UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Servicios distribuidos con mensajería avanzada en la plataforma SPADE

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

AUTOR/A: García Sastre, Nicolás

Tutor/a: Julian Inglada, Vicente Javier

Cotutor/a: Palanca Cámara, Javier

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Universitat Politècnica de València

Departamento de Sistemas Informáticos y Computación

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Trabajo Fin de Máster

Servicios distribuidos con mensajería avanzada en la plataforma SPADE

Autor(a): García Sastre, Nicolás
Director(a): Julian Inglada, Vicente Javier
Cotutor(a): Palanca Camara, Javier

Valencia, Diciembre 2022

Este Trabajo Fin de Máster se ha depositado en el Departamento de Sistemas Informàticos y Computación de la Universitat Politècnica de València para su defensa.

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Título: Servicios distribuidos con mensajería avanzada en la plataforma SPADE

Diciembre 2022

Autor(a):	García Sastre, Nicolás
Director(a):	Julian Inglada, Vicente Javier Departamento de Sistemas Informáticos y Computación Universitat Politècnica de València
Co-director(a):	Palanca Camara, Javier Departamento de Sistemas Informáticos y Computación Universitat Politècnica de València

Resum

Aquest treball aborda la possibilitat d'incorporar crides a procediments remots a través del protocol Jabber-RPC de XMPP dins de la plataforma de sistemes multiagent SPADE (Smart Python Agent Development Environment) mitjançant un complement. Aquest complement obre les portes a una gran quantitat de possibilitats, ja que ara la plataforma està preparada perquè els agents puguin executar serveis que ofereixin altres agents.

Com a cas d'estudi es desenvolupa una mixtura d'experts, en què un agent executa prediccions que ofereixen agents experts prèviament entrenats. També es desenvolupa un agent directori, que actua com a repertori de serveis que ofereixen els agents dins del sistema.

Paraules clau: Jabber-RPC, XMPP, Sistemes Multiagent, SPADE, Mixtura d'Experts, Agent Directori.

Resumen

Este trabajo aborda la posibilidad de incorporar llamadas a procedimientos remotos a través del protocolo Jabber-RPC de XMPP dentro de la plataforma de sistemas multiagente SPADE (Smart Python Agent Development Environment), a través de un complemento. Este complemento abre las puertas a una gran cantidad de posibilidades, ya que ahora la plataforma está preparada para que los agentes puedan ejecutar servicios que ofrezcan otros agentes.

Como caso de estudio se desarrolla una mixtura de expertos, en la que un agente ejecuta predicciones que ofrecen agentes expertos que han sido previamente entrenados. También se desarrolla un agente directorio, que actúa como repertorio de servicios que ofrecen los agentes dentro del sistema.

Palabras clave: Jabber-RPC, XMPP, Sistemas Multiagente, SPADE, Mixtura de Expertos, Agente Directorio.

Abstract

This work addresses the possibility of incorporating calls to remote procedures through the Jabber-RPC protocol of XMPP within the multi-agent system platform SPADE (Smart Python Agent Development Environment), through a plugin. This plugin opens the door to a great number of possibilities, since the platform is now prepared for agents to execute services offered by other agents.

As a case study, a mixture of experts is developed, in which an agent executes predictions offered by expert agents that have been previously trained. A directory agent is also developed, which acts as a repertoire of services offered by agents within the system.

Keywords: Jabber-RPC, XMPP, Multiagent Systems, SPADE, Mixture of Experts, Agent Directory.

Tabla de contenidos

Capítulo 1

Introducción

Los sistemas multiagente son una clase de sistemas de computación distribuida en la que los componentes interactúan entre sí de forma autónoma y cooperativa. Estos sistemas están formados por un conjunto de agentes inteligentes que pueden interactuar entre sí y con el entorno en el que se encuentran.

Estos sistemas se presentan a través de agentes, que se tratan de sistemas autónomos y distribuidos, que cooperan entre si para alcanzar los objetivos propuestos. La comunicación entre agentes debe ser lo más fluida y estándar posible, dado que esto permite una mayor integración con otros sistemas o elementos externos al mismo. Tanto si hay que coordinar drones, sistemas IoT (Internet of Things), o bien crear simulaciones de plagas o de tráfico, el enfoque multiagente posibilita la concepción de soluciones complejas que requieren la interacción de varias entidades.

Estos sistemas se presentan por medio de agentes autónomos distribuidos cuya cooperación es indispensable para alcanzar los objetivos del sistema. Una característica fundamental de los SMA reside en la capacidad de los agentes de reorganizarse de forma dinámica con el fin de adaptarse a variaciones del entorno. Resulta lógico pensar que la forma más natural a la hora de modelar un sistema complejo sea hacerlo a partir de una serie de componentes autónomos capaces de actuar e interrelacionarse con flexibilidad para lograr sus objetivos, y también es cierto que los agentes constituyen una buena abstracción para el modelado de sistemas compuestos de muchos subsistemas, componentes y relaciones entre ellos

Así mismo, la estandarización en la Informática es el proceso mediante el cual se establecen y se aplican normas y especificaciones técnicas para los sistemas y componentes de computación. Esto permite que los diferentes sistemas y componentes sean compatibles entre sí y se puedan intercambiar de manera eficiente.

Existen grandes entidades en la actualidad que se preocupan por la estandarización de los sistemas informáticos. El mayor ejemplo es IEEE. Algunos ejemplos de estandarización presentes en la actualidad son las normas *IEE 802.11* (estándar de LAN inalámbrica WIFI), o como caso fundamental para este trabajo, los estándares basados en XML como el *Protocolo extensible de mensajería y comunicación de presencia (XMPP)*

Es por esto por lo que la estandarización dentro de los sistemas multiagente es esencial, dado que es necesario que todos los agentes puedan cooperar entre si y realicen

una comunicación efectiva.

Este trabajo trata de implementar llamadas a procedimientos remotos en otros agentes en *SPADE (Smart Python Agent Deelopment Environment)*. Esto permitirá que de forma estándar, un agente pueda llamar a otro agente para que dicho agente ejecute un método previamente definido. SPADE es una plataforma de sistemas multiagente desarrollada en Python que emplea como protocolo de comunicación XMPP. De esta forma, la llamada a procedimiento remoto puede ser realizada mediante la implementación de la tecnología *RPC* haciendo uso del estándar *XEP-0009* de *XMPP*.

El desarrollo realizado ha sido validado mediante la implementación de dos casos de estudio. Los ejemplos elegidos para llevar a cabo la experimentación son: una mixtura de expertos y un agente directorio.

En la mixtura de expertos, un agente quiere conocer la clasificación de un conjunto de datos, y para ello, realiza la misma llamada preguntando por predicciones a diferentes agentes expertos mediante el uso de *Jabber-RPC*. Cada uno de estos agentes realizará su propia predicción de forma interna de una forma diferente, y devolverá el resultado del cliente. El cliente con todos los resultados podrá decidir cual es la respuesta correcta de acuerdo a algún mecanismo de toma de decisiones, como por ejemplo, quedarse con la respuesta dada por la mayoría.

El agente directorio (o AgenteDF) trata de un agente que lista los servicios que ofrecen los agentes dentro de un sistema multiagente, como si se tratara de un directorio de páginas amarillas. Los agentes podrán registrar, desregistrar o consultar al agente directorio para modificar o consultar el directorio de los servicios de los agentes.

1.1. Motivación

SPADE provee una interfaz sencilla para crear agentes. Utiliza un modelo de comunicaciones basado en mensajería instantánea, en concreto en *XMPP*, que intercambia mensajes en tiempo real entre aplicaciones. Como ventajas sobre otras plataformas se puede mencionar su compatibilidad con lenguajes de contenido muy usados en la actualidad como *FIPA* y *KQML*, la posibilidad de usar notificación de presencia, la autenticación de usuario y contraseña para aumentar la seguridad así como una conexión cifrada a la plataforma para garantizar la confidencialidad y la movilidad de los agentes. Estas ventajas han hecho que tenga una gran aceptación en la actualidad y sea muy utilizada por distintos desarrolladores.

Pese a estas cualidades, *SPADE* tiene una limitación, y es que actualmente no da soporte para realizar llamadas a procedimientos remotos a otros agentes. Esta implementación daría paso a que los agentes programados en *SPADE* pudieran realizar llamadas a procedimientos en otros agentes, de forma que estos agentes realicen una función y devuelvan una respuesta de vuelta al agente que realiza la llamada. Esto podría servir, por ejemplo, para que un agente obtenga de otros agentes una predicción de una clasificación, y a partir de las respuestas obtenidas, hacer la elección de aquella predicción más probable sin tener que utilizar una gran capacidad de cómputo en este agente.

Con el desarrollo presentado en este trabajo, se pretende facilitar la comunicación y diseño de las aplicaciones basadas en sistemas multiagente. Esto favorece la integración de este tipo de sistemas, en concreto *SPADE*, en todo tipo de entornos, tanto

simulados como reales.

Además, este desarrollo permitirá que un agente proporcione servicios de forma automática a otros agentes, mediante la ejecución de métodos remotos.

1.2. Objetivos

El objetivo principal es proporcionar a SPADE un método para que los agentes puedan realizar llamadas a procedimientos remotos de otros agentes.

Los objetivos específicos para llevar a cabo dicho objetivo son los siguientes:

- Estudio de plataformas de SMA así como de las tecnologías existentes para poder proporcionar el acceso a ejecución de métodos remotos en una plataforma de agentes.
- Proporcionar a la librería *aioxmpp* (librería que utiliza *SPADE* para el uso de *XMPP*) de una implementación del protocolo estándar *Jabber-RPC* para llamadas a procedimientos remotos a través de XMPP.
- Proporcionar a *SPADE* el plugin necesario para el uso de *Jabber-RPC*, de modo que puedan realizar llamadas a procedimientos remotos entre agentes de forma transparente y sencilla.
- Validar los desarrollos realizados a través de ejemplos de uso del plugin, así como realización de pruebas de ejecución.

1.3. Estructura de la memoria

La memoria de este trabajo se compone de siete capítulos, cuyo contenido se detalla a continuación:

- **Capítulo 1, Introducción:** En este capítulo se expone la motivación del proyecto, así como los objetivos que se quieren cumplir.
- **Capítulo 2, Estado del arte:** En este apartado se realiza un análisis de los sistemas de agentes que existen actualmente. Posteriormente se exponen las herramientas disponibles actualmente para la realización de este proyecto. Finalmente, se explica la propuesta que se hace en este proyecto.
- **Capítulo3, Implementación del protocolo Jabber-RPC:** En este capítulo se explica el funcionamiento del protocolo *Jabber-RPC*, y se desarrolla una ampliación para la librería *aioxmpp*.
- **Capítulo 4, Ampliación de SPADE:** En este capítulo, se explica cómo se realiza la integración de SPADE en RPC.
- **Capítulo 5, Casos de uso:** En este capítulo, se muestran dos casos de uso desarrollados para el plugin de SPADE. Estos dos casos de uso desarrollados son una mixtura de expertos y un agente directorio.
- **Capítulo 6, Resultados:** En este resultado, se exploran los tests unitarios desarrollados, y se muestran los resultados obtenidos con la mixtura de agentes.

- **Capítulo 7, Conclusiones:** En este capítulo se han resumido los temas abarcados en el trabajo, junto a los objetivos cumplidos y su solución. Además, se ha relacionado el proyecto con los estudios cursados y se han propuesto trabajos futuros que se pueden realizar sobre este proyecto.

  Finalmente se incluye una serie de referencias bibliográficas que han sido utilizadas como estudio y referencia para abordar la investigación necesaria para la realización del proyecto.

Capítulo 2

Estado del arte

En este apartado hablaremos sobre el avance de la tecnología de sistemas multiagente y de protocolos de llamadas a procedimientos remotos, imprescindible para la realización de este proyecto, de las herramientas de software necesarias y sobre las aplicaciones relacionadas ya existentes.

Estas tecnologías son aquellas relacionadas con las plataformas de sistemas multiagente, como pueden ser *JADE*, *JACK*, *JASON* o *SPADE*, además de las distintas posibilidades para ejecutar métodos remotos. Por último, se indica qué opción es la más indicada para este trabajo en las conclusiones.

2.1. Plataformas de sistemas multiagente

Hoy en día, el software relacionado con los sistemas multiagente está muy avanzado. Por ende, también se dispone de grandes entornos con muchas funcionalidades que permiten programar de forma sencilla estos sistemas multiagente. A continuación se mencionan algunas de las más conocidas.

SPADE es una plataforma de sistemas multi-agente desarrollada en Python [22] y basada en la tecnología de mensajería instantánea XMPP [19]. Algunas de las características más notables de la plataforma SPADE son: soporte de organizaciones virtuales [3], notificación de presencia, compatible con FIPA [15] e independencia del lenguaje y la plataforma. Fue desarrollada por la Universidad Politécnica de Valencia en 2005 y se distribuye como software libre. Fue desarrollada en el grupo de investigación en el que se realiza este trabajo.

JADE [5], es una plataforma software implementada en Java [4] para el desarrollo de agentes. Esta plataforma soporta la coordinación de múltiples agentes FIPA [15] 2.1 y proporciona una implementación estándar del lenguaje de comunicación FIPA-ACL, que facilita la comunicación entre agentes y permite la detección de servicios que se proporcionan en el sistema. JADE fue desarrollado originalmente por Telecom Italia y se distribuye como software libre.

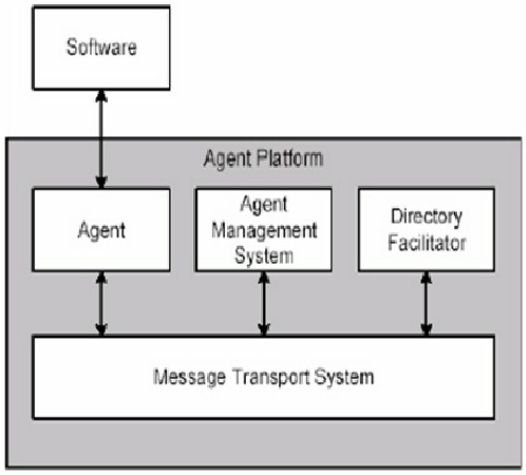

Figura 2.1: Modelo de referencia de la plataforma de agentes FIPA (imagen extraída de [2])

Cabe mencionar también sobre JADE que ha sido una de las plataformas más extendidas y de mas impacto, no obstante, el desarrollo de JADE ha sido discontinuado, por lo que su uso hoy en día no es tan extenso.

JADE no permite hacer uso de RPC. La mejor aproximación para realizar llamadas a procedimientos remotos utilizando JADE es mediante reflexión, creando un servicio para poder invocar un método.

JACK [6] es un entorno multiplataforma para crear, ejecutar e integrar agentes múltiples de nivel comercial. Está construido sobre la lógica del modelo BDI (2.2). Es un entorno desarrollado en su totalidad en Java [4]. En contrapartida con los demás entornos, JACK no se trata de un entorno de código libre.

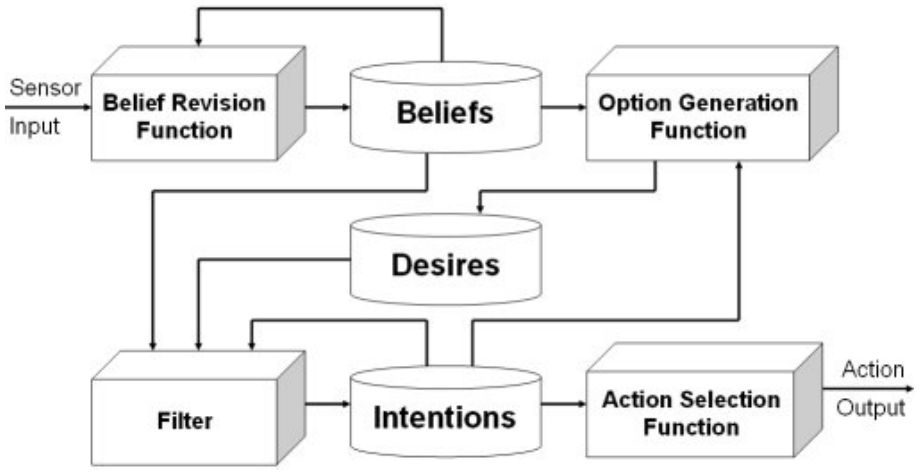

Figura 2.2: El proceso de razonamiento práctico en un agente BDI (imagen extraída de `https://github.com/ingridnunes/bdi4jade`)

MASON [10] es un kit de herramientas de simulación multiagente de eventos discretos desarrollado en Java [4].

MASON fue diseñado para servir como base para una amplia gama de tareas de simulación de múltiples agentes, que van desde robótica de enjambre hasta aprendizaje automático y entornos de complejidad social.

Además, MASON diferencia entre el modelo y la visualización, lo que permite que los modelos puedan separarse y adherirse a los visualizadores. Esto también permite que se cambie de plataforma a mitad de camino.

Por último, MASON se desarrolló con el principio de servir las necesidades de las áreas de investigación. Su filosofía de diseño es desarrollar un modelo rápido y minimalista para que los desarrolladores puedan añadir fácilmente características.

MASON solo está disponible para Windows, lo que implica una desventaja sobre el resto de plataformas.

PADE [12] (Python Agent DEvelopment framework) es un framework para el desarrollo, ejecución y administración de sistemas multi agente en entornos distribuidos. PADE está desarrollado en Python y utiliza las librerías *Twisted* para implementar la comunicación entre los nodos en red.

PADE cumple con el estándar FIPA para que sus agentes puedan interactuar con agentes escritos en otros idiomas y ejecutarse en otras plataformas.

PADE proporciona a los agentes la capacidad de comunicación mediante el uso de protocolos estandarizados como GOOSE (Generic Object Orientated Substation Event protocol), MMS (Manufacturing Message Specification protocol) o CIM (Common Information Model). Estos protocolos son capaces de ejecutarse en redes LAN o a través de internet.

2.1.1. SPADE

A continuación, se va a presentar con más profundidad la plataforma de agentes SPADE, puesto que es una de las más utilizadas en la actualidad, además, su diseño encaja a la perfección con la propuesta de este trabajo.

SPADE es una plataforma de sistemas multiagente muy usada en la actualidad. Brinda varias ventajas ante otras, como ya se ha visto, lo que ha hecho que tenga una gran aceptación a nivel mundial. En esta sección se profundizará en los agentes que se pueden crear con ella y en el funcionamiento de la misma.

El código ejecutado por cada agente SPADE puede estructurarse en comportamientos, por lo que es posible implementar diferentes aspectos de la lógica del agente por separado. Un comportamiento es un proceso que un agente puede ejecutar.

El modelo de agente está básicamente compuesto por un mecanismo de conexión a la plataforma, un dispatcher de mensajes y un conjunto de diferentes comportamientos a los que el dispatcher entrega los mensajes. La arquitectura de SPADE se muestra en la figura 2.3. Para poder conectarse al servidor XMPP, cada agente debe tener un identificador (JID) y una contraseña. El dispatcher de mensajes actúa como un cartero: cuando llega un mensaje para el agente, lo coloca en el "buzón" correcto. Cuando el agente necesita enviar un mensaje, el dispatcher de mensajes lo pone en el flujo de comunicación.

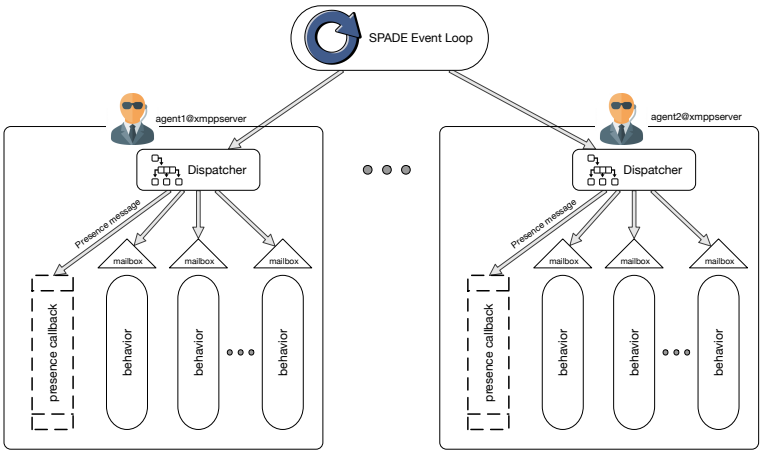

Figura 2.3: Arquitectura de SPADE

Además, SPADE permite a cada agente contar con una interfaz gráfica. Esta interfaz gráfica es útil tanto a la hora de debugear como a la hora de implementar un sistema en producción, dado que es posible crear interfaces personalizadas que muestren información del agente o modifiquen o alteren su comportamiento.

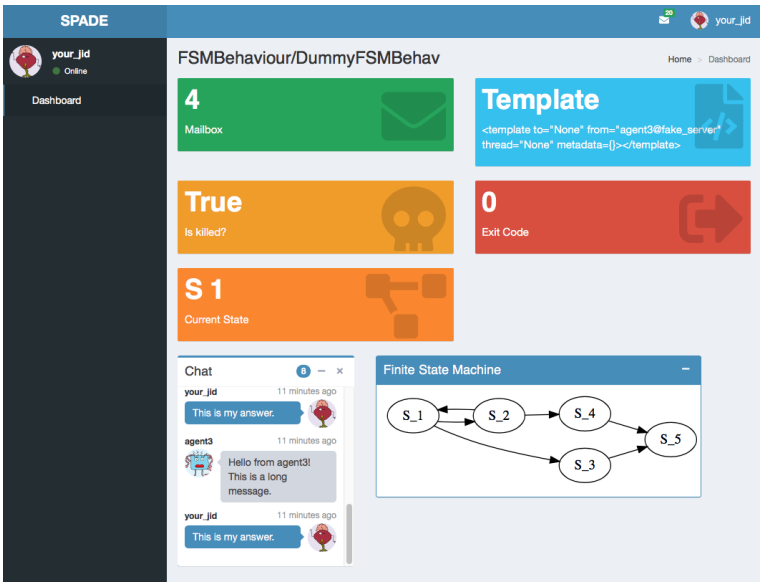

Figura 2.4: Interfaz gráfica web de SPADE

2.1.1.1. Agentes

El modelo de Agente implementado en SPADE se basa en abstracciones y mecanismos sencillos. Cada agente es un objeto dentro del sistema que realiza acciones o trata de alcanzar un objetivo. Para su correcto funcionamiento, los agentes realizan una serie de tareas básicas.

La primera y más esencial de las tareas es la conexión. Para realizar la conexión, los agentes únicamente necesitan un Jabber ID (o JID) y una contraseña válida para realizar la conexión con un servidor XMPP. De esta forma, los agentes se registran

en el servidor XMPP, y no en el ordenador desde el que se han creado, haciendo posible una distribución más transparente, dado que un mismo agente se puede conectar desde diferentes ordenadores con el mismo JID, creando una independencia de ubicación.

En SPADE, los agentes ejecutan sus rutinas en "comportamientos". Los comportamientos son tareas individuales en las que se ejecutan las acciones del agente. Para ello, en SPADE hay implementados una serie de comportamientos: cíclico, one-shot, periódico, time-out y máquina de estado finito.

Por último, SPADE utiliza un despachador de mensajes, donde SPADE asocia cada agente registrado. El despachador de mensajes es, en esencia, un cartero que redirige los mensajes entrantes del agente al comportamiento, o en su defecto, si el agente tiene más de un comportamiento registrado, a todos aquellos comportamientos que esperen el mensaje. Por otro lado, el despachador de mensajes también realiza el comportamiento contrario; transmitir los mensajes que emiten los comportamientos al sistema de comunicación de SPADE.

2.1.1.2. Comportamientos

Los comportamientos son una parte esencial para el funcionamiento de los agentes en SPADE. Como se ha mencionado anteriormente, los comportamientos son tareas individuales en las que se ejecutan las acciones del agente.

Estos comportamientos pueden ser de varios tipos, cada uno de los cuales tiene un patrón de ejecución diferente. Esto es así para poder soportar un requisito de ejecución típico de los agentes en un sistema muti-agente [16]. SPADE tiene implementados cinco tipos de comportamientos base: cíclico, one-shot, periódico, time-out y máquina de estado finito, que se deben utilizar dependiendo de las necesidades de los agentes.

Los comportamientos cíclico y periódico se deben utilizar para taras repetitivas; one-shot y time-out para tareas ocasionales y la máquina de estado finito se debe utilizar para modelar comportamientos que requieran de una mayor complejidad.

2.1.1.3. Despachador de mensajes

SPADE asocia un despachador de mensajes a cada agente que se registre en el sistema. Cuando SPADE recibe un mensaje, el despachador de mensajes redirecciona dicho mensaje a la cola de comportamientos del agente correspondiente. Se decide si un comportamiento debe tener en cuenta el mensaje a través de plantillas asociadas a cada comportamiento.

El despachador de mensajes también se encarga de transmitir los mensajes salientes de los comportamientos al sistema de comunicación de SPADE. Los mensajes son enviados automáticamente cuando llega un nuevo mensaje o está a punto de ser enviado [16].

2.1.2. Artefactos

También es imprescindible destacar que en SPADE existen artefactos. Los artefactos están implementados en SPADE a través de un plugin. Los artefactos son componen-

tes que se encargan de publicar mensajes. Los artefactos, a diferencia de los agentes, no deliberan, es decir, no tienen parte inteligente.

Habitualmente los artefactos publican mensajes enviando a todos los agentes suscritos, que recibirán todas las observaciones que publiquen los artefactos. Para ello utilizan el protocolo PubSub [13]. Este protocolo permite que un cliente envíe mensajes a todos los demás clientes que estén suscritos a dichos mensajes.

SPADE explota este protocolo para conseguir que los agentes puedan suscribirse a las observaciones que pueda publicar un artefacto.

Un agente puede interactuar con un artefacto heredando de la clase *ArtifactMixin*. Esta clase provee las funciones necesarias para que el agente pueda centrarse en las observaciones que publique el artefacto.

2.1.3. Notificaciones de presencia

Como se puede observar en la figura 2.3 sobre la estructura de SPADE, los agentes cuentan también con un manejador para las notificaciones de presencia. Esta característica es heredada de la tecnología *XMPP*.

Las notificaciones de presencia facilitan a los agentes recibir notificaciones en tiempo real sobre el estado de los agentes en su lista de contactos. Además, cada agente cuenta con una propiedad llamada *presence* que implementa todos los métodos y atributos para manejar las notificaciones de presencia de un agente.

Un objeto *presence* cuanta con tres atributos:

- **State**: Este atributo muestra si el agente está o no disponible. Esto realmente indica si el agente está conectado al servidor *XMPP*. Adicionalmente, se pueden dar otros grados de disponibilidad como ocupado, lejos e interesado en recibir mensajes.
- **Status**: Este atributo indica con lenguaje natural el estado de presencia de un agente. Un ejemplo sería ”Ausente”.
- **Priority**: Este atributo indica la prioridad de un estado para el agente. Esto existe debido a que un agente puede tener múltiples conexiones a un mismo servidor *XMPP*. Este atributo mostrará la prioridad para cada conexión.

Gracias a las notificaciones de presencia y a las listas de contactos en los agentes, los agentes pueden tener en cuenta el estado de otros agentes a la hora de enviar mensajes.

2.1.4. BDI

También cabe destacar sobre SPADE que existe un plugin para el uso de la arquitectura *BDI* [18].

La arquitectura *BDI* dota al agente de un conjunto B de creencias, un conjunto D de deseos y un conjunto I de intenciones. El concepto para esta arquitectura es que B contenga los hechos del entorno actual y D el estado que se desea alcanzar. De esta manera, *BDI* modela la forma en la que un agente debe tomar decisiones para lograr llegar al estado que se desea alcanzar.

Para lograr llegar al estado deseado, *BDI* añade las intenciones *I*, que se definen como un conjunto de planes que el agente debe ejecutar para lograr sus objetivos.

2.2. Protocolos de llamadas a procedimientos remotos

A continuación, se van a presentar diversos protocolos que podrían servir para llevar a cabo la finalidad de este trabajo: realizar llamadas a procedimientos remotos en otros agentes.

Existen varias formas de ejecutar procedimientos en máquinas remotas. Dado que SPADE utiliza el protocolo *XMPP*, la forma mas simple sería enviar un mensaje (*XEP-0226* [8]) y que el servidor decodificara y ejecutara el método. Sin embargo, no sería compatible con todos los sistemas, dado que la codificación y decodificación de los mensajes dependería del programador.

Por ejemplo, un programador podría desarrollar una ejecución de métodos en su servidor para los mensajes que comiencen por el token "exec", mientras que otro programador podría hacer lo mismo con el token "execute". Esto haría que aunque los sistemas desarrollados sean similares, no podrían comunicarse entre ellos.

Por consiguiente se han explorado aquellos métodos estándar para la ejecución remota de procedimientos. En *XMPP* existen dos formas de abordar este tipo de llamadas: mediante el *XEP-0050* (*Ad-Hoc* [14]) o mediante el *XEP-0009* (*RPC* [1]).

Estos dos métodos tienen como gran diferencia al anteriormente mencionado (el uso de mensajes mediante el *XEP-0226*) que, al ser estándares, esperan una respuesta del servidor donde, a diferencia del envío de mensajes, es el servidor el que tiene la iniciativa de enviar la respuesta.

Además, al realizar esta estandarización, sería posible conectar este sistema con otros sistemas externos. Esto permitiría a los agentes comunicarse con elementos externos a SPADE, es decir, se podrían hacer llamadas a procedimientos remotos a elementos de sistema que no necesariamente sean agentes. Por ejemplo, se podrían llamar a procedimientos en sistemas AWS que estén a la escucha de mensajes XMPP y que ejecuten y devuelvan la respuesta de la ejecución remota de un procedimiento.

Es necesario mencionar que existen otros protocolos de llamadas a procedimientos remotos que no necesariamente utilizan el estándar XMPP. Ejemplos de esto serían COBRA, SOAP, DCOM o Java RMI.

2.2.1. XMPP

XMPP es un protocolo de presencia y mensajería extensible, un conjunto de tecnologías abiertas de múltiples usos, entre los que se incluyen mensajería instantánea, notificación de presencia, chat de múltiples partes, llamadas de voz y vídeo, colaboración, middleware liviano, redifusión de contenido y enrutamiento generalizado de datos XML.

XMPP se desarrolló originalmente en la comunidad de código abierto de *Jabber* para proporcionar una alternativa abierta y descentralizada a los servicios de mensajería instantánea cerrados en ese momento.

El protocolo *XMPP* es utilizado en una amplia gama de ámbitos: En IoT (*Internet of Things*) para aplicaciones como *Google Cloud Print* o *Firebase Cloud Messaging*, en videojuegos online como *Origin* o *Star Trek Online*, para redes sociales como *BuddyCloud* o *Catpush* o para aplicaciones *WebRTC* que tras la pandemia sucedida han ampliado sus número de usuarios como *Jitsi Meet* o *Otalk*. Incluso aplicaciones de mensajería tan conocidas como *Whatsapp* utilizan una versión de XMPP modificada.

XMPP se basa en los principios de código abierto y de estandarización, de forma que es apropiado para el proyecto que se va a desarrollar.

2.2.2. XEP

XEP significa «XMPP Extension Protocol», en castellano «Propuestas de extensiones XMPP». La fundación de estándares *XMPP* utiliza los *XEP* para desarrollar de forma activa extensiones de *XMPP*.

Estas extensiones deben ser desarrolladas y aceptadas, siguiendo los estándares del *XEP-0001*, llamado «XMPP Extension Protocols», que especifica los tipos de *XEPs* y los protocolos para la creación, publicación y aceptación de los mismos, además de explicar su ciclo de vida, dado que pueden quedar obsoletos.

Sin embargo, las extensiones también pueden ser propuestas por cualquier individuo, proyecto de software u organización. Para mantener la interoperabilidad, las extensiones comunes son administradas por XSF (XMPP Software Foundation). Durante la escritura de este documento, *XMPP* cuenta con 130 *XEPs* activos y en su versión final[1], entre los cuales se incluyen los anteriormente mencionados *Ad-Hoc* y *Jabber-RPC*

2.2.3. Ad-Hoc

El protocolo *Ad-Hoc* (*XEP-0050* [14]) especifica una extensión de *XMPP* que permite que una entidad ejecute comandos en un servidor remoto. También especifica una extensión para listar todos aquellos comandos que se pueden ejecutar en remoto.

Este mecanismo permite que una base grande de entidades Jabber sea partícipe de arquitecturas de aplicaciones más grandes. Aunque en muchos entornos se preferirían los clientes especializados, este protocolo permite que las aplicaciones tengan una audiencia más amplia (es decir, cualquier cliente compatible con Jabber).

Este protocolo, además de ejecutar comandos, permite a los clientes listar los comandos que puede ejecutar el servidor remoto. Este protocolo también permite que los comandos se ejecuten en múltiples fases, es decir, este protocolo admite que una vez comenzada la ejecución del comando, el cliente pueda introducir datos que el programa ejecutado solicite. También permite que el cliente pueda cancelar la ejecución del comando en cualquier momento.

2.2.4. Jabber-RPC

Dentro del marco de la computación distribuida, una llamada a procedimiento remoto (en inglés, Remote Procedure Call, RPC [21]) es cuando un programa de computadora hace que un procedimiento o subrutina se ejecute en un espacio de direcciones

[1] `https://xmpp.org/extensions/`

diferente, comúnmente en otro ordenador en red, que se codifica como si fuera una llamada local, es decir, el programador escribe esencialmente el mismo código tanto si el procedimiento se va a ejecutar en local o en remoto.

El modelo RPC implica un nivel de transparencia de ubicación, es decir, que los procedimientos de llamada son en gran medida los mismos ya sean locales o remotos, pero por lo general no son idénticos, por lo que las llamadas locales se pueden distinguir de las llamadas remotas.

Jabber-RPC (*XEP-0009* [1]) es un método de codificación de solicitudes y respuestas *RPC* en *XML*. Esta codificación permite que el cliente haga llamadas *RPC* a servidores que ofrezcan este servicio, utilizando el estándar *XML-RPC*.

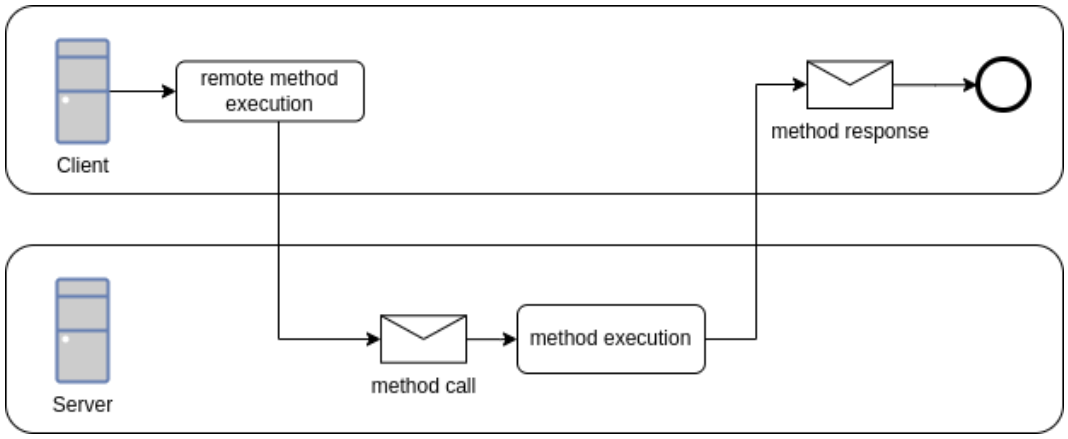

Figura 2.5: Diagrama BPMN del protocolo RPC (elaboración propia)

La figura 2.5 muestra cómo funciona el protocolo RPC. El cliente quiere ejecutar un método remoto, por lo que avisa al servidor enviándole la petición con los parámetros deseados. El servidor ejecuta el método con los parámetros deseados y envía la respuesta de nuevo al cliente.

2.3. Conclusiones

En este capítulo se han analizado distintas plataformas de agentes, como *JADE*, *JACK*, *JASON* o *SPADE*. Además, se han distinguido diferentes métodos de realizar ejecuciones remotas que podrían ser compatibles con estas plataformas. De estos métodos destacamos lo siguiente:

Ad-Hoc permite listar los comandos, cosa que no es posible hacer de forma estándar en *RPC*, aunque sí se puede incluir un método para la retribución de los mismos, no obstante, esto no sería un método estándar.

El protocolo *Jabber-RPC* permite distinguir los parámetros del método al que se llama, dado que los parámetros se incluyen como objetos dentro del mensaje *XML*, no como en *Ad-Hoc*, donde se envía el comando que se quiere ejecutar con los parámetros incluidos en el mismo.

En el protocolo *Jabber-RPC*, las dos entidades entienden completamente el propósito y el comportamiento del comando antes de la ejecución. *Ad-Hoc* está más orienta-

do a la interacción humana, donde el agente del usuario probablemente no tenga conocimiento previo del propósito y el comportamiento del comando.

Es por ello que *Jabber-RPC* se considera una mejor opción para el propósito perseguido, dado que permite una mayor abstracción de forma más estándar.

Capítulo 3

Implementación del protocolo Jabber-RPC

Como se ha explicado anteriormente, el protocolo *Jabber-RPC* es un método de codificación de solicitudes y respuestas para llamadas a procedimientos remotos en *XML*.

El entorno elegido, SPADE, utiliza la librería *aioxmpp*[1] para gestionar el envío y recepción de mensajes. No obstante, *aioxmpp* no tiene implementado el estándar *Jabber-RPC*. Por esto, se debe primero desarrollar el protocolo *Jabber-RPC* en la librería *aioxmpp* para poder realizar llamadas a procedimientos remotos dentro del entorno *SPADE*.

Para la comprensión y desarrollo del protocolo, se debe seguir la propuesta de extensión definida en el *XEP-0009* [1] de *XMPP*. Por ello, durante este capítulo, se explica el protocolo *Jabber-RPC* y posteriormente, su implementación en *aioxmpp*.

3.1. Servicios del XEP-0009

Para su correcto funcionamiento, algunos *XEPs* requieren de otros protocolos adicionales a los del propio *XEP*. Estos protocolos son necesarios para que el cliente o servidor puedan prestar correctamente el servicio.

Concretamente, el *XEP-0009* especifica que son necesarios dos protocolos:

- **XML-RPC:** Es el protocolo propio de *Jabber-RPC*. Especifica los mensajes propios que se realizan para las llamadas a procedimientos remotos.
- **Service discovery:** Es el *XEP-0030*[2]. Este *XEP* se utiliza para saber qué servicios ofrece un cliente *XMPP*. En este caso se utiliza para indicar que el servidor de métodos es capaz de utilizar *Jabber-RPC*.

Dado que *aioxmpp* ya tiene implementado el *XEP-0030*, no será necesario implementar este *XEP*, pero sí habrá que incluirlo en el desarrollo del *XEP-0009*.

[1] `https://github.com/horazont/aioxmpp`
[2] `https://xmpp.org/extensions/xep-0030.html`

3.2. XML-RPC

A continuación, se explica cual debe ser la estructura que deben seguir los mensajes de tipo *RPC*. Esta estructura está especificada según el *XEP-0009* (se ha seguido el estándar descrito en [1]).

XML-RPC explica que el protocolo RPC debe enviar los mensajes dentro de etiquetas "<*iq*>". Las instancias "<*iq*>" son métodos de *XMPP* que se utilizan para solicitar o modificar datos, similares a los métodos *GET* y *POST* de HTTP.

La intención de los métodos viene expresada en un atributo dentro de la instancia "<*iq*>" llamado «*type*», que puede ser «*get*» o «*set*». Adicionalmente, también puede ser «*result*» o «*error*» en las peticiones enviadas como respuesta. A diferencia de otros protocolos como *HTTP* o *SQL*, *XMPP* no incluye el método *DELETE*, también tiene como diferencia que otros métodos como *CREATE* o *UPDATE* están unificados dentro de «*set*».

Cada instancia de "<*iq*>" debe tener un atributo «*id*» que asocie la instancia del envío con la instancia de la respuesta, dado que los clientes de *XMPP* deben siempre dar una respuesta a los mensajes de tipo "<*iq*>" con un mensaje, que tendrá tipo «*result*».

Además, las instancias "<*iq*>" deben contener el *Jabber ID* del cliente al que se desea enviar el mensaje. Un *Jabber ID* o *JID* es un identificador que expresa información de pertenencia o ruta en una red *Jabber/XMPP*. Este identificador permite al servidor *XMPP* identificar a quién tiene que enviar el mensaje.

Además, el protocolo *XEP-0009* indica que el *JID* debe contener el recurso. Esto significa que el *JID* debe tener el formato «id@host/recurso», siendo un ejemplo «jid@localhost/rpc», donde el recurso es *rpc*. Esto es muy importante, dado que un mismo *JID* puede tener varios clientes conectados al mismo servidor gracias al recurso. Por lo tanto, cada cliente podrá ofrecer distintos métodos. Por contra, en el caso de que no se especifique el recurso, los servidores *XMPP* no son capaces de gestionar los mensajes de tipo "<*iq*>".

Así pues, para desarrollar el protocolo *Jabber-RPC*, se deben enviar y recibir mensajes "<*iq*>". Estos mensajes deben contener etiquetas de tipo '<*query*>'. Estas etiquetas de tipo '<*query*>' son en realidad un contenedor para los mensajes *RPC*.

Es común, y así se indica en el *XEP-0099* (*IQ Query* [20]) que las etiquetas '<*query*>' tengan un atributo «*action*» con valor «*read*», «*create*», «*update*» o «*delete*». Esto logra suplir la deficiencia de tipos comparado con otros protocolos comentada anteriormente. Aún así, este atributo queda fuera del dominio *XML-RPC*, pues como es lógico, no es necesario especificar la intención si se llama a un método concreto, por lo que queda fuera de su especificación.

Estos contenedores de tipo '<*query*>' tendrán que contener en su interior o bien etiquetas de tipo "<*methodCall*>" o de tipo "<*methodResponse*>", dependiendo de si se envía la petición de la ejecución del método o la respuesta.

Tanto "<*methodCall*>" como "<*methodResponse*>" pueden contener una lista de parámetros dentro de la etiqueta "<*params*>". Esta lista contendrá la lista de parámetros para la llamada al método en el "<*methodCall*>" o la lista de parámetros de respuesta para el "<*methodResponse*>". "<*methodCall*>" además contendrá el nombre del método que se desea ejecutar dentro de una instancia "<*methodName*>".

Por último, si el cliente solicita un método el cual no tiene permisos para ejecutar, el servidor devolverá la misma petición que recibe de "<*methodCall*>" junto una instancia de error que indique el error «*forbidden*».

Un ejemplo de "<*methodCall*>" es el siguiente:

```
<iq type='set'
    from='requester@company-b.com/jrpc-client'
    to='responder@company-a.com/jrpc-server'
    id='rpc1'>
  <query xmlns='jabber:iq:rpc'>
    <methodCall>
      <methodName>examples.getStateName</methodName>
      <params>
        <param>
          <value><i4>6</i4></value>
        </param>
      </params>
    </methodCall>
  </query>
</iq>
```

Y un ejemplo de "<*methodResponse*>" es el siguiente:

```
<iq type='result'
    from='responder@company-a.com/jrpc-server'
    to='requester@company-b.com/jrpc-client'
    id='rpc1'>
  <query xmlns='jabber:iq:rpc'>
    <methodResponse>
      <params>
        <param>
          <value><string>Colorado</string></value>
        </param>
      </params>
    </methodResponse>
  </query>
</iq>
```

La estructura de etiquetas se ha explicado también en la figura 3.2, donde se muestran todas las etiquetas posibles del protocolo *XEP-0009* y sus relaciones.

3.3. Implementación en *aioxmpp*

Para la implementación en la librería *aioxmpp*, se creó un fork de la librería y una rama en este fork. Se realizó la implementación del protocolo y posteriormente, se creó un pull request para la integración del desarrollo en la librería.

Para la implementación en la librería, es necesario crear un archivo que defina los

XSO (clase que representa las etiquetas XML), otro que defina los servicios necesarios y además, un ejemplo de su uso y tests que cubran el código desarrollado (figura 3.1). En este capítulo, se explica únicamente la implementación del protocolo, es decir, los XSO y los servicios.

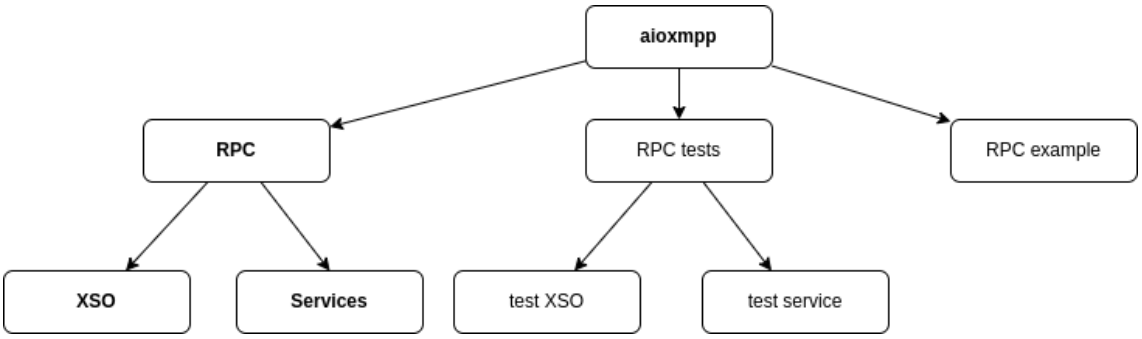

Figura 3.1: Estructura de la implementación (elaboración propia)

3.3.1. XSO

Para el desarrollo de un *XEP* en la librería *aioxmpp*, es necesario tanto especificar las etiquetas que se han mencionado anteriormente como crear el servicio que sea capaz de manejar los mensajes que se envíen y reciban dentro del marco de este protocolo.

Para ello, se crea un archivo llamado *xso.py*, que contiene todos los *XSOs* que serán necesarios para la implementación del servicio. En la librería, un *XSO* es una etiqueta (por ejemplo "<*iq*>" o "<*query*>"). Cada uno de estos *XSO* deben definir qué otras etiquetas o atributos pueden contener.

Los *XSO* creados se han dividido para su mejor comprensión, los *datatypes*, que definen los tipos de datos que se pueden transmitir tanto en la petición de ejecución de un protocolo remoto como en la respuesta, y en los restantes que son necesarios para crear la petición.

Los XSO creados para definir los *datatypes* son, además de *i4, integer, string, double, base64, boolean y datetime*, los siguientes:

- **member:** Esta etiqueta representa un elemento del struct. Esta etiqueta contiene una etiqueta *name* y una etiqueta *value*.
- **name:** Esta etiqueta contiene el nombre del elemento del *struct*.
- **struct:** Esta etiqueta es el equivalente al tipo de dato *dict* en *Python*. Contiene una lista de etiquetas de tipo *member*.
- **data:** Esta etiqueta representa una variable dentro de una etiqueta *array*.
- **array:** Esta etiqueta representa una lista de *Python*. Contiene una lista de etiquetas *data*.

El resto de XSO creados son:

- **query:** Esta etiqueta es el contenedor para el resto del mensaje. Puede contener una etiqueta de tipo *methodCall* o *methodResponse*.

- **methodResponse:** Esta etiqueta se utiliza para devolver la respuesta de la ejecución de un método. Puede contener una etiqueta de tipo *Params*.
- **methodCall:** Esta etiqueta se utiliza para solicitar una ejecución de un procedimiento remoto. Debe contener una etiqueta de tipo *MethodName* y puede contener una etiqueta de tipo *Params*.
- **methodName:** Esta etiqueta contiene el nombre del método a ejecutar.
- **params:** Esta etiqueta contiene una lista de etiquetas de tipo *Param*. Puede estar contenida tanto dentro de *MethodCall* como de *MethodResponse*.
- **param:** Representa un parámetro. Contiene una etiqueta de tipo *Value*.
- **value:** Esta etiqueta representa un valor de un parámetro. Contiene una etiqueta indicando el tipo (i4, string, boolean...) e indica su valor.

Para mejor entendimiento, se ha realizado el diagrama de la figura 3.2. En este diagrama se muestran las relaciones entre los XSO creados.

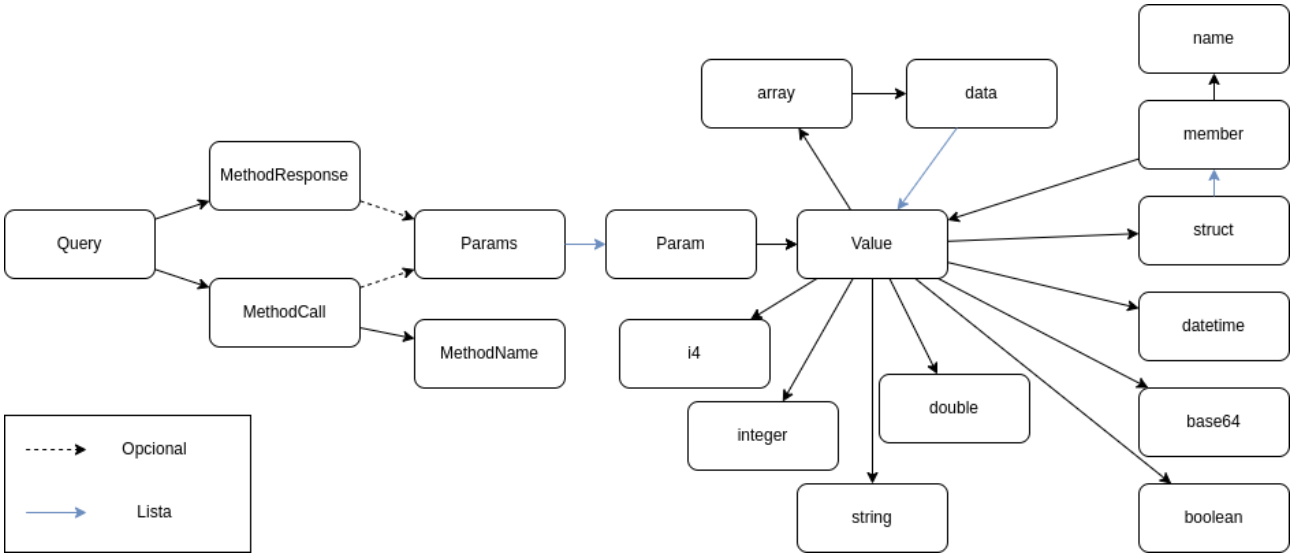

Figura 3.2: Relaciones entre XSO (elaboración propia)

3.3.2. Servicios

Además de las etiquetas, se deben implementar dos servicios: el servicio del cliente y el servicio del servidor.

Estos servicios deben ser capaces de utilizar los XSO implementados para enviar y recibir los mensajes del protocolo que se quiere implementar.

El servicio del cliente debe enviar la petición de la ejecución del método y devolver el resultado de la ejecución remota, mientras que el servidor debe ejecutar dicho método y responder con el resultado de la ejecución.

3.3.2.1. Cliente

El servicio del cliente, para que sea capaz de llevar a cabo las funciones especificadas en el *XEP*, contiene dos funciones: *supports_rpc* y *call_method*. Ambas funciones sirven para suplir todos los casos de uso de la figura 3.3.

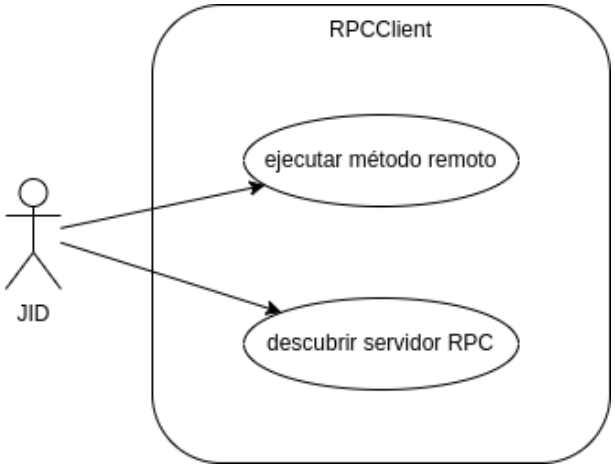

Figura 3.3: Caso de uso de RPCClient (elaboración propia)

La función *supports_rpc* comprueba mediante el protocolo mencionado anteriormente *Service discovery* si el *JID* de entrada es capaz de utilizar *Jabber-RPC*. La función únicamente devuelve un *booleano* indicando si soporta o no dicha función.

La función *call_method* es aquella que se encarga de formar el mensaje en el formato *XML* que se ha explicado anteriormente. Una vez lo envía al *JID* introducido, espera y devuelve la respuesta del servidor.

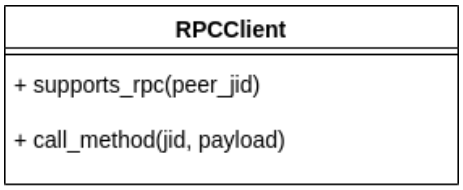

Figura 3.4: Clase RPCCLient (elaboración propia)

3.3.2.2. Servidor

El servicio del servidor contiene tres funciones, de las cuales, el usuario utilizaría dos: *register_method* y *unregister_method*. Además, contiene la función *_handle_method*. Las dos primeras suplen en caso de uso de la figura 3.5, mientras que el tercero sirve para suplir el caso de uso en el que el cliente quiere ejecutar un método de forma remota.

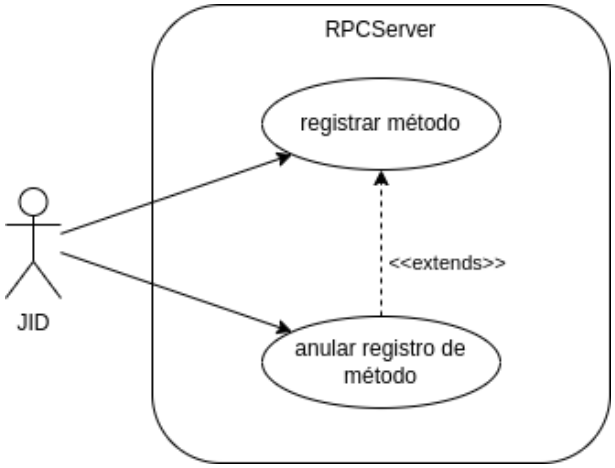

Figura 3.5: Caso de uso de RPCServer (elaboración propia)

Las funciones *register_method* y *unregister_method* se utilizan para que el servidor ofrezca funciones en el servicio *RPC*. Cuando se registra un método, se debe especificar el nombre con el que la función se va a ofrecer, la función que se va a ejecutar y opcionalmente, un método *is_allowed*, cuya respuesta determina si el agente que realiza la solicitud tiene permitida la ejecución del método.

La función *_handle_method* se encarga de comprobar que la función solicitada esté registrada en el servidor. Posteriormente, comprueba con la función *is_allowed* especificada en el registro del método si el agente tiene permitida la ejecución del método. En el caso que el agente no tenga permitida la ejecución, el servidor cancela la ejecución y devuelve un error. En el caso contrario, el servidor ejecutará el método solicitado y devolverá la respuesta si no hay ningún error adicional durante la ejecución del método.

Además, el servidor contiene un *disco_node*, que se encarga de dar servicio al protocolo *Service discovery*. En la clase del servidor, se indica que este servicio tiene la *feature RPC*. Esto se consigue registrando la *feature* "http://jabber.org/protocol/rpc" en el *disco_node*. Así, cuando un cliente utilice el protocolo *Service discovery*, el servidor devolverá una lista de *features* que estén registradas en ese momento, en el caso de ser una instancia de RPCServer, devolverá la *feature* de *RPC*, además de otras en el caso de que se registren.

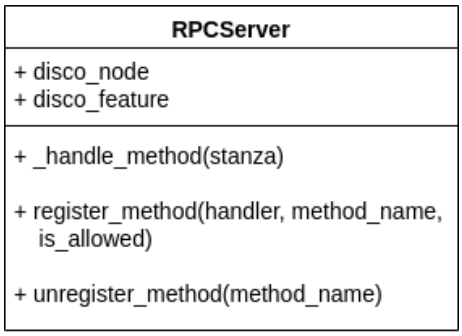

Figura 3.6: Clase RPCServer (elaboración propia)

3.4. Ejemplo de uso

Por último, se ha desarrollado un ejemplo para el uso del protocolo dentro de la librería. Este ejemplo consiste en un servidor de métodos, que ofrece un método llamado "sum". Este método simplemente recibe dos parámetros y los suma. Finalmente, devuelve el resultado y el tiempo que ocupa la operación. Este ejemplo se incluye con la distribución del plugin para facilitar la comprensión de su uso.

El ejemplo tiene como objetivo que otros desarrolladores puedan aprender el funcionamiento del servicio desarrollado. Para ello, se utiliza la llamada a la función *register_method* de *RPCServer*, con la que un JID registra el método "sum". Posteriormente, otro JID hace uso de la función *call_method* de *RPCClient* para obtener el resultado de la suma de dos números, e imprime el resultado obtenido en pantalla.

3.5. Conclusiones

En este capítulo, se ha explicado el funcionamiento del protocolo *Jabber-RPC*. A partir de la información obtenida gracias a esta explicación, ha sido posible desarrollar la ampliación de este protocolo para la librería *aioxmpp*.

Además, se ha desarrollado un ejemplo para el uso del protocolo dentro de la librería. Este ejemplo consiste en un servidor de métodos, que ofrece un método llamado "sum", que al recibir dos parámetros, los suma y devuelve el resultado y el tiempo empleado. Mediante este ejemplo sencillo se pretende mostrar a los desarrolladores que quieran utilizar la librería *aioxmpp* cómo se utiliza el servicio desarrollado en este trabajo.

El desarrollo del protocolo *RPC* dentro de la librería *aioxmpp* permitirá que dentro del entorno de SPADE se pueda utilizar este protocolo. Aunque para ello, hará falta implementar un tipo de agentes capaz de utilizarlo. Este proceso de implementación es el que se describe en el siguiente capítulo.

Capítulo 4

Ampliación de SPADE

En este capítulo, se explica como se realiza la integración de *RPC* en SPADE, utilizando la ampliación realizada en el capítulo anterior, dado que SPADE utiliza la librería *aioxmpp* para la gestión de mensajes mediante el protocolo *XMPP*.

Esta ampliación se ha realizado mediante la implementación de un *plugin* para SPADE. Este plugin consta de una clase que se encarga de gestionar los métodos y clases de *aioxmpp* para *RPC*. El objetivo de este plugin es que los agentes sean capaces de hacer llamadas a procedimientos remotos con el uso de una función.

4.1. Plugins en SPADE

Existen varios plugins desarrollados para SPADE. Los plugins añaden funcionalidades a SPADE y se instalan a través de módulos de *python* que se instalan de forma independiente. Estos módulos se pueden instalar a través de *pip*.

Los plugins son los siguientes:

- **spade_bdi:** Este plugin proporciona la creación de agentes con una capa BDI. Esto facilita que los agentes puedan leer y ejecutar archivos *ASL* escritos en *AgentSpeak*, ademas de añadir *performatives* y añade funciones y acciones personalizadas.
- **spade_pubsub:** Este plugin proporciona el envío de mensajes entre agentes a través del protocolo *pubsub*. Esto implica que los agentes puedan suscribirse a los mensajes que pueda enviar otro agente.
- **spade_artifact:** Este plugin proporciona el uso de artefactos dentro de SPADE. Este plugin contiene tanto la clase artefacto como el *mixin* necesario para la interacción de agentes con artefactos.
- **spade_bokeh:** Este plugin proporciona a SPADE la posibilidad de renderizar diagramas dinámicos de *bokeh* dentro de las plantillas de agetnes de una manera muy fácil.

Para la creación de un plugin se recomienda que el plugin sea creado mediante *Cookiecutter*. *Cookiecutter* es una utilidad para crear proyectos a partir de plantillas.

El plugin de SPADE contendrá en la carpeta raíz una serie de archivos que definen el módulo desarrollado y sus propiedades y autores. Además, una carpeta que contiene el código del propio plugin. Esta carpeta tendrá el nombre "spade_${Nombre del plugin}". También son necesarios otros directorios como *docs* para la documentación, *tests* para los tests unitarios o *examples* para los ejemplos de uso del plugin. Esta estructura se ha plasmado en un diagrama en la figura 4.1.

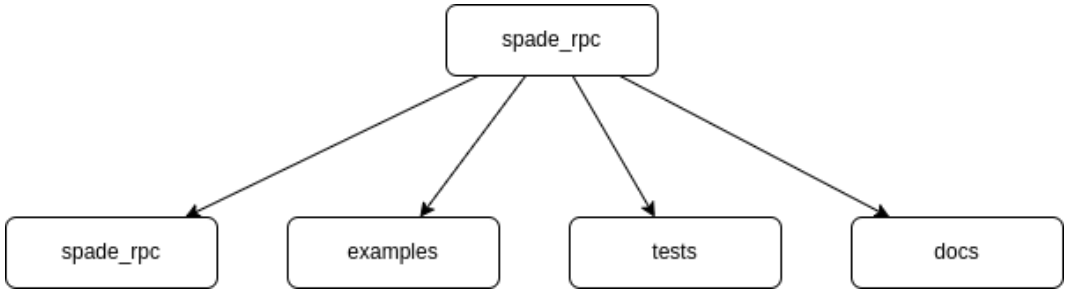

Figura 4.1: Estructura de directorios del plugin creado con *Cookieccutter* (elaboración propia)

4.2. Implementación

Para la implementación, se ha creado un tipo de agente llamado *RPCAgent*. Este agente hereda de la clase *Agent*, por lo que se trata en su esencia de un agente cualquiera de *SPADE* ya que tendrá todas sus funcionalidades heredadas.

Este agente *RPCAgent* al conectarse, creará una instancia de una clase llamada *RPCComponent*. Este componente es el *wrapper* del código desarrollado en la librería, que dota al agente de las funciones necesarias para hacer llamadas a procedimientos remotos. Cada agente será a la vez servidor y cliente de métodos.

La clase *RPCComponent* contiene tres métodos esenciales que utilizarán los agentes para poder llevar a cabo las llamadas de procedimientos remotos. Estos métodos son:

- **call_method**: Esta función se encarga de realizar la llamada a un procedimiento remoto.
- **register_method**: Esta función se encarga de dar servicio a un método para que los demás agentes puedan utilizarlo.
- **unregister_method**: Esta función elimina métodos de la lista de métodos servidos por un agente.

Dado que a la hora de hacer una llamada utilizando la librería es necesario crear un *XSO* que contenga toda la información necesaria, se ha contenido la lógica de la creación en el componente. Por lo tanto, para llamar un método, basta con llamar a la función *call_method* con un *JID* destino, el nombre del método y una lista de parámetros. El componente creado se encarga de transformar los parámetros de la lista de parámetros en parámetros de XMPP.

Puesto que los tipos de datos utilizados en *XMPP* y en *python* no son iguales, se ha creado una equivalencia que está reflejada en la tabla 4.1.

Python	**XMPP**
int	i4
int	integer
str	string
float	double
double	double
bool	bolean
datetime	datetim
list	array
dict	struct

Cuadro 4.1: Equivalencia de tipos de datos entre *XMPP* y *Python* (Elaboración propia)

Dado que en *XMPP* existen dos formas de almacenar datos enteros y en python una, se realiza una equivalencia de dos a uno. Lo mismo sucede para *double* y *float*, dado que ambos existen en *python*, pero en *XMPP* únicamente existe *double*.

Además, dada esta equivalencia, si se encuentra una lista o un diccionario, los elementos de estos se transformarán de forma recursiva.

Esta conversión se realiza tanto a la hora de llamar a un método con *method_call* como a la hora de devolver la respuesta del método en un *method_response*.

Para incluir esta transformación en el *method_response*, se ha hecho que a la hora de llamar a *register_method*, en lugar de registrar el método que se pasa como parámetro, en la librería se registra un método *wrapper* que se encarga de transformar la respuesta del método en un mensaje *methodResponse* de *XMPP*. Este método *wrapper* también se encarga de transformar los paráemtros recibidos en el *method_call* en parámetros de *python*, de forma que el método que el usuario registre será llamado directamente con los parámetros de *python*.

Esto es completamente transparente al usuario, que únicamente tiene que llamar al método *register_method* con dos parámetros: el nombre del método y el método a ejecutar o *handler*.

4.3. Conclusiones

Durante este capítulo se han explicado la implementación del plugin *SPADE_RPC*. Para ello, primero se ha explorado cómo funcionan los plugins en SPADE a través de distintos ejemplos.

Se implementado la posibilidad de crear agentes para SPADE que sean capaces de realizar llamadas a procedimientos remotos. Esta implementación da paso a múltiples escenarios abiertos para la plataforma SPADE. Un ejemplo de ello es la mixtura de expertos que se desarrolla en el siguiente capítulo.

Otro ejemplo propuesto sería un agente directorio. Este agente sería capaz de decir a otros agentes qué agentes ofrecen un servicio. Esto se haría mediante la implementación de un agente que tuviera un directorio. Este directorio se rellenaría cuando un agente haga uso del protocolo *RPC* para registrar un método en el directorio.

Capítulo 5

Casos de Uso: Mixtura de expertos y Agente directorio

En este capítulo, se muestran dos casos de uso desarrollados para el plugin. Estos dos casos son el de una mixtura de expertos y un agente directorio.

En ambos se prepara un sistema de agentes en SPADE que interactúan entre si utilizando *RPC* para hacer llamadas a procedimientos remotos a través del plugin desarrollado.

5.1. Mixturas de expertos

La mixtura de expertos es uno de los métodos de combinación más populares e interesantes, que tiene un gran potencial para mejorar el rendimiento en el aprendizaje automático. La mixtura de expertos se establece sobre el principio de divide y vencerás [11], en el que el espacio del problema se divide entre varios expertos.

La finalidad de esta técnica es adaptar tareas complejas en una división natural de subtareas. Este enfoque subyace en muchos enfoques automatizados para el modelado predictivo, así como para la resolución de problemas en general.

Aunque esta técnica se describió inicialmente utilizando expertos basados en redes neuronales y compuertas, la mixtura se puede generalizar para utilizar modelos de cualquier tipo.

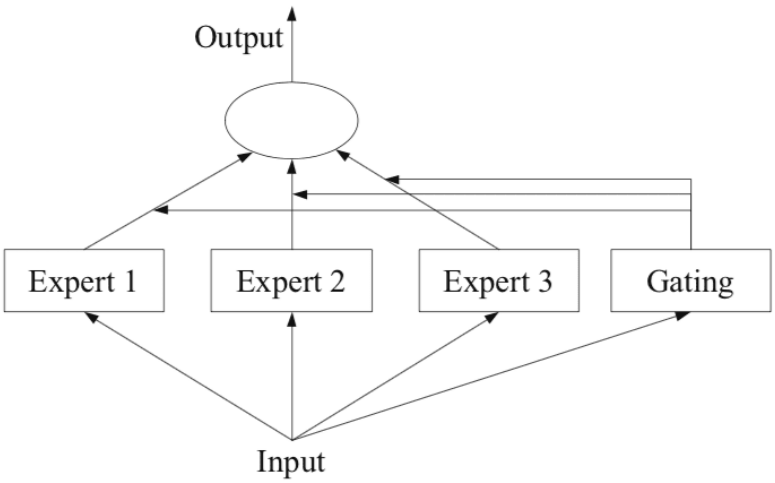

Figura 5.1: Estructura de una Mixtura de Expertos (imagen extraída de [11])

Una posible estructura para una mixtura de expertos sería la de la figura 5.1. En esta estructura, todos los expertos reciben un *input*, del cual hacen una predicción. Con estas predicciones y mediante un sistema de compuertas, se decide cuál es el *output* correcto.

5.2. Desarrollo de una mixtura de expertos en SPADE

Para el desarrollo de una mixtura de expertos en SPADE, se ha requerido del desarrollo previo que dota a SPADE de llamadas a procedimientos remotos en otros agentes, explicado en el capítulo anterior. Esto facilita que un agente pueda preguntar a otros agentes predicciones a través de llamadas a procedimientos remotos.

Se ha dividido el desarrollo en dos tipos de agentes: Los agentes expertos, que se encargan de dado un *input*, predecir un *output*, y el agente al que llamaremos *manager*, el agente que se encarga de enviar los *inputs* entre los agentes y recoger los *outputs* para generar una única predicción. Este proceso se explica visualmente en la figura 5.2

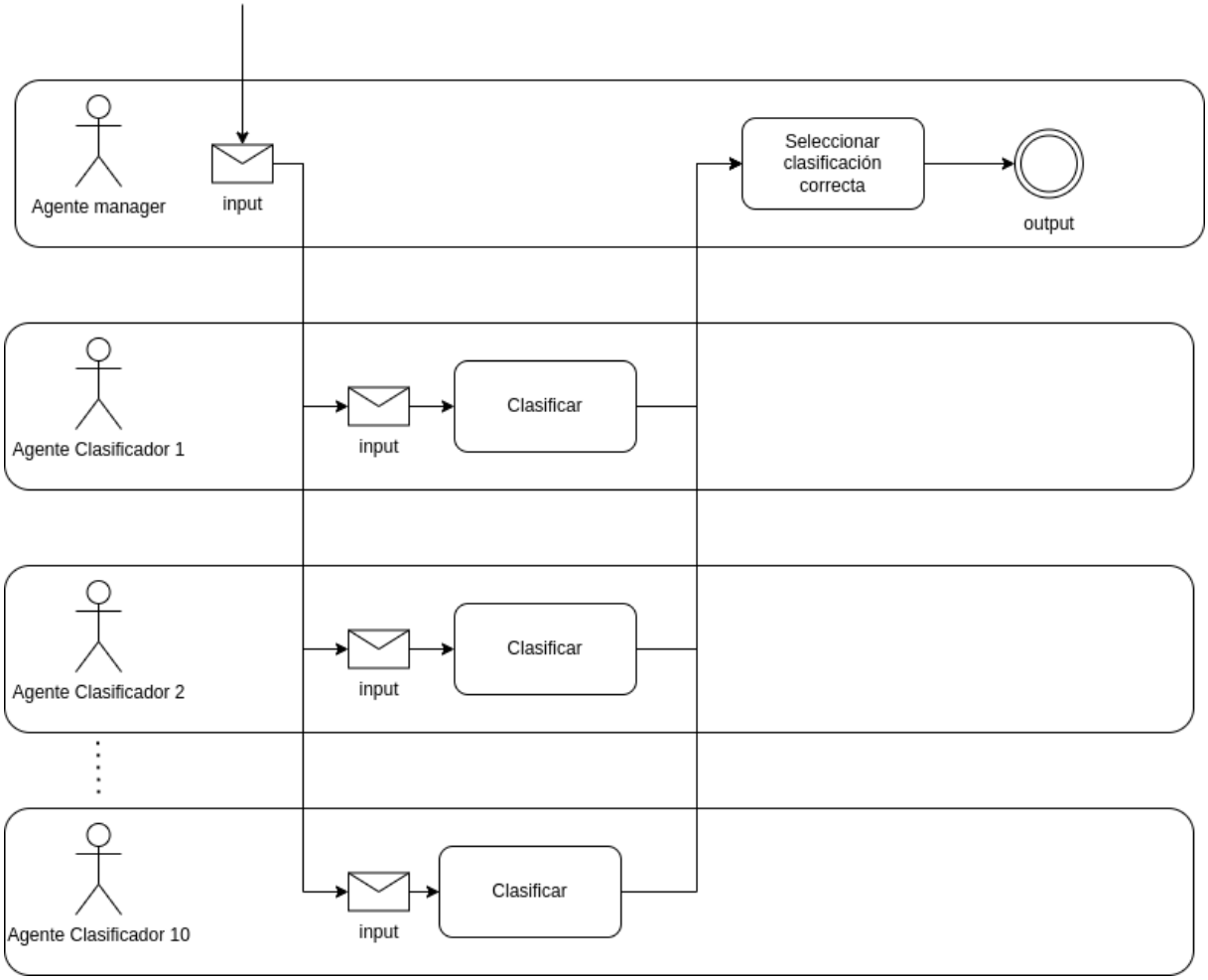

Figura 5.2: Diagrama BPMN del proceso de mixtura (Elaboración propia)

5.2.1. Agentes expertos

Para el desarrollo, se ha creado un tipo de agente llamado *AgentClassifier*, que en el *setup*, registra un método "predict" en el servicio *RPC* para recibir un *input*, y generar su predicción mediante un objeto "*self.classifier*". Esto se ha hecho así para poder registrar de forma sencilla varios métodos de clasificación (se ha elegido clasificación para este caso). Para registrar un agente, únicamente es necesario que el agente herede de este tipo de agente *AgentClassifier* y registre un clasificador en el objeto "*self.classifier*". Un ejemplo para el registro de un agente sería el siguiente:

```
    classifier = AgentClassifier(jid, password)
    future = classifier.start()
    future.result()

    agent_classifier.classifier = sklearn.neighbors.
       RandomForestClassifier()
```

De esta forma, el agente contiene un clasificador Random Forest (aún sin entrenar), que se utilizará para hacer la clasificación cuando otro agente realice una llamada (mediante RPC) al método llamado "predict". Este agente devolverá el resultado de la clasificación utilizando el clasificador Random Forest.

De esta forma, se han creado un total de diez agentes, cada uno de ellos con un método de clasificación diferente. Los métodos de clasificación seleccionados para los agentes han sido aquellos disponibles a través de la librería *scikit-learn* [17], siendo los métodos de clasificación: KNeighborsClassifier, MLPClassifier, SVC, LinearSVC, GaussianProcessClassifier, RandomForestClassifier, DecisionTreeClassifier, AdaBoost-Classifier, GaussianNB y QuadraticDiscriminantAnalysis. Para algunos experimentos se ha utilizado un agente por cada tipo de clasificador, mientras que para otros, se han repetido clasificadores.

Cada uno de estos clasificadores ha sido previamente entrenado con una porción del total de datos del dataset.

5.2.2. Agente manager

Este agente es el encargado de, dado un conjunto de datos *input*, preguntar a los expertos y devolver una única respuesta *output*.

Para ello, se ha modelado un comportamiento de SPADE. Este comportamiento es de tipo *OneShotBehaviour*, dado que únicamente se va a ejecutar una vez.

Este comportamiento recibe unos datos de entrada *input*. Estos datos de entrada se envían a los demás agentes expertos mediante la función *call_method* explicada en el capítulo anterior.

Esta llamada se ha introducido en un método *wrapper* para poder preguntar a una lista de *JIDs*, en lugar de a un único agente, y devolver una lista de respuestas. Por lo tanto, esta llamada a *call_method* queda de la siguiente manera:

```
    async def ask_to(self, JID, x):
        if type(JID) == list:
            tasks = [self.ask_to(jidx, x) for jidx in JID]
            return await asyncio.gather(*tasks)

        return await self.rpc.call_method(JID, 'predict', x)
```

Apuntar que se utiliza *asyncio.gather* para poder realizar las llamadas en paralelo, en lugar de tener que esperar a que un agente responda para preguntar al siguiente.

Además, se puede observar en el ejemplo de código, la simplicidad con la que se realiza una llamada a una función en un agente remoto.

Una vez envía los datos de entrada, el agente manager simplemente espera a recibir los datos de salida de todos los expertos. Una vez recibidos todas las predicciones, el agente simplemente elige la opción más votada.

Por ejemplo, si el agente pide predicciones entre dos clases 1 o 0 a siete clasificadores, y cinco clasificadores predicen 1 y los otros dos predicen 0, la clase elegida sería 1.

Por último, se ha añadido una serie de parámetros para poder realizar las estadísticas que se verán en el capítulo de resultados.

5.3. Agente directorio

Además de la mixtura de expertos, se ha desarrollado otro ejemplo para el plugin desarrollado. Este ejemplo trata de un agente directorio (o AgenteDF).

Un AgenteDF es un componente que proporciona un servicio de directorio de páginas amarillas a los agentes.

En FIPA [7], se proponen agentes directorio como custodios confiables y benignos del directorio de agentes. Se dice que es confiable porque el agente debe esforzarse por mantener una lista precisa, completa y oportuna de agentes. Además, es benigno porque debe proporcionar la información más actualizada sobre los agentes en su directorio de manera no discriminatoria a todos los agentes autorizados.

Este agente ofrece a los demás agentes la posibilidad de saber qué agente ofrece un servicio. Esto es, por ejemplo, dentro de un sistema de agentes, existen agentes que ofrecen un servicio para realizar predicciones (como los agentes expertos de la mixtura de agentes realizada). Si otro agente quisiera saber qué agentes dentro del sistema son capaces de hacer una predicción, preguntaría al agente directorio por aquellos agentes que ofrezcan el servicio "predict". El agente directorio le devolvería una lista de agentes que ofrezcan dicho servicio.

Para su mayor comprensión, se ha realizado el diagrama de casos de uso de la figura 5.3.

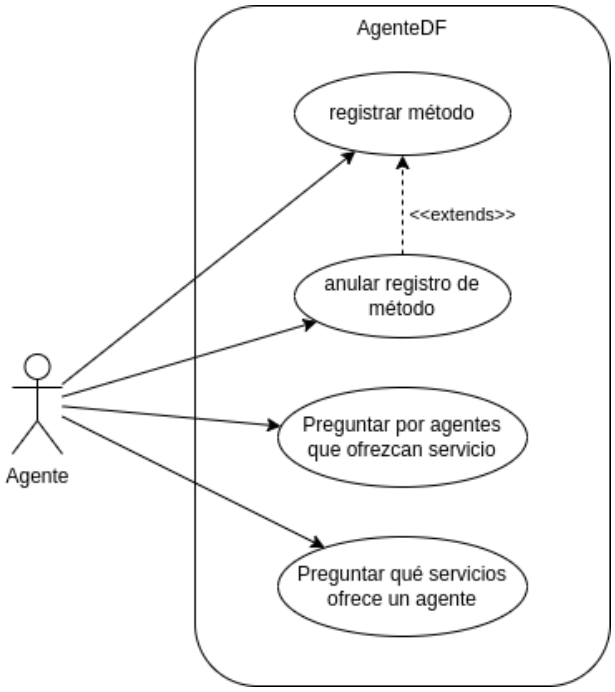

Figura 5.3: Caso de uso de AgenteDF

Así, este agente ofrece los métodos "register_method", "unregister_method", "list_methods"

y "list_jids". A través de estos métodos, utilizando el protocolo *RPC*, los demás agentes pueden registrar sus métodos y listar métodos ajenos. Esto es bastante útil para la plataforma SPADE, dado que da la posibilidad de descubrir servicios a los agentes.

5.4. Conclusiones

En este capítulo, se ha mostrado un posible ejemplo de la gran utilidad que es la ejecución en métodos remotos a través de la plataforma SPADE. En este ejemplo, se ha desarrollado una mixtura de expertos, en la que cada cada agente tiene un clasificador diferente.

La ejecución de métodos remotos da además la posibilidad de hacer que cada clasificador sea privado. Esto significa, que el agente *manager* no conoce la forma de clasificar de cada uno de los agentes, únicamente pide una clasificación y los expertos la devuelven. Esto añade una transparencia, pues cada clasificador puede entrenar con unos datos y tener un método de clasificación diferente.

Añadir también que, dado un sistema de agentes, pueden haber expertos que hayan sido entrenados con datos privados, y aquel agente que pregunte por una predicción no tiene por qué conocer estos datos.

Durante el siguiente capítulo, se mostrarán algunos ejemplos de la mixtura desarrollada.

Capítulo 6

Resultados

En este capítulo, se van a mostrar los resultados obtenidos. En primer lugar, se exploran los tests desarrollados para la librería *aioxmpp*, para justificar su correcto funcionamiento y que el código desarrollado sea funcional, tanto para SPADE, como para otros usos de la librería.

A continuación, se mostrarán los resultados obtenidos con la mixtura de agentes.

6.1. Tests

En esta sección, se explican los tests desarrollados para la librería *aioxmpp*.

Dado que la librería *aioxmpp* es de código abierto, es necesario y obligatorio el desarrollo de tests que justifiquen el correcto funcionamiento del código desarrollado. Además, al seguir un estándar, es necesario e importante que existan tests que validen el código, puesto que en el caso de que se incumpla el estándar, no sería posible la comunicación con otras aplicaciones.

Ya que, como se ha explicado en el tercer capítulo, se han desarrollado principalmente dos módulos en la librería *aioxmpp*; uno para las etiquetas, y otro para los servicios, se han creado dos archios de tests unitarios, un archivo para cada uno. Ambos archivos se encuentran en la ruta "test/rpc/" dentro de la librería.

6.1.1. Tests sobre etiquetas

El archivo de test que se encarga comprobar el funcionamiento de las etiquetas o *XSOs*, se llama *test_xso.py*. Este archivo contiene 79 tests que comprueban cada una de las etiquetas.

El primer aspecto que se tiene en cuenta para todas las etiquetas, es el correcto nombre de la etiqueta, es decir, que si se crea una etiqueta "methodCall" mediante su inicialización en pyton "methodCall()", que la etiqueta se inicialice como "<methodCall>".

Otro aspecto que tienen en cuenta los tests para cada una de las etiquetas es la correcta inicialización de los atributos de las etiquetas, es decir, que aquellos atributos que se pasen como parámetro en la inicialización de la etiqueta se asignen correctamente. Un ejemplo de ello, es que si se inicializa la etiqueta "methodCall" con

los parámetros ”methodName” y ”params”, utilizando la llamada en python ”methodCall(methodName, params)”, los valores ”methodCall.methodName” y ”methodCall.params” se hayan asignado correctamente.

Por último, también se tiene en cuenta que únicamente se puedan inicializar los atributos de las etiquetas con los tipos de etiquetas correspondientes. Esto significa, que dado que ”methodCall.methodName” solo puede contener una etiqueta de tipo ”methodName”, a la hora de inicializar dicho atributo, este atributo sea realmente de tipo ”methodName”.

6.1.2. Tests sobre los servicios

El achivo de test que se encarga de comprobar el funcionamiento de los servicios, se llama *test_service.py*. Este archivo contiene 15 tests que comprueban el funcionamiento de los servicios.

Dado que se han desarrollado dos servicios (un servicio para el cliente y otro para el servidor), se han separado los tests en dos clases; TestRPCServer, que contiene los tests del servidor y TestRPCClient, que contiene los tests del cliente.

6.1.2.1. TestRPCServer

TestRPCServer contiene un total de diez tests. Hay que tener en cuenta, que como se explica en el capítulo 3, los servidores deben implementar además de *XML-RPC*, el protocolo *Service discovery*. Por ello, se han desarrollado tests que se encargan de la correcta inicialización e identidad del nodo. Estos tests comprueban que el nodo se registra correctamente, además de que el servicio que ofrece es el correcto (*Jabber-RPC*).

Los tests del servidor comprueban principalmente el correcto funcionamiento del protocolo desarrollado *RPC*. Para ello, se han desarrollado tests que el servidor es capaz de manejar mensajes de tipo ”<iq>” con etiquetas ”methodCall” en su interior. Además, se comprueba que si se hacen llamadas a métodos que no están registrados en el servidor, el servidor devuelva un mensaje de error con el mensaje ”no such method: ’method”’. Dado que puede haber clientes sin acceso a determinados métodos, también se comprueba que, en el caso de que un cliente solicite la ejecución de un método al que no tiene acceso, el servidor devuelva un error de tipo *FORBIDDEN*.

Por último, se comprueba que si un cliente hace una llamada válida, el servidor devuelva una respuesta correctamente.

6.1.2.2. TestRPCClient

TestRPCClient contiene un total de cinco tests. Al igual que los tests de *TestRPC-Server*, en el cliente también se comprueba el correcto funcionamiento del protocolo *Service discovery*. Estos tests comprueban que si existe un servidor que ofrece el protocolo *Jabber-RPC*, el cliente es capaz de identificar que el servidor ofrece dicho protocolo. También se comprueba el caso contrario, es decir, si el servidor no ofrece el servicio, cuando el cliente trata de identificar si el servidor lo ofrece, devuelve un valor negativo.

Por otro lado, se comprueba que cuando se realiza una llamada a un procedimiento remoto, la estructura del mensaje enviado es correcta.

6.1.3. Cobertura de los tests

Entre ambos archivos de tests, se han desarrollado un total de 94 tests. Estos tests han sido validados con su correcta ejecución. Además, se ha ejecutado un test de cobertura, para observar cuál es el total de código desarrollado cubierto a través de los test realizados. Esto se ha realizado a través de la librería de python *coverage.py*, que ofrece la posibilidad de generar un reporte de la cobertura de los tests.

```
Name                                                Stmts   Miss  Cover   Missing
---------------------------------------------------------------------------------
aioxmpp/aioxmpp/rpc/service.py                         68      7    90%   111, 119, 123, 161, 220-223
/aioxmpp/aioxmpp/rpc/xso.py                           135      0   100%
---------------------------------------------------------------------------------
TOTAL                                                 203      7    97%
```

Figura 6.1: Cobertura del código de los tests sobre el código desarrollado

Se observan en la figura 6.1 los resultados del test de cobertura. Los tests unitarios del servicio tienen un total de 68 *statements*, de los cuales se cubren 61, teniendo un total de 90 % de cobertura.

De los tests unitarios de las etiquetas o *XSOs*, existe un total de 135 *statements*, de los cuales se cubren todos, el 100 % de los *statements*.

Se ha limitado la ejecución de los tests a los tests desarrollados, y la cobertura a el código desarrollado, no a toda la librería.

En total, gracias a los tests, se cubre un total del 97 % del código desarrollado.

6.2. Mixtura de agentes

A continuación, se va a estudiar el ejemplo desarrollado de mixtura de agentes. Para ello, se ha hecho una selección de diversos clasificadores, mezclándolos entre sí, con varios datasets.

Además, para hacer el ejemplo lo más real posible, se ha repartido el dataset entre los agentes. La forma de escoger la clase correcta de la clasificación es mediante votación; la clase más repetida entre las predicciones de los agentes expertos es la clase seleccionada.

La selección de datasets ha sido a través de dos librerías: *scikit-learn* [17] y *datasets* [9] de *hugging-face*. Los datasets elegidos son los siguientes:

- **Covertype:**

  Este dataset contiene observaciones de árboles de cuatro áreas del Bosque Nacional Roosevelt en Colorado. Todas las observaciones son variables cartográficas (sin sensores remotos) de secciones de bosque de 30 metros x 30 metros.

Este dataset incluye información sobre el tipo de árbol, la cobertura de la sombra, la distancia a puntos de referencia cercanos, el tipo de suelo y la topografía local.

Este dataset tiene como objetivo predecir el tipo de cobertura de los árboles.

- **Digits:**

  Este dataset contiene mapas de bits normalizados de dígitos escritos a mano a partir de un formulario impreso. De un total de 43 personas, 30 contribuyeron al conjunto de entrenamiento y otras 13 al conjunto de prueba.

  Los mapas de bits de 32x32 se dividen en bloques no superpuestos de 4x4 y el número de píxeles se cuenta en cada bloque. Esto genera una matriz de entrada de 8x8 donde cada elemento es un número entero en el rango 0-16. Esto reduce la dimensionalidad y da invariancia a las pequeñas distorsiones.

  El dataset tiene como objetivo predecir el dígito escrito a mano.

- **Breast Cancer:**

  Este dataset contiene datos de cáncer de mama que se obtuvieron a partir de datos de los Hospitales de la Universidad de Wisonsin.

  Las características se calculan a partir de una imagen digitalizada de una aspiración con aguja fina de una masa mamaria. Describen características de los nucleos celulares presentes en la imagen.

  El dataset tiene como objetivo predecir si existe o no cáncer de mama.

- **Diabetes Readmission**

  Este dataset está configurado por datos de *Center for Clinical and Translational Research, Virginia Commonwealth University*.

  Este dataset representa 10 años de atención clínica en 130 hospitales de EE.UU. Incluye más de 50 características que representan los resultados del paciente y del hospital.

  El dataset tiene como objetivo predecir la readmisión de pacientes con diabetes.

Para el primer experimento, se han utilizado cinco clasificadores del mismo tipo: *LinearSVC*. Como se ha explicado anteriormente, cada clasificador ha sido entrenado con datos únicos.

Dataset		**Covertype**	**Digits**	**Breast Cancer**	**Diabetes Readmission**
Accuracy (%)	**LinearSVC1**	54	80	92	50
	LinearSVC2	60	100	94	55
	LinearSVC3	56	100	96	50
	LinearSVC4	46	70	95	50
	LinearSVC5	56	80	95	45
	Mixtura	62	100	96	60

Cuadro 6.1: Experimento con cinco clasificadores del mismo tipo

Como se puede observar en el experimento realizado en el cuadro 6.1, la precisión de la mayoría suele mejorar ligeramente las decisiones individuales; a excepción del caso del dataset de *digits*, en el que la precisión llega al máximo en algunos clasificadores.

Se puede observar además, como para el dataset más complejo de aprender, *diabetes readmission*, la mejoría es notable.

A continuación, se realiza el siguiente experimento utilizando ocho clasificadores de distinto tipo. Dado que los datos de entrenamiento se tienen que repartir entre ocho agentes, es esperado que se reduzca su precisión individual.

Dataset		**Covertype**	**Digits**	**Breast Cancer**	**Diabetes Readmission**
Accuracy (%)	**MLPClassifier**	40	65	90	65
	SVC	60	95	100	50
	LinearSVC	65	80	95	35
	Gaussian Process	0	15	85	45
	Random Forest	70	80	100	65
	Decision Tree	60	65	95	60
	Ada Boost	55	30	95	60
	GaussianNB	55	65	100	55
	Mixtura	75	95	100	55

Cuadro 6.2: Experimento con ocho clasificadores diferentes

Se puede observar en el cuadro 6.2, que para el caso del dataset *covertype*, la mixtura obtiene un mejor resultado global que los agentes individuales. Para *digits* y *breast cancer*, la mixtura obtiene un resultado global igual a la del mejor agente, en el caso de breast cancer, porque existen agentes capaces de converger.

Para el caso de *diabetes readmission*, se observa que el resultado global de la mixtura es inferior a la de los agentes con mayor precisión. Esto seguramente se deba a que muchos clasificadores no sean capaces de acertar las precisiones. Por ello, se han utilizado dos instancias de los cuatro mejores clasificadores y se ha realizado el mismo experimento.

Dataset		**Covertype**	**Digits**	**Breast Cancer**	**Diabetes Readmission**
Accuracy (%)	**MLPClassifier 1**	38	65	90	65
	MLPClassifier 2	52	90	100	55
	Random Forest 1	66	80	100	45
	Random Forest 2	62	60	100	60
	Decision Tree 1	52	65	95	65
	Decision Tree 2	50	55	95	60
	Ada Boost 1	64	10	100	60
	Ada Boost 2	62	40	100	65
	Manager	70	90	100	70

Cuadro 6.3: Experimento con dos instancias de los cuatro mejores clasificadores

Como se puede observar en el cuadro 6.2, utilizando dos instancias de los cuatro mejores clasificadores, la precisión global de la mixtura incrementa considerablemente. Por lo tanto, podemos concluir que en el caso anterior, la precisión global empeoraba debido a que los peores clasificadores influían negativamente en la elección.

Para finalizar, se propone que a futuro se utilicen otros mecanismos distintos para la selección, en lugar de votaciones. Una de las opciones propuestas sería la fusión de datos. No obstante, esto se sale del ámbito del trabajo.

6.3. Conclusiones

Durante este capítulo, se ha revisado el correcto funcionamiento del código desarrollado a través de tests unitarios. Se ha explorado, cómo estos tests cubren la amplia mayoría del código desarrollado.

También se ha aprovechado el código desarrollado para experimentar con una mixtura de agentes. Se han realizado diferentes pruebas sobre esta mixtura de agentes y se han mostrado los resultados obtenidos.

Capítulo 7

Conclusiones

Gracias a este proyecto, no solo se ha ampliado la funcionalidad de SPADE a través de un plugin, si no que también se ha ampliado *aioxmpp*, aportando un nuevo XEP a la librería. Esto se ha logrado mediante la implementación de estándares, que hacen que el proyecto desarrollado tenga mayor fiabilidad y sea más versátil.

Mediante un primer análisis del problema, se ha logrado encontrar una solución al problema de realizar llamadas a procedimientos remotos en otros agentes, utilizando el protocolo *RPC*.

Además de conseguir con satisfacción las expectativas iniciales del proyecto, se ha logrado completar todos los objetivos especificados al inicio del mismo. A pesar de todas las funcionalidades del proyecto y de su desarrollo, todavía quedan opciones por explorar, como se puede consultar en la sección de trabajos futuros de esta memoria.

Durante la realización de este proyecto, se han encontrado distintas dificultades. La principal dificultad ha sido la comprensión del funcionamiento de los *XEPs*, la librería *aioxmpp* y el entorno *SPADE*.

También se han encontrado otras dificultades menores que se han ido resolviendo durante el desarrollo del proyecto. Un ejemplo de estas dificultades es la cantidad de datos que se pueden enviar a través de XMPP, dado que en ocasiones es necesario el envío de muchos parámetros a métodos remotos.

Poco a poco, gracias a este trabajo, mis conocimientos dentro del campo de SPADE y de los sistemas multiagente han mejorado notablemente haciendo, además, crecer mi interés. Ahora, conociendo más este campo, concibo muchos más proyectos que podrían ser interesantes y aportar utilidad.

Por último, agradecer a mis tutores, Vicente Javier Julian Inglada y Javier Palanca Camara, por brindarme la oportunidad, conocimiento y apoyo para realizar un proyecto tan interesante y didáctico como este.

7.1. Conclusiones finales

A continuación, se van a repasar los objetivos presentados al inicio de la memoria, para comprobar si estos se han cumplido con éxito:

- **Estudio de plataformas de SMA así como de las tecnologías existentes para poder proporcionar el acceso a ejecución de métodos remotos en una plataforma de agentes.**

  Este objetivo se ha cumplido con éxito en el capítulo dos, mediante un estudio del estado del arte, tanto de sistemas multiagente como de los posibles protocolos que se podrían utilizar para ampliar SPADE.

- **Proporcionar a la librería aioxmpp (librería que utiliza SPADE para el uso de XMPP) del protocolo Jabber-RPC.**

  Este objetivo se ha cumplido con éxito en el capítulo tres, con el desarrollo de la ampliación de la liberería *aioxmpp*, mediante la implementación del *XEP-0009, Jabber-RPC* en la librería.

  Gracias a esta implementación, no únicamente es posible que SPADE se beneficie del uso del protocolo *RPC*, sino que también puede beneficiarse del uso de este protocolo cualquier otra aplicación que utilice la librería *aioxmpp*.

- **Proporcionar a SPADE el plugin necesario para el uso de Jabber-RPC.**

  Este es el objetivo principal del proyecto. Este objetivo se ha cumplido con éxito en el capítulo cuatro, mediante el desarrollo del plugin que proporciona a los agentes de los métodos necesarios para el uso del protocolo *RPC*.

  Este objetivo cumplido permite que los agentes ofrezcan servicios a través de métodos, y que otros agentes puedan realizar llamadas a estos métodos. Este objetivo también permite que los agentes puedan realizar llamadas a procedimientos remotos que utilicen *XML-RPC*, no únicamente realizar llamadas a agentes dentro del sistema, sino también a otros servicios externos.

- **Validar los desarrollos realizados a través de un ejemplo de uso del plugin.**

  Este objetivo se ha cumplido durante el capítulo cinco, no únicamente con un ejemplo sino con dos; una mixtura de agentes y un agente directorio.

  Gracias a este objetivo, aquellos desarrolladores que quieran hacer uso del protocolo *RPC* dentro de SPADE, tendrán un claro ejemplo de su uso.

7.2. Relación con los estudios cursados

Desde el primer momento ha sido necesaria la puesta en práctica de los estudios cursados a lo largo del *Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital*. Desde la fase inicial del proyecto, se aplicaron conocimientos de la especialización *Inteligencia Artificial*, en concreto, de la asignatura *Sistemas Multiagente*.

Además, gracias al ejemplo propuesto de la mixtura de expertos, se han aplicado conocimientos de la especialización *Reconocimiento de formas*.

7.3. Trabajos futuros

En lo relativo al trabajo futuro, este proyecto ha abierto varias puertas para próximos proyectos que se recogen a continuación, dado que se puede utilizar el protocolo

desarrollado para otros proyectos:

- **Implementación de AgenteDF en un plugin:**

  En este trabajo se ha planteado un agente directorio a modo de ejemplo, sin embargo, sería conveniente la implementación de dicho agente a través de un plugin, para poder ser utilizado directamente dentro del entorno de SPADE.

- **Interfaz gráfica en AgenteDF:**

  Se propone además la implementación de una interfaz gráfica del agente directorio, para que sea visible a través de una interfaz aquellos métodos disponibles en todo el sistema de agentes. Además, se propone que mediante esta implementación, se puedan realizar llamadas a procedimientos remotos de forma manual mediante la interfaz.

- **Ampliación para el envío de grandes cantidades de datos:**

  Se propone como proyecto futuro, una ampliación que permita a SPADE la transferencia de datos de mayor capacidad, como podrían ser imágenes o cualquier otro tipo de archivo.

- **Paralelización de las llamadas a procedimientos remotos:**

  Dado que las llamadas a procedimientos remotos son bloqueantes, se propone que se puedan paralelizar las tareas bloqueantes dentro del sistema de SPADE de comportamientos.

Bibliografía

[1] D. Adams, «Jabber-RPC,» XMPP Standards Foundation, XEP 0009, ver. 2.2.1, 14 de sep. de 2001-4 de mar. de 2021. dirección: https://xmpp.org/extensions/xep-0009.html.

[2] G. Ali, N. Shaikh, M. Shah y Z. Shaikh, «Decentralized and Fault-tolerant FIPA-compliant Agent Framework Based on. Net,» *Australian Journal of Basic and Applied Sciences*, vol. 4, págs. 844-850, mayo de 2010.

[3] E. Argente, V. Julian y V. Botti, «Multi-agent system development based on organizations,» *Electronic Notes in Theoretical Computer Science*, vol. 150, n.º 3, págs. 55-71, 2006.

[4] K. Arnold, J. Gosling y D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[5] F. Bellifemine, A. Poggi y G. Rimassa, «JADE–A FIPA-compliant agent framework,» en *Proceedings of PAAM*, London, vol. 99, 1999, pág. 33.

[6] N. Howden, R. Rönnquist, A. Hodgson y A. Lucas, «JACK intelligent agents-summary of an agent infrastructure,» en *5th International conference on autonomous agents*, vol. 6, 2001.

[7] F. for Intelligent Physical Agents, «FIPA Agent Discovery Service Specification,» 2003.

[8] J. Karneges y P. Saint-Andre, «Message Stanza Profiles,» XMPP Standards Foundation, XEP 0226, ver. 0.3, 1 de ago. de 2007-5 de nov. de 2008. dirección: https://xmpp.org/extensions/xep-0226.html.

[9] Q. Lhoest, A. Villanova del Moral, Y. Jernite y col., «Datasets: A Community Library for Natural Language Processing,» en *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online y Punta Cana, Dominican Republic: Association for Computational Linguistics, nov. de 2021, págs. 175-184. arXiv: 2109.02846 [cs.CL]. dirección: https://aclanthology.org/2021.emnlp-demo.21.

[10] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan y G. Balan, «Mason: A multi-agent simulation environment,» *Simulation*, vol. 81, n.º 7, págs. 517-527, 2005.

[11] S. Masoudnia y R. Ebrahimpour, «Mixture of experts: a literature survey,» *Artificial Intelligence Review*, vol. 42, n.º 2, págs. 275-293, 2014.

[12] L. Melo, R. Sampaio, R. Leao, G. Barroso y J. Bezerra, «Python-based multi-agent platform for application on power grids,» *International Transactions on Electrical Energy Systems*, vol. 29, abr. de 2019. DOI: 10.1002/2050-7038.12012.

[13] P. Millard, P. Saint-Andre y R. Meijer, «Publish-Subscribe,» XMPP Standards Foundation, XEP 0060, ver. 1.24.1, 19 de nov. de 2002-21 de ene. de 2022. dirección: https://xmpp.org/extensions/xep-0060.html.

[14] M. Miller, «Ad-Hoc Commands,» XMPP Standards Foundation, XEP 0050, ver. 1.3.0, 8 de oct. de 2002-9 de jun. de 2020. dirección: https://xmpp.org/extensions/xep-0050.html.

[15] P. D. O'Brien y R. C. Nicol, «FIPA—towards a standard for software agents,» *BT Technology Journal*, vol. 16, n.º 3, págs. 51-59, 1998.

[16] J. Palanca, A. Terrasa, V. Julian y C. Carrascosa, «SPADE 3: Supporting the New Generation of Multi-Agent Systems,» *IEEE Access*, vol. 8, págs. 182 537-182 549, 2020. DOI: 10.1109/ACCESS.2020.3027357.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort y col., «Scikit-learn: Machine Learning in Python,» *Journal of Machine Learning Research*, vol. 12, págs. 2825-2830, 2011.

[18] A. S. Rao y M. P. Georgeff, «Modeling rational agents within a BDI-architecture,» *KR*, vol. 91, págs. 473-484, 1991.

[19] P. Saint-Andre, «Extensible messaging and presence protocol (XMPP): Core,» inf. téc., 2011.

[20] I. Shigeoka, «IQ Query Action Protocol,» XMPP Standards Foundation, XEP 0099, ver. 0.1.1, 25 de jun. de 2003-3 de nov. de 2018. dirección: https://xmpp.org/extensions/xep-0099.html.

[21] R. Srinivasan, «RPC: Remote procedure call protocol specification version 2,» inf. téc., 1995.

[22] G. Van Rossum y F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.