UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño e implementación de un simulador basado en agentes estilo JGOMAS en Python

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Sergio Alemany Ibor

Tutor: Carlos Carrascosa Casamayor

Javier Palanca Cámara

Vicente Javier Julián Inglada

Curso 2017-2018

Resumen

La propuesta de Trabajo de Fin de Grado (TFG) está enmarcado dentro del Grupo de Tecnología Informática e Inteligencia Artificial de la Universitat Politècnica de València (GTI-IA).

Se ha desarrollado un simulador de agentes inteligentes en el lenguaje de programación Python basado en la aplicación JGOMAS, utilizada en las prácticas de la asignatura Agentes Inteligentes del Grado de Ingeniería Informática.

Para ello, se han realizado varias tareas sobre otras capas del sistema, como la plataforma de agentes o el sistema de comunicaciones, que son necesarias para el desarrollo de dicha aplicación:

- Se añade el soporte de autorregistro de clientes en servidores XMPP a la biblioteca de software libre AIOXMPP, añadiendo un nuevo módulo que implemente la extensión al protocolo XMPP 0077 In-Band Registration.
- Se modifica la plataforma de agentes SPADE para añadir el autorregistro desarrollado anteriormente.
- Se migran las funcionalidades de la aplicación PGOMASv1 a una nueva versión de la aplicación, PGOMASv2, cambiando de Python2 a Python3 y añadiendo asincronía entre los comportamientos de los agentes mediante la biblioteca ASYNCIO.

Todo esto ha sido desarrollado en el lenguaje de programación Python3, montado sobre Ubuntu Linux, utilizando el servidor XMPP Prosody IM y el entorno de programación PyCharm Community.

Palabras clave: Agentes inteligentes, Sistema multiagente, JGOMAS, PGOMAS, SPADE, XMPP, ASYNCIO, AIOXMPP, XEP0077, In-Band Registration, Prosody IM.

Resum

La proposta de Treball de Fi de Grau (TFG) està emmarcada dins del Grup de Tecnologia Informàtica e Intel·ligència Artificial de la Universitat Politècnica de València (GTI-IA).

S'ha desenvolupat un simulador d'agents intel·ligents en el llenguatge de programació Python basat en la aplicació JGOMAS, utilitzada en les pràctiques de la assignatura Agents Intel·ligents del Grau en Enginyeria Informàtica.

Per aquest motiu, s'han realitzat varies tasques sobre altres capes del sistema, com la plataforma de agents o el sistema de comunicacions, necessàries per al desenvolupament de l'aplicació:

- S'afegeix el suport d'autorregistre de clients en servidors XMPP a la biblioteca de software lliure AIOXMPP, afegint un nou mòdul que implementa l'extensió al protocol XMPP 0077 In-Band Registration.
- Es modifica la plataforma d'agents SPADE per a afegir el autorregistre desenvolupat amb anterioritat.
- Es migren les funcionalitats de l'aplicació PGOMASv1 a una nova versió de l'aplicació, PGOMASv2, canviant de Python2 a Python3, y afegint asincronia entre els comportaments dels agents mitjançant la biblioteca ASYNCIO.

Tot açò ha sigut desenvolupat en el llenguatge de programació Python3, muntat sobre Ubuntu Linux, utilitzant el servidor XMPP Prosody IM y l'entorn de programació PyCharm Community.

Paraules clau: Agents intel·ligents, Sistema multiagente, JGOMAS, PGOMAS, SPADE, XMPP, ASYNCIO, AIOXMPP, XEP0077, In-Band Registration, Prosody IM.

Abstract

This End of Degree Project (EDP) proposal is framed within the Computer Technology and Artificial Intelligence Group of the Universitat Politècnica de València (GTI-IA).

A simulator of intelligent agents has been developed in the Python programming language based on the JGOMAS application, used in the practices of the Intelligent Agents subject of the Computer Engineering Degree.

For this, several tasks have been carried out on other layers of the system, such as the agent platform or the communications system, which are necessary for the development of said application:

- Added self-registration support for clients on XMPP servers to the free software library AIOXMPP, adding a new module that implements the extension to the XMPP protocol 0077 In-Band Registration.
- The SPADE agent platform is modified to add the self-registration previously developed.
- The functionalities of the PGOMASv1 application are migrated to a new version of the application, PGOMASv2, changing from Python2 to Python3 and adding asynchrony between the behaviours of the agents through the ASYNCIO library.

All this has been developed in the programming language Python3, mounted on Ubuntu Linux, using the XMPP Prosody IM server and the PyCharm Community programming environment.

Keywords: Intelligent agents, Multiagent system, JGOMAS, PGOMAS, SPADE, XMPP, ASYNCIO, AIOXMPP, XEP0077, In-Band Registration, Prosody IM.

Tabla de contenidos

Índice de contenido

Índice de figuras

1. Introducción

Entendemos como agentes inteligentes a entidades software capaces de percibir su entorno, procesar tales percepciones y responder o actuar en su entorno de manera racional, es decir, de manera correcta y tendiendo a maximizar el resultado esperado.

Entendemos también un sistema multiagente como un conjunto de agentes inteligentes que colaboran entre sí para alcanzar una meta común, difícil de alcanzar por un agente individual o sistema monolítico. Los sistemas multiagentes suelen referirse también a "sistemas autoorganizados", ya que tienden a encontrar la mejor solución para sus problemas "sin intervención".

Tanto los agentes inteligentes como los sistemas multiagentes tienen una gran importancia en el campo de la Inteligencia Artificial, tanto es así que hay una asignatura dedicada a ello en el grado de Ingeniería Informática llamada Agentes Inteligentes. En dicha asignatura se estudian tanto los agentes inteligentes como los sistemas multiagente, así como las interacciones entre ellos.

En las prácticas de esta asignatura, se plantea a los alumnos el desarrollo de un sistema multi-agente basado en una simulación de un juego estilo "atrapa la bandera" llamado JGOMAS [1]. En dicha simulación, un equipo de tropas de élite debe conseguir atrapar la bandera defendida por otro equipo de tropas de élite, y llevar la bandera de vuelta hasta su base. Estas tropas de élite consisten en agentes inteligentes que son programados por los alumnos.

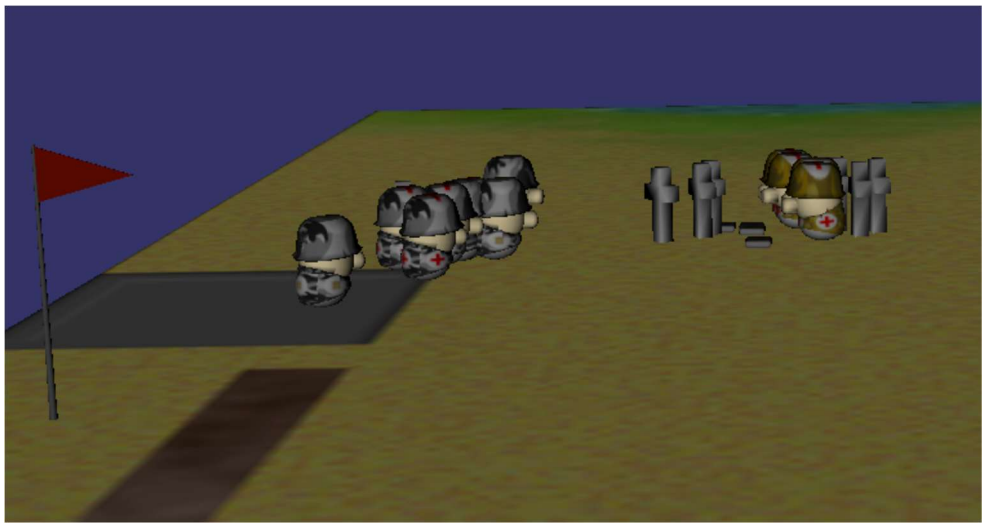

***Figura 1**: Captura JGOMAS*

Es un juego que integra la necesidad de cooperar a la vez que es una competición entre dos equipos. Los alumnos se organizan en equipos de 2 personas y se les da la posibilidad de elegir si desean programar las tropas atacantes (Allied) o defensoras (Axis) y se les proporciona una serie de agentes con comportamiento básico para que tengan una base sobre la que empezar a programar. A partir de esta base, el alumno deberá programar las tropas de élite, no como agentes inteligentes aislados, sino como partes de un sistema multiagente (el equipo atacante o defensor) en el que los agentes acuerdan, negocian y se organizan, formando una estrategia de equipo.

1.1. Motivación

Actualmente, JGOMAS está basado en el lenguaje de programación JASON [2], al menos la parte que concierne a los alumnos que deben programar los agentes. Los motivos por los que utilizar este lenguaje son varios:

- Está basado en la arquitectura BDI (Beliefs, Desires, Intentions), en la que un agente es visto como un agente racional con un conjunto de actitudes mentales (Creencias, Deseos e Intenciones) en la que realiza acciones en función de sus estados.
- Es un lenguaje interpretado, es decir, no necesita compilación para su ejecución, por lo que facilita las modificaciones durante el desarrollo.
- Está basado en Java, ya que el sistema base sobre el que se ejecutan los agentes inteligentes es JADE [3] quien, a su vez, también esté basado en Java.

A pesar de esto, la utilización de JASON tiene múltiples desventajas, entre las cuales cabe destacar las siguientes:

- Es un lenguaje poco extendido, con poca documentación.
- No ha sido utilizado con anterioridad por los alumnos y tiene pocas probabilidades de que sea utilizado en el futuro, dado el ámbito tan específico para el que fue creado.

Es por esto por lo que se hace ineficiente que los alumnos deban aprender un nuevo lenguaje considerado generalmente como “poco útil”, solamente para poder realizar las prácticas de una sola asignatura y difícilmente volver a emplearlo.

Además, actualmente gran parte del tiempo presencial de las prácticas de la asignatura se dedica a resolver dudas acerca de la sintaxis JASON, por lo que la productividad de las prácticas respecto a temas específicos de la asignatura es menor de lo deseado.

Por todos estos motivos, surge la necesidad de cambiar esta estructura y permitir a los alumnos utilizar un lenguaje más generalizado en la actualidad, como Python, el cual es un lenguaje interpretado, muy extendido, con gran cantidad de información, fácil de usar y ha sido utilizado anteriormente en otras asignaturas.

Esto permitirá reducir al mínimo el tiempo dedicado a aprender el lenguaje y maximizar el tiempo dedicado a programar y mejorar el comportamiento de las tropas de élite.

1.2. Objetivos

El objetivo es, por tanto, proporcionar a los alumnos que cursen la asignatura de Agentes Inteligentes, un simulador de agentes basado en el lenguaje de programación Python que les permita desarrollar las prácticas propuestas utilizando dicho lenguaje. Para conseguir este objetivo, se han planteado diversos subobjetivos en los que puede dividirse el objetivo principal:

- Desarrollar una plataforma de agentes en Python que gestione eficientemente los agentes creados, similar a lo que hace JADE.
- Trasladar el comportamiento de los agentes de JGOMAS a la nueva plataforma en Python.
- Realizar pruebas que validen el desarrollo realizado.

Sobre estos objetivos se fundamenta todo el peso de este trabajo.

1.3. Impacto Esperado

La aplicación de este trabajo a la asignatura de Agentes Inteligentes supondrá una serie de mejoras tanto para los alumnos como para los profesores de la asignatura.

En concreto, se espera una mejora del rendimiento de los futuros alumnos de la asignatura Agentes Inteligentes, debido a la reducción del tiempo empleado para aprender un nuevo lenguaje de programación.

También permitirá a los alumnos afianzar su conocimiento en el lenguaje Python y, por lo tanto, mejorar el rendimiento en la realización de asignaturas posteriores que también utilicen Python.

También, dado que los alumnos tendrán un mayor conocimiento sobre el lenguaje empleado, se realizarán menos preguntas sobre la sintaxis del lenguaje, por lo que el profesor dedicará menos tiempo a resolver dudas sobre este tipo y más tiempo a resolver dudas específicas sobre la asignatura, aumentando la productividad del tiempo de prácticas.

Por último, se dispondrá de una herramienta de simulación de Sistemas multiagente, que no solo se aplique a este uso en concreto, sino que será útil en otros entornos, como por ejemplo otras asignaturas similares o en la investigación de sistemas multiagentes.

1.4. Estructura

Este documento introduce progresivamente las nociones necesarias para la comprensión del trabajo realizado.

En primer lugar, se introducen los objetivos del trabajo y la motivación para su ejecución, así como las ventajas que supondrá el correcto desarrollo del trabajo.

A continuación, se repasará la situación actual de la arquitectura del sistema objeto de estudio, así como el de sus componentes. También se expondrán cuáles son los puntos débiles del sistema actual, y que justifican el desarrollo de este trabajo.

Más adelante, se analizará con más detalle el problema a resolver en este trabajo, indicando los requisitos que deberá cumplir el sistema final, enumerando posibles soluciones a este problema, y seleccionando justificadamente una de estas soluciones. También se añadirá un presupuesto aproximado del coste de la solución propuesta.

A partir de la solución seleccionada, se aumentará el nivel de detalle para describir cómo se llegará a la solución propuesta, mediante la estructura de las clases que se utilizarán, y la estructura del código final.

Finalmente, se describirán las pruebas realizadas para verificar el desarrollo, se expondrán las conclusiones del trabajo realizado, y se indicarán posibles trabajos futuros a realizar a partir de este trabajo.

2. Estado del arte

En este capítulo se pretende introducir todos los conceptos necesarios para la correcta comprensión del trabajo:

En un primer lugar, se resumirán brevemente las reglas dentro del juego JGOMAS, así como la taxonomía de los agentes en JGOMAS y cuál es la función de cada uno de ellos.

A continuación, se describirá la arquitectura actual utilizada en las clases de prácticas por los alumnos de la asignatura de Agentes Inteligentes, la cual es el objeto de cambio en un futuro próximo.

Seguidamente, explicaremos cual es la arquitectura correspondiente a una primera aproximación a la arquitectura objetivo del trabajo que se desarrolló mucho antes del comienzo de éste.

Una vez terminadas las explicaciones, se pasará a exponer los motivos por los que esa primera aproximación no cumple con los objetivos del trabajo.

Por último, se propondrá sin entrar en detalle, una nueva configuración de la arquitectura que permita cumplir con los objetivos del trabajo.

2.1. Reglas del juego

En JGOMAS, dos equipos (Alianza y Eje) se enfrentan en un mapa limitado, con tiempo limitado. El comportamiento de cada uno de los equipos no es directamente interactivo con los usuarios, sino que los agentes siguen un comportamiento basado en la programación que se le haya implementado.

Este enfrentamiento se asemeja al conocido juego de atrapa la bandera con ambientación militar, ya que cada agente representa una tropa de élite de una facción militar.

Cada equipo tiene una base propia, donde comienzan los agentes de cada equipo al empezar el juego y en el caso del equipo del eje, también es donde se sitúa la bandera, objetivo central del juego.

El objetivo del equipo de la alianza es capturar la bandera situada en la base enemiga y llevarla a la base propia, mientras que el objetivo del equipo del eje es defender la bandera y evitar que el equipo enemigo se la lleve a su base.

El equipo de la alianza gana si consigue trasladar la bandera a su base. Al contrario, el equipo del eje gana si todos los agentes del equipo de la alianza son eliminados o si se agota el tiempo de juego.

Todos los agentes tienen armas de fuego, con las que son capaces de disparar a distancia, con una determinada munición que causan un determinado daño a la salud del agente que recibe el disparo.

La cantidad de munición, así como el daño infligido y la salud de cada agente, viene determinado por el tipo de agente. Existen tres tipos de agentes clasificados de la siguiente manera:

- Los soldados son la tropa pesada del equipo, tienen más salud y sus armas infligen más daño. Además, son capaces de responder a llamadas de ayuda de agentes aliados y actuar en consecuencia.
- Los médicos son los encargados de la curación de los demás miembros del equipo. Son capaces de soltar paquetes de salud, los cuales brindan una determinada salud a aquel agente aliado que pase por encima de él. Responden a llamadas de asistencia médica de agentes aliados.
- Los operadores de campo cargan una mochila llena de munición. Son capaces de soltar paquetes de munición, que es proporcionada a los agentes que pasen por encima. Responden a las llamadas de munición de los agentes aliados.

Los paquetes soltados tanto por los médicos como por los operadores de campo se mantienen en la posición en la cual han sido soltados y se destruyen cuando un agente pasa por encima o cuando ha pasado un tiempo determinado. Si el agente que pasa por encima es un aliado, éste se beneficiará de lo que proporcione el paquete. En caso de que lo pise un enemigo, este se destruirá sin más.

Las balas disparadas dañan al primer agente con el que se cruzan, sea amigo o enemigo por lo que, si un agente dispara hacia un agente enemigo, habiendo un agente aliado en medio, este último será el que reciba el daño.

Antes de comenzar cada batalla, se eligen los integrantes de cada equipo, el mapa sobre el que se va a desarrollar el juego y un tiempo límite de juego.

2.2. Taxonomía de los agentes en JGOMAS

En JGOMAS, existen diversos tipos de agentes, y aunque la arquitectura de estos agentes varíe, siempre podemos clasificar los agentes que aquí participan de la siguiente manera:

JGomasAgent: Extiende de la clase Agent o equivalente de la plataforma que se esté utilizando. Es la clase base para todos los agentes dentro de JGOMAS:

- **Manager**: Es una clase interna no accesible por los usuarios. Los principales servicios ofertados por este agente especial son: la coordinación y sincronización de los otros agentes y la aportación de información a los agentes según su campo de visión.
- **Paquete**: Es otra clase interna que representa objetos relevantes sobre el entorno, con los cuales las tropas pueden interactuar.
- **Tropa**: Es la única clase externa, de la cual todos los subtipos de tropa deben extender:
    - **Soldado**: Estos son los agentes que lideran la batalla, ya que ofrecen a su equipo un servicio de soporte.
    - **Médico**: Este agente ofrece un servicio de curación, por lo que cualquier agente de su equipo puede solicitarle ayuda médica durante la batalla.
    - **Operador de campo**: Este agente ofrece un servicio de munición, por lo que cualquier agente de su equipo puede solicitarle munición durante la batalla.

2.3. Arquitectura actual

JGOMAS está compuesto principalmente de dos subsistemas. Por una parte, hay un sistema multiagente con dos clases diferentes ejecutándose. Uno de estos tipos controla la lógica actual del juego, mientras que los otros pertenecen a uno de los dos equipos, y estarán jugando el juego completo. Realmente, este subsistema es una capa que funciona por encima de una plataforma de sistema multiagente, específicamente JADE.

JASON-JGOMAS, que es la versión actual de la plataforma JGOMAS, permite el uso de agentes Jason para formar los equipos de agentes participantes.

Por otro lado, se ha desarrollado un visor gráfico (Render Engine) ad-hoc para visualizar un entorno virtual 3D. De acuerdo con los requerimientos propios de aplicaciones gráficas (alto coste computacional por cortos periodos) este visor gráfico ha sido diseñado como un módulo externo (y no como un agente). Ha sido escrito en C++ usando la biblioteca gráfica OpenGL.

También existe una versión del visor gráfico desarrollado en el motor gráfico Unity, con gráficos mejorados y más realista.

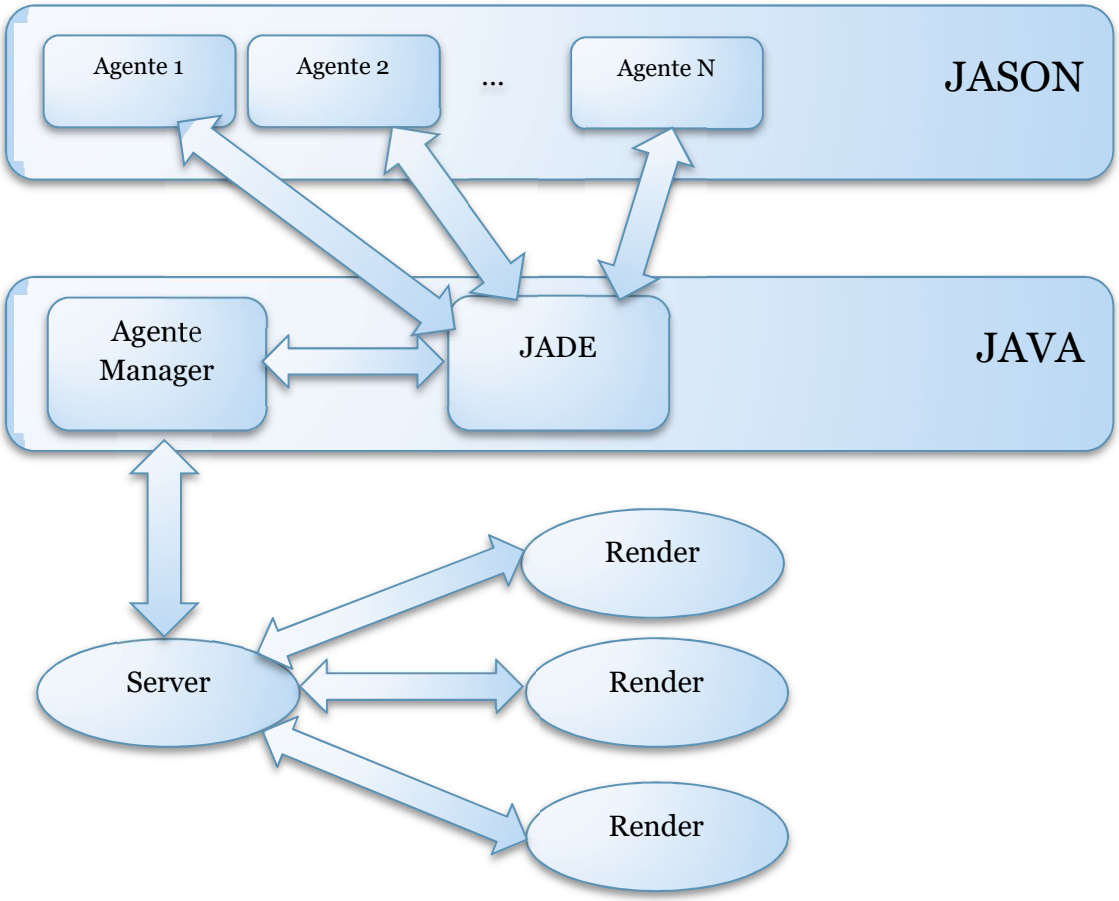

Figura 2: *Arquitectura JGOMAS*

Como se puede observar en la figura 2, los agentes correspondientes a las tropas de élite están escritos en JASON, mientras que el agente mánager encargado de la lógica del juego, está escrito en Java.

El visor gráfico, al estar programado como un módulo externo, es independiente de la plataforma utilizada, y por tanto queda fuera de los límites de actuación de este trabajo.

Excluyendo el visor gráfico, y teniendo en cuenta los campos de aplicación de este trabajo, podemos dividir la arquitectura actual del sistema en tres capas claramente diferenciadas: aplicación, plataforma y sistema de comunicaciones.

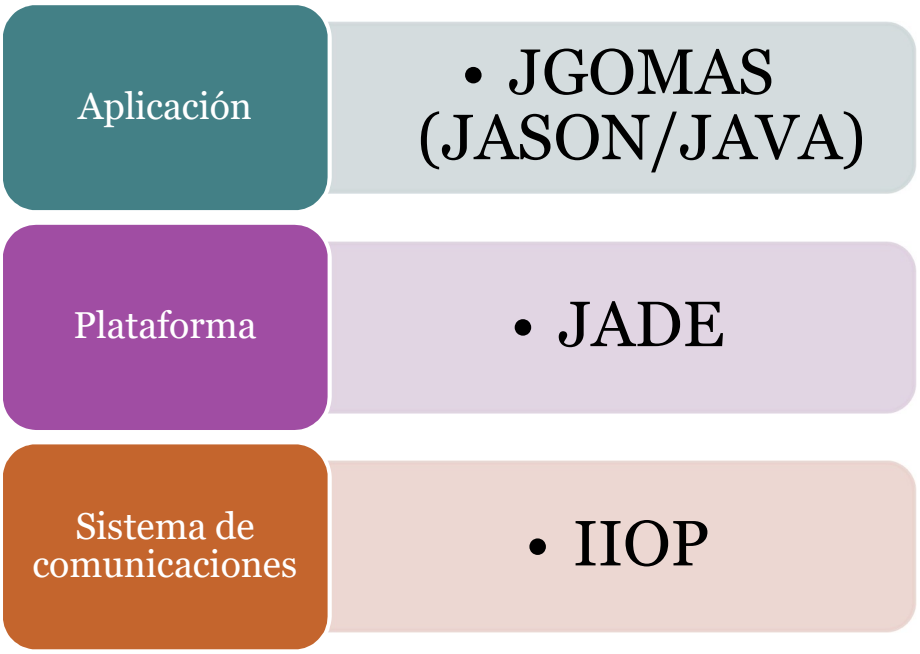

Figura 3: *Arquitectura antigua*

2.3.1. JGOMAS

En la capa de aplicación se sitúa JGOMAS (Game Oriented Multi-Agent System, based on JADE and JASON). Está formado por un conjunto de clases escritas en JASON y Java (todas las clases pertenecientes a las tropas están escritas en JASON, mientras que la clase mánager está escrita en Java). Anteriormente, todas las clases, sin excepción, estaban escritas en Java.

La estructura de las clases es la siguiente:

- **jgomas.asl**: Fichero principal de los agentes JGOMAS. Implementa los métodos y comportamientos comunes a todos los tipos de tropas de élite.
- **Jgomas.jar**: Contiene una biblioteca Java donde se encuentra el mánager de JGOMAS. Implementa los métodos y comportamientos relacionados con el agente mánager que se encarga de gestionar el transcurso del juego. En esta biblioteca también se encuentra la definición de los agentes que definen los paquetes soltados por las tropas de élite.

Los siguientes elementos de la lista incluyen el fichero jgomas.asl y representan un tipo específico de tropa, implementando el comportamiento específico de cada uno de ellos:

- **jasonAgent_ALLIED.asl**: Representa un soldado de la alianza.
- **jasonAgent_ALLIED_FIELDOPS.asl**: Representa un operador de campo de la alianza.
- **jasonAgent_ALLIED_MEDIC.asl**: Representa un médico de la alianza.
- **jasonAgent_AXIS.asl**: Representa un soldado del eje.
- **jasonAgent_AXIS_FIELDOPS.asl**: Representa un operador de campo del eje.
- **jasonAgent_AXIS_MEDIC.asl**: Representa un médico del eje.

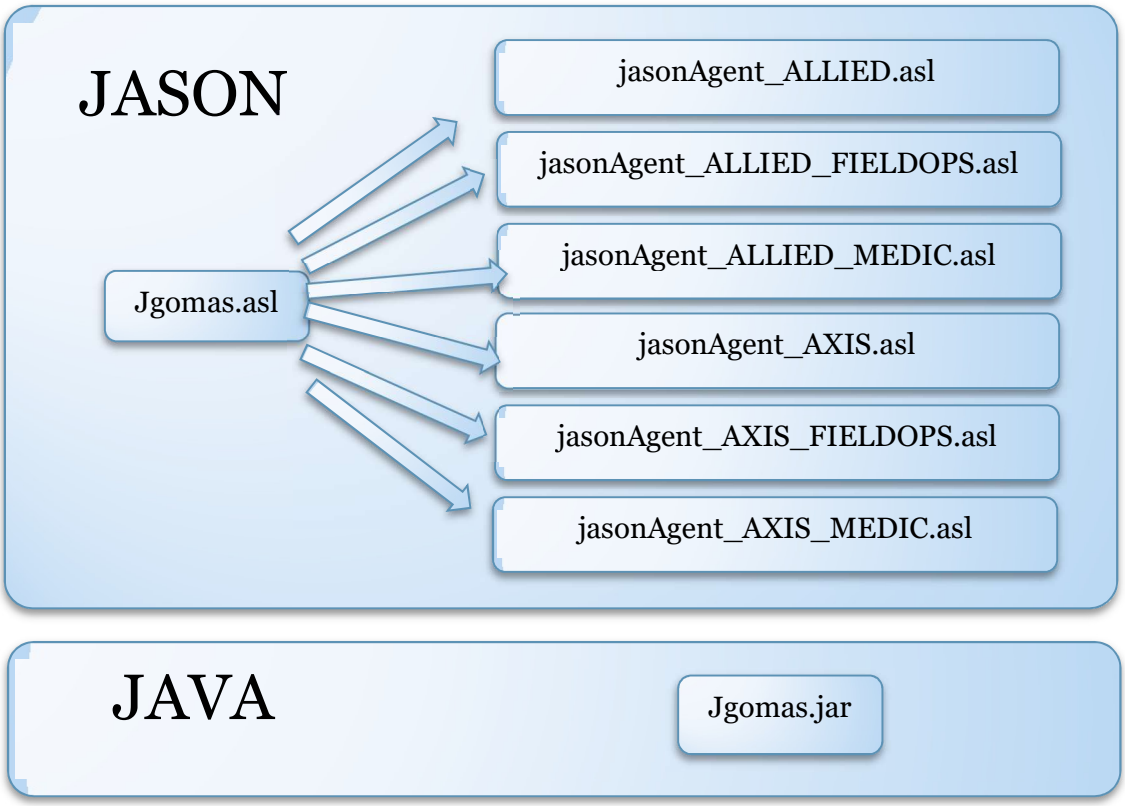

Figura 4: *Estructura JGOMAS*

Según esta estructura, cada tropa de élite constituye un agente por sí solo. También el mánager aparece como un agente omnipresente, capaz de comunicarse con todas las tropas con el fin de organizarse.

2.3.2. JADE

JADE (Java Agent Development Framework) es un framework implementado en Java, que simplifica la implementación de sistemas multiagentes a través de un middleware que cumple el estándar FIPA [4] y unas herramientas gráficas que soportan las fases de depuración e implementación.

Además de la abstracción de agentes, JADE proporciona un modelo de ejecución simple pero potente, comunicación agente-agente basada en el intercambio de mensajes asíncrono, soporte de páginas amarillas para los servicios de los agentes, y otras funcionalidades que facilitan el desarrollo de un sistema distribuido.

Podemos extender las clases propias de JADE para ampliar las funcionalidades de los agentes, o para adaptarlos según las necesidades de cada aplicación, precisamente lo que se realiza en esta arquitectura (la clase JGomasAgent extiende de la clase Agent de JADE).

2.3.3. IIOP

En computación distribuida, llamamos GIOP (General Inter-ORB Protocol) [5] al protocolo por el cual los ORBs (Object Request Broker) se comunican.

Un ORB consiste en una capa de software que permite a los objetos realizar llamadas a métodos situados en máquinas remotas, a través de una red.

IIOP (Internet Inter-ORB Protocol) es la implementación más importante de GIOP para TCP/IP. Es una realización concreta de las definiciones abstractas de GIOP. Es decir, IIOP es un protocolo que posibilita a los programas distribuidos escritos en diferentes lenguajes poder comunicarse a través de Internet.

La sintaxis de este protocolo está basada en el CDR (Common Data Representation), que mapea los tipos de datos en una representación a bajo nivel para su transferencia “por el hilo” entre ORBs.

A efectos prácticos, este protocolo es casi exclusivo del lenguaje de programación Java, ya que existe poco soporte para su implementación en otros lenguajes. Por tanto, no resulta interesante aplicar en otros lenguajes de programación.

2.4. Primera aproximación a la nueva arquitectura

En 2005, se realizó un primer intento a la tarea de trasladar la plataforma de JGOMAS a Python, generando una nueva plataforma de agentes, llamada SPADE, y traduciendo las clases de JGOMAS a Python, quedándose su estructura de la siguiente forma:

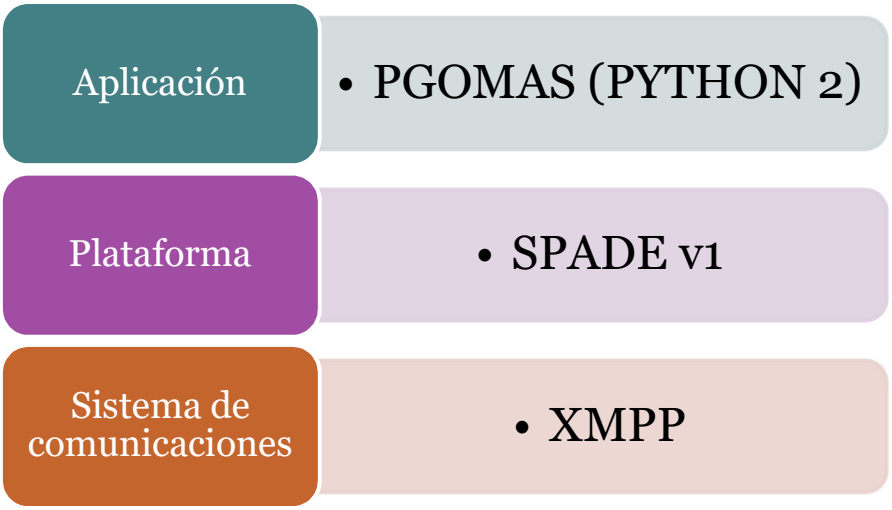

Figura 5: *Primera aproximación a la nueva arquitectura.*

Este primer intento es completamente funcional, aunque exageradamente ineficiente por múltiples razones que se explicarán en apartados siguientes.

2.4.1. PGOMAS

PGOMAS (Game Oriented Multi-Agent System, based on Python) es el equivalente a JGOMAS en Python 2. Consiste básicamente en una traducción literal de las clases equivalentes en java. Es decir, a partir de las clases antiguas de Java, que definían los agentes antes que se implementaran las clases JASON, se ha traducido de manera prácticamente literal a Python 2, cambiando la sintaxis solamente para adecuarse la sintaxis de Python.

La estructura de clases es claramente diferente a la de Jason-JGOMAS, en la que los métodos complementarios están modularizados en otras clases, y además aparecen nuevos agentes.

Según esta estructura, ya no solo se consideran las tropas y el mánager como agentes, sino que los objetos inanimados, como los packs que suelta el médico o la bandera objetivo, también se consideran agentes.

Podemos dividir las clases de PGOMAS entre aquellas que extienden directamente de un agente y aquellas que sirven de utilidad:

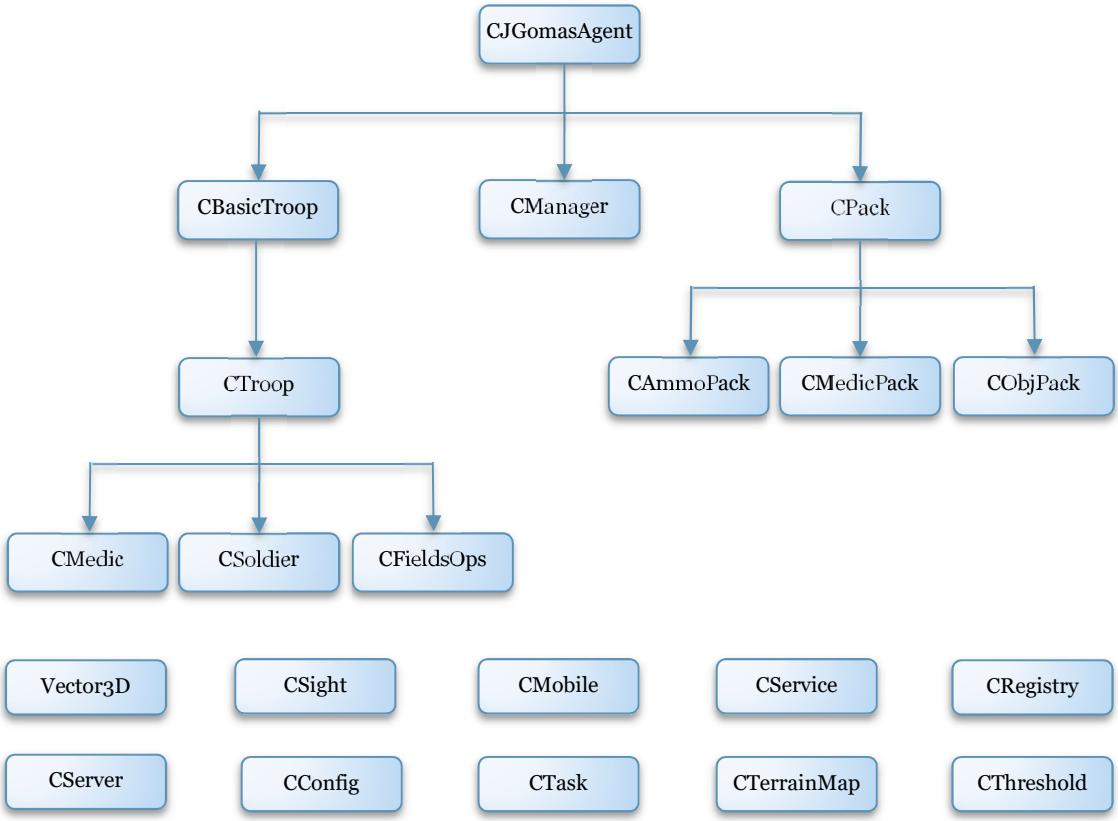

Figura 6: *Diagrama clases PGOMAS*

2.4.2. SPADE v1

SPADE (Smart Python multi-Agent Development Environment) [6] es una plataforma para sistemas multiagente basada en la tecnología XMPP y escrita en el lenguaje de programación Python.

La librería de agentes SPADE es un módulo Python para crear agentes SPADE. Es una colección de clases, funciones y herramientas para crear nuevos agentes SPADE que puedan trabajar con la plataforma SPADE.

Esta plataforma está basada en dos principales conceptos: la clase Agent de JADE y el sistema de mensajería XMPP, que se explicará en el siguiente apartado.

Esto significa que se partió de la clase Agent, trasladando la clase desde Java a Python todas aquellas funciones necesarias para el correcto funcionamiento de la plataforma, para luego expandir este sistema con múltiples funcionalidades adicionales que el sistema XMPP ofrece.

2.4.3. XMPP

XMPP (Extensible Messaging and Presence Protocol) [7], anteriormente llamado JABBER, es un protocolo abierto para mensajería instantánea basado en XML. Fue desarrollada originariamente para proporcionar una alternativa abierta y descentralizada a la mensajería instantánea cerrada de la época.

Este protocolo presenta una serie de ventajas sobre otros servicios como las siguientes:

- **Abierto** - Los protocolos XMPP son gratis, abiertos, públicos y fácilmente entendibles. Además, existen múltiples implementaciones en forma de servidores, clientes y bibliotecas.
- **Estandarizado** - La IETF (Internet Engineering Task Force)[1] ha formalizado el protocolo XMPP como una tecnología de mensajería instantánea estándar, y sus especificaciones han sido publicadas. Las especificaciones del núcleo del protocolo están publicadas en la RFC3920 [7] y en la RFC3921 [8] en 2004 y se revisaron en 2011, obteniendo las especificaciones más actuales de este protocolo (RFC6120 [9], RFC6121 [10] y RFC7622 [11]).

  Además, la Fundación de Estándares de XMPP ha publicado y sigue publicando una serie de "Extensiones al Protocolo XMPP" (XEP) [12] que forman una serie de propuestas adicionales al núcleo del protocolo, el desarrollo de las cuales es opcional para cada implementación del protocolo.
- **Probado** - Desde que Jeremie Miller desarrolló el protocolo JABBER el 1998, ha habido gran cantidad de implementaciones de este protocolo. Existen numerosos servidores XMPP corriendo actualmente, abundantes clientes implementados disponibles para su descarga y bibliotecas que implementan este protocolo.

  Algunos ejemplos de estas implementaciones son las siguientes:

  - Clientes: Gajim[2], AstraChat[3] y Spark[4]
  - Servidores: Ejabberd[5] y Prosody IM[6]

[1] https://www.ietf.org/
[2] https://gajim.org/
[3] https://astrachat.com/
[4] https://igniterealtime.org/projects/spark/
[5] https://www.ejabberd.im/
[6] https://prosody.im/

- **Bibliotecas**: Como ejemplo más práctico tenemos el de la biblioteca que se va a utilizar en este trabajo, AIOXMPP, la cual se verá más adelante.
- **Descentralizado** - La arquitectura de una red XMPP es similar a la del correo electrónico, por lo que cualquiera puede desplegar su propio servidor XMPP, permitiendo tanto a las organizaciones como a las personas individuales, tomar el control de las comunicaciones.
- **Seguro** - El protocolo XMPP permite los cifrados TLS y SASL para sus comunicaciones en las especificaciones. Además, los servidores XMPP se pueden usar en redes internas (como intranets de empresas) aisladas de redes públicas.
- **Extensible** - A parte de las especificaciones que forman el "núcleo" del protocolo XMPP, existen una serie de "Extensiones al Protocolo" las cuales permiten ampliar la funcionalidad de cada servidor o cliente de forma optativa. Además, al ser un protocolo basado en XML, cualquiera puede desarrollar extensiones privadas por encima de las especificaciones núcleo si se desea.
- **Flexible** - Las aplicaciones del protocolo XMPP no solo se limitan a la mensajería instantánea, sino que pueden usarse para muchas otras aplicaciones, incluyendo: gestión de redes, compartición de archivos, monitorización de sistemas remotos, juegos, servicios web y computación en la nube.

2.5. SPADE v3

SPADEv3 es una iteración sobre la primera versión SPADEv1, que trata de solucionar algunos de los fallos que se detectaron en esta y que están explicados más adelante.

Esta nueva versión está implementada con Python 3, y tiene un diagrama de clases mucho más simplificado que su primera versión, ya que en este caso no se optó por añadir el máximo número de servicios posibles a la plataforma, sino de dejar los componentes fundamentales de un agente, para aumentar su eficiencia. El resultado es una plataforma mucho más sencilla de mantener y entender, y más ligera que su predecesora.

En concreto, la plataforma tan solo contiene seis clases, cada una implementando una funcionalidad específica:

- **agent.py:** Implementa el concepto de agente, el cual es capaz de conectarse a un servidor XMPP para el intercambio de mensajes con otros agentes y puede contener una serie de comportamientos (Behaviours).
- **behaviour.py:** Implementa el concepto de comportamiento, el cual es capaz de ejecutar acciones y enviar y recibir mensajes. También implementa un comportamiento basado en una máquina finita de estados.
- **message.py:** Define un mensaje y los métodos útiles para trabajar con él.
- **presence.py:** Gestiona todo lo relacionado con la notificación de presencia de los agentes y poder determinar el estado actual del agente en tiempo real.
- **template.py:** Implementa los modelos de mensajes, que permiten discriminar para los mensajes que llegan a un agente, el comportamiento al cual va dirigido ese mensaje.
- **web.py:** Permite abrir un socket mediante el cual cada agente puede tener una interfaz web de gestión. También se pueden crear webs personalizadas para cada agente. Esta clase aún no está terminada, aunque no tiene demasiada relevancia para el desarrollo del proyecto.

Como características más destacables de esta plataforma podemos destacar la aplicación de concurrencia con la biblioteca ASYNCIO [13], y la aplicación de la mensajería instantánea también concurrente con la biblioteca AIOXMPP [14].

2.5.1. ASYNCIO

Una de las características fundamentales de los programas informáticos es que se ejecutan de manera secuencial, una línea detrás de otra. Si en un programa se accede a un recurso situado en un servidor remoto, significa que mientras se accede al recurso, el programa estará sin hacer nada, esperando a recibir la respuesta. En algunos casos, este comportamiento no es aceptable.

El sistema estándar para lidiar con este comportamiento es el uso de ‘threads’ o hilos. Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea. Un programa puede lanzar múltiples hilos, cada uno haciendo una cosa al mismo tiempo. Juntos, estos hilos permiten al programa realizar múltiples tareas al mismo tiempo.

Sin embargo, el uso de hilos también tiene complicaciones asociadas, como el aumento de la dificultad del programa, o errores de concurrencia asociados al uso de hilos que no aparecen al ejecutar un programa secuencialmente, como las condiciones de carrera, el bloqueo mutuo o la inanición.

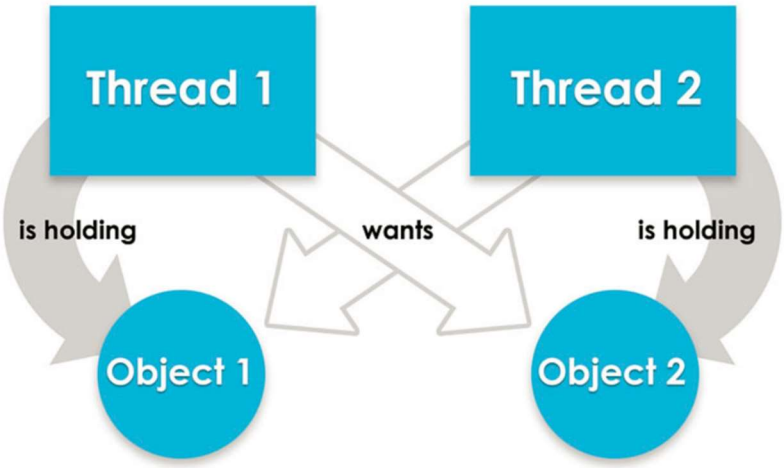

Figura 7: *Ejemplo bloqueo mutuo*

Además, cada núcleo del procesador tan solo puede ejecutar un hilo a la vez, por lo que cuando se tienen más hilos en ejecución que núcleos en el procesador, el núcleo del procesador está cambiando constantemente de contexto, es decir, cambia el hilo en ejecución a cada instante para 'simular' la ejecución concurrente de varios hilos. Este proceso es llamado cambio de contexto y aunque es muy rápido en los procesadores actuales, sigue teniendo un coste temporal asociado.

2.5.1.1. Programación asíncrona

La programación asíncrona es esencialmente la simulación de hilos por software y no por hardware, donde es la aplicación quien maneja los hilos y los cambios de contexto y no el núcleo del procesador.

Esta diferencia presenta varias ventajas respecto a la programación multi-hilos:

- Disminuye la probabilidad de que se produzcan errores de concurrencia: Con la utilización de código asíncrono, se sabe exactamente cuándo el código cambiará de una tarea a otra, por lo que dificulta la aparición de este tipo de errores.
- Disminuye el tiempo de cambio de contexto: Al realizar el cambio de tarea a nivel de software, ya no es necesario realizar cambios de contexto de un hilo a otro constantemente. Este hecho, unido a que los cambios son controlados y suelen

ser mucho menos frecuentes, conlleva una reducción de los tiempos dedicados a este propósito.

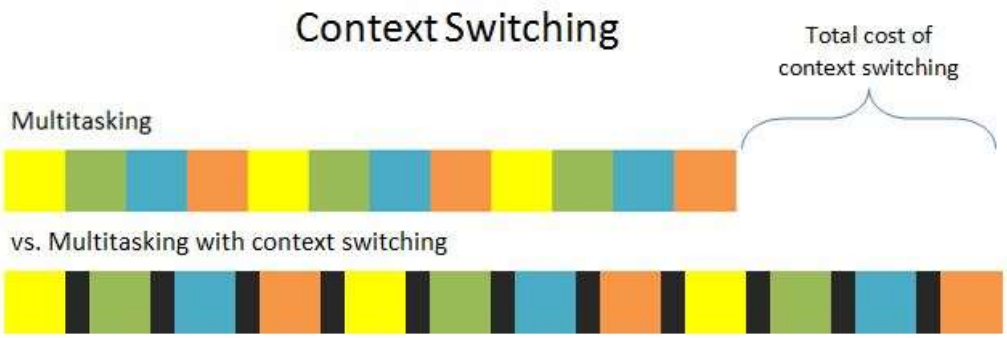

Figura 8: *Cambio de contexto*

- Disminuye la memoria utilizada: Si se utilizan hilos, cada uno tendrá su propia pila de variables, con memoria asignada a ella. En la programación asíncrona se utiliza tan solo una pila con la reducción de memoria necesaria que esto conlleva.

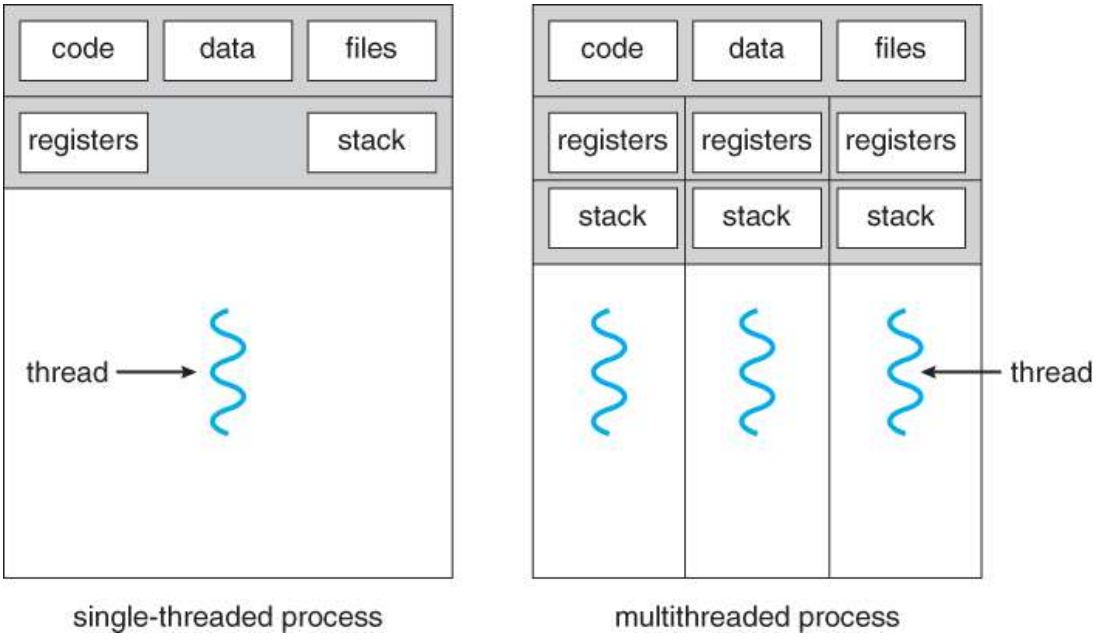

Figura 9: *Hilos*

En la programación asíncrona, tan solo se utiliza un núcleo del procesador por lo que, en tareas muy intensivas computacionalmente, la utilización de esta técnica supone un descenso del rendimiento global en la aplicación.

Sin embargo, en tareas donde el porcentaje de carga computacional es bajo, y la entrada y salida de datos es alta (como el acceso a recursos en memoria, o las peticiones a un servidor), esta técnica mejora sustancialmente el rendimiento global.

En el caso de este trabajo, al ser las agentes aplicaciones muy basados en la mensajería, tienen una gran cantidad de entrada/salida y se ven muy beneficiados del modelo de programación asíncrona.

2.5.1.2. Programación asíncrona en Python

En los últimos tiempos, la programación asíncrona en Python se ha vuelto más y más popular. Existen diferentes librerías para realizar programación asíncrona, como Tornado[7] o Gevent[8], siendo las más populares en versiones de Python anteriores a la 3.4.

A partir de la versión 3.4 se incluye una nueva librería a Python llamada ASYNCIO, diseñado para simplificar código asíncrono y hacerlo casi tan leíble como el código secuencial.

El concepto de ASYNCIO es simple: existe un bucle de eventos en el que podemos incluir tareas. Estas tareas se ejecutan secuencialmente hasta que se encuentran con una instrucción bloqueante, como una operación de entrada y salida o un sleep. En este momento, el bucle pasa a la siguiente tarea del bucle y la ejecuta como la primera, así indefinidamente.

Para ello, ASYNCIO utiliza las corrutinas, las cuales son subrutinas que se pueden pausar y resumir. Estas corrutinas tiene una sintaxis especifica en Python:

Para definir una función como corrutina se puede realizar de dos maneras distintas:

- Añadir el decorador @asyncio.coroutine a la función:

```
@asyncio.coroutine
def function():
```

- Añadir la palabra reservada async antes de la definición de la función (solo a partir de Python 3.6):

```
async def function():
```

[7] http://www.tornadoweb.org/en/stable/index.html
[8] http://www.gevent.org/

Para cambiar de contexto, también existen dos diferentes formas:

- Utilizar las palabras reservadas yield from.

```
result = yield from function()
```

- Utilizar la palabra reservada await (solo a partir de Python 3.6).

```
result = await function()
```

Es decir, cuando se invoca un yield from o un await (en adelante siempre await), la función principal se detendrá y se cambiará de contexto para ejecutar la corrutina definida a la derecha del await, en este caso function().

Cuando se termina una corrutina, se lanza la excepción StopIteration, y el atributo valor de la excepción contiene el valor de retorno, el cual se propaga por el iterador y se asigna a la variable indicada.

En ASYNCIO, por tanto, un bucle de eventos funcionaria de la siguiente manera:

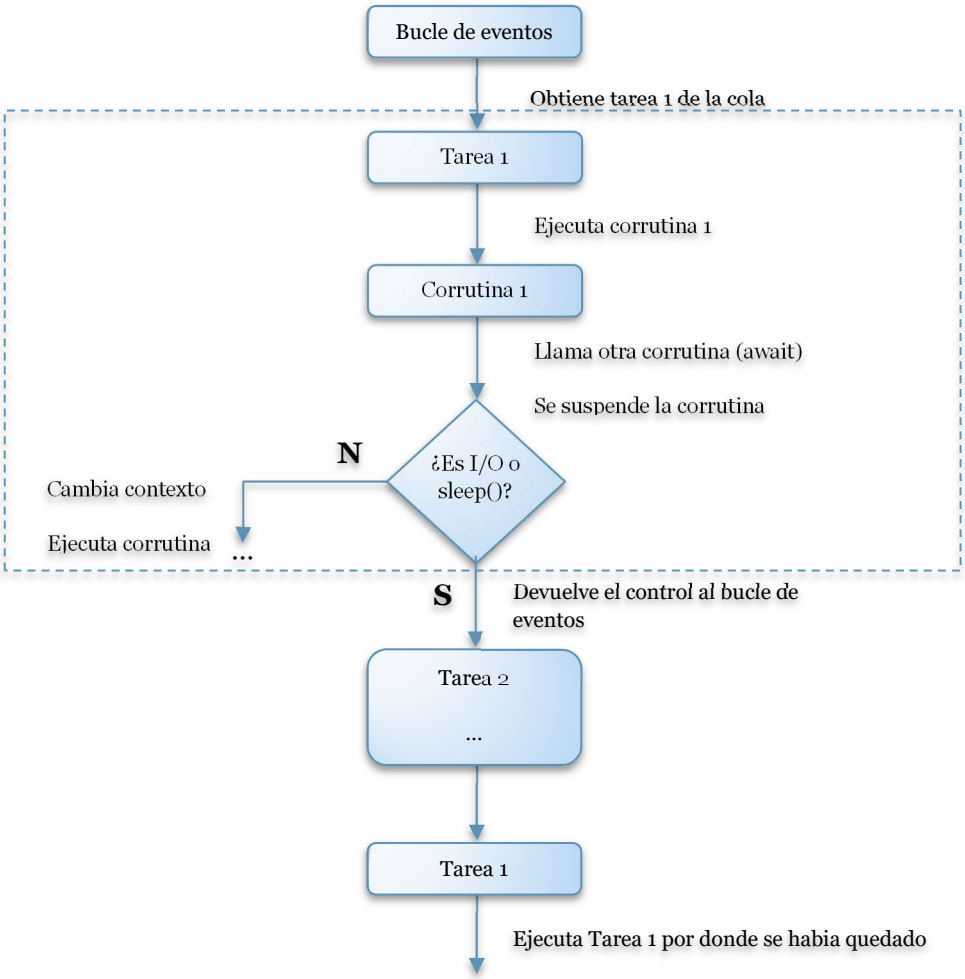

Figura 10: *Flujo bucle de eventos*

Como se puede observar, el bucle de eventos comenzaría ejecutando la primera tarea programada. Si la corrutina llama a otra corrutina (await …), la corrutina actual se suspende y se cambia de contexto a la nueva corrutina. Esto se puede anidar tantas veces como se quiera hasta que la corrutina invocada es una operación de entrada/salida o un sleep, en cuyo caso la corrutina se suspende y se devuelve el control al bucle de eventos, que comenzaría con la ejecución de la segunda tarea programada.

Esto permite ejecutar otras instrucciones mientras se realizan operaciones bloqueantes y a partir de una ejecución secuencial surge una ejecución concurrente sin apenas variar el código original.

2.5.2. AIOXMPP

AIOXMPP es una librería Python que implementa el protocolo XMPP usando la librería ASYNCIO, por lo que es muy buen complemento al usarse en un proyecto donde también se utiliza ASYNCIO.

Implementa el protocolo XMPP a nivel de cliente, pero no a nivel de servidor. Esto significa que, si en alguna especificación del protocolo se define el mensaje que debe comunicar el cliente y la respuesta que debe dar el servidor, esta librería tan solo implementará la parte correspondiente al cliente.

Es un proyecto de software libre, que permite aplicar la concurrencia a la mensajería instantánea, aplicando múltiples XEPs (XMPP Extension Proposal), securización TLS para las conexiones creadas y validación de certificados.

Esta librería puede dividirse entre tres partes:

- El ‘núcleo’ formado por un número reducido de clases que implementan el protocolo XMPP sin adiciones.
- Los ‘servicios’ formado por un número mayor de clases que implementan diversos XEPs.
- Los ‘útiles’ formado por clases que sirven para implementar métodos útiles como la gestión de excepciones.

Aunque la librería sea completamente funcional, es una librería en pleno desarrollo que continúa añadiendo nuevos XEPs. Cabe destacar que, a pesar de estar en desarrollo, el núcleo del protocolo está libre de errores y completamente probado mientras que, en total, la cobertura de tests unitarios de todo el proyecto es superior al 98%,

Se trata, por tanto, de una librería de completa actualidad y que utiliza las últimas tecnologías disponibles.

2.6. Crítica al estado del arte

Aunque existe una primera aproximación funcional al objetivo de este trabajo, esta primera aproximación es extremadamente ineficiente, aumentando enormemente los tiempos de procesamiento e imposibilitando la rápida simulación por parte del alumno.

Utiliza, además, una versión muy antigua del software SPADE, que necesita una clara actualización, así como la modernización de los métodos utilizados en su construcción.

A continuación, se expondrán las deficiencias de cada una de las capas en las que hemos dividido el sistema, a partir de la primera aproximación de la nueva arquitectura, el cual utilizaremos como punto de partida para desarrollar una nueva arquitectura.

2.6.1. PGOMAS

El principal problema con PGOMAS es su inexistente soporte a ningún tipo de concurrencia. PGOMAS es una copia directa de las antiguas clases de Java, y en Java, la concurrencia era controlada completamente por JADE, por lo que no hay rastros de concurrencia en la capa de aplicación.

Además, esta capa está programada con Python 2, el cual dejará de ser soportado próximamente (1 de enero de 2020), por lo que dentro de poco tiempo Python 2 será considerado un lenguaje antiguo en favor de su sucesor Python 3. Este punto es crítico, ya que el objetivo de este trabajo es que los alumnos usen un lenguaje de programación actual para realizar las prácticas de la asignatura.

2.6.2. SPADE v1

Al igual que PGOMAS, esta capa está programada con Python 2 y no tiene ningún soporte para la concurrencia de los agentes, lo que provoca que la ejecución de varios agentes simultáneos sea muy lenta.

2.6.3. XMPP

El módulo xmpp de Python se encarga de las comunicaciones entre agentes, pero al igual que las capas superiores, no implementa asincronía entre estas comunicaciones, por lo que el envío de mensajes entre agentes se vuelve una tarea bloqueante.

2.6.4. SPADE v3

A pesar de que esta plataforma ha implementado asincronía utilizando Python 3 y ASYNCIO, no ha sido probada exhaustivamente y contiene algunos fallos de programación, puesto que es una versión no finalizada de la plataforma actual.

Además, a pesar de estar implementando un sistema de comunicación con AIOXMPP, no tiene implementado un sistema de autorregistro con el servidor, algo muy útil cuando hablamos de agentes que pueden crearse y desaparecer muy rápidamente en un entorno dinámico como puede ser JGOMAS.

Es por ello que será necesario en este trabajo aportar el código necesario para finalizar la versión 3 de SPADE, otorgándole específicamente de soporte para auto-registrar a los agentes que se conecten a la plataforma. Además, se ha detectado durante este análisis, que el soporte de autorregistro (In-Band Registration) no está incluido en la biblioteca AIOXMPP, por lo que será necesario mejorarla para incluir este soporte, lo cual implica entrar en contacto con el desarrollador de la biblioteca y realizar una aportación al proyecto (que es software libre) siguiendo los mecanismos establecidos por el líder del proyecto.

2.7. Propuesta

La propuesta que se plantea en este trabajo implica desarrollar una arquitectura que se asemeje a la siguiente figura:

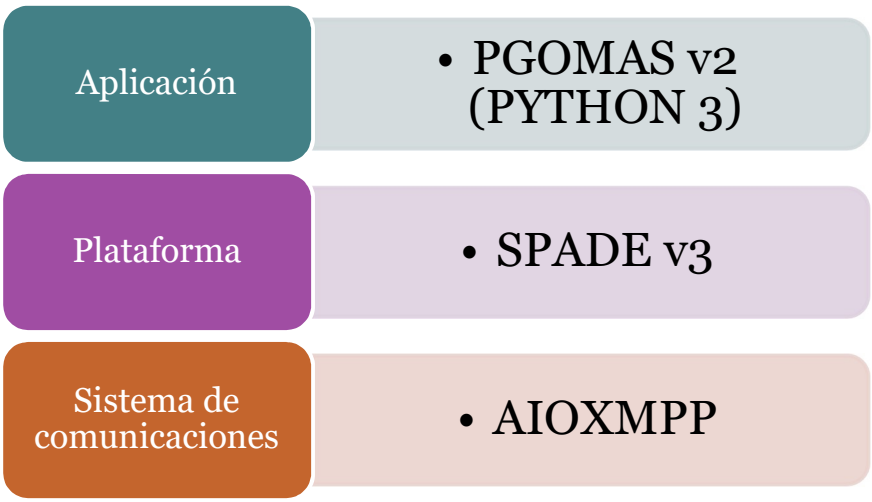

Figura 11: *Arquitectura propuesta*

Esta nueva arquitectura implica un PGOMAS programado en Python 3, con soporte a la concurrencia entre agentes, utilizando la plataforma SPADE v3 como gestor de agentes y basándose en el protocolo de comunicación XMPP a través de la librería AIOXMPP.

Para esta propuesta será necesario realizar diversas tareas en las 3 capas:

- Añadir soporte de auto-registro en el proyecto de software libre AIOXMPP.
- Finalizar la v3 de SPADE, añadiendo dicho soporte de autorregistro.
- Escribir la v2 de PGOMAS con Python3 y SPADE v3.

No se debe olvidar que, aunque no se modifique nada del visor gráfico, este seguirá funcionando ya que la nueva arquitectura será compatible con el visor.

3. Análisis del problema

En este capítulo se analizarán cuáles son los requisitos necesarios para conseguir cumplir los objetivos marcados.

En primer lugar, se definirán los requisitos que debería cumplir la plataforma SPADE, y se analizarán el estado de dichos requisitos. Se entrará en más detalle en uno de los requisitos no cumplidos de la plataforma SPADE, y se listarán las especificaciones de dicho requisito.

A continuación, se definirán los requisitos de la aplicación PGOMAS, extrayendo las características de la aplicación actual JGOMAS, para que la migración a la nueva plataforma sea completa y satisfactoria.

También se analizarán el tipo de pruebas necesarias para determinar que el desarrollo realizado es correcto.

Se revisarán las licencias de los recursos utilizados, para comprobar que no habrá ningún tipo de problema legal ni durante ni al finalizar el desarrollo.

Se analizarán las posibles soluciones que se han detectado y se elegirá una de ellas, exponiendo los motivos detrás de esa elección.

Finalmente, se realizará un presupuesto del trabajo a realizar.

3.1. Requisitos SPADE

Siendo lo más minimalista posible, podemos concebir un agente como una entidad que tiene comportamientos y que puede intercambiar mensajes con otros agentes. Este es el planteamiento principal de la plataforma SPADEv3, que busca implementar los conceptos fundamentales de un agente, en pos de la simplicidad.

Por tanto, según la definición de agente propuesta, un Agente SPADE:

- Debe poder conectarse a un servidor XMPP. Al tratarse de un entorno tan dinámico, significa no solo que el agente se autentique con una cuenta ya creada, sino que debería poder registrarse automáticamente en el servidor.

- Debe poder gestionar comportamientos, es decir, añadir nuevos comportamientos, eliminar comportamientos existentes y consultar si un comportamiento existe o no.

La parte de envío y recepción de mensajes se incluirá dentro de los comportamientos, por lo que un comportamiento SPADE:

- Debe poder ejecutar métodos que alteren su estado interno.
- Debe poder enviar y recibir mensajes de otros agentes y actuar en consecuencia.
- Dependiendo del tipo de comportamiento, éste se comportará de una forma u otra. En concreto, existen los siguientes tipos de comportamiento (Behaviour):
    - OneShotBehaviour: El comportamiento se ejecutará una vez inmediatamente en el momento de crearse.
    - PeriodicBehaviour: El comportamiento se ejecutará periódicamente cada cierto intervalo. Si deseamos que el comportamiento se vuelva a ejecutar instantáneamente al acabar, el intervalo será cero.
    - TimeoutBehaviour: El comportamiento se ejecutará una sola vez pasado un cierto tiempo desde su creación.
    - FSMBehaviour: Es un comportamiento que simula una Máquina Finita de Estados. Cada vez que se ejecuta este comportamiento, éste ejecutará el método principal del comportamiento, para luego pasar al siguiente estado según los resultados obtenidos en el método ejecutado.

Estas son, a grandes rasgos, las funcionalidades que definen la plataforma SPADE.

3.2. Estado requisitos SPADE

Analizando el código actual de SPADEv3, podemos observar cómo, aunque prácticamente todas las funcionalidades requeridas se cumplen en esta versión, hay una que no está implementada. Esta se trata de la de permitir el autorregistro de los agentes. Es decir, actualmente los agentes son capaces de conectarse a un servidor XMPP siempre y cuando éstos estén previamente registrados en dicho servidor.

En el caso objetivo del trabajo, en la aplicación de la simulación de JGOMAS, esto podría no ser un problema muy crítico si los únicos agentes implicados en el desarrollo del

juego fueran las tropas de élite y el agente mánager, ya que son agentes que no varían a lo largo de la partida y su número es limitado.

Sin embargo, en JGOMAS, y también en PGOMAS, los objetos soltados por las tropas de élite, como las cajas de salud o munición, también son considerados como agentes, y esto hace crecer considerablemente la cantidad de agentes a tratar, lo que provoca que el autorregistro de los agentes en el servidor no sea recomendable, sino esencial.

3.3. In-Band Registration

Analizando las especificaciones del protocolo XMPP, observamos que existe un XEP que permite el autorregistro de usuarios a un servidor XMPP. En concreto, se trata del XEP0077, titulado In-Band Registration [15].

Este XEP especifica el formato de los mensajes XML que el cliente debe mandar al servidor, así como como las respuestas del servidor, para permitir el registro, la modificación y la cancelación de cuentas en el servidor.

En una primera instancia, se comprobó si en la biblioteca que se está utilizando para implementar el protocolo XMPP, se encuentra implementado también este XEP, pero se observa que aún no está implementado en dicha biblioteca. Por lo que se hace necesario el desarrollo de dicha funcionalidad en este trabajo, lo cual supone un subobjetivo adicional añadido.

Se debe, por tanto, definir las especificaciones del registro automático, adaptándolas a las necesidades del proyecto para tenerlas en cuenta a la hora del desarrollo.

El registro automático:

- Debe poder establecer una conexión con el servidor sin llegar a autenticarse, ya que no tiene sentido autenticarse antes de crear una cuenta.
- Debe poder ignorar la verificación de certificado, ya que normalmente se trabajará en un servidor local sin ningún tipo de certificado.
- Debe ser capaz de enviar los mensajes de registro definidos en el XEP0077 y recibir las respuestas a estos mensajes.

3.4. Requisitos PGOMAS

Una vez definidos los requisitos que debe cumplir la plataforma SPADE para estar completa, podemos pasar a las especificaciones que debe cumplir la nueva aplicación PGOMASv2.

En concreto, la simulación obtenida durante la ejecución de PGOMASv2:

- Debe cumplir con las reglas del juego definidas en la introducción de este trabajo, las cuales no se van a repetir por duplicidad.
- Debe permitir su ejecución a una velocidad que no ralentice al alumno en la realización de las prácticas asociadas a esta aplicación. En concreto, debería poder ejecutarse a una velocidad similar a la velocidad actual de la simulación con la arquitectura actual.
- Debe estar programado en el lenguaje de programación Python.
- Debe estar programado sobre la plataforma SPADE, es decir, los agentes de PGOMAS heredarán de los agentes de SPADE.
- Debe permitir la fácil inclusión de nuevos elementos, para que en el momento que los alumnos deban programar los agentes con comportamientos complejos, no desfallezcan tratando de añadir nuevos comportamientos al sistema.

3.5. Requisitos pruebas

Con el fin de validar el desarrollo y comprobar que el código escrito es claro y legible, se definirán las pruebas necesarias que deberán pasarse para este fin.

Por un lado, se pretende mejorar la legibilidad del código, por lo que el código desarrollado durante el transcurso del trabajo se basará en la guía de estilo para código Python PEP 8 [16]. Este documento proporciona una serie de convenciones a la hora de escribir código Python, con el fin de mejorar la legibilidad del código y hacerlo consistente a través de su amplio espectro en la comunidad Python. Para ello, se buscará una total aplicación de las reglas de estilo definidas en este documento.

Por otro lado, se pretende asegurar que la funcionalidad del código es la correcta. Para ello, será necesario desarrollar una serie de pruebas unitarias y de integración que validen el resultado de los métodos desarrollados. En este caso, se buscará una cobertura del 100% del código desarrollado, es decir, las pruebas unitarias verificarán

que el flujo del programa recorre todas las líneas de código de cada método y que el resultado es correcto.

3.6. Análisis del marco legal y ético

Antes de empezar cualquier tipo de desarrollo, es necesario tener en cuenta diversos aspectos legales para evitar cualquier tipo de sanción. En concreto, en este caso será necesario analizar tanto los derechos del software utilizado, como los derechos de los recursos y bibliotecas empleados.

3.6.1. Software utilizado

Este punto puede no tener demasiada importancia en al ámbito personal o académico, sin embargo, dentro del ámbito de una empresa, no tener en cuenta los derechos del software utilizado puede acarrear graves sanciones económicas.

En el caso de este trabajo, se han utilizado dos elementos software durante el desarrollo del trabajo:

- Por un lado, tenemos el entorno de desarrollo integrado (IDE) para Python PyCharm[9], el cual cuenta con una distribución open-source con licencia 'Apache License Version 2'. Esta licencia permite el uso del software tanto comercialmente como no comercialmente.
- Por otro lado, tenemos el servidor XMPP utilizado para realizar pruebas, Prosody IM[10], que cuenta con la licencia 'MIT License'. Al igual que antes, esta licencia también permite el uso del software bajo cualquier circunstancia.

3.6.2. Bibliotecas utilizadas

Así como el análisis del software utilizado es relevante durante el desarrollo, el análisis de las bibliotecas utilizadas se vuelve relevante después del desarrollo, cuando se publican o distribuyen, pues es cuando se tiene en cuenta, por ejemplo, si una librería es de uso privado o si solamente puede usarse en caso de que la distribución sea open-source.

En este caso hay que tener en cuenta varios elementos:

[9] https://www.jetbrains.com/pycharm/
[10] https://prosody.im/

- El protocolo de mensajería instantánea XMPP es un protocolo gratis y abierto, por lo que puede usarse para cualquier fin.
- La biblioteca ASYNCIO pertenece a Python Software Foundation, el cual distribuye todas las bibliotecas estándar Python, que son open-source y usables para cualquier fin.
- La biblioteca AIOXMPP posee la licencia LGPLv3 (GNU Lesser General Public License v3.0) la cual, al igual que todas las demás, también permite su uso para cualquier fin.

Tras analizar los recursos utilizados, llegamos a la conclusión de que todas las bibliotecas y protocolos permiten cualquier tipo de uso y, por tanto, no habrá ningún tipo de problema con su empleo.

3.7. Identificación y análisis de soluciones posibles

Para el desarrollo de este trabajo, se han identificado dos elementos principales sobre los que se deben tomar decisiones respecto a cómo se implementarán. Por un lado, es necesario definir cómo se va a implementar el registro automático en la plataforma SPADE. Por otro lado, es necesario determinar cómo se desarrollará la aplicación PGOMASv2.

3.7.1. Posibles soluciones registro automático

En el caso de la implementación del registro automático, surgen varias posibles alternativas:

- Desarrollar el registro automático dentro de la propia librería SPADE, implementando las funcionalidades descritas el en apartado de In-Band Registration.
- Desarrollar una pequeña biblioteca aparte para implementar el registro automático y al igual que en punto anterior, implementando las funcionalidades descritas.
- Modificar la biblioteca existente de AIOXMPP para adaptarse a este nuevo XEP, siguiendo las convenciones utilizadas en la biblioteca y contribuyendo al software libre. A continuación, modificar la plataforma SPADE para usar esta nueva funcionalidad.

Las desventajas de las dos primeras opciones son claras. No solo es necesario implementar el intercambio de mensajes XML, sino también la conexión cifrada con el servidor y la verificación de certificados, aumentando tanto la complejidad de la plataforma y añadiendo numerosas bibliotecas como dependencias.

Esto no ocurre con la tercera opción, quien ya tiene implementada las funcionalidades de conexión con el servidor y la verificación de certificados, y tan solo serían necesarias pequeñas modificaciones para añadir la ausencia de autenticación en la conexión.

Sin embargo, la tercera opción tiene como principal desventaja que, al ser una biblioteca pública, si se desea aportar modificaciones de código a dicha biblioteca, no se puede implementar tan solo la parte correspondiente al registro de nuevos usuarios, sino que sería necesario implementar todo el XEP077 para que se diera por bueno el desarrollo y sea aceptada por el equipo de desarrollo de la biblioteca. Además, será necesario seguir los estándares de calidad exigidos por dicho equipo.

3.7.2. Posibles soluciones desarrollo PGOMASv2

Al igual que en el anterior apartado, surgen diversas alternativas a la hora de decidir cómo implementar la aplicación PGOMASv2:

- Extraer la funcionalidad a partir de la arquitectura actual en JGOMAS, obteniendo la funcionalidad de las tropas de élite a partir de las clases JASON, y la funcionalidad del mánager y de los objetos de las clases java dentro de la librería jgomas.jar. A continuación, desarrollar desde cero la aplicación aplicando las funcionalidades extraídas usando la concurrencia con ASYNCIO y AIOXMPP.
- Migrar la funcionalidad a partir de PGOMASv1, depurando las clases, añadiendo soporte asíncrono con ASYNCIO y AIOXMPP, eliminando comportamientos innecesarios en esa nueva arquitectura y aplicando un estilo más limpio, es decir, mejorando la calidad del código.

La principal ventaja de la primera opción es que, al no tener referencias en Python sobre el estilo de programación, no existen influencias a la hora de desarrollar y se puede llegar a las mismas funcionalidades aplicando una aproximación completamente distinta y más clara. Pero, aunque eso sea su principal ventaja, también puede ser su principal desventaja, en el que la aproximación que se realice tenga menor claridad de código y sea más ineficiente.

También cabe destacar el tiempo de desarrollo que ambas opciones supondrían, siendo la primera de ellas mucho mayor y seguramente saliéndose fuera de los límites del trabajo.

3.8. Solución propuesta

Tras valorar las ventajas y desventajas de cada una de las alternativas para los dos elementos de decisión, se han tomado las siguientes decisiones:

- Se modificará la librería AIOXMPP para añadir el soporte al XEP0077, para luego modificar SPADE y añadir dicha funcionalidad a la plataforma.
- Se migrará la funcionalidad a partir de PGOMASv1, aplicando todas las mejoras descritas.

Con esto, se pretende tanto minimizar el tiempo de desarrollo, como mantener la simplicidad de SPADE, así como evitar riesgos de ineficiencia a la hora de implementar PGOMASv2.

3.9. Presupuesto

3.9.1. Licencias de software

Para llevar a cabo el proyecto se han utilizado diferentes programas informáticos, así como las bibliotecas Python y el sistema operativo utilizado.

Software	Unidades	Precio Unitario (€/ud)	Coste
Ubuntu 16.04 LTS	1	Licencia gratuita	0,00€
PyCharm Community	1	Licencia gratuita	0,00€
Servidor XMPP Prosody IM	1	Licencia gratuita	0,00€
Biblioteca ASYNCIO	1	Licencia gratuita	0,00€
Biblioteca AIOXMPP	1	Licencia gratuita	0,00€
		Total	0,00€

Figura 12: *Licencias de software*

3.9.2. Equipo informático

El ordenador utilizado tendrá las siguientes características:

- Sistema operativo Ubuntu 16.04 LTS
- Procesador Intel Core i5 4670K 3.40Ghz
- Memoria RAM 8.00GB DDR3 1199MHz
- Placa base ASUSTeK COMPUTER INC. Z97-K (SOCKET 1150)
- Gráfica 2047MB NVIDIA GeForce GTX 660 (Gigabyte)
- Pantalla BenQ GL2250H
- Almacenamiento 119GB SanDisk SDSSDHP128G

Se considerará un periodo de amortización de 3 años, con una utilización media de 8 horas diarias durante 240 días al año (1920 horas al año).

Equipo	Uds.	Horas / año	Horas de trabajo	Precio unitario (€/ud)	Amortización	Coste (€)
PC	1	1920	300	1.000	5,21%	52,08€
					Total	52,08€

Figura 13: *Equipo Informático*

3.9.3. Mano de obra

Concepto	Tiempo (h)	Precio unitario (€/h)	Coste (€)
Instalación entorno	15	30	450€
Implementación XEP077			
Análisis	45	30	1.350€
Implementación	30	30	900€
Pruebas	30	30	900€
Actualización SPADE			

Análisis	15	30	450€
Implementación	15	30	450€
Pruebas	15	30	450€
Traducción de JGOMAS			
Análisis	30	30	900€
Implementación	75	30	2.250€
Pruebas	30	30	900€
		TOTAL	9.000€

Figura 14: *Mano de obra*

3.9.4. Coste total

Partida	**Coste (€)**
Licencias de software	0,00€
Equipo informático	52,08€
Mano de obra	9.000,00€
Subtotal	9.052,08€
Beneficio industrial (6%)	543,12€
IVA (21%)	1.900,94€
TOTAL	11.496,14€

Figura 15: *Coste Total*

4. Diseño de la solución

Una vez identificados los requisitos del sistema, es necesario diseñar cual será la forma final de la solución propuesta, puesto que una buena organización es esencial si el objetivo es crear código legible y bien organizado.

En este capítulo, se definirá la arquitectura final del sistema, así como el diagrama de clases de cada capa de la arquitectura.

A continuación, se profundizará en los detalles de implementación en cada capa, desde la capa más baja, donde se definirá el XEP a implementar y como se implementará, hasta la capa más alta en la que se definirá el comportamiento de cada agente en PGOMAS, así como los mensajes que se intercambian entre ellos.

4.1. Arquitectura del Sistema

Tal y como se ha hablado en el capítulo de Estado del arte, la arquitectura propuesta tendrá un aspecto de este estilo:

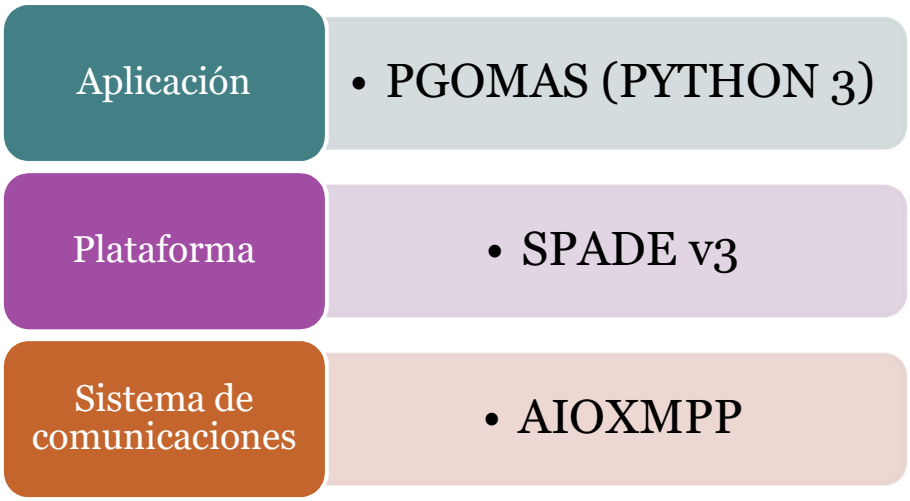

Figura 16: *Arquitectura final*

Esta arquitectura propuesta no variará, por lo que esta es la arquitectura que se mantendrá en el diseño final.

A continuación, se expondrá un diseño más detallado de cada capa. En cada una de ellas se mostrará su diagrama de clases de la manera que más ayude a comprender su estructura.

4.1.1. Diseño de clases AIOXMPP

En la biblioteca AIOXMPP existen muchas clases, la mayoría de las cuales no son de especial relevancia para el desarrollo del trabajo.

Sin embargo, será interesante realizar un esquema sobre la estructura que sigue la biblioteca y como se dividen las diversas clases.

Dividimos las clases de la biblioteca en dos grandes grupos:

- Las clases del núcleo de la AIOXMPP, entre las que se incluyen tanto las clases principales de la biblioteca que implementan el protocolo XMPP, como las clases con métodos que sirven de utilidad.
- Las clases que implementan diversos XEPs y que se agrupan formando servicios, uno por cada implementación de XEP, los cuales se entrará en detalle más adelante.

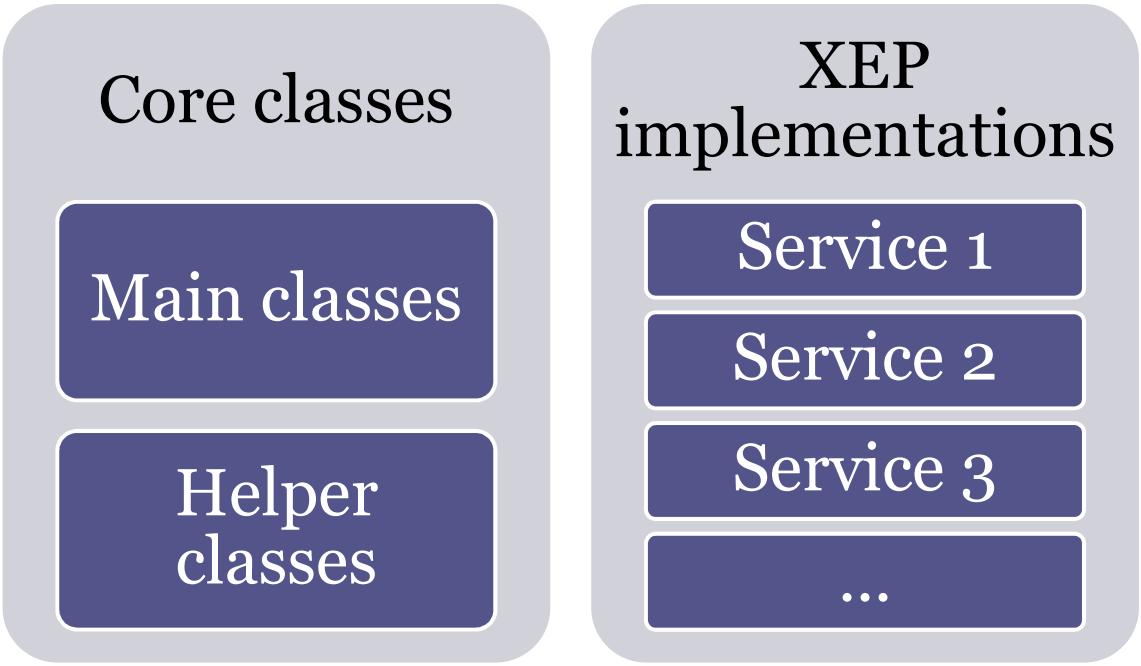

Figura 17: *Estructura clases AIOXMPP*

Nuestro objetivo en esta capa será la implementación de un nuevo XEP, el 'XEP077 In-Band Registration', en forma de servicio, siguiendo la estructura y convenciones propios de AIOXMPP.

4.1.2. Diseño de clases de SPADE

SPADE está formado tan solo por seis clases separadas, una de las cuales (la interfaz web) aún no está terminada, por lo que ya no existe interés en clasificar las clases por su tipo, sino que se hace más interesante estructurar las clases según las relaciones que tienen entre ellas. Es decir, utilizando el siguiente diagrama:

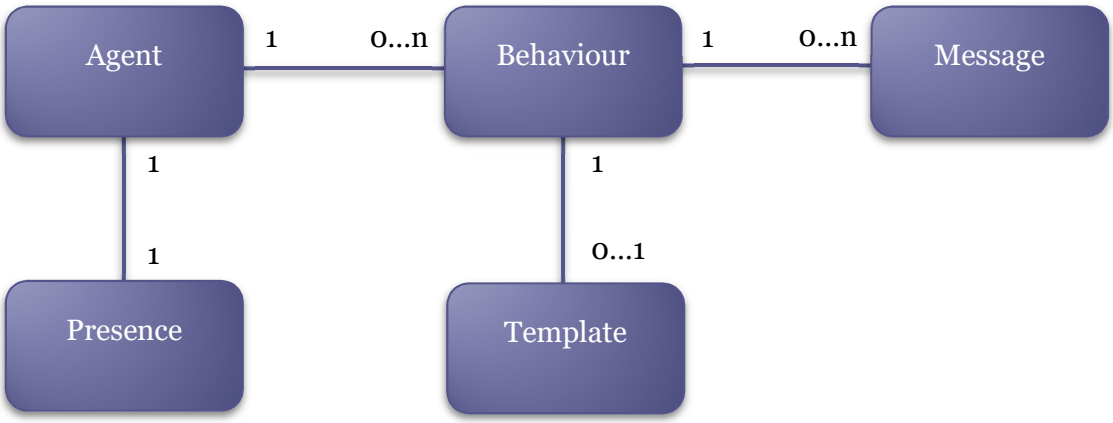

Figura 18: *Estructura clases SPADE*

Como se puede observar, en este diagrama no se incluye la clase web.py, ya que aún no está finalizada. Además, tampoco es de especial interés para el desarrollo de este proyecto porque no se usa en ningún sitio dentro de la aplicación PGOMASv2.

La clase Agent es la clase base de la plataforma, que contendrá instancias de las demás clases.

Por cada Agent existe un objeto Presence, que gestiona las notificaciones de presencia de los agentes.

Por cada Agent puede haber un número indeterminado de comportamientos, en los que se definirán las acciones que puede ejecutar el agente.

Cada comportamiento puede tener o no un Template asociado, el cual sirve como filtro para determinar si un mensaje que recibe el agente va dirigido a ese comportamiento o no.

Y, por último, y aunque la relación no es exacta, puesto que un comportamiento, no contiene mensajes como tal, sí que puede enviar múltiples mensajes a otros agentes.

4.1.3. Arquitectura de PGOMASv2

La distribución de clases en PGOMASv2 se asemeja en gran medida a la estructura de PGOMASv1, ya que según se ha determinado en el "Análisis del problema", el objetivo es mantener la estructura ya existente depurando en lo posible el código actual.

Para este caso, debido a que el aspecto más interesante de las clases de PGOMAS es su distribución a nivel de herencia de objetos, el diagrama de clases estará centrado en las relaciones de herencia entre las clases:

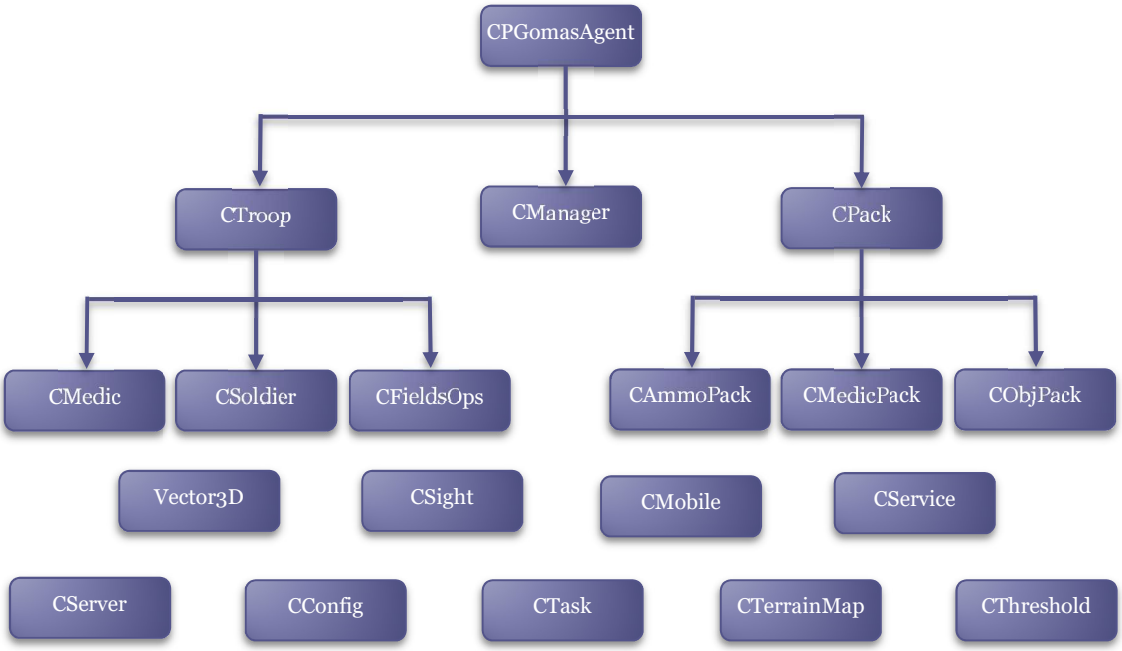

Figura 19: *Estructura clases PGOMASv2*

Se observa cómo ha desaparecido alguna clase auxiliar que se hace innecesaria con la nueva plataforma, como es la clase CService, y también han desaparecido otra clase que no tenía demasiado sentido, como la clase CBasicTroop, ya que las clases finales que se utilizarán son las más específicas. Esta clase se ha unificado con la clase CTroop, agrupando los comportamientos.

Observamos como las clases CTroop, CManager y CPack heredan de la clase CPGomasAgent, la cual heredará a su vez de la clase Agent de SPADE.

A su vez, tanto las clases CTroop como CPack son heredadas por otras subclases más específicas que implementarán los comportamientos propios de cada tipo de tropa y de cada tipo de objetos respectivamente, tal y como veremos más adelante.

Por último, las demás clases restantes son clases de utilidad y son utilizadas por CManager.

4.2. Diseño Detallado

Al estar trabajando con múltiples capas de la arquitectura, el procedimiento a seguir será el de definir de abajo a arriba las actuaciones que deben seguirse en cada capa, empezando por la librería AIOXMPP, siguiendo con la plataforma SPADE, para finalizar con la aplicación PGOMASv2.

4.2.1. AIOXMPP

La librería AIOXMPP no solo implementará el protocolo XMPP base, sino que también contiene la implementación de múltiples XEPs, que añaden extensiones al protocolo opcionales.

Estas funcionalidades se pueden dividir en dos grupos genéricos:

Por una parte, están aquellos que se integran de forma nativa y que están siempre disponibles, o que dan soporte a algún tipo de datos, como por ejemplo el 'XEP0198 Stream Management' que proporciona soporte a la gestión de las secuencias XML de forma activa, o el 'XEP0004 Data Forms' que proporciona soporte a un nuevo tipo de estructura de datos mediante formulario.

Este tipo de XEP se encuentra siempre disponible y es decisión del usuario el usarlo o no. Se puede, por ejemplo, definir un objeto Data Form en el momento que desees sin ningún tipo de acción anterior.

Por otra parte, nos encontramos con los XEPs que solamente estarán disponibles en caso de que los invoques antes de conectarse al servidor. Estos XEPs son considerados en AIOXMPP como 'Servicios'.

La idea de los Servicios de AIOXMPP es la de implementar un XEP específico, pero encapsulados en una clase separada, lo que permite al usuario elegir cuales son los XEPs que desea implementar en su cliente con facilidad.

Algunos ejemplos de estos servicios serian el 'XEP0084 User Avatar', que permite el uso de avatares de usuario, o el 'XEP0199 XMPP Ping', que permite tanto el envío de pings a nivel de aplicación, como la respuesta a estos por parte de otro cliente.

La intención original de este trabajo es que el 'XEP0077 In-Band Registration' sea un servicio dentro de AIOXMPP.

4.2.2. Especificación XEP0077

El 'XEP0077 In-Band Registration' está especificada de forma canónica en la página web oficial de XMPP [15], que es el lugar donde se publican las especificaciones de estas extensiones al protocolo XMPP. En esta especificación, se definen una serie de mensajes XML que el cliente envía al servidor, y las respuestas que deben devolver los servidores.

En concreto, se definen una serie de mensajes que permiten registrar, modificar, y borrar un usuario de un servidor. A continuación, describiremos brevemente los casos de uso en los que se divide esta especificación.

4.2.2.1. Entidad se registra con un servidor

Para saber cuáles son los campos necesarios para el registro con un servidor, la entidad debería enviar primero un mensaje de la forma:

```
<iq type='get' id='reg1' to='shakespeare.lit'>
  <query xmlns='jabber:iq:register'/>
</iq>
```

Si la entidad no está conectada al servidor, y éste soporta el 'XEP007 In-Band Registration', el servidor contestará un mensaje informando de los campos necesarios para registrarse:

```
<iq type='result' id='reg1'>
  <query xmlns='jabber:iq:register'>
    <username/>
    <password/>
    <email/>
  </query>
</iq>
```

En el caso de que la entidad ya esté conectada al servidor, el servidor devolverá un mensaje con la información de registro actual de la entidad:

```
<iq type='result' id='reg1'>
  <query xmlns='jabber:iq:register'>
    <registered/>
    <username>juliet</username>
    <password>R0m30</password>
    <email>juliet@capulet.com</email>
  </query>
</iq>
```

Una vez se conocen cuáles son los campos necesarios para el registro, la entidad debe enviar un mensaje con la información solicitada:

```
<iq type='set' id='reg2'>
  <query xmlns='jabber:iq:register'>
    <username>bill</username>
    <password>Calliope</password>
    <email>bard@shakespeare.lit</email>
  </query>
</iq>
```

A lo que el servidor, en caso de que el registro sea correcto, responderá con un:

```
<iq type='result' id='reg2'/>
```

En caso de que el registro falle, devolverá un mensaje indicando el error de la siguiente forma:

```
<iq type='error' id='reg2'>
  <query xmlns='jabber:iq:register'>
    <username>bill</username>
    <password>m1cro$oft</password>
    <email>billg@bigcompany.com</email>
  </query>
  <error code='409' type='cancel'>
    <conflict xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

El tipo de error puede variar según el origen de éste. Puede ser porque la entidad no ha proporcionado toda la información necesaria, o porque ya existe un usuario con ese nombre de usuario, por ejemplo.

Los campos que se pueden solicitar en el registro son limitados y predefinidos, al menos en su implementación básica:

```
<iq type='result' id='reg1'>
  <query xmlns='jabber:iq:register'>
    <username/>
    <nick/>
    <password/>
    <name/>

    <first/>
    <last/>
    <email/>
```

```
    <address/>
    <city/>
    <state/>

    <zip/>
    <phone/>
    <url/>

    <date/>
    <misc/>
    <text/>

    <key/>
  </query>
</iq>
```

Opcionalmente, se puede ampliar el XEP añadiendo soporte a los Data Forms, que permiten solicitar cualquier campo deseado, aunque esto queda fuera de los límites de este trabajo.

4.2.2.2. Entidad cancela registro existente

En el caso que se desee cancelar un registro con un servidor XMPP, el cliente deberá estar conectado al servidor y enviar un mensaje de la forma:

```
<iq type='set' from='bill@shakespeare.lit/globe' id='unreg1'>
  <query xmlns='jabber:iq:register'>
    <remove/>
  </query>
</iq>
```

En caso de que se haya cancelado correctamente la cuenta, el servidor devolverá:

```
<iq type='result' to='bill@shakespeare.lit/globe' id='unreg1'/>
```

Al igual que pasaba con el registro, en el caso que la cancelación haya fallado, se devolverá un mensaje indicando el tipo de error:

```
<iq type='error' from='shakespeare.lit'
to='bill@shakespeare.lit/globe' id='unreg1'>
  <error code='400' type='modify'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

4.2.2.3. Usuario cambia de contraseña

Por último, existe la posibilidad de cambiar la contraseña de un usuario. Para ello, el usuario debe estar conectado al servidor y enviar un mensaje de la forma:

```
<iq type='set' to='shakespeare.lit' id='change1'>
  <query xmlns='jabber:iq:register'>
    <username>bill</username>
    <password>newpass</password>
  </query>
</iq>
```

El cambio correcto de contraseña se indica de la siguiente manera:

```
<iq type='result' id='change1'/>
```

Mientras que cualquier fallo al cambiar la contraseña devolverá un error indicando su tipo:

```
<iq type='error' from='shakespeare.lit'
to='bill@shakespeare.lit/globe' id='change1'>
  <error code='400' type='modify'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

4.2.3. Diseño 'XEP0077 In-Band Registration'

Teniendo en cuenta la especificación del 'XEP0071 In-Band Registration', podríamos dividir la funcionalidad de la especificación a nivel de cliente en 5 métodos:

- **get_registration_fields():** Este método devolverá los campos necesarios que deben informarse para que el registro con el servidor sea correcto.
- **get_client_info():** Este método devolverá la información de registro de un cliente ya conectado.
- **register():** Este método registrará un usuario en un servidor.
- **change_pass():** Este método cambiará la actual contraseña de un cliente conectado.
- **cancel_registration():** Este método cancelará la cuenta actual de un cliente conectado.

Con estos cinco métodos, estarían cubiertos todos los casos de uso descritos por la especificación del XEP.

Se observa que, aunque tres de los métodos necesitan que el cliente esté conectado y autenticado (get_client_info, change_pass y cancel_registration), dos de ellos (get_registration_fields y register) no deberán estar autenticados, pues no tiene sentido que un cliente deba estar autenticado para crear una cuenta. Por tanto, dado que AIOXMPP considera en sus servicios que los clientes deben estar conectados y autenticados, no podemos crear un servicio como tal que englobe los 5 métodos descritos.

La solución planteada en este trabajo consistirá en agrupar los 3 métodos que necesiten conexión en un servicio, mientras que los otros dos métodos no estarán sujetos a ningún servicio y serán funciones del módulo.

Por tanto, para seguir con la estructura propia de AIOXMPP, en la que cada servicio está encapsulado en una carpeta, definiremos la siguiente estructura para el módulo In-Band Registration:

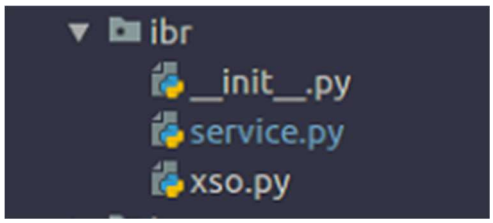

Figura 20: *Estructura módulo IBR*

En **xso.py** se definirá el objeto Query, el cual representa el mensaje a enviar al servidor dentro del <iq/>. Tendrá todos los atributos predefinidos para el registro, así como las etiquetas <registered/> y <removed/>, necesarias para recibir los mensajes de información de registro y para el envío de la cancelación.

Se añadirá también en esta clase un método llamado **get_query_xso()**, el cual creará un objeto Query a partir de un diccionario de valores, para simplificar la creación de este tipo de objeto.

En **service.py** se incluirá tanto el servicio definido anteriormente con los tres métodos que requieren conexión, como los métodos no ligados que no requieren autenticación.

También añadiremos un método auxiliar llamado **get_used_fields()**, el cual servirá para obtener dado un objeto de tipo Query, una lista de los campos que vengan informados,

a fin de facilitar dicha tarea y no tener que recorrer todos los atributos del objetos para comprobar si vienen o no informados.

Por último, el archivo **__init__.py** se usará para exportar las clases y métodos dentro de los archivos anteriores, lo que permite que se pueda importar el módulo IBR y todas sus clases fácilmente de la siguiente manera:

```
import aioxmpp.ibr
```

4.2.4. Detalle funcionamiento SPADE

La estructura de las clases en SPADE es bastante simple, ya que ese es uno de sus objetivos. Tan solo está compuesto por seis clases definidas en el capítulo de "Estado del arte". Una vez implementada la autenticación en la librería AIOXMPP, implementar el autorregistro en la plataforma SPADE es bastante sencillo.

Sin embargo, para poder desarrollar la aplicación PGOMASv2 basada en SPADE, es necesario antes conocer el detalle de funcionamiento de esta plataforma, y no solo conocer su estructura de clases.

Para ello, la explicación se basará principalmente en dos figuras que representan el funcionamiento interno de SPADE sin entrar en demasiados detalles. La primera figura muestra la estructura típica de un agente SPADE, mientras que la segunda permite seguir un ejemplo de interacción entre agentes.

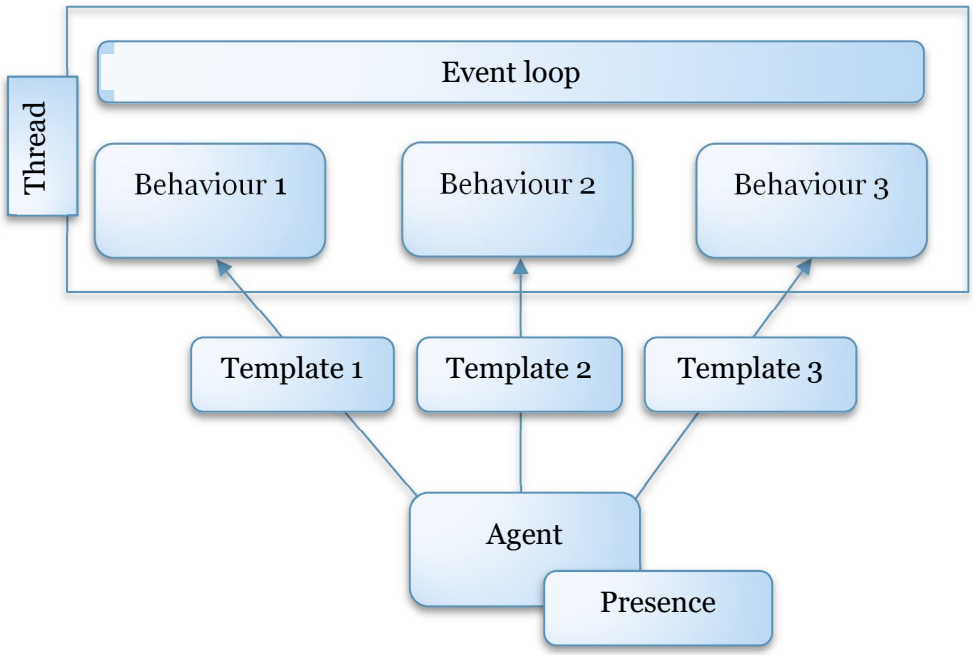

***Figura 21**: Estructura Agente SPADE*

La figura 21 muestra el esquema típico de un agente SPADE. En él, un agente tiene un objeto Presence y varios comportamientos. Estos comportamientos se encuentran encapsulados dentro de un hilo o thread, y dentro de este hilo, los comportamientos se ejecutan de forma asíncrona mediante un bucle de eventos de ASYNCIO.

Cada uno de estos comportamientos puede tener o no un Template que actúe de filtro. Un Template es un tipo de mensaje que puede hacer match con otro mensaje dependiendo de si su contenido coincide con él. Es decir, es una especie de plantilla sobre los mensajes recibidos.

Por ejemplo, un Template vacío hará match con todos los mensajes recibidos, y por tanto, los dejará entrar a todos. Un Template con el campo metadata {performative = “Hello”}, tan solo hará match con aquellos mensajes que tengan el mismo campo metadata {performative = “Hello”} y, por tanto, tan solo permitirá encolar este tipo de mensajes en el comportamiento asociado.

4.2.4.1. Ejemplo funcionamiento SPADE

Para entender mejor el comportamiento de SPADE, supongamos el siguiente escenario:

Tenemos dos agentes, ‘Agente 1’ y ‘Agente 2’.

El agente 1 tiene 2 comportamientos. El primer comportamiento recibirá un mensaje y tiene un Template con el filtro metadata {performative = “Saludo”}, mientras que el segundo comportamiento enviará un mensaje al Agente 2 y tiene un Template con el filtro metadata {performative = ”Otros”}.

El Agente 2 tiene un solo comportamiento que recibe un mensaje y lo devuelve al remitente. Este comportamiento tiene un Template con un filtro metadata {performative = “Saludo”}

El mensaje que se enviará en el segundo comportamiento del Agente 1 tendrá el cuerpo del mensaje como “Hello World” y contendrá el metadata {performative = “Saludo”}.

Dado este planteamiento, se produce la siguiente interacción:

1. El comportamiento 2 del agente 1 envía un objeto tipo Message al agente 2.
2. El agente 2 recibe el mensaje.
3. El agente 2 encola el mensaje en todos los Behaviour que pasen el filtro del Template. Para ello, compara el Message con el Template, y solamente si los

campos informados en el Template coinciden con los campos del objeto Message, el mensaje se encolará en el comportamiento determinado.

4. El mensaje se encola en el comportamiento 1 del agente 2.
5. El comportamiento 1 del agente 2 procesa el mensaje recibido y envía otro mensaje de vuelta al Agente 1.
6. El Agente 1 recibe el mensaje respuesta.
7. El mensaje pasa por el filtro del Template 1 para entrar en el comportamiento 1.
8. El mensaje se encola en el comportamiento 1 del agente 1, el cual recibirá el mensaje y lo tratará.
9. El agente también trata de encolar el mensaje en el comportamiento 2, pero este no supera el filtro del Template, ya que estos no coinciden.

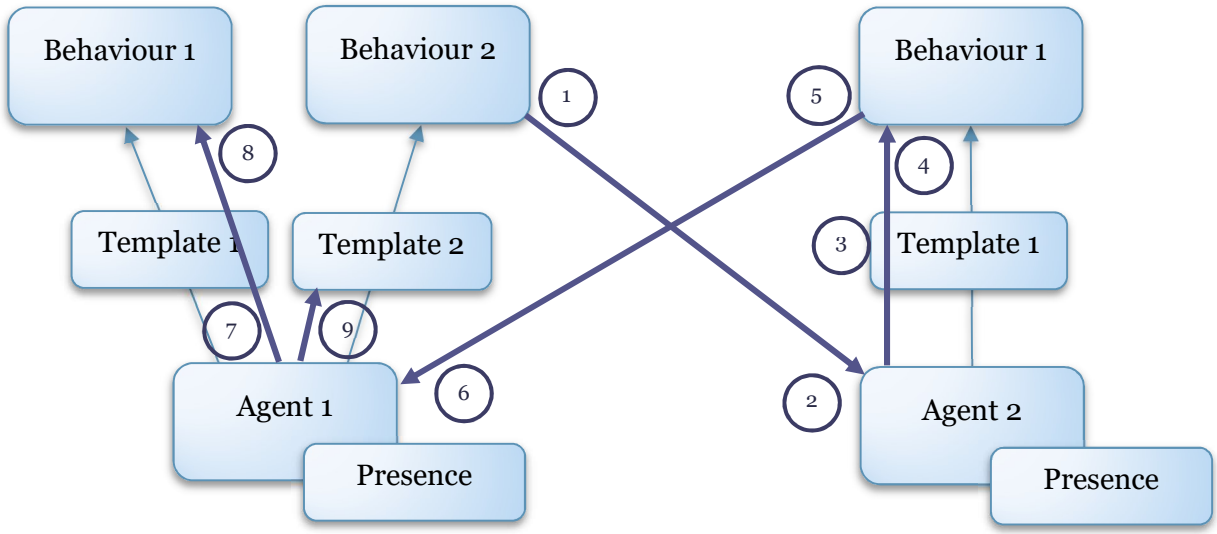

Figura 22: *Ejemplo ejecución SPADE*

Esta interacción queda visualmente reflejada en la figura 22, indicando el flujo de los mensajes intercambiados.

4.2.5. PGOMASv2

Dado que la actuación necesaria sobre la aplicación PGOMASv2 consiste en la reescritura completa de todas las clases para pasarlas a Python 3 y aplicar la asincronía entre los comportamientos, es necesario definir el comportamiento de cada una de las clases que formarán parte de la aplicación PGOMASv2

Sin embargo, al contrario que en la definición de la implementación del módulo IBR de AIOXMPP, en la que no solo definimos los ficheros que compondrá el módulo, sino que también definimos los métodos que se implementarán y el detalle de cada uno de ellos, en este caso no se describirá cada método existente por los siguientes motivos:

- Dado que es una reescritura a partir de algo ya existente, los métodos resultantes serán muy similares a los existentes.
- Sería inviable describir todos los métodos basándose en el número de métodos existentes, ya que aumentaría la extensión del presente documento enormemente.

Es por esto por lo que se describirán las clases a desarrollar a grandes rasgos. Se ha seleccionado un orden tal que permita la introducción de las clases progresivamente y sin referencias a clases definidas a posteriori:

- **Vector3D.py:** Contiene métodos útiles para trabajar con vectores de 3 elementos (x, y, z), como el módulo, normalización, suma, resta y producto vectorial.
- **CThreshold.py:** Aplica límites tanto por abajo como por arriba a los atributos de un Agente (Salud, Munición, Disparos consecutivos, etc).
- **CTerrainMap.py:** Contiene métodos útiles para la carga de un mapa a partir de un fichero de texto, así como su almacenaje.
- **CConfig.py:** Contiene la ruta donde se almacenan los mapas relativo al proyecto, para facilitar su obtención a la hora de inicializar la aplicación.
- **CMobile.py:** Contiene diversos elementos tipo Vector3D que definen los elementos de Posición, Velocidad, Velocidad y Orientación de un Agente.
- **CSigth.py:** Representa un objeto dentro del campo de visión de un agente. Cuando un Agente solicita al Mánager cuales son los elementos en su campo de visión, este devolverá una lista de objetos CSight en las que tan solo se informa de la posición, tipo, equipo y salud del objeto a la vista.
- **CTask.py:** Representa una tarea que puede tener un agente. Contiene datos sobre el tipo de tarea y la prioridad de ésta misma, así como otros datos dependientes del tipo de tarea.
- **CServer.py:** Implementa la conexión de la aplicación de PGOMASv2 al visor gráfico, para poder visualizar la ejecución de la aplicación.

- **CService.py:** Es un agente que implementa un diccionario de que agentes implementan determinados servicios. Hereda de la clase Agent de SPADE, y contiene un diccionario de listas en las que cada elemento del diccionario representa un servicio, y el valor del elemento es una lista con direcciones de los agentes que ofrecen dicho servicio. Este diccionario se gestiona mediante comportamientos que están a la escucha de mensajes de registro, deregistro, y obtención de agentes para un servicio determinado.
- **CPGomasAgent.py:** Es un agente del que heredarán todos los demás agentes de PGOMAS (excepto CService). Implementa los comportamientos básicos de registro y deregistro de servicios.
- **CPack.py:** Hereda de CPGomasAgent y representa un objeto genérico estático en el campo de batalla. Añade el atributo de tipo de paquete y su posición. Implementa el comportamiento de creación del paquete, el cual le enviará in mensaje al CManager informando de su creación.
- **CAmmoPack.py:** Hereda de CPack y representa un paquete de munición soltado por los CFieldOps. Implementa un comportamiento de autodestrucción, el cual, pasado cierto tiempo desde su creación, enviará un mensaje al mánager indicando de la destrucción del paquete, y acto seguido se autodestruye.
- **CMedicPack.py:** Hereda de CPack y representa un paquete de salud soltado por los Cmedic. Al igual que el CAmmoPack, implementa un comportamiento de autodestrucción al pasar un cierto tiempo.
- **CObjPack.py:** Hereda de CPack y representa la bandera objetivo de la simulación. Implementa los comportamientos que gestionan la obtención de la bandera por parte de una tropa del equipo de la alianza, o la pérdida de esta provocada por la muerte del agente que la porta.
- **CManager.py:** Hereda de CPGomasAgent e implementa los métodos y comportamientos relacionados con el agente mánager que se encarga de gestionar el transcurso del juego. En concreto, este agente tiene varios comportamientos:
    - Enviar periódicamente al visor gráfico información sobre los agentes presentes en el mapa, para que este pueda representar correctamente la situación de la simulación.

  - Obtener periódicamente la información de posición, velocidad y orientación de los agentes.
  - Cálculo de los objetos en el campo de visión de un agente que lo solicite.
  - Cálculo del resultado de un disparo efectuado por el agente que lo solicite. Esto implica el cálculo del agente al que ha golpeado la bala, así como la substracción de salud de dicho agente y la disminución de munición del agente solicitante.
  - Gestión de los objetos estáticos soltados por las tropas. Esto implica tanto la creación como la destrucción de estos objetos.
  - Informar a las tropas de la finalización de la simulación.

- **CTroop.py:** Hereda de PGomasAgent y representa una tropa genérica de élite. De forma similar a la clase CManager, este agente también tiene varios comportamientos:
  - Recibe del CManager la situación de la Bandera al inicio de la simulación.
  - Recibe del CManager el mensaje de finalización de la simulación.
  - Recibe del CManager información de que ha pasado por un paquete de salud o munición.
  - Recibe de CManager información de que ha recibido un disparo.
  - Informa periódicamente al CManager sobre el estado del Agente.
  - Regenera periódicamente las variables de poder, estamina y salud.
  - Solicita paquetes de salud a los médicos aliados.
  - Ejecuta el comportamiento que simula una máquina finita de estados.
- **CSoldier.py:** Hereda de CTroop y representa un soldado. Implementa el comportamiento de respuesta a llamadas de asistencia de protección.
- **CMedic.py:** Hereda de CTroop y representa un médico. Implementa tanto el comportamiento de respuesta ante una llamada de asistencia médica, como el de creación de paquetes de salud.
- **CFieldOps.py:** Hereda de CTroop y representa un operador de campo. Implementa tanto el comportamiento de respuesta ante una llamada de solicitud de munición como el de creación de dichos paquetes.

4.3. Tecnología Utilizada

Para el desarrollo de este trabajo, se utilizará el lenguaje de programación Python en todas las capas de la arquitectura. Además, cada capa se basará en la anterior para desarrollar sus funcionalidades.

Se dispone, por tanto, de la biblioteca ASYNCIO, incluida dentro de las bibliotecas estándar de Python, sobre la cual se basa la biblioteca AIOXMPP.

A continuación, se encuentra la plataforma SPADE, fundamentada sobre las bibliotecas AIOXMPP y ASYNCIO.

Por último, la aplicación PGOMASv2 se basa en los Agentes definidos en la plataforma SPADE.

5. Desarrollo de la solución propuesta

5.1. Modificaciones al 'core' de AIOXMPP

En AIOXMPP, la conexión de un cliente a un servidor supone por defecto tanto la conexión como la autentificación del cliente, sin la posibilidad de conectarse, pero no autenticarse.

Sin embargo, la inclusión del módulo IBR supone la inclusión en AIOXMPP de una nueva situación no contemplada, en la que se requiere que un cliente sea capaz de establecer una conexión con el servidor, pero no se autentique.

En este caso, la solución requiere que se modifique el núcleo de AIOXMPP para poder dar soporte a esta nueva situación.

En concreto, por un parte se deberá permitir crear un cliente sin autenticador informado, mientras que por otra parte será necesario comprobar si existe autenticador, y en el caso que no exista, saltarse el paso de autenticación.

Estas modificaciones, aunque a nivel de código no supongan una gran complicación ni muchas líneas de código, sí que supone un nivel de entendimiento elevado de la biblioteca AIOXMPP en su conjunto.

En el caso de crear un cliente sin autenticador informado, será necesario modificar la clase security_layer.py, encargado de gestionar las opciones se seguridad de la conexión. Actualmente, en el caso que no se especificara ningún método de conexión, por defecto asignaba un autenticador utilizando el método SASL. En concreto, ha sido necesario modificar el método make, eliminando la condición que provocaba que entrara por defecto:

```
if password_provider is not None ~~or anonymous is False~~:
      sasl_provider.append(
            PasswordSASLProvider(
                  Password_provider,
            ),
      )
```

Así, si no existe autenticador, no se insertará un método de autenticación.

Una vez tenemos resuelto este primer problema, es necesario que detecte si no hay autenticador, y en el caso que así sea, no se autentique en el servidor.

Para ello, se ha añadido una condición en el método _try_options de la clase node.py, que es la clase que gestiona las conexiones. Esta condición se añade justo antes de autenticarse con el servidor, y en el caso que la cumpla, se saltará dicha validación devolviendo los valores de la conexión sin realizar la autenticación:

```
if not metadata.sasl_providers:
      return transport, xmlstream, features
try:
      **Autenticación**
```

5.2. Uso del módulo IBR de AIOXMPP

Tras desarrollar el módulo IBR en AIOXMPP, se procederá a definir como se invocarán los servicios añadidos durante el desarrollo.

5.2.1. Métodos incluidos en el servicio

Los métodos incluidos en el servicio son aquellos que necesitan de autenticación para poder ejecutarse. Estos son get_client_info, change_pass y cancel_registration. Para invocar cualquiera de dichos métodos la estructura básica será la siguiente:

```
client = aioxmpp.PresenceManagedClient(
      jid,
      aioxmpp.make_security_layer(password)
)

async with client.connected() as stream:
      service = ibr.RegistrationService(stream)
      #Dependiendo del método a ejecutar se incluirá una de estas
      #líneas
      reply = await service.get_client_info()
      reply = await service.change_pass(new_password)
      reply = await service.cancel_registration()
```

5.2.2. Métodos incluidos en el servicio

Lo métodos que necesitan estar conectados pero no autenticados (get_registration_fields y register) tienen una forma distinta de invocación, ya que deben conectarse primero manualmente, para luego poder ejecutar el método correspondiente:

```
metadata = aioxmpp.make_security_layer(None)
_, stream,_ = await aioxmpp.node.connect_xmlstream(jid, metadata)
```

```
#Dependiendo del método a ejecutar se incluirá una de estas líneas
reply = await ibr.get_registration_fields(stream)
reply = await ibr.register(stream, Query(jid.localpart,password))
```

5.3. Modificaciones SPADE

Será necesario cambiar la clase Agent, dentro del método **start()**. En este método es donde se realiza la autenticación del agente con el servidor, por lo que se añadirá un nuevo parámetro que indique si se desea autorregistro, y en el caso de que así sea, el agente se registrará en el servidor utilizando el método **register()** implementado en la librería AIOXMPP.

5.4. Uso del autorregistro en SPADE

Para indicar a un agente SPADE si debe utilizar el autorregistro o no, se indicará al lanzar el método start mediante el parámetro auto_register.

Por tanto, el flujo de ejecución usual de un agente SPADE será el siguiente:

```
agent = spade.Agent()
agent.start(auto_register=True)
```

5.5. Ejecución de PGOMASv2

Para la ejecución de la simulación de PGOMAS, será necesario lanzar los agentes en un orden específico. A continuación, se muestra el fragmento de código necesario para lanzar una simulación en PGOMASv2 simple con una tropa de cada tipo para cada equipo.

```
manager = CManager("cmanager@localhost","secret", 0, 6)
manager.start()
x_sol = CSoldier("solAxis@localhost", team=CTroop.TEAM_AXIS))
x_sol.start()
x_med = CMedic("medAxis@localhost", team=CTroop.TEAM_AXIS))
x_med.start()
x_ops = CFieldOps("opsAxis@localhost", team=CTroop.TEAM_AXIS))
x_ops.start()
a_sol = CSoldier("solAllied@localhost", team=CTroop.TEAM_ALLIED))
a_sol.start()
a_med = CSoldier("medAllied@localhost", team=CTroop.TEAM_ALLIED))
a_med.start()
a_ops = CSoldier("opsAllied@localhost", team=CTroop.TEAM_ALLIED))
a_ops.start()
```

6. Implantación

6.1. Subida de código a la librería pública

A la hora de colaborar en un proyecto de software libre, no basta con subir el código al repositorio propio, sino que lo recomendable es que el código desarrollado llegue a integrarse en el proyecto original, a fin de que otros desarrolladores puedan añadir funcionalidades a partir del código subido, y que el proyecto crezca adecuadamente.

Se deben seguir, por tanto, una serie de pasos desde que se crea una copia del proyecto hasta que el código propio se integra con el proyecto en el repositorio original. En concreto, los pasos a seguir en un proyecto situado en un repositorio Git como es GitHub[11] son:

- Realizar un **fork** del repositorio: Un fork consiste básicamente en realizar una copia del proyecto en el repositorio propio. Esto permite experimentar libremente con cambios sin afectar el proyecto original. Para realizar un fork en GitHub, simplemente se ha de navegar al proyecto correspondiente y pulsar el botón fork.
- **Clonar** el proyecto al equipo local: Para ello, se utilizará el comando git clone. Por ejemplo, el comando:

  ```
  git clone https://github.com/horazont/aioxmpp
  ```

  Clonará el proyecto aioxmpp a un directorio local. Lo ideal es clonar un repositorio propio, para luego subirlo sin problemas.
- **Desarrollo**: Se desarrollará el código sobre la copia local del proyecto.
- Realizar un **commit**: Este paso subirá el código desarrollado al repositorio GitHub propio.
- Realizar un **Pull Request**: Un Pull Request es básicamente una solicitud de anexión de código. En este paso se compara el código original con el código desarrollado, para analizar los cambios que se han realizado.

[11] https://github.com/

Este Pull Request se hace visible a todo el mundo, y se abre un plazo de revisión, durante el cual cualquier persona puede agregar un comentario sobre los cambios realizados en el código. Esta fase se llama Code Review.

En el caso de este trabajo, el responsable de la librería AIOXMPP agregó comentarios añadiendo sugerencias de cambio de código.

En caso de que sea necesario, o lo requiera el responsable del proyecto original, se realizarán los cambios necesarios al proyecto y se volverá a solicitar un Pull Request, hasta que los responsables del proyecto acepten las modificaciones.

- Una vez los responsables del código original **acepten las modificaciones**, el código desarrollado pasará a formar parte del proyecto original.
- Actualmente los responsables del proyecto han evaluado y aceptado nuestras modificaciones mediante esta metodología de Pull Request y Code Review, como puede verse en la URL: https://github.com/horazont/aioxmpp/pull/211 Además, será incluido en la próxima versión de la biblioteca AIOXMPP, la v0.10.

6.2. Configuración de Prosody IM

Todos los agentes que se crean a lo largo de una ejecución de la simulación de PGOMASv2, y de forma genérica, todos los agentes creados a partir de la plataforma SPADE, deben conectarse a un servidor XMPP para poder enviar y recibir mensajes.

Además, en la actualidad, debido a que el XEP0077 In-Band Registration intercambia nombres y contraseñas sin ningún tipo de seguridad mas que el propio cifrado de la conexión, unido al hecho que la activación del autorregistro en un servidor XMPP supone permitir el registro automático de usuarios externos sin ningún tipo de control, pudiéndose explotar con facilidad, hace que en la mayor parte de servidores XMPP actuales el soporte al autorregistro está desactivado por motivos de seguridad.

Se hace necesaria pues, la utilización de servidores instalados en la máquina local, el los que se tiene total control sobre su configuración, y así poder activar libremente el soporte al autorregistro sin temer por ningún tema de seguridad.

En nuestro caso, el servidor utilizando para realizar las pruebas durante el desarrollo, se ha utilizado el servidor XMPP Prosody IM, el cual puede instalarse mediante el comando:

```
sudo apt-get install prosody
```

Una vez el servidor está instalado, podemos arrancar y paralo en la máquina local mediante los siguientes comandos:

```
sudo prosodyctl start
sudo prosodyctl stop
```

También es posible añadir y borrar usuarios manualmente, lo que resulta de utilidad para desarrollar el módulo IBR con facilidad:

```
sudo prosodyctl adduser
sudo prosodyctl deluser
```

Por último, para añadir el soporte al autorregistro dentro del servidor local, es necesario modificar el archivo de configuración prosody.cfg.lua. En concreto, será necesario tanto activar el módulo “register” como permitir el autorregistro:

```
modules_enabled = {
      “register”;
      …
}
allow_registration = true
```

7. Pruebas

Para confirmar que el desarrollo realizado cumple con las especificaciones dadas y está libre de fallos, se han realizado una serie de pruebas y validaciones, las cuales se detallan a continuación.

7.1. Calidad del código

Con el objetivo de aumentar la legibilidad del código y hacerlo consistente a través de su amplio espectro en la comunidad Python, el código desarrollado durante el transcurso del trabajo se basará en la guía de estilo para código Python PEP 8. Este documento proporciona una serie de convenciones a la hora de escribir código Python, como, por ejemplo:

- El uso de cuatro espacios por nivel de indentación.
- La escritura de líneas de longitud menor a 79 caracteres.
- El número de líneas en blanco entre métodos o clases.

Estos son algunas de las múltiples convenciones que se incluyen en esta guía, y revisarlo manualmente sería tarea inalcanzable. Por eso, para comprobar si un fichero Python sigue las convenciones dentro de la guía, existe un comando Python llamado pep8, el cual comprueba automáticamente si se siguen o no las convenciones de la guía, y muestra las incidencias que encuentre.

```
./CManager.py:165:80: E501 line too long (87 > 79 characters)
./CManager.py:272:80: E501 line too long (89 > 79 characters)
./CManager.py:274:33: E231 missing whitespace after ','
./CManager.py:275:21: E265 block comment should start with '# '
./CManager.py:279:25: E265 block comment should start with '# '
./CManager.py:297:80: E501 line too long (105 > 79 characters)
./CManager.py:300:80: E501 line too long (90 > 79 characters)
./CManager.py:306:80: E501 line too long (106 > 79 characters)
./CManager.py:309:80: E501 line too long (113 > 79 characters)
./CManager.py:312:17: E265 block comment should start with '# '
./CManager.py:313:56: E261 at least two spaces before inline comment
./CManager.py:313:57: E262 inline comment should start with '# '
./CManager.py:316:9: E303 too many blank lines (2)
./CManager.py:337:9: E265 block comment should start with '# '
./CManager.py:339:9: E265 block comment should start with '# '
./CManager.py:339:80: E501 line too long (110 > 79 characters)
./CManager.py:340:9: E265 block comment should start with '# '
```

Figura 23: *Ejemplo ejecución pep8*

En este trabajo se ha aplicado dicho comando a todos los ficheros desarrollados, logrando un completo cumplimiento de dichas convenciones.

7.2. Pruebas unitarias y de integración

Las pruebas unitarias son una forma de comprobar si un método determinado funciona correctamente de forma aislada. Con esto, verificamos que cada método hace lo que tiene que hacer, es decir, ante unas determinadas entradas, se conoce cuál es el resultado.

Una prueba unitaria consiste básicamente en un test que ejecutará el método sobre el que se desea realizar la prueba unitaria, y ante unas determinadas entradas, se verificará que las salidas son las esperadas. A ser posible, también se controlarán estados internos del método que sean de interés para la verificación.

Estas pruebas no pretenden comprobar las interacciones del método con el exterior, sino solamente el comportamiento interno. Por tanto, si un método contiene interacciones con otros sistemas, se suele simular realizando **stubs** (objetos con comportamiento programado ante ciertas llamadas de un test en concreto) o **mocks** (objetos ya programados con los datos que se espera recibir).

Las pruebas de integración, por el contrario, pretenden validar la interacción entre distintos componentes del sistema. Este tipo de pruebas verifican que los componentes de la aplicación funcionan correctamente actuando en conjunto.

Este tipo de pruebas son dependientes del entorno al que se ejecutan. Si fallan, puede ser porque el código esté bien, pero haya un cambio en el entorno.

7.2.1. unittest

En Python, existe un framework que permite simplificar el diseño y ejecución de pruebas unitarias. Este módulo es llamado `unittest` [17], y soporta la automatización de pruebas de forma sencilla. Tan solo es necesario que la clase que contenga los tests unitarios herede de la clase `unittest.TestCase`.

A continuación, en la terminal, se ejecutará el comando nosetests, el cual ejecutará todos los métodos dentro de clases del tipo `unittest.TestCases` presentes dentro del módulo indicado. La sintaxis del comando con los parámetros que interesan en este proyecto es la siguiente:

```
nosetests [module] [args]*
```

Utilizando los siguientes argumentos:

```
--with-coverage: indica que se desea medir también la cobertura
de los tests y no solo si pasan correctamente o no.
--cover-html: crea una estructura de ficheros html que detallan
cuáles son las líneas de código que cubren los tests.
--cover-package=[package]: indica el modulo sobre el que se desea
medir la cobertura.
```

En el trabajo realizado objeto de este proyecto se han desarrollado numerosas pruebas unitarias.

En concreto, para el módulo desarrollado en AIOXMPP, se han realizado pruebas unitarias para todos los métodos desarrollados, provocando que el flujo de código pase por todos los posibles flujos, con lo que obtenemos una cobertura de pruebas unitarias del 100%.

```
sergio@sergio:~/aioxmpp/tests$ nosetests ibr --with-coverage --cover-package=aioxmpp/ibr
SSSSS.........................
Coverage.py warning: Module aioxmpp/ibr was never imported. (module-not-imported)
Name                              Stmts   Miss  Cover
-----------------------------------------------------
aioxmpp/ibr/__init__.py               3      0   100%
aioxmpp/ibr/service.py               33      0   100%
aioxmpp/ibr/xso.py                   35      0   100%
-----------------------------------------------------
TOTAL                                71      0   100%
----------------------------------------------------------------------
Ran 33 tests in 0.052s

OK (SKIP=5)
```

Figura 24: *Ejecución pruebas unitarias AIOXMPP*

También, aunque la configuración del sistema de AIOXMPP no permite la correcta inclusión de pruebas de integración automáticas para el módulo In-Band Registration, se han desarrollado una serie de ejemplos en las que se crean, modifican y eliminan usuarios en un servidor XMPP local, con el objetivo de suplir esa carencia.

En cuanto a la aplicación PGOMASv2, también se han desarrollado pruebas unitarias para todas las clases desarrolladas. Sin embargo, dado que una gran parte del código no tiene sentido si no es dentro de un sistema multiagente conectado a un servidor, se ha decidido prescindir de las simulaciones de integración con otros sistemas, como los stubs o mocks, resultando así en una serie de pruebas unitarias que son a su vez, pruebas de integración del sistema.

7.3. Pruebas de sistema

Las pruebas de sistema son aquellas cuyo objetivo es probar todo el sistema software completo e integrado, normalmente desde el punto de requisitos de la aplicación, es decir, comprobar que es software cumple con la función para la que se había pensado.

Por eso, este tipo de pruebas se incluyen entre las llamadas pruebas de caja negra: no se centran en cómo se generan las respuestas del sistema, sino que se analizan los datos de entrada y los resultados obtenidos.

En este trabajo se ha ejecutado la aplicación PGOMAS como prueba de que la simulación funciona correctamente.

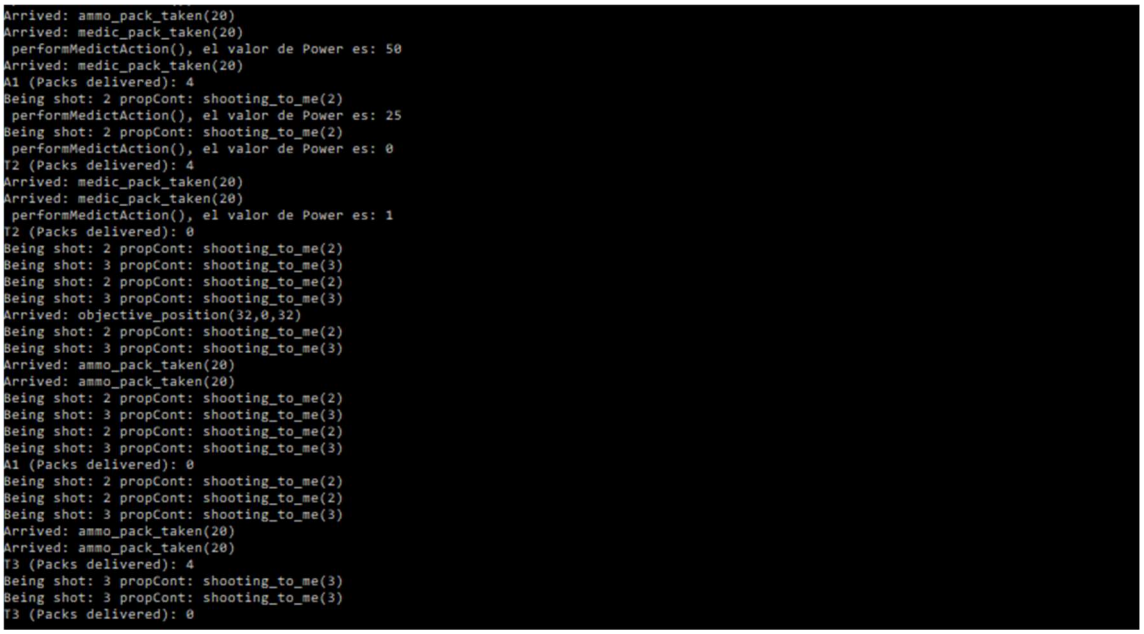

Figura 25: *Ejemplo ejecución PGOMAS*

Como se puede observar, PGOMASv2 se ejecuta correctamente, los agentes siguen el comportamiento básico, moviéndose hacia el objetivo y disparándose entre sí, dañándose entre ellos.

8. Conclusiones

Se ha desarrollado un simulador de agentes basado en el lenguaje de programación Python similar a la aplicación JGOMAS usada en la asignatura Agentes Inteligentes del Grado en Ingeniería Informática.

Se han cumplido, por tanto, todos los subobjetivos propuestos:

- Se ha desarrollado un nuevo módulo en la biblioteca AIOXMPP, implementando el XEP0077: In-Band Registration, y este ha sido subido y aceptado en la biblioteca de origen, contribuyendo al software libre.
- Se ha modificado la plataforma SPADE para permitir el autorregistro de los agentes, colaborando en la nueva revisión v3.0 de la plataforma.
- Se ha migrado el código de PGOMASv1 a PGOMASv2, cambiando de Python2 a Python3 e implementando asincronía en los comportamientos de los agentes mediante la biblioteca AIOXMPP y la v3.0 de SPADE.

Las aportaciones de cada subobjetivo corresponden a tres piezas de software que se pueden usar independientemente. Dos de ellos pertenecen a proyectos de software libre, mientras que el tercero se utilizará en las prácticas de la asignatura de Agentes Inteligentes en el Grado de Ingeniería Informática de la Universitat Politècnica de València.

Se ha conseguido un 100% de cobertura y un total cumplimiento del pep8 en todas las clases desarrolladas.

Este trabajo se ha realizado dentro del grupo de investigación GTI-IA de la Universitat Politècnica de València.

8.1. Relación del trabajo desarrollado con los estudios cursados

Durante el análisis y desarrollo de este trabajo, se han aplicado múltiples campos de estudio relacionados con los estudios del Grado de Ingeniería Informáticas. Entre los conceptos más importantes aplicados se incluyen:

- El análisis y aplicación de los **agentes inteligentes** software, el impacto que estos tienen y como se estructuran y se comportan.
- El análisis y aplicación de los **sistemas multiagentes**, es decir, la colaboración entre un conjunto de agentes inteligentes.
- La **concurrencia** dada entre los hilos de los agentes, y la **asincronía** presente en los comportamientos de éstos.
- La posibilidad de distribución de los agentes en diferentes servidores, situados en diferentes máquinas formando **sistemas distribuidos**.
- El estudio de diversos **estándares de comunicación**, así como la aplicación de parte de uno, XMPP.

8.2. Trabajos Futuros

A partir del trabajo realizado en este proyecto, se plantean diversos posibles trabajos que se pueden realizar en un futuro.

Por una parte, teniendo en cuenta que la plataforma SPADE es independiente de PGOMASv2, surge la posibilidad de utilizar SPADE en otros ámbitos fuera de la aplicación de PGOMASv2.

Por otra parte, actualmente se está trabajando junto a los responsables de la asignatura para la realización del manual de usuario y la redacción del enunciado de prácticas, de forma que sus contenidos se adecuen a los objetivos de dichas prácticas.

9. Referencias

[1] Antonio Barella, Soledad Valero and Carlos Carrascosa. JGOMAS: New Approach to AI Teaching. 2009
IEEE Transactions on Education (Vol. 52 No. 2), Pages 228 – 235

[2] Rafael H. Bordini and Jomi F.Hübner. Jason, A Java-based interpreter for an extended version of AgentSpeak,2007
Available from: http://jason.sourceforge.net/Jason.pdf (Viewed: 07/07/2018)

[3] Fabio Bellifemine, Agostino Poggi and Giovanni Rimassa. JADE – A FIPA-compliant agent framework, 1999
Available from: http://jade.tilab.com/papers/PAAM.pdf (Viewed: 07/07/2018)

[4] Stefan Poslad. Specifying Protocols for Multi-Agent Systems Interaction, 2007
ACM Transactions on Autonomous and Adaptive Systems (Vol. 2 No.4) Articles 12-17

[5] Object Management Group. The Common Object Request Broker: Architecture and Specification. Version 3.3, 2012
Available from: https://www.omg.org/spec/CORBA/3.3/ (Viewed: 07/07/2018)

[6] Miguel Escrivá Gregori, Javier Palanca Cámara and Gustavo Aranda Bada. A Jabber-based Multi-Agent System Platform, 2006
Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, Pages 1282 - 1284

[7] Jabber Software Foundation. Extensible Messaging end Presence Protocol (XMPP): Core, 2004
Available from: http://www.ietf.org/rfc/rfc3920.txt (Viewed: 07/07/2018)

[8] Jabber Software Foundation. Extensible Messaging end Presence Protocol (XMPP): Instant Messaging and Presence Protocol, 2004
Available from: http://www.ietf.org/rfc/rfc3921.txt (Viewed: 07/07/2018)

[9] Jabber Software Foundation. Extensible Messaging end Presence Protocol (XMPP): Core, 2011
Available from: http://www.ietf.org/rfc/rfc6120.txt (Viewed: 07/07/2018)

[10] Jabber Software Foundation. Extensible Messaging end Presence Protocol (XMPP): Instant Messaging and Presence Protocol, 2011
Available from: http://www.ietf.org/rfc/rfc6121.txt (Viewed: 07/07/2018)

[11] Jabber Software Foundation. Extensible Messaging end Presence Protocol (XMPP): Address Format, 2011
Available from: http://www.ietf.org/rfc/rfc6122.txt (Viewed: 07/07/2018)

[12] Peter Saint-Andre and Dave Cridland. XMPP Extension Protocols, Version 1.21.1
Available from: https://xmpp.org/extensions/xep-0001.html (Viewed: 07/07/2018)

[13] Python Software Foundation. ASYNCIO – Asynchronous I/O, event loop, coroutines and tasks
Available from: https://docs.python.org/3/library/asyncio.html (Viewed: 07/07/2018)

[14] Jonas Wielicki. AIOXMPP official documentation
Available from: https://docs.zombofant.net/aioxmpp/devel/ (Viewed: 07/07/2018)

[15] Peter Saint-Andre. XEP-0077: In-Band Registration, Version 2.4
Available from: https://xmpp.org/extensions/xep-0077.html (Viewed: 07/07/2018)

[16] Python Software Foundation. Style Guide for Python Code, 2001
Available from: https://www.python.org/dev/peps/pep-0008/ (Viewed: 07/07/2018)

[17] Python Software Foundation. Unit testing framework
Available from: https://docs.python.org/2/library/unittest.html (Viewed: 07/07/2018)