada a eventos define métodos que son invocados cuando los eventos llegan a la aplicación. Estos métodos contienen código de la aplicación que inspecciona el evento y realiza un procesado para manejar el evento. El código de la aplicación puede interactuar con el recurso que emitió el evento u otros recursos, lanzar nuevos eventos o actualizar el estado de la aplicación.
Un ejemplo común de aplicación orientada a eventos es una aplicación que implementa una máquina de estados. Una máquina de estados maneja estados y transiciones. Una transición sucede cuando hay un suceso que causa que la máquina de estados se mueva de un estado a otro. La transición puede ser registrada mediante la computación que tiene lugar durante la transición.
Cuando una aplicación implementa una máquina de estados, las variables de la aplicación determinan el estado de la máquina, los eventos representan sucesos en las transiciones de estado, y en el código de la aplicación se computan las acciones durante las transiciones.
Una forma común de construir una aplicación orientada a eventos es disponer de un solo método para manejar todos los eventos. Cuando el método recibe un evento, lo inspecciona y ejecuta el procesado adecuado en función del tipo de evento. El modelo de componentes de SLEE modela la interfaz externa de una aplicación orientada a eventos como un conjunto de eventos que la aplicación puede recibir. Cada tipo de evento es manejado por su propio método, esto refuerza una interfaz de eventos bien definida. El modelo de componentes orientado a eventos permite a SLEE proveer la lógica de enrutado de eventos para la aplicación.

2.1.2.3. Componentes de las aplicaciones

La arquitectura SLEE define cómo una aplicación puede ser compuesta de componentes. Estos componentes son conocidos como bloques de construcción de servicios (*Service Building Block*).
Cada componente SBB tiene definido los tipos de eventos aceptados y tiene métodos para manejar cada tipo de evento. Estos métodos son los que contienen el código de la aplicación que procesa los eventos. Además un componente SBB puede tener una interfaz para invocación de métodos síncronos por parte de otros componentes SBB.
En tiempo de ejecución, SLEE crea instancias de esos componentes para procesar los eventos y elimina los que ya no van a recibir más eventos.

2.1.3. Recursos

Un recurso representa a un sistema que es externo a SLEE. Ejemplos de esto incluyen dispositivos de red, pilas de protocolos y bases de datos. Estos recursos pueden tener o no API Java. Los recursos con APIs Java definen clases Java o interfaces que representan los eventos emitidos por el recurso.

2.1.3.1. Adaptadores de recursos

La arquitectura SLEE define cómo las aplicaciones ejecutándose dentro de SLEE interactúan con los recursos a través de los adaptadores de recursos. Los adaptadores de recursos son los componentes que adaptan los recursos a los requerimientos de SLEE. La arquitectura SLEE define los siguientes conceptos para los adaptadores de recursos.

- Tipo de adaptador de recursos (*resource adaptor type*).
  El tipo de un adaptador de recursos declara las características comunes para un conjunto de adaptadores de recursos. Define las interfaces Java que deben ser implementadas por los adaptadores de recursos que pertenezcan a un mismo tipo. Una de estas interfaces es conocida como la interfaz del adaptador de recursos (*resource adaptor interface*). Además define los tipos de eventos que serán emitidos por los adaptadores de recursos de un mismo tipo dentro del contenedor SLEE. La especificación SLEE incluye recomendaciones para los tipos de adaptadores de recursos JCC, SIP y TCAP.

- Adaptador de recursos (*resource adaptor*).
  Un adaptador de recursos es la implementación de uno o más tipos de adaptadores de recursos. Puede haber también múltiples implementaciones del mismo tipo de adaptador de recursos. Un adaptador de recursos consiste en un conjunto de clases Java y un descriptor de despliegue (*deployment descriptor*). Debe incluir una clase Java que

implemente la interfaz del adaptador de recursos de su tipo de adaptador de recursos. Un adaptador de recursos puede ser suministrado por un proveedor del recurso externo o un proveedor SLEE que adapte ese recurso.

- Entidad del adaptador de recursos (*resource adaptor entity*).
  La entidad de un adaptador de recursos es una instancia del adaptador de recursos. Se pueden instanciar múltiples entidades de un mismo adaptador de recursos. Por ejemplo se pueden configurar varios adaptadores de recursos que actúen como Proxy SIP para llamadas VoIP. Cada uno de esos adaptadores de recursos puede estar atendiendo en la misma IP pero sobre diferentes puertos para determinadas funciones. Cada uno de esos adaptadores de recursos sería una entidad diferente del mismo adaptador de recursos SIP.

2.1.4. Bloques de construcción de servicios

2.1.4.1. Componente SBB

Un componente SBB está definido por los siguientes elementos:

- Tipo de eventos que puede recibir y emitir el componente SBB.
- Estado de la instancia SBB.
  El estado de una instancia SBB se debe mantener en un campo de persistencia gestionado por el contenedor *(Container Managed Persistent)* para mantener datos persistentes del SBB.
- Métodos de los eventos.
  El componente SBB dispone de un método para manejar cada tipo de evento recibido en el componente SBB. El método que maneja el evento contiene la lógica de la aplicación para procesar eventos de un tipo de evento específico. El componente SBB puede declarar también un tipo de evento para ser emitido dentro del contenedor SLEE.
- Métodos de la interfaz local de un SBB.
  Un componente SBB declara la interfaz local del componente SBB (*SBB local interface*). Esta interfaz especifica los métodos del componente SBB que pueden ser invocados de forma síncrona. La interfaz local de una instancia del componente SBB solo puede ser invocada por otra instancia de componente SBB dentro del mismo árbol de instancias de componentes SBB. Además de la interfaz, el componente SBB provee la implementación de los métodos de su interfaz local.
- Relaciones padre-hijo.
  El componente SBB puede tener relacionados cero o más componentes SBB hijos. El

componente SBB tiene que tener especificadas sus relaciones con componentes SBB hijos. En el descriptor de despliegue del componente SBB se indica: qué otros componentes SBB corresponderán como hijos, un método para acceder a estas relaciones y un nivel de prioridad a la hora de recibir eventos en cada componente SBB hijo.

- Datos compartidos.
  Un componente SBB define los datos que quiere compartir con otros componentes como un conjunto de atributos dentro del Contexto de Actividad (*Activity Context*). Cada atributo del Contexto de Actividad tiene un nombre y un tipo. Estos atributos se almacenan en uno o más Contextos de Actividad. El componente SBB define una interfaz del Contexto de Actividad (*Activity Context Interface*) que provee métodos accesores para establecer u obtener sus valores.

2.1.4.2. Árboles de SBBs y SBB raíz

El desarrollador de un SBB compone el SBB especificando sus relaciones padre-hijo. Un SBB dentro de el grafo de SBBs corresponde a un nodo y las relaciones padre-hijo son flechas dirigidas desde el nodo SBB padre al nodo hijo. En cada flecha se indica la prioridad por defecto que tendrá la entrega de eventos en la relación padre-hijo. Un padre puede tener cero o más relaciones padre-hijo pudiendo incluirse a sí mismo como hijo. En el ejemplo de la figura 2.2 el desarrollador de SBB ha creado tres componentes SBB: el X, Y, y Z. El SBB X puede tener el Y y el Z como hijos. El SBB Y puede tener el Z como SBB hijo. El SBB Z puede tener el SBB Y además de sí mismo como hijo.
Un SBB puede ser padre de cero o más SBBs incluído él mismo. En la figura 2.2, Z es padre de Y y de si mismo. Al mismo tiempo puede ser hijo de cero o más SBB incluído él mismo. Z puede ser hijo de X, Y, y él mismo.
Un SBB raíz (*SBB root*) es un SBB que puede ser instanciado por SLEE para procesar eventos. Un SBB root debe declarar al menos un evento de tipo inicial (*initial event*) para que SLEE pueda crear la instancia del SBB. Un SBB es declarado raíz porque sus instancias pueden ser raíces de árboles de entidades de SBB.
Normalmente un SBB raíz representa un -servicio completo-. Por ejemplo, el desarrollador de SBB puede crear un SBB de bloqueo de llamadas y un SBB de redirección de llamadas para implementar el servicio de -bloqueo de llamadas- y el servicio de -redirección de llamadas-. Estos SBB serían raices porque instancias de estos SBB se pueden utilizar para bloquear o redireccionar llamadas respectivamente, pero el desarrollador de SBB puede crear un nuevo SBB raíz de bloqueo y redirección de llamadas para implementar el servicio de -bloqueo y redirección de llamadas- utilizando los SBB anteriores de bloqueo y redirección.
Un SBB padre puede estar relacionado con el mismo SBB hijo a través de múltiples rela-

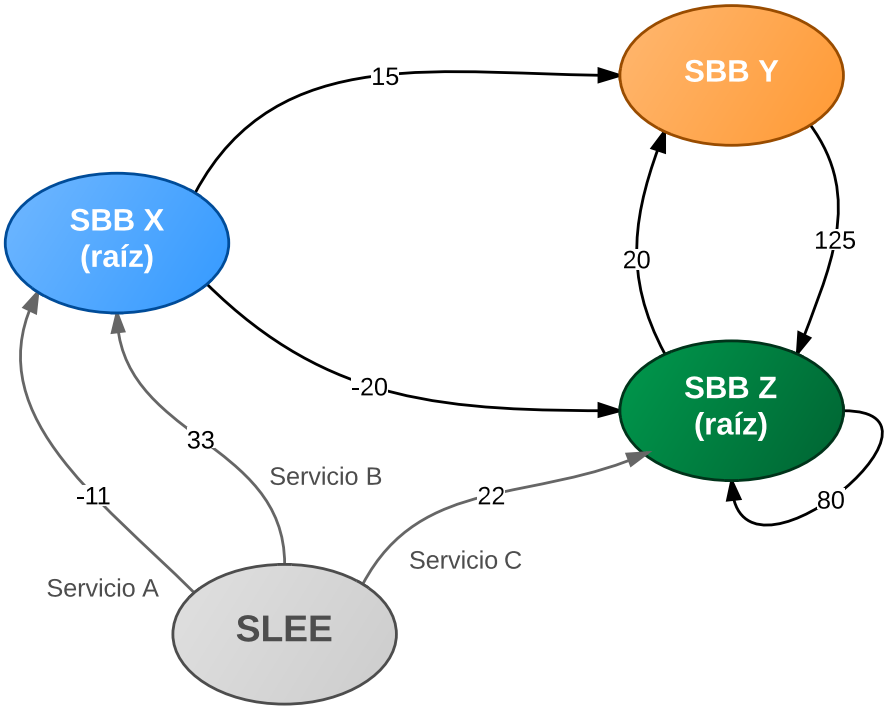

Figura 2.2: Ejemplo de árbol de SBBs

ciones. Para este tipo de relaciones se debe asignar una prioridad diferente de entrega de eventos.

2.1.4.3. Entidad de un componente SBB

Una entidad SBB (*SBB entity*) es una instancia de un componente SBB. Es una entidad lógica que representa el estado persistente de la instancia de un componente SBB.

2.1.4.4. Árboles de entidades SBB y entidad SBB raíz

Una entidad SBB puede crear cero o más entidades SBB en tiempo de ejecución, pero una entidad SBB solo puede ser creada una vez, por lo que una entidad SBB solo puede tener un único padre.
El árbol de entidades SBB representa las relaciones padre-hijo entre entidades SBB. Un nodo en este árbol representa una entidad SBB. Una flecha indica la relación padre-hijo entre dos entidades SBB. La etiqueta hace referencia a la prioridad de entrega de eventos relativo a los SBB hermanos.
SLEE solo creará instancias de SBBs raíz, que serán entidades SBB raíz. Estas entidades SBB son raíces porque son los nodos raíz de sus respectivos árboles de entidades SBB. Una

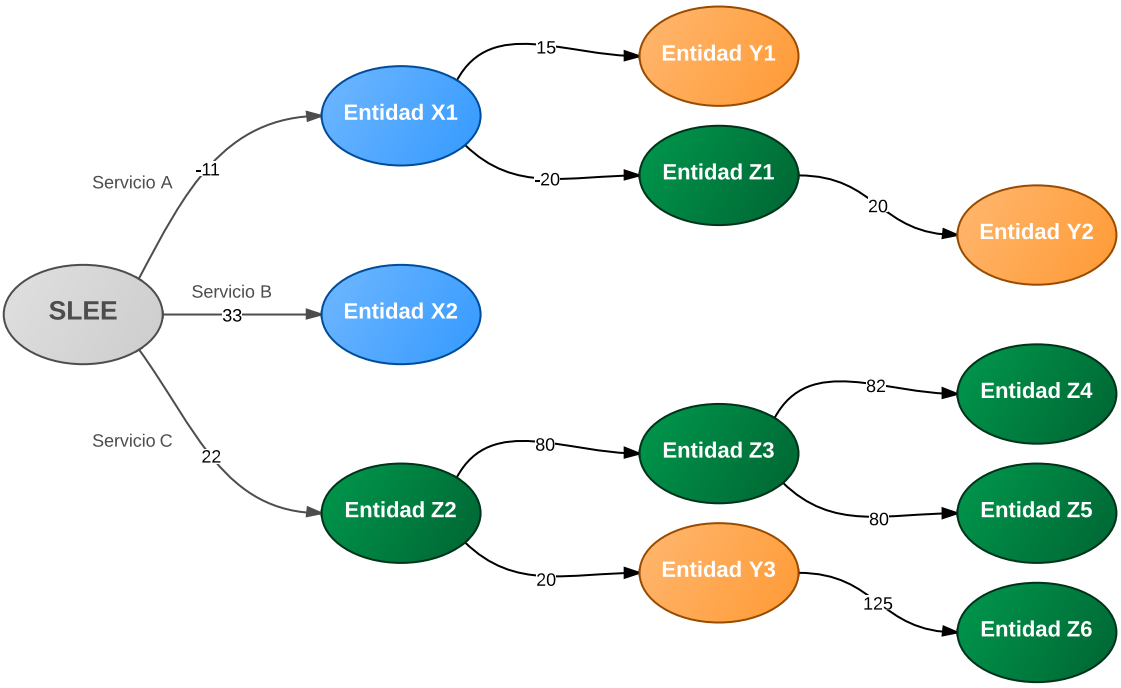

Figura 2.3: Ejemplo de árbol de entidades SBB

entidad SBB pertenece únicamente a un árbol de entidades SBB.

Por ejemplo, SLEE puede instanciar un SBB -bloqueo y redirección de llamadas- raíz. Esta entidad SBB podrá crear cero o más entidades SBB hijos -bloqueo de llamadas- o -redirección de llamadas- si es necesario. Este árbol de entidades SBB tiene un SBB raíz y cero o más flechas desde el SBB raíz hacia la entidad SBB -redirección de llamadas- o -bloqueo de llamadas-.

Para entender que que un SBB hijo de un SBB raíz puede ser a su vez SBB raíz en este ejemplo, el SBB de -bloqueo de llamadas- es un SBB raíz. SLEE puede crear una entidad SBB raíz -bloqueo de llamadas- para un servicio que requiera únicamente bloqueo de llamadas. Este árbol de entidades SBB solo contendría un SBB raíz.

En la figura 2.3 se pueden ver varios árboles de entidades SBB que pueden ser instanciados del árbol de SBB de la figura 2.2. Además muestra a SLEE como el *padre lógico* de todas las entidades SBB raíz. X1, X2 y Z2 son las entidades SBB raíz de este ejemplo. X1, Y1, Z1, e Y2 pertenecen al árbol de entidades SBB con raíz X1. X2 pertenece al árbol de entidades SBB con raíz X2. Z2, Z3, Y3, Z4, Z5, Z6 pertenecen al árbol de entidad SBB con raíz Z2.

2.1.4.5. Prioridad de entrega de eventos

Un evento es entregado en el orden en el que cada entidad SBB se interesó por recibir el evento. Una entidad SBB padre siempre recibe el mismo evento antes que sus entidad SBB

hijas. La prioridad de entrega de eventos de una entidad SBB determina el orden en el que cada entidad hermana recibirá el evento.
El SBB padre especifica una prioridad por defecto de entrega de eventos para cada relación con sus hijos. Cuando un SBB padre crea un SBB hijo, SLEE asigna la prioridad por defecto especificada en la relación padre-hijo. En tiempo de ejecución las prioridades de entrega de eventos pueden ser cambiadas. En el ejemplo de la figura 2.3, todos los SBB hijo mantienen su prioridad por defecto de entrega de eventos especificada en su relación padre-hijo, excepto la entidad SBB Z4, en la que se ha cambiado la prioridad al valor 82.

2.1.5. Servicios

Un Servicio es un elemento de despliegue. En él se especifica la relación padre-hijo entre un SBB raíz y SLEE como *padre lógico*. Para ello se identifica el SBB que será raíz y su prioridad por defecto de entrega de eventos del Servicio respecto a otros SBB raíz que vayan a recibir los mismos tipos de evento. Además provee otra información necesaria para que SLEE pueda instanciar entidades SBB raíz del Servicio porque SLEE tiene que determinar cuándo es necesario crear una nueva entidad SBB o procesar el evento por parte de una entidad SBB existente.
Para definir un Servicio, el Administrador provee un descriptor de despliegue del Servicio en el que se indica lo siguiente:

- Nombre global y único del servicio.
- Referencia a un SBB que será raíz.
- Una referencia opcional a una Tabla de Perfiles de Direcciones.
  Esta tabla contiene datos de direcciones que usará SLEE para determinar cuando es necesario crear una nueva entidad SBB raíz o enrutar el evento a una entidad SBB existente. No es necesario utilizar esta tabla, SLEE posee diversas formas de obtener este mismo objetivo.

Dos Servicios distintos pueden hacer referencia al mismo SBB raíz, pero estos Servicios deben provisionar diferentes prioridades de entrega de eventos y diferentes Tablas de Perfiles de Direcciones para que el contenedor SLEE pueda determinar cómo enrutar los eventos a las aplicaciones.

2.1.5.1. Estados del servicio

Un Servicio puede estar en uno de los siguientes estados operacionales.

Figura 2.4: Estados operacionales de un Servicio

- Inactivo.
  El Servicio ha sido instalado correctamente en el contenedor SLEE y está listo para ser activado. El Servicio no está en ejecución.

- Activo.
  El Servicio ha sido activado. Está en ejecución. SLEE creará una entidad SBB raíz de el SBB raíz del Servicio para recibir eventos iniciales e invocar entidades SBB del árbol de entidades SBB del Servicio.

- Parando.
  El Servicio está siendo desactivado. Sin embargo, algunas entidades SBB del árbol del Servicio todavía no han completado sus procesos. SLEE esperará a estas entidades SBB. Una entidad SBB ha completado su proceso y puede ser eliminada cuando ella y todas sus entidades SBB hijas no están ligadas a ningún Contexto de Actividad.

Los siguientes pasos describen el ciclo de vida de un Servicio:

- Un Servicio entra en estado Inactivo cuando es instalado correctamente en SLEE.

- Un Servicio entra en estado Activo desde Inactivo cuando el Servicio es Activado. En este punto SLEE puede iniciar una Actividad de Servicio (*Service Activity*) para el Servicio y emitir un evento de tipo inicial *Service Started Event* en esa Actividad.

- Un Servicio entra en el estado Parando del estado Activo cuando el Servicio es desactivado. En este punto, SLEE termina la actividad *Service Activity* asociada con el Servicio, si existe, y emite un evento de fin de actividad (*Activity End Event*) en esa

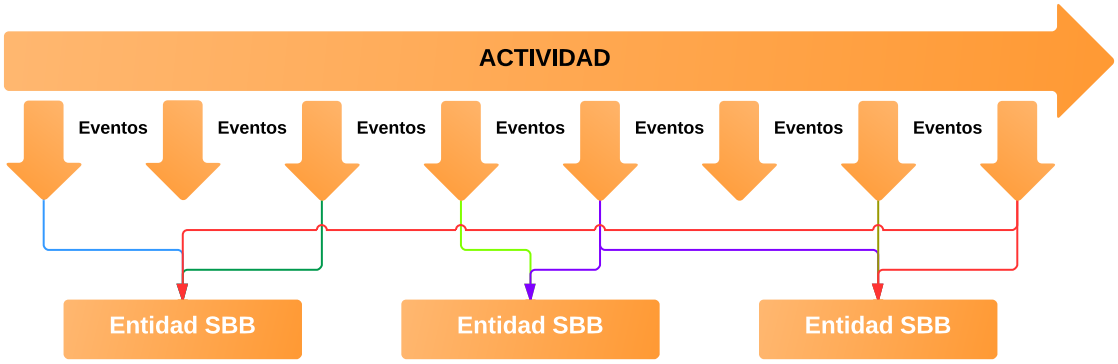

Figura 2.5: Ejemplo de Actividad

Actividad. Otras entidades SBB que pertenezcan al Servicio deben recibir también este evento para terminar sus procesos de forma correcta. Opcionalmente SLEE puede decidir eliminar forzosamente las entidadades SBB del árbol del Servicio si es necesario.

- SLEE mueve el estado a Inactivo desde Parando cuando todas las entidades SBB del árbol del Servicio han completado sus procesos.

El estado operacional de un Servicio es persistente. Aunque el contenedor SLEE sea reiniciado, este recuerda el estado en el que se encontraba el Servicio y restaura el estado operacional previo del Servicio.

2.1.6. Actividades

Una Actividad representa un hilo de eventos relacionados. Estos eventos representan los sucesos que han ocurrido en la entidad representada por la Actividad. Desde el punto de vista de los recursos, una Actividad representa una entidad entre el recurso que emite eventos y los componentes que lo reciben. En la figura 2.5 se puede ver un ejemplo de varias entidades SBB recibiendo eventos en una Actividad.
Un ejemplo típico de Actividad es una llamada telefónica desde que comienza hasta que termina.

2.1.6.1. Objeto de Actividad

El objeto de una Actividad (*Activity object*) es el objeto java que encapsula una Actividad y puede disponer de métodos para interactuar con la Actividad. Cada tipo de adaptador de recursos puede definir uno o más tipos de objetos de Actividad. Habitualmente los objetos de Actividad son gestionados por las entidades del adaptador de recursos.

En el ejemplo de una llamada telefónica el objeto Actividad puede contener información de llamante y llamado, duración de llamada, etc...

2.1.6.2. Contexto de Actividad

SLEE utiliza el Contexto de Actividad (*Activity Context*) para representar y encapsular un objeto de Actividad dentro del contenedor SLEE en una entidad lógica que no tiene API Java visible. Hay una relación uno-a-uno entre un objeto Actividad y un Contexto de Actividad.

- Un Contexto de Actividad además es un almacén de atributos que pueden ser compartidos entre múltiples entidades SBB que interactúan con el objeto Actividad representado mediante el Contexto de Actividad. Estas entidades SBB pueden leer y modificar los atributos almacenados en el Contexto de Actividad.

- Un Contexto de Actividad es también un canal de eventos que acepta eventos emitidos en el Contexto de Actividad y distribuye esos eventos a las entidades SBB ligadas a ese Contexto de Actividad. Una entidad SBB puede invocar a otra entidad SBB de manera asíncrona emitiendo eventos dentro de su Contexto de Actividad.

Una entidad SBB puede estar ligada a uno o más Contextos de Actividad. Una entidad SBB solo puede recibir eventos emitidos en un Contexto de Actividad al que está ligado.
SLEE no mantiene la cuenta de cuántas veces una entidad SBB se ha ligado o desligado de un Contexto de Actividad. Ligarse (*attach*) múltiples veces consecutivas o desligarse (*detach*) varias veces se tiene en cuenta como una única vez.
SLEE liga o desliga una entidad SBB de un Contexto de Actividad en las siguientes situaciones:

- Cuando se crea una entidad SBB raíz para procesar un evento inicial, SLEE liga la nueva entidad SBB raíz al Contexto de Actividad en el que el evento inicial fue emitido.

- Cuando termina el objeto Actividad correspondiente a un Contexto de Actividad, SLEE desliga las entidades SBB ligadas a ese Contexto de Actividad.

- Antes de que SLEE elimine una entidad SBB como consecuencia de una eliminación en cascada de un árbol de entidades SBB, SLEE desliga la entidad SBB de todos sus Contextos de Actividad.

Una entidad SBB también puede invocar explícitamente métodos para determinar a qué Contextos de Actividad está ligado él u otras entidades SBB dentro del mismo árbol de entidades SBB o desligarse de Contextos de Actividad. Por ejemplo,

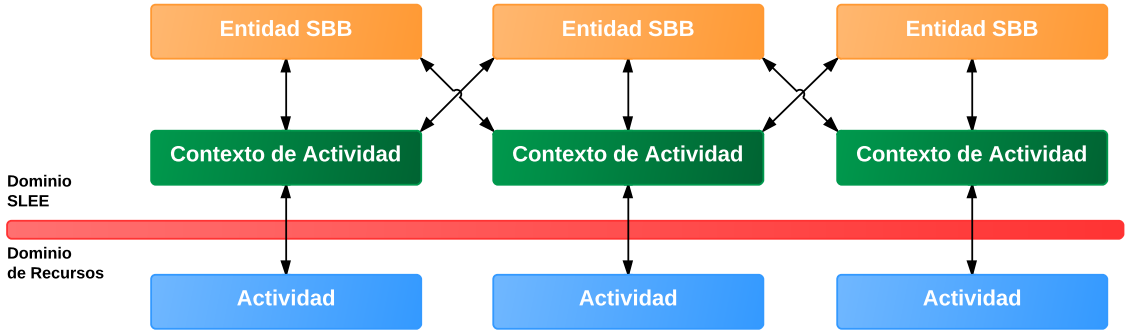

Figura 2.6: Ejemplo de Contexto Actividad

- Una entidad SBB se puede ligar a sí misma a un Contexto de Actividad de un nuevo objeto de Actividad que haya sido creado. Esto permite a la entidad SBB recibir eventos emitidos en ese Contexto de Actividad.
- Una entidad SBB puede desligarse a sí misma de un Contexto de Actividad si no quiere recibir más eventos lanzados en ese Contexto de Actividad.
- Una entidad SBB puede ligar explícitamente otra entidad SBB a un Contexto de Actividad. Esta funcionalidad permite la delegación a otra entidad SBB de todo o parte del procesado de eventos emitidos en el Contexto de Actividad.

Un ejemplo de lo mencionado anteriormente se puede ver reflejado en la figura 2.6.

2.1.6.3. Interfaz del Contexto de Actividad

Una entidad SBB interactúa con un Contexto de Actividad a través del objeto interfaz del Contexto de Actividad (*Activity Context Interface*). Este objeto provee un Contexto de Actividad con una API Java visible que puede ser invocada. A través del objeto interfaz del Contexto de Actividad un SBB puede acceder al objeto Actividad subyacente de un Contexto de Actividad.
Hay dos tipos de objeto interfaz del Contexto de Actividad:

- Objeto genérico interfaz del Contexto de Actividad.
  Un objeto genérico interfaz del Contexto de Actividad implementa la interfaz del Contexto de Actividad genérica. Esta interfaz no se puede utilizar para acceder a atributos almacenados en el Contexto de Actividad.
- Objeto SBB interfaz del Contexto de Actividad.
  Un SBB puede definir una interfaz SBB del Contexto de Actividad que extienda la interfaz del Contexto de Actividad genérica. Esta interfaz define la vista específica de

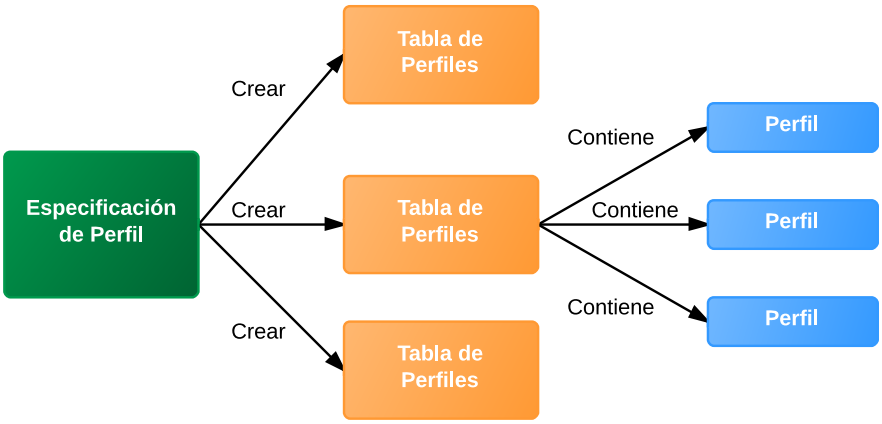

Figura 2.7: Ejemplo de Perfiles de datos

un SBB sobre un Contexto de Actividad. En ella se declaran los atributos compartidos del el SBB que quedan almacenados en el Contexto de Actividad.

2.1.7. Perfiles de datos

Un Perfil (*Profile*) es un componente para almacenar datos. En el esquema de un Perfil se definen los atributos que pueden ser accedidos. Una Tabla de Perfiles contiene cero o más Perfiles que conforman el mismo esquema. En la Especificación de un Perfil se definen las interfaces, clases, y descriptores de despliegue necesarios para definir un esquema de Perfil y las interfaces utilizadas para provisión y acceso a un Perfil definido con ese esquema de Perfil. Cero o más Tablas de Perfiles pueden tener la misma Especificación de Perfil.

2.1.8. Herramientas SLEE

La especificación SLEE define una serie de herramientas que pueden ser usadas por los componentes SBB, la Especificación de Perfiles y los Adaptadores de Recursos.

- Timer Facility.
  Permite a las entidades SBB realizar acciones periódicas o temporizadas.

- Alarm Facility.
  Las alarmas pueden ser utilizadas por SBBs, Adaptadores de Recursos y Perfiles para solicitar al contenedor SLEE iniciar o cancelar una alarma. Esto sirve para que un administrador externo a SLEE pueda recibir notificaciones de alarma cuando se cumplan unas condiciones.

- Trace Facility.
  Fuentes como SBBs, Adaptadores de Recursos, Perfiles y componentes internos de SLEE pueden usar estas trazas para generar mensajes dirigidos a clientes de gestión externos como por ejemplo el entorno de desarrollo.

- Activity Context Naming Facility.
  Provee un espacio de nombres global para identificar Contextos de Actividad. Permite que un objeto SBB asigne un identificador a un Contexto de Actividad y que otro SBB pueda obtenerla mediante su identificador.

- Profile Facility.
  Permite que los SBBs obtengan una Tabla de Perfiles y con ello acceder a los Perfiles de datos.

- Event Lookup Facility.
  Es utilizado por los Adaptadores de Recursos para obtener información acerca de los tipos de evento instalados en el contenedor SLEE obtener los adecuados y convertirlos en objetos para ser emitidos.

2.1.9. Plataformas JAIN SLEE

2.1.9.1. Rhino

Rhino es un servidor de aplicaciones JAIN SLEE producto de OpenCloud. OpenCloud es una empresa fundada en 2000 para crear los estándares software basados en Java de JAIN SLEE para la capa de servicios en telecomunicaciones. Además del servidor, Rhino proporciona todas las herramientas necesarias para facilitar el desarrollo de componentes para que funcionen dentro de su servidor.
La plataforma Rhino cumple con la especificación JAIN SLEE 1.1.

2.1.9.2. Mobicents

Mobicents es una alternativa de código abierto para el desarrollo en JAIN SLEE. De la misma forma que Rhino, Mobicents proporciona un servidor de aplicaciones JAIN SLEE que funciona sobre un servidor JBoss. JBoss es un servidor de aplicaciones J2EE de código abierto implementado en Java. También proporciona las herramientas y documentación necesaria para el desarrollo de componentes. Entre ellas cabe destacar el *plugin* de Eclipse (*ECLIPSLEE*) que agiliza enormemente el desarrollo de componentes y su despliegue dentro del servidor JAIN SLEE.
La plataforma Mobicents también cumple con la especificación JAIN SLEE 1.1.

2.1.9.3. Elección de la plataforma

En un principio el desarrollo se basó en la plataforma Rhino porque al pertenecer al grupo que desarrolló la especificación JAIN SLEE daba ciertas garantías de calidad en el producto y proporciona suficiente documentación y un servidor estable para un desarrollo adecuado. Pero finalmente se optó por el uso de la plataforma Mobicents por su filosofía de código abierto que permite poder inspeccionar el código de todos los componentes desarrollados para aprender a desarrollar otros nuevos. Tener el código fuente y en conjunto con la activa comunidad que hay detrás de Mobicents, facilitó mucho el desarrollo.
La conclusión respecto a qué plataforma es la más adecuada es que las dos cumplen con lo esperado, son estables y el desarrollo no es complicado en ninguna de las dos pero es un punto muy a favor por parte de Mobicents que sea de código abierto.

2.2. Drools

Drools o JBoss Rules [2] es un sistema de gestión de reglas de negocio con un motor de reglas basado en inferencia de encadenamiento hacia delante, conocido como sistema de reglas de producción, usando una implementación avanzada del algoritmo Rete. El proyecto comenzó en 2001 y en las últimas versiones (5.x) introduce el razonamiento con encadenamiento hacia atrás, es por ello que se empieza a usar el término *sistema de razonamiento híbrido* haciendo referencia a los dos tipos de encadenamiento.

2.2.1. Beneficios de un motor de reglas

- Programación Declarativa.
  Los motores de reglas permiten especificar *Qué hacer* y no *Cómo hacerlo*.
  La ventaja de esto es que usando reglas se pueden expresar fácilmente soluciones a problemas complejos. Los sistemas basados en reglas proveen una explicación de cómo se llega a la solución y por qué se toma cada decisión. De esta forma se elabora la solución a un problema.

- Separación de Lógica y Datos.
  Los datos permanecen en los objetos, la lógica en las reglas. Se rompe la unión de datos y lógica de la programación orientada a objetos. Esto no tiene por qué ser una ventaja, solo determinados problemas se verán resueltos de forma óptima usando reglas. Una gran ventaja es que respecto a cambios futuros es mucho más fácil realizar modificaciones cuando se mantiene la lógica separada de los datos. Una vez que el problema está definido, todo el trabajo queda delegado a modificaciones en la lógica.

- Velocidad y Escalabilidad.
  Especialmente el algoritmo de Drools (ReteOO) es muy eficiente en el proceso de comparar hechos y reglas que pueden ser disparadas cuando los cambios que se producen en los datos son pequeños porque el motor puede recordar coincidencias en reglas disparadas anteriormente.

- Centralización de Conocimiento.
  Usando reglas lo que se crea es un repositorio de conocimiento. Es un único punto donde se almacena la lógica y que idealmente puede servir como documentación en relación a cómo se resuelve un problema.

- Integración de Herramientas.
  Drools provee herramientas para funcionar dentro de entornos como Eclipse o interfaces Web. Esto permite gestionar y editar reglas con inmediata retroalimentación y validación.

- Reglas y Lenguaje natural.
  A partir del modelo de datos en los objetos y un lenguaje específico que modele el problema, se pueden escribir reglas muy próximas al lenguaje natural, esto da como resultado una lógica entendible y la posibilidad de que expertos no especializados en esta tecnología puedan colaborar en la elaboración de soluciones sin conocer cómo funciona el código.

2.2.2. Razonamiento y representación de conocimiento

KRR (*Knowledge Representation and Reasoning*) trata sobre cómo representamos el conocimiento en forma de símbolos. El razonamiento trata sobre qué hacemos en relación al acto de pensar usando ese conocimiento. En lenguajes orientados a objetos como C++, Java y C#, existe una forma de definir datos en forma de objetos que describen la composición y comportamiento de las entidades modeladas. Sin embargo estos sistemas de clasificación son limitados por tener que asegurar eficiencia computacional, por ello se han desarrollado formas más sofisticadas de representar nuestro mundo. Una de ellas son las ontologías, que permiten representar el conocimiento de manera exhaustiva, aunque siempre queda un hueco entre qué puede ser teóricamente representado y qué puede ser computado de forma eficiente en la práctica.

2.2.3. Motor de reglas y sistema de reglas de producción

Un motor de reglas es un programa de ordenador que proporciona al desarrollador la funcionalidad de Representación de Conocimiento y Razonamiento (KRR). Hay que destacar tres componentes básicos:

- Ontología
- Reglas
- Datos

Una ontología es el modelo de representación que usamos para las *cosas* que queremos evaluar o analizar. Las reglas son las que desempeñan el razonamiento y los datos son características propias de lo que estamos evaluando.
Drools comenzó como un tipo específico de motor de reglas llamado Sistema de Reglas de Producción (PRS) y estaba basado en el algoritmo Rete. El algoritmo Rete fue desarrollado por Charles Forgy en 1974 y forma el núcleo de un Sistema de Reglas de Producción. El sistema está formado por una estructura de dos partes: el motor compara hechos y datos contra las reglas para inferir conclusiones que dan como resultado acciones.

Código 2.1: Las dos partes de un sistema de reglas de producción

```
when
    <conditions>
then
    <actions>
```

Las acciones se ejecutan en respuesta a los cambios en los datos. Las acciones por sí mismas pueden cambiar datos que al mismo tiempo pueden hacer que se disparen otras reglas.
Drools implementa y extiende el algoritmo Rete. La implementación de Drools del algoritmo Rete es conocida como *ReteOO*, se ha mejorado y optimizado el algoritmo Rete para sistemas orientados a objetos.
Las reglas se almacenan en la memoria de producción (*Production Memory*) y los hechos que el motor de inferencia compara contra estas, se mantienen en la memoria de trabajo (*Working Memory*). La agenda gestiona el orden de ejecución de las reglas y evita conflictos como por ejemplo dos reglas que se pueden disparar al mismo tiempo.

2.2.4. Sistemas de razonamiento híbrido

Drools es un sistema de tipo HSR (*Hybrid Reasoning Systems*), lo que quiere decir que permite realizar encadenamiento hacia delante y hacia atrás. Normalmente los sistemas so-

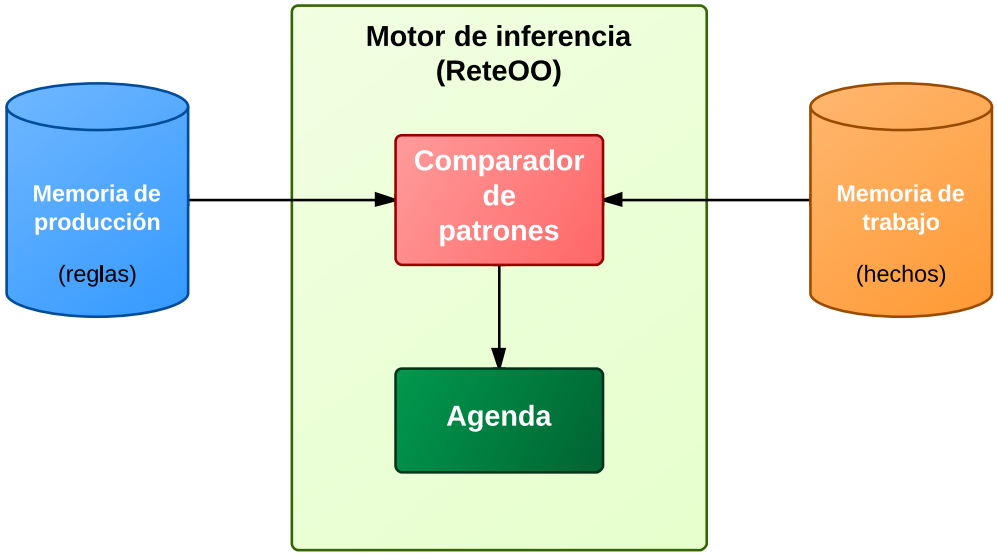

Figura 2.8: Vista de alto nivel de un Sistema de Reglas de Producción

lo permitían utilizar uno de los dos, como OPS5 (encadenamiento hacia delante) o Prolog (encadenamiento hacia atrás).

El encadenamiento hacia delante es de tipo reactivo. Cuando se insertan hechos en la memoria de trabajo, se obtiene como resultado el disparo de una o más reglas ejecutadas por la agenda. Se comienza con un hecho, se propaga a través de las reglas y se termina con una conclusión. El encadenamiento hacia atrás es orientado a objetivos, se comienza con una conclusión que el motor trata de satisfacer. Si no puede, entonces buscará otras conclusiones que pueda satisfacer. Estas se conocen como sub-objetivos que ayudarán a satisfacer algunas partes del objetivo final. Este proceso continua hasta que se consigue el objetivo inicial o no quedan más sub-objetivos.

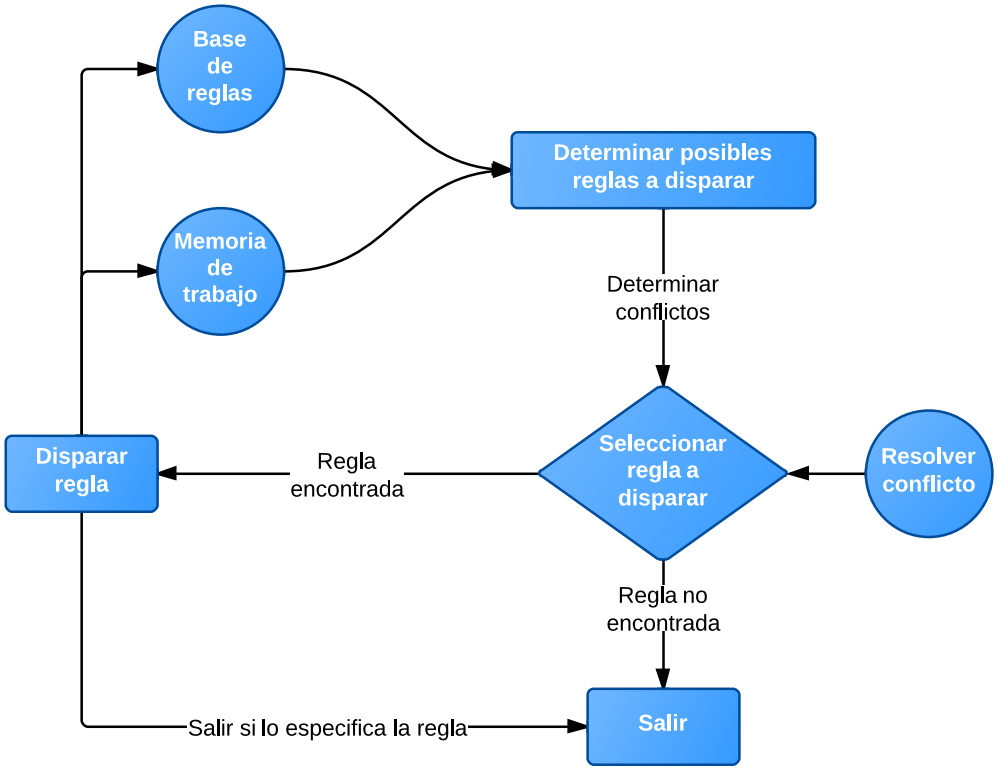

Figura 2.9: Encadenamiento hacia delante

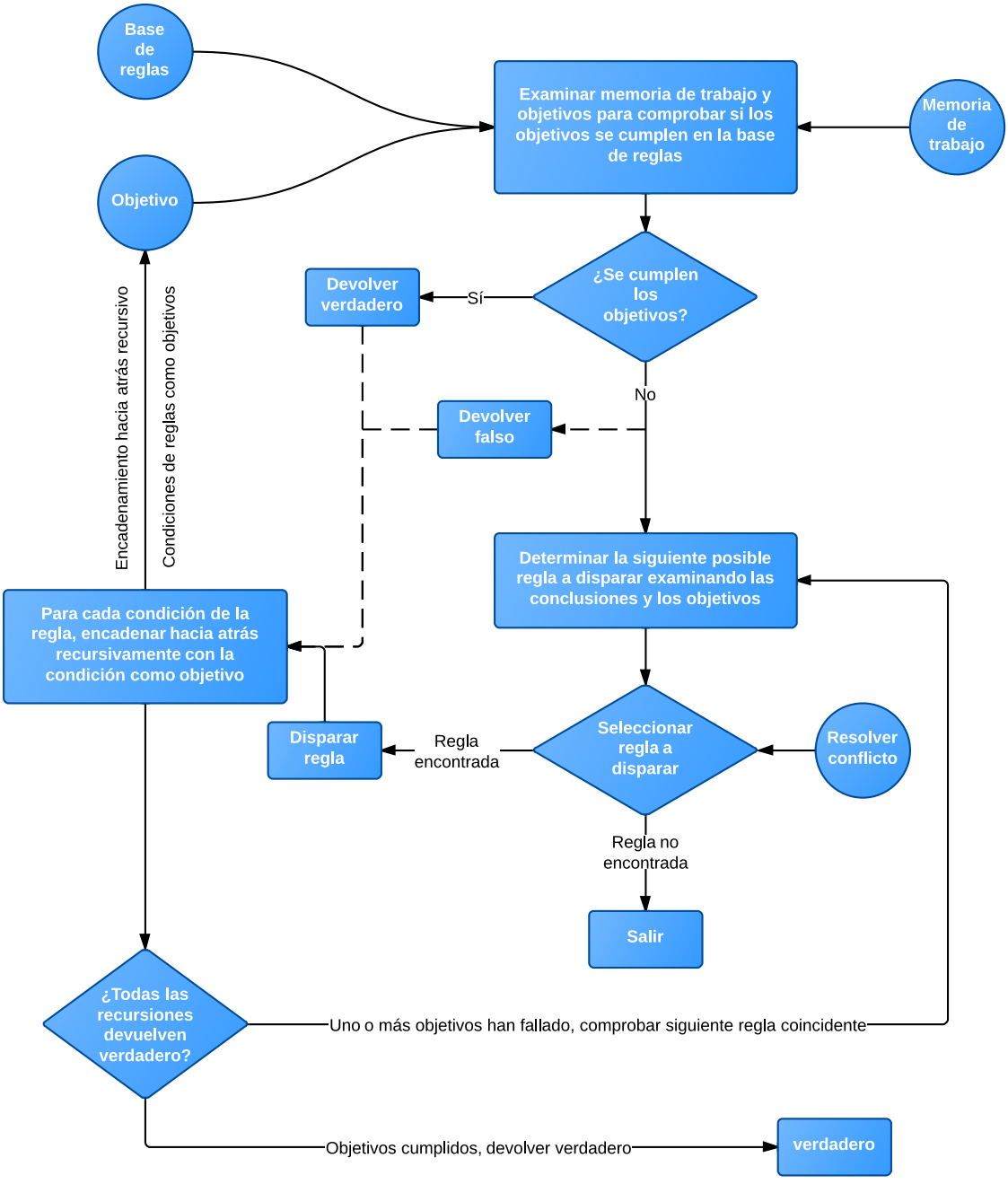

Figura 2.10: Encadenamiento hacia atrás

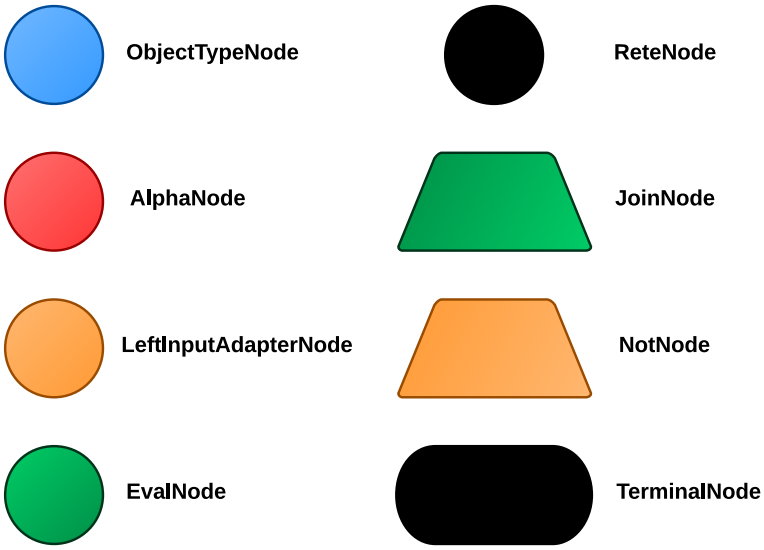

Figura 2.11: Nodos Rete

2.2.5. Algoritmo Rete

El algoritmo Rete [3] fue ideado por el Dr. Charles Forgy en 1979 y se publicó una versión simplificada del artículo [4] en 1982. En latín la palabra *rete* significa *red*. El algoritmo Rete se divide en dos partes: compilación de reglas y tiempo de ejecución.
El algoritmo de compilación describe cómo las reglas son procesadas en la memoria de producción para generar una red eficiente de discriminación. Una red de discriminación se usa para filtrar datos según se propagan a través de la red. Los nodos en la parte más alta de la red pueden tener muchas coincidencias y según se baja hay menos coincidencias. En la parte más baja de la red se encuentran los nodos terminales. El Dr. Forgy describe en el artículo de 1982 4 nodos básicos: raíz, 1-entrada, 2-entradas y terminal.
El nodo raíz es por donde entran todos los objetos en la red. Desde allí pasan inmediatamente al nodo *ObjectTypeNode*. El propósito del *ObjectTypeNode* es asegurar que el motor no hace más trabajo del necesario. Para hacer las cosas de forma eficiente el motor solo debe pasar a los nodos los objetos que coincidan con ese tipo. En Drools se hace esto para aplicar las reglas definidas únicamente al objeto del tipo adecuado. Los nodos *ObjectTypeNode* pueden propagar hacia nodos *AlphaNode*, *LeftInputAdapterNode* y *BetaNode*. Los nodos *AlphaNode* se usan para evaluar condiciones literales. Si hay varias condiciones, estas se ejecutan en cadena. El Dr. Forgy hace referencia a estas como condiciones *IntraElement*. Drools extiende Rete optimizando la propagación de nodos *ObjectTypeNode* hacia nodos *AlphaNode* usando *hashing*. Cada vez que se incluye un nodo *AlphaNode* en un nodo *ObjectTypeNode*, se añade el valor literal como clave del *HashMap* con el nodo *AlphaNode* como

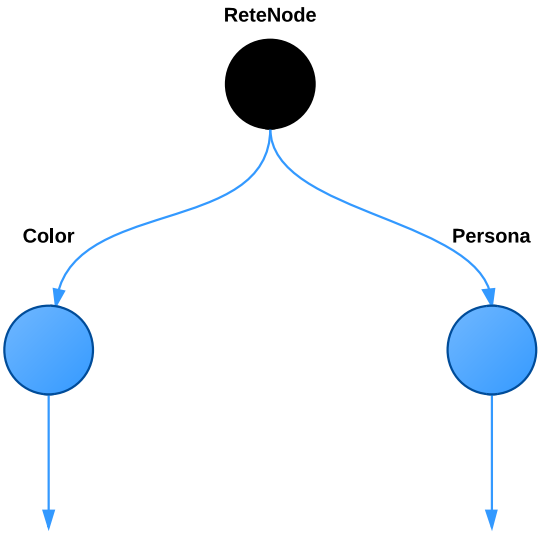

Figura 2.12: Nodos ObjectTypeNode

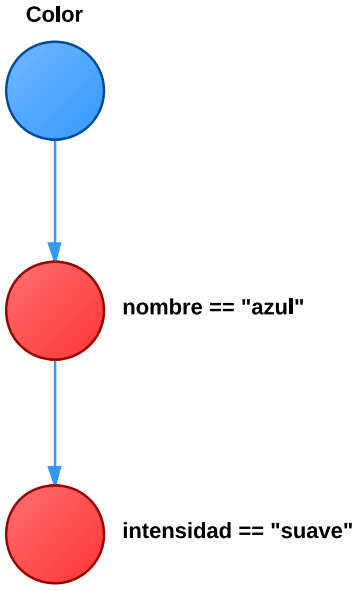

Figura 2.13: Nodos AlphaNode

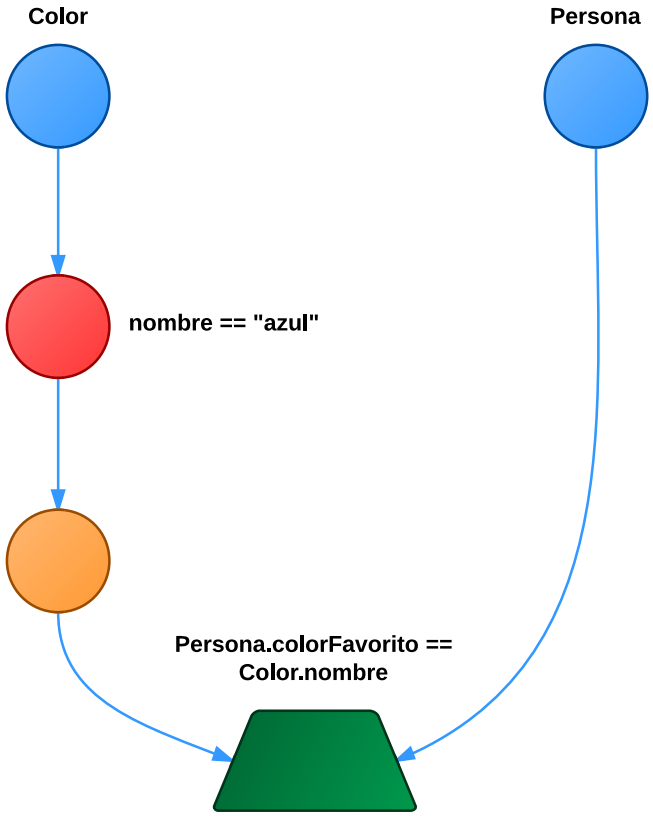

Figura 2.14: Nodos JoinNode

valor. Cuando entra una instancia nueva a un nodo *ObjectTypeNode*, en vez de propagarse a cada nodo *AlphaNode*, puede obtener el nodo *AlphaNode* adecuado del *HashMap* en vez de realizar innecesarias comparaciones literales.

Hay dos clases de nodos del tipo 2-entradas, *JoinNode* y *NotNode*, y los dos son del tipo *BetaNode*. Los nodos *BetaNode* se usan para comparar dos objetos y sus campos. Los objetos pueden ser del mismo y distinto tipo. Por convención se hace referencia a las dos entradas como izquierda y derecha. La entrada izquierda de un nodo *BetaNode* es generalmente una lista de objetos, en Drools esto es una tupla. La entrada derecha es un único objeto. Se pueden usar dos nodos para implementar comprobaciones de existencia de un objeto. Los nodos *BetaNode* también tienen memoria. La entrada izquierda se llama *Beta Memory* y recuerda las tuplas entrantes. La entrada derecha se llama *Alpha Memory* y recuerda todos los objetos entrantes. Drools extiende Rete desarrollando indexado en los nodos *BetaNode*.

Un nodo *LeftInputAdapterNode* coge un objeto como entrada y lo propaga por la red como una única tupla de objeto.

Los nodos terminales se usan para indicar que en una regla se han dado todas las condiciones de disparo. Una regla puede tener múltiples nodos terminales.

Drools también implementa compartición de nodos. Muchas reglas repiten los mismos patrones y la compartición de nodos permite colapsar esos patrones para que no tengan que ser evaluados varias veces para cada instancia. Los códigos mostrados a continuación son un ejemplo de esto:

Código 2.2: Compartición de nodos: Regla 1

```
rule
  when
    Color(color : name == "blue")
    person : Person(favouriteColor == blue)
  then
    System.out.println(person.getName() + " likes blue");
end
```

Código 2.3: Compartición de nodos: Regla 2

```
rule
  when
    Color(color : name == "blue")
    person : Person(favouriteColor != blue)
  then
    System.out.println(person.getName() + " does not like blue");
end
```

Como se puede ver en la Figura 2.15, la red Rete compilada muestra un nodo *alpha* compartido, pero los nodos *beta* no. Cada nodo *beta* tiene su propio nodo *TerminalNode*. En el caso de haber tenido el mismo segundo patrón, también podrían haberlo compartido.

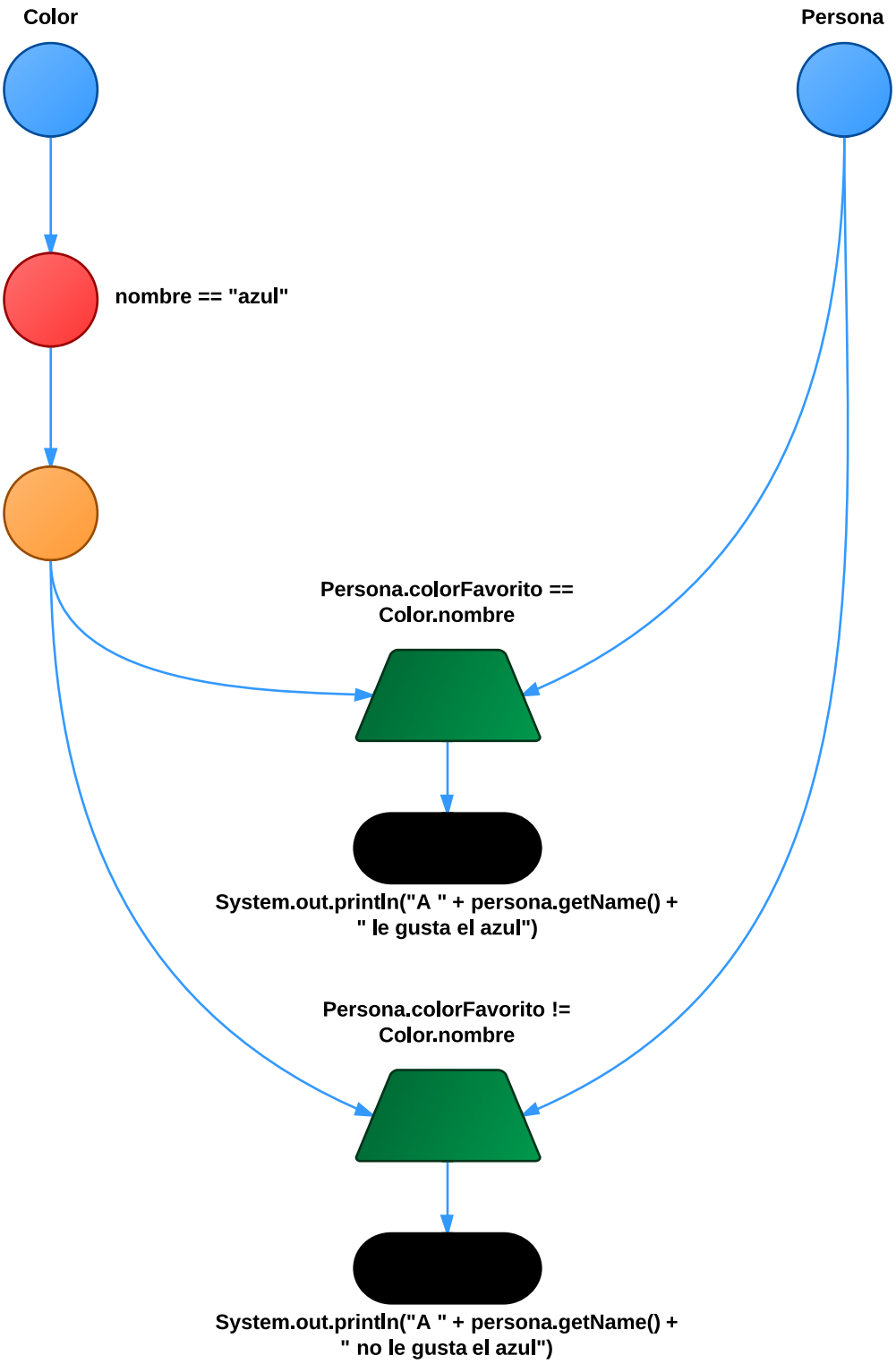

Figura 2.15: Compartición de nodos

2.3. Conceptos de agentes

En esta sección se van a introducir algunos conceptos que ayudan a comprender qué objetivo se persigue al hablar del desarrollo de una arquitectura de agentes.

2.3.1. Agentes

"Un agente es un sistema informático que está situado en un entorno y que es capaz de actuar autónomamente para conseguir sus objetivos". [6]

"Un agente es un sistema informático capaz de actuar de forma flexible y autónoma en un entorno abierto, dinámico e impredecible". [7]
Este sistema está diseñado para adaptarse a estas definiciones ofreciendo entidades capaces de recibir información de su entorno en forma de eventos a través de los adaptadores de recursos, interactuar con su entorno por medio de la interfaz que le permite ejecutar funciones y enviar mensajes a su entorno, y procesar la información recibida haciendo uso de la base de conocimiento del motor de reglas.

2.3.2. Arquitectura de Agentes

Las arquitecturas de agentes describen la interconexión de los módulos software/hardware que permiten a un agente comportarse tal y como se ha descrito en su definición. La arquitectura de agentes SLAA proporciona a los agentes los medios necesarios para que pueda percibir su entorno e interactuar con él.

2.3.3. Arquitectura de Agentes basada en SLEE

Es una arquitectura basada en SLEE. Esto quiere decir que los agentes implementados en ella son agentes orientados a procesar eventos, funcionan y tienen los beneficios de cualquier aplicación desarrollada para la especificación SLEE.

Como hemos podido ver en este capítulo, las herramientas principales que forman la arquitectura son JAIN SLEE y Drools. En el siguiente capítulo se hará un análisis de qué funciones se podrán realizar con el sistema y qué requisitos se han marcado como objetivo.

Capítulo 3

Análisis funcional

Este capítulo presenta los diferentes actores que tienen presencia dentro del sistema y las funciones que pueden realizar. Posteriormente se muestra un análisis de los diferentes requisitos propuestos para la arquitectura y en base a qué casos de uso se pueden llevar a cabo.

3.1. Casos de uso

SLEE Based Agent Architecture (SLAA) es la arquitectura desarrollada mediante las tecnologías descritas anteriormente. En esta sección se va a describir en qué consiste el sistema comenzando por una descripción de los actores, un análisis de casos de uso y una descripción de los requisitos de la arquitectura.

3.1.1. Diccionario de actores

En la tabla 3.1 se presentan todos los actores que participan en los casos de uso descritos en este capítulo.

Identificador del actor	Papel	Descripción
ACT-1	Administrador	Administrador de SLAA
ACT-2	Usuario	Usuario de SLAA
ACT-3	Desarrollador	Desarrollador de SLAA

Tabla 3.1: Diccionario de actores

- Administrador
  El papel de un administrador es el de gestionar la plataforma. Tiene la capacidad de crear o eliminar usuarios y de configurar la plataforma.

- Usuario
  Un usuario es la persona que hace uso del servicio que ofrece la plataforma. Controla su propia existencia dentro de SLAA, la de sus agentes y de la programación de sus bases de conocimiento. Es el encargado de crear sus agentes dentro de la arquitectura SLAA y hacerlos funcionar.

- Desarrollador
  Un desarrollador de SLAA se encarga de mantener, ampliar y mejorar la arquitectura. Ello consiste en conseguir que los agentes implementados en la arquitectura sean capaces de acceder a más recursos externos o de procesar información y realizar funciones que mejoren su comportamiento.

3.1.2. Descripción de los casos de uso

En esta sección se van a describir los diferentes casos de uso disponibles en el sistema. En la figura 3.1 se muestran los actores y las correspondientes funcionalidades que pueden utilizar en el sistema.

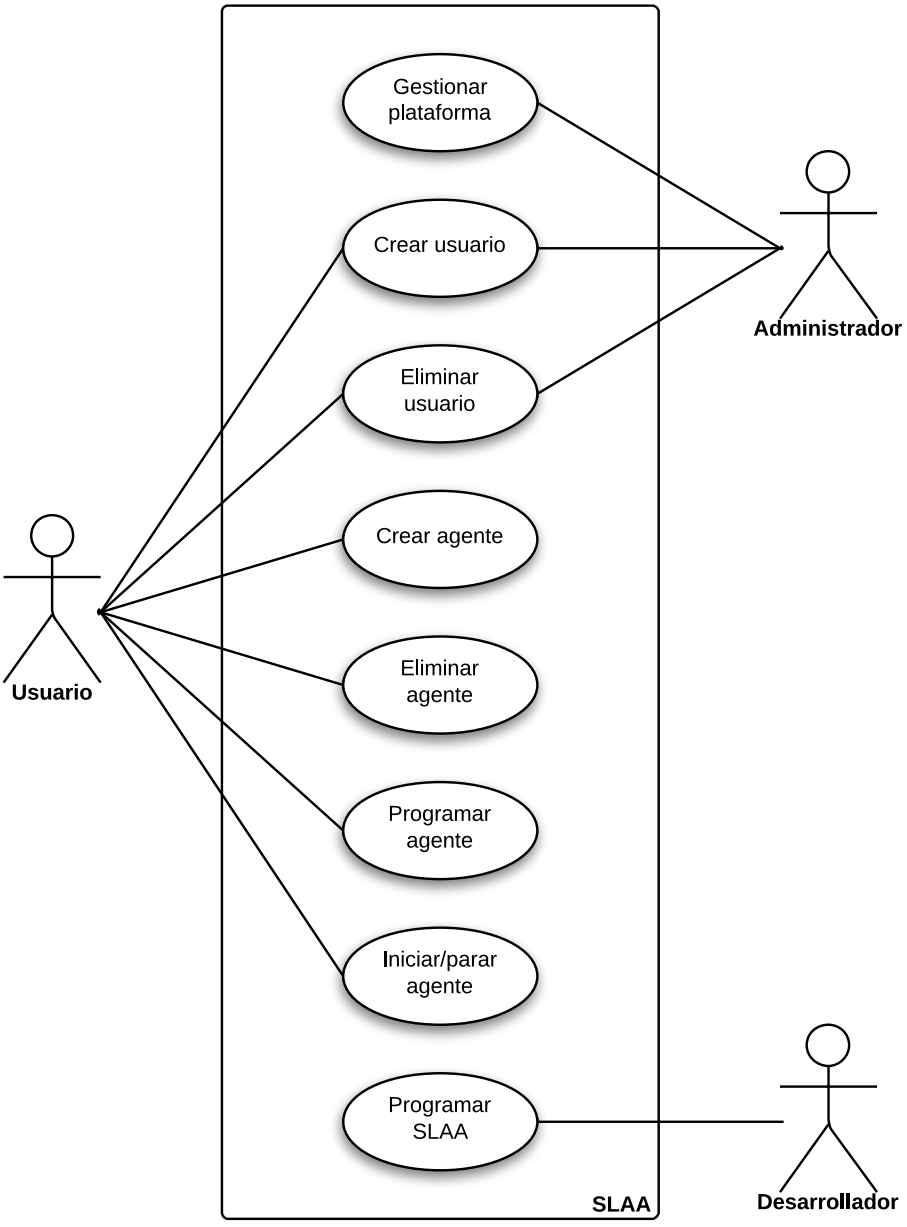

Figura 3.1: Diagrama de casos de uso

- CU-1: Gestionar la plataforma (tabla 3.2)
  Un administrador desde el punto de vista de la gestión de la plataforma, puede iniciarla, pararla o modificar su configuración.

Identificador	CU-1
Nombre	Gestionar la plataforma.
Descripción	Un administrador puede gestionar la plataforma.
Actores	Administrador.
Precondiciones	Arquitectura SLAA en funcionamiento.
Flujo iniciar	1. El administrador solicita iniciar la plataforma. 2. Se inicia la plataforma. 3. Se reinician todos los agentes que estaban activos.
Flujo parar	1. El administrador solicita parar la plataforma. 2. Se paran temporalmente todos los agentes activos. 3. Se procede a parar la plataforma.
Flujo configurar	1. El administrador modifica la configuración de la plataforma. 2. El gestor comprueba que se puede realizar la acción. 3. Se paran temporalmente todos los agentes activos. 4. Se aplica la nueva configuración de la plataforma. 5. Se reinician todos los agentes que estaban activos.

Tabla 3.2: Caso de uso gestionar la plataforma

- CU-2: Crear un usuario (tabla 3.3)
  La plataforma permite añadir diferentes usuarios. Estos usuarios son los que podrán crear y programar sus agentes.

- CU-3: Eliminar un usuario (tabla 3.4)
  Si es requerido, un usuario se puede eliminar completamente de la plataforma. Cuando

Identificador	CU-2
Nombre	Crear un usuario.
Descripción	La plataforma permite que se puedan crear usuarios.
Actores	Usuario y Administrador.
Precondiciones	Arquitectura SLAA en funcionamiento.
Flujo normal	1. El usuario/administrador solicita crear un usuario nuevo. 2. Se procesa la petición y se registran los datos.

Tabla 3.3: Caso de uso crear un usuario

un usuario es eliminado, se eliminan todos los agentes correspondientes y sus datos asociados.

Identificador	CU-3
Nombre	Eliminar un usuario.
Descripción	Un administrador puede solicitar eliminar un usuario. Un usuario puede solicitar eliminar su propio usuario.
Actores	Usuario y Administrador.
Precondiciones	Arquitectura SLAA en funcionamiento y un usuario existente.
Flujo normal	1. El usuario/administrador solicita eliminar un usuario. 2. Se eliminan todos los agentes del usuario y sus datos. 3. Se elimina toda la información del usuario.

Tabla 3.4: Caso de uso eliminar un usuario

- CU-4: Crear un agente (tabla 3.5)
  Un usuario tiene que crear un agente para poder configurarlo y programarlo. Una vez que el agente ha sido creado, se puede acceder a su base de conocimiento para empezar a programar las reglas y funciones que determinan su comportamiento.

- CU-5: Eliminar un agente (tabla 3.6)
  Si un usuario lo desea, puede eliminar por completo de la plataforma a uno de sus

Identificador	CU-4
Nombre	Crear un agente.
Descripción	La plataforma permite que se puedan crear agentes.
Actores	Usuario.
Precondiciones	Arquitectura SLAA en funcionamiento y un usuario existente.
Flujo normal	1. El usuario solicita crear un agente. 2. Se registran los datos del agente. 3. Se asigna una base de conocimiento al agente.

Tabla 3.5: Caso de uso crear un agente

agentes. Ello implica eliminar toda su información y datos persistentes que haya podido almacenar durante su funcionamiento.

Identificador	CU-5
Nombre	Eliminar un agente.
Descripción	Un usuario puede solicitar eliminar uno de sus agentes.
Actores	Usuario.
Precondiciones	Arquitectura SLAA en funcionamiento, usuario y agente existente.
Flujo normal	1. El usuario solicita eliminar un agente. 2. Se para el agente si estaba activo. 3. Se elimina toda la información del agente.

Tabla 3.6: Caso de uso eliminar un agente

- CU-6: Programar un agente (tabla 3.7)
  Para que un agente interactúe con su entorno debe ser programado. La herramienta de la que dispone un usuario para programar su agente es el motor de reglas Drools. Un usuario define el comportamiento que puede tener el agente cuando recibe un evento en función de los datos del evento u otros datos que el agente haya podido almacenar

durante su funcionamiento. Este comportamiento puede ser disparar otros eventos, realizar funciones o almacenar información.

Identificador	CU-6
Nombre	Programar un agente.
Descripción	Un usuario debe programar su agente para que actúe.
Actores	Usuario.
Precondiciones	Arquitectura SLAA en funcionamiento, usuario y agente existente.
Flujo normal	1. El usuario crea un agente. 2. El usuario configura el agente. 3. El usuario programa el agente.

Tabla 3.7: Caso de uso programar un agente

- CU-7: Iniciar/parar un agente (tabla 3.8)
  Un agente debe ser iniciado para que comience a recibir información de su entorno y pueda procesarla para decidir acciones. También puede ser parado para que deje de procesar y recibir información.

Identificador	CU-7
Nombre	Iniciar/parar un agente.
Descripción	Un usuario puede iniciar/parar un agente.
Actores	Usuario.
Precondiciones	Arquitectura SLAA en funcionamiento, usuario existente y agente existente.
Flujo normal	1. El usuario solicita iniciar/parar su agente. 2. Se procede a iniciar/parar el agente.

Tabla 3.8: Caso de uso iniciar/parar un agente

- CU-8: Progrmar SLAA (tabla 3.9)

Un desarrollador puede modificar el código fuente de la arquitectura SLAA para mejorar sus capacidades. Esto consiste por ejemplo en que el desarrollador posibilite que un agente pueda utilizar todas las funcionalidades de un nuevo adaptador de recursos o añadir nuevas funciones dentro de la plataforma que mejoren la forma en la que el agente actúa dentro de SLAA siguiendo la forma con la que se ha programado la arquitectura.

Identificador	CU-8
Nombre	Programar SLAA.
Descripción	Un desarrollador puede modificar el código fuente.
Actores	Desarrollador.
Precondiciones	Acceso al código fuente de SLAA.
Flujo normal	1. El desarrollador diseña una mejora para SLAA. 2. El desarrollador implementa la mejora para SLAA. 3. El desarrollador documenta la mejora para SLAA.

Tabla 3.9: Caso de uso programar SLAA

3.2. Análisis y captura de requisitos

En los casos de uso se han visto los diferentes elementos que pueden actuar en la arquitectura y las funciones que pueden realizar. En esta sección se va a hacer un análisis de los requisitos que debe cumplir esta arquitectura para que sea completamente funcional cumpliendo todos los objetivos necesarios. En la tabla 3.10 se presenta el resultado del análisis de requisitos:

Finalmente se ha mostrado qué es lo que se espera de la arquitectura desde un punto de vista funcional. En el siguiente capítulo se va a describir de qué forma se usan las tecnologías del capítulo 1 para obtener los requisitos analizados en este capítulo.

Id	Descripción	Tipo	Prioridad	Traza
1	Servicio en internet para el desarrollo de agentes orientados a las comunicaciones y el procesado de eventos.	Funcional	Obligatorio	Todos
2	Plataforma que admita múltiples usuarios y agentes por usuario.	Funcional	Obligatorio	CU-2 CU-3 CU-4 CU-5
3	Integración de un motor de reglas para la programación de agentes.	Funcional	Obligatorio	CU-6
4	Programación de agentes reactivos y proactivos con capacidad de comunicación entre instancias de agente.	Funcional	Obligatorio	CU-6
5	Interfaz web para acceder a la plataforma por parte de usuarios y administradores.	No Funcional	Deseable	Todos
6	Plataforma escalable para futuras mejoras y ampliaciones sobre el desarrollo.	No Funcional	Obligatorio	CU-8

Tabla 3.10: Requisitos de la arquitectura SLAA

Capítulo 4

Diseño arquitectónico y detallado

Este capítulo describe la arquitectura completa del sistema. Comenzando por una descripción inicial sobre un esquema de la arquitectura se presentarán todos los elementos clave que intervienen y después se introducirá en detalle cada uno de ellos para terminar de comprender cómo funciona.

4.1. Descripción general del sistema

En la figura 4.1 se presenta la captura de una de las posibles situaciones en las que se puede encontrar la arquitectura en un momento determinado. Esta figura sirve para introducir todos los elementos que intervienen en el funcionamiento del sistema. Posteriormente se explicará con detalle en este capítulo cada uno de ellos. La situación que se muestra corresponde a tres instancias de agente utilizando diferentes servicios y herramientas para llevar a cabo su cometido. A continuación se presentan los elementos más relevantes mostrados en la figura.

- Recursos externos.
  Son los elementos externos a la arquitectura con los que se comunican principalmente los agentes. Estos pueden ser cualquier sistema conectado a Internet con el que intercambian mensajes a través de los adaptadores de recursos.

- Adaptadores de recursos.
  Los adaptadores de recursos son la parte de la arquitectura que une los servicios con los recursos externos. Se encargan de intermediar entre estos procesando los mensajes en ambas direcciones para encaminarlos adecuadamente hasta sus destinatarios.

- Actividades.
  Las actividades, representadas como una línea discontinua, son los canales que sirven para enviar eventos desde un elemento a otro. Habitualmente un evento lo genera un adaptador de recursos ante la llegada de un mensaje de un recurso externo, pero también es posible generar un evento desde una entidad SBB y enviarla a otra a través de una actividad, como se muestra en la figura, el gestor de agentes es capaz de comunicarse con los agentes de esta manera. Solo existen actividades cuando una entidad SBB espera recibir eventos de otro elemento, pudiendo no establecerse nunca una actividad cuando no es necesario recibir eventos.

- Interfaces.
  En el esquema se han representado las interfaces como una línea continua entre los diferentes elementos dentro des la arquitectura. En el caso de las interfaces, estas sirven para ejecutar acciones desde una entidad SBB a través de otro elemento, por ejemplo, un SBB enviando un mensaje hacia un recurso externo hace uso de la interfaz del adaptador de recursos para generar el mensaje y enviarlo, o una entidad SBB que quiere acceder a una tabla de datos o un temporizador necesita acceder a la interfaz de las herramientas SLEE para obtenerlo.

- Herramientas SLEE.

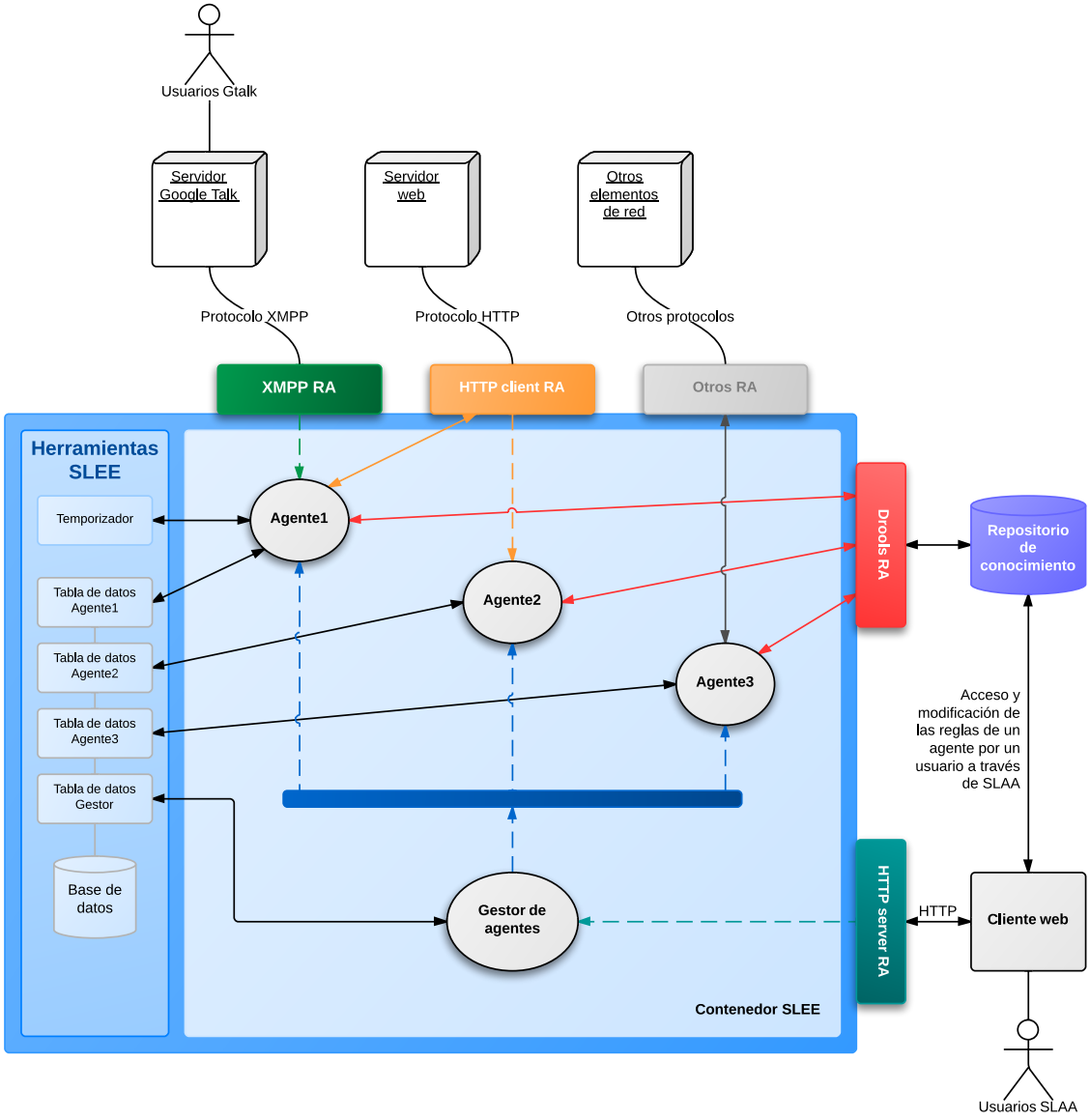

Figura 4.1: Arquitectura SLAA

Las herramientas SLEE son servicios propios de SLEE que ofrecen funcionalidades útiles y que se han aprovechado para la elaboración de la arquitectura. Las herramientas utilizadas son los temporizadores para generar eventos dentro del propio servidor hacia las entidades SBB agente, y las tablas de datos para almacenar información de forma persistente.

- Cliente web.
  Uno de los recursos externos a destacar es la interfaz web. Esta permite todo el control de la plataforma por parte de un administrador o un usuario creando y programando sus agentes. Desde el cliente web se pueden enviar diversas peticiones a la plataforma para realizar acciones y recibir respuestas con diferente información en función de la petición.

- Repositorio de conocimiento.
  Este recurso externo es la parte de la arquitectura en la que se almacenan los ficheros de reglas de los agentes y que se pueden modificar desde el cliente web. Desde el adaptador de recursos Drools, los agentes pueden usar su interfaz para obtener el medio que les permite ejecutar las reglas que determinan su comportamiento.

4.2. Adaptadores de recursos

En esta sección se van a presentar los adaptadores de recursos que se utilizan en el sistema y permiten la comunicación con los recursos externos. Para desarrollar la arquitectura se han utilizado adaptadores de recursos que ya se disponen implementados para el servidor JAIN SLEE de Mobicents y se han creado otros para cubrir otras funcionalidades necesarias. En el modelo de comunicación siempre habrá un elemento externo con el que intercambiará datos un SBB a través de un adaptador de recursos.

4.2.1. Adaptador de recursos HTTP servidor

Este adaptador de recursos es el único que es utilizado al mismo tiempo por el gestor de agentes y los agentes debido a que es el que permite recibir peticiones HTTP desde un recurso externo y ambos necesitan esa funcionalidad. Con él se consigue que la plataforma pueda recibir peticiones y generar respuestas usando el protocolo HTTP. Para que esto sea posible, únicamente es necesario desplegar el adaptador de recursos HTTP server ya implementado para el servidor JAIN SLEE de Mobicents. Una vez desplegado se puede comenzar a recibir peticiones HTTP desde cualquier recurso externo hacia el servidor JAIN SLEE. Cuando el servidor recibe una petición HTTP, este genera un evento de tipo inicial que será recibido por cualquier entidad SBB que haga uso de él, en este caso puede ser el gestor de agentes o

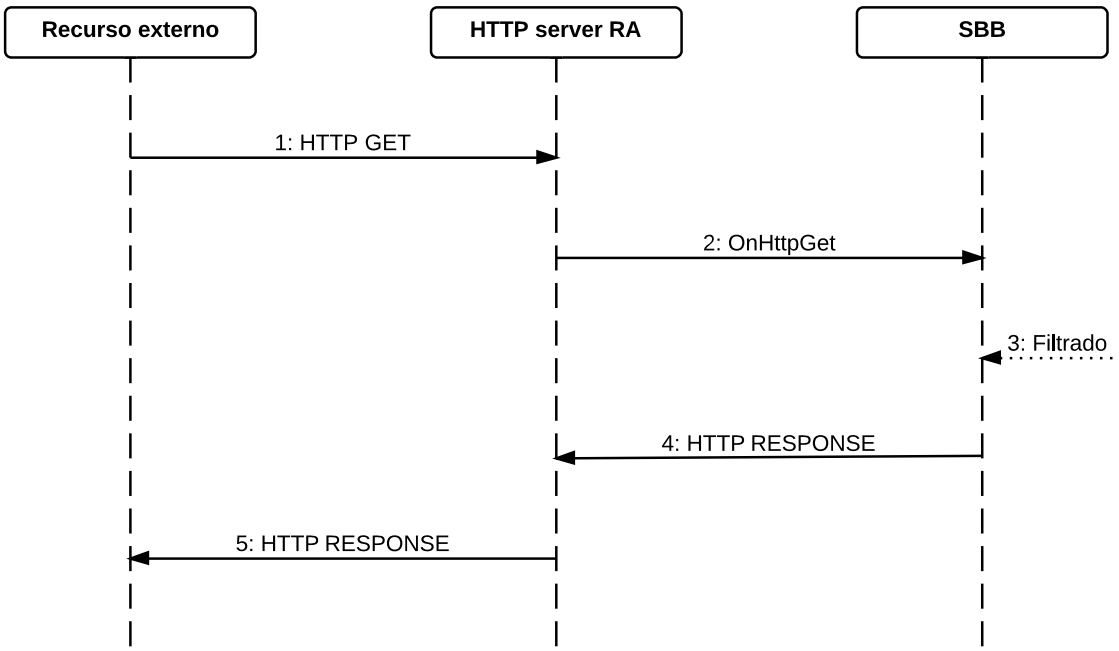

Figura 4.2: Diagrama de secuencias HTTP server RA

algún agente en concreto. Para filtrar las peticiones y que solo el componente deseado reciba el evento, se han definido URLs propias para cada componente, así por ejemplo para que un agente reciba una petición HTTP esta debe ser dirigida a la plataforma indicando el usuario y el nombre del agente para que de forma unívoca un agente tenga una URL de acceso como puede verse a continuación:

localhost:8080/mobicents/user-name/agent-name

Diagrama de secuencia (figura 4.2)

En la figura 4.2 se muestra el diagrama de secuencia habitual de una interacción entre un recurso externo (por ejemplo un usuario desde un navegador web) y uno de los servicios configurados para procesar eventos HTTP. En este caso el servicio puede ser un agente o el gestor de agentes.

- 1: HTTP GET
  Un usuario envía una petición HTTP GET hacia la plataforma. Esta petición será recibida por el adaptador de recursos y la procesará.

- 2: OnHttpGet
  Una vez que el adaptador de recursos recibe la petición, este genera un evento y lo dispara dentro de una actividad. Las peticiones HTTP se utilizan sin sesión, esto quiere

decir que por cada evento se genera una actividad nueva y el evento es de tipo inicial. La consecuencia de esto es que cada entidad SBB configurada para funcionar con este adaptador de recursos va a recibir el evento, así es como funciona JAIN SLEE, por ello es necesario realizar un filtrado para determinar qué entidad es la que tiene que procesar el evento.

- 3: Filtrado
  Cuando la entidad SBB recibe el evento, se analiza la URL para determinar quién es el destinatario. Cada agente se identifica unívocamente por su nombre y el usuario al que pertenece, y extrayendo estos datos de la URL, se puede determinar si el destinatario es un agente en concreto o el gestor de agentes.

- 4: HTTP RESPONSE
  Una vez el SBB ha procesado el evento, puede generar una respuesta o realizar una acción. Normalmente se incluirá algún tipo de información en relación al evento que se ha recibido para que el emisor reciba una respuesta. En el caso de un agente, se disparará una regla en su base de conocimiento, en ella se ejecutarán las acciones que sean necesarias y se podrá incluir una respuesta para el emisor.

4.2.2. Adaptador de recursos HTTP cliente

Este adaptador de recursos permite que los agentes puedan enviar peticiones HTTP hacia cualquier recurso externo que use este protocolo. Esto también ha permitido que los agentes puedan comunicarse entre ellos, porque haciendo uso de el adaptador de recursos cliente y servidor se pueden enviar peticiones y generar respuestas entre agentes.

Diagrama de secuencia (figura 4.3)

- 1: HTTP GET
  Un agente decide enviar una petición HTTP GET porque está definido así en las reglas de su base de conocimiento. Para ello utiliza su interfaz, que le permite enviar una petición HTTP GET a una URL cualquiera. Dentro de SLEE lo que sucede es que la entidad SBB recurre a la interfaces del adaptador de recursos HTTP cliente para enviar una petición HTTP GET que quedará asociada a una actividad creada en la que se recibirá la repuesta en forma de evento.

- 2: HTTP GET
  El adaptador de recursos recibe la orden de la entidad SBB para enviar una petición HTTP GET a la URL indicada. Este la envía al recurso externo, que puede ser un servidor web o cualquier elemento que funcione con el protocolo HTTP.

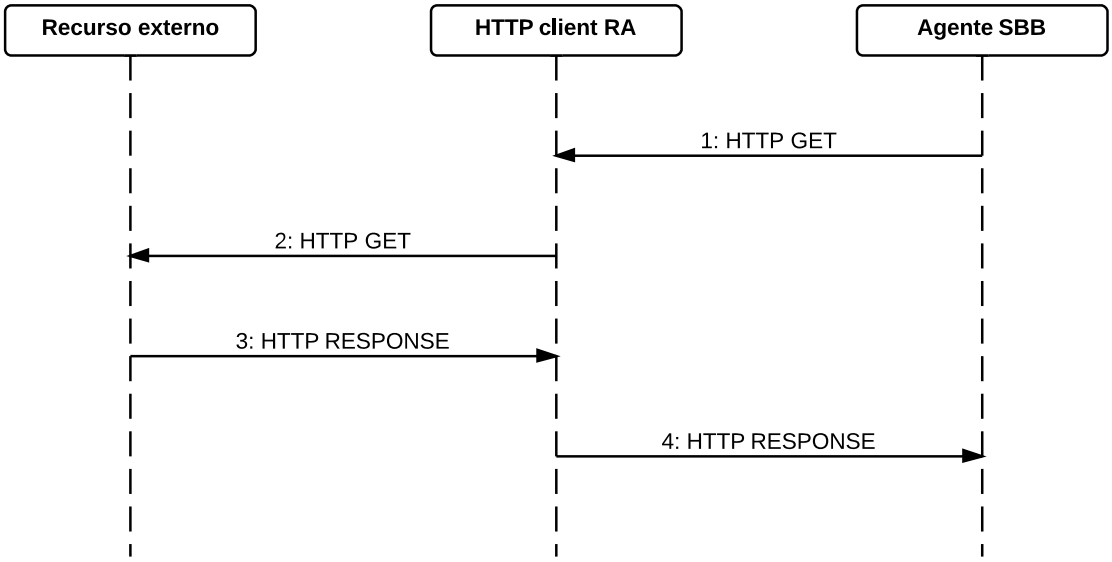

Figura 4.3: Diagrama de secuencias HTTP client RA

- 3: HTTP RESPONSE
  El recurso externo recibe la petción HTTP GET, la procesa y genera una respuesta que recibirá el adaptador de recursos HTTP cliente.

- 4: HTTP RESPONSE
  Cuando el adaptador de recursos recibe la respuesta HTTP del recurso externo, este sabe a qué actividad estaba asociada. La entidad SBB, que en este caso es el agente que hizo la petición, está ligada a esa actividad, por ello recibirá la repuesta directamente, y si alguna regla lo determina, se realizará alguna acción.

4.2.3. Adaptador de recursos XMPP

Otro de los adaptadores de recursos ya incluidos en el servidor JAIN SLEE de Mobicents permite establecer una conexión con un servidor XMPP de mensajería instantánea. Con ello se consigue que un agente se conecte a un servicio de chat como un usuario cualquiera e intercambiar mensajes de texto como se hace habitualmente con los contactos de Gtalk.

Diagrama de secuencia (figura 4.4)

- 1: CONNECT
  Cuando un agente está configurado para funcionar con XMPP, al iniciarse se conectará con un servidor XMPP, por ejemplo Gtalk. Para ello utiliza la interfaz del adaptador de recursos XMPP que le permite crear una conexión con el recurso externo que en

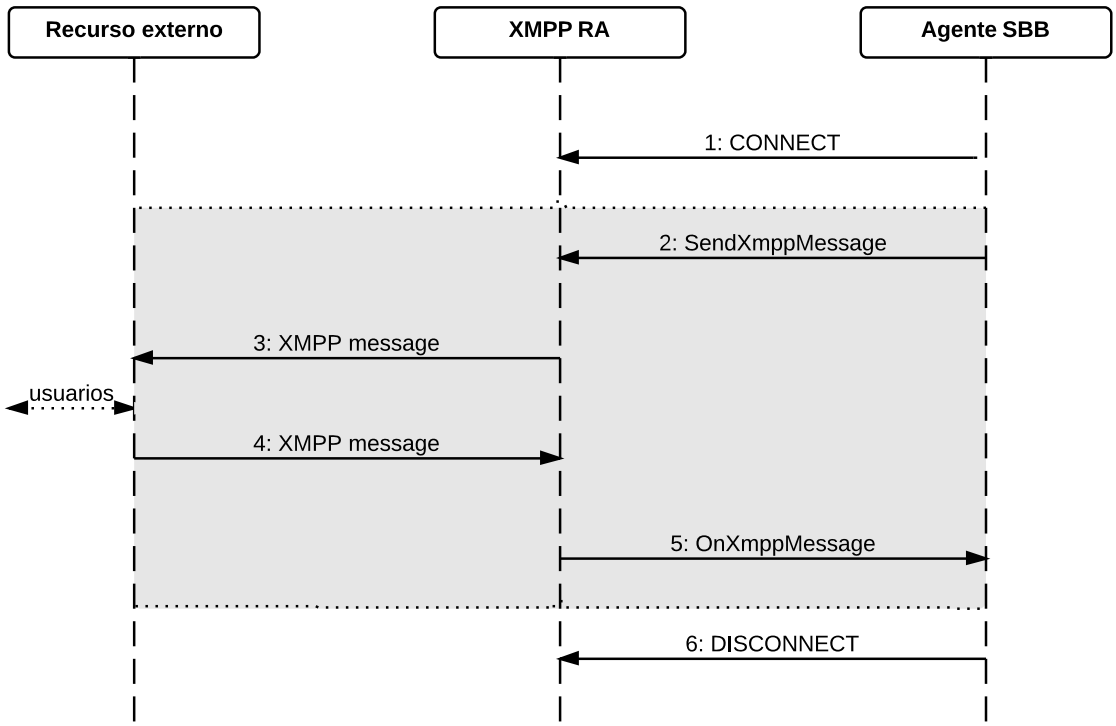

Figura 4.4: Diagrama de secuencias XMPP RA

este caso será el servidor XMPP y una actividad SLEE entre el adaptador de recursos y la entidad SBB asocidada a esa conexión.

- 2: sendXmppMessage
  Una vez el agente está conectado con el servidor XMPP, puede empezar a enviar o recibir mensajes de forma asíncrona. Este intercambio de mensajes se representa en el diagrama como una zona sombreada. Si en sus reglas está determinado, un agente puede usar su interfaz para enviar un mensaje XMPP con un contenido de texto a cualquier usuario de su lista de contactos.

- 3: XMPP message
  Dicho mensaje generado por el agente será enviado por el adaptador de recursos hacia el servidor XMPP. Finalmente el servidor XMPP es el encargado de hacerlo llegar directamente al usuario destinatario, que lo recibirá como si de cualquier conversación de *chat* se tratase.

- 4: XMPP message
  Si un usuario cualquiera de XMPP quiere comunicarse con un agente, puede enviarle un mensaje de *chat* por ejemplo desde la ventana de su cliente de mensajería instantánea.

El mensaje será recibido por su servidor XMPP, y este lo reenviará al adaptador de recursos XMPP.

- 5: onXmppMessage
  El adaptador de recursos XMPP recibe en una de sus conexiones XMPP un mensaje de *chat* de un usuario. Cada conexión XMPP está asociada a una actividad SLEE en la que se enviarán todos los eventos recibidos en esa conexión para que los procesen las entidades SBB ligadas a esa actividad. En este caso la entidad es la que corresponde al agente que inició la conexión, y por ello, será la que recibirá el evento con el contenido del emisor y el mensaje, y si está determinado, se disparará alguna regla.

- 6: DISCONNECT
  Cuando el agente se para, también se cierra la conexión con el servidor XMPP, de esta forma todos los usuarios verán el agente como desconectado y sabrán que si le envían mensajes no le van a llegar.

4.2.4. Adaptador de recursos SocialNetwork

Este adaptador de recursos se ha desarrollado específicamente para la arquitectura con el objetivo de demostrar cómo se puede ampliar su funcionamiento implementando un nuevo adaptador de recursos. La misión de este adaptador de recursos es que el agente pueda hacer uso de redes sociales, en concreto se ha desarrollado un servicio que permite que un agente pueda enviar mensajes a Tweeter.

Diagrama de secuencia (figura 4.5)

- 1: CONNECT
  Cuando un agente está configurado para usar el servicio Twitter, al iniciarse se creará una sesión a través de la interfaz SBB del adaptador de recursos SocialNetworkRA, que le permitirá enviar *tweets*.

- 2: sendTweet
  Un agente puede enviar un mensaje de Twitter a través de su interfaz simplemente llamando a la función que le permite hacerlo.

- 3: Tweet
  El adaptador de recursos mantiene la relación entre agentes y sus sesiones de Twitter, de esta forma enviará el mensaje que el agente a generado con la cuenta de Twitter correspondiente.

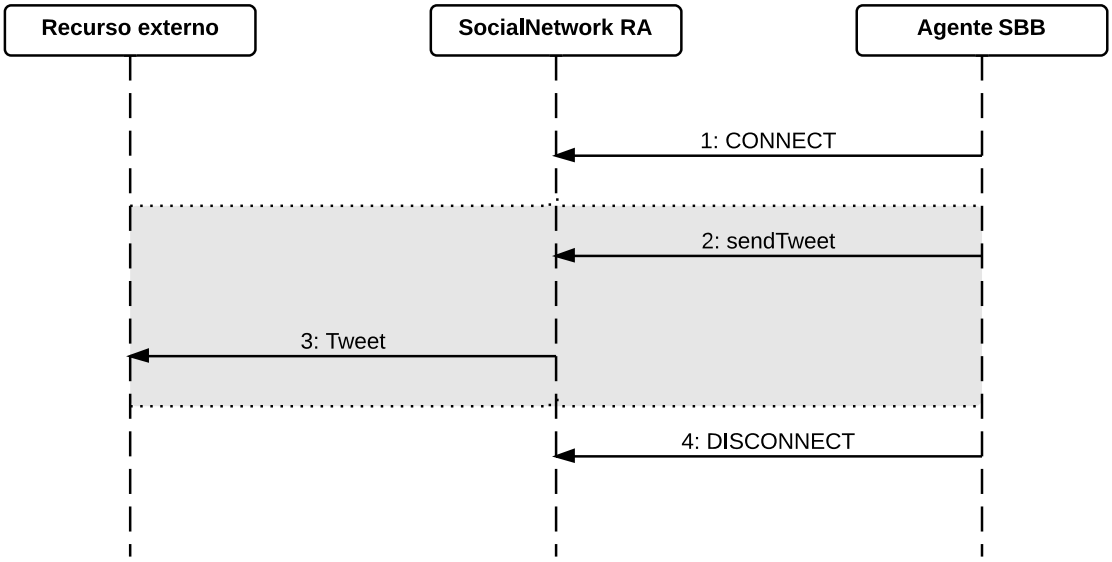

Figura 4.5: Diagrama de secuencias SocialNetwork RA

- 4: DISCONNECT
  Una vez que el agente no va a hacer más uso de este servicio (por ejemplo al pararse), puede desconectarse para eliminar la sesión asociada a su cuenta en el adaptador de recursos.

4.2.5. Adaptador de recursos Drools

Este adaptador de recursos es el que permite que un agente pueda hacer uso del motor de reglas Drools para procesar eventos y generar respuestas. Para ello, la interfaz SBB del adaptador de recursos proporciona una vía a través de la cual el agente puede obtener su memoria de trabajo en la que se irán añadiendo los hechos por cada evento recibido y las reglas del agente definidas en su fichero de reglas.

Diagrama de secuencia (figura 4.6)

- 1: GET WorkingMemory
  Un agente necesita su memoria trabajo para incluir los hechos que dispararán las reglas. El adaptador de recursos Drools es el encargado de proporcionarle una memoria de trabajo con las reglas determinadas en el fichero de reglas propio del agente.

- 2: GET Agent DRL file
  El adaptador de recursos recurre al repositorio de conocimiento para obtener el fichero de reglas. El repositorio de conocimiento es el lugar en el que están almacenados todos

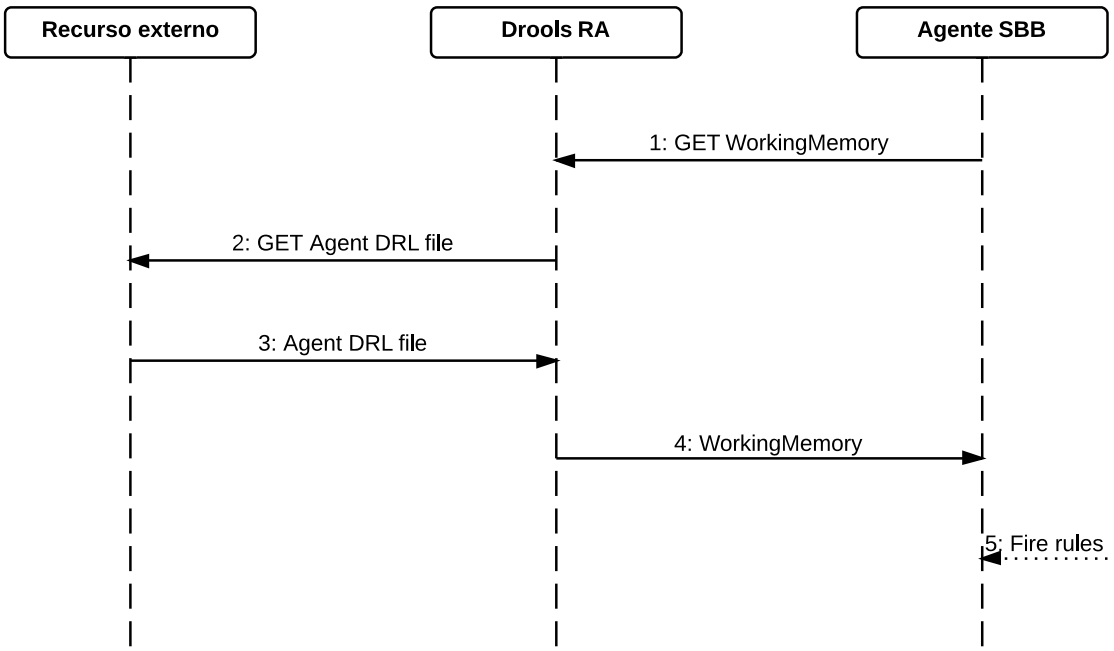

Figura 4.6: Diagrama de secuencias Drools RA

los ficheros de reglas de todos los agentes del servidor.

- 3: Agent DRL file
  Una vez obtenido el fichero de reglas del repositorio de conocimiento, el adaptador de recursos puede generar y entregar la memoria de trabajo que necesita el agente.

- 4: WorkingMemory
  El agente usa su memoria de trabajo para incluir los hechos que dispararán las reglas. Estos hechos se crean por cada evento que llega al SBB agente. Cada suceso en la entidad SBB crea un objeto que será el hecho incluido en la memoria de trabajo del agente.

- 5: Fire rules
  Cuando se añade el hecho en la memoria de trabajo, se disparan todas las reglas que coinciden con ese hecho. De esta forma, por cada evento el agente realiza una acción, y así es como funcionan los agentes orientados a eventos en la arquitectura SLAA.

4.2.6. Otros adaptadores de recursos

Como se ha mostrado en esta sección, se pueden aprovechar los adaptadores de recursos ya existentes o crear otros nuevos para que los usen los agentes y puedan comunicarse con

cualquier recurso externo. En algunos casos bastará con configurar correctamente el SBB del agente para que funcione con uno de los adaptadores de recursos ya disponibles en el servidor JAIN SLEE de Mobicents. En otros casos, si se requiere alguna funcionalidad o comunicación con un servicio que no está disponible en el servidor por defecto, será necesario programar el adaptador de recursos y de la misma forma que antes, configurar el SBB para que funcione con él y el agente lo utilice.

4.3. Actividades

Las actividades son las vías a través de las cuales llegan los eventos a los SBB. En esta arquitectura hay dos tipos de SBB: El gestor de agentes, y todos los agentes ejecutándose en la plataforma, y ambos recibirán eventos.
También se distinguen dos tipos de actividades importantes: Una de comunicación interna entre los SBB para control por parte del gestor de agentes hacia los agentes, y otras entre los recursos externos cuando envían mensajes hacia el gestor de agentes o lo propios agentes. En esta sección se va a explicar cada una de ellas.

4.3.1. Actividad entre gestor y agentes

Es una vía necesaria para que el gestor de agentes pueda comunicarse con los agentes y de esa forma controlarlos. Esto se consigue en JAIN SLEE mediante la creación de lo que se conoce como una *nullActivity*, que es proporcionada por una de las herramientas SLEE. Al iniciarse la plataforma, el gestor crea la *nullActivity*. Esta actividad permite enviar eventos desde un SBB hacia otro que esté configurado para recibirlos. En este caso, estos eventos permiten iniciar y parar los agentes, son eventos diseñados para que en la plataforma el servicio gestor de agentes pueda controlar todos los agentes.
El ciclo de vida de un agente depende de la recepción de estos eventos, cuando recibe el evento de tipo iniciar agente, SLEE creará una entidad SBB de tipo agente y la configurará con los parámetros incluidos en el evento. A partir de ese momento podrá comenzar a comunicarse con los recursos externos y el agente recibirá eventos para procesarlos con el motor de reglas y generar respuestas. Cuando el agente recibe un evento de tipo parar agente, el agente tiene que estar bien programado dentro de SLEE para desligarse de todas las actividades y cerrar conexiones con los adaptadores de recursos para que el contenedor pueda eliminar la entidad SBB correctamente.

4.3.2. Actividad entre adaptador de recursos y gestor o agentes

Estas actividades se crean cuando un SBB espera recibir eventos de un recurso externo. Cuando el servicio es de tipo petición/respuesta, la actividad dura hasta que se recibe la respuesta, en el caso de conexiones persistentes como XMPP, la actividad se crea cuando el SBB inicia la conexión y recibe eventos a través de ella mientras la conexión esté abierta. Cuando el SBB inicia la conexión, se liga con esa actividad, y el adaptador de recursos asocia la actividad a esa conexión iniciada, de esta forma, cuando llega un evento a esa conexión, el adaptador de recursos sabe a qué actividad corresponde y dispara el evento en esa actividad para que llegue al SBB que se ligó con esa actividad. Al cerrarse la conexión, el SBB se desliga de la actividad y esta es eliminada por SLEE al comprobar que ningún SBB está interesado en recibir más eventos en esa actividad.

4.4. Eventos

Los eventos son los objetos que representan la información de un suceso ocurrido en un recurso externo. Si por ejemplo se trata de un mensaje XMPP, el evento contendrá información del emisor y el contenido del mensaje. Si es un mensaje HTTP, contendrá toda la información de la petición para poder ser procesada.

4.4.1. Eventos internos de la arquitectura

Estos eventos son los que se disparan por el gestor de agentes en la *nullActivity* hacia los agentes. Y como caso especial, se hará mención de otro evento interno proporcionado por una de las herramientas SLEE que permite configurar eventos temporizados.

StartAgentEvent

Cuando el gestor de agentes recibe la orden de iniciar un agente, crea un objeto de tipo *StartAgentEvent* con toda la información necesaria para iniciar correctamente el agente. Una vez creado, lo dispara en la *nullActivity*. El SBB de los agentes está configurado para recibir este tipo de eventos, además este evento en concreto es de tipo *initial*, esto quiere decir que cuando se dispare este evento dentro de la arquitectura se creará una nueva entidad SBB agente y se configurará con la información contenida en él. Es importante destacar que se ha decidido crear una sola *nullActivity* para todos los agentes y no una por cada agente. Esto simplifica la solución porque solo hay que mantener abierta una actividad, pero hay que tener en cuenta que todos los eventos disparados en esta actividad llegan a todos los SBB agente, porque así es como funciona SLEE, y por ello es necesario un filtrado en los SBB

agente que permite determinar a qué agente va dirigido el evento, o si es necesario crear un nuevo agente.

StopAgentEvent

Este evento funciona de la misma forma que el anterior, pero su objetivo es parar de forma adecuada un agente dentro de la arquitectura para que pueda cerrar todas las conexiones que estén abiertas y realizar las acciones necesarias antes de ser parado y eliminado posteriormente por el contenedor SLEE. Como el estado de un agente depende de su información persistente, si se vuelve a reiniciar, el agente puede volver a recuperar el estado en el que quedó antes de parar. Esto depende de cómo esté programada su base de conocimiento con Drools.

TimerEvent

Este es un tipo de evento especial que se dispara dentro del contenedor SLEE. Una de las herramientas SLEE permite configurar este tipo de eventos y que sean recibidos por la entidad SBB interesada.
En el caso de la arquitectura SLAA, una de las funcionalidades de los agentes, es que puedan tomar decisiones sin que ningún recurso externo haya generado un evento, esto proporciona la característica de proactividad a los agentes. Un agente puede configurar un temporizador para que de forma eventual o periódica pueda analizar cualquier dato y en el caso que sea necesario tomar una decisión dependiendo de cómo haya sido programada su base de conocimiento Drools. Como ocurre con otras funcionalidades, el agente posee en su interfaz unos métodos que le permite controlar todos los temporizadores que sean necesarios.

4.4.2. Eventos externos a la arquitectura

Los eventos externos a la arquitectura son generados cuando un adaptador de recursos recibe un suceso de una fuente externa. Por ejemplo en el caso del gestor de agentes, estos eventos se crean cuando un usuario envía una petición HTTP hacia la plataforma desde la interfaz web. El adaptador de recursos recibe la petición y crea un evento con toda la información del mensaje HTTP y lo dispara dentro de una actividad hacia el SBB configurado para recibir este tipo de eventos.
Los agentes reciben eventos de todos los recursos externos para los que han sido programados. En la siguiente sección se explicará cómo se procesan los eventos para que sean utilizados en el motor de reglas Drools.

4.4.3. Mapeado de eventos entre SLEE y Drools

Una de las soluciones planteadas para que los agentes procesen los eventos desde su base de conocimiento Drools, es añadirlos a su memoria de trabajo directamente tras ser recibidos en la entidad SBB. Esto supone un problema para el procesado de eventos porque cada evento contiene toda la información del mensaje y complica extraer de forma directa la información relevante para procesar el evento.
Para solucionar este problema, se plantea crear unos nuevos objetos análogos a los eventos recibidos pero con un preprocesado al ser recibidos en la entidad SBB. Esto consiste en analizar qué información es la que realmente interesa del evento y añadirla en campos específicos del nuevo objeto, que será el que finalmente se entregue en la memoria de trabajo Drools del agente. En la mayoría de los casos solo consiste en añadir métodos *accesores* para disponer de la información de interés de inmediato y evaluarla desde el motor de reglas.

4.5. Herramientas SLEE

La arquitectura hace uso de algunas de las herramientas SLEE para obtener de ellas funcionalidades necesarias para cumplir parte de los requisitos planteados. Las principales herramientas que se usan son los temporizadores y las tablas de datos SLEE. Los temporizadores se utilizan para que los agentes puedan tomar decisiones aunque no hayan recibido ningún evento de un recurso externo y actuar de forma más allá de la puramente reactiva, y las tablas de datos se aprovechan para almacenar información persistente.

4.5.1. Temporizadores

En SLEE los temporizadores se configuran desde las entidades SBB. La forma en la que se configuran consiste en crear y ligar la entidad SBB a una actividad en la que se recibirán los eventos asociados al temporizador. Después el temporizador se configura con un identificador, el número de repeticiones, la duración entre repeticiones y la actividad en la que se recibirán los eventos. De esta forma los agentes pueden crear temporizadores configurados de la forma que necesiten y recibirán un evento por cada periodo del temporizador con un identificador para saber con qué objetivo se creó ese temporizador. Para ello, como con todas las funcionalidades, el agente dispone en su interfaz las funciones necesarias que simplifican la creación y control de temporizadores.

4.5.2. Tablas de datos

Las tablas de datos son otra de las herramientas que proporciona SLEE y sirven para almacenar información de forma persistente dentro del contenedor SLEE. Como ya se co-

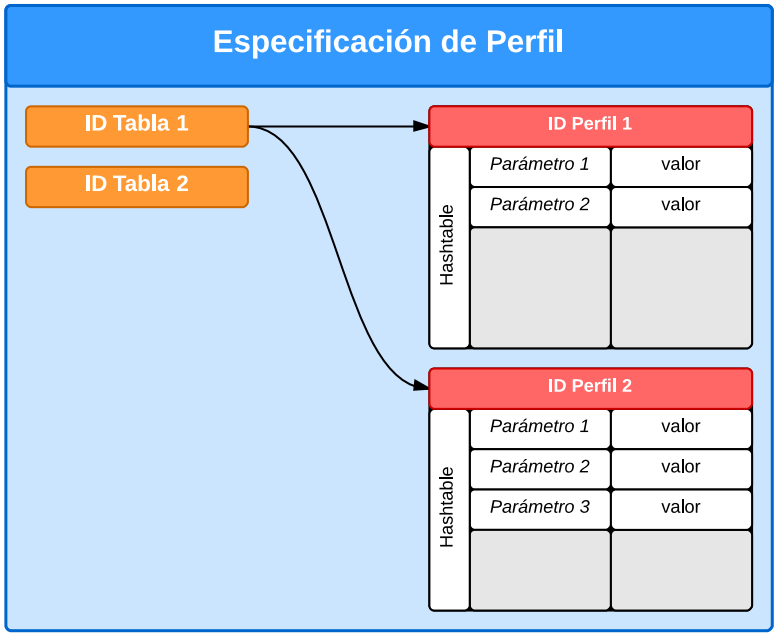

Figura 4.7: Ejemplo de tablas de datos

mentó en los capítulos de introducción, primero se define un componente SLEE conocido como especificación de perfil. Este componente se despliega como cualquier otro dentro del contenedor y en él se define cómo será la estructura de los datos almacenados en esos perfiles. En esta arquitectura la especificación de perfil se ha definido como una tabla basada en nombre de parámetro y valor.

Cuando ya se dispone de la especificación del perfil desplegada en el contenedor, una entidad SBB puede crear tablas de datos. Las tablas de datos son las que contendrán los diversos perfiles con los datos en el formato de la especificación del perfil. Para entender mejor este concepto, al final una tabla de datos es un identificador que contiene una lista de perfiles. Cada perfil está definido también por un identificador y finalmente el contenido del perfil son los datos en el formato de la especificación del perfil.

En la figura 4.7 se muestra un ejemplo visual de una especificación de perfil para la que se han creado dos tablas. La primera de ellas contiene dos perfiles de datos con diferente información almacenada, y la segunda de ellas no contiene ningún perfil de datos. Siguiendo este esquema, así es como se almacena información en la arquitectura SLAA.

Tabla de datos del gestor de agentes

El gestor de agentes tiene una tabla de datos en la que almacena los perfiles necesarios para el funcionamiento la plataforma SLAA. Uno de los perfiles representa la configuración de la plataforma en sí, y el resto representa a cada agente de la plataforma. El contenido

son los datos de configuración necesarios para que todo funcione correctamente.

Tabla de datos de agentes

Cada agente dispone de su propia tabla de datos. En ella estarán almacenados todos los perfiles de datos del agente. Algunos de esos perfiles son comunes a todos los agentes y contienen información necesaria para su funcionamiento. Los demás perfiles son los que el agente ha ido creando y almacenando datos desde su base de conocimiento Drools utilizando su interfaz. Esta le permite crear tantos perfiles como desee, y en ellos añadir los parámetros y valores que sean necesarios, acceder a ellos o eliminarlos cuando convenga.

4.6. Servicios

La arquitectura está implementada como servicios SLEE. Como se ha visto en los capítulos de introducción, un servicio en sí no es más que un elemento de despliegue. En él se identifica qué SBB será raíz y su prioridad. Como se verá en esta sección, cada servicio está compuesto por su descriptor de despliegue haciendo referencia a un SBB que será el que contiene el código del servicio. De los descriptores de despliegue y las clases Java solo se van a destacar los elementos relevantes de cada uno.

4.6.1. Servicio gestor de agentes

Este es el servicio que dentro de la arquitectura está definido como gestor de agentes. Está compuesto por el descriptor de despliegue del código 4.1 y el SBB *AgentManagerSbb* al que hace referencia.

Código 4.1: agent-manager-service.xml

```
<service-xml>
  <service>
    <service-name>AgentManagerService</service-name>
    <service-vendor>es.upm.dit.gsi</service-vendor>
    <service-version>1.0</service-version>
    <root-sbb>
      <sbb-name>AgentManagerSbb</sbb-name>
      <sbb-vendor>es.upm.dit.gsi</sbb-vendor>
      <sbb-version>1.0</sbb-version>
    </root-sbb>
    <default-priority>0</default-priority>
  </service>
</service-xml>
```

En el descriptor de despliegue se define como SBB *root* el *AgentManagerSbb*. Este SBB es el que permite crear y eliminar agentes y usuarios, su control, configuración y la recepción de órdenes a través de la interfaz web.

4.6.1.1. AgentManagerSbb

4.6.1.2. Descriptor de despliegue

A continuación se van a destacar los elementos más importantes del descriptor de despliegue de *AgentManagerSBB*.

Descripción del SBB (código 4.2)

Para identificar el SBB dentro de la plataforma es necesario especificar los parámetros básicos que se indican en el código.

Código 4.2: agent-manager-description

```
<sbb-name>AgentManagerSBB</sbb-name>
<sbb-vendor>es.upm.dit.gsi</sbb-vendor>
<sbb-version>1.0</sbb-version>
```

Clases del SBB (código 4.3)

En este apartado del descriptor del SBB se indica la clase Java que contiene el código del SBB y los campos CMP que se van a utilizar. En el campo CMP se almacenará la actividad usada para la comunicación entre el gestor y los agentes.

Código 4.3: agent-manager-classes

```
<sbb-classes>
  <sbb-abstract-class>
    <sbb-abstract-class-name>
      es.upm.dit.gsi.slaa.services.AgentManagerSbb
    </sbb-abstract-class-name>
    <cmp-field>
      <cmp-field-name>nullACI</cmp-field-name>
    </cmp-field>
  </sbb-abstract-class>
</sbb-classes>
```

Adaptadores de recursos que usa el SBB (código 4.4)

Cuando un SBB va a hacer uso de un adaptador de recursos es necesario indicar todos los parámetros que lo identifican dentro del contenedor SLEE. En este caso, el gestor de agentes solo hará uso del adaptador de recursos HTTP servidor para poder recibir peticiones desde los clientes web.

Código 4.4: agent-manager-ra-binding

```
<resource-adaptor-type-binding>
  <resource-adaptor-type-ref>
    <resource-adaptor-type-name>
      HttpServletResourceAdaptorType
    </resource-adaptor-type-name>
    <resource-adaptor-type-vendor>
      org.mobicents
    </resource-adaptor-type-vendor>
    <resource-adaptor-type-version>1.0</resource-adaptor-type-version>
  </resource-adaptor-type-ref>
  <resource-adaptor-entity-binding>
    <resource-adaptor-entity-link>
      HttpServletRA
    </resource-adaptor-entity-link>
  </resource-adaptor-entity-binding>
</resource-adaptor-type-binding>
```

Eventos del SBB (código 4.5)

Aquí se muestran algunos de los eventos que puede recibir el gestor de agentes. En el descriptor de despliegue es necesario indicar el identificador del tipo de evento que estará desplegado en SLEE, si va a ser recibido, disparado o si es de tipo inicial. Como puede verse en el código al que se hace referencia, en este caso el SBB puede recibir dos tipos de eventos de tipo inicial.
ServiceStartedEvent es un evento interno que dispara SLEE hacia una entidad SBB cuando es iniciada y es útil para realizar acciones en ese momento. También *GET* es un evento de tipo inicial, por ello es necesario proporcionar a SLEE una forma de determinar si la entidad SBB ya ha sido creada, porque en caso contrario, se creará una entidad SBB por cada recepción de un evento de tipo inicial. Para eso se utiliza el campo *initial-event-selector* que sirve para tener una forma de hacer saber a SLEE si ya se creó la entidad SBB y no es necesario crear otras nuevas. En este caso es importante porque solo es necesaria una entidad gestor de agentes en la plataforma.

Código 4.5: agent-manager-events

```
<event event-direction="Receive" initial-event="True">
  <event-name>ServiceStartedEvent</event-name>
  <event-type-ref>
    <event-type-name>
      javax.slee.serviceactivity.ServiceStartedEvent
    </event-type-name>
    <event-type-vendor>javax.slee</event-type-vendor>
    <event-type-version>1.1</event-type-version>
  </event-type-ref>
  <initial-event-selector-method-name>
    onServiceStartedEventInitialEventSelect
  </initial-event-selector-method-name>
</event>
<event event-direction="Fire" initial-event="False">
  <event-name>StartAgentEvent</event-name>
  <event-type-ref>
    <event-type-name>StartAgentEvent</event-type-name>
    <event-type-vendor>es.upm.dit.gsi</event-type-vendor>
    <event-type-version>1.0</event-type-version>
  </event-type-ref>
</event>
<event event-direction="Receive" initial-event="True">
  <event-name>GET</event-name>
  <event-type-ref>
    <event-type-name>
      net.java.slee.resource.http.events.incoming.request.GET
    </event-type-name>
    <event-type-vendor>net.java.slee</event-type-vendor>
    <event-type-version>1.0</event-type-version>
  </event-type-ref>
  <initial-event-selector-method-name>
    onGETInitialEventSelect
  </initial-event-selector-method-name>
</event>
```

Especificación de perfil del SBB (código 4.6)

Finalmente se muestra el código en el que se indican los campos que identifican la especificación de perfil que se va a utilizar y que estará desplegada en el contenedor SLEE.

Código 4.6: agent-manager-profile

```
<profile-spec-ref>
  <profile-spec-name>SlaaProfileSpec</profile-spec-name>
  <profile-spec-vendor>es.upm.dit.gsi</profile-spec-vendor>
  <profile-spec-version>1.0</profile-spec-version>
  <profile-spec-alias>SlaaProfile</profile-spec-alias>
</profile-spec-ref>
```

4.6.1.3. AgentManagerSBB.class

Para explicar los elementos clave de la clase que compone el gestor de agentes se van a destacar solo los elementos más importantes que hacen posible su funcionamiento. En los siguientes apartados se van a detallar cada uno de ellos.

Contexto del SBB (código 4.7)

Cada vez que SLEE crea una instancia del SBB, establece su contexto. El gestor de agentes va a necesitar las herramientas SLEE que permitirán crear la *nullActivity* para comunicarse con los agentes, y la especificación de perfil para almacenar datos persistentes.

Código 4.7: AgentManagerSBB setSbbContext

```
public void setSbbContext(SbbContext context) {
  this.sbbContext = (SbbContextExt) context;

  Context myEnv = (Context) new InitialContext();
  this.nullACIFactory = (NullActivityContextInterfaceFactory) myEnv.lookup(
      NullActivityContextInterfaceFactory.JNDI_NAME);
  this.nullActivityFactory = (NullActivityFactory) myEnv.lookup(
      NullActivityFactory.JNDI_NAME);

  profileFacility = (ProfileFacility)myEnv.lookup(ProfileFacility.JNDI_NAME);
}
```

Evento de servicio iniciado (código 4.8)

En el descriptor de despliegue se indicaron los eventos de los que iba a hacer uso el gestor de agentes. En el código Java será necesario implementar los métodos que procesarán esos eventos.
Este evento es el que se recibe cuando se inicia el servicio. Este servicio es el que representa a la plataforma, por lo que será necesario realizar todas las acciones necesarias para que la plataforma esté operativa. Los elementos más importantes son la tabla de perfiles que usará el gestor para guardar los diferentes perfiles con información persistente relativa a los usuarios, los agentes y la plataforma en sí. Además creará, almacenará en un campo CMP y se ligará a la *nullActivity* en la que disparará los eventos para controlar las entidades SBB agente y finalmente restaurará el estado en el que estaban todos los agentes de la plataforma en el momento en el que se paró por última vez.

Código 4.8: AgentManagerSBB onEvents

```
public void onServiceStartedEvent(ServiceStartedEvent event,
    ActivityContextInterface aci) {
  this.createProfileTable("agentManagerProfileTable");

  this.createProfile("usersProfile");
  this.createProfile("agentsProfile");
  this.createProfile("platformProfile");

  aci.detach(sbbContext.getSbbLocalObject());

  ActivityContextInterface nullACI = this.nullACIFactory.
  getActivityContextInterface(this.nullActivityFactory.createNullActivity());
  this.setNullACI(nullACI);
  nullACI.attach(sbbContext.getSbbLocalObject());

  this.setProfileParam("platformProfile", "status", "started");
  this.startAllAgents();
}
```

Evento HTTP tipo get (código 4.9)

El evento tipo get es el que se recibe cada vez que un usuario envía una petición desde la interfaz web. En él se destaca la identificación del tipo de petición que hace el usuario para saber que acción ejecutar, la recuperación de la *nullActivity* a través del campo CMP para

comunicarse con los agentes si es necesario y procesar la petición realizando las acciones necesarias y generando una respuesta en formato JSON para que en el cliente web haya conocimiento del resultado de la petición. También se puede destacar cómo es importante desligarse de la actividad en la que se recibe el evento porque con este tipo de peticiones en las que no se mantiene sesión, la actividad se crea ante la llegada de un nuevo evento y en la anterior ya no se recibirán más, por lo que de esta forma SLEE detectará que esa actividad ya no se va a utilizar y la podrá eliminar adecuadamente.

Código 4.9: AgentManagerSBB onGET

```
public void onGET(HttpServletRequestEvent event, ActivityContextInterface aci
    ) {
  String req = event.getRequest().getParameter("req");

  ActivityContextInterface nullACI = this.getNullACI();

  HttpServletResponse response = event.getResponse();
  HttpServletRequest request = event.getRequest();
  response.setContentType("text/javascript;charset=UTF-8");
  response.setHeader("Access-Control-Allow-Origin:", "*");
  PrintWriter out = response.getWriter();

  if (req.equals("startAgent")) {
    String agentId = event.getRequest().getParameter("agentName") + "@" +
        event.getRequest().getParameter("userName");
    this.startAgent(agentId);
    String callback = request.getParameter("callback");
    String json = "{ 'platform': [ ";
    json += " { 'name':'status' , 'value':'" + this.getProfileParam("
        platformProfile", "status") + "'},  ";
    json += "]}";
    String renderHTML = callback + "(" + json + ")";
    out.println(renderHTML);
    out.close();
  }
  aci.detach(sbbContext.getSbbLocalObject());
}
```

Selector de evento inicial (código 4.10)

Cuando una entidad SBB puede recibir varios eventos de tipo inicial y no se requiere que por cada uno de ellos se cree una nueva entidad, con estos métodos se puede establecer

un identificador ante la llegada de estos eventos y solo con el primero se creará una entidad. Con los siguientes eventos, el identificador ya se reconoce como creado anteriormente, y de esta forma SLEE sabe que no es necesario crear una nueva entidad, sino que la entidad ya creada es la que procesará el evento.

Código 4.10: AgentManagerSBB onInitialEventSelect

```
public InitialEventSelector onGETInitialEventSelect(InitialEventSelector ies)
    {
  ies.setCustomName("ManagerSbbCustomName");
  return ies;
}

public InitialEventSelector onServiceStartedEventInitialEventSelect(
   InitialEventSelector ies) {
  ies.setCustomName("ManagerSbbCustomName");
  return ies;
}
```

Método para iniciar agentes (código 4.11)

Cuando el gestor de agentes va a iniciar un agente, el proceso es crear el evento diseñado para ello con los datos del agente y dispararlo dentro de la *nullActivity*. Después SLEE creará una nueva entidad SBB agente y en el método que procesa ese evento se ejecutarán las acciones necesarias y las determinadas por su JSON de configuración. Cuando un SBB puede disparar eventos, es necesario añadir en SLEE el método abstracto dedicado a ello.

Código 4.11: AgentManagerSBB startAgent

```
private void startAgent(String agentID) {
  ActivityContextInterface nullACI = this.getNullACI();

  StartAgentEvent event = new StartAgentEvent();
  event.setAgentID(agentID);
  event.setAgentJson((String) this.getProfileParam(agentID, "agentJson"));

  this.fireStartAgentEvent(event, nullACI, null);
  this.setProfileParam(agentID, "status", "started");
}

public abstract void fireStartAgentEvent(StartAgentEvent event,
   ActivityContextInterface aci, Address address);
```

Métodos de los campos CMP del SBB (código 4.12)

Los campos CMP definidos en el descriptor de despliegue del SBB necesitan tener definidos métodos abstractos para acceder o almacenarlos. El gestor de agentes necesita un campo CMP para almacenar la *nullActivity*.

Código 4.12: AgentManagerSBB CMP

```
public abstract void setNullACI(javax.slee.ActivityContextInterface value);

public abstract javax.slee.ActivityContextInterface getNullACI();
```

Gestión de perfiles de datos (código 4.13)

En el siguiente código se pueden ver los métodos encargados de crear tablas de datos para contener perfiles, métodos para crear perfiles que contendrán los datos, y métodos para almacenar y leer datos de los perfiles.

Código 4.13: AgentManagerSBB profiles

```
private void createProfileTable(String tableId) {
  ProfileSpecificationID profileSpecificationID = new ProfileSpecificationID(
      "SlaaProfileSpec", "es.upm.dit.gsi", "1.0");
  Object[] params = { profileSpecificationID, tableId };
  String[] signatures = {profileSpecificationID.getClass().getName(), tableId
      .getClass().getName() };
  SleeContainer.lookupFromJndi().getMBeanServer().invoke(new ObjectName(
      ProfileProvisioningMBean.OBJECT_NAME), "createProfileTable", params,
      signatures);
}


private void createProfile(String profileName) {
  ProfileTable profileTable = profileFacility.getProfileTable(
      profileTableName);
  profileTable.create(profileName);
}

private void setProfileParam(String profileName, String paramName, Object
    paramValue) {
  ProfileTable profileTable = profileFacility.getProfileTable(
      profileTableName);
  java.util.Hashtable<String, Object> hT = ((AgentsProfileCMP) profileTable.
      find(profileName)).getList();
```

```
    hT.put(paramName, (String) paramValue);
    ((AgentsProfileCMP) profileTable.find(profileName)).setList(hT);
  }

  private Object getProfileParam(String profileName, String paramName) {
    ProfileTable profileTable = profileFacility.getProfileTable(
        profileTableName);
    java.util.Hashtable<String, Object> hT = ((AgentsProfileCMP) profileTable.
        find(profileName)).getList();
    return hT.get(paramName);
  }
```

4.6.1.4. Diagrama funcional

La principal función del gestor de agentes entre otras es recibir peticiones de los usuarios en SLAA. En el diagrama de la figura 4.8 se puede seguir todo el proceso llevado a cabo desde que un usuario solicita una petición hasta que recibe una respuesta.
Resumiendo el diagrama, el gestor de agentes recibe las peticiones de los usuarios y procesa los datos contenidos en ella. Después realiza acciones en la plataforma que pueden implicar acceder a datos persistentes. Por ejemplo, una de las acciones puede ser crear un agente nuevo. Finalmente, si es necesario, el gestor puede enviar una respuesta para que el usuario tenga conocimiento de que la acción llevada a cabo ha sido satisfactoria.

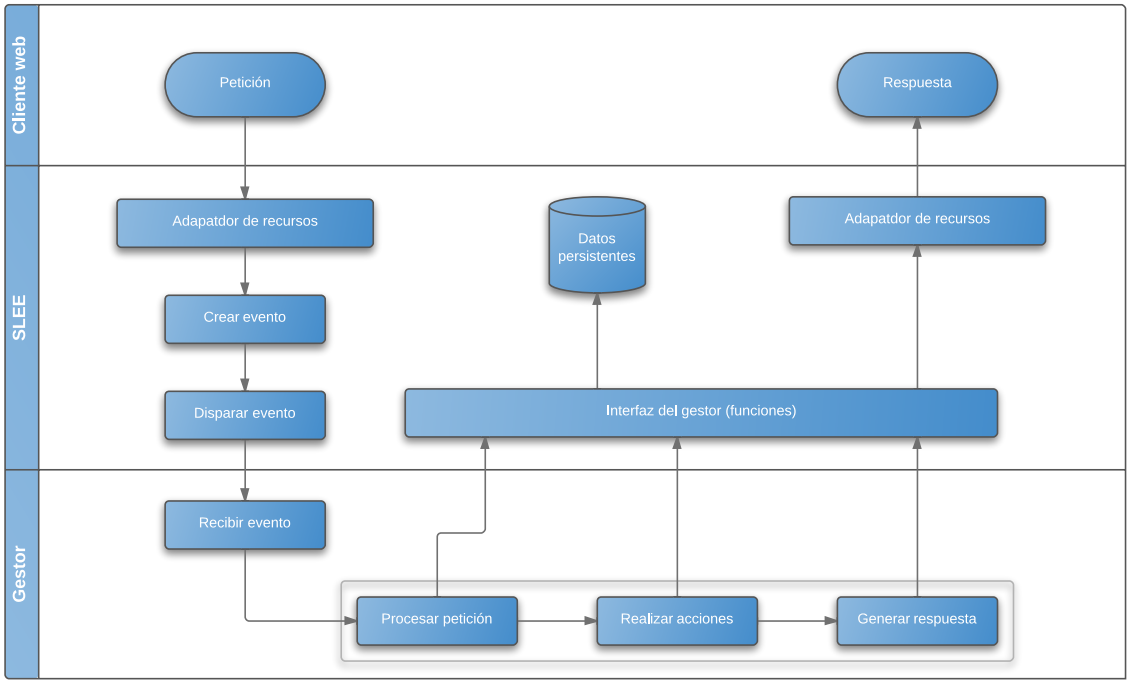

Figura 4.8: Diagrama funcional del gestor de agentes

4.6.2. Servicio agente

Este servicio es el que compone un agente en la arquitectura SLAA. Su descriptor de despliegue es el código 4.14 y el SBB es *SlaaAgentSBB*.

Código 4.14: slaa-agent-service.xml

```
<service-xml>
  <service>
    <service-name>SlaaAgentService</service-name>
    <service-vendor>es.upm.dit.gsi</service-vendor>
    <service-version>1.0</service-version>
    <root-sbb>
      <sbb-name>SlaaAgentSBB</sbb-name>
      <sbb-vendor>es.upm.dit.gsi</sbb-vendor>
      <sbb-version>1.0</sbb-version>
    </root-sbb>
    <default-priority>0</default-priority>
  </service>
</service-xml>
```

4.6.2.1. SlaaAgentSbb

El agente como cualquier SBB en SLEE está formado por su descriptor de despliegue y su clase Java. En los siguientes apartados se van a destacar los elementos más relevantes y aquellos análogos al caso del gestor de agentes se van a omitir porque son similares.

4.6.2.2. Descriptor de despliegue

Adaptadores de recursos que usa el SBB (código 4.15)

En el caso del agente este apartado del descriptor de despliegue muestra cómo se hace referencia al adaptador de recursos XMPP y sus interfaces para poder iniciar la conexión, enviar mensajes y crear actividades desde la entidad SBB.

Código 4.15: slaa-agent-ra-binding

```
<resource-adaptor-type-binding>
  <resource-adaptor-type-ref>
    <resource-adaptor-type-name>
      XMPPResourceAdaptorType
    </resource-adaptor-type-name>
    <resource-adaptor-type-vendor>
```

```
      org.mobicents
    </resource-adaptor-type-vendor>
    <resource-adaptor-type-version>2.0</resource-adaptor-type-version>
  </resource-adaptor-type-ref>
  <activity-context-interface-factory-name>
    slee/resources/xmpp/2.0/factoryprovider
  </activity-context-interface-factory-name>
  <resource-adaptor-entity-binding>
    <resource-adaptor-object-name>
      slee/resources/xmpp/2.0/xmppinterface
    </resource-adaptor-object-name>
    <resource-adaptor-entity-link>XMPPRA</resource-adaptor-entity-link>
  </resource-adaptor-entity-binding>
</resource-adaptor-type-binding>
```

Eventos del SBB (código 4.16)

Este es el evento que recibe el agente por parte del gestor para ser iniciado. Es necesario establecer un selector de evento inicial para el caso en el que un agente vaya a recibir más eventos de tipo inicial y no sea necesario crear una nueva entidad SBB. Solo se crearán entidades SBB agente ante la llegada de este evento.

Código 4.16: slaa-agent-events

```
<event event-direction="Receive" initial-event="True">
  <event-name>StartAgentEvent</event-name>
  <event-type-ref>
    <event-type-name>StartAgentEvent</event-type-name>
    <event-type-vendor>es.upm.dit.gsi</event-type-vendor>
    <event-type-version>1.0</event-type-version>
  </event-type-ref>
  <initial-event-selector-method-name>
    agentIDSelect
  </initial-event-selector-method-name>
</event>
```

4.6.2.3. SlaaAgentSBB.class

Contexto del SBB (código 4.17)

Lo que se va a destacar del contexto del SBB agente es que es necesario obtener todas las interfaces de los adaptadores de recursos y herramientas SLEE que se van a utilizar como es el caso de los temporizadores.

Código 4.17: SlaaAgentSBB setSbbContext

```
public void setSbbContext(SbbContext context) {
  this.sbbContext = (SbbContextExt) context;

  Context myEnv;
  Context myEnv2;

  myEnv = (Context) new InitialContext().lookup("java:comp/env");

  httpSbbInterface = (HttpClientResourceAdaptorSbbInterface) myEnv
      .lookup("slee/resources/http-client/sbbinterface");
  httpActivityContextInterfaceFactory = (
      HttpClientActivityContextInterfaceFactory) myEnv
      .lookup("slee/resources/http-client/acifactory");

  xmppSbbInterface = (XmppResourceAdaptorSbbInterface) myEnv
      .lookup("slee/resources/xmpp/2.0/xmppinterface");
  xmppActivityContextInterfaceFactory = (XmppActivityContextInterfaceFactory)
       myEnv
      .lookup("slee/resources/xmpp/2.0/factoryprovider");

  socialNetworkInterface = (SocialNetworkRAProvider) myEnv
      .lookup("slee/resources/social/2.0/socialrainterface");

  droolsInterface = (DroolsRAProvider) myEnv
      .lookup("slee/resources/drools/1.0/droolsrainterface");

  myEnv2 = (Context) new InitialContext();

  timerFacility = (TimerFacility) myEnv2
      .lookup(TimerFacility.JNDI_NAME);
}
```

Evento de agente iniciado (código 4.18)

En este método se procesa la llegada de un evento de tipo iniciar agente con su configuración. Lo que se hace es establecer el identificador del agente, crear los perfiles de datos que tienen todos los agentes por defecto, ligar la entidad SBB a la *nullActivity* de los agentes, obtener el JSON que se definió por parte del usuario desde la interfaz web y de él extraer los servicios que se van a utilizar para activarlos, y añadir un evento de tipo agente iniciado en la memoria de trabajo para que se dispare la regla de agente iniciado.

Código 4.18: SlaaAgentSBB onStartAgentEvent

```
public void onStartAgentEvent(StartAgentEvent event, ActivityContextInterface
    aci) {
  this.setAgentID(event.getAgentID());
  this.createProfile("timers");
  this.createProfile("logs");
  this.setParam("logs", "info", "");
  this.info("Started");
  aci.attach(sbbContext.getSbbLocalObject());
  JSONObject agentJson = new JSONObject(event.getAgentJson());
  JSONArray agentServices = agentJson.getJSONArray("services");

  for (int i = 0; i < agentServices.length(); i++) {
    if (agentServices.getJSONObject(i).getBoolean("isActive")) {
      if (agentServices.getJSONObject(i).getString("name").equals("XMPP")) {
        JSONArray fields = agentServices.getJSONObject(i).getJSONArray("
            fields");
        this.startXMPPService(fields.getJSONObject(0).getString("value"),
            fields.getJSONObject(1).getString("value"));
      }
      if (agentServices.getJSONObject(i).getString("name").equals("Twitter"))
          {
        JSONArray fields = agentServices.getJSONObject(i).getJSONArray("
            fields");
        this.startTwitterService(fields.getJSONObject(0).getString("value"),
            fields.getJSONObject(1).getString("value"),fields.getJSONObject
            (2).getString("value"),fields.getJSONObject(3).getString("value")
            );
      }

    }
  }
  fireRules(new OnAgentStarted());
}
```

Ejemplo de mapeado de evento (código 4.19)

Los eventos que llegan a la entidad SBB se encapsulan en otros con la información del evento depurada para que sea más fácil procesarlo desde Drools. En este caso se puede ver cómo la llegada de un mensaje XMPP en la entidad SBB se procesa y se crea un nuevo objeto que será el que se introduce en la memoria de trabajo para que se dispare la regla ante la llegada de un mensaje XMPP.

Código 4.19: SlaaAgentSBB onMessage

```
public void onMessage(Message message, ActivityContextInterface aci) {
  OnXmppMessage msg = new OnXmppMessage();
  msg.setFrom(message.getFrom().split("/")[0]);
  msg.setMessage(message.getBody());
  fireRules(msg);
}
```

Disparo de reglas en el agente (código 4.20)

Esta es la parte del código en la que se disparan las reglas. La operación consiste en obtener la memoria de trabajo a través de la interfaz del adaptador de recursos Drools y en ella introducir el evento que acaba de llegar a la entidad SBB. Además del evento, siempre se introduce como hecho el propio agente para que desde Drools se pueda utilizar la interfaz que se ha definido en el código del SBB que sirve para realizar diferentes funciones como puede verse en el apartado siguiente.

Código 4.20: SlaaAgentSBB fireRules

```
public void fireRules(Object fact) {
  WorkingMemory workingMemory = droolsInterface.getWorkingMemory(this.
      getAgentID());

  workingMemory.insert(fact);
  workingMemory.insert(this);

  workingMemory.fireAllRules();
  workingMemory.dispose();
}
```

Método de la interfaz del agente (código 4.21)

Esta es una de las funciones que se puede ejecutar desde Drools por parte del agente. Permite de forma inmediata enviar un mensaje de texto a un destinatario haciendo una llamada al método con los parámetros adecuados.

Código 4.21: SlaaAgentSBB sendXmppMessage

```
public void sendXmppMessage(String user, Object agentMessage) {
  String msg = agentMessage.toString();
  Message xmppMessage = new Message(user, Message.Type.CHAT);
  xmppMessage.setBody(msg);
  xmppSbbInterface.sendPacket(this.getAgentID(), xmppMessage);
}
```

4.6.2.4. Diagrama funcional

Funcionalmente, el esquema de un agente es muy parecido al del gestor de agentes. Dentro de la arquitectura, la forma en la que han sido programados es la misma, pero la gran diferencia reside en que el agente solo toma decisiones que estén programadas en su base de conocimiento.
En la figura 4.9 se puede ver cuál es el proceso completo que se lleva a cabo cuando un agente recibe un suceso producido por un recurso externo. La plataforma SLAA se encarga de que el suceso llegue al agente adecuado y este procesa todos los datos en función de lo que determine su base de conocimiento. Durante el proceso puede ejecutar funciones implementadas en su interfaz, por ejemplo para acceder a su memoria. Finalmente el resultado puede ser ejecutar otras funciones o generar un mensaje para que el recurso externo reciba una respuesta.

Se ha visto en este capítulo qué elementos componen la arquitectura y cómo se han programado usando la tecnología JAIN SLEE para conseguirlo. En los capítulos 5 y 6 se mostrará el resultado de esta implementación. En el capítulo 5 se presenta la interfaz web programada que permite interactuar con la arquitectura por parte de los usuarios, y en el capítulo 6 se presentará un caso de estudio real para demostrar su funcionamiento.

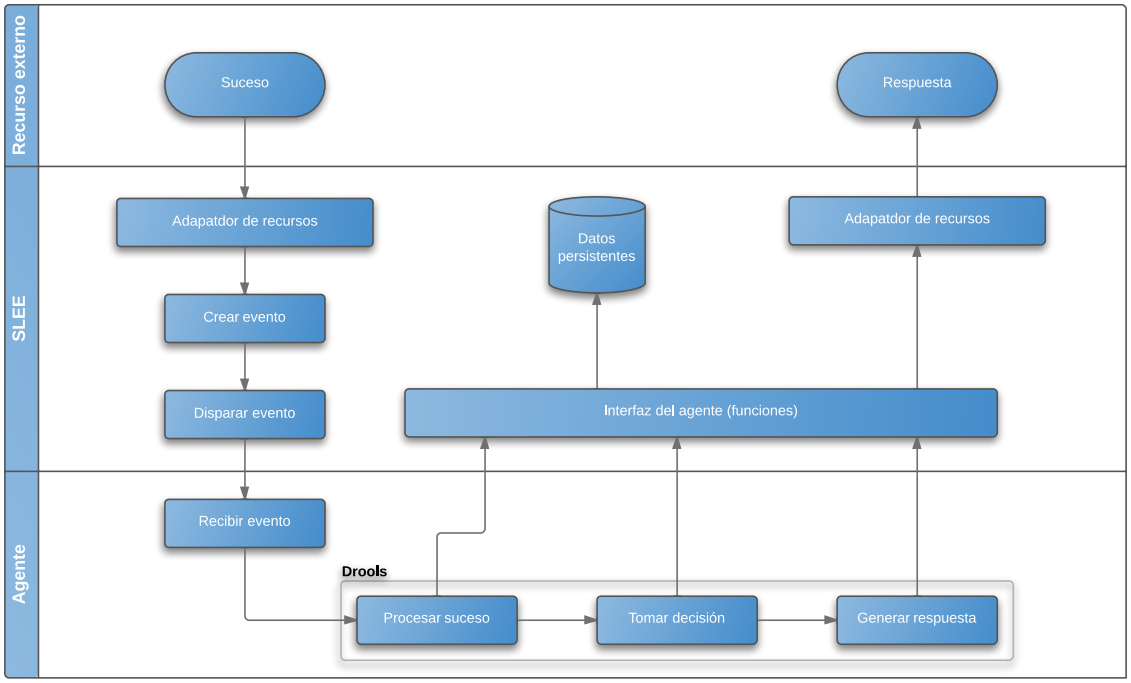

Figura 4.9: Diagrama funcional de un agente

Capítulo 5

Interfaz web de administración

Este capítulo presenta uno de los elementos clave que permite la interacción de los usuarios con la arquitectura de agentes. La interfaz web de administración es un desarrollo web completo que permite la visualización de información y el envío de peticiones al servidor de la arquitectura desde cualquier navegador web. Como punto a destacar es importante señalar que la configuración y programación de agentes también se realiza por completo desde esta interfaz.

5.1. Introducción

La interfaz web de administración que se presenta como prototipo sirve para tener una vista vía Internet del estado del servidor y poder interactuar con él. El funcionamiento se basa en el envío de órdenes para ejecutar acciones y el intercambio de datos en formato JSON [8] para ser representados. La principal tecnología utilizada es *knockout.js* [9], una librería de código abierto en JavaScript que simplifica la creación de interfaces y su interacción con el modelo de datos.

5.2. Vista de administración

Estas son las vistas a las que un administrador puede acceder. En ellas tiene acceso a todos los usuarios del servidor, sus agentes y puede realizar acciones de gestión de la plataforma.

Usuarios (figura 5.1)

En esta vista se puede ver la lista de todos los usuarios presentes en la plataforma. Permite crear otros nuevos o eliminar los existentes. Al eliminar un usuario, se eliminan todos sus agentes y toda su información dentro del servidor.

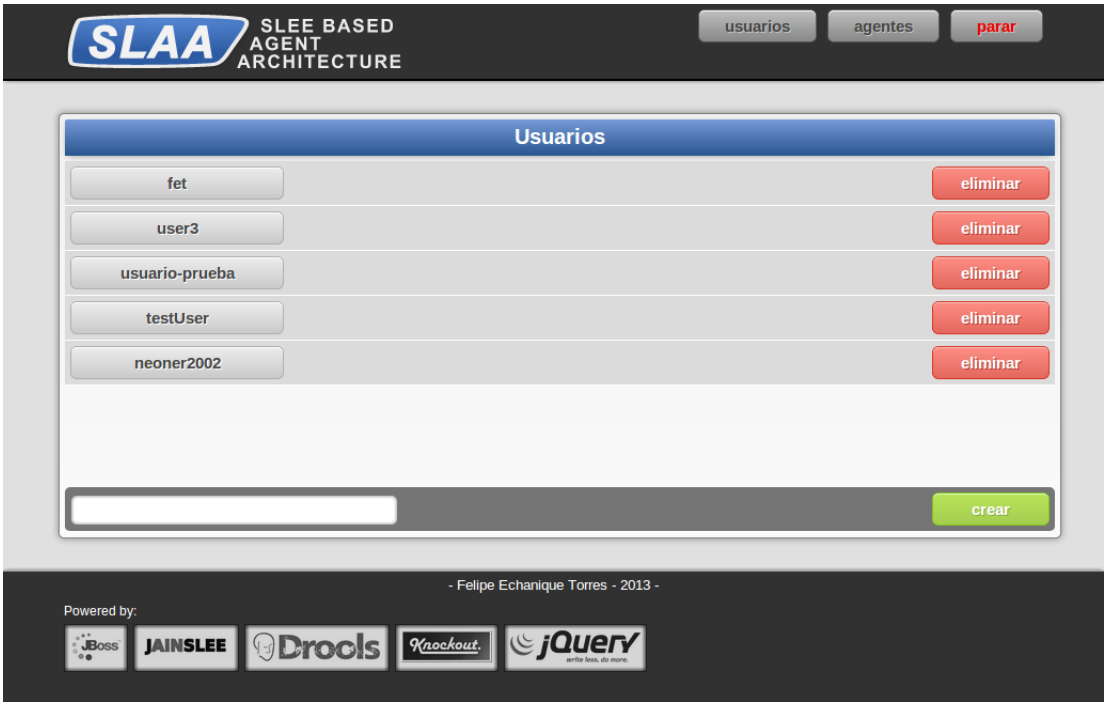

Figura 5.1: Vista de usuarios

Agentes (figura 5.2)

En esta lista se muestran todos los agentes de la plataforma y el usuario al que pertenecen. También permite su control y ver toda la información correspondiente de cada agente.

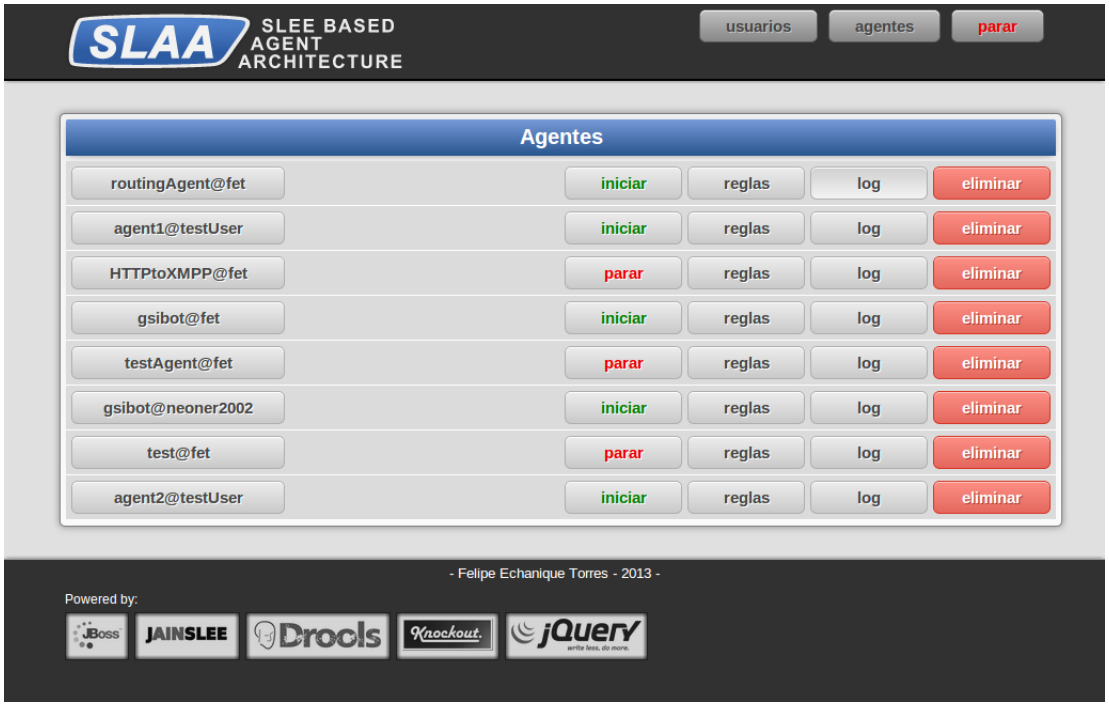

Figura 5.2: Vista de agentes

5.3. Vista de usuarios

Los usuarios de la plataforma también tienen su vista correspondiente. En ella pueden crear o eliminar sus agentes, configurarlos y programarlos.

Agentes del usuario (figura 5.3)

Un usuario tiene acceso a todos los agentes que ha creado. Desde esta lista puede controlar su estado, acceder a su configuración, ver y modificar las reglas o ver las trazas que haya ido generando.

Configuración del agente (figura 5.4)

Desde esta sección un usuario configura su agente. Sirve para poder seleccionar los servicios que quiera utilizar y en el caso de que sea necesario completar el formulario con los datos requeridos por el servicio.

Figura 5.3: Vista de agentes del usuario

Figura 5.4: Vista de configuración del agente

Reglas del agente (figura 5.5)

Una vez configurado un agente, desde esta vista se pueden modificar todas las reglas que rigen su comportamiento. En el ejemplo se muestra un caso básico en el que un agente muestra un mensaje a través de sus trazas tras ser iniciado.

Figura 5.5: Vista de reglas del agente

Trazas del agente (figura 5.6)

Es útil acceder a las trazas del agente para comprobar que su comportamiento es el adecuado. En la vista anterior se mostró una regla que imprime un mensaje en las trazas al ser iniciado un agente, en esta vista se puede ver el resultado de esa traza.

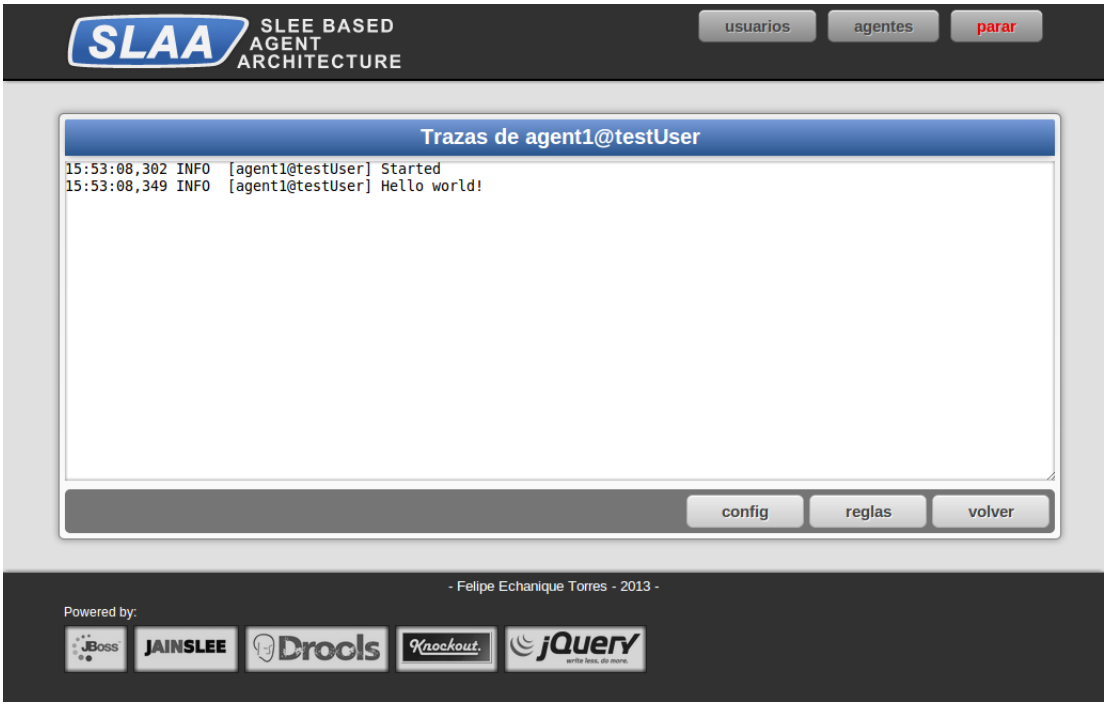

Figura 5.6: Vista de trazas del agente

Lo que se ha presentado en este capítulo es la parte que ven los usuarios de la arquitectura. Esta interfaz es desde la que podrán hacer uso de todas las funciones del sistema. En el siguiente capítulo, se muestra cómo haciendo uso de ella se puede programar un agente.

Capítulo 6

Caso de estudio: Agente enrutador de conversaciones

Este capítulo detalla cómo funciona y se programa un caso real utilizando la arquitectura desarrollada. Para ello se comenzará con una descripción funcional del caso de estudio elaborado y después un análisis completo del código que ha sido necesario programar para conseguir que funcione este agente.

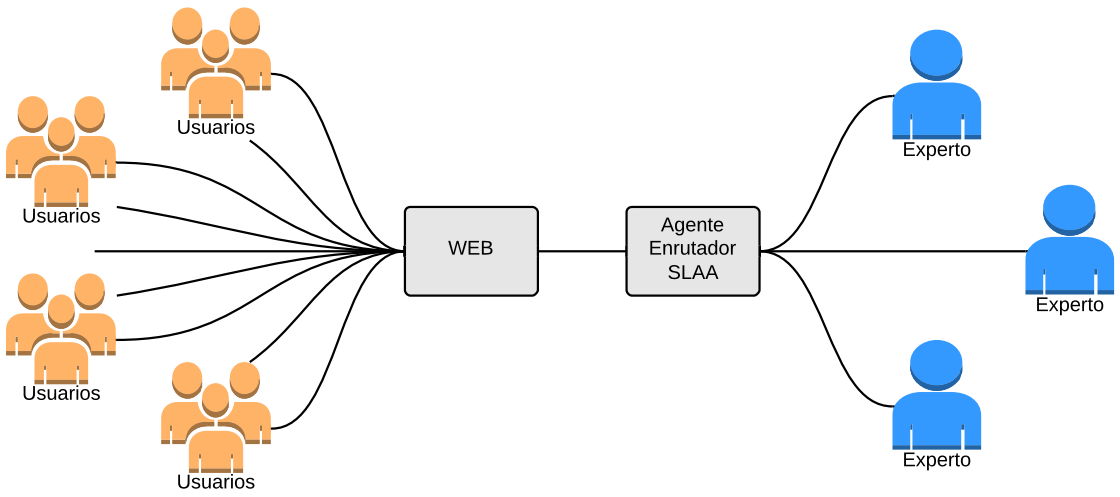

Figura 6.1: Escenario del caso de estudio

6.1. Introducción

Este caso de estudio permite comprender el funcionamiento de la arquitectura y lo ágil que es la creación con ella un agente que se comunica a través de múltiples protocolos de comunicaciones y ofrece un servicio.
El ejemplo consiste en un servicio que instalado a modo de *plugin* en cualquier web (código HTML + JavaScript) permite que los usuarios que acceden a ella se puedan comunicar directamente con otros usuarios, que en este caso se denominan expertos, y cuyo objetivo es orientar o resolver cualquier duda que pueda tener el usuario de la web.
Primero se va a explicar el escenario en el que sucede el caso de estudio, después los estados en los que puede estar el agente en un determinado momento y finalmente todas las reglas que rigen su comportamiento.

6.2. Escenario

El escenario de la figura 6.1 muestra la conexión entre todos los elementos más relevantes del caso de estudio. Por un lado se tiene un gran número de usuarios que accede a una web y que pueden requerir asistencia en tiempo real por parte de los expertos. Los expertos son usuarios que tienen al agente enrutador en su lista de contactos de mensajería instantánea de forma que éste puede comunicarse con ellos y trasladar los mensajes que envíen los usuarios desde la web para que se establezca una conversación directa entre usuarios y expertos.

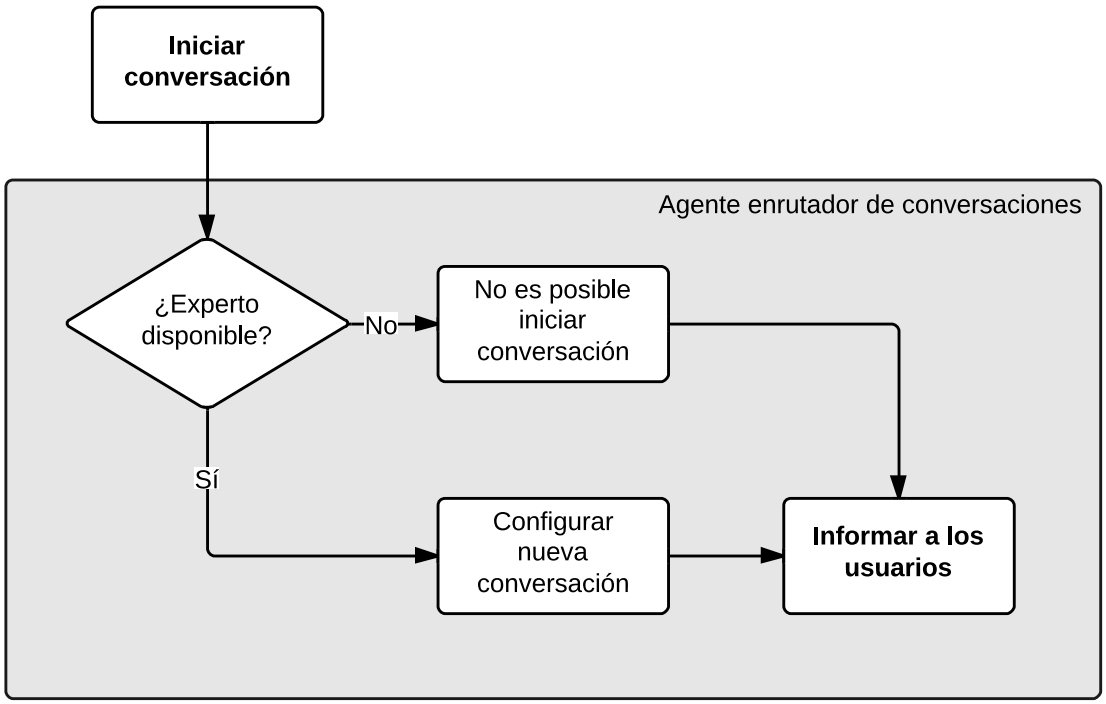

Figura 6.2: Secuencia de inicio de conversación

6.3. Estados del agente

Los estados en los que puede estar un agente respecto a una conversación son los tres que se describen a continuación. Cada uno de ellos corresponde al estado de iniciar una conversación, una conversación en curso, y una conversación finalizada.

6.3.1. Iniciar conversación

Cuando un usuario abre la ventana para iniciar una conversación, se envía un mensaje desde el cliente web hacia el agente a través de HTTP para intentar establecer una comunicación de mensajería instantánea entre el usuario y el experto. Al recibir el mensaje, el agente intentará buscar un experto disponible para que atienda al usuario, en el caso de que lo encuentre establecerá la comunicación entre ambos (figura 6.3), en caso contrario informará al usuario de que en ese momento no puede establecerse la comunicación y que lo intente más tarde (figura 6.4). Todo este proceso puede seguirse en la figura 6.2

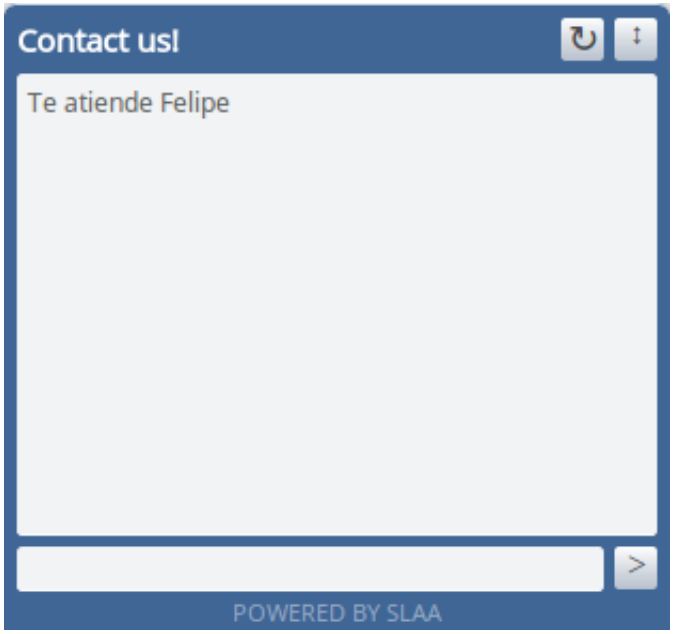

Figura 6.3: Ventana de conversación iniciada

Figura 6.4: Ventana de conversación fallida

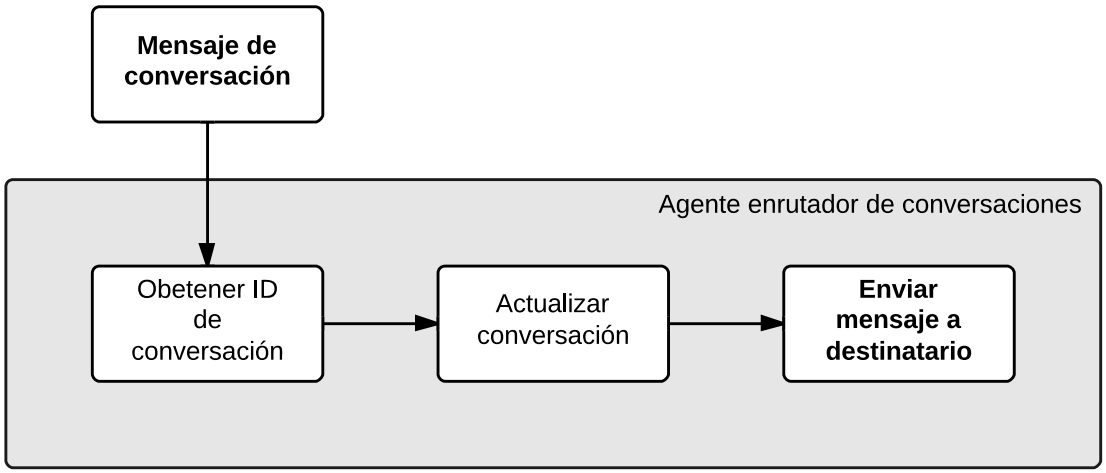

Figura 6.5: Secuencia de mensaje de conversación

6.3.2. Conversación

Una vez establecida la conversación, el usuario desde la web y el experto desde su cliente de mensajería instantánea pueden comenzar a intercambiar mensajes. Para que esto sea posible, el agente va recibiendo los mensajes de ambas partes y los redirige (figura 6.5) tal y como se estableció al iniciar la conversación. En la figura 6.6 se puede ver un ejemplo de conversación entre un usuario y un experto.

Figura 6.6: Ventana de conversación en curso

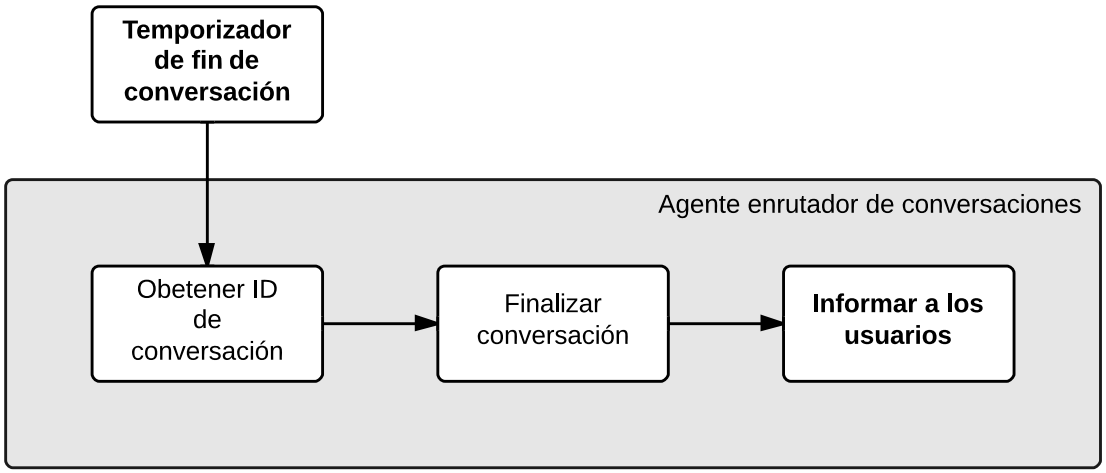

Figura 6.7: Secuencia de fin de conversación

6.3.3. Finalizar conversación

Una de las formas para finalizar una conversación automáticamente y volver a establecer al experto como disponible, es configurar un temporizador en el agente por cada conversación en curso para que tras un periodo de inactividad se finalice la comunicación. En el diagrama de la figura 6.7 se muestra el proceso que tiene lugar en el agente cada vez que se recibe un evento del temporizador, y en el caso determinado, finalizar la conversación. Una vez finalizada la conversación, se informa a cada una de las partes para que tengan constancia de ello. En la figura 6.8 se puede ver la ventana web con un mensaje que indica que la conversación ha finalizado.

6.4. Código del agente

En esta sección se muestra por completo todo el código que permite el funcionamiento del agente tal y como se ha descrito anteriormente.

6.4.1. Clases importadas

Código 6.1

Este código contiene todas las clases que son necesarias importar en el fichero de reglas. La primera de ellas representa al agente en sí, y permite acceder a su interfaz para realizar acciones. Las demás representan los diferentes hechos que pueden ocurrir durante el funcionamiento del agente, y son los que dispararán las reglas.

Figura 6.8: Ventana de conversación finalizada

Código 6.1: Clases importadas del agente enrutador

```
import es.upm.dit.gsi.slaa.services.SlaaAgentSbb;
import es.upm.dit.gsi.slaa.facts.OnAgentStarted;
import es.upm.dit.gsi.slaa.facts.OnXmppPresence;
import es.upm.dit.gsi.slaa.facts.OnXmppMessage;
import es.upm.dit.gsi.slaa.facts.OnHttpGet;
import es.upm.dit.gsi.slaa.facts.OnTask;
```

6.4.2. Regla de agente iniciado

Código 6.2

Esta regla se dispara tras ser el agente iniciado. En ella se establece la configuración inicial del agente, que consiste en determinar los usuarios que actuarán de expertos añadiendo en una lista sus direcciones de mensajería instantánea correspondiente y un nombre para ser mostrado.
Se crea un perfil para almacenar la configuración general del agente, otro perfil para la lista de expertos, otro para las conversaciones en curso, y otro por cada experto para almacenar su dirección de mensajería, iniciarlo como no disponible, el identificador de la conversación en curso y su nombre.

Código 6.2: Regla de agente iniciado

```
rule "Agent Started"
when
  started:OnAgentStarted()
  agent:SlaaAgentSbb()
then
  String[] experts = {"example@gmail.com"};
  String[] names = {"Felipe"};

  agent.setProfile("config");
    agent.setParam("config","numExperts", experts.length);
  agent.setProfile("experts");
  agent.setProfile("chats");
  for(int i=0 ; i<experts.length ; i++){
    agent.setProfile(experts[i]);
    agent.setParam("experts",i,experts[i]);
    agent.setParam(experts[i],"available","false");
    agent.setParam(experts[i],"chatId","");
    agent.setParam(experts[i],"name",names[i]);
  }
  agent.info("Received OnAgentStarted event!");
end
```

6.4.3. Regla de mensaje HTTP de inicio de conversación

Código 6.3

Cuando un usuario quiere iniciar una conversación desde el cliente web, este envía una petición HTTP con un identificador de sesión. El agente buscará entre la lista de expertos uno que esté disponible.
En el caso de no encontrar ninguno responderá la petición del cliente con un mensaje indicando esta situación.
Si encuentra un experto disponible se configurará una conversación indicando que el experto queda ocupado, añadiendo el identificador de la sesión para asociar el experto con esa conversación, y creando una variable en la lista de conversaciones para almacenar el historial de conversación que será mostrado en el cliente web en el que se muestra un primer mensaje.
Finalmente se crea un temporizador que disparará una regla con el identificador de la conversación que hará que esta finalice si durante un tiempo no se ha enviado ningún mensaje y se responde a la petición HTTP indicando que la conversación se ha iniciado correctamente.

Código 6.3: Regla de mensaje HTTP de inicio de chat

```
rule "HTTP Start Chat"
when
  http:OnHttpGet()
  agent:SlaaAgentSbb()
  eval(http.getParam("type").equals("start"))
then
  int numExperts = agent.getIntParam("config","numExperts");
  String chatId = http.getParam("id");
  for(int i=0 ; i<numExperts ; i++){
    String expert = agent.getStringParam("experts",i);
    String name = agent.getStringParam(expert,"name");
    if(agent.getStringParam(expert,"available").equals("true")){
      agent.info(expert + " disponible");
      agent.setParam(expert,"available","false");
      agent.setParam(expert,"chatId",chatId);
      agent.setParam("chats",http.getParam("id"),"Te atiende " + name + "\n\n
          ");
      agent.setTask(chatId,1,60000);
      http.setResponse("true");
      return;
    }
  }
  http.setResponse("false");
end
```

6.4.4. Regla de tarea temporizada

Código 6.4

Cuando se dispara esta regla es porque ha expirado el tiempo de inactividad de una conversación. Al darse esta situación se buscará la conversación con el identificador extraído del propio evento y se eliminará toda la información de esta dejando el experto disponible de nuevo.

Código 6.4: Regla de tarea temporizada

```
rule "On Task"
when
  task:OnTask()
  agent:SlaaAgentSbb()
then
```

```
  agent.info("On TaskID ="+task.getTaskID());
  int numExperts = agent.getIntParam("config","numExperts");
  String chatId = task.getTaskID();
  for(int i=0 ; i<numExperts ; i++){
    String expert = agent.getStringParam("experts",i);
    if(agent.getStringParam(expert,"chatId").equals(chatId)){
      agent.setParam(expert,"available","true");
      agent.setParam(expert,"chatId","");
      agent.info(expert + " disponible");
      return;
    }
  }
end
```

6.4.5. Regla de mensaje HTTP de conversación

Código 6.5

Cuando una conversación se ha iniciado, desde el cliente web llegan mensajes a través de HTTP y se reciben al dispararse esta regla. Estos mensajes incluyen el identificador de sesión que se creó al iniciar la conversación. De esta forma se sabe a qué experto corresponde y el proceso consiste en redirigir el mensaje del usuario hasta el experto, actualizar el historial de la conversación y renovar el temporizador de fin de conversación.

Código 6.5: Regla de mensaje HTTP de chat

```
rule "HTTP Chat Message"
when
  http:OnHttpGet()
  agent:SlaaAgentSbb()
  eval(http.getParam("type").equals("chat"))
then
  String log = "TYPE:" + http.getParam("type") + " ID:" + http.getParam("id")
       + " MSG:" + http.getParam("msg");
  String chatId = http.getParam("id");
  int numExperts = agent.getIntParam("config","numExperts");
  for(int i=0 ; i<numExperts ; i++){
    String expert = agent.getStringParam("experts",i);
    if(agent.getStringParam(expert,"chatId").equals(chatId)){
      agent.sendXmppMessage(expert,http.getParam("msg"));
      String tmp = agent.getStringParam("chats",http.getParam("id"));
      tmp += "U+FFFD" + http.getParam("msg") + "\n";
      agent.setParam("chats",http.getParam("id"),tmp);
      agent.delTask(chatId);
```

```
        agent.setTask(chatId,1,60000);
        return;
      }
    }
  end
```

6.4.6. Regla de mensaje XMPP de conversación

Código 6.6

A través de esta vía se reciben los mensajes de los expertos. Si el experto tiene una conversación en curso con un usuario, se añadirá el mensaje al historial de conversación para ser mostrado en el cliente web y se renovará el temporizador de fin de sesión. En el caso de no tener ninguna conversación en curso, se informará al experto de esta situación.

Código 6.6: Regla de mensaje XMPP de chat

```
rule "XMPP Message"
when
  xmpp:OnXmppMessage()
  agent:SlaaAgentSbb()
then
  String chatId = agent.getStringParam(xmpp.getFrom(),"chatId");
  String name = agent.getStringParam(xmpp.getFrom(),"name");
  if(!chatId.equals("")){
    String tmp = agent.getStringParam("chats",chatId);
    tmp += name+": " + xmpp.getMessage() + "\n";
    agent.setParam("chats",chatId,tmp);
    agent.delTask(chatId);
    agent.setTask(chatId,1,60000);
  }else{
    agent.sendXmppMessage(xmpp.getFrom(),"No hay conversaciones abiertas.");
  }
end
```

6.4.7. Regla de mensaje XMPP de presencia

Código 6.7

Es necesario que si un experto no quiere ser molestado no le llegue ningún mensaje de los usuarios. Por ello, cuando un experto cambia su estado en su cliente de mensajería instantánea, este hecho queda registrado en su configuración para que no se le asigne ninguna

conversación hasta que vuelva a estar disponible. Esta regla se dispara cada vez que un experto cambia su estado.

Código 6.7: Regla de mensaje XMPP de presencia

```
rule "XMPP Presence"
when
  xmpp:OnXmppPresence()
  agent:SlaaAgentSbb()
then
  String mode = xmpp.getMode();
  String type = xmpp.getType();
  String chatId = agent.getStringParam(xmpp.getFrom(),"chatId");
  if(mode.equals(type) && chatId.equals("")){
    agent.setParam(xmpp.getFrom(),"available","true");
  }else{
    agent.setParam(xmpp.getFrom(),"available","false");
  }
  agent.info("USER:" + xmpp.getFrom()+" TYPE:" + xmpp.getType()+" MODE:" +
      xmpp.getMode());
end
```

6.4.8. Regla de petición de historial de conversación

Código 6.8

El cliente web actualiza el historial de la conversación tras cada petición. Con esta regla se permite que durante una conversación en curso se pueda mostrar al usuario todos los mensajes que han ido llegando por ambas partes cada vez que el cliente web lo pida. A partir del identificador de conversación incluido en uno de los parámetros, se obtiene el historial de la conversación de la lista de conversaciones y se devuelve en la respuesta al cliente web que hizo la petición.

Terminado este capítulo, se ha mostrado un ejemplo real de uso de la arquitectura. Como se ha podido comprobar, solo ha sido necesario definir unas pocas reglas para poder implementar un servicio completo que hace uso de varios protocolos y los temporizadores. La forma de programar los agentes basados en reglas permite pensar qué acciones tomar ante la llegada de determinados eventos, simplificando la propuesta de soluciones. En el próximo capítulo se mostrará qué conclusiones y posibles mejoras se pueden realizar en base a todo lo visto anteriormente.

Código 6.8: Regla de petición de historial de conversación

```
rule "HTTP Chat Log"
when
  http:OnHttpGet()
  agent:SlaaAgentSbb()
  eval(http.getParam("type").equals("log"))
then
  String chatId = http.getParam("id");
  int numExperts = agent.getIntParam("config","numExperts");
  for(int i=0 ; i<numExperts ; i++){
    String expert = agent.getStringParam("experts",i);
    if(agent.getStringParam(expert,"chatId").equals(chatId)){
      http.setResponse(agent.getStringParam("chats",http.getParam("id")));
      return;
    }
  }
  http.setResponse("false");
end
```

Capítulo 7

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones y resultados obtenidos tras la realización del proyecto. Además se incluyen las posibles mejoras que se han considerado necesarias para poder ampliar las capacidades del desarrollo y obtener un sistema con mejores y nuevas características

7.1. Conclusiones

Se ha visto cómo ha sido posible desarrollar una primera versión funcional de la arquitectura SLAA capaz de gestionar múltiples usuarios y agentes.
El principal objetivo era el de desarrollar una arquitectura para la implementación de agentes orientados a las comunicaciones, y con los beneficios de JAIN SLEE ha sido posible obtener este resultado aprovechando sus capacidades.
Una de las destacables características de esta arquitectura es que puede hacer uso de los componentes que se pueden encontrar por defecto en las últimas versiones de JAIN SLEE de Mobicents. Esto es importante porque ya existen implementaciones de adaptadores de recursos para la mayoría de protocolos, y como se ha podido ver, con una pequeña adaptación son completamente funcionales dentro de la arquitectura de agentes.
Usar un motor de reglas como Drools para el procesado de eventos ha permitido que la lógica de los agentes se programe de forma externa a SLEE y de esta manera abstraer al desarrollador de todo lo que sucede por dentro. Un desarrollador solo tiene que conocer cómo programar reglas en Drools y qué funcionalidades presenta para su desarrollo un agente en la arquitectura SLAA.
Otro de los puntos a destacar es que el desarrollo de agentes está contenido únicamente en su fichero de reglas, es decir, un agente son sus reglas. Esto permite compartir y reutilizar el código de un agente para crear otros nuevos.
La comunicación entre agentes SLAA en una misma plataforma y entre plataformas ha sido posible gracias a los beneficios de JAIN SLEE. Simplemente adaptando la comunicación HTTP cliente y servidor a la arquitectura, ha dado como resultado que un agente pueda comunicarse con otro directamente y de esta forma poder desarrollar servicios multiagente en el que cada uno de ellos puede estar dedicado a una tarea en concreto. También se puede señalar que no solo los agentes se pueden comunicar entre ellos mediante el protocolo HTTP, realmente pueden usar cualquiera de los servicios que permite la comunicación en ambos sentidos simplemente añadiéndose entre ellos como emisor y receptor.
El hacer posible que toda la plataforma se controle a través de un navegador web también ha sido posible gracias a las capacidades de JAIN SLEE en el manejo de protocolos. Usando el gestor de agentes como receptor de peticiones HTTP y generador de respuestas hacia los clientes se ha conseguido una interfaz completamente funcional para el control de toda la plataforma, la programación de agentes y la visualización de toda la información contenida en ella.

7.2. Trabajos futuros

Durante la elaboración del proyecto y tras su finalización se han analizado las posibles mejoras y ampliaciones de cara al futuro. En esta sección se detallan aquellas que se han considerado más relevantes:

- Añadir más vías de comunicación con recursos externos.
  En esta primera versión se han implementado las vías necesarias para poder hacer uso de algunos de los adaptadores de recursos disponibles y de otros nuevos, pero sería interesante el uso de otros como el adaptador de recursos SIP que permite por ejemplo el uso de servicios de VoIP.

- Ampliar las funciones de los agentes.
  Los agentes disponen de diversas funciones o acciones que pueden realizar mediante su interfaz. Se pueden seguir implementando más funciones que permitan que los agentes traten con la información almacenada de forma más eficiente o realicen nuevas acciones para mejorar su comportamiento.

- Mejorar el procesado de eventos.
  Una de las mejoras evidentes es que los agentes deben poder evaluar de mejor forma la información de los eventos y de sus perfiles de datos en las condiciones que disparan las reglas para evitar tener que programar gran parte de la lógica de los agentes en las acciones que se ejecutan tras el disparo de una regla.

- Hacer compatible la arquitectura con otras existentes.
  Sería interesante poder hacer compatible la arquitectura SLAA con otras existentes para ampliar sus características. Aprovechando la filosofía detrás de JAIN SLEE, un adaptador de recursos nuevo podría ser utilizado para la comunicación con diversas plataformas de agentes.

- Extender la funcionalidad de la interfaz web.
  La interfaz web aunque plenamente funcional puede ser mejorada en diversos aspectos.
  Uno de ellos seria la inclusión de un sistema de control de acceso para usuarios.
  Otra mejora interesante sería mejorar la programación de reglas, que en esta versión consiste en edición de texto plano.
  También se podría incluir un repositorio de ficheros de reglas para que los usuarios puedan almacenar y acceder a versiones guardadas por ellos mismos u otros usuarios para cambiar el comportamiento del agente cambiando sus reglas de forma rápida.

Bibliografía

[1] Sun Microsystems, Open Cloud. JAIN SLEE (JSLEE) 1.1 Specification, Final Release, 2008.

[2] The JBoss Community. Drools, JBoss Rules, Version 5.5.0.Final. http://www.jboss.org/drools/.

[3] Charles Forgy. A network match routine for production systems, 1974.

[4] Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence, 19, pp 17–37*, 1982.

[5] Bellifemine, Fabio, Agostino Poggi, and Giovanni Rimassa. JADE: A FIPA-compliant agent framework. *Proceedings of PAAM. Vol. 99. No. 97-108*, 1999.

[6] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review, 10(2):115-152*, 1995.

[7] Michael Luck, Peter McBurney, and Chris Preist. Agent technology: Enabling next generation computing. *Technical report, AgentLink*, 2003.

[8] Douglas Crockford. JSON: JavaScript Object Notation. http://www.json.org/.

[9] Barnard, Eric M. KnockoutJS Starter. *Packt Publishing Ltd*, 2012.

[10] Eduardo Martins. Mobicents JAIN SLEE Development Environment Setup. https://community.jboss.org/wiki/MobicentsJAINSLEEDevelopmentEnvironmentSetup.

[11] Eduardo Martins. Hello SLEE World Service JAIN SLEE Tutorial. https://community.jboss.org/wiki/HelloSLEEWorldServiceJAINSLEETutorial.

Apéndice A

Configuración del servidor y entorno de desarrollo

En este manual se detallan las herramientas necesarias y la configuración para la puesta en funcionamiento del servidor y el entorno de programación de la misma forma que se ha hecho para la elaboración del proyecto. Por un lado se muestra qué es necesario para disponer de un servidor SLAA operativo, y por otro lado cómo configurar el entorno de desarrollo para programar sobre el código fuente de SLAA.

A.1. Repositorio del proyecto

Se puede acceder a la última versión en *https://github.com/neoner2002/slaa-dev*.
En este repositorio se incluyen todas las herramientas y componentes necesarios para la puesta en marcha del servidor o su desarrollo. En el código A.1 se puede ver una muestra del árbol de directorios que contiene el proyecto.
Si solo se quiere acceder a la última versión del servidor SLAA, este se puede encontrar en *https://github.com/neoner2002/slaa*.

Código A.1: Árbol de directorios del proyecto

```
pfc
  mobicents
    ide
      eclipse
      workspace
    slee
      binary
    tools
      apache-ant-1.8.4
      apache-maven-2.2.1
```

A.2. Configuración del sistema operativo

El proyecto se ha elaborado sobre un sistema operativo Ubuntu con nombre de máquina *slaa* y de usuario *fet* con versión 12.10 de 64 bits y la versión 7 de Java de Oracle. Para el correcto funcionamiento es necesario establecer las variables de entorno mostradas en el código A.2.
Estas variables de entorno están pensadas para funcionar teniendo en cuenta que la carpeta *pfc* que contiene todo el proyecto debe estar en */home/fet/Desktop*.

Código A.2: Variables de entorno

```
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")

export MOBICENTS_HOME=/home/fet/Desktop/pfc/mobicents

export ANT_HOME=$MOBICENTS_HOME/tools/apache-ant-1.8.4
export M2_HOME=$MOBICENTS_HOME/tools/apache-maven-2.2.1
export JBOSS_HOME=$MOBICENTS_HOME/slee/binary/2.7.0.FINAL/jboss-5.1.0.GA
```

```
export PATH=$ANT_HOME/bin:$M2_HOME/bin:$JBOSS_HOME/bin:$PATH

export JAVA_OPTS="-Xmn256m -XX:CMSInitiatingOccupancyFraction=80 -Dorg.jboss.
    resolver.warning=true -Dsun.rmi.dgc.client.gcInterval=3600000 -XX:
    MaxPermSize=256M -Xmx7000M -Xms512M -XX:+UseConcMarkSweepGC"
export MAVEN_OPTS=$JAVA_OPTS
```

A.3. Puesta en marcha del servidor

Una vez configurado el sistema operativo y situado el proyecto en la ruta indicada, ya se puede acceder al servidor para ponerlo en marcha. Para ello accedemos a la siguiente ruta: */pfc/mobicents/slee/binary/2.7.0.FINAL/jboss-5.1.0.GA/bin*
Una vez en ella ejecutamos la órden: *./run.sh*
Cuando el servidor se ha iniciado correctamente ya se puede acceder a la interfaz web de administración a través de la siguiente ruta desde un navegador web:
http://localhost:8080/slaa
Con todo esto ya se puede tener acceso a un servidor con la arquitectura SLAA en funcionamiento en la que comenzar a crear usuarios y programar agentes.

A.4. Desarrollo en Eclipse

En el proyecto se incluye la versión *Juno* del entorno de desarrollo Eclipse configurada con el *plugin* EclipSLEE [10] que permite el desarrollo de componentes en JAIN SLEE de forma más rápida [11].
Todos los proyectos Eclipse con el código de la arquitectura SLAA están en la ruta:
/pfc/mobicents/ide/workspace

A.5. Desplegar y compilar componentes

En esta sección se muestra cómo se despliegan los componentes necesarios en el servidor JAIN SLEE. Se distinguen por un lado los que ya se disponen en el servidor de Mobicents y por otro lado los que se han desarrollado para SLAA y que es necesario compilar si se han modificado.

A.5.1. Componentes propios de Mobicents

Estos componentes se despliegan con la órden *ant* o se repliegan con la órden *ant undeploy*. Por ello ha incluido la herramienta ANT en el repositorio del proyecto.

Adaptador de recursos HTTP servidor

/pfc/mobicents/slee/binary/2.7.0.FINAL/resources/http-servlet

Adaptador de recursos HTTP cliente

/pfc/mobicents/slee/binary/2.7.0.FINAL/resources/http-client

Adaptador de recursos XMPP

/pfc/mobicents/slee/binary/2.7.0.FINAL/resources/xmpp

A.5.2. Componentes desarrollados para SLAA

Estos componentes se compilan y despliegan con la órden *mvn install* o se repliegan con la órden *mvn clean*

slaa-socialNetwork-ra

/pfc/mobicents/ide/workspace/slaa-socialNetwork-ra
Contiene el código de la especificación de perfil, los eventos y el servicio agente y gestor de la arquitectura SLAA.

slaa-drools-ra

/pfc/mobicents/ide/workspace/slaa-drools-ra
Es el código del adaptador de recursos Drools utilizado en SLAA.

slaa-agent-service

/pfc/mobicents/ide/workspace/slaa-agent-service
Este es el adaptador de recursos implementado como ejemplo para hacer uso del servicio Twitter por parte de los agentes.

Con todo lo visto en este apéndice se puede poner en marcha el servidor o el entorno de desarrollo.

Apéndice B

Programación de un agente

Este apéndice tiene como objetivo presentar toda la información necesaria para programar un agente en SLAA. Esta orientado a un usuario que accede al servidor y quiere crear un agente. Se describe en qué consiste la configuración de un agente, los eventos que puede recibir y que dispararán las reglas, y las funciones accesibles desde su interfaz.

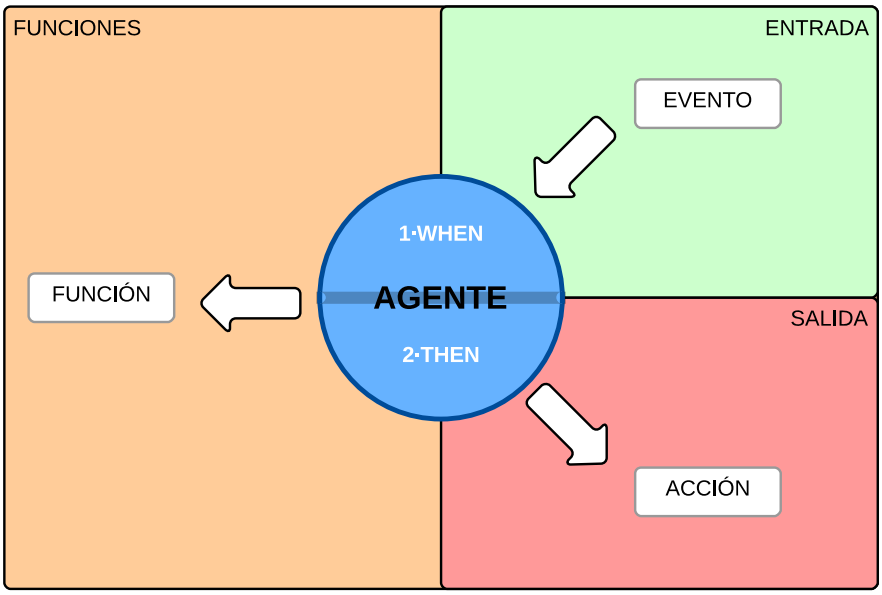

Figura B.1: Esquema general de un agente

B.1. Esquema general de un agente

El esquema de la figura B.1 muestra de una forma directa cómo funciona un agente en la arquitectura SLAA. Básicamente un agente recibirá un evento y realizará una acción y durante este proceso podrá llamar a otras de las funciones que dispone.
Estas funciones pueden ser crear un temporizador, evaluar un parámetro de su memoria u otras que hayan sido definidas.
Los eventos serán sucesos generados por su entorno que ha recogido SLEE y ha entregado al agente para que se dispare alguna de sus reglas.
Finalmente, las acciones serán sucesos que el agente puede generar hacia su entorno tras el disparo de alguna de sus reglas.

B.2. Ciclo de vida

Un agente dentro de la arquitectura SLAA primero tiene que ser creado por un usuario, luego configurado para que pueda hacer uso de los diferentes servicios y finalmente programado para que muestre un comportamiento ante la llegada de eventos.
Una vez todo esto se ha completado, se puede iniciar el agente en la plataforma. Al ser iniciado, SLAA se encarga de aplicar la configuración del agente para iniciar todos los servicios que se quieren utilizar, después por cada evento recibido por la entidad SBB del agente se recibirá alguno de los eventos mostrados en los siguientes apartados, que serán los que

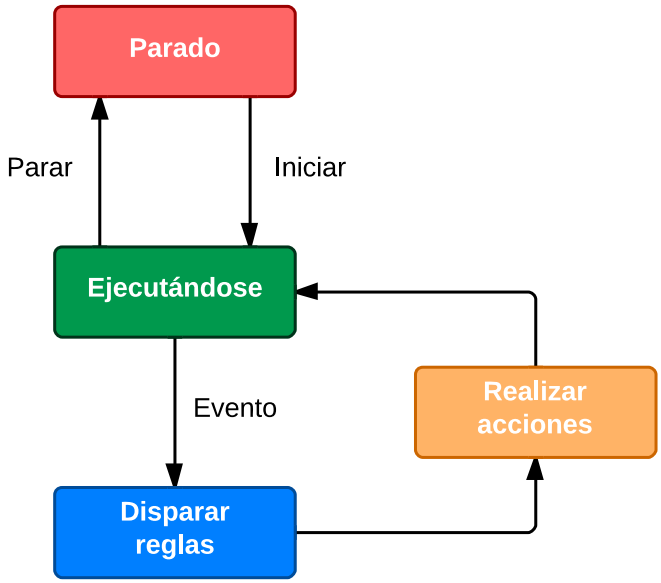

Figura B.2: Ciclo de vida de un agente

dispararán determinadas reglas en las que se ejecutará alguna acción.
En la figura B.2 se puede ver un esquema del ciclo de vida de cualquier agente funcionando dentro de la arquitectura SLAA.

B.3. Configuración

La plataforma permite que un usuario pueda configurar sus agentes desde la interfaz web. La configuración consiste en determinar los parámetros requeridos para que se pueda hacer uso de los servicios disponibles. Por ejemplo, para usar el servicio XMPP, el usuario debe añadir la información de usuario y contraseña adecuada para que el agente pueda iniciar una conexión con un servidor en el que pueda conectarse como un usuario más para intercambiar mensajes con otros usuarios.
Toda la configuración de un agente queda encapsulada en formato de datos JSON como puede verse en el código de ejemplo B.1 y es enviada al servidor para que la procese cuando sea necesario.

Código B.1: JSON de configuración de un agente

```
agentConfiguration = {
  "name": "agent1",
  "user": "user1",
  "services": [
    {
      "name": "XMPP",
      "isActive": false,
      "fields": [
        {
          "name": "xmppuser",
          "field": "Usuario XMPP",
          "type": "input",
          "value": ""
        },
        {
          "name": "xmpppass",
          "field": "Clave XMPP",
          "type": "password",
          "value": ""
        }
      ]
    },
    {
      "name": "HTTP server",
      "isActive": false,
      "fields": [
  {
          "name": "httpserverurl",
          "field": "Http server URL",
          "type": "view",
          "value": endPoint,
        }
      ]
    }
  ]
}
```

B.4. Eventos recibidos por el agente

Las reglas que se definen en los agentes se disparan al recibir alguno de los eventos disponibles en la plataforma. Estos eventos corresponden a las diferentes funcionalidades y servicios para los que se ha programado la arquitectura. Cuando la entidad SBB del agente recibe un evento, este es *mapeado* en otro como se detalló en capítulos anteriores con la información relevante accesible de forma directa. De esta manera, por ejemplo se puede evaluar un parámetro del evento para decidir si disparar una regla o no.

B.4.1. OnAgentStarted

Este evento lo recibe un agente cuando se acaba de iniciar. Sirve para que se programen determinadas acciones al iniciarse el agente.

B.4.2. OnAgentStopped

De forma análoga al anterior, este evento se recibe cuando un agente va a ser parado. Se puede aprovechar este evento para realizar alguna acción antes de que el agente se pare en la plataforma.

B.4.3. OnXmppMessage

Este evento se recibe cuando un agente recibe un mensaje XMPP de cualquier usuario de mensajería instantánea de su lista de contactos.

getFrom()

Con esta función podemos obtener el emisor del mensaje.

getMessage()

Llamando a este método se obtiene el contenido del mensaje que escribió el emisor.

B.4.4. OnXmppPresence

Un mensaje de tipo presencia es habitual en mensajería instantánea. Cuando un usuario cambia su estado, este se recibe en forma de evento por el agente.

getFrom()

Con esta función podemos saber el emisor del mensaje de presencia.

getType()

Este parámetro nos indica si un usuario está conectado (*available*) o desconectado (*unavailable*).

getMode()

Obteniendo este parámetro podemos saber el estado concreto del usuario: (*available*, *away*, *dnd*).

B.4.5. OnHttpGet

Cuando un agente recibe una petición HTTP GET de un recurso externo, esta es recibida como un evento de este tipo por el agente.

getParam(paramName)

Se puede obtener cualquier parámetro de la petición a partir de su nombre.

setResponse(response)

Si se quiere se puede añadir una respuesta para que sea recibida por el emisor.

B.4.6. OnHttpResponse

Cuando un agente envía una petición HTTP GET, recibirá una respuesta. La respuesta llegará en este evento.

getResponse()

Del evento se puede obtener la respuesta en forma de cadena de texto.

getUri()

También se puede extraer la URI de la respuesta del mensaje HTTP.

getStatusCode()

Además se puede extraer el código de estado, que permite saber información acerca de la respuesta. Por ejemplo si ha habido algún error.

B.4.7. OnTask

Los agentes pueden programar temporizadores. Cada temporizador generará este tipo de eventos dependiendo de su configuración.

getTaskID()

Con este método podemos saber a cual de los temporizadores configurados corresponde el evento.

B.5. Interfaz del agente

En esta sección se definen las acciones que puede ejecutar un agente.

B.5.1. Perfiles de datos

Estas funciones permiten crear o eliminar perfiles de datos para almacenar información persistente.

B.5.1.1. setProfile(profileName)

Con este método el agente crea un perfil de datos con un identificador propio.

B.5.1.2. delProfile(profileName)

Si es necesario elimiar un perfil de datos, se puede hacer llamando a este método con su identificador.

B.5.1.3. delAllProfiles()

Puede ser necesario eliminar todos los perfiles de datos en un agente. Con este método queda limpia la memoria persistente de un agente.

B.5.2. Parámetros y valores

Cuando existe un perfil de datos, en él se pueden almacenar datos en forma de nombre de parámetro y valor.

B.5.2.1. setParam(profileName, paramName, paramValue)

De esta forma se almacena un parámetro y su valor dentro de un perfil de datos dado.

B.5.2.2. delParam(profileName, paramName)

Cuando es necesario elminar un parámetro se hace con este método.

B.5.2.3. getStringParam(tableName, paramName)

Esta es la forma estándar de obtener el valor de un parámetro. Siempre se guardan como cadena de texto *String*.

B.5.2.4. getIntParam(tableName, paramName)

Con este método obtenemos el valor de un parámetro en forma de número entero. A veces es necesario obtener el valor de un parámetro con un tipo en concreto, por ello se han definido estas formas de obtener el valor de un parámetro.

B.5.2.5. getLongParam(tableName, paramName)

Otro de los tipos que puede interesar para obtener el valor de un parámetro es *Long*.

B.5.3. Servicios

Estas funciones permiten que el agente envíe datos a los recursos externos a través de alguno de los servicios disponibles.

B.5.3.1. sendXmppMessage(user, message)

Llamando a este método, el agente envía un mensaje XMPP a un usuario concreto. Para ello tiene que estar configurado correctamente para usar este servicio.

B.5.3.2. sendHttpGet(url)

Esto permite enviar mensajes HTTP a un recurso externo, en la url tiene que estar indicado el destinatario, y si es necesario los parámetros con datos. Con esta función se puede enviar un mensaje a cualquier otro agente SLAA porque todos los agentes pueden usar comunicación HTTP por defecto.

B.5.3.3. sendTweet(message)

Otros de los servicios disponibles es *Twitter*. Para enviar un *tweet* solo es necesario llamar a esta función con el mensaje, y que el agente esté correctamente configurado para usar el servicio.

B.5.4. Temporizadores

Para crear o eliminar temporizadores se pueden utilizar estos métodos.

B.5.4.1. setTask(taskID, repetitions, duration)

Un temporizador se crea llamando a este método. Es necesario indicar un identificador, el número de repeticiones (0 para repeticiones indefinidas), y la duración en milisegundos entre repeticiones.

B.5.4.2. delTask(taskID)

Si queremos que un temporizador configurado deje de enviar eventos y se elimine, es necesario llamar a este método con el identificador del temporizador.

B.5.4.3. delAllTasks()

Esta función sirve para eliminar todos los temporizadores configurados por el agente.

B.5.5. Trazas

Otra de las funciones que puede usar un agente es crear mensajes de trazas que pueden ser visualizados a través de la interfaz web del agente.

B.5.5.1. info(message)

Esta función se utiliza para generar un mensaje de información que aparecerá en las trazas.

B.5.5.2. error(message)

Cuando en mensaje sea de tipo error, se puede llamar a este método para generarlo.

B.6. Programación de un agente

Código B.2: agentRulesExample.drl

```
import es.upm.dit.gsi.slaa.services.SlaaAgentSbb;
import es.upm.dit.gsi.slaa.facts.OnAgentStarted;

rule "Hello world"
when
  started:OnAgentStarted()
  agent:SlaaAgentSbb()
then
  agent.info("Hello world!");
end
```

La programación de un agente consiste en modificar su fichero de reglas. Una vez que un agente es creado y configurado se le asigna automáticamente un fichero de reglas en el servidor que puede ser modificado directamente desde la interfaz web. El comportamiento de un agente se puede modificar mientras está en ejecución, los cambios sucederán ante la siguiente llegada de un evento.
El esquema que sigue la programación de un agente consiste siempre en definir una regla ante la llegada de un evento que se disparará en determinadas condiciones, y posteriormente realizar una acción.
En el código B.2 se muestra un ejemplo en el que un agente genera un mensaje de trazas desde su interfaz al recibir un evento que indica que el agente acaba de ser iniciado por el gestor de agentes.

Todo lo que se ha mostrado en este apéndice es lo necesario para que un usuario que quiera hacer uso de la arquitectura SLAA pueda programar sus agentes. Como se ha podido comprobar, la programación de un agente consiste en determinar las condiciones de las reglas que se dispararán ante la llegada de eventos y el valor de algún parámetro (si es necesario), y una vez que se dispara la regla, el código de la condición determinará qué acciones realizará el agente debido a la llegada de ese evento. Estas acciones pueden ser usar una de las funciones definidas en la arquitectura o generar una respuesta hacia el exterior.

Glosario de términos

ANT Apache Ant. Es una herramienta usada en programación para la realización de tareas mecánicas, normalmente durante la fase de compilación y construcción. Es un software para procesos de automatización en lenguaje Java.

API Application Programming Interface. Es el conjunto de funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software.

Drools Es un sistema de gestión de reglas de negocio con un motor basado en inferencia usando una implementación del algoritmo Rete.

Eclipse Es un programa compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollo de aplicaciones.

EclipSLEE Es un *plugin* para Eclipse de desarrollo en JAIN SLEE para la ayuda en la creación de componentes.

Gtalk Google Talk es un servicio de mensajería instantánea que provee comunicaciones de texto y voz.

HTML HyperText Markup Language. Es un lenguaje de marcado para la creación de páginas web y otra información que pueda ser mostrada por un navegador web.

HTTP HyperText Transfer Protocol. Es un protocolo de aplicación orientado a transacciones usado en comunicaciones que sigue el esquema de petición-respuesta entre un cliente y un servidor.

Inferencia Es la operación lógica utilizada en los motores de inferencia de los sistemas expertos.

J2EE Es una plataforma de programación definida por una especificación para desarrollar y ejecutar software de aplicaciones en lenguaje Java.

JADE Java Agent DEvelopment Framework. Es una plataforma para el desarrollo de sistemas multiagente implementada en Java.

JAIN Java APIs for Integrated Networks. Es una actividad dentro del *Java Community Process* del desarrollo de APIs para la creación de servicios de telefonía (voz y datos). La actividad JAIN consiste en una serie de "Grupos de Expertos", cada uno desarrollando una única especificación API.

JavaScript Es un lenguaje de programación interpretado utilizado principalmente en el lado del cliente en desarrollo web.

JBoss JavaBeans Open Source Software Application Server. Es un servidor de aplicaciones Java EE de código abierto implementado en Java. Este término también puede hacer referencia a otros elementos.

JSON JavaScript Object Notation. Es un formato ligero de intercambio de datos.

MAVEN Es una herramienta de software para la gestión y construcción de proyectos Java. Es similar en funcionalidad a Apache Ant pero tiene un modelo de configuración de construcción más simple. Una característica clave de Maven es que está listo para usar en red, puede dinámicamente descargar plugins de un repositorio.

Ontología Hace referencia a la formulación de un esquema conceptual con la finalidad de facilitar la comunicación y el intercambio de información entre diferentes sistemas.

OPS5 Es un lenguaje de programación basado en reglas. Uno de los primeros lenguajes usados en sistemas expertos.

Prolog PROgrammation en LOGique. Es un lenguaje para programar mediante lógica usado normalmente en inteligencia artificial.

Rete Es un algoritmo de reconocimiento de patrones usado para implementar un sistema de reglas.

SIP Session Initiation Protocol. Es un protocolo desarrollado con la intención de ser el estándar para la iniciación, modificación y finalización de sesiones interactivas de usuario donde intervienen elementos multimedia.

SLEE Service Logic Execution Environment. Un Entorno de Ejecución de Lógicas de Servicio es un entorno de ejecución de servicios con baja latencia, alto rendimiento y orientación a eventos de forma asíncrona.

URI Uniform Resource Identifier. Es una cadena de caracteres que identifica inequívocamente un recurso normalmente accesible en una red o sistema.

URL Uniform Resource Locator. Es una secuencia de caracteres que se usa para nombrar recursos en Internet para su localización o identificación.

VoIP Voice over IP. Es un grupo de recursos que hacen posible que la señal de voz viaje a través de Internet empleando un protocolo IP.

XMPP Extensible Messaging and Presence Protocol. Es un protocolo abierto y extensible basado en XML, originalmente ideado para mensajería instantánea.

El ingeniero Felipe Echanique Torres,

__ de _____de 2013