Escuela
Politécnica
Superior

Modelado de un sistema multiagente para coordinación de dispositivos IoT

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Mario Nieto Pérez

Tutor/es:

Francisco José Mora Lizán

Universitat d'Alacant
Universidad de Alicante

Julio 2020

Este trabajo cuenta con el apoyo del Ministerio de Ciencia, Innovación y Universidades (España), proyecto RTI2018-096219-B-100. Proyecto cofinanciado con fondos FEDER.

Introducción

Durante estos últimos años, los avances habidos tecnología han propiciado el auge de los dispositivos IoT y la adopción por parte de una gran mayoría de la población de dichos dispositivos. Esto ha desembocado en una gran cantidad de empresas y fabricantes que han diseñado y sacado al mercado sus productos, cada uno según han querido diseñar sin seguir ningún tipo de estándar, lo que aumenta la dificultad a la hora de interconectar estos dispositivos. Además, estos dispositivos trabajan únicamente de manera reactiva, sin tener capacidad de planificación para cumplir con objetivos a largo plazo.

Con esto surge la idea de este Trabajo de Fin de Grado, la idea de crear una capa intermedia entre dos tecnologías: por un lado, IoT, y por otro lado los Sistemas Multiagentes, una tecnología ya desarrollada que se centra en crear sistemas con diversos elementos autónomos que son capaces de planificar y negociar para cumplir con objetivos propios o comunes. Se busca así juntar estas dos tecnologías de manera en que se pueda emplear cada una de ellas para suplir con las carencias de la otra, creando un software general que pueda emplearse en multitud de entornos.

Agradecimientos

Me gustaría agradecer en primer lugar a mi tutor; Francisco José Mora, por haberme guiado y haber tenido la paciencia que ha tenido conmigo para ayudarme y resolverme mis dudas.

Agradecer a mi familia y a mis amigos, de nuevo, por apoyarme a lo largo de todo este tiempo y haberme ayudado a seguir adelante.

A mi pareja, que ha estado a mi lado en los peores momentos y me ha querido y apoyado siempre.

Y en último lugar, gracias, mamá, por haberme educado y haber hecho de mí la persona que soy ahora, y porque sin ti no habría logrado llegar hasta aquí.

Contenido

Indice de figuras

Justificación y objetivos

El principal objetivo de este trabajo de fin de grado es el de crear una capa intermedia entre dos tecnologías con el fin de suplir sus carencias correspondientes: por un lado, Sistemas Multiagentes, y por otro la tecnología de dispositivos IoT. Con esto obtendremos una capa de planificación para decidir objetivos y otra a bajo nivel donde podremos actuar y observar el entorno mediante el uso de estos dispositivos IoT.

Por otra parte, con el fin de justificar y comprobar la implementación de esta capa intermedia, se busca crear un sistema de ejemplo donde poder probar y demostrar el uso de este middleware.

Por último, queremos que este middleware sea genérico y se pueda emplear en multitud de escenarios, por lo que el último objetivo que se busca cumplir en este trabajo es el de lograr que esta capa sea genérica, ampliable y escalable y permitir su uso en diversos escenarios con multitud de dispositivos distintos.

1. Marco teórico y estado del arte

1.1 IoT

Es complicado hacer una definición formal de lo que es el IoT, pero podemos decir de forma genérica que el IoT es "Concepto que se refiere a la interconexión de objetos cotidianos con Internet" (Mohammadi Zanjireh & Larijani, 2015). Podemos entender una "cosa" (Thing) como cualquier elemento de nuestras vidas que hemos conectado a Internet, como, por ejemplo, un hombre con un sistema de monitorización de corazón, un coche con sensores o una bombilla inteligente.

El IoT, por tanto, no se trata de una tecnología, sino de un conjunto de soluciones y tecnologías tanto hardware como software.

1.1.1 De qué se componen los dispositivos IoT

Los dispositivos IoT se caracterizan por estar conectados a la red, y suelen estar compuestos de otros dos elementos:

- Sensores, que recogen datos del entorno como la temperatura, humedad, etc.
- Actuadores, que realizan alguna acción alterando el entorno, bien porque el usuario lo especifica o bien por su propia programación.

Un dispositivo IoT puede estar compuesto de varios o de ninguno de estos elementos, dependiendo de su funcionalidad o del grado de complejidad a alcanzar deseado. Una bombilla inteligente puede cambiar de color e intensidad lumínica y no tener ningún sensor y funcionar únicamente mediante el input que recibe del usuario o puede tener un sensor de intensidad lumínica y detectar momentos donde se empieza a hacer de noche o a oscurecer y aumentar la intensidad automáticamente.

Otro ejemplo: supongamos un ejemplo de una puerta controlada mediante una cámara de reconocimiento facial. El dispositivo, cuando alguien se pone en frente

de la cámara (el sensor), escanearía la cara de la persona, y si es alguien que tiene acceso le desbloquearía la puerta para que pudiese pasar. Sin embargo, si la persona no fuese alguien que puede acceder, le bloquearía el paso y avisaría mediante Internet al administrador documentando un intento fallido de acceso con una foto de la persona.

1.1.2 Conectividad en IoT

La conexión de los dispositivos IoT a la red plantea una serie de problemas. Para empezar, IPv4 no permitirá llegado el momento soportar la enorme cantidad de dispositivos distintos conectados, motivo principal por el que surgió IPv6. Por otra parte, el uso de redes convencionales tiene algunas desventajas a la hora de emplearlas para el IoT, por lo que nuevas tecnologías han surgido para intentar ofrecer alternativas.

Las soluciones que existen para la conectividad se dividen en dos tipos: por un lado, las redes locales (LAN) o personales (PAN) y por otro las redes con una cobertura más amplia (WAN), donde el empleo de bajas energías es algo más relevante o donde hay que tener en cuenta las conexiones intermitentes, bien porque no existe una necesidad de una conexión continua o bien porque el entorno no permite mantener constantemente dicha conexión.

Las tecnologías de LAN y PAN más importantes son:

- Wi-Fi: el empleo de nuevos estándares más robustos, así como la integración de las redes mesh hace de Wi-Fi uno de los estándares más usados para redes locales.
- Bluetooth: se emplea normalmente para redes personales, donde los dispositivos interconectados entre sí suelen tener como 'nodo central' un teléfono móvil.

- Zigbee: tecnología de bajo consumo basada en redes de malla. Suele emplearse en hogares cuando se desea una velocidad de transferencia alta.
- Z-Wave: similar a Zigbee, también emplea tecnología de mallas, pero suele escogerse cuando se desea que la señal llegue más lejos que con Zigbee.
- Thread: alternativa para redes personales que emplea IPv6 y que tiene un consumo energético bajo.

Tecnologías WAN:

- LoRaWAN: emplea frecuencias bajas, y está construida para soportar numerosos dispositivos que mandan datos a la nube en entornos abiertos. La implementación de esta red en dispositivos es Open Source, mientras que la construcción de antenas es ahora mismo propietaria.
- NarrowBand-IoT: creada por 3GPP, se trata de una tecnología que surgió para habilitar a la mayor cantidad de dispositivos a comunicarse empleando bajo coste en entornos cerrados con altas densidades de dispositivos.
- LTE M: estándar para redes móviles para el soporte de dispositivos IoT, que se emplea mientras se avanza en el despliegue de las redes 5G.
- 5G: redes más flexibles, rápidas y de baja latencia, que permitirán multitud de aplicaciones IoT como, por ejemplo, coches autónomos o realidad virtual. Se encuentran en proceso de desarrollo y despliegue.

Cabe recalcar que, sin embargo, para nuestro proyecto, este aspecto nos es menos relevante, ya que vamos a abstraer respecto a la capa de comunicaciones, siendo esta tecnología de transmisión irrelevante para nosotros.

1.1.3 Conexión de dispositivos a Internet

Para lograr conectar los dispositivos a Internet, existen numerosas formas de estructurar el procesamiento de los datos y de estructurar las capas y servicios

disponibles. Además, el empleo de tecnologías en la Nube resulta muy atractivo por la poca capacidad de cómputo de la que disponen estos dispositivos. Dependiendo del grado de complejidad que se desea, han surgido distintos enfoques con distintas capas:

- Conexión de dispositivo a dispositivo: mediante una red local, ambos dispositivos se comunican entre sí. Por ejemplo, una pulsera inteligente se conecta directamente por bluetooth a nuestro teléfono móvil.
- Dispositivo a nube: se emplea direcciones IP para lograr comunicar a los dispositivos entre sí a través de Internet.
- Dispositivo a gateway a nube: se emplea un nodo/gateway intermedio entre los dispositivos y la nube. Esta capa puede así realizar un primer procesamiento y análisis de los datos antes de comunicarse con la nube. A esto se le conoce como "Procesamiento en la niebla" o "Edge Computing".

A modo de facilitar el proceso de adquisición y análisis de datos, se ha creado un modelo de conexión entre dispositivos y la Nube que se está estableciendo como el principal estándar a seguir, empleando el Gateway intermedio en entornos como el hogar.

Este modelo consiste en establecer 4 capas que se comunican entre ellas. Estas son la capa de sensores/actuadores, capa de adquisición de datos, capa de computación en la niebla y la capa de la Nube (*What is IoT architecture?*, s. f.). Las comunicaciones entre estas capas deben ser bidireccionales, de forma que tras realizar los cálculos necesarios se pueda enviar las respuestas o acciones a realizar pertinentes.

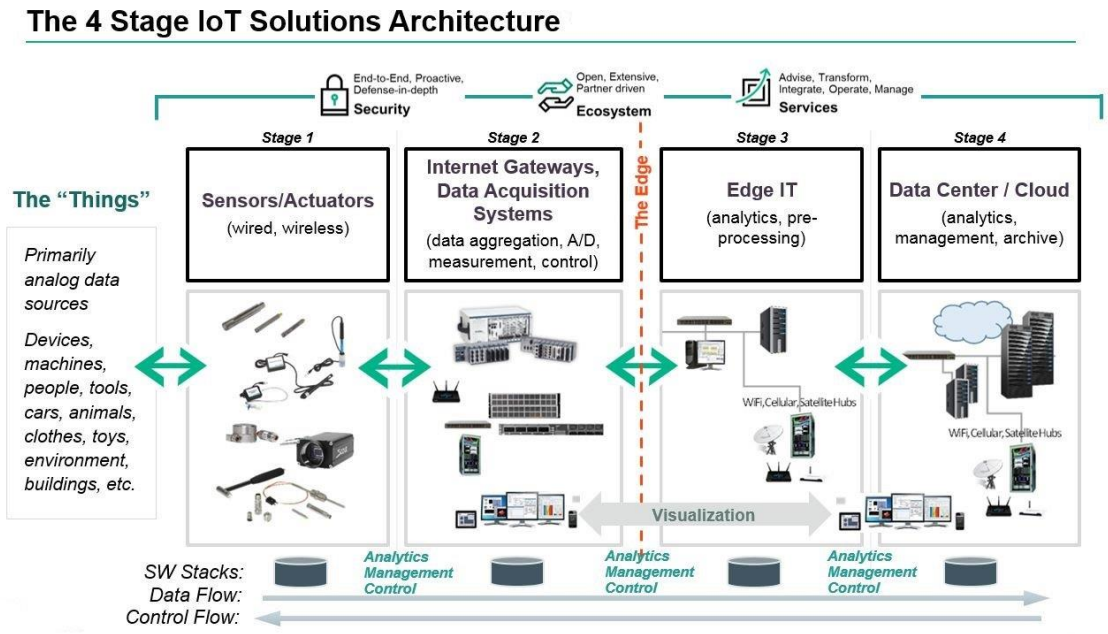


Figura 1 - Arquitectura en capas IoT

La primera capa, sensores y actuadores, se encargaría de recolectar datos del entorno y de actuar en este. Estos datos se enviarán a la segunda capa, la capa de adquisición, que se encarga de recolectar los datos y de efectuar un primer preprocesado agregando o filtrando datos, de forma que sean más fáciles de procesar. Estos se enviarán a la capa de la Niebla, que provee de una respuesta más rápida que la Nube y se puede emplear cuando sea necesario una rápida actuación o cuando los bloques de datos no sean muy grandes, agilizando así el proceso. Cuando esto no sea posible, los datos se enviarán a la Nube, que se encarga de almacenar todos estos datos masivos, así como de los procesos de cómputo más pesados y de análisis de todos estos datos. También puede darse el caso contrario, en el que son los dispositivos los que acceden a la Nube para obtener datos para actuar, es decir, la comunicación es bidireccional.

1.1.4 Características del IoT

Seguridad y privacidad: Los dispositivos IoT recopilan una gran cantidad de información y de datos del usuario que deben de asegurarse y de protegerse

empleando las debidas técnicas y estándares necesarios. Esto sin embargo no ocurre, y la seguridad es uno de los mayores problemas a los que se enfrenta el IoT hoy en día, ya que, debido al rápido crecimiento, las medidas de seguridad adoptadas no son las adecuadas para la cantidad de dispositivos, existiendo graves vulnerabilidades en aspectos como la autenticación, encriptación de mensajes o privacidad de los datos. Es este aspecto de la seguridad el que está frenando la adopción de esta tecnología por los problemas que presenta, y que no se encuentran resueltos al 100%. Otra de las trabas a las que se enfrenta el IoT para lograr avanzar es un cierto recelo por parte de consumidores y expertos relacionados con la privacidad del usuario. No en vano, ejemplos como el uso de Alexa por parte de Amazon para escuchar las conversaciones de los clientes (Phelan, s. f.) han puesto en alerta sobre los usos indebidos e invasivos que puede presentar el empleo de estas tecnologías.

Heterogeneidad: IoT se trata de una tecnología moderna y no estandarizada, por lo que existe una enorme heterogeneidad de tanto dispositivos como tecnologías que hay que tener en cuenta, sobre todo al tratar el aspecto de la comunicación, donde existen numerosos protocolos como ZigBee o MQTT. Como hemos explicado anteriormente, no existen unos estándares en todos los dispositivos, y cada empresa o fabricante suele decidir por su cuenta las tecnologías y protocolos de comunicación a emplear sus dispositivos. A la hora de tratar con dispositivos de diferentes compañías esto puede dar paso a muchos problemas para lograr comunicar todos los dispositivos de un mismo ecosistema.

Sensores y Big Data: si tenemos en cuenta la cantidad de dispositivos IoT, así como que estos dispositivos se encuentran de manera constante recopilando datos a través de sus sensores, podemos imaginarnos que la cantidad de datos obtenida es descomunal. El uso y procesamiento de estos datos, comúnmente referidos como Big Data, tendrá un papel fundamental en el IoT ya que el empleo de miles de

dispositivos para la recopilación de datos de manera constante provoca un gran problema a la hora de tratar y almacenar esta enorme cantidad de datos de manera local, donde el empleo de tecnologías de Big Data, así como el uso de una arquitectura a capas donde se preprocesan los datos antes de ser enviados toman una gran relevancia.

Escalabilidad: debemos implementar aplicaciones y dispositivos que sean capaces de escalar fácilmente con el aumento de dispositivos conectados. La existencia de una increíble heterogeneidad de dispositivos, plataformas, tecnologías y empresas de IoT ha causado que cuando pasamos de dispositivos concretos de IoT, donde sí existen ciertas guías a la hora de crear dispositivos (incluyendo productos open source de hardware, como las placas Arduino) a un nivel mayor de abstracción, donde es sumamente importante el disponer de una arquitectura que nos facilite la escalabilidad y estandarización, nos encontramos con una falta de concreción y estandarización que nos dificulta esta escalabilidad.

Eficiencia energética: los dispositivos IoT, por definición, son múltiples y muy variados. Para poder disponer de multitud de estos dispositivos, por tanto, un aspecto clave es que estos deben de consumir la menor energía posible y de ser lo más eficientes posibles, puesto que pese a que un dispositivo puede tener mayor o menor importancia, cuando se tiene en cuenta la enorme cantidad, este factor pasa a ser muy importante.

Naturaleza dinámica: el estado de los dispositivos puede cambiar de manera dinámica, así como el contexto, es decir, el entorno en el que están. Por esto existe gran relevancia en el aspecto de obtención de datos a través de sensores.

Densidad de dispositivos: el uso de dispositivos IoT es algo que está en alza, y la investigación en el campo, el desarrollo comercial de dispositivos y el conocimiento de la tecnología están haciendo que la cantidad de dispositivos

aumente con mucha rapidez a escalas muy grandes que es necesario tener en cuenta. En 2018, el número de dispositivos es de 7 billones (*State of the IoT 2018*, s. f.) y se esperaba que para 2020 hubiese en torno a 20,4 billones (*Gartner Says 8.4 Billion Connected «Things» Will Be in Use in 2017, Up 31 Percent From 2016*, s. f.).

1.2 ROS

1.2.1 Uso de ROS para el IoT

La principal razón por la que hemos decidido emplear ROS para el tratamiento de dispositivos a bajo nivel es que permite el uso de una capa de abstracción y estandarización que nos facilita el tratar con la enorme cantidad y variedad de dispositivos IoT existentes actualmente. Emplear el sistema hará que la conexión de dispositivos consista en implementar la conexión con la interfaz de ROS, que es estándar e independiente de los dispositivos a emplear.

1.2.2 ¿Qué es ROS?

ROS (Robot Operating System) es un framework Open Source diseñado para escribir software de robots. Se trata de una colección de herramientas, librerías y convenciones que tienen como objetivo simplificar la creación de comportamientos robustos para robótica para una gran mayoría de plataformas robóticas distintas.

A pesar de denominarse Sistema Operativo lo cierto es que no se trata de uno, sino que funciona sobre Linux, y funciona como uno en el sentido de que proporciona multitud de facilidades para, por ejemplo, paso de mensajes, abstracción del hardware, control sobre dispositivos de bajo nivel, manejo de memoria, etc., simplificando así una gran cantidad de procesos y tareas que de otra forma requerirían de un esfuerzo para su implementación/realización.

ROS se basa en una arquitectura de grafos donde el procesamiento se realiza en estos mismos nodos, que son capaces de recibir y mandar mensajes entre ellos o a

sensores, actuadores, sistemas de control o planificación, etc. ROS provee así de una infraestructura para las comunicaciones de forma distribuida.

Además, ROS ofrece también una serie de herramientas que simplifican el proceso de desarrollo, como un sistema de visualización en 3D de entornos, una consola de comandos, etc.

1.2.3 Características de ROS

Ventajas

Distribuido y modular: ROS está implementado de forma modular, de tal forma que es el usuario el que decide qué partes quiere usar del sistema, cuáles no y cuales quiere implementar el mismo para que se adecuen a la funcionalidad deseada sin afectar al resto de módulos.

Abstracción respecto a hardware: ROS permite que no sea necesario implementar gran cantidad del software necesario para controlar el hardware ya que provee de los métodos necesarios para su control.

Compatibilidad con lenguajes: el uso de los paquetes ROS permite que sea compatible con multitud de lenguajes de programación. Por defecto, ROS permite tanto C++ como Python, pero puede ampliarse con paquetes para añadir, por ejemplo, JAVA.

Integración con librerías: ROS permite el uso de librerías muy empleadas y de transmitir datos y transformarlos entre ellas y entre distintos sistemas. Algunos ejemplos de estas librerías son OpenCV, empleada en el tratamiento de imágenes por computador, o Gazebo, un simulador de entornos 3D para robótica.

Inconvenientes

No multiplataforma: ROS requiere de un sistema Linux para poder funcionar, no funcionando en Windows o MacOS.

Curva de aprendizaje: ROS requiere de un cierto aprendizaje inicial para entender cómo funciona el sistema y los distintos paquetes y librería para poder empezar a emplearlo.

Configuración y versiones: ROS hace uso de multitud de versiones, y la actualización de este hace que los paquetes tengan a su vez que actualizarse a nuevas versiones, cosa que no ocurre siempre. Es necesario un poco de investigación previa para ver qué paquetes van a ser necesarios y en qué versión están implementados.

1.2.4 Conceptos

ROS se compone de 2 tipos de conceptos generales: el Sistema de Archivos y Grafo de Computación.

En el Sistema de Archivos podemos encontrar los recursos que se encuentran en nuestro disco, a saber:

- Paquetes: la unidad básica para organizar software; puede contener procesos en ejecución, librerías, datasets, archivos de configuración, etc.
- Metapaquetes: representación de una relación entre paquetes.
- Manifiesto de paquetes: archivo .xml que contiene los metadatos de un paquete.
- Mensajes (msg): descripción de mensajes, definen la estructura de los mensajes mandados dentro de ROS.
- Servicios (srv): descripción de servicios, definen la estructura de datos de petición y respuesta de los servicios en ROS.

Por otro lado, el Grafo de Computación es la red de procesos ROS que se encuentran trabajando y procesando datos en conjunto, que se compone de los siguientes elementos:

- Nodo: procesos que realizan una tarea. ROS está pensado para ser modular, y un sistema se compondrá de múltiples nodos trabajando en conjunto. Se escriben empleando una de las librerías cliente propias de ROS, como roscpp o rospy.
- Máster: nodo maestro encargado de coordinar al resto de nodos.
- Servidor de parámetros: parte del Master, proporciona un lugar donde guardar y acceder a parámetros de configuración.
- Mensajes: estructura de datos con tipos preestablecidos que sirve para comunicar distintos nodos entre sí.
- Topics: los mensajes que enrutan mediante un sistema de publicación/subscripción. Los nodos mandan los mensajes publicándolos a un Topic, al cual otros nodos se suscribirán para recibir los mensajes.
- Servicios: comunicaciones del tipo petición/respuesta, que complementa a los mensajes enviados a los distintos Topics, que funcionan como un sistema muchos a muchos.

1.2.5 Funcionamiento básico de ROS

Por un lado, tenemos este nodo Master, que actúa como un servicio ofreciendo información de los distintos nombres, topics y servicios registrados. Los nodos se comunican con el Master ofreciendo su propia información de registro, y pueden entonces recuperar del Master la información necesaria sobre otros nodos, permitiendo así establecer las conexiones entre ellos. Sin embargo, esta conexión entre nodos no se realiza a través del Master. El master tan solo ofrece al nodo la información necesaria para que se conecte, funcionando así de forma similar a un servidor DNS. Los nodos que se conectan a un topic solicitan la información del resto de nodos que publican en este topic, y establecen esta conexión mediante un protocolo, normalmente TCPROS (standard TCP/IP).

Los nodos quedan así separados y sin conocer de la existencia del resto de nodos. Tan solo se encargarán de publicar y suscribirse a los distintos Topics, y cuando un nodo publica un mensaje simplemente lo manda al Topic, sin saber a quién se lo manda, y cuando un nodo recibe un mensaje tan solo sabe que recibe el mensaje por estar suscrito a ese Topic, sin saber quién lo ha mandado. Los nodos pueden así inicializados, detenidos y reiniciados, en cualquier orden, sin provocar errores.

1.3 AGENTE

Un agente es un sistema informático que se encuentra situado en un entorno y es capaz de percibir a través de sensores y actuar en dicho medio a través de actuadores de manera autónoma. Cuando hablamos de que perciben nos referimos a que son capaces de recibir entradas en cualquier momento, y la decisión a tomar vendrá dada por esta secuencia de percepción, tanto pasadas como actuales.

Tienen normalmente una función de decisión, que asociará las percepciones con las acciones a realizar. Esta función normalmente viene dada en forma de tabla. La implementación de una tabla en un agente concreto con un lenguaje concreto se denomina programa del agente.

1.3.1 Bucle de funcionamiento de un agente

Percepción - Se obtiene información a través de sensores del entorno.

Decisión - Se toma una decisión en base a lo percibido y a los objetivos.

Actuación - Se mandan las órdenes a los actuadores de acuerdo con lo decidido.

Aprendizaje (opcionalmente) – En base al feedback devuelto del entorno por las actuaciones decididas se aprende acerca las decisiones a tomar en el futuro.

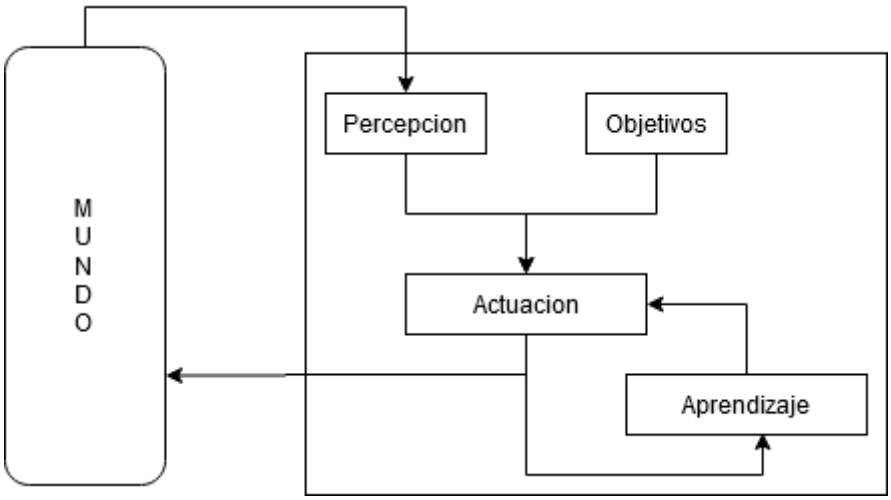


Figura 2 - Bucle básico de funcionamiento de un agente

1.3.2 Propiedades de los entornos

Los agentes actúan en entornos muy diversos y variados. Sin embargo, podemos identificar una serie de características que nos permitirán categorizar estos entornos ((Russel & Norvig, 2004):

- **Totalmente observable vs Parcialmente observable**: En un entorno accesible, el agente puede recibir información completa, precisa y actualizada sobre el entorno. Por norma general, la mayoría de los entornos no son de este tipo, como, por ejemplo, el propio mundo real o Internet.
- **Determinista vs Estocástico**: En un entorno determinista, cada acción posible tiene un único efecto garantizado en el entorno, es decir, no existe incertidumbre sobre el estado que será resultado de la acción a realizar.
- **Estático vs Dinámico**: En un entorno estático, se asume que permanecerá sin cambios salvo por las acciones del agente. En un entorno dinámico, existen otros procesos que cambiarán el entorno sobre los que el agente no tiene control alguno.
- **Episódico vs Secuencial**: En un entorno episódico, la experiencia del agente se divide en episodios, y cada episodio conlleva una única acción a lo percibido, sin relación con otros episodios futuros o pasados. En un entorno secuencial, las decisiones presentes pueden afectar a decisiones futuras.
- **Discreto vs Continuo**: Un entorno discreto es aquel con un número finito y fijo de acciones y percepciones posibles.

1.4 SISTEMA MULTIAGENTE

Sistema compuesto de varios agentes inteligentes que interactúan entre ellos, normalmente a través del paso de mensajes a través de algún tipo de red. Típicamente, estos agentes tendrán distintos dueños/usuarios, por lo que cada uno de ellos tendrá unos objetivos propios e independientes. Para que puedan interactuar con éxito deberán poder cooperar, coordinar y negociar.

1.4.1 ¿Qué hace a un agente inteligente?

Hemos especificado que los agentes son inteligentes, pero ¿qué quiere decir que un agente sea inteligente? Un agente inteligente es aquel que toma la mejor decisión en cada momento de acuerdo con sus objetivos y necesidades.

Matemáticamente, si definimos una medida de rendimiento, las mejores decisiones son aquellas que llevan a maximizar dicha medida de rendimiento. Esto nos plantea un problema a la hora de diseñar, y es que no va a existir una única medida adecuada para todos los agentes, y es muy importante que a la hora de diseñar los agentes que se decida objetivamente dicha medida.

Podemos también definir en base a qué capacidades debe poseer un agente inteligente. Para que un agente sea inteligente, debe de tener las tres siguientes características (Wooldridge & R. Jennings, 1995):

Reactividad: Deben ser capaces de percibir el entorno y de responder a cambios que ocurran en este para satisfacer sus objetivos y necesidades.

Proactividad: Deben de ser capaces de tomar la iniciativa para satisfacer sus objetivos y necesidades.

Habilidad social: Son capaces de interactuar con otros agentes para satisfacer sus objetivos y necesidades. Para ello los agentes deben negociar y cooperar con otros agentes.

El principal problema residirá pues en crear un balance adecuado entre proactividad (comportamiento dirigido a cumplir con objetivos) y reactividad (capacidad de percibir y responder a cambios).

1.4.2 Características de un entorno multiagente

Diremos que un entorno es multiagente cuando cumpla con las siguientes características:

- Proveen la infraestructura necesaria para la comunicación e interacción entre los distintos agentes.
- Suelen ser típicamente abiertos, descentralizados y dinámicos.
- Contienen agentes autónomos y distribuidos, que pueden tener intereses propios o comunes y que en base a esos intereses negociaran o cooperaran entre ellos.

1.5 RELACIÓN ENTRE AGENTES Y OTROS CONCEPTOS

1.5.1 Diferencia entre agente y objeto

Se puede tener la idea de que un agente y un objeto no tienen en sí ninguna diferencia. Sin embargo, esto no es así. Definimos un objeto como una entidad computacional que encapsula un estado y es capaz de realizar acciones o métodos sobre ese estado, y que es capaz de comunicarse mediante el paso de mensajes. Es cierto que ambos tienen similitudes, pero existen tres grandes diferencias que separan a un agente de un objeto como tradicionalmente se conoce:

- Los agentes tienen una mayor autonomía que los objetos, en concreto, deciden por sí mismos cuando o no ejecutar una acción.
- Los agentes son capaces de tener un comportamiento más flexible, como comportamientos reactivos, proactivos o sociales, mientras que el objeto tradicional no es capaz de tener dichos comportamientos.
- Un sistema multiagente es inherentemente multihilo, donde cada agente tiene control sobre al menos un hilo.

1.5.2 Diferencia entre agente y sistema experto

Un sistema experto es capaz de resolver problemas o dar consejos en un dominio sobre el que se tiene gran conocimiento. La principal diferencia que tienen estos sistemas con los agentes es que estos sistemas son incorpóreos, es decir, no perciben ni actúan sobre un entorno. Necesitan de un intermediario que les proporcione la información que estos sistemas necesitan y no actúan sobre el entorno, sino que dan un feedback o consejo a un tercero que es el que decide o no actuar. Así, las diferencias serían las siguientes:

- Los sistemas clásicos expertos son incorpóreos, es decir, no se sitúan en un entorno.
- Por norma general, los sistemas expertos no son capaces de tener un comportamiento reactivo o proactivo.
- Los sistemas expertos no tienen la capacidad de habilidad social, en el sentido de cooperación, coordinación y negociación.

1.5.3 Sistemas distribuido y sistemas multiagente

Para empezar, hay que decir que es necesario emplear conocimiento y técnicas de los sistemas distribuidos a la hora de implementar los sistemas multiagentes, puesto que es necesario emplear técnicas y estrategias como la exclusión mutua de recursos. Sin embargo, existen dos grandes diferencias:

En los sistemas multiagentes, puesto que cada uno es autónomo, es necesario crear un sistema de sincronización y coordinación capaz de ejecutarse en tiempo de ejecución.

En un sistema distribuido todos los elementos buscan cumplir un objetivo común, mientras que en un sistema multiagente cada agente puede tener un objetivo propio distinto al del resto de agentes. Aquí es donde entran la capacidad de los

agentes de negociar para que cada uno cumpla sus objetivos y de cooperar para cumplir los objetivos comunes.

Dos grandes problemas:

- Cómo diseñamos estos agentes para que sean capaces de actuar de manera independiente
- Cómo diseñamos estos agentes para que sean capaces de interactuar con otros agentes

1.6 PARADIGMAS DE AGENTES

1.6.1 Agentes puramente reactivos

Son aquellos en los que la decisión a tomar viene basada exclusivamente en el presente y en el estado del entorno actual. Formalmente, sea E el conjunto finito de estados del entorno:

$$E = \{e_0, e_1, \dots, e_n\}$$

Y Ac el conjunto de posibles acciones:

$$Ac = \{\alpha_0, \alpha_1, \dots, \alpha_{n\}}$$

Definimos la función de actuación Ag como:

$$Ag: E \rightarrow Ac$$

La función queda pues como una secuencia de estados a los que les corresponde una acción. Por ejemplo, un termostato podría definirse como un agente reactivo con la siguiente función de actuación:

$$Ag(e) = \{Calentador\ apagado \quad si\ e \geq temperatura\ OK\ Calentador\ encendido \quad si\ e < temperatura\ OK$$

1.6.2 Agentes con estado

Los agentes guardan un estado interno que se modifica en base a lo que se percibe, y la acción a realizar vendrá dada por el estado actual. La percepción del entorno, junto al estado actual, determinarán el estado siguiente.

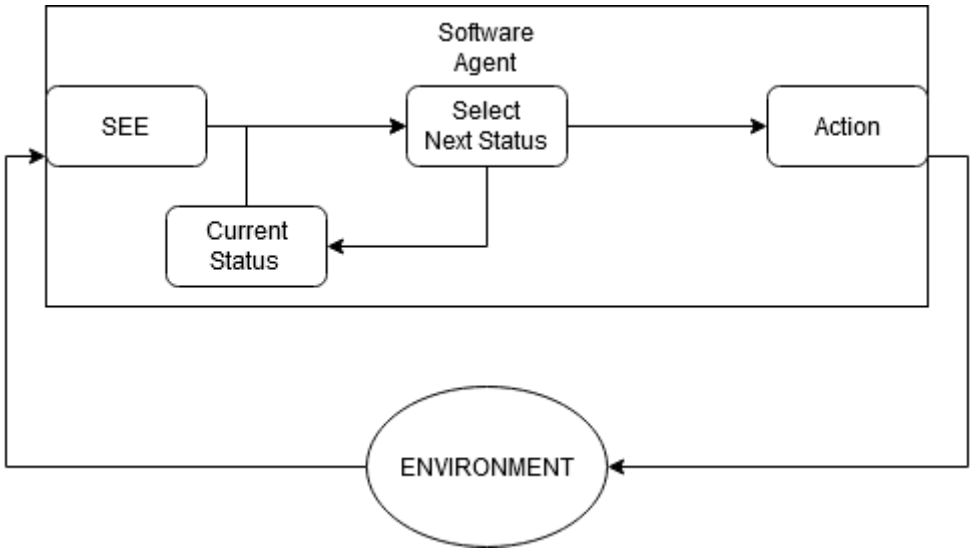


Figura 3 - Diagrama de actuacion de agente con estaod interno

El agente dispondrá de un estado interno y de tres funciones para su comportamiento. Sea *Per* el conjunto de percepciones del entorno, *I* el conjunto de estados internos, *E* el conjunto de estados de entorno y *Ac* la lista de acciones:

- Función "ver": percibe el estado del entorno y obtiene unas percepciones.

$$ver: E \rightarrow Per$$

- Función "siguiente": decide el siguiente estado en base al estado actual y a lo percibido.

$$siguiente: I\ x\ Per \rightarrow I$$

- Función "acción": decide como actuar en base al estado.

$$accion: I \rightarrow Ac$$

1.6.3 Agentes deliberativos

Los agentes deliberativos suelen basarse en la teoría clásica de planificación de IA: dado un estado inicial, un conjunto de acciones/planes y un estado objetivo, la

deliberación del agente consiste en determinar qué pasos debe encadenar para lograr su objetivo (Skarmeta, Pujol, & Ramón, s. f.).

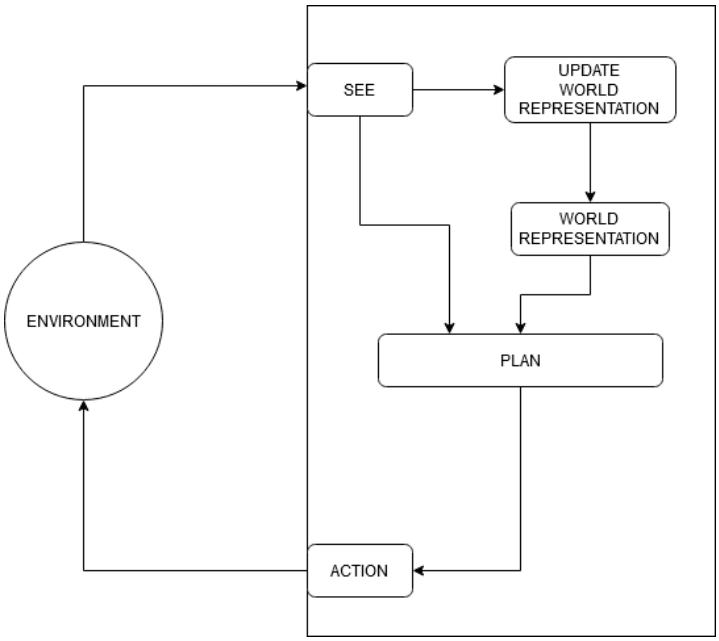


Figura 4 - Diagrama actuación agentes deliberativos

De esta forma, un agente en un momento dado tiene un objetivo, percibe el mundo y crea en base a lo que percibe el modelo del mundo. Con este modelo puede planificar que pasos debe tomar para acercarse a cumplir este objetivo, y tras planificar, ejecuta dichas acciones. Este proceso se repetirá hasta cumplir con el objetivo.

El problema de estos agentes está en que el mundo es demasiado complejo como para guardar un modelo de este. Además, el hecho de que se guarde con mayor complejidad implica una lentitud en el proceso, mientras que un modelo con menor complejidad puede llevar a que este incompleto o a que exista incertidumbre sobre el entorno.

1.6.4 Agentes BDI

La toma de decisiones se realiza en base a las creencias del agente sobre el mundo y teniendo en cuenta las intenciones y las acciones posibles.

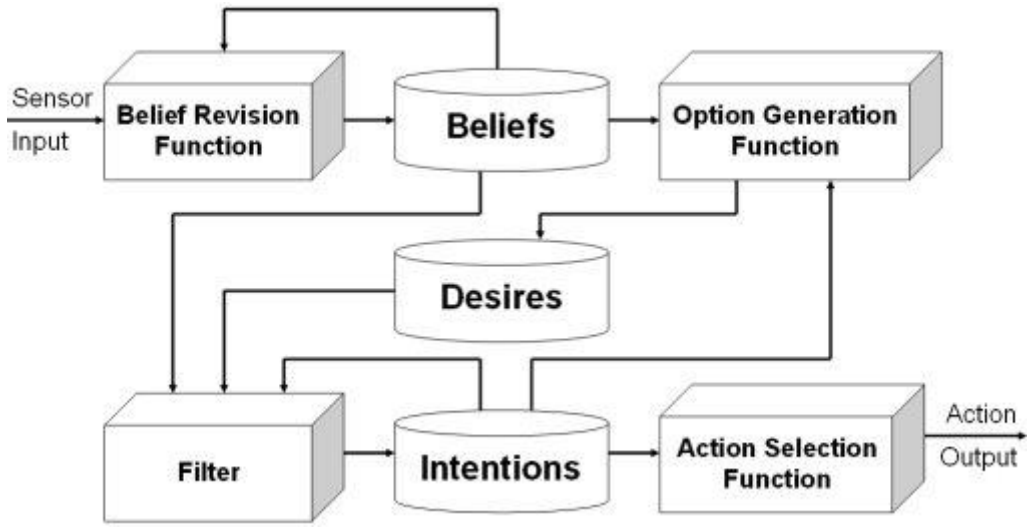


Figura 5 - Esquema agente BDI

Los componentes de estos agentes son los siguientes:

- Creencias (belief): que sabe el agente sobre el mundo.
- Deseos (desire): representan objetivos que el agente quiere cumplir. Se podrían llamar 'Objetivos a largo plazo'.
- Intenciones (intention): representan el estado de deliberación del agente. Es decir, representan qué ha decidido el agente que va a cumplir ahora. Se tratarían de aquellos 'Objetivos a corto plazo' que sirven para alcanzar los a largo plazo.
- Acciones
- Funciones de deliberación: decidir qué metas se quieren conseguir.
- Razonamiento de fines y medios: decidir qué acciones tomar para cumplir con las intenciones.

El proceso sería el siguiente: el agente en primer lugar en base a sus creencias y sus deseos emplea un proceso deliberativo para ver que posibles opciones tiene (intenciones) y para comprometerse (commit) con una de ellas (decidir intenciones). Una vez se ha comprometido, se decidirá qué acciones se van a tomar para cumplir con estas intenciones que tiene actualmente, descartando todas ellas

que no ayuden a cumplir con las intenciones. Además, deberá persistir sobre estas intenciones para cumplirlas.

La toma de decisión acerca de las intenciones viene ligada a los deseos y a las creencias del entorno, puesto que no se elegirán intenciones que se crean que son difíciles o imposibles de cumplir. Las creencias y deseos se irán actualizando conforme se vaya sintiendo del mundo.

Debe existir, sin embargo, un balance entre comprometerse a cumplir con las intenciones y en pararse cada cierto tiempo a reconsiderar estas intenciones, puesto a que se pueden haber cumplido o porque puede que con las creencias nuevas que se han obtenido del mundo se haya descubierto que es imposible de realizar. Es decir, debe existir un balance adecuado entre proactividad (cumplir intenciones) y reactividad (reaccionar a nuevos eventos en el entorno).

1.6.5 Agentes lógicos

En los agentes lógicos, la toma de decisiones se trata como una deducción lógica. Se tiene una base de datos de fórmulas lógicas de primer orden que es el estado del agente y que representa la información que tiene el agente sobre el entorno. Así, la toma de decisiones vendría dada por el estado de la base de datos y un conjunto de reglas de deducción que tiene el agente.

El uso de agentes lógicos provee una solución elegante con una semántica clara, pero nos trae a su vez varias desventajas:

- No son instantáneos: la complejidad del cómputo en el proceso de demostración de teoremas lógicos hace que el agente sea lento en la toma de decisiones, y en problemas en tiempo real donde el entorno puede cambiar en periodos de tiempo muy cortos hace que sea inviable el uso de estos agentes.

- Representación del entorno: estos agentes hacen uso de una representación simbólica del entorno, que en casos muy complejos y dinámicos puede hacer que la relación entre los simbolismos y el entrono real no esté clara.
- Razonamiento sobre el entorno: la representación de como el agente tiene que proceder en un entorno es poco intuitiva en casos simples, y muy compleja y difícil de entender cuando hablamos de entornos dinámicos o si tenemos el tiempo en cuenta.

1.6.6 Agentes híbridos

Son aquellos que combinan distintos paradigmas para la toma final de decisiones, usando distintos módulos que emplean distintos paradigmas en conjunto para tomar una decisión final. Por ejemplo, un agente híbrido puede tener emplear el paradigma reactivo para entradas simples que requieran de una actuación rápida y el paradigma deliberativo para decidir más a largo plazo y lograr un balance entre reactividad y proactividad.

Precisamente por este balance entre reactividad y proactividad suele ser el paradigma más empleado a la hora de diseñar agentes.

1.6.7 Agentes reactivos a capas

En una agente a capas, la toma de decisiones está particionada en varias capas de abstracción donde cada una de ellas trata el entorno de manera distinta. Así se crean subsistemas donde cada uno se encarga de tratar un comportamiento distinto, y así poder lograr un balance entre proactividad y reactividad.

Existen dos tipos de capas:

- Capas horizontales. Capas directamente conectadas con unos sensores y unos actuadores. Cada capa genera sugerencias sobre posibles acciones a tomar.

- Capas verticales. Se encargan de transformar y manipular las entradas sensoriales y actuaciones para abstraer el entorno. Tendríamos dos tipos, de un paso o de dos pasos.

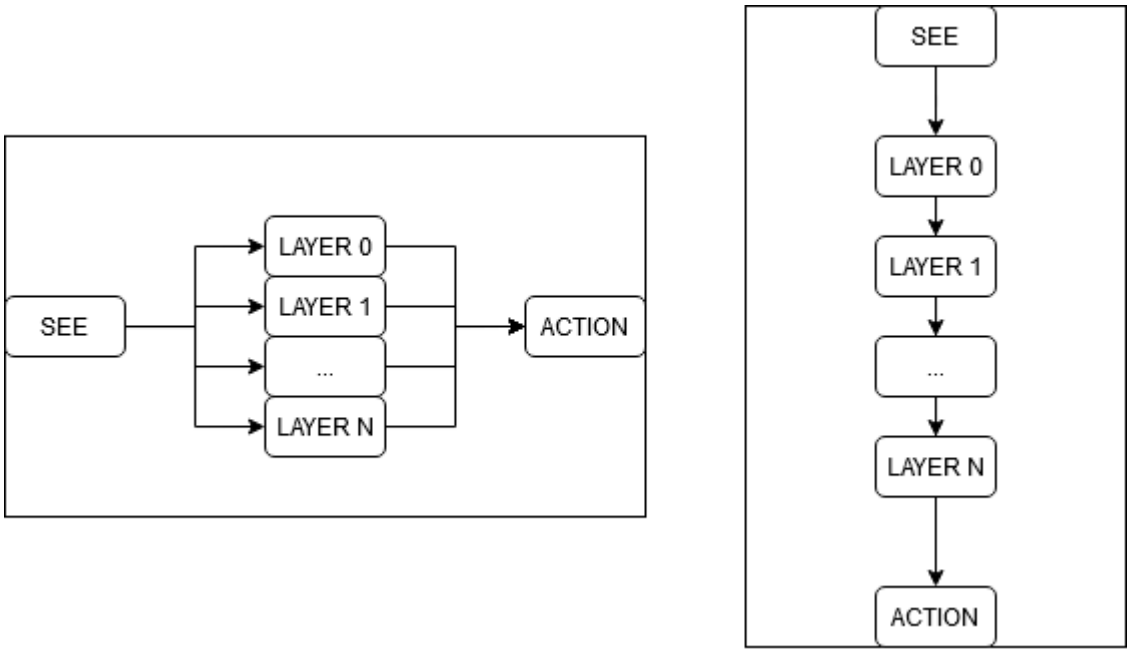


Figura 6 - Agente de capas horizontales (izq) y agente de capas verticales de un paso (der)

1.7 ARQUITECTURAS DE AGENTES

Al tratar los agentes de manera abstracta no hemos entrado en detalle sobre en cómo es este estado interno o sobre cómo implementamos la función de actuación. Vamos ahora a explicar distintos tipos de agente:

1.7.1 TouringMachines

Se trata de una arquitectura organizada en capas horizontales, donde existen 3 capas generadoras de actividades y un subsistema de control que se encarga de decidir cuál de los 3 sistemas toma el control de las acciones del agente. Se implementa como un conjunto de reglas de control.

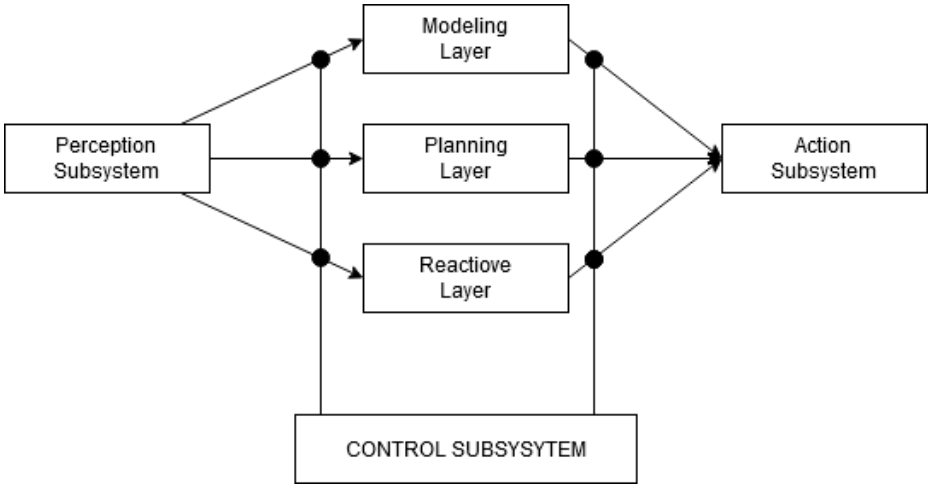


Figura 7 - Estructura TOURINGMACHINE

La **capa reactiva** dota al agente de una respuesta más inmediata al entorno que se implementa como un conjunto de reglas situación-acción.

La **capa de planificación** sirve para dotar de un comportamiento proactivo al agente, donde se emplean un conjunto de reglas llamadas esquemas, que se componen de planes ordenados jerárquicamente. Para lograr el objetivo actual, se busca el plan que más se adecue a dicho objetivo, que a su vez tendrán subobjetivos que la capa empleará para encontrar correspondencias entre los esquemas.

Por último, la **capa de modelado** representa las diversas entidades del mundo, incluido el propio agente, e intenta predecir conflictos entre agentes y generar en base a estos conflictos nuevos objetivos que son pasados a las capas de planificación y reactividad.

1.7.2 Interrap

Interrap es un ejemplo de arquitectura en capas verticales en dos pasos. En esta, igual que en Touring Machines, existen 3 capas de control que se encargan de tomar las decisiones, que se corresponden con las mismas que en Touring Machines: reactiva, para comportamiento inmediato, planificación, con planes para lograr objetivos, y modelado, que se encarga de las interacciones sociales.

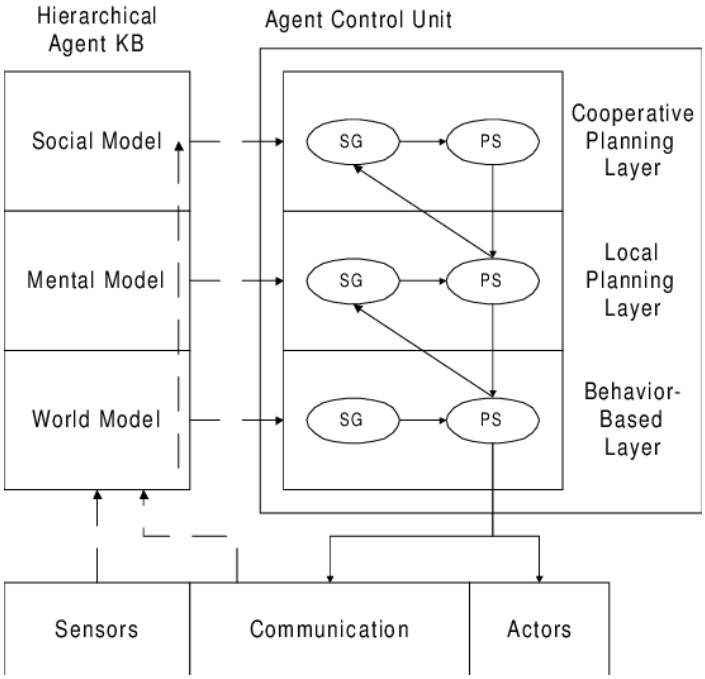


Figura 8 – Arquitectura INTERRAP

¿Dónde radica la diferencia respecto a Touring Machines? Para empezar, existen 3 representaciones del mundo que corresponden con cada capa, que representan al agente en distintos niveles de abstracción. Este conocimiento, explícitamente declarado, se contrapone a Touring Machines, donde cada capa está conectada directamente a los sensores.

La otra principal diferencia es en cómo se escoge el comportamiento a realizar. En Touring, existe un subsistema de control que se encarga de seleccionar que capa toma el control en cada momento. En Interrap, sin embargo, las capas interactúan entre ellas para decidir cómo actuar, empleando o bien interacción bottom-up o top-down.

Cuando la información llega a una capa, esta comprueba la información que le llega. Si con la información de la que dispone no puede tomar una decisión, manda la información a la capa superior (interacción bottom-up). Cuando una capa superior toma la decisión de cómo actuar, manda la información necesaria a la capa de abajo (top-down) para que cumpla con uno se sus objetivos y actúe para

cumplirlo. De esta forma el control en Interrap va desde la capa inferior a alguna de las superiores si es necesario y luego de vuelta hacia abajo.

Cada una de las capas implementa, además de las reglas propias para el conocimiento y actuación, dos funciones generales:

- Función de reconocimiento y activación de objetivos, que se encarga de mapear conocimiento y objetivos actuales con nuevos objetivos.
- Función de planificación y organización que, en base a un plan actual, unos objetivos y un conocimiento del mundo decide qué plan ejecutar a continuación en la capa actual.

1.7.3 AuRA

La arquitectura AuRA (C. Arkin & Balch, 1997) es una arquitectura que se diseñó para utilizar una aproximación híbrida a los sistemas de navegación de robots. Se compone en dos capas diferenciadas, una superior, la deliberativa, que contiene el subsistema planificador y el sistema cartográfico y una capa inferior, la reactiva, que contiene el subsistema motor y el subsistema de percepción. Entre ambas se encuentra el subsistema homeostático que modifica los comportamientos en función de las necesidades internas.

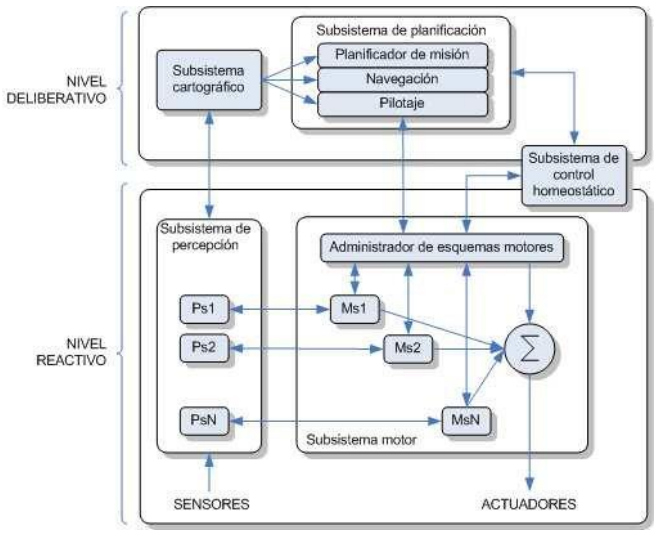


Figura 9 - Arquitectura AuRA

Los subsistemas que se muestran realizan las siguientes funciones:

- Planificador: se encarga de escoger objetivos y de planificar las tareas necesarias para lograrlos.
- Cartográfico: contiene las funciones de modelado, y empleando la información leída por los sensores se encarga de crear un modelo del mundo que poder emplear.
- Motor: se encarga de modelar los esquemas motores que representan los comportamientos.
- Percepción: esquemas de percepción que adquieren información de los sensores. Esta información es la que se manda al sistema cartográfico.
- Homeostático: modifica los comportamientos en base a necesidades internas, por ejemplo, en caso de peligro inminente.

1.7.4 Saphira

La arquitectura Saphira (Konolige & Myers, 1996) nace para la construcción de agentes móviles autónomos. Se trata de una arquitectura en dos capas en torno a un componente central de representación interna, el espacio LPS (Local Perceptual Space).

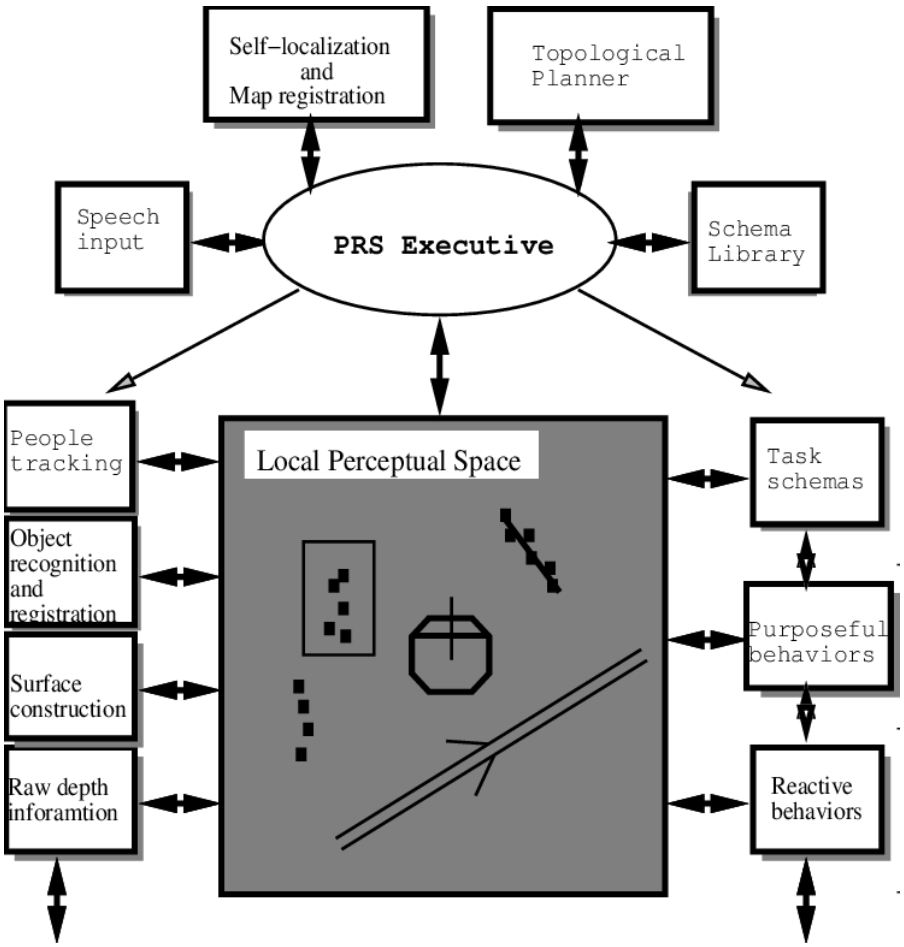


Figura 10 - Arquitectura Saphire

Las partes de las que se compone esta arquitectura son las siguientes:

- Rutinas de percepción: se encargan de procesar y añadir información de los sensores al LPS.
- Comportamientos reactivos: se definen empleando lógica difusa, que da como resultado reglas que se combinan para obtener las acciones a realizar.
- LPS: eje central de la arquitectura, sirve para representar en varios niveles de abstracción el espacio local del agente en el mundo.
- Localización y mantenimiento mapa: sirve para localizar al agente dentro del mundo.
- PRS-Lite: (Procedural Reasoning System) Sistema basado en el paradigma BDI para la deliberación de acciones y procedimientos. Se encarga de definir los objetivos y subobjetivos a realizar.

- Planificador: transforma los subobjetivos en tareas a realizar.
- Tareas: almacena las tareas a realizar y manda información al sistema de comportamientos para que puedan decidir cómo actuar.

1.7.5 3T

La arquitectura 3T es otro ejemplo de arquitectura en 3 capas, como InteRRaP o TouringMachines. Se compone de 3 capas: la capa de habilidades reactivas, la capa de secuenciación y la capa de deliberación.

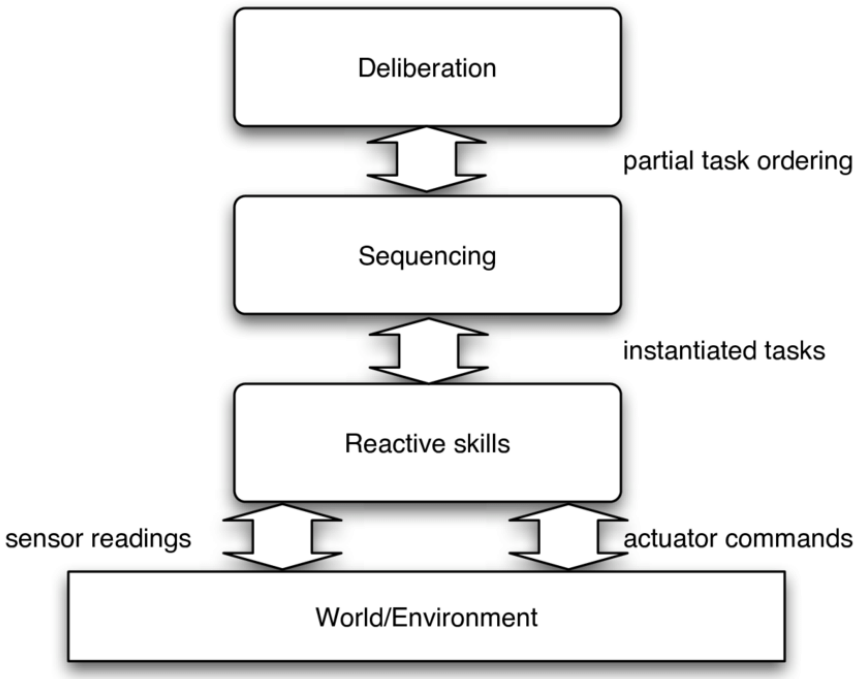


Figura 11 - Arquitectura 3T

Una habilidad es un comportamiento primitivo que encapsula algún tipo de habilidad. La capa de habilidades reactivas contiene todas estas habilidades, que pueden ser llamadas a ejecución en cualquier momento.

La capa de secuenciación se encarga de instanciar y seleccionar habilidades, y la capa de deliberación es la responsable de generar planes de alto nivel/abstracción. Es la que se encarga también de modelar el mundo.

Así, el esquema de funcionamiento sería el siguiente:

- La capa deliberativa escoge, en base a el modelo, plan y objetivos actuales un plan de objetivos nuevos. Estos planes están compuestos de una serie de subobjetivos, y para el cumplimiento de cada subobjetivo se requieren una serie de tareas.
- La información de estas tareas se manda a la capa de secuenciación, que se encarga de seleccionar las habilidades necesarias para lograr cumplir con ellas.
- Finalmente, la capa de secuenciación va instanciando las habilidades necesarias en la capa reactiva conforme vaya siendo necesaria.
- En todo momento existe feedback y comunicación entre las distintas capas, de forma que se pueda replanificar y tomar decisiones conforme va cambiando el mundo.

La comunicación entre capas se realiza de manera asíncrona, de tal forma que se puede así dotar de inmediatez a las capas inferiores mientras que algoritmos y procesos más lentos se dejan en capas superiores.

1.8 SISTEMAS MULTIAGENTES / SOCIEDADES DE AGENTES

A la hora de trabajar con agentes, estos se encuentran en un entorno que puede contener otros agentes, y surge la necesidad de entre ellos comunicarse e interactuar para poder funcionar de manera efectiva y productiva.

Podemos distinguir dos grandes temas a tratar con relación a esto: protocolos de comunicación, que especifican como deben comunicar y entender mensajes, y protocolos de interacción, que nos dicen que mensajes y con qué objetivo deben mandarse los agentes entre sí para coordinarse.

1.8.1 Métodos de comunicación entre agentes

Método de la pizarra: la pizarra es una estructura de datos centralizada que es usada como mecanismo general de comunicaciones. Se trata de un área común donde los agentes pueden escribir su información propia y leer de la información que otros agentes escriben.

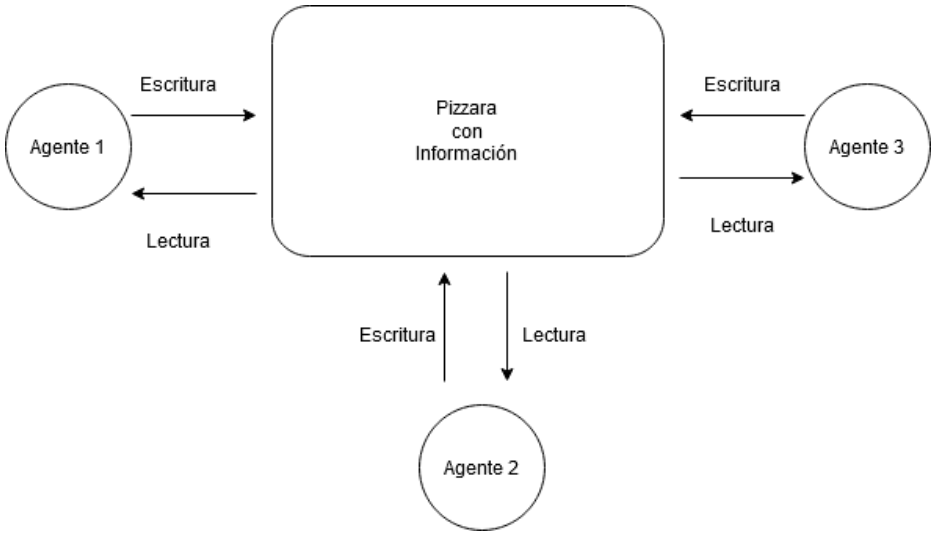


Figura 12 - Método de la pizarra para la comunicación

En este método no existe una comunicación directa entre agentes. Cuando estos requieran de más información deberán acceder a la pizarra sí o sí. Esto origina el problema principal de este método, y es que esta pizarra forma un cuello de botella

importante en la red, y pese a que en estudios posteriores se modifica el ejemplo básico empleando moderadores o más de una pizarra, este problema persiste.

Método de mensajes: forma una base muy flexible para la comunicación entre agentes. Los mensajes son intercambiados entre los agentes directamente, y pueden ser empleados para establecer tanto comunicaciones como protocolos y mecanismos de cooperación.

Para que este mecanismo funcione es necesario establecer una serie mecanismos y estrategias comunes a todos los agentes, en concreto, el proceso de comunicación y la semántica del lenguaje común.

1.8.2 Tipos de mensaje

Para poder comunicarse de manera efectiva, los agentes deben poseer distintas capacidades, la principal la de establecer un diálogo con otros agentes. Este diálogo corresponderá a mensajes de distintos tipos que se intercambiarán los agentes, funcionando alternativamente bien de forma pasiva (receptores) o bien de forma activa (emisores).

En base a esta división podemos definir dos tipos básicos de mensajes: preguntas y afirmaciones. Con estos dos mensajes, un agente activo puede solicitar información o expresar un deseo mediante una pregunta y recibir una afirmación de un agente pasivo confirmando la realización de una acción o la veracidad de la información.

Estos corresponderían a los mensajes básicos, pero de estos pueden derivarse muchos otros derivados de la teoría de actos de habla.

1.8.3 Actos de habla

La comunicación entre agentes toma como base la comunicación entre humanos, y uno de los modelos más utilizados para formalizar y analizar la comunicación

humana es el modelo de actos de habla. Esta representa el lenguaje humano como acciones. Por ejemplo, cuando un juez declara a alguien culpable la acción que sucede es que el estado social del acusado cambia.

Se definen 3 aspectos que componen un acto de habla:

- Locución: la parte física del acto correspondería a las palabras emitidas o escritas
- Ilocución: la intención del emisor, es decir, que intenta transmitir el emisor
- Perlocución: la acción que deriva de la locución.

En la comunicación humana, la intención del mensaje puede no estar clara para un receptor. Por ejemplo, al decir 'Tengo hambre' mi intención puede ser la de informar o la de querer ir a un restaurante a comer. Sin embargo, para la comunicación entre agentes buscamos que no exista ningún tipo de duda en el tipo de mensaje. Para identificar la intención ilocutora del mensaje se emplean verbos 'performativos', como decir, demandar, convencer, etc, que declaran por si mismos la fuerza ilocutora.

La solución consiste pues en restringir la semántica de los mensajes mediante el uso de estos verbos para que no exista duda. Esto no quiere decir que el contenido del mensaje pueda o no ser ambiguo, pero donde no existirá duda es en el tipo de mensaje empleado dentro del protocolo de comunicación. De esta forma, distinguimos cinco clases de actos de habla:

- Representativos: el hablante niega, afirma o corrige. Ejemplo: afirmar.
- Directivos: intento del hablante de que el oyente realice una acción. Ejemplo: petición.
- Compromisorios: el hablante asume un compromiso o responsabilidad. Ejemplo: prometer.
- Expresivo: el hablante expresa su estado psicológico. Ejemplo: agradecer.

- Declarativo: el hablante pretende cambiar el estado de algo. Ejemplo: declarar guerra.

1.8.4 Ontologías

Una ontología es una especificación de objetos, conceptos y relaciones dentro de un área de interés. Define un conjunto de clases de objetos y la relación que existe entre ellos, pero no las posibles instancias concretas de estos objetos.

Las ontologías sirven para declarar el cuerpo de un área de conocimiento. Normalmente suele emplear el uso de clases y subclases relacionadas mediante una definición de la relación que existe entre ellas. Estas ontologías vienen dadas en distintos grados de formalidad, que harán que sean más o menos complejas.

A la hora de la comunicación entre agentes, el uso de una ontología común es imprescindible para que el intercambio de información exista. En los mensajes, además del propio mensaje dado en una cierta ontología, se deberá especificar la ontología empleada para que dicho mensaje no se confunda y se pueda extraer de él la información necesaria.

1.9 LENGUAJES DE AGENTES

1.9.1 KQML

Knowledge Query and Manipulation Language (KQML) es un lenguaje para la comunicación entre agentes que define el formato del mensaje de forma similar a un objeto: cada mensaje tiene una *performative,* que puede entenderse como la clase de mensaje (Informar, mostrar acuerdo, mostrar desacuerdo…), y una serie de parámetros en formato pareja atributo / valor.

Veamos un ejemplo de mensaje:

```
(ask-one
     :content (PRICE IBM ?price)
     :receiver stock-server
```

```
                    :language LPROLOG
                    :ontology NYSE-TICKS
                )
```

La *performative* correspondería a `ask-one` un mensaje de pregunta, y el valor que pregunta viene dado por el contenido, en este caso, el precio de IBM. El receptor sería el `stock-server`, y se emplea el lenguaje `LPROLOG` y la ontología `NYSE-TICKS`.

KQML fue empleado durante mucho tiempo por la comunidad, y surgieron varias implementaciones y distribuciones de este lenguaje. Sin embargo, fue bastante criticado por algunos por varios aspectos:

- Las distintas implementaciones de KQML, precisamente por su flexibilidad, no eran interoperables entre sí.
- No se definen mecanismos de transporte de mensajes KQML.
- La semántica de KQML no se define de forma rigurosa, de tal forma que si dos agentes empleaban KQML con semánticas distintas no se podía decir que ninguno de los dos lo estuviese empleando de manera incorrecta.
- El conjunto de *performatives* era bastante grande.
- A pesar de ser un conjunto grande, no existían las *performative* del tipo compromiso, dificultando en ocasiones la coordinación de sistemas multiagentes.

Estas críticas llevaron al diseño de un nuevo lenguaje bastante similar por la asociación FIPA.

1.9.2 FIPA

La Fundation for Intelligen Physical Agents (FIPA) se fundó en 1999 con el objetivo de desarrollar los estándares necesarios para los sistemas de agentes. El eje central consistía en un lenguaje de comunicación entre agentes (ACL) que sacó su primera versión en 1999. La versión actual de lenguaje salió en 2002.

FIPA es similar en varios aspectos a KQML: define un lenguaje para especificar la capa exterior del mensaje y no el contenido, el cual deja a libre elección y la sintaxis es similar a KQML.

```
(request
        :sender Agent_A
        :receiver Agent_B
        :content
                (….)
        :in-reply-to action
        :replay-with reponse
        :language FIPA-SL0)
```

Figura 13 - Ejemplo de mensaje FIPA

Una de las diferencias radica en el conjunto de *performatives* que define cada lenguaje. Los siguientes ejemplos corresponden a algunas de las performatives más importantes del lenguaje FIPA:

REQUEST	The sender asks the receiver to perform some action.
AGREE	The sender agrees to perform the requested action
REFUSE	The sender refuses to perform the requested action.
NOT_UNDERSTOOD	Someone participating in the conversation didn't understand something
INFORM	A positive reply because an action is performed or the answer to some pervious question
FAILURE	A failure in performing the requested action happened after an initial agreement.

Figura 14 – Ejemplo de performatives de lenguaje FIPA

La otra principal diferencia radica en la semántica de los lenguajes. FIPA establece una semántica formal para su lenguaje

1.10 PROTOCOLOS DE INTERACCIÓN ENTRE AGENTES

Tal y como explicábamos al principio de este apartado, los agentes deben comunicarse entre ellos para lograr cumplir con sus objetivos, y para esto, deben

decidir qué agente va a realizar que acciones. Los protocolos de interacción nos especifican que mensajes deben mandarse entre los agentes para decidir sobre las acciones y como deben coordinarse entre sí los agentes.

Existen multitud de protocolos de interacción, pero podemos generalizar y agrupar estos protocolos según si:

- Los agentes tienen objetivos comunes o similares, en cuyo caso deberán **cooperar** entre ellos.
- Los agentes tienen conflicto a la hora de resolver los objetivos y deben **negociar** entre ellos

1.10.1 Cooperación

Los agentes tienen objetivos comunes y similares y la actuación de todos los agentes en conjunto es necesaria para reducir la complejidad de la tarea. Se realiza así un proceso de "Divide y vencerás" donde este objetivo común se divide en subtareas, y los agentes se coordinan para realizar estas tareas y lograr cumplir con el objetivo. Así, la dificultad radicará en cómo realizar una correcta descomposición de las tareas teniendo en cuenta recursos y capacidades de agentes, así como que puede que ciertas descomposiciones causen conflicto entre los agentes. Esta descomposición puede venir dada por el diseñador del sistema, de forma que pueda darse especificada en la propia implementación del sistema o bien realizada por planificación de los agentes.

Por último, estas tareas deben distribuirse en los agentes de forma que se cumplan requisitos como el uso coordinado de recursos críticos, priorización en caso de tareas urgentes cuando vayan apareciendo, informar a otros agentes cuando se cumplan las tareas, etc.

Algunas de las técnicas empleadas para la distribución son, por ejemplo:

- Sistemas de subasta: las tareas se van ofertando y los agentes pujan por ellas, asignándose al que mejor puja realice. El agente realiza la tarea e informa cuando este completa.
- Mecanismos de mercado: las tareas se encargan a agentes por un acuerdo general o por aceptación mutua.

1.10.2 Negociación

Cuando dos o más agentes intentan cumplir con sus objetivos y estos causan conflicto entre ellos, los agentes deben negociar entre ellos para llegar a un acuerdo conjunto, comunicando entre ellos sus situaciones y buscando concesiones y alternativas para lograr este acuerdo.

Existen numerosas técnicas y sistemas de negociación, pero podemos clasificarlos en dos categorías: centrados en el entorno, es decir, como diseño un entorno para que independientemente de los agentes puedan negociar entre ellos, y centrados en el agente, que sería lo opuesto, como dado un entorno puedo operar de la mejor manera posible.

Por último, hay que recalcar que para que un protocolo de negociación funcione correctamente, tiene que cumplir con las siguientes características:

- Eficiencia: no se deben gastar recursos para llegar a un acuerdo.
- Estabilidad: los agentes no deben tener incentivos para desviarse de las estrategias de acuerdo.
- Simplicidad: bajo coste tanto computacional como de ancho de banda.
- Distribución: no debe existir un mecanismo central de decisión.
- Simetría: no se debe favorecer a ningún agente por motivos arbitrarios.

1.11 SOCIEDADES DE AGENTES

Las aproximaciones tradicionales de IA se han centrado en cómo construir un agente para que opere como un sistema inteligente. Sin embargo, estos agentes

suelen trabajar en un entorno que contiene otros agentes inteligentes. Por tanto, una aproximación a estos sistemas consiste en verlos en términos sociales. Los son muy grandes, complejos, dinámicos y abiertos, y una aproximación centralizada o tradicional no es viable en estos entornos. Debe distribuirse en agentes que vayan funcionando de manera autónoma, que sean capaces de ir explorando el entorno por su cuenta y obtener información de él y de ser capaces de interactuar y coordinarse con otros agentes.

Un grupo de agentes puede formar una sociedad donde cada uno cumple con un rol distinto. El grupo define estos roles, y cuando un agente nuevo llega asume uno o más roles y con estos sus respectivas responsabilidades. Es aquí donde este estudio de sistemas multiagentes centrándose en la sociabilidad toma relevancia para las tareas de cooperación.

Las responsabilidades sociales son aquellas que tiene un agente con otros agentes. Son distintas de las propias del agente, y suelen, por su naturaleza, restringir el comportamiento del agente debido a que existe una dependencia de otro agente.

Por ejemplo, un agente X puede depender de otro Y para lograr un objetivo P, y este objetivo no podrá cumplirse hasta que Y no realice A.

Estas dependencias se van creando de manera natural conforme se asignan roles. Se van creando así relaciones jerárquicas donde un agente dependerá de otro para poder cumplir con sus objetivos. Además, puede darse el caso de que existan relaciones bidireccionales, donde ambos agentes deben cooperar en conjunto para lograr un objetivo común, asumiendo cada uno su tarea para cumplir con el objetivo.

1.12 METODOLOGÍAS DE DESARROLLO DE AGENTES

Conforme la investigación en teoría de agentes fue tomando forma y el concepto de agente se fue introduciendo cada vez más en el mundo de la informática, surgió un esfuerzo por parte de los desarrolladores de software basado en tecnologías de agentes de crear una metodología que ayudase al desarrollo de estos sistemas. De cara al desarrollo de estas metodologías centradas en agentes, surgieron dos grupos de metodologías:

- Las que toman inspiración del desarrollo orientado a objetos y adaptan o extienden metodologías ya existentes de esta disciplina.
- Las que adaptan tecnologías relacionadas con la ingeniería del conocimiento.

1.12.1 AAII

La metodología AAII (Australian AI Institute) surge como una extensión de las metodologías orientadas a objetos y realiza una distinción entre dos modelos: el modelo externo, que corresponde a agentes y las relaciones entre ellos, y el modelo interno, que corresponde al agente en sí, usando sistemas BDI.

Así, el modelo interno del agente resultaría de implementar un modelo BDI de agente. Por el contrario, el modelo externo, estaría compuesto de otros submodelos: El modelo de agentes, que define por un lado las clases de agentes y las relaciones entre clases, como herencias, agregaciones e instanciaciones, y el modelo de interacciones que define como los agentes interactúan entre sí.

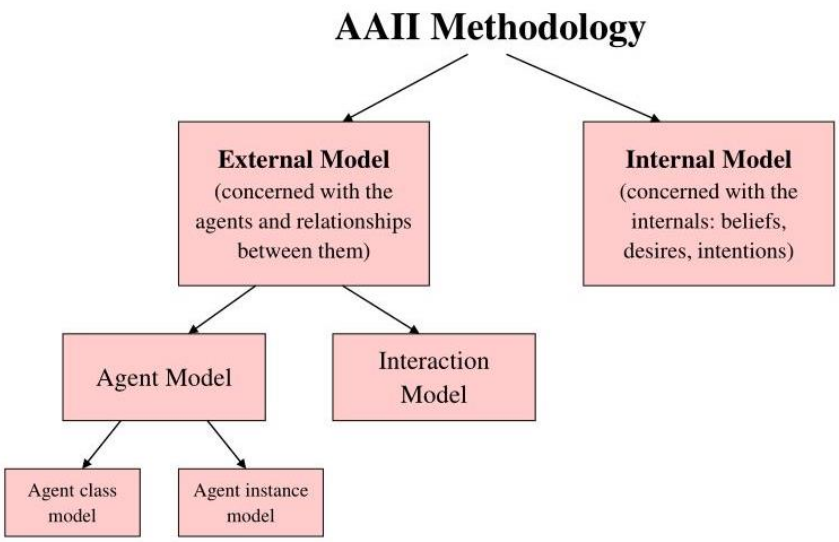


Figura 15 - Modelos de la metodología AAII

La metodología AAII para el desarrollo de estos modelos sería la siguiente:

- Identificar los roles en el dominio de la aplicación y en base a estos la jerarquía de clases de agentes.
- Identificar las responsabilidades de cada rol, los servicios que provee y que requiere cada rol y los objetivos asociados a cada servicio.
- Para cada objetivo se identifican los planes asociados y las condiciones para cada plan
- Determinar la estructura de creencias (beliefs), es decir, los requerimientos de información.

1.12.2 GAIA

GAIA es una metodología que ayuda a definir un sistema multiagente como un proceso de diseño organizativo, desde un punto de vista social de estos agentes. Permite al analista a desarrollar sistemáticamente, empezando por conceptos abstractos, que pueden no tener ningún impacto a la hora de implementar y que sirven para ir definiendo el sistema, y poco a poco ir definiendo conceptos más concretos que sí hay que implementar.

Abstract Concepts	Concrete Concepts
Roles	Agent Types
Permissions	Services
Responsibilities	Acquaintances
Protocols	
Activities	
Liveness properties	
Safety properties	

Figura 16 - Conceptos concretos y abstractos de GAIA

La metodología toma como base el entendimiento de un sistema como una *organización.* Esto es una colección de roles y sus relaciones entre ellos mediante el uso de interacciones sistemáticas empleando ciertos patrones (protocolos).

Un rol se define por 4 atributos: responsabilidades, permisos, actividades y protocolos. Las responsabilidades determinan la funcionalidad, y pueden ser de dos tipos: de vitalidad, que buscan los objetivos que intentará alcanzar el agente en ciertos estados y los de seguridad, que aseguran que siempre que cumplan una serie de características en todos los estados. Los permisos, que identifican los recursos que dispone un agente (normalmente la información disponible o que puede generar/modificar). Las actividades corresponden a cálculos o acciones que puede llevar a cabo un rol sin necesidad de interactuar, y, por último, los protocolos definen como un agente puede interactuar con el resto. Una entidad no tiene por qué tener un único rol, ni los roles ser fijos o estar instanciados a una única entidad.

Una vez están definidos estos conceptos abstractos, podemos diseñar los modelos concretos. Los tipos de agentes, que vienen dados por el conjunto de roles asociados y sus instancias, los servicios, que serían las funciones que corresponden a cada rol y por último los conocidos, es decir, con quien puede interactuar, basado de nuevo en los conjuntos de roles.

La metodología se compone de dos fases: análisis y diseño.

Análisis

1. Identificar los roles del sistema, correspondientes normalmente a:
    - Individuos, bien trabajando en una organización o trabajando individualmente.
    - Departamentos en una organización
    - Organizaciones por sí mismas
2. Para cada rol, identificar los protocolos asociados con el resto de los roles.
3. Ampliar el modelo de roles obtenido en el paso 1 con los protocolos, los permisos y las responsabilidades

Diseño

1. Crear el modelo de agente:
    a. Agregar roles dentro de los tipos de agentes y definir la herencia de tipos
    b. Documentar las posibles instancias de cada tipo
2. Desarrollar un modelo de servicios, examinando actividades, protocolos y propiedades, tanto de seguridad como de vitalidad.
3. Desarrollar un modelo de conocidos en base al modelo de interacción y a los modelos de agentes existentes.

1.12.3 TROPOS

La metodología Tropos (Bresciani et al., 2004) surge basándose en dos nuevas premisas en el desarrollo de sistemas multiagentes:

- Uso de la noción de agente y de nociones de conocimiento en todas las fases del desarrollo de software, desde los requerimientos iniciales a la propia implementación, basados en sistemas BDI.

- Empleo de una fase de requerimientos iniciales que tiene un papel fundamental a la hora del desarrollo del sistema. Esta fase no estaba representada en otras metodologías de desarrollo de agentes.

La metodología Tropos se basa en dos grandes bloques: por un lado, un lenguaje propio, Formal Tropos, que sirve como base para definir los conceptos y las bases del conocimiento empleados en la metodología; por otro lado, una serie de fases que establecen los pasos para implementar el sistema.

El lenguaje Tropos sirve como la base para el modelado de las distintas fases del proceso. Por un lado, define como deben ser los diagramas que realizar, así como la notación a emplear, y por otro, define una serie de conceptos necesarios para el modelado:

- Actor: sirve para modelar una entidad con unos objetivos e intencionalidad propia dentro del sistema. Esto representa por un lado a los agentes software, a los roles, que son una abstracción de comportamiento dentro de un contexto o dominio concreto, y a las posiciones, que representan conjuntos de roles. Un agente normalmente ocupará una posición, mientras que las posiciones sirven para ocupar los roles.
- Objetivos: representan los intereses de los actores, que podrán clasificarse en objetivos fuertes o débiles, estos segundos no teniendo normalmente una definición clara de cuando están cumplidas y sirviendo normalmente para modelar requerimientos no funcionales.
- Planes: representan de manera abstracta una forma de hacer algo, usualmente para el cumplimiento de objetivos.
- Recursos: entidades físicas o información.
- Dependencia: entre dos actores, se emplea para decir que un actor depende de otro para cumplir un objetivo.

- Capacidad: representan la habilidad del actor de en ciertas circunstancias de definir, elegir y ejecutar un plan para cumplir un objetivo.
- Creencias: conocimiento del mundo del actor.

Las fases sirven para definir los pasos concretos del desarrollo del sistema. Consiste en las siguientes 5 fases:

1. Análisis de requerimientos temprano: consiste en obtener el listado de las partes interesadas y en sus intenciones, que serán modeladas como objetivos. Estos interesados se definirán como actores sociales que dependen los unos de los otros para cumplir con los objetivos, realizar planes y recursos que ser distribuidos. Una vez obtenido este listado inicial, mediante un proceso de descomposición se va decidiendo los objetivos en subobjetivos más concretos y detallados, especificando de nuevo los actores de los que depende.
2. Análisis de requerimientos tardío: se centra en el sistema a implementar en su entorno operativo, así como en funciones relevantes y cualidades. Se representa el sistema como un actor, que tiene un numero de dependencias en otros actores de la organización y que servirán para obtener los requerimientos funcionales y no funcionales del sistema. En otras palabras, el sistema se define como un actor que contribuye a la completitud de los objetivos de las partes interesadas del análisis temprano.
3. Diseño de la arquitectura: se define la arquitectura del sistema en forma de subsistemas (actores) conectados mediante datos y flujos de control (dependencias). Se define con los 3 siguientes pasos:
    a. Definir la organización general del sistema, añadiendo nuevos actores para delegar subtareas o para lograr cierta estructura interna.

b. Identificación de las capacidades de los actores necesarias para cumplir los objetivos y planes. Se obtienen analizando el diagrama de actores obtenido del paso anterior y mirando las dependencias.
c. Definir los tipos de agentes y asignar a cada uno una o más capacidades.

4. Diseño detallado: introducir el detalle en cada componente del sistema. Esto dependerá de la plataforma de desarrollo, así como de otros detalles de implementación. Para este paso, Tropos recomienda seguir el estándar FIPA y emplear la extensión para el modelado de agentes de UML (AUML).
5. Implementación: la última fase corresponde a la implementación en el lenguaje concreto deseado del sistema.

1.12.4 MESSAGE

MESSAGE surge como una metodología de agentes que surge de extender UML incluyendo conceptos relacionados con agentes y de adaptar el modelo RUP (Rational Unified Process) de desarrollo software a el análisis y diseño de agentes.

El proceso general de MESSAGE es el de obtener en la fase de análisis 5 modelos que servirán para definir el sistema. En este análisis, se irán refinando los modelos en varios niveles las veces que se desee para tener el detalle apropiado o distintas vistas con información sobre estos modelos. Una vez se ha completado el análisis, estos modelos se emplearán para la fase de diseño, donde se hará una primera fase de alto-nivel y una de bajo nivel.

Los modelos representados en la metodología son las siguientes:

- Organización: Estructura general del Sistema Multiagente (entidades concretas, entorno y relaciones entre ellas)
- Objetivos/Tareas: objetivos del MAS y tareas necesarias para cumplirlos
- Agentes/Roles: se centra en los agentes individuales y los roles.

- Interacción: como los agentes interactúan entre sí y con los usuarios humanos
- Dominio (información): entidades específicas del dominio del problema y relaciones con el sistema en desarrollo.

CONCEPTOS

Agente: unidad mínima autónoma capaz de realizar una acción útil. Estas capacidades funcionales se establecen como los 'servicios' del agente, y su motivación interna que hace que sea autónomo se denomina 'propósito'. Rol: permite separar la parte de actuación que un agente efectúa de la identidad del agente en sí. De esta forma, nos permite separar de la misma manera que en la programación orientada a objetos una Interfaz separa comportamientos de una Clase. Así logramos un mayor nivel de abstracción, y permitimos que un rol sea posible que lo efectúe distintos agentes, así como a un agente realizar distintos roles.

Organización: grupo de agentes trabajando juntos con un mismo objetivo. Los agentes se conectan entre sí mediante relaciones organizativas, procedimientos de control y flujos de trabajo e interacciones.

Objetivo: sirve para asociar agentes con un estado. Si un objetivo se encuentra dentro de la memoria del agente, este buscará alcanzar el estado asociado a dicho objetivo. Estos objetivos se derivan del propósito del agente.

Tarea: unidad de conocimiento sobre una actividad con un único actuador principal. Establece una serie de pre y post condiciones (estados), así como un estado de realizando la tarea.

Interacción: unidad de conocimiento sobre una actividad con más de un actuador. Tiene un objetivo común que los distintos participantes deben alcanzar trabajando

juntos. Los Protocolos de interacción definen patrones de mensajes a intercambiarse asociados a una interacción.

Mensaje: objeto comunicado entre agentes para transmitir información.

FASE DE ANÁLISIS

El objetivo de la fase de análisis es el de producir una especificación del sistema que describa el problema a resolver, representado en forma de un modelo abstracto. El proceso de elaboración consiste en ir refinando en distintos niveles e ir descomponiendo la abstracción.

Primero se elabora el nivel 0, donde el sistema se define como una serie de organizaciones que interactúan con recursos, actores u otras organizaciones. En primer lugar, se elaboran el modelo de Organización y el de Objetivos/Tareas. Con estos modelos se elaboran entonces el de Agentes/Roles y el de Dominio, y por último con estos 4 el de Interacciones. Este nivel 0 se centra en identificar las entidades y sus interacciones entre ellas.

A partir de este nivel 0 se realizan subsecuentes niveles con el detalle. Un nivel 1 es requerido, donde se definen las estructuras y comportamientos de las organizaciones, tareas, agentes y objetivos con mayor nivel de detalle. Si se desea especificar y detallar en algún aspecto concreto, se crearán los niveles 2, 3, etc, deseados, donde se podrá tratar en detalle las funcionalidades, así como aspectos no funcionales del sistema.

FASE DE DISEÑO

La fase de diseño consiste en una serie de actividades que servirán para transformar modelos obtenidos del análisis en artefactos que serán implementados posteriormente. MESSAGE distingue dos fases de diseño: alto nivel y detalle.

En la fase de alto nivel se refina el modelo obtenido del análisis para producir una primera versión del sistema multiagente. MESSAGE propone los siguientes pasos:

- Asignar roles a agentes
- Relacionar servicios con tareas, tareas con objetivos y objetivos con roles
- Refinar los protocolos de interacción
- Especificar el comportamiento de los roles en un protocolo de interacción

En la fase de diseño detallado ya se tiene en cuenta la implementación, así como la plataforma donde se va a realizar dicha implementación. Consiste en mapear del diseño en alto nivel a elementos implementables de la plataforma escogida. MESSAGE ha sido diseñada teniendo en cuenta dos maneras complementarias de efectuar esa especificación:

- Diseño detallado dirigido por organización: el proceso es dirigido por el modelo de Organización para poder asignar responsabilidades, definir interacciones entre agentes y modelar conocimiento social.
- Diseño dirigido por plataforma de agentes: considera que cada agente puede mapearse a una clase, y está enfocado según la plataforma a emplear para la implementación, como JADE, donde se puede especificar tipos de agente derivados de otros.

1.12.5 CARENCIAS DE LAS METODOLOGÍAS MULTIAGENTES

Cabe destacar que estas metodologías de agentes tienen en común una serie de problemas que no se han logrado solventar del todo.

- Por un lado, pese a que en todas las arquitecturas el análisis y diseño de los sistemas a alto nivel está completamente descrito y detallado, cuando pasamos a la implementación a bajo nivel de los agentes o al uso de herramientas concretas nos encontramos con que todo se deja en manos del

responsable del sistema, y no ofrecen ningún tipo de pautas para su desarrollo o especificación.

- Al no proporcionar estas pautas, existe una falta de estandarización en el proceso de desarrollo a bajo nivel de estos agentes.
- Por último, no existe una metodología que se haya definido como el estándar, por lo que con los años van surgiendo nuevas metodologías que tratan de paliar las carencias de las anteriores

1.13 RESUMEN Y CONCLUSIONES

Cuando hablamos de una solución IoT, debemos tener en cuenta que toma en consideración dos aspectos fundamentales que deben trabajar en conjunto: por un lado, tenemos toda la parte electrónica, encargada de elaborar la circuitería y de distribuir los sensores y actuadores, y por otra parte el software que debe desarrollarse que haga uso de la circuitería y de los datos obtenidos por los sensores para implementar la lógica deseada.

Si se busca por Internet, se puede observar como la inmensa mayoría de información que existe al respecto hacen referencia, por un lado, al empleo de placas como Arduino para el desarrollo de prototipos, y por otro lado a el software que es necesario en estos dispositivos, donde suele emplearse C o C++. Es cuando se busca información sobre cómo interconectar diversos dispositivos donde se puede ver que no existe un consenso sobre una arquitectura.

Si tomamos por ejemplo un hogar inteligente o domotizado, podemos ver cómo o bien las empresas se ofrecen sus propias soluciones donde integran todos los distintos aspectos. Las empresas crean así sus propios ecosistemas donde es el cliente que adquiere los dispositivos el encargado de, si desea juntar distintos dispositivos de varias marcas, lograr que todo funcione en conjunto. La construcción de dispositivos corresponde a una parte electrónica más que estudiada, así como el software asociado, mientras que **a la hora de implementar una arquitectura común entre dispositivos no existe un consenso ni un estándar que las empresas usen**, dejando en manos del consumidor la responsabilidad de lograr que todo funcione en conjunto.

Por otro lado, como hemos podido ver, la teoría de sistemas multiagentes se encuentra en un estado bastante maduro, donde multitud de arquitecturas de agentes y metodologías de desarrollo han surgido para facilitar la construcción de estos sistemas. Sin embargo, **la implementación concreta y de bajo nivel de los**

propios agentes es algo que todo el mundo suele obviar, dejando a manos del equipo de desarrollo la decisión sobre esto.

Así, la intención de este trabajo de fin de grado queda definida por estas dos deficiencias: desarrollar una abstracción para IoT donde la implementación a bajo nivel quede implementada empleando el framework de ROS, mientras que la arquitectura y alto nivel corresponda al campo de las tecnologías de agentes, y lograr eliminar las deficiencias de cada tecnología haciendo que se complementen entre sí.

2. Proyecto

2.1 Objetivos

Generales

- Conseguir integrar los modelos de sistemas multiagentes con las tecnologías a bajo nivel de IoT.

Específicos

- Elaborar una capa de abstracción que permita el uso de arquitecturas multiagentes para su empleo y aplicación en el ámbito del IoT.
- Demostrar la viabilidad con la aplicación de un caso de uso concreto.

2.2 Plan de proyecto

El proyecto se ha dividido en dos grandes bloques. Por un lado, desde el inicio del mes de octubre a finales de febrero, se ha trabajado principalmente en un estudio del estado del arte actual sobre la teoría de agentes y las tecnologías del IoT, así como en empezar a definir de manera más específica el proyecto de TFG.

Por otra parte, durante la segunda parte, desde finales de febrero a finales de mayo, se ha trabajado en la puesta en marcha del proyecto y de su implementación

2.3 Especificación del sistema

El objetivo es desarrollar un sistema que ayude en la gestión en las tareas de planificación y mantenimiento de un hotel. Para ello, desarrollaremos un sistema multiagente que tendrá incluidos elementos de IoT con los que podremos obtener información en tiempo real y podremos responder con las medidas adecuadas, de manera que consigamos mantener el gasto energético lo más bajo posible, con el incluido ahorro económico.

Beneficios de emplear un sistema multiagente

Emplear una arquitectura de sistema multiagente nos permitirá definir agentes encargados de cumplir cada uno con las distintas tareas del hotel de manera coordinada y cada uno funcionando de manera independiente. Así, por ejemplo, si disponemos de un agente encargado de la ocupación este podrá decir al agente de mantenimiento que habitaciones están o no ocupadas y pueden necesitar ser limpiadas, permitiendo así que este tenga la información necesaria para poder planificar y responder adecuadamente. Con esto el sistema es capaz de coordinarse y de ir satisfaciendo las distintas necesidades que puedan surgir sin necesidad de tener la supervisión de un usuario.

2.4 Metodología

La metodología que vamos a seguir para definir el sistema es Message. Esta metodología se define por una notación que extiende UML con los siguientes elementos:

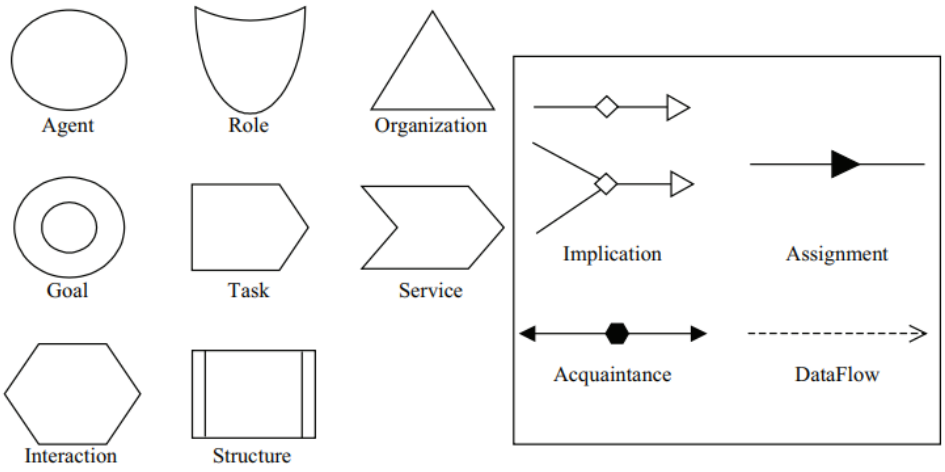


Figura 17 – Elementos gráficos de MESSAGE

La metodología consta de dos procesos, el proceso de análisis, donde obtendremos una descripción del sistema a través de varias vistas, empezando en un nivel 0 con

una mayor abstracción y refinando en sucesivos subniveles que estarán más detallados. Una vez terminado el análisis, pasaremos a la fase de diseño, donde obtendremos de estos modelos entidades concretas que implementar en un sistema de agentes, primero con el diseño a alto nivel y después el diseño a bajo nivel.

2.5 Nivel 0

MESSAGE define varios modelos para describir el sistema (2.12.4). En este nivel, para empezar, vamos a definir los modelos de Organización y Objetivos. El modelo de la organización hace referencia a la estructura del sistema y los objetivos que se va a querer lograr con dicho sistema. Al tratarse del nivel 0, esto se encuentra muy a alto nivel, y lo iremos concretando con los sucesivos niveles.

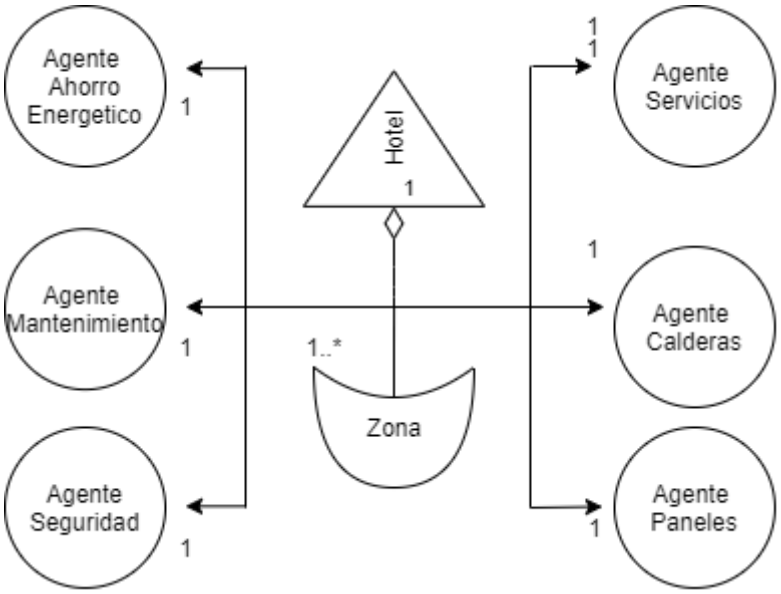


Figura 18 - Diagrama de organización, LvL 0

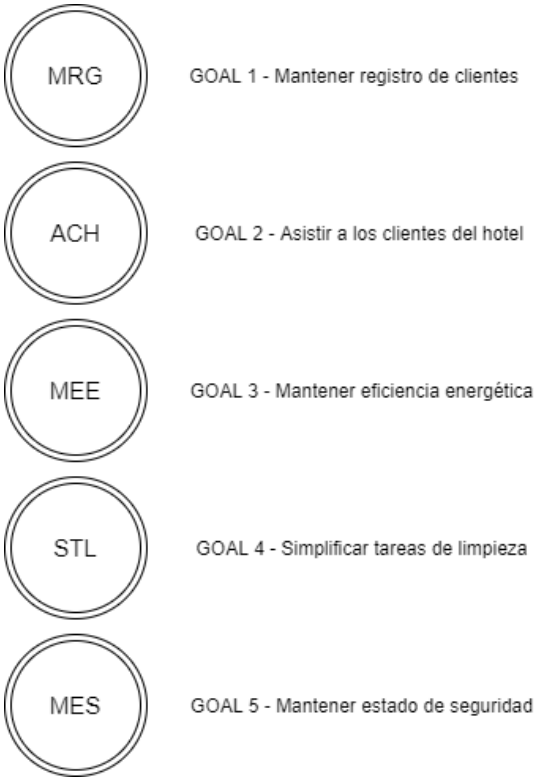


Figura 19 - Goals nivel 0, alto nivel

El hotel está compuesto por una serie de agentes a alto nivel, encargados de gestionar y planificar el funcionamiento del hotel, junto a una serie de zonas que se encargarán de tener a bajo nivel a agentes reactivos, que tendrán integrados los sensores y actuadores IoT que ayudarán a reaccionar a estos cambios del entorno. Las zonas tendrán un agente encargado de gestionar y comunicar (por ejemplo, una planta tendrá varios pasillos -comunes- y varias habitaciones gestionadas y controladas por un único agente de control). También existirán dos agentes encargados de regular dos áreas particulares: las calderas y una instalación de placas solares situadas en el tejado del edificio, que suministra energía al hotel.

Con estos, vamos a sacar los modelos de Agentes/Roles y Dominio. El modelo de dominio corresponde a la información específica del sistema en forma de entidades. El modelo de agentes y roles muestra los agentes individuales que van a existir en el sistema con sus respectivos roles, así como tareas que sabe realizar, reglas de comportamiento, etc. En este primer nivel, vamos a sacar los tipos

agentes que van a existir en el sistema, y cuando en sucesivos niveles obtengamos los objetivos y los respectivos roles asignados a cada objetivo, modelaremos los agentes junto a los roles que van a cumplir.

Abreviatura	Nombre de agente	Descripción
Srv	Servicios	Gestionar los distintos servicios, como la ocupación o el uso de la piscina
AE	Ahorro Energético	Encargado de gestionar el uso de energía en función del clima, la ocupación, etc.
Mant	Mantenimiento	Encargado de gestionar las tareas de limpieza y mantenimiento
Seg	Seguridad	Encargado de mantener la seguridad en el hotel
DF	Páginas Amarillas	Funciona como intermediario en las comunicaciones, permite a los agentes comunicarse entre sí y mostrar los servicios que ofertan
Ctrl. Z	Agente de Control de Zona	Encargado de gestionar una zona del hotel
Calderas	Agente de las calderas	Encargado de regular las calderas
Paneles	Agente de los paneles	Encargado de regular los paneles solares

Figura 20 - Tabla de agentes y sus abreviaturas

Dominio

El dominio del problema podemos definirlo empleando un diagrama de clases UML, que hará referencia a los tipos de información que vamos a encontrar en el sistema:

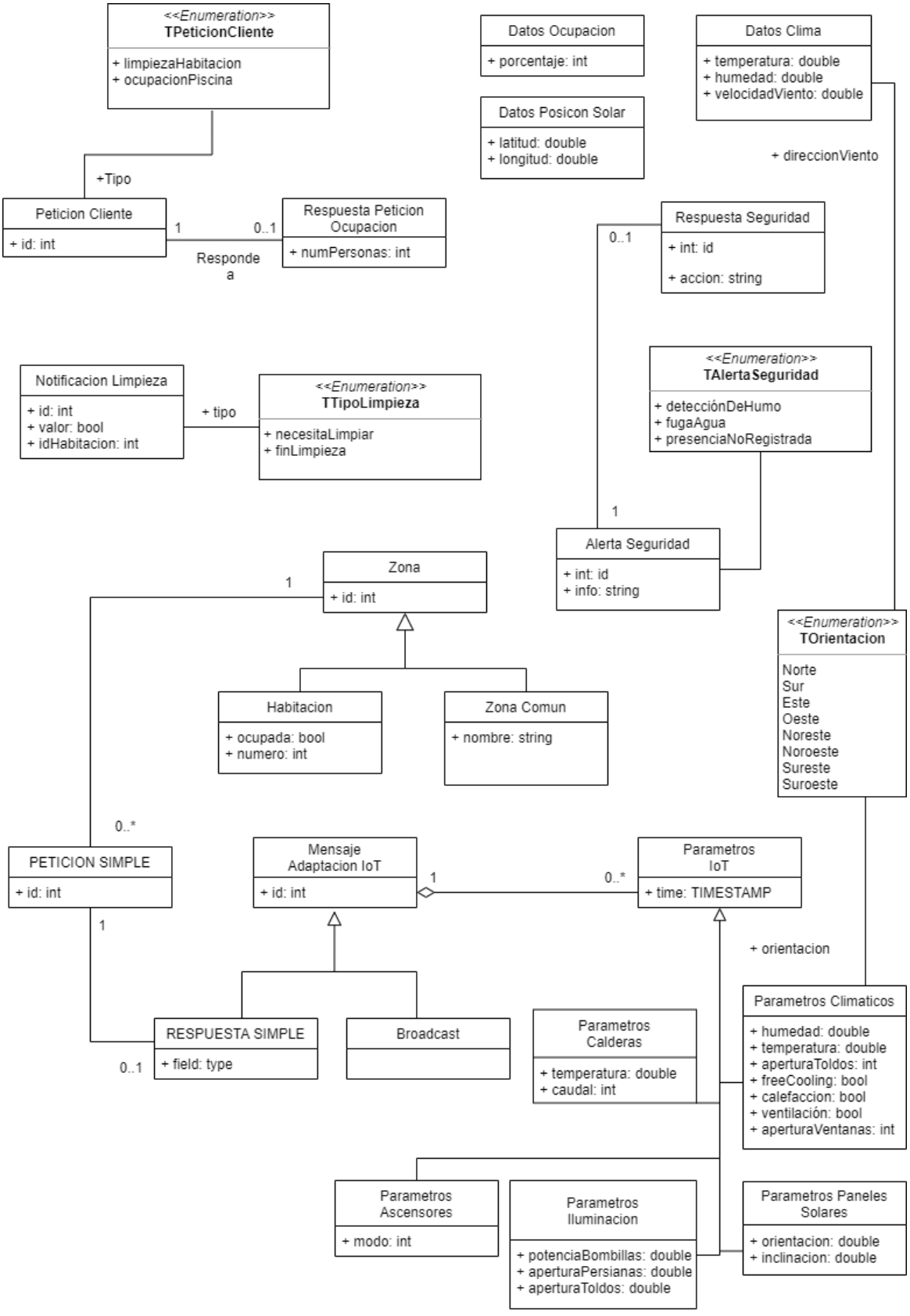


Figura 21 - Dominio, diagrama de clases

El agente de Páginas Amarillas permite a los agentes publicar los servicios que ofertan para que el resto de los agentes sean capaces de llamar a dichos servicios, de acuerdo con el estándar FIPA. El ofrecer este descubrimiento de agentes a otros se ve implementado como un servicio del agente de páginas amarillas. JADE, la plataforma para el desarrollo de agentes en Java, permite implementar este agente fácilmente.

Por último, las interacciones presentes en el sistema a nivel de agentes serán de dos tipos:

- Interacción agente-agente, que empleara como intermediario el agente de páginas amarillas para lograr comunicarse.
- Interacción agente-broadcast: realizada cuando un agente de planificación desea enviar los parámetros a adaptar a las distintas áreas del hotel. De nuevo, se empleará al agente de páginas amarillas para obtener el listado de los agentes de zona disponibles.

En la figura 23 se puede ver un ejemplo de ambas interacciones:

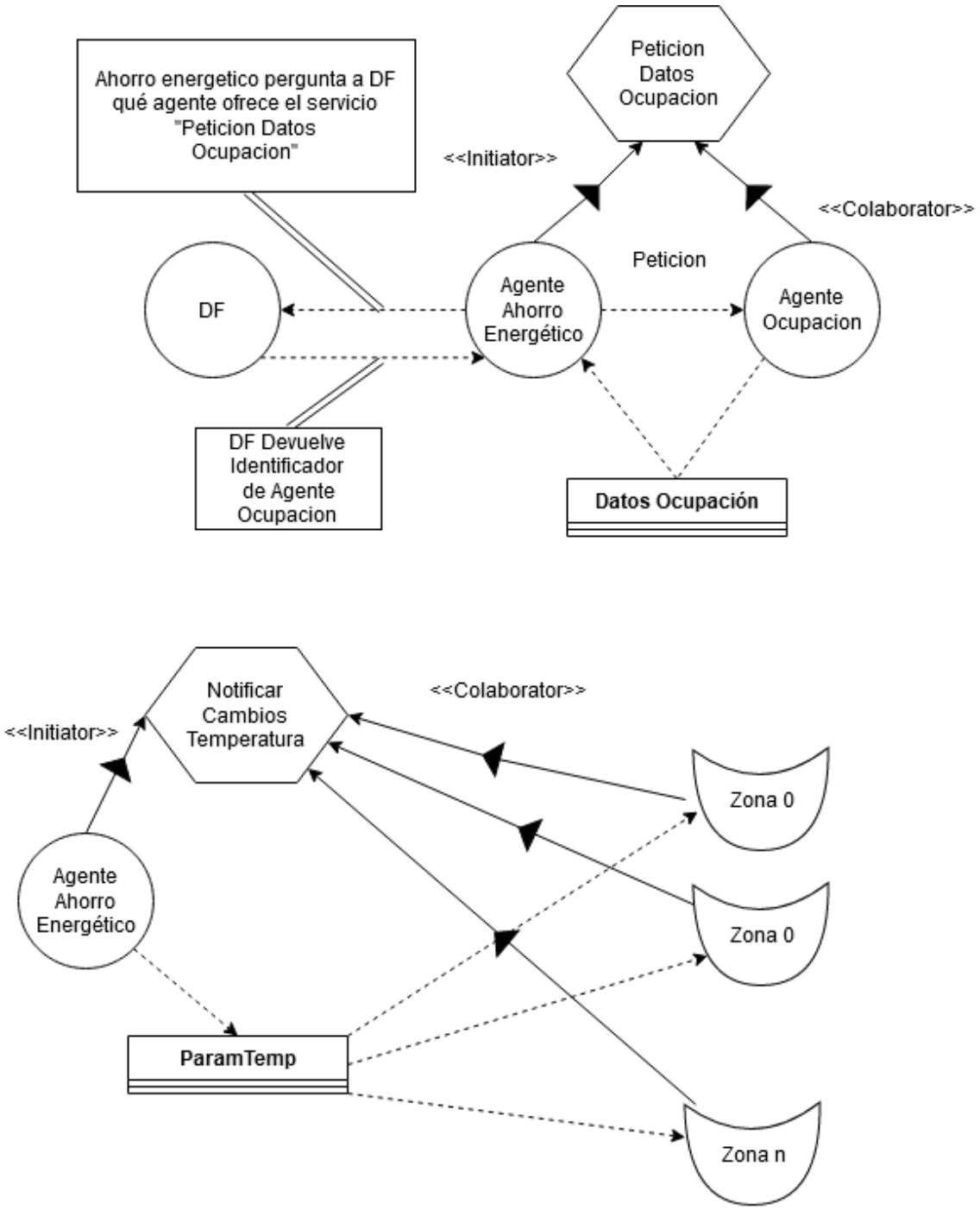


Figura 22 - Ejemplo de comunicación entre agentes de planificación (superior) y tipo broadcast (inferior)

2.6 Nivel 1

Para empezar, vamos a descomponer los objetivos genéricos que teníamos en el nivel 0 en subobjetivos, y asignar dicho subobjetivo a un rol concreto, que luego asignaremos a los distintos agentes que tenemos.

Mantener registro clientes

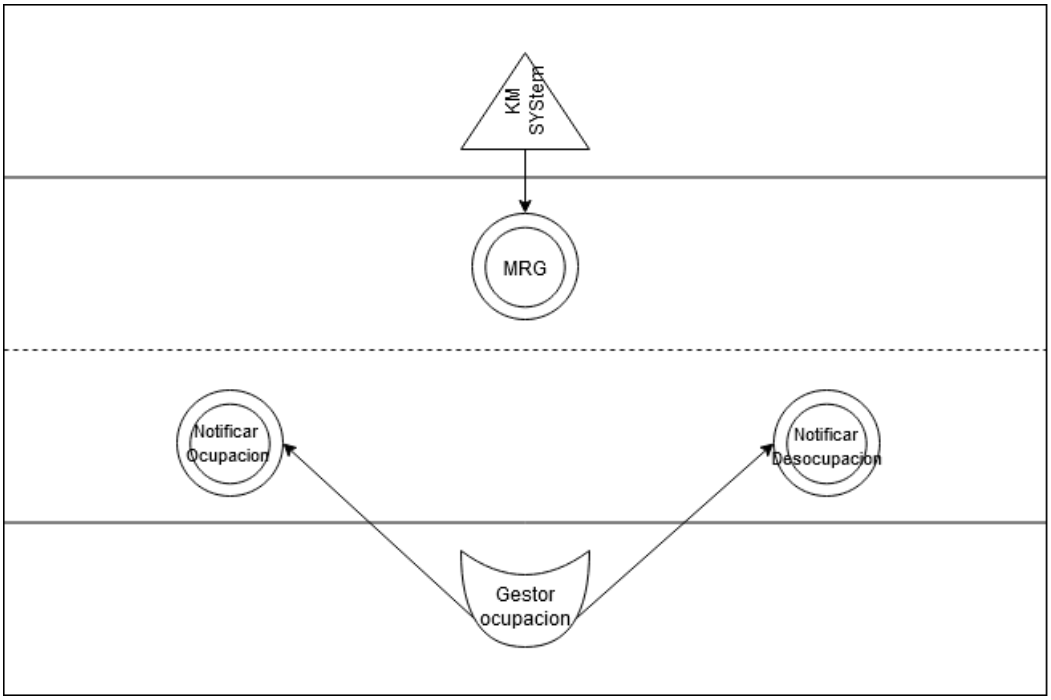


Figura 23 - Descomposición goal MRG

Asistir al cliente del hotel

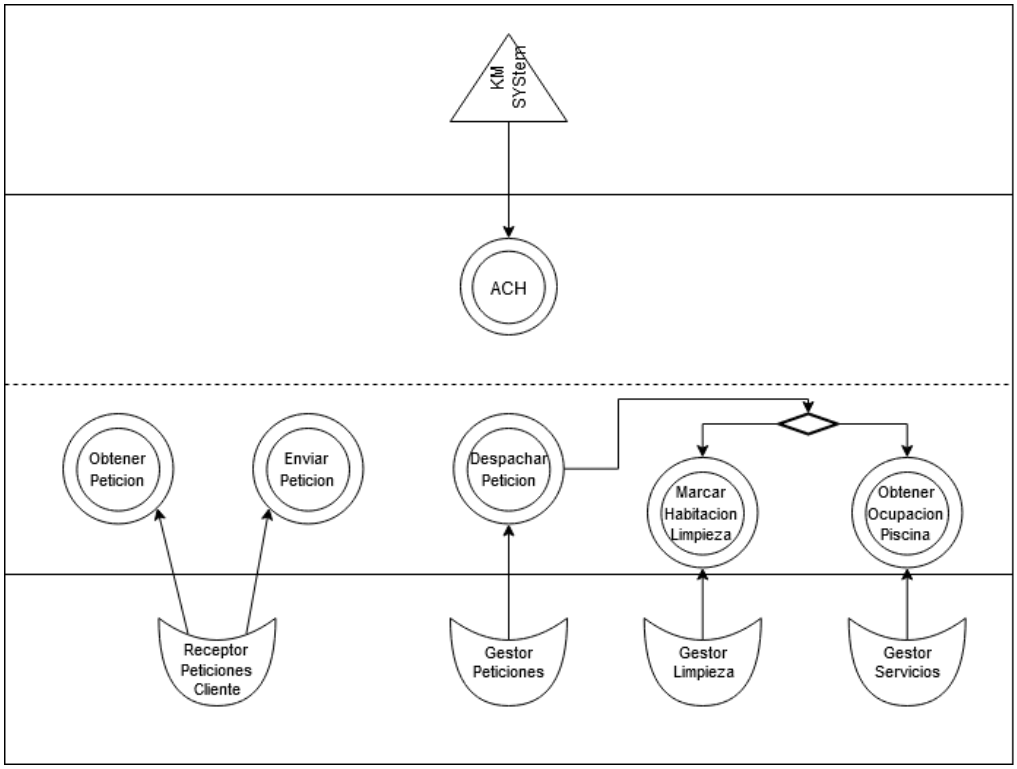


Figura 24 - Descomposición de goal ACH

Mantener eficiencia energética

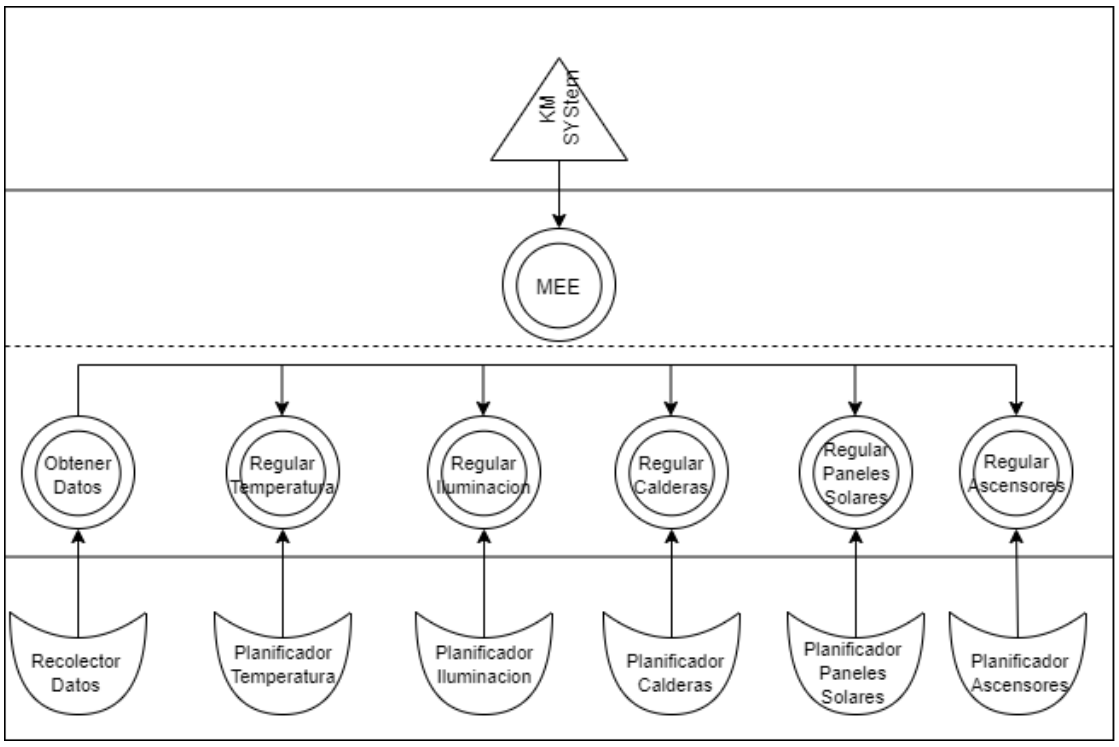
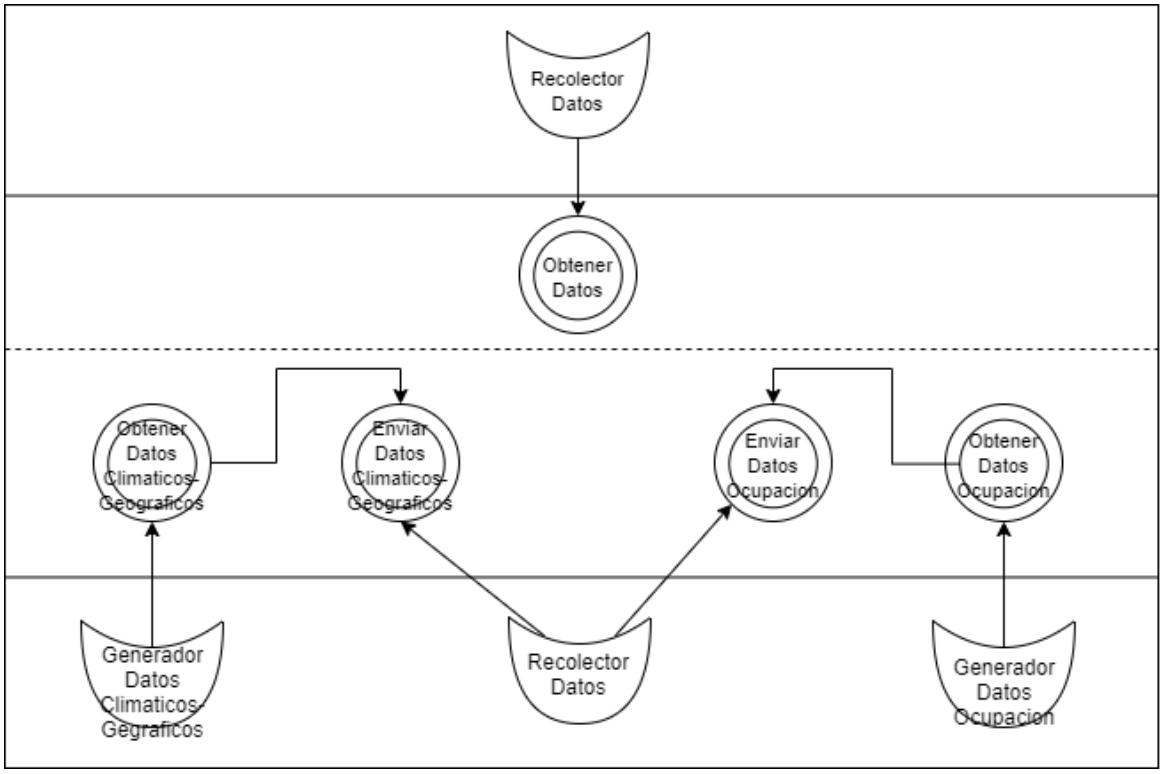


Figura 25 - Descomposición goal MEE

Cabe recalcar que nuestra implementación se centrará en implementar este sistema de ahorro y eficiencia energética. Los objetivos para la regulación requieren de la obtención de los datos previos climáticos, geoclimáticos y de ocupación, y con estos se planificará cada uno de los distintos aspectos energéticos relacionados con el hotel. Estos objetivos se explican en detalle en los diagramas de nivel 2. Puesto que van a ser los únicos de este nivel, y por conveniencia para luego definir los agentes, los vamos a detallar a continuación:

Goal Lvl 2: Obtener Datos



Goal Lvl 2: Regular Temperatura

En este objetivo hemos introducido los primeros ejemplos de nuestro sistema con agentes y elementos del IoT. Para ello, vamos a introducir la siguiente notación en los esquemas:

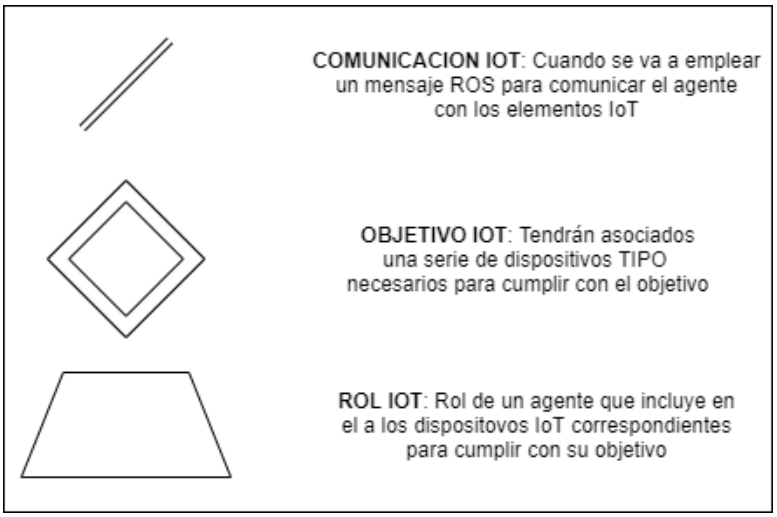


Figura 26 - Notación de elementos IoT

La comunicación a bajo nivel, entre los agentes y los elementos IoT, se va a realizar gracias al uso del framework ROS. Entraremos más en detalle en el diseño, pero a modo de adelanto, los agentes quedarán como nodos ROS que se comunican entre ellos, los distintos elementos IOT quedarán en otro nodo ROS, y los agentes mandarán un mensaje a estos nodos El objetivo quedaría descompuesto de la siguiente manera:

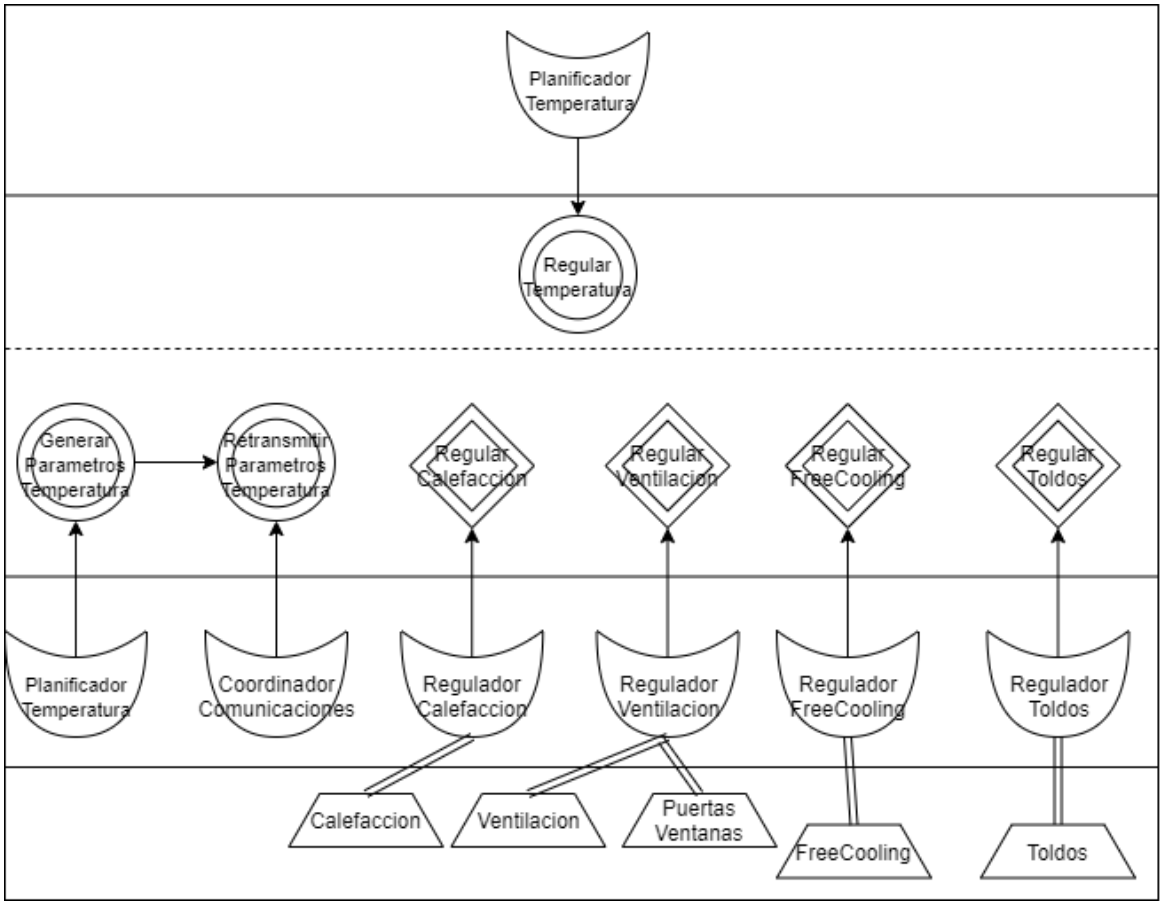


Figura 27 - Goal Regular Temperatura

En este caso el objetivo "Regular Temperatura" tendría asociados una serie de subobjetivos IoT, como por ejemplo "Regular Calefaccion", que implementa el rol "Regulador Calefaccion" y que se comunica a elementos de tipo "Calefaccion". Esta comunicación es la que tendrá asociada un mensaje de tipo ROS.

El resto de los objetivos de LvL 1 y 2 quedan definidos y detallados en el Anexo.

Interacción LvL 1

Mientras que en el nivel 0 hemos definido las interacciones entre los distintos agentes de planificación, en este nivel 1 lo que vamos a hacer es definir la comunicación entre agentes y elementos IoT y las comunicaciones entre los distintos agentes empleando ROS.

ROS nos permite emplear dos formas de envío de mensajes: por un lado, utilizar servicios a los que los nodos envían sus peticiones y reciben respuestas, funcionando como un servidor-cliente, y por otro lado el método de subscripción, donde se exponen los distintos 'topic' a los que los nodos se suscriben y empezarán a recibir los distintos mensajes que se manden por el 'topic'. El primer modelo, a modo de cliente-servidor, nos será útil para las comunicaciones entre los agentes de planificación con el agente de páginas amarillas, mientras que la segunda forma nos servirá para:

- Por un lado, comunicar al agente de control central con los distintos agentes de zona.
- Por otra parte, comunicar a los nodos correspondientes a los elementos IoT con la zona asociada en el sistema, pudiendo así recibir los mensajes para el cambio de parámetros.

Dominio LVL 1

Tal y como hemos hecho en el apartado anterior, en este apartado expondremos la información del problema que estará disponible a bajo nivel empleando ROS. Esto corresponderá por un lado a:

- Topics y los mensajes enviados por estos
    - Por cada agente de zona existirán varios topics correspondientes a los tipos de elementos IoT asociados. Así, por ejemplo, si existe una zona correspondiente a 'Zona de Terrazas' tendrá un topic para los toldos, que permitirá regular la extensión de estos de acuerdo con lo especificado por el agente de planificación de ahorro energético.
- Servicios y los mensajes que lo compondrán

Una gran parte de estos mensajes corresponderán a un equivalente en el diagrama de dominio de LvL 0 o lo encapsularan, ya que corresponderá a las comunicaciones

entre los agentes que se situarán en los distintos nodos de ROS. El resto de los mensajes estarán dirigidos entre los agentes y los dispositivos IoT de los que están a cargo, que tendrán en ellos los parámetros de ajuste necesarios.

2.7 Diseño Alto Nivel

Agentes y Roles

De acuerdo con lo que hemos especificado en la tabla 10, nuestro sistema se compondrá de los agentes de planificación de alto nivel y de agentes de zona encargados de los elementos IoT. Para estos últimos agentes, pese a tener un único rol de zona, vamos a implementar dos tipos de agentes que cumplan con este rol: un agente de zona pública y un agente de zona habitación. De esta manera podremos diferenciar los comportamientos de ambos. Para lograr un nivel más personalizado de control, se podrían implementar más agentes con este rol atendiendo a los distintos tipos de zonas públicas, pasillos, etc.

Servicios

Para mostrar estos agentes emplearemos los diagramas de vista de agentes que especifica MESSAGE. Tomemos por ejemplo el agente de servicios:

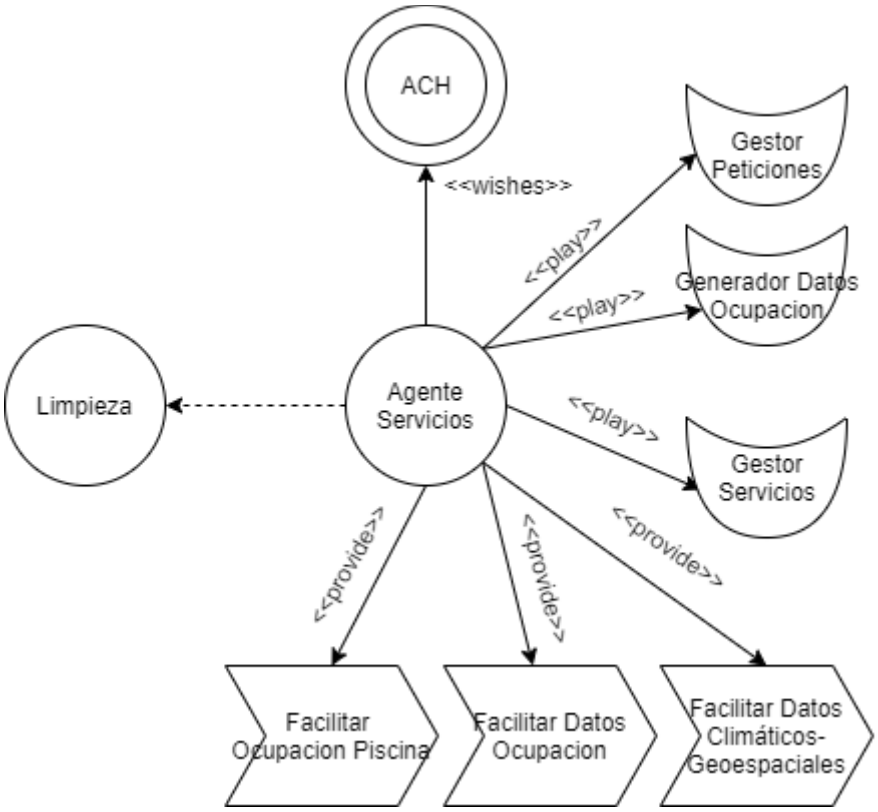


Figura 28 - Vista Agente de Servicios

Para facilitar el entender el agente a simple vista se ha intentado separar por un lado a la derecha los roles que implementa el agente, como "Gestor Peticiones", así como los servicios que provee al resto de agentes, como "Facilitar Datos Ocupación". A la izquierda se encuentra, marcado con una línea discontinua, los agentes o servicios con los que el agente necesita comunicarse. En este caso, el agente se comunicará con el agente de limpieza cuando las peticiones que le lleguen sean del tipo "Limpiar Habitación", enviando al agente de Limpieza la información necesaria para que este la marque como lista para limpiar.

En este ejemplo el agente no necesita llamar a ningún servicio para cumplir con sus tareas y/o roles. En el siguiente ejemplo veremos un ejemplo de esto.

Ahorro Energético

En el agente de ahorro energético (Figura 25) tenemos de nuevo a la derecha los roles y servicios que implementa el agente. En este caso, además, el agente se comunicará no solo con agentes individuales sino con todos los agentes que implementen el rol de "Zona", y necesita dos servicios para cumplir con sus deberes: aquellos que proveen de información geoespacial y climática, así como de información de Ocupación, que oferta el agente de servicios.

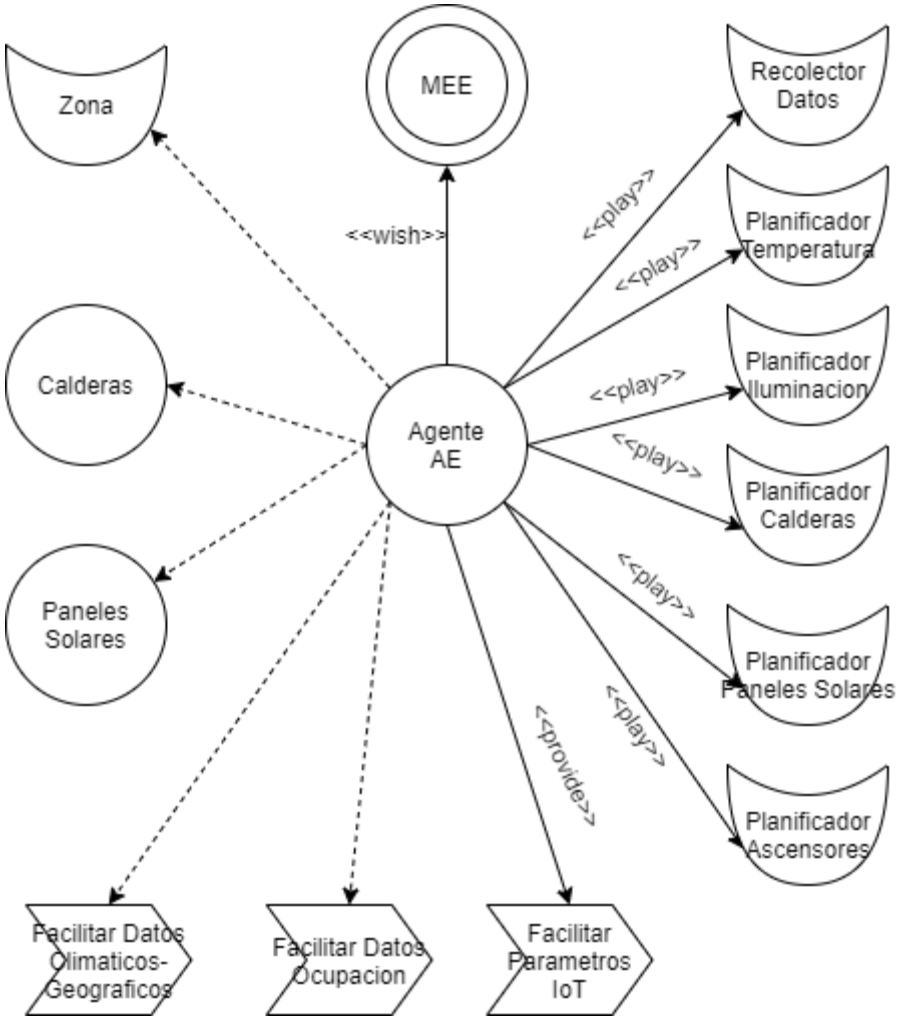


Figura 29 – Vista de Agente Ahorro Energetico

Agente de las calderas

En la figura 26 podemos ver el último tipo elemento que pueden tener los agentes. Puesto que queremos conectar el alto nivel de los sistemas multiagentes con el bajo nivel de IoT, vamos a especificar en cada agente los tipos de elementos IoT con los que se comunicará, designados por la etiqueta "<<operate>>" y con la anotación que habíamos introducido anteriormente. En este caso, el agente de Calderas no necesita empezar la comunicación con ningún otro agente. Será el agente de Ahorro Energético, como hemos visto en el diagrama anterior, el que iniciará la comunicación enviándole los nuevos parámetros a los que adaptarse, y este responderá a estos mensajes a fin de cumplir con sus dos objetivos.

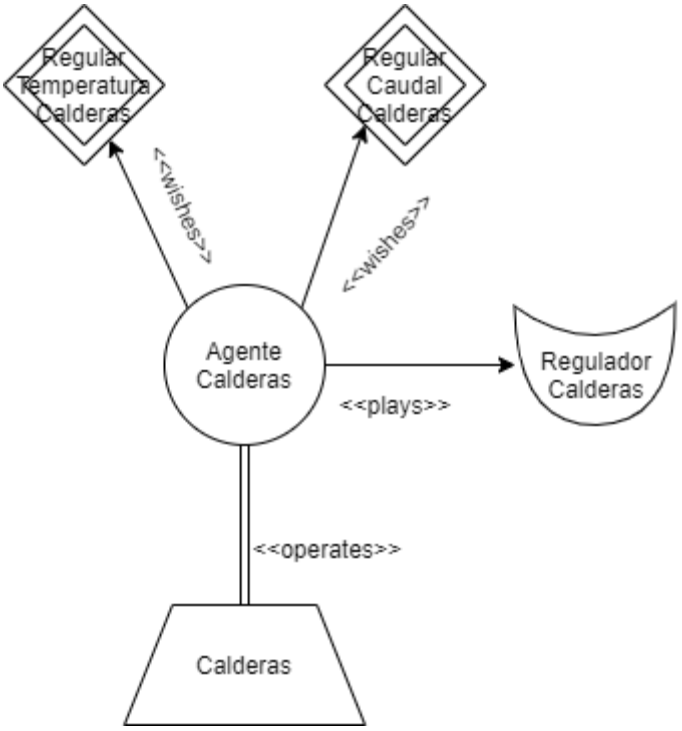


Figura 30 - Vista de Agente Calderas

Servicios y tareas

En este apartado nos centraremos en descomponer los servicios y tareas/objetivos complejos en tareas simples. Puesto que la mayoría de los servicios y tareas corresponden a tareas sencillas no es necesario descomponerlas. Para el servicio de Facilitar Parámetros de IoT, sin embargo, si necesitamos descomponerlo, para lo que emplearemos el siguiente diagrama de flujo de trabajo:

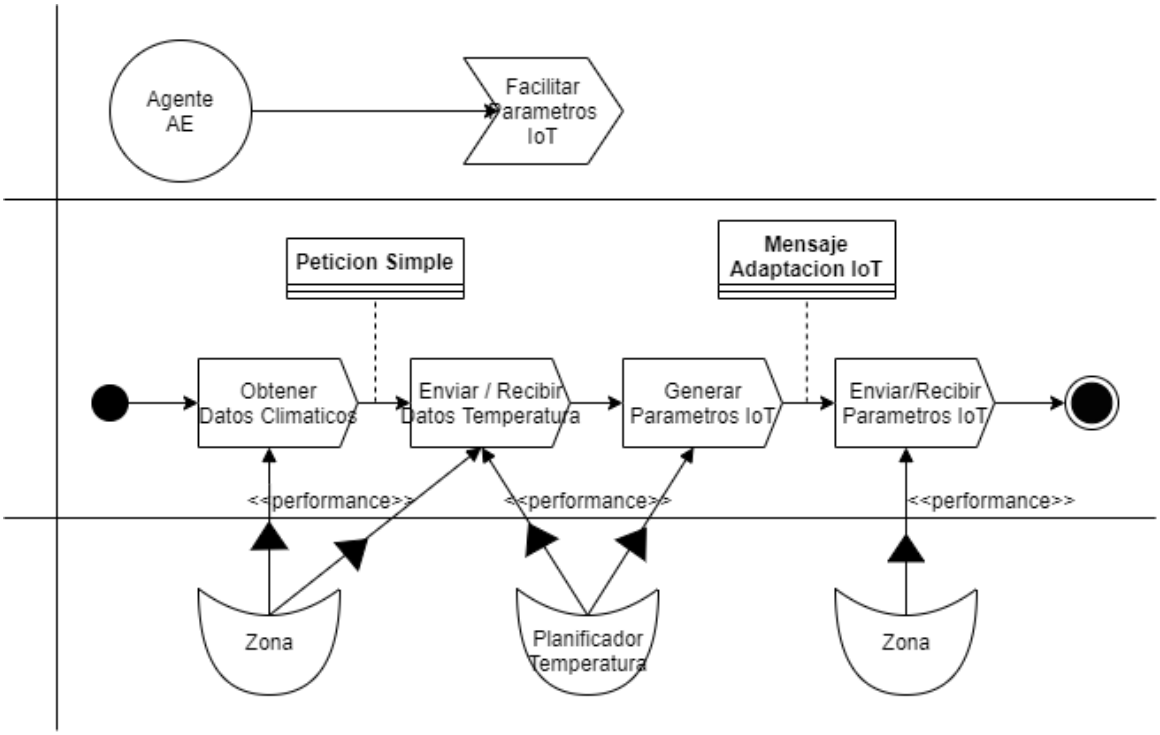


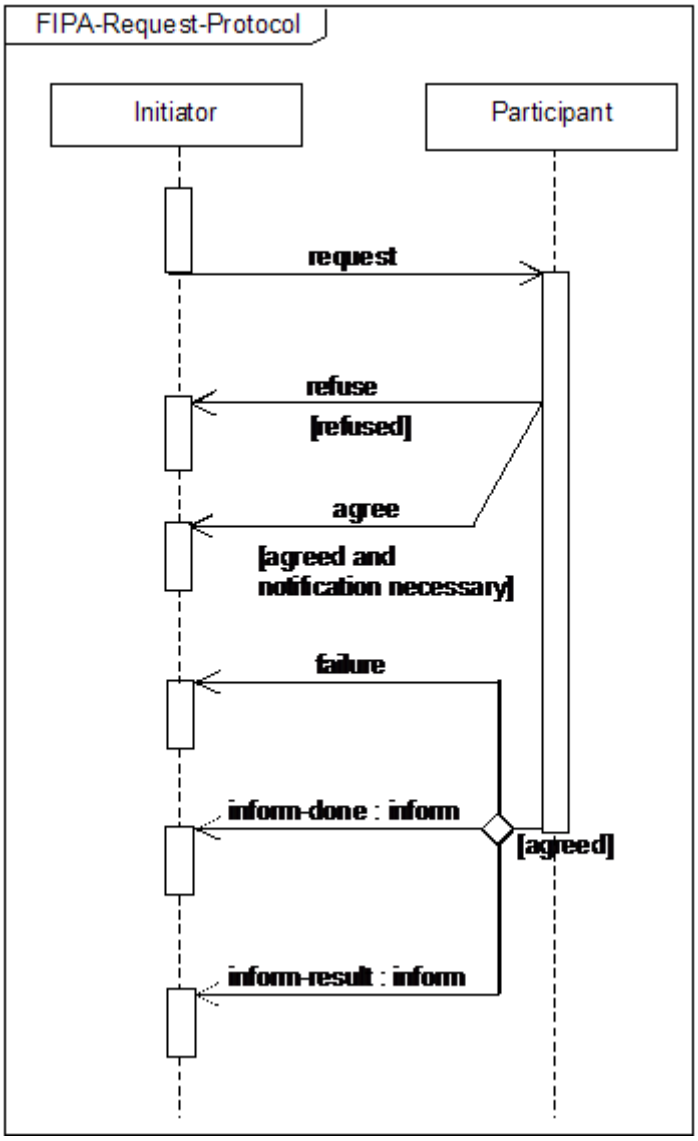
Figura 31 - Flujo de trabajo de tarea "Facilitar Parametros IoT"

El diagrama se asemeja al que vimos en el análisis para descomponer en subobjetivos más simples los objetivos complejos. En la parte superior tenemos el agente/rol con el servicio a descomponer; en la parte centrar vemos las subtareas a realizar con los roles implicados en la zona inferior, junto a los elementos de información de dominio que se van a crear y/o enviar asignados con la línea discontinua. En este ejemplo, el agente que implemente el rol de "Zona" será el que llame al servicio enviándole una petición ("Petición Simple") solicitando nuevos parámetros para adaptarse a los cambios en el clima, el planificador

recibirá esta petición, creara los parámetros y los enviará al que lo ha solicitado mediante un "Mensaje de adaptación IoT".

Protocolos Interacciones

Las interacciones se basarán en emplear el estándar FIPA para las comunicaciones. El protocolo básico de interacción es el protocolo de petición, que funciona de la siguiente manera:



Cabe destacar dos cosas: por un lado, la existencia de un timeout en caso de que ocurran problemas en la comunicación. Por otro lado, los actos de respuesta del tipo "inform" pueden ser de dos tipos: por un lado, simplemente confirmando que

se ha realizado la petición, en el caso por ejemplo en el que un agente solicita a otro que realice una determinada acción y, por otro lado, en el que la respuesta incluye información cuando por ejemplo se solicita los datos de ocupación.

2.8 Diseño Bajo Nivel

2.8.1 MENSAJES ROS:

Distinguiremos dos tipos de mensajes:

- Las comunicaciones entre los nodos de cada zona a sus respectivos elementos IoT, con las ordenes de actuación que han de realizar para lograr cumplir con los objetivos establecidos.
- Los mensajes sobre alertas de seguridad que se realizan desde los elementos IoT, cuando leen el ambiente y detectan una anomalía, a los nodos de control de cada zona, de tal manera que estos sean capaces de enviarlos al agente de zona para que indique las acciones a realizar, como la apertura de ventanas y encendido de sistemas de ventilación con el fin de eliminar el humo de un posible incendio.

2.8.2 Sistema Multiagente y Nodos Ros

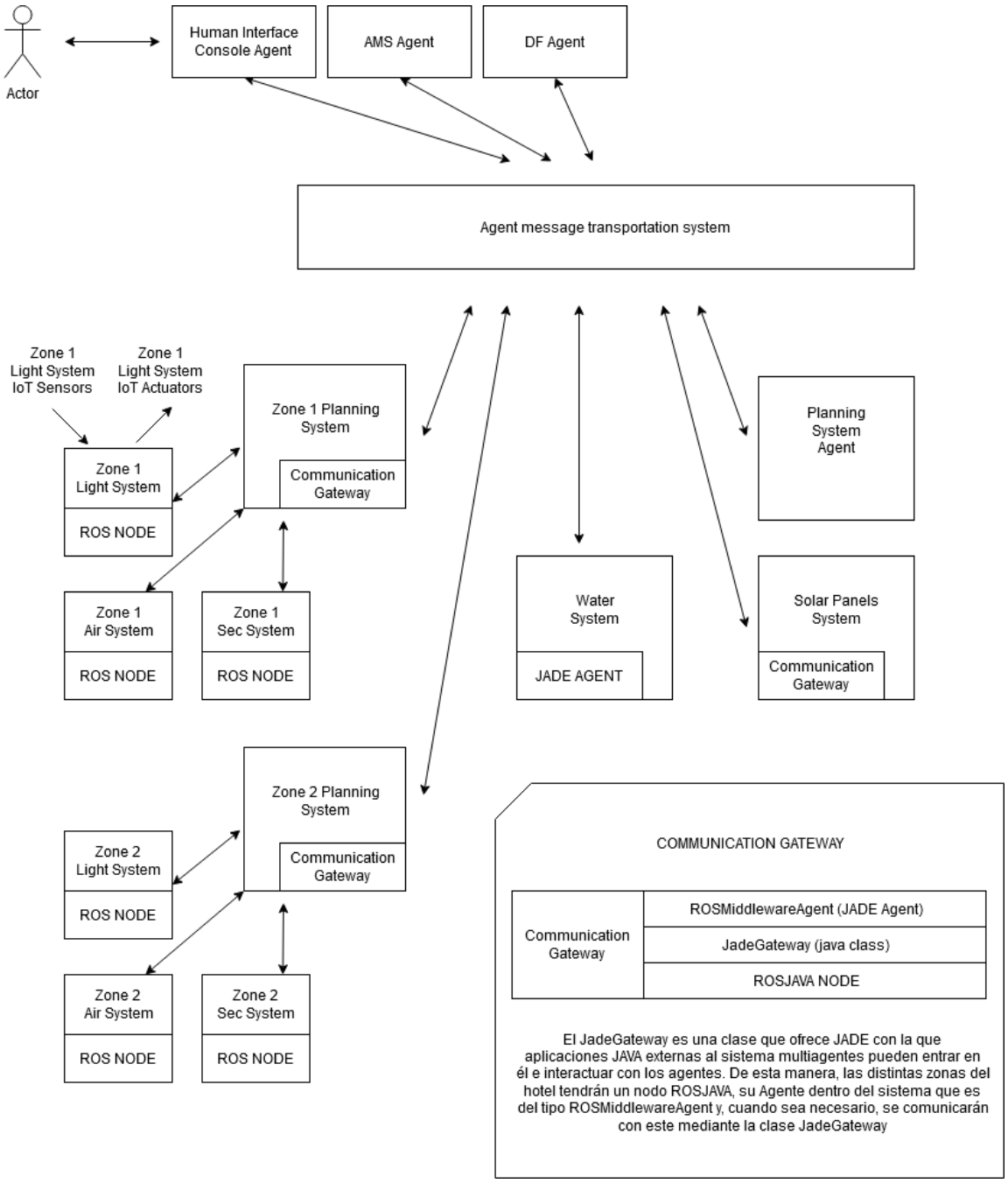


Figura 32 – Sistema multiagente ROS-JADE

3. Implementación y demostración del sistema

La implementación del sistema se ha realizado siguiendo un modelo de desarrollo ágil, enfocado en iterar sobre el software e ir añadiendo funcionalidad con cada una de las iteraciones.

El desarrollo se ha estructurado por tres focos principales: desarrollar los agentes en JADE, desarrollar los nodos de ROS necesarios para la gestión de los elementos IoT y la creación del Gateway de comunicación entre ROS y JADE: un nodo de ROS escrito en rosjava para poder lanzar el agente correspondiente para la gestión dentro del MAS.

3.1 Entorno de desarrollo

El entorno de desarrollo empleado ha consistido en el siguiente:

- Máquina virtual Ubuntu 16.04
- ROS versión Kinetic + rosjava
- Java 1.8
- JADE 4.5.0

El porqué de este entorno ha venido determinado por la necesidad de emplear el paquete de ROS "Rosjava" para tener compatibilidad con JADE que funciona con Java. "Rosjava" es admitido en ROS Kinetic y no en versiones más recientes, y este funciona bajo Ubuntu 16.04.

3.2 Entorno de pruebas

Para realizar las pruebas del sistema, así como su posible aplicación en un entorno real con dispositivos IoT y distintos elementos distribuidos, se ha decidido emplear el siguiente listado de dispositivos hardware:

- Ordenador Thinkpad E490 como servidor central de computación.
- Raspberry Pi 3B+ como dispositivo para control de zona.

- Placa Arduino para simular dispositivos IoT, tanto sensores como actuadores.
- Bombilla Xiaomi Lightbulb, como dispositivo IoT.

3.3 Arquitectura del software desarrollado

El software desarrollado en este proyecto puede diferenciarse en 2 bloques. Por un lado, el núcleo principal del proyecto consiste en el software del sistema multiagente con sus agentes de planificación, el software ROS para la comunicación a bajo nivel y el middleware desarrollado para la comunicación entre ellos. Con esto tendríamos lo básico para el sistema funcional, y para comunicar este núcleo con los dispositivos IoT, se han añadido paquetes de ROS que se encargan del control y mensajes a los dispositivos IoT. Estos paquetes son totalmente opcionales y dependerán del sistema desarrollado y de los dispositivos IoT que se vayan a emplear en el sistema, pudiendo implementar nuevos paquetes para el control de dispositivos IoT y añadirlos al sistema y comunicarlos con el agente de zona para poder tenerlos en cuenta en el sistema.

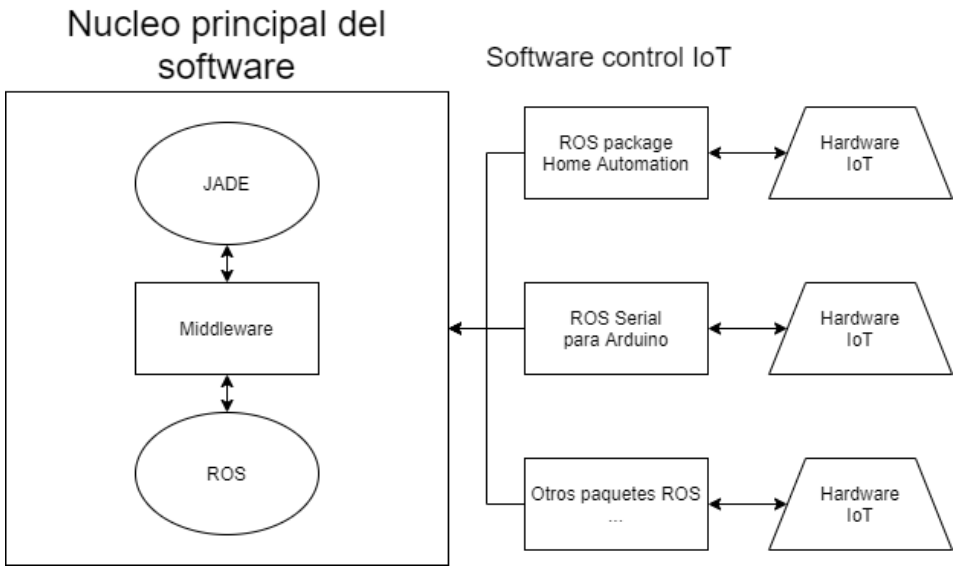


Figura 33 - Arquitectura software

3.4 Implementación de agentes con JADE

JADE proporciona un framework para el desarrollo de agentes software, y proporciona una implementación que cumple con los estándares de FIPA así como herramientas gráficas que nos permiten depurar e inspeccionar el sistema.

3.4.1 Ejemplo de agente básico

Para implementar un agente es necesario crear una clase JAVA que herede la clase `jade.core.Agent`. El único método necesario de implementar es el setup(), que corresponde a que tiene que hacer el agente cuando se inicializa. Por otro lado, el que acciones va a realizar un agente vendrá dada por los "Behaviours", que son otra clase de JADE `jade.core.behaviours`. Existen multitud de comportamientos, como cíclicos o de una sola ejecución, y será necesario crear clases para implementar estos comportamientos. Estos se añaden a la clase agente y se van ejecutando uno a uno.

Pongamos un ejemplo básico: el agente de Ahorro Energético, en su versión más básica, se inicializa y su función es la de recibir mensajes con peticiones para saber que datos son los objetivo. El método inicial con el que se lanza el agente es el método setup() , y dentro de este añadiríamos el comportamiento mediante el método 'addbehaviour()':

```
addBehaviour(new waitForMessages(this));
```

El comportamiento sería `waitForMessages()`, que viene explicado en la siguiente clase:

```
class waitForMessages extends CyclicBehaviour{
    public waitForMessages(Agent a){
        super(a);
    }
    public void action () {
        MessageTemplate mt1 = MessageTemplate.MatchContent("get_iot_params");
        MessageTemplate mt2 = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
        ACLMessage msg = receive(MessageTemplate.and(mt1,mt2));
        if(msg != null){
            try {
                String content = msg.getContent();
                if ( content != null && content.equals("get_iot_params")){
                    myAgent.addBehaviour(new sendIoTParamsBehaviour(myAgent,msg));
                }
            }catch(Exception e){
                e.printStackTrace();
            }
        }else{
            block();
        }
    }
```

De forma simplificada, este comportamiento espera a recibir los mensajes de otros agentes, y si son del tipo correcto, añade el comportamiento de responder con los parámetros necesarios. Así, los comportamientos serán más o menos complejos y el paso de uno a otro debe ser implementado por el programador, empleando el método "addBehaviour" para añadirlos al agente.

3.4.2 DF

El Directory Facilitator (DF) es un agente que nos da JADE que hace el papel de agente de Páginas Amarillas. Este es un agente con el que el resto de los agentes se comunican con dos fines principales: exponer servicios que ofrecen y obtener servicios registrados por otros agentes. Por ejemplo, el agente de servicios registra el servicio de "obtener_datos_climaticos" para que cuando otro agente busque este servicio el DF le devuelva el identificador del agente y sea capaz de llamar al servicio.

En la implementación, se ha decidido que se registren los agentes con su nombre como servicio, es decir, el agente de ahorro energético (AEAgent) se registra como "ae_agent", y es el agente el encargado de solicitar el servicio que desea mediante

el cuerpo del mensaje que le envía. Si este es incorrecto, el agente devolverá un mensaje de error.

```
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("ae_agent");
sd.setName("JADE-iot_params");
dfd.addServices(sd);
```

Por otro lado, el agente ZoneAgent (que explicamos en el siguiente apartado), dado a que necesita obtener los parámetros climáticos objetivo para regular los dispositivos IoT, necesita comunicarse con el agente de ahorro energético. Para saber cuál es el identificador de este, se comunica con el DF y busca el servicio correspondiente (“ae_agent”).

```
class searchAEAgentBehaviour extends OneShotBehaviour {
    public void action(){
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("ae_agent");
        template.addServices(sd);
        try{
            DFAgentDescription[] result = DFService.search(myAgent  ,template);
            ae_agents = new AID[result.length];
            for (int i=0;i<result.length;++i){
                System.out.println(result[i].getName());
                ae_agents[i] = result[i].getName();
            }
        }catch(FIPAException e){
            e.printStackTrace();
        }
    }
}
```

Esto permite que, si es necesario, el sistema pueda escalar y existan varios agentes que oferten un mismo servicio y se puedan añadir varios agentes para permitir redundancia y una mayor tolerancia a fallos.

3.5 Agente de Zona

El agente de zona (ZoneAgent) es el encargado de representar dentro del Sistema Multiagente a cada una de las zonas del hotel. Este agente, como explicaremos más

adelante en el proceso de creación e instanciación de clases del sistema al completo, es instanciado por un nodo ROS que lo introduce mediante código JAVA dentro del sistema multiagente. Así es capaz de comunicarse con el resto de los agentes y de enviarse mensajes empleando todas las herramientas que nos ofrece JADE.

Este agente tiene dos principales funciones: por un lado, como hemos mencionado, comunicarse con el resto de los agentes del sistema para poder solicitar y recibir los parámetros objetivo de temperatura, humedad y luminosidad. Para ello se comunicará únicamente con el agente de Ahorro Energético, buscándolo en el DF como se ha explicado anteriormente.

Por otra parte, este agente además de dedicarse a las labores de planificación se encargará también de reaccionar a posibles eventos que ocurran que requieran de una respuesta inmediata. Por ejemplo, si llega una alerta de seguridad, el agente determinará que es necesario actuar de inmediato, como, por ejemplo, abriendo ventanas en caso de haber un incendio y detectar humo mediante un sensor. Notificará también al agente de Seguridad para que este avise y actúe de manera pertinente a nivel global, pero después de haber mandado acciones rápidas para responder. Otro posible ejemplo de esta reactividad sería la de mandar una serie de acciones básicas en caso de que el sistema central de planificación haya tenido algún problema y no se encuentre disponible.

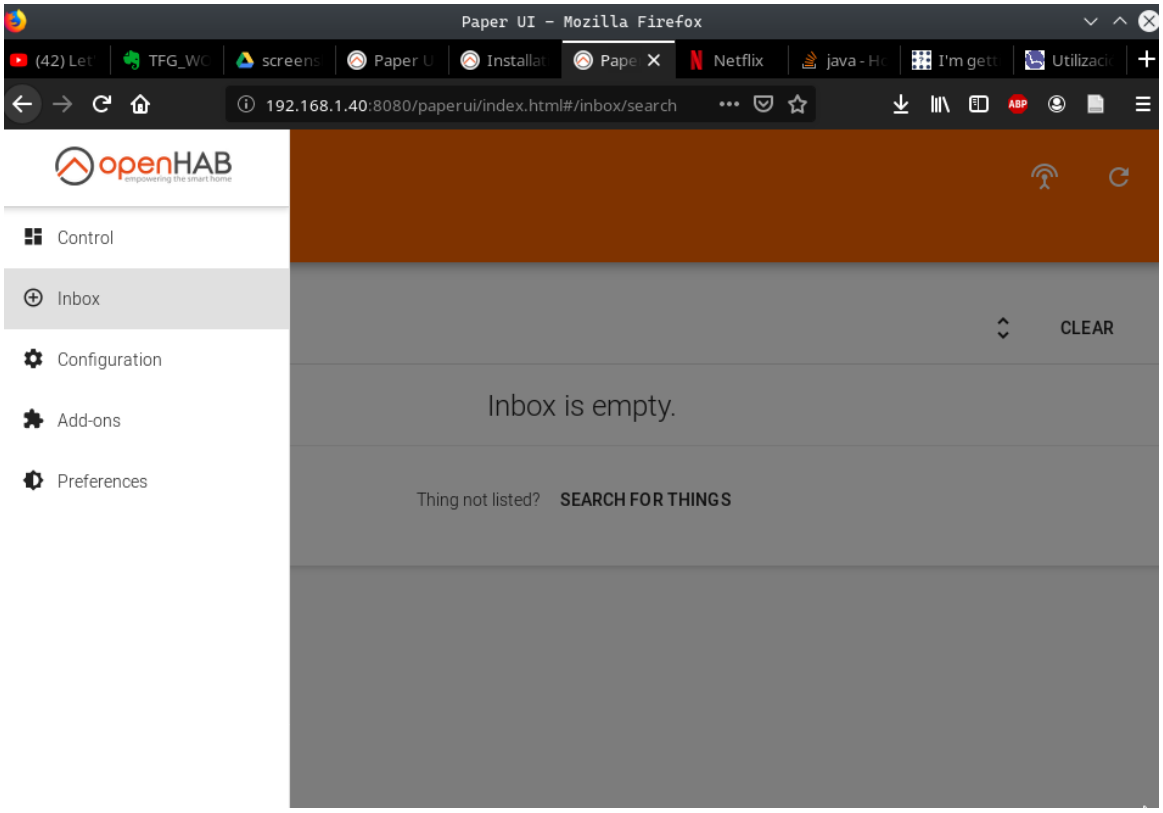
3.6 NODOS ROS IOT

3.6.1 OpenHAB

OpenHAB es un software para el control de elementos de domótica e IoT OpenSource que permite agregar dispositivos y establecer reglas de automatización y control del sistema. A nosotros sin embargo nos interesa únicamente emplearlo para conectar a este software los elementos IoT, en concreto, una bombilla inteligente de la marca Xiaomi. Del control se encargará el Nodo ROS

correspondiente, que enviará los comando al software a través del nodo "iot_bridge".

Los requisitos de este software pasan únicamente por tener instalado una versión de Java compatible, cosa que cumplimos al tener instalado Java 8. Tras instalar el software, este estará disponible a través del puerto 8080 por defecto de la máquina en la que se ha instalado. Tras acceder a la interfaz a través de un navegador podemos observar lo siguiente, haciendo click en el menú desplegable de la izquierda:



Para entender cómo funciona openHAB es necesario explicar brevemente los elementos que lo componen, al menos lo necesario para lo que nos interesa a nosotros:

- Binding: interfaz con dispositivos reales. Se instalan a través de los Add-ons, y se buscan desde la pestaña "Inbox".

- Channels: cada binding correspondiente a un dispositivo tiene una serie de valores que están asociados. Por ejemplo, un SWITCH ON-OFF o el valor leído de un sensor de humedad. Cada Channel corresponderá con una de estas características. Se pueden ver desde la pestaña "Configuration".
- Item: elemento empleado para leer y actuar en un Channel. Es decir, si tenemos este SWITCH en una bombilla, crearemos un Item que sea capaz de cambiar entre ON-OFF. Un Item puede tener asociado varios Channels, y así, por ejemplo, encender todas las bombillas de una habitación a la vez. Pueden crearse al seleccionar un Channel de manera automática o de manera manual desde la pestaña "Configuration".

Luego existen otros elementos para añadir complejidad y automatización, pero para nuestro uso no nos interesan. Para probar todo esto, se ha probado simplemente a detectar dispositivos dentro de la red. Primero buscaremos los dispositivos (figura 34):

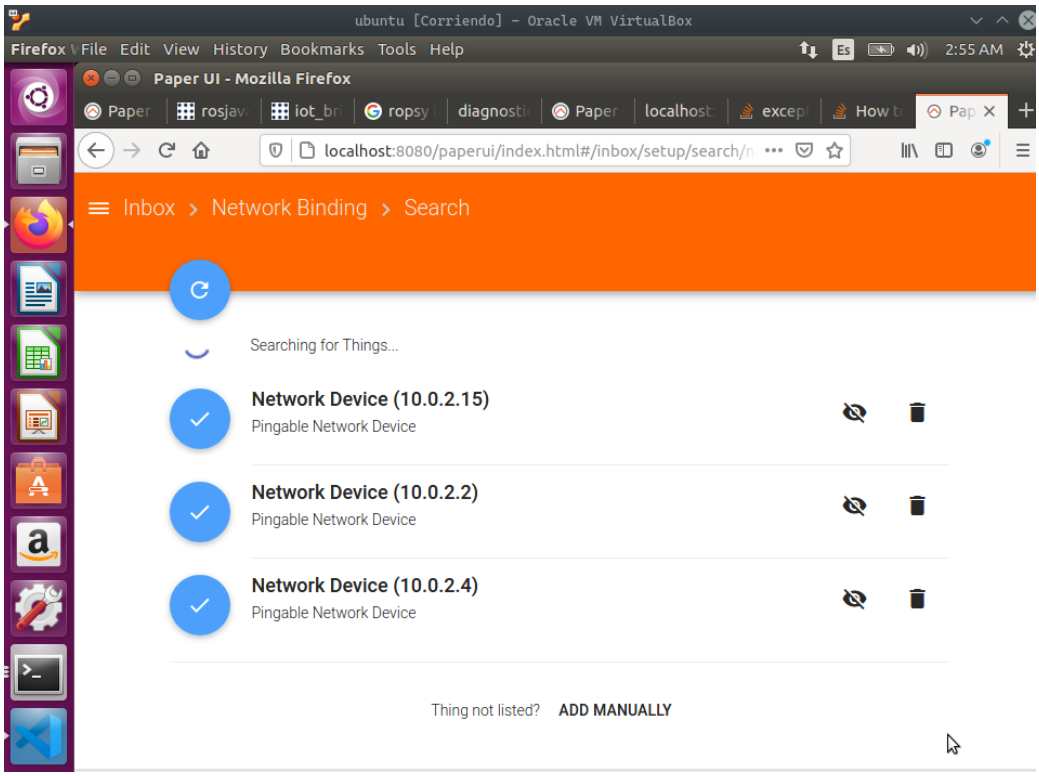


Figura 34 - Elementos descubiertos en OpenHAB al utilizar la herramienta de descubrimiento automático

Luego, añadiremos estos elementos y crearemos los ítems necesarios para monitorizar los campos de uno de estos elementos (figura 35):

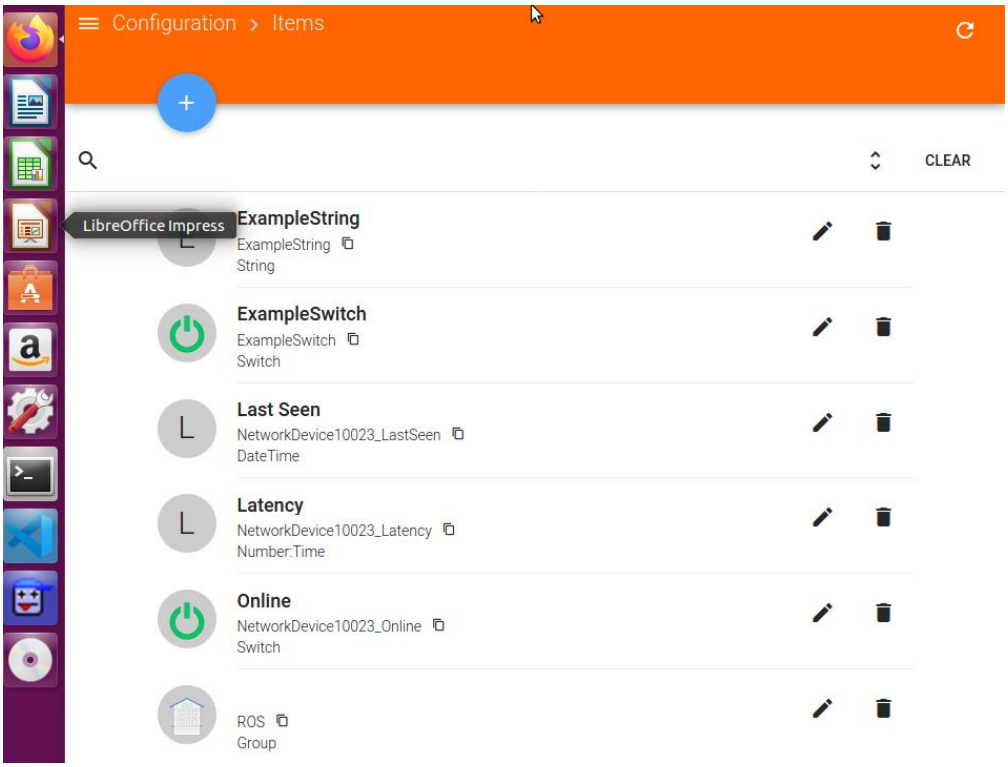


Figura 35 - Items configurados dentro de OpenHAB

Finalmente, vemos que efectivamente estos ítems se han asociado correctamente y están disponibles desde el menú inicial (figura 36):

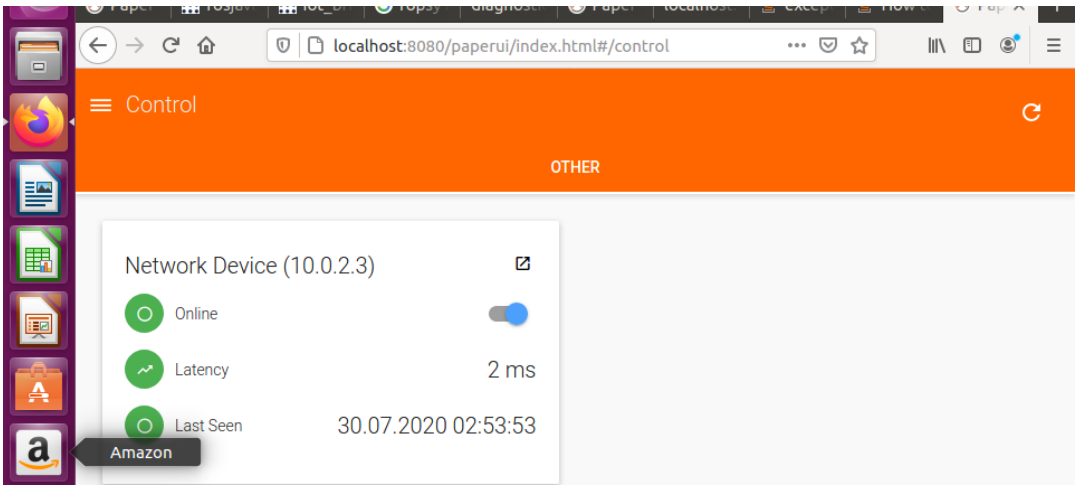


Figura 36 - Items monitorizados dentro del menú de inicio

Con esto podríamos apagar el elemento si le diésemos al botón de Online. Puesto que lo que queremos es hacerlo a través de ROS, emplearemos para esto el paquete "iot_bridge". Este paquete nos ofrece dos cosas:

- Una forma de monitorizar elementos empleando un grupo dentro de OpenHAB (un grupo es simplemente una asociación de elementos).
- Una forma de enviar órdenes a través de un topic de ROS.

El primer punto puede verse de dos maneras: bien empleando el topic que nos proporciona ROS o bien viendo la respuesta que da al hacer una petición a https://localhost:8080/rest/items/ROS (figura 37):

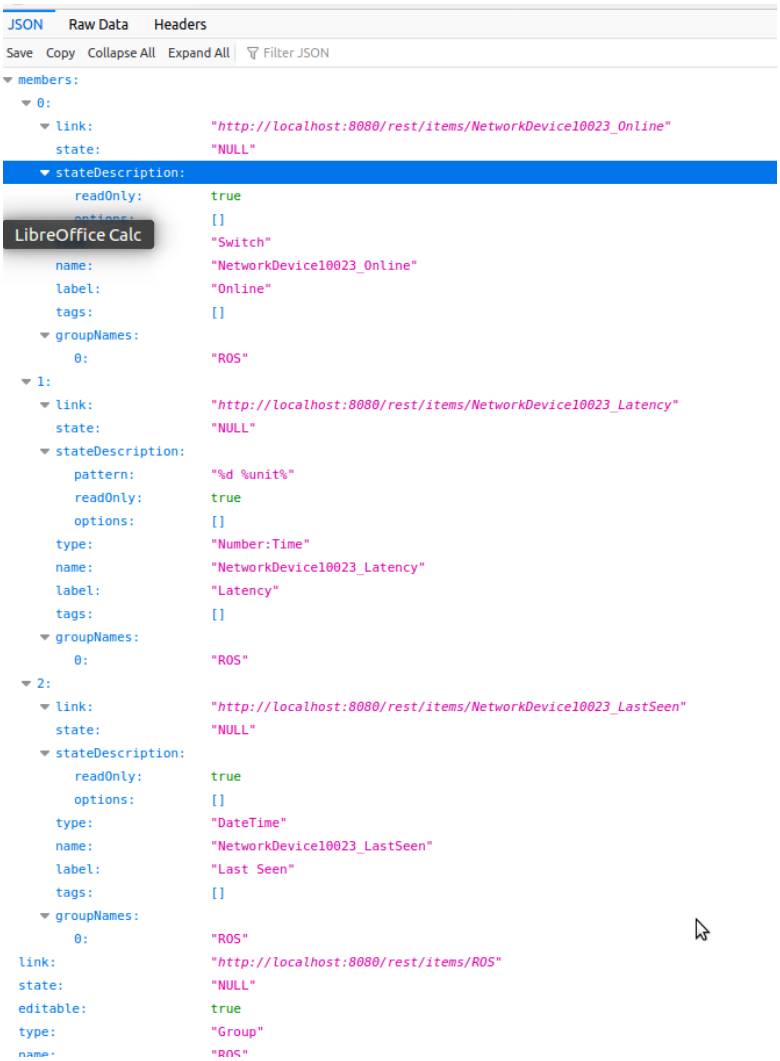


Figura 37 - JSON de respuesta con la información que le llega a ROS

Para enviar comandos se emplea un mensaje compuesto de una pareja de strings "Key": "Value", donde la Key es el Item de OpenHAB (figura 40):

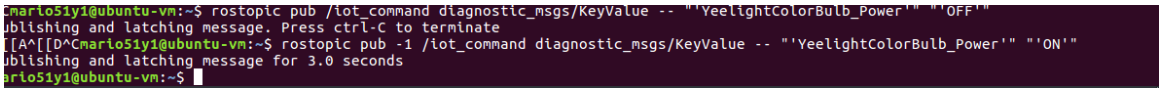


Figura 38 - Ejemplo de mensaje enviado mediante topic de ROS a OpenHAB

3.6.2 Rosserial Arduino

El uso de placas Arduino dentro de ROS se consigue empleando el paquete Arduino Bridge. Este paquete añade soporte para varios dispositivos como Arduino, ESP32, Leonardo, etc. Es necesario para ello realizar los siguientes pasos:

- Instalación del IDE Arduino
- Instalación del paquete rosserial_arduino
- Ejecutar el comando "rosrun rosserial_arduino make_libraries <directory>"

Con esto se consigue instalar las librerias en el IDE e importarlas mediante la instrucción "#include <ros.h>". Una vez se comprueba que el código compila y detecta la librearía, se ha probado su uso con un sencillo programa que enciende y apaga un LED cuando se lee un mensaje de un topic al que el nodo ROS-Arduino estará suscrito.

El código puede verse en la figura 39:

Figura 39 - Código simple en Arduino para comprobar funcionamiento con ROS

Cada vez que llega un mensaje al topic se ejecuta la función "messageCb", que recibe el mensaje y actúa en consecuencia. En este caso, nos da igual que mensaje, solo deseamos comprobar que el programa lo detecta. Los mensajes los lanzamos a mano con la herramienta "rostopic_pub".

Por una cuestión de claridad en la memoria, las fotografías que demuestran que esto funciona se pueden ver en el anexo. Con esto sabemos que tenemos lista la placa Arduino para implementar la lógica correspondietne en los entornos de pruebas.

3.7 Gateway ROS-JADE

La implementación del Gateway entre ROS y JADE ha consistido en la parte más importante de este desarrollo por ser el problema principal que se quería resolver con este TFG: el uso de tecnologías multiagente para tareas de planificación y

seguimiento de dispositivos IoT. Desde el inicio del TFG, cuando estaba aún en fase de puesta en marcha, se decidió emplear estas dos tecnologías por ser muy importantes y relevantes en sus respectivas áreas. Se buscó también, sin embargo, la forma de lograr comunicar estas dos plataformas, y conforme el desarrollo fue avanzando se plantearon distintas posibilidades para resolver este problema.

Desde el inicio se decidió emplear el paquete de Rosjava para poder facilitar la comunicación, puesto que mientras que ROS si permite varios lenguajes desde su versión sin ningún paquete añadido como C++ o Python, JADE está disponible únicamente en Java. El siguiente paso consistió en buscar como comunicar ambas aplicaciones y ver las distintas maneras de lanzar tanto agentes como nodos de ROS, bien mediante código o mediante instrucciones por terminal. JADE permite lanzar agentes desde programas de JAVA externos, y de primeras se decidió que el encargado de lanzar los agentes de cada zona correspondería al nodo ROS asociado a cada zona.

Una vez se decidió esto, se buscaron diversas maneras de comunicar las dos aplicaciones JAVA. Para esto existen varias tecnologías y métodos, como, por ejemplo, sockets, RMI o simples archivos en el sistema operativo. Sin embargo, tras revisar de nuevo la documentación de JADE, se vio que la propia librería permite la comunicación de aplicaciones JAVA externas a un sistema multiagente mediante la clase JADEGateway. Tras estudiarlo, se vio que esta era la solución más simple y efectiva para resolver el problema.

Con esto, la forma de comunicarse ROS con JADE queda de la siguiente forma:

- Se crea un nodo de ROS para una zona.
- Dentro de la inicialización del nodo, este crea su representación dentro del sistema multiagente mediante un agente de zona.

- Cuando el nodo de ROS lo necesita, crea un agente JADEGateway y crea un comportamiento para comunicarse con el agente de zona.
- El agente realiza las acciones necesarias dentro del MAS.
- La respuesta llega de vuelta al JADEGateway que lo almacena.
- El nodo de ROS accede a la respuesta dentro del JADEGateway y realiza las acciones necesarias de acuerdo con el contenido.

Para entender mejor cómo quedarían todas las partes en la comunicación, así como ver qué partes se comunican con qué, se puede observar con el siguiente gráfico:

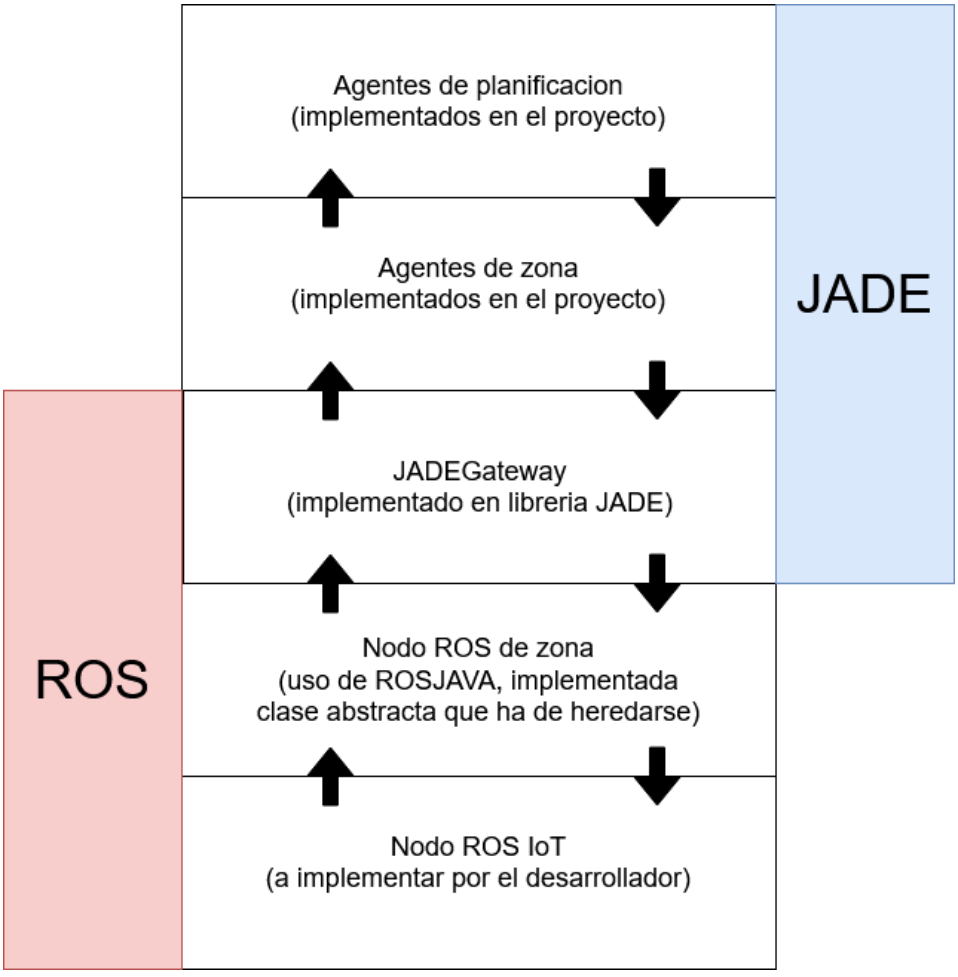


Figura 40 - Comunicación entre las partes del sistema

Los agentes se encargarían de decidir las acciones a realizar, donde por una parte los agentes de planificación determinarán los objetivos a un nivel general y los agentes de zona se encargarían de transmitir estos objetivos a los nodos ROS para

que gestionen las acciones necesarias, además de implementar la reactividad mencionada anteriormente.

3.7.1 Rosjava

Rosjava es un paquete de ROS que permite crear nodos empleando el lenguaje JAVA. Con esto podemos aprovechar y delegar la creación de cada agente al nodo ROS encargado de gestionar una zona del hotel, y este empleará su nodo para tener representación en el sistema multiagente.

La clase `AbstractRosjavaNode` es la encargada de todo esto. Se trata de una clase que implementa todo el proceso de instanciar el agente y de comunicarse con el cuándo es necesario mediante el agente JADEGateway, que es llamado solo cuando es necesario por cada agente. Esta clase sirve como una clase genérica que implementa únicamente la comunicación entre nodo y agente, así como toda la configuración y gestión necesaria para lograr esto. Con esto lo que se pretende es que se use como clase base para crear otros nodos distintos, donde cada clase representará una zona del hotel y se podrá configurar con los dispositivos IoT que tenga y la lógica para comunicarse con ellos empleando nodos ROS para cada grupo de dispositivos IoT, permitiendo que sea válida para los distintos paquetes que podrían añadirse.

En esta clase abstracta tenemos implementado todo el proceso de instanciación y gestión de la comunicación entre ROS y JADE. En primer lugar, la clase se encarga de instanciar un nuevo agente dentro del Sistema Multiagente empleando las clases que ofrece JADE para crear agentes de JADE de manera programable desde clases JAVA y se inicializa también el nodo ROS. Por otro lado, se implementa un comportamiento que es el que se encargará de ejecutar el JADEGateway para comunicarse con el agente de zona correspondiente y de devolver el mensaje de respuesta al finalizar. También se ofrece el esqueleto de cómo funcionará el bucle

de ejecución una vez inicializado el agente, con funciones abstractas que son que hay que implementar en cada Nodo de Zona concreto.

De manera esquematizada, el agente se comportaría de la siguiente manera:

1. Inicialización de la clase
2. Obtener parámetros de configuración mediante ROS
3. Creación del Agente de Zona correspondiente dentro de JADE
4. Inicialización de los canales en ROS (publishers)
5. Bucle for:
    a. Esperar hasta que se recibe una petición para actualizar dispositivos IoT
    b. Una vez recibida, se crea el agente JADEGateway para comunicarse con el Sistema Multiagente
    c. Al JADEGatweay se le crea un comportamiento para comunicarse con el agente de zona y esperar la respuesta
    d. Se recibe la respuesta
    e. Se procesa la respuesta y se crean las acciones a devolver
    f. Se mandan las respuestas a los nodos ros mediante los publishers.

De estos apartados, los puntos 4, 5.e y 5.f corresponden a los métodos abstractos que son responsables los programadores de implementar para las distintas zonas con sus distintos nodos.

De esta manera, crear Zonas distintas que comuniquen con dispositivos IoT distintos correspondería con implementar una clase que herede de AbstractRosjavaNode y que implemente los comportamientos abstractos.

Como ejemplo de esto puede verse la siguiente clase en la figura 41 (Nota: la clase final no corresponde a esta implementación, es únicamente con fin de simplificar para la explicación):

```
public class RosjavaNodeArd extends AbstractRosjavaNode {
    Publisher<Climate_act_data> climateDataPublisher;
    Climate_act_data data;

    void preparePublishers(final ConnectedNode connectedNode){
        climateDataPublisher =
            connectedNode.newPublisher("climate_data", Climate_act_data._TYPE);
    }

    void createIoTActionsResponse(){
        data = climateDataPublisher.newMessage();

        float a = 0.0f;

        data.setTemp(a+sequenceNumber);
        data.setAirConditioning(true);

    }
    void sendIoTActionsResponse(){
        climateDataPublisher.publish(data);
    }
}
```

Figura 41 - Nodo simple implementando AbstractRosjavaNode

Para las pruebas, como se verá a continuación, se han tomado dos escenarios, con un total de 3 nodos de zona distintos: uno con un nodo conectado a una placa Arduino (para simular dispositivos distintos), uno que conecta con el software OpenHAB, que permite mandarle órdenes y un último nodo que conectaría con ambos a la vez. Los nodos ROS para los dispositivos IoT mencionados se definen a continuación.

3.8 ROSLAUNCH

ROS incluye en su instalación una herramienta para controlar la puesta en marcha de sistemas complejos con múltiples nodos. Esta herramienta es Roslaunch, que nos permite definir estos sistemas mediante lenguaje XML, así como configurar y cambiar nombres y parámetros de los nodos todo desde estos archivos XML.

El uso de ROSLAUNCH en este proyecto se ha empleado para definir los entornos que se iban a probar. Por un lado, los archivos más simples de Roslaunch definen las zonas del hotel con sus respectivos nodos IoT, como muestra la figura 42:

```
<launch>
    <group ns="zona_comun_1">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.RosjavaNodeArd">
            <param name="agent" value="name1"/>
            <param name="zone_name" value="zona comun 1"/>
        </node>
        <node pkg="ros_iot" type="listener.py" name="listener_" output="screen">

        </node>
    </group>
</launch>
```

Figura 42 - Roslaunch simple con dos nodos

Este roslaunch lanzaría dos nodos, uno que es una clase que hereda de AbstractRosjavaNode y un nodo en Python que simplemente escucha los mensajes y los imprime en pantalla, empleado para depurar. Otro ejemplo del uso de roslaunch para realizar las pruebas se muestra en la figura 43:

```
<launch>
    <group ns="zona_comun_1">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.MyRosjavaNode">
            <param name="agent" value="Agente 1"/>
            <param name="zone_name" value="1 - Zona comun 1"/>
        </node>
    </group>
    <group ns="zona_comun_2">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.MyRosjavaNode">
            <param name="agent" value="Agente 2"/>
            <param name="zone_name" value="2 - Zona comun 2"/>
        </node>

    </group>
    <group ns="zona_comun_3">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.MyRosjavaNode">
            <param name="agent" value="Agente 3"/>
            <param name="zone_name" value="3 - Pasillo planta 1"/>
        </node>

    </group>
    <group ns="zona_comun_4">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.MyRosjavaNode">
            <param name="agent" value="Agente 4"/>
            <param name="zone_name" value="4 - Pasillo planta 2"/>
        </node>
    </group>
</launch>
```

Figura 43 - Roslaunch con el mismo nodo lanzado en varios namespace

Este archivo se encarga de lanzar 4 zonas distintas del hotel, para poder probar que cada una de las zonas crea su agente por separado y que no existe conflicto entre agentes y zonas. Esto se ha logrado gracias a una característica de ROS que es la de el uso de los *namespace*. Por defecto, todo lo que se crea en ros, tanto nodos como *topics*, se crean en el *namespace* "/", como si fuese el inicio del sistema operativo Linux. Un nodo que crease un *topic* llamado "chatter" lo crearía y su

nombre sería el de "/chatter". Al cambiar el *namespace* y utilizar valores relativos, se puede lanzar el mismo nodo ROS en distintos *namespaces* para poder tener varias instancias del nodo. El anterior nodo, por ejemplo, se podría lanzar en el *namespace* por defecto y luego otro en el *namespace* "/another", y se tendría así dos topics, uno "/chatter" y el otro "/another/chatter".

Con esto el proceso de lanzar el sistema queda simplificado a un único comando en la terminal. Al tener todos los archivos sincronizados mediante un repositorio en Github, cada máquina tiene acceso a los archivos y somos capaces de lanzar cada uno de una manera sencilla y sincronizada ante cualquier cambio en el desarrollo.

3.9 Instanciación y creación de clases del sistema completo

Por último, vamos a explicar cómo una vez implementado todo el sistema vamos a lanzar y ejecutar cada parte de este. Para ello disponemos de la herramienta Roslaunch anteriormente mencionada para la parte de ROS. Para la parte de JADE el lanzamiento de los agentes se lanza mediante comando por terminal, así que se ha creado un script auxiliar llamado "launch_jade_core.sh" que se encargará de lanzar la parte de planificación en JADE.

El proceso se describe en el diagrama de la figura 44:

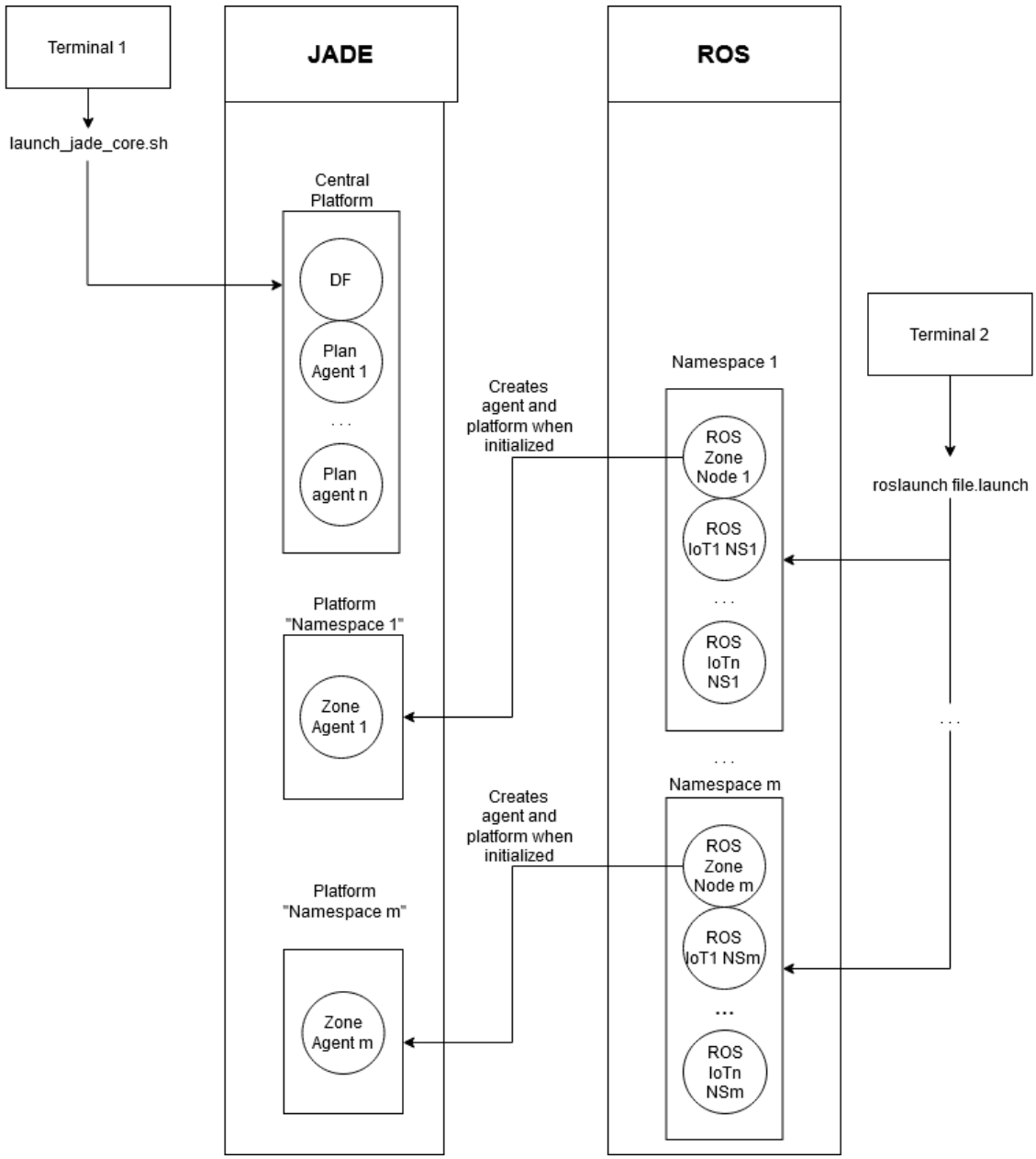


Figura 44 - Ejecución e instanciación de clases

El proceso se realiza ejecutando en dos terminales el script de JADE por un lado y el programa roslaunch en el otro. El script vemos que instancia la plataforma central y en ella lanza todos los agentes encargados de la planificación, descritos dentro del script. Por otra parte, el programa roslaunch tiene descrito las zonas del hotel divididas en "namespaces". En cada namespace, instanciará un nodo ROS de zona, que a su vez creará su representación dentro del sistema multiagente con un agente de zona, creado en una plataforma con el mismo nombre que el namespace.

Tras esto, creará los nodos de ROS encargados de los elementos IoT necesarios dentro del namespace. Este proceso se repite por cada una de las zonas descritas en el archivo “roslaunch”.

Para lanzar cada sistema que se desee el procedimiento será el de cambiar el archivo roslaunch empleado en la segunda terminal, puesto que el núcleo central de planificación será el mismo en cada sistema.

4. Pruebas realizadas

Dado a que se ha trabajado con varias tecnologías distintas y el objetivo era el de lograr interconectar estas tecnologías, a la hora de realizar las pruebas se han diferenciado los siguientes tipos de pruebas que se han ido realizando conforme avanzaba el desarrollo del proyecto:

- Pruebas de los sistemas de manera individual. Esto son pruebas en JADE y ROS respectivamente para comprobar que los mensajes son creados y enviados a través de los sistemas de manera separada, y que los mensajes son los correctos con los datos recibidos.
- Pruebas de comunicación entre las tecnologías empleando el middleware desarrollado. Aquí comprobaremos que el middleware desarrollado es capaz de transmitir mensajes desde el sistema JADE al sistema ROS.
- Pruebas de sistemas al completo con elementos IoT reales. Por último, realizaremos pruebas sobre sistemas completos con elementos IoT físicos, para demostrar así la viabilidad en entornos reales. En el primer entorno comprobaremos como se crean las distintas zonas que controlan los elementos IoT, cada uno por separado y distribuido en un ordenador de la red. En el segundo comprobaremos como una zona puede tener varios nodos con elementos IoT distintos y como es capaz de controlar todos los elementos.

4.1 Programas y herramientas auxiliares

Bash: se han empleado scripts programados en Bash para automatizar tareas en el desarrollo y testeo del sistema. La mayoría de estos simplemente se encargan de lanzar los comandos correspondientes y eliminar la necesidad de estar constantemente reescribiéndolos en la terminal, así como para configurar el entorno de ejecución.

VirtualBox: empleado para la ejecución de la máquina de desarrollo y de la máquina de ejecución en los entornos de prueba (véase Entorno de pruebas 1). Necesario cambiar la configuración de red para poder comunicar todos los dispositivos.

Git: usada para control de versiones, así como para tener acceso desde las distintas máquinas de pruebas al código necesario para realizar las pruebas de entorno.

Python y C++: ROS permite la implementación de Nodos de base con código en C++ o Python como principales lenguajes de desarrollo. Se han empleado ambos lenguajes, dependiendo del paquete a usar o de si era para realizar más simples de conexión para implementar los distintos nodos ROS empleados en el sistema.

Arduino IDE: empleado para desarrollar el código de la placa Arduino que responderá a las órdenes que envíe el nodo ROS.

SSH: para conectar con la Raspberry Pi.

4.2 Pruebas individuales

4.2.1 JADE

El sistema de planificación al completo se corresponde de los siguientes agentes, que cumplen las siguientes tareas:

- Agente de Ahorro Energético, se encarga de decidir los valores deseados de temperatura, humedad y luminosidad y de decidir qué elementos IoT se van a emplear para lograrlos (por ejemplo, si hace buen clima en el exterior se abrirán las ventanas para no usar la calefacción o el aire acondicionado).
- Agente de Servicios. Se encarga de ofrecer la información tanto climática como de ocupación del hotel.
- Agente de Seguridad. Registra e informa al personal de las alertas que llegan de seguridad al sistema,

Además de esto, vamos a hablar de las herramientas que se han empleado para depurar el Sistema Multiagente. JADE nos proporciona varias herramientas que facilitan esta labor, todas ellas implementadas a modo de Agente dentro del software.

En primer lugar, tenemos al agente "Introspector". Este agente sirve para comprobar el estado interno del agente, y poder ver así cosas como el comportamiento actual, ver una lista de mensajes enviados y recibidos, cambiar el estado del agente, etc. Por ejemplo, en la figura 45 podemos ver a un agente de zona que se ha lanzado de manera manual para comprobar el paso de mensajes completo dentro del sistema multiagente:

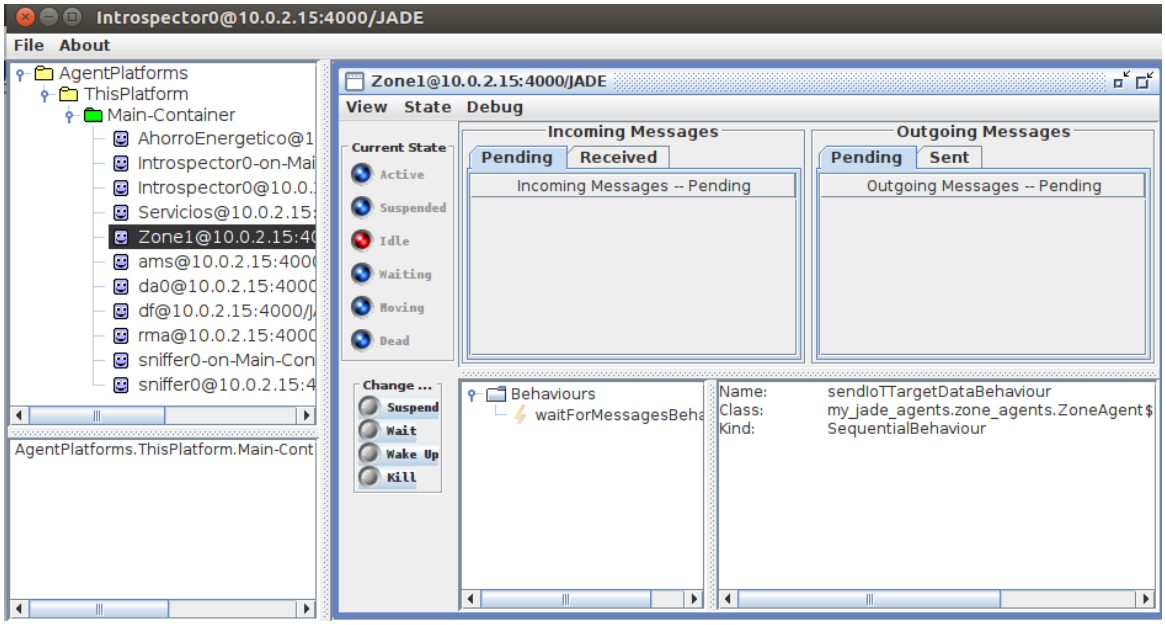


Figura 45 - Agente Introspector sobre agente de zona Zone1

El segundo agente que vamos a emplear es el DummyAgent. Este agente, como su nombre indica, no se comporta de manera autónoma, sino que sirve para que el usuario pueda testear y mandar mensajes dentro del sistema para poder ver que ocurre y que en qué acciones repercuten. Permite editar todos los campos que hay dentro de un mensaje FIPA, así como ver las respuestas que llegan e inspeccionarlas para ver el detalle del contenido.

La figura 46 muestra cómo se ha enviado una petición (REQUEST) al agente de zona Zone1 solicitando los objetivos a conseguir. Como el mensaje se ha formulado correctamente y se ha enviado al destinatario correspondiente, se recibe un mensaje del tipo INFORM con los datos (columna derecha).

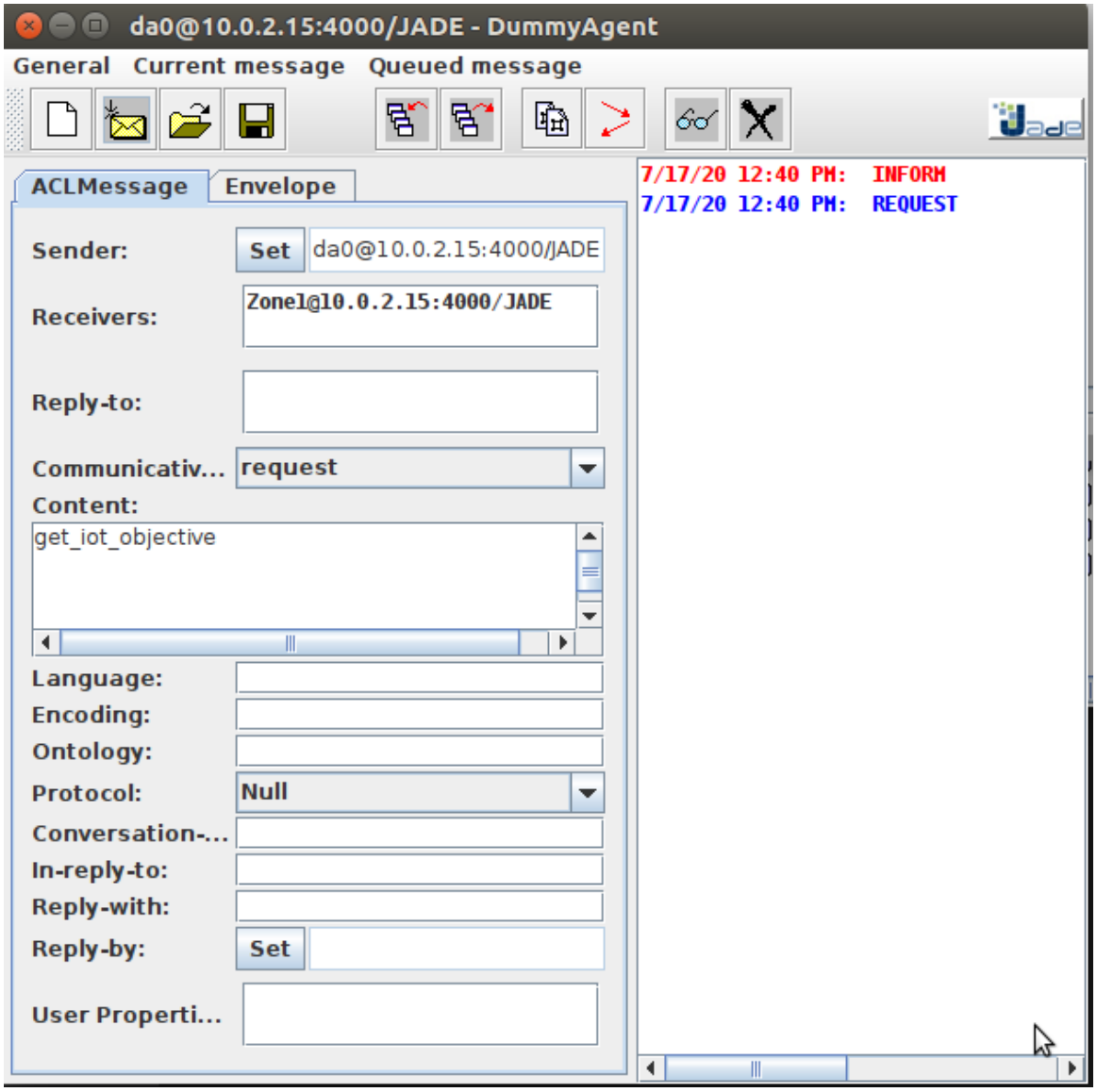


Figura 46 - Interfaz de agente DummyAgent

Por último, pese a que este agente es útil para enviar mensajes y ver que las respuestas que llegan son las que corresponden, no tenemos idea del proceso intermedio que ha habido para lograr esta respuesta. Por ello, JADE nos da el último agente empleado en el proceso de depuración que es el "Sniffer". Este

agente permite seleccionar agentes activos en el sistema y poder ver las conversaciones y mensajes intercambiados entre ellos y entre otros agentes del sistema.

En la figura 47 se puede ver todas las herramientas en conjunto para probar que el sistema funcionaba correctamente:

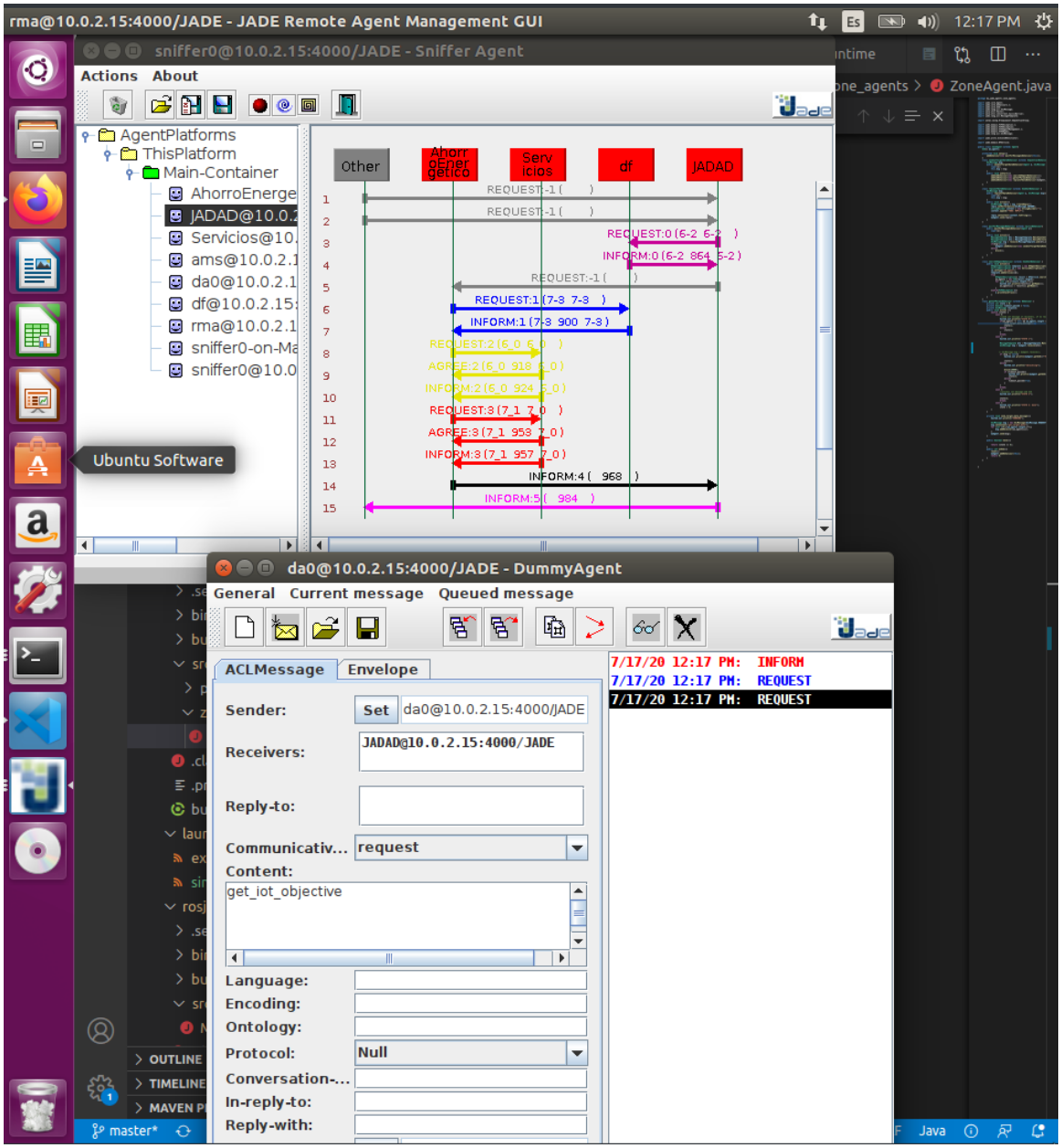


Figura 47 - Ejemplo del proceso de depuración

En primer lugar, en la parte superior vemos al agente Sniffer, mientras que debajo vemos al DummyAgent. Se han enviado dos mensajes a un agente de zona (JADAD), como se puede ver tanto en la parte de la derecha del dummy cono en el diagrama del Sniffer. El primero ha sido ignorado al no estar correctamente formado, mientras que el segundo si ha recibido una respuesta. Podemos ver que, para lograr construir esta respuesta, el agente ha tenido que hacer los siguientes pasos:

1. Solicitar en el DF un agente de Ahorro Energético (primera conversación rosa)
2. Llamar al ahorro energético para recibir los parámetros IoT (request gris a AhorroEnergetico)
3. El agente de ahorro energético necesita información para construir esta respuesta
    a. Solicita al DF un agente de Servicios (conversación azul)
    b. Consulta la ocupación del hotel (primera conversación, color amarillo)
    c. Solicita los datos climáticos exteriores (segunda conversación, color rojo)
    d. Crea la respuesta y la devuelve (inform negro)
4. El agente procesa lo obtenido y responde (inform rosa final)

Con esto vemos como efectivamente los agentes del sistema son capaces de comunicarse con otros y vemos como los agentes de planificación buscan la información necesaria dentro del sistema y crean los objetivos correspondientes.

4.2.2 ROS

ROS también nos proporciona multitud de herramientas para depurar el sistema. Las dos empleadas en el proceso de desarrollo, sin embargo, han sido las

siguientes: por un lado, los grafos generados mediante rqt_plot y por otra la salida por terminal.

Rqt_graph es una herramienta que tiene ROS que nos permite visualizar el estado de grafo computacional de ROS actual, así como ver cómo se comportan los valores que se transmiten a través de los distintos topics. Para nosotros, sin embargo, nos importa la opción que crea y visualiza el grafo al completo del sistema, ya que con el podremos visualizar si se han creado los nodos correctamente mediante los archivos roslaunch.

Vamos a poner como ejemplo los sistemas expuestos en el apartado de ROSLAUNCH. En primer lugar, un nodo agente con un listener en Pytohn para comprobar que los mensajes se lanzan correctamente:

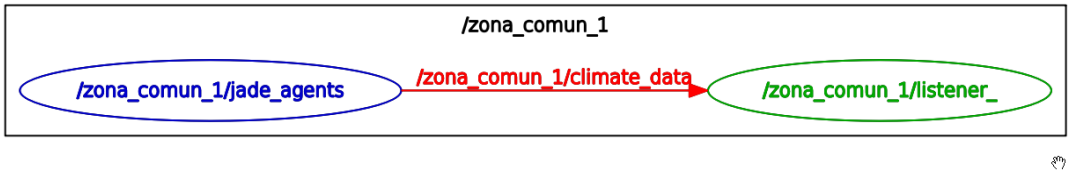


Figura 48 - Sistem Roslaunch simple visto en rqt_plot

En el grafo de la figura 48 se pueden ver 2 nodos dentro del namespace /zona_comun_1: el nodo jade_agents y el nodo listener_, y vemos que este segundo nodo esta suscrito al topic sobre el que publica el primer nodo “climate_data”. En la figura 49 podemos ver el sistema con varios nodos en distintos espacios de nombre:

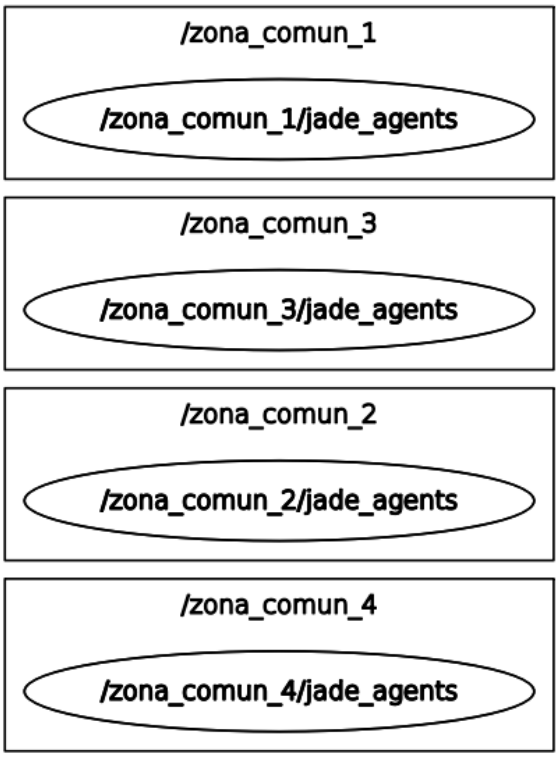


Figura 49 - Roslaunch de varios nodos a traves de rqt_plot

La terminal se ha empleado para ver de una manera rápida que los mensajes que se enviaban a través de los distintos topics están formados correctamente y que no hay errores al enviarlos. Esto se verá mejor en el siguiente apartado.

Tras confirmar que ROS estaba funcionando como se esperaba, el siguiente paso consistió en crear el nodo AbstractRosjavaNode y en comprobar que se creaba el agente de zona correspondiente, se creaba el JADEGateway para comunicarse y se enviaban los parámetros y acciones a lo largo del sistema al completo.

4.2.3 Middleware, prueba End-to-End

Por último, antes de empezar con las pruebas de entorno, era necesario comprobar que el sistema al completo era capaz de enviar información de un extremo a otro, pasando de JADE a ROS y enviando en cada momento la información pertinaz.

El sistema usado ha sido el sistema simple de roslaunch, ya que contiene una gente de zona y un nodo ros representado el IoT, que simulara un nodo final del sistema. Para empezar, lanzamos el sistema multiagente y lanzamos en otra terminal el roslaunch correspondiente. Si miramos el agente Sniffer veremos que muestra los siguientes mensajes (figura 50):

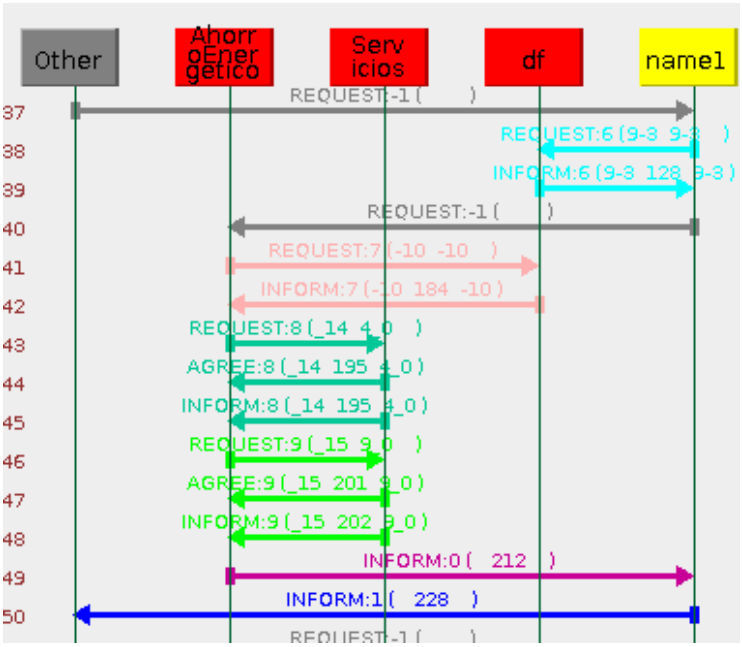


Figura 50 - Snifer en prueba end-to-end

Se puede ver como un agente externo ha enviado una petición request a el agente de zona "name1", que se ha realizado el proceso completo para la obtención de datos y se ha enviado de vuelta a dicho agente.

Por terminal, al lanzar el sistema mediante roslaunch, podemos observar en la figura 51 que se inicia ROS y se lanzan todos los nodos especificados en el archivo Roslaunch:

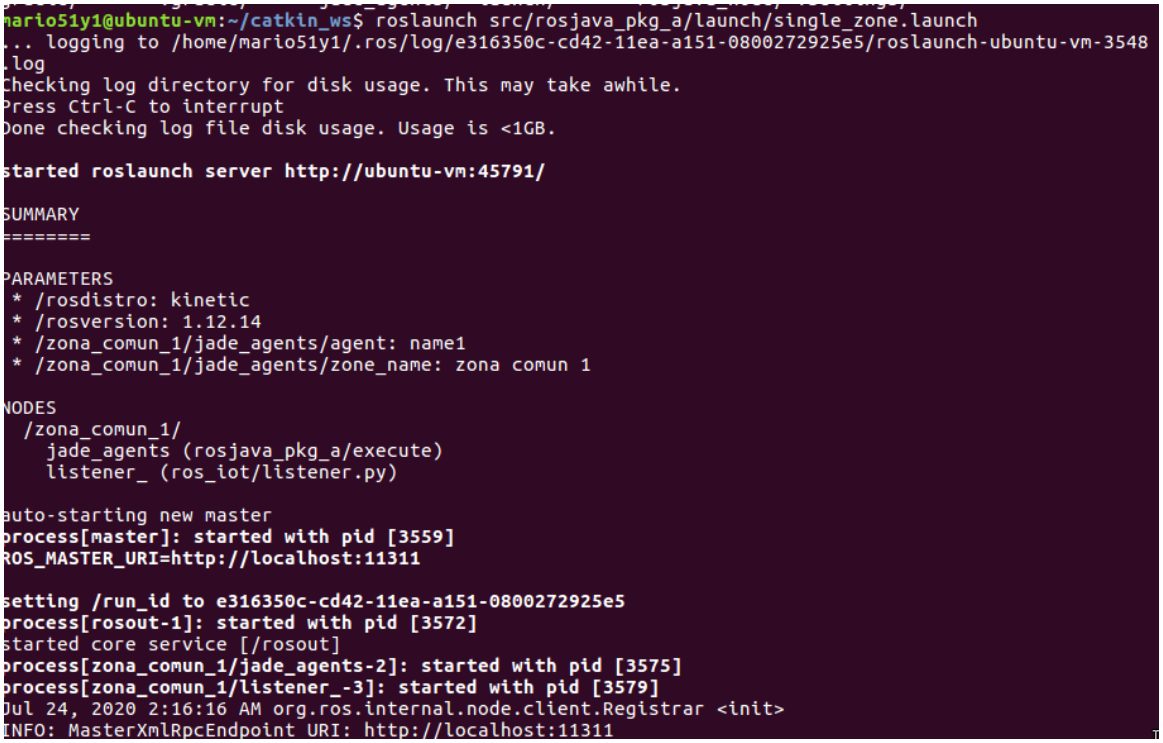


Figura 51 - Terminal end-to-end 1, inicialización de ROS

A continuación, si seguimos viendo el output en la terminal, vemos que el nodo crea y registra un Agente dentro del sistema multiagente, dentro de la plataforma "zona común 1" (figura 52).

```
Jul 24, 2020 2:16:16 AM org.ros.internal.node.client.Registrar <init>
INFO: MasterXmlRpcEndpoint URI: http://localhost:11311
Jul 24, 2020 2:16:16 AM org.ros.internal.node.client.Registrar onPublisherAdded
INFO: Registering publisher: Publisher<PublisherDefinition<PublisherIdentifier<NodeIdentifier</zona_c
omun_1/jade_agents, http://127.0.0.1:43243/>, TopicIdentifier</rosout>>, Topic<TopicIdentifier</rosou
t>, TopicDescription<rosgraph_msgs/Log, acffd30cd6b6de30f120938c17c593fb>>>>
Jul 24, 2020 2:16:16 AM org.ros.internal.node.client.Registrar callMaster
INFO: Response<Success, Registered [/zona_comun_1/jade_agents] as publisher of [/rosout], [http://ubu
ntu-vm:45291/]>
Jul 24, 2020 2:16:16 AM org.ros.internal.node.topic.DefaultPublisher$1 onMasterRegistrationSuccess
INFO: Publisher registered: Publisher<PublisherDefinition<PublisherIdentifier<NodeIdentifier</zona_co
mun_1/jade_agents, http://127.0.0.1:43243/>, TopicIdentifier</rosout>>, Topic<TopicIdentifier</rosout
>, TopicDescription<rosgraph_msgs/Log, acffd30cd6b6de30f120938c17c593fb>>>>
Jul 24, 2020 2:16:16 AM jade.core.Runtime beginContainer
INFO: ----------------------------------
    This is JADE 4.5.0 - revision 6825 of 23-05-2017 10:06:04
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
Jul 24, 2020 2:16:16 AM jade.imtp.leap.LEAPIMTPManager initialize
INFO: Listening for intra-platform commands on address:
- jicp://10.0.2.15:1099

Jul 24, 2020 2:16:16 AM jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
Jul 24, 2020 2:16:16 AM jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
Jul 24, 2020 2:16:16 AM jade.core.BaseService init
INFO: Service jade.core.resource.ResourceManagement initialized
Jul 24, 2020 2:16:16 AM jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
Jul 24, 2020 2:16:16 AM jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
Jul 24, 2020 2:16:17 AM jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container zona comun 1@10.0.2.15 is ready.
--------------------------------------------
```

Figura 52 - Terminal end-to-end 2, inicialización de plataforma JADE 'zona_comun_1'

Tras esto, siguiendo con el esquema que hemos descrito, el nodo crea los publishers que va a emplear para transmitir los datos al resto de nodos ROS con elementos IoT que hay en sui zona, es decir, en su namespace (figura 53).

```
Jul 24, 2020 2:16:17 AM org.ros.internal.node.client.Registrar onPublisherAdded
INFO: Registering publisher: Publisher<PublisherDefinition<PublisherIdentifier<NodeIdentifier</zona_c
omun_1/jade_agents, http://127.0.0.1:43243/>, TopicIdentifier</zona_comun_1/climate_data>>, Topic<Top
icIdentifier</zona_comun_1/climate_data>, TopicDescription<ros_iot/Climate_act_data, 8f0f35728fb04934
6e7499017ec7680c>>>>
```

Figura 53 .- Terminal end-to-end 3, inicialización de Publisher por parte

A continuación, el nodo de ROS inicia el proceso de crear el agente JADEGateway (dentro de la plataforma "zona común 1-1"), le añade el comportamiento para comunicarse dentro del Sistema con el nodo de Zona correspondiente e inicia su ejecución. Esto correspondería con la petición de un agente externo ("other" en imagen del Sniffer) al agente, que no se puede visualizar como de un agente JADEGateway ya que al finalizar esta ejecución el agente no persiste en memoria (figura 54).

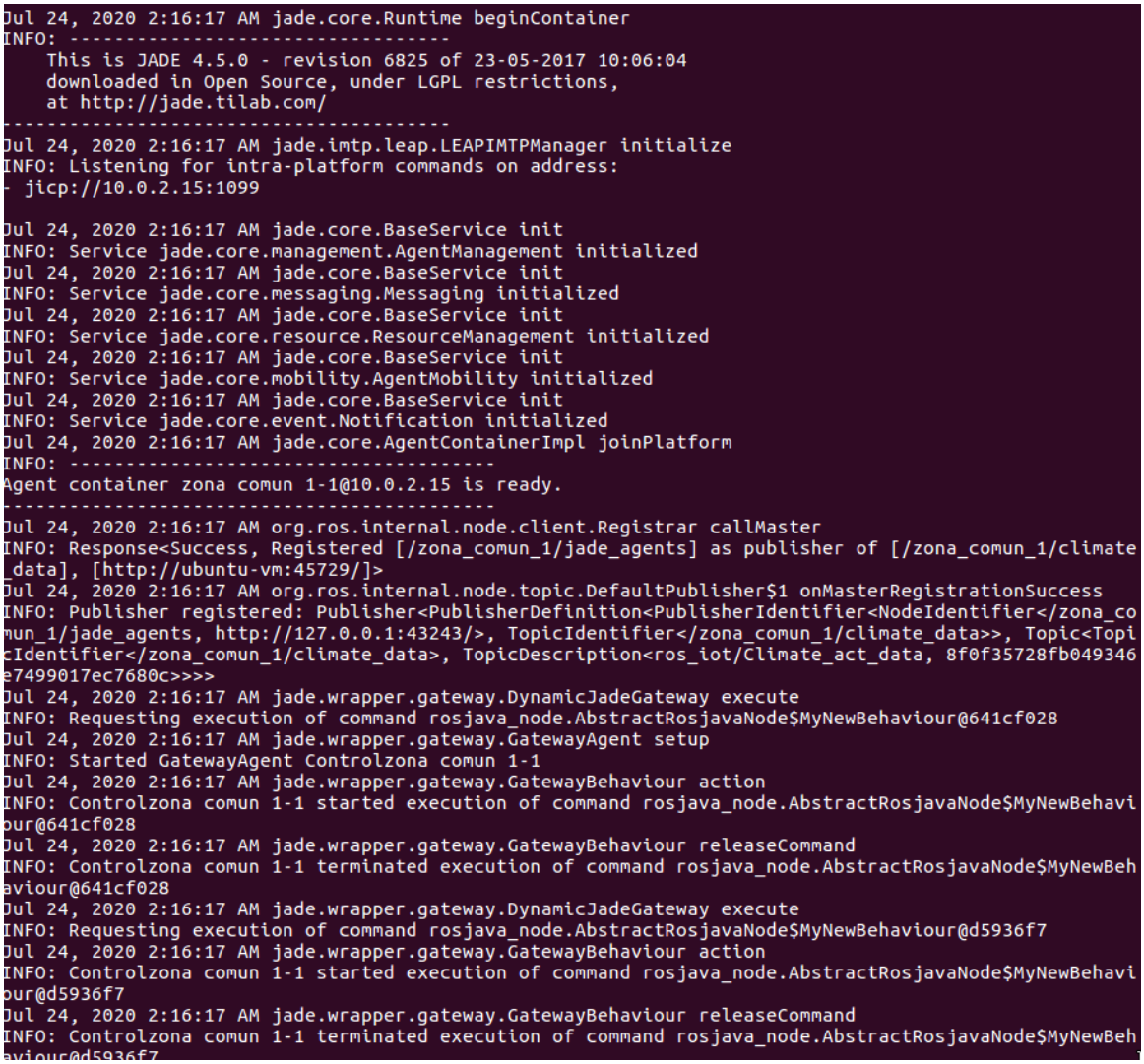


Figura 54 - Terminal end-to-end 4, comunicación mediante JADEGateway

Por último, en la figura 55 vemos la respuesta que el nodo ROS genera y envía al topic correspondiente.

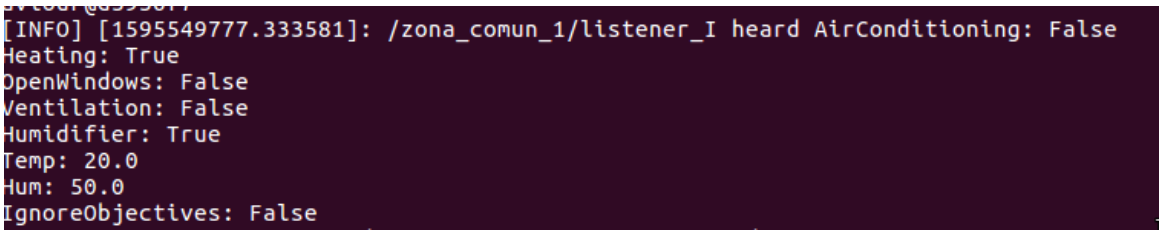


Figura 55 - Terminal end-to-end 5, respuesta de nodo ROS

Con todo este proceso hemos podido comprobar que efectivamente, la generación de peticiones dentro de ROS se transforma en una petición para los agentes en

JADE y que, cuando finaliza, vuelve a ROS la información necesaria de planificación para que pueda crear las acciones en respuesta que serán enviadas a los elementos IoT que hay en la zona.

4.2.4 Reactividad de agente de zona

A pesar de tener un sistema de planificación que nos dirá que objetivos son los que se buscan cumplir, es necesario también que los agentes sean capaces de reaccionar de manera rápida ante situaciones peligrosas o delicadas. Así, se han implementado alertas de seguridad que al recibirlas el agente reacciona rápidamente sin consultar con el sistema de planificación.

Las alertas son de dos tipos, fuego y fuga de agua, y la reacción será, por ejemplo, la de abrir ventanas y activar la ventilación en caso de incendio con el fin de expulsar del edificio todo el humo posible. Esto se verá reflejado en el mensaje con las ordenes de actuación que los agentes envían a sus nodos.

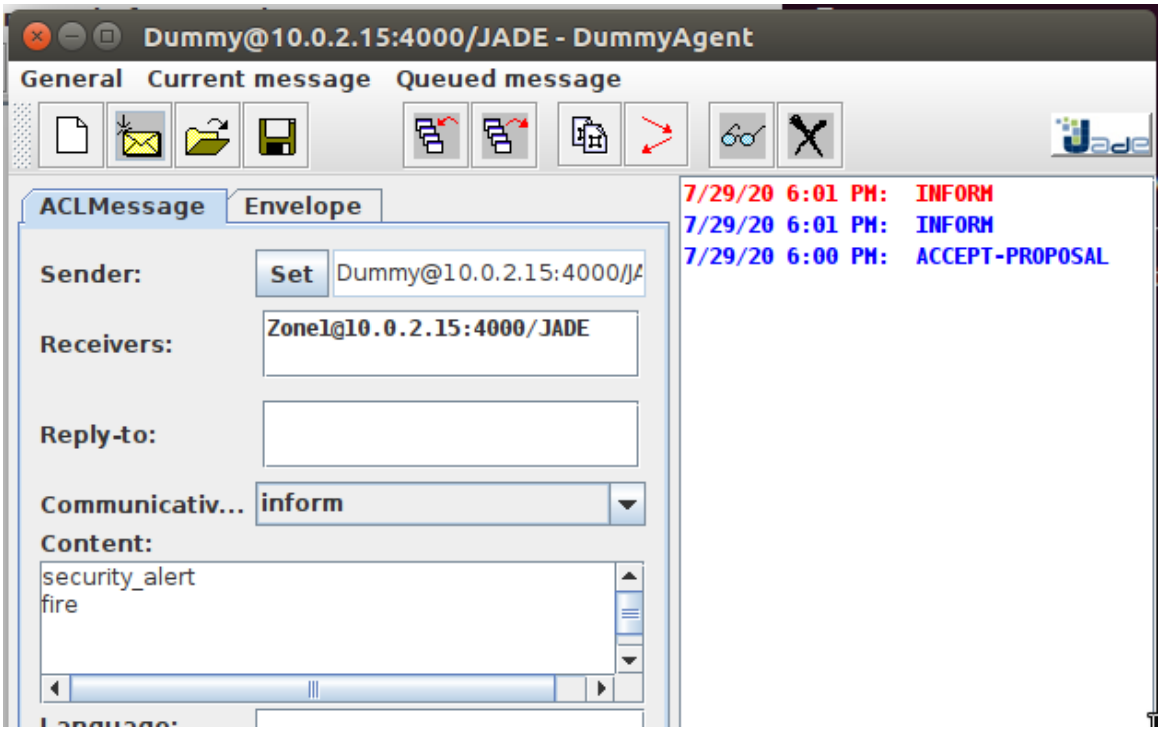


Figura 56 - Reactividad de agente 1, mensaje de ejemplo

Con el DummyAgent simulamos un mensaje de alerta. Vemos en la figura 56 de nuevo que el primer menaje no genera ningún tipo de respuesta al no estar formado correctamente.

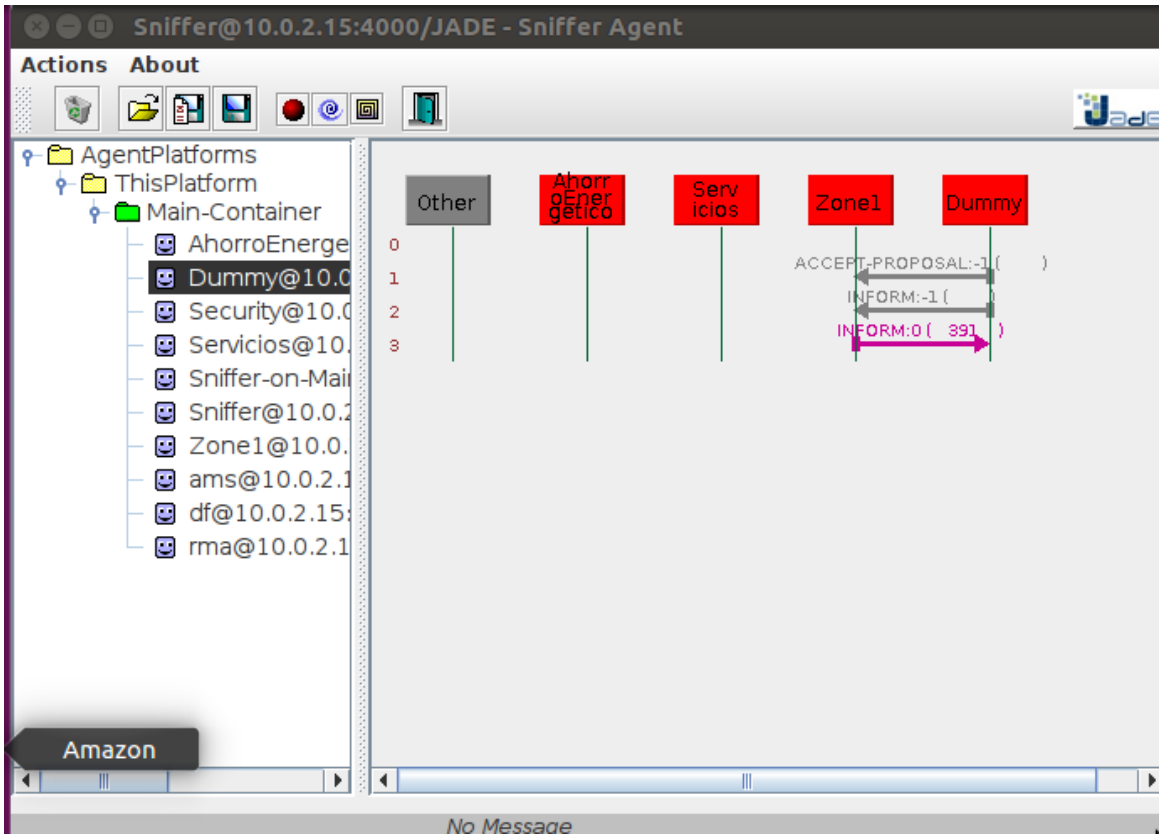


Figura 57 - Reactividad de agente, 2

En la figura 57 observamos como efectivamente al llegar el mensaje de alerta el agente de zona no ha necesitado comunicar con el resto de los agentes, sino que ha mandado una respuesta de actuación de manera inmediata a los elementos IoT que controla, demostrando así la capacidad de reacción del agente.

4.3 Pruebas de sistema

4.3.1 Lógica interna básica de los elementos IoT

Para ver las pruebas de entorno y los resultados obtenidos es necesario explicar la lógica interna de los agentes para decidir sobre qué elementos IoT van a actuar en según qué circunstancias. Puesto que el objetivo de este TFG no es el de conseguir un sistema real que tenga toda la complejidad en cuenta se ha simplificado estas reglas de actuación para que sea visible como actúan los elementos, pero a su vez tengan en cuenta los aspectos de la parte de planificación.

En primer lugar, se busca conseguir tres valores objetivo:

- Temperatura
- Humedad
- Luminosidad

Estos parámetros objetivo se emplearán junto a otros datos sobre la situación climática exterior para decidir qué elementos han de actuar del sistema IoT, separados en 2 tipos: climáticos e iluminación.

Los elementos IoT a emplear corresponden a los siguientes:

- Climáticos
    - Apertura de ventanas
    - Calefacción
    - Aire acondicionado
    - Ventilación
    - Humidificador
- Iluminación
    - Bombilla con regulación de intensidad de Xiaomi
    - Toldos
    - Persianas

Además de esto, existirá un valor para decir si los valores de actuación corresponden a una respuesta a una alerta de seguridad, y por lo tanto no se busca el cumplir los objetivos sino mantener la seguridad de los huéspedes del hotel.

Así pues, teniendo en cuenta los datos de entrada y las salidas quedaría todo de la siguiente manera:

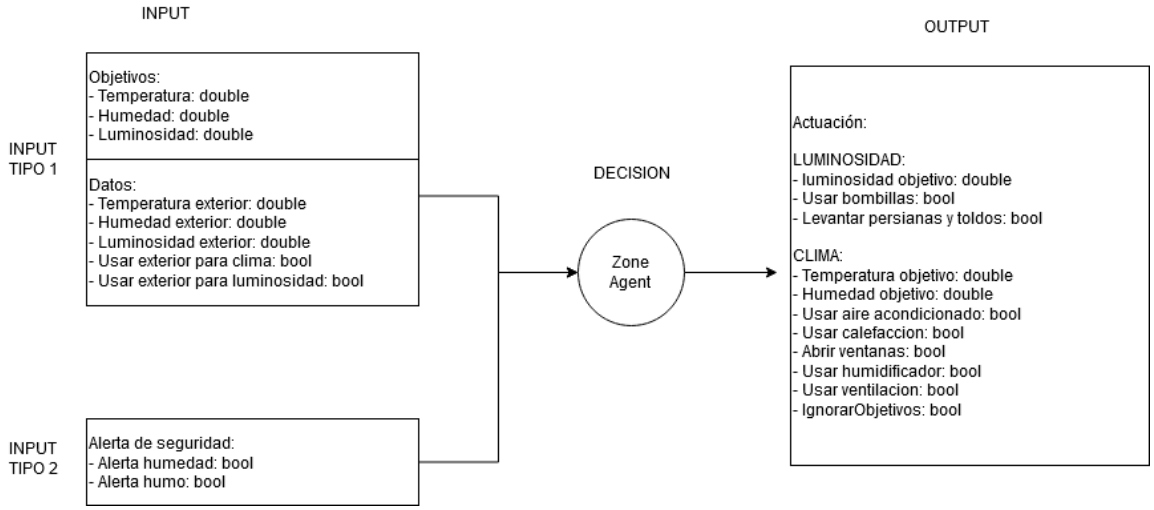


Figura 58 - Decisión y actuación de agente sobre IoT

El encargado de decidir si se puede utilizar el exterior o no para los objetivos será el Agente de Ahorro Energético, que determinará si esto es posible en función de valores como la velocidad del viento o de si está lloviendo ahora mismo o no.

Los elementos actuadores se simularán en la placa Arduino con leds que indicas si se activan o no para la consecución de los objetivos.

4.3.2 Sistema 1: Dos zonas en dos ordenadores distintos cada una con un único nodo IoT

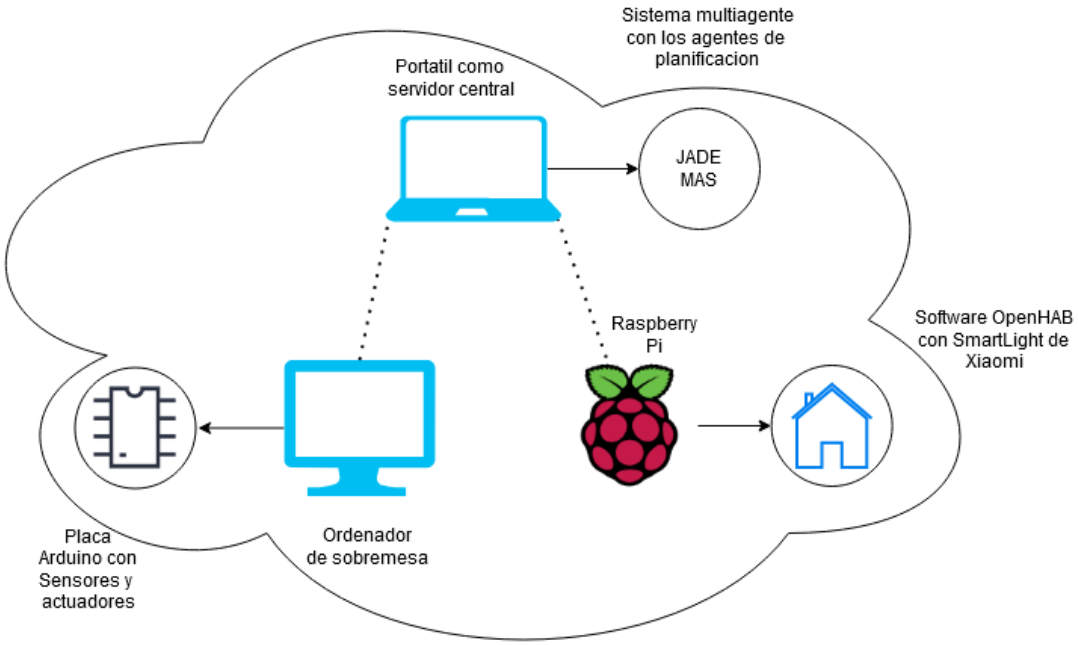


Figura 59 - Entorno de pruebas 1

El primer entorno de pruebas tiene como objetivo probar el sistema con diversos nodos de ROS representando varias zonas del hotel. El ordenador portátil hará el papel en ambos entornos de servidor central, donde se encuentra el sistema multiagente con los diversos agentes de planificación. Por otra parte, se dispone de otros dos ordenadores que harán el papel de cada unidad de computación asociada a una zona del hotel. Lo idóneo sería emplear ordenadores pequeños con la suficiente potencia de cómputo, como una raspberry pi, pero al no disponer del hardware suficiente, se ha utilizado un ordenador de sobremesa a modo de simulación de otro de estos dispositivos, con una máquina virtual Ubuntu donde se ejecutará el nodo de ROS.

Para empezar, siguiendo con el diagrama de decisión y actuación del apartado anterior, se han colocado varios leds en una placa Arduino que simulan los elementos actuadores. Puesto que en esta ocasión el nodo Arduino tendrá

únicamente los actuadores de los elementos IoT climáticos, se han añadido 6 LED de la forma que muestra la figura 60:

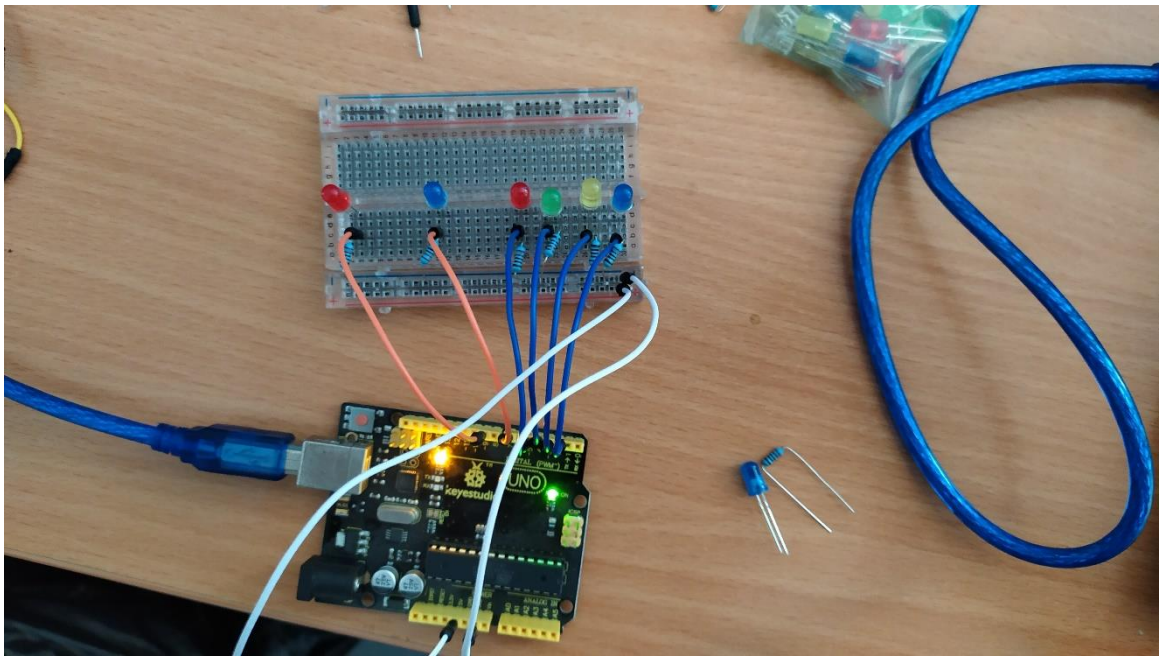


Figura 60 - Circuito placa Arduino

De izquierda a derecha, los leds corresponden a:

- Led de seguridad, ignorar objetivos IoT
- Uso de apertura de ventanas para climatización
- Humidificador
- Ventilación
- Calefactor
- Aire acondicionado

Lo primero ha sido probar que los LEDs eran accesibles desde el código y funcionaban correctamente (véase Anexo). Tras esto, y al haber comprobado ya que la placa Arduino era capaz de conectar con ROS, el proceso ha consistido en añadir el mensaje creado por nosotros para transmitir la información a través de ROS, hacer que conecte con el Publisher y crear un archivo que permita lanzar al agente y a este nodo para poder empezar a hacer pruebas:

```
<launch>
    <group ns="zona_comun_1">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.RosjavaNodeArd">
                <param name="agent" value="name1"/>
                <param name="zone_name" value="zona comun 1"/>
        </node>
        <node pkg="rosserial_python" type="serial_node.py" name="serial_node" args="/dev/ttyACM0">
        </node>
    </group>
</launch>
```

Figura 61 - Roslaunch de zona con nodo Arduino

Al lanzar el sistema, podemos observar con la herramienta de RosGraph que se ha creado el siguiente grafo (figura 62):

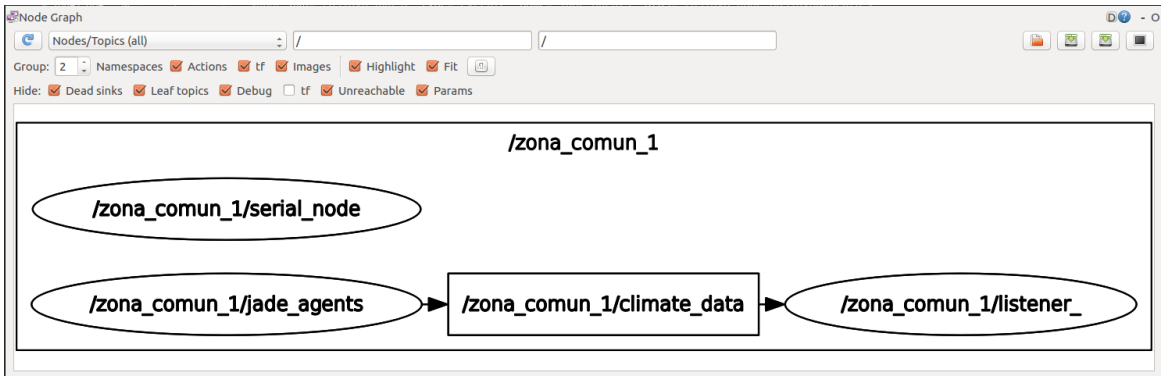


Figura 62 - Grafo zona con nodo Arduino (serial_node)

El hecho de que el nodo serial no aparezca como suscrito al topic climate_data se debe a que este nodo no está suscrito, sino que actúa como puente entre la placa Arduino, que a pesar de escuchar del topic no aparece directamente en el grafo. Por la terminal podemos observar que efectivamente se mandan las ordenes que corresponden. El mensaje que se envía al agente de zona es que muestra la figura 63:

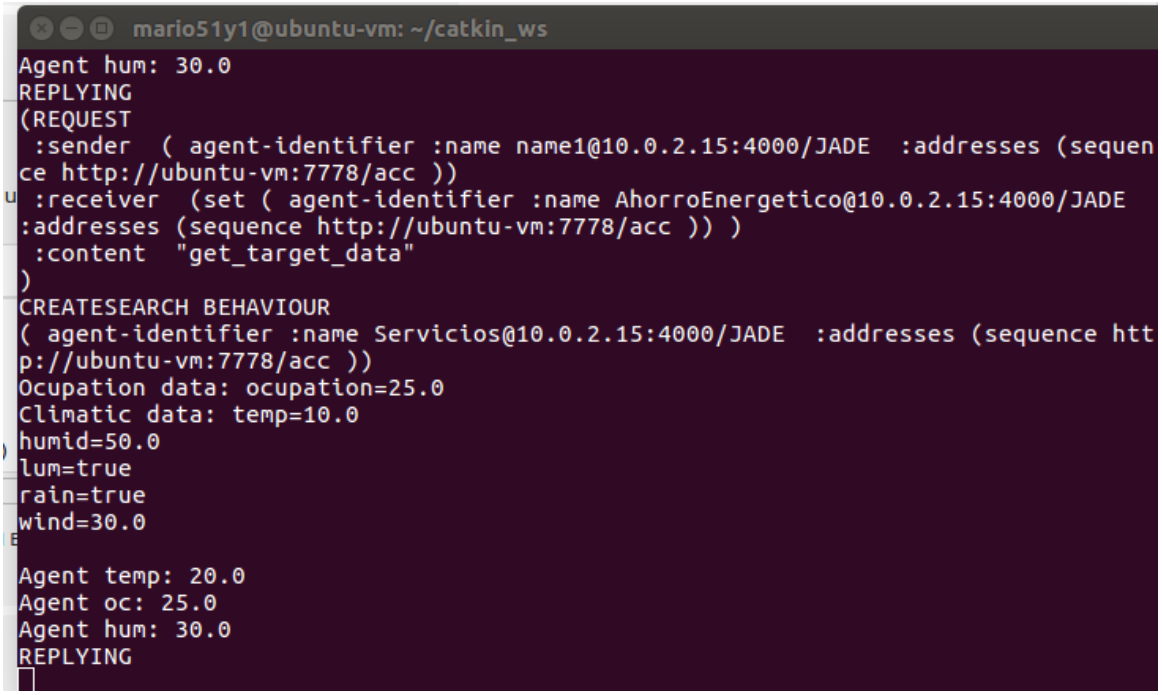


Figura 63 - Entorno 1, Mensaje JADE a agente

El agente procesa los datos recibidos y elabora un mensaje de actuación a los nodos que tiene asociados. En este caso, como solo tiene un nodo de climatización, genera el siguiente mensaje con los parámetros de actuación de los elementos de climatización (figura 64):

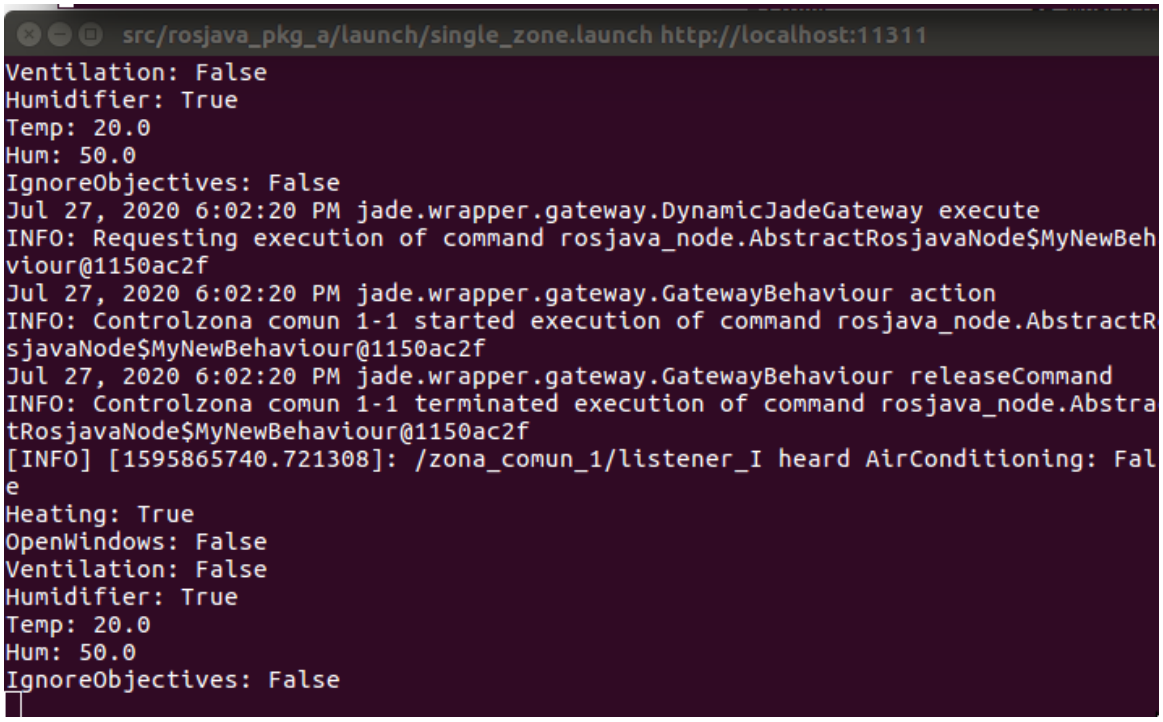


Figura 64 - Entorno de pruebas 1, mensaje agente-ROS clima

Y el nodo de la Arduino recibe este mensaje y activa los actuadores correspondientes:

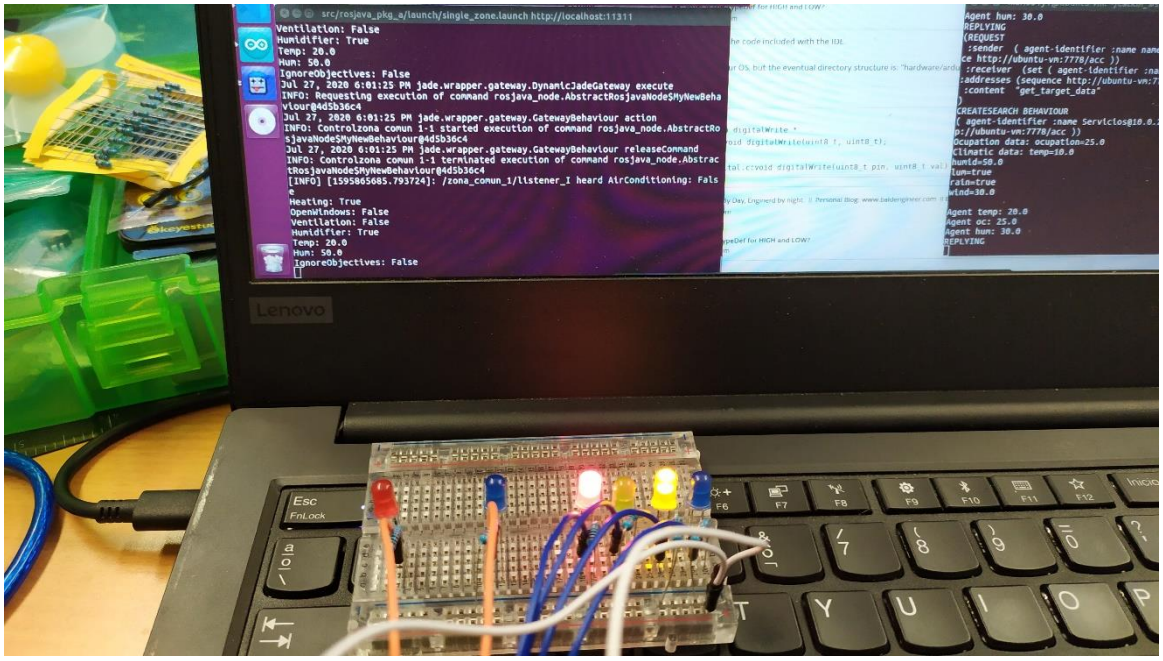


Figura 65 - Entorno de pruebas 1, nodo arduinio actuación

Podemos ver más ejemplos de como la placa Arduino reacciona correctamente y enciende los LEDS necesarios en las dos siguientes imágenes:

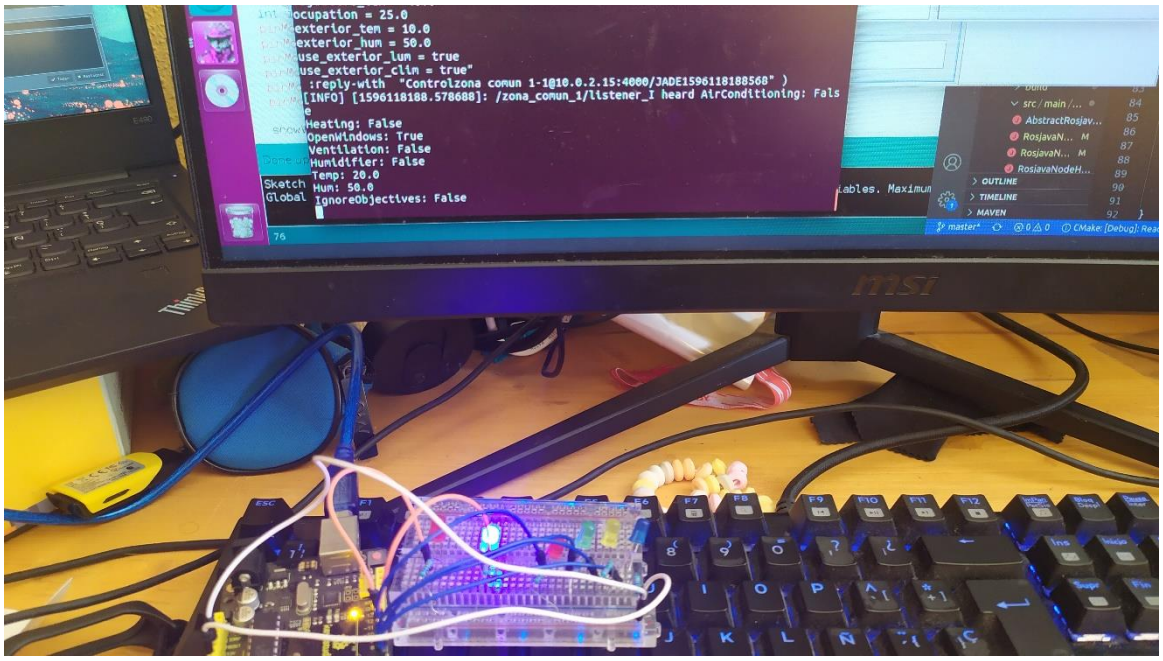


Figura 66 - Respuesta de arduino a mensaje ROS número 2

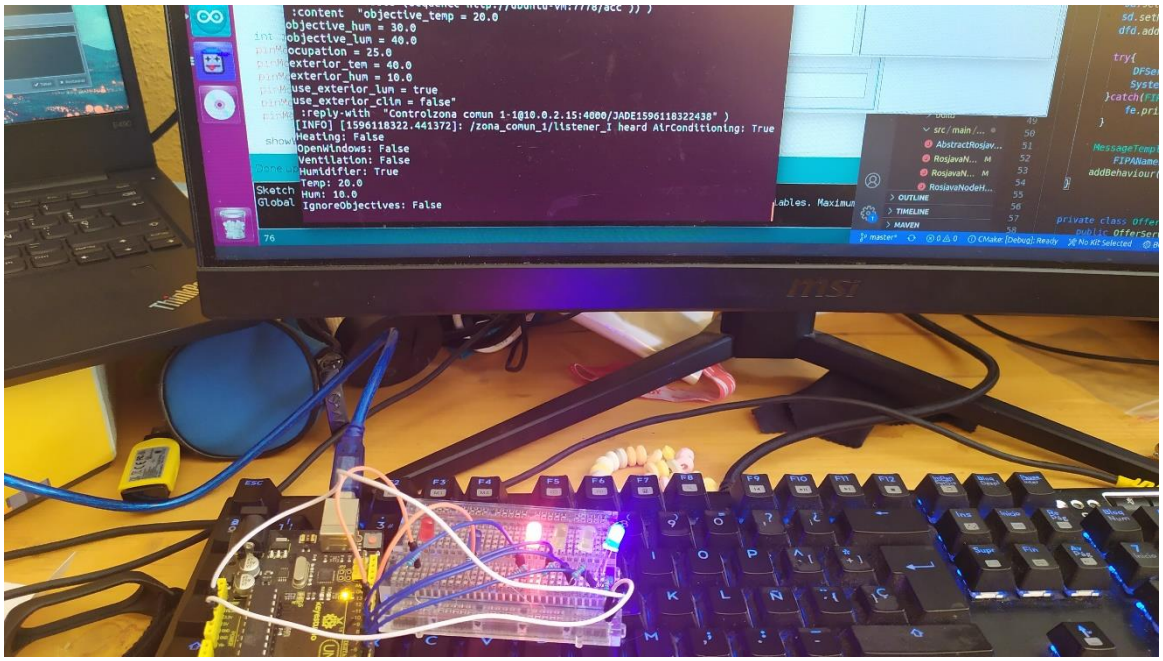


Figura 67 - Respuesta de arduino a mensaje número 3

Centrándonos en OpenHAB, el primer paso ha sido añadir la bombilla Xiaomi al software y establecer los Items para poder controlarla.

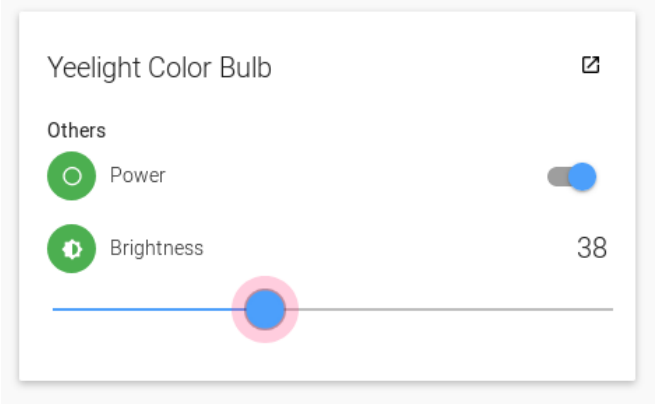


Figura 68 - Bombilla Xiaomi Yeelight dentro de OpenHAB

En la figura 69 podemos observar el JSON recibido en ROS con los elementos de OpenHAB disponibles:


```
members:
  0:
    link:          "http://192.168.1.50:8080/rest/items/YeelightColorBulb_RGBColor"
    state:         "NULL"
    type:          "Color"
    name:          "YeelightColorBulb_RGBColor"
    label:         "RGB Color"
    tags:          []
    groupNames:
      0:           "ROS"
  1:
    link:          "http://192.168.1.50:8080/rest/items/YeelightColorBulb_Power"
    state:         "ON"
    type:          "Switch"
    name:          "YeelightColorBulb_Power"
    label:         "Power"
    tags:          []
    groupNames:
      0:           "ROS"
  2:
    link:          "http://192.168.1.50:8080/rest/items/YeelightColorBulb_Brightness"
    state:         "16"
    stateDescription:
      pattern:     "%.0f"
      readOnly:    false
      options:     []
    type:          "Dimmer"
    name:          "YeelightColorBulb_Brightness"
    label:         "Brightness"
    tags:          []
    groupNames:
      0:           "ROS"
link:              "http://192.168.1.50:8080/rest/items/ROS"
state:             "NULL"
editable:          true
type:              "Group"
name:              "ROS"
label:             "ROS"
tags:              []
groupNames:        []
```


Figura 69 - Elementos OpenHAB visibles al nodo de control de ROS

Una vez se ha comprobado que efectivamente la bombilla es reconocida y manipulable desde el software es momento de comprobar que nuestro nodo es

capaz de procesar la información que le llega desde el Sistema Multiagente y transformarla en las ordenes de actuación correspondiente.

```
haviour@2f88eac8
Jul 30, 2020 1:19:48 AM jade.wrapper.gateway.GatewayBehaviour action
INFO: Controlzona comun 2-1 started execution of command rosjava_node.Abstract
RosjavaNode$MyNewBehaviour@2f88eac8
Jul 30, 2020 1:19:49 AM jade.wrapper.gateway.GatewayBehaviour releaseCommand
INFO: Controlzona comun 2-1 terminated execution of command rosjava_node.Abstr
actRosjavaNode$MyNewBehaviour@2f88eac8
[INFO] [1596064789.006992]: /zona_comun_2/openhabI heard
UseLightbulb: True
ColdLight: False
Intensity: 100.0
```

Figura 70 - Mensaje con órdenes de actuación para elementos de iluminación

Finalmente, lanzamos mediante roslaunch (figura 71) todos los elementos correspondientes a esta zona: el agente, el código que corresponde al paquete iot_bridge y nuestro agente que controla la bombilla Xiaomi con el respectivo grafo (figura 72):

```
<launch>
    <group ns="zona_comun_1">
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.RosjavaNodeHAB">
                <param name="agent" value="name1"/>
                <param name="zone_name" value="zona comun 1"/>
        </node>
        <node pkg="ros_iot" type="openhab.py" name="openhab" output="screen">
        </node>
        <include file="$(find iot_bridge)/launch/iot.launch"></include>
    </group>
</launch>
```

Figura 71 - Roslaunch de zona con nodo OpenHAB

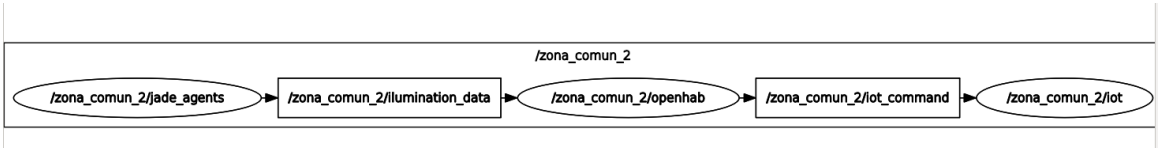


Figura 72 - Grafo de zona con nodo OpenHAB

Vemos que efectivamente se lanza el nodo que controla el software de Openhab (/iot) y que nuestro agente recibe los datos de iluminación y los transforma en comandos de openhab para controlar la bombilla.

La demostración de que la bombilla reacciona a los comandos se puede observar en el siguiente video (Nota: el comportamiento real no corresponde al mostrado.

La bombilla se enciende y apaga con el único fin de demostrar de manera visible que el sistema es capaz de modificar su estado):

https://youtu.be/nQ34UsFuNVk

Lanzamiento del sistema al completo

Finalmente, lo último que nos queda por probar de este sistema es que se lance de manera modular al completo. Para empezar, en nuestra máquina virtual con Ubuntu, lanzamos el sistema Multiagente, y desde nuestra raspberry lanzamos el nodo correspondiente. Si miramos dentro de JADE, podemos ver que se ha creado una nueva plataforma con un agente que correspondería al generado dentro de la raspberry. Podemos ver también como efectivamente la IP es distinta, de otra red, ya que la máquina virtual se encuentra en la red virtual 10.0.2.0/24 y la raspberry en la red asociada al router 192.168.1.0/24.

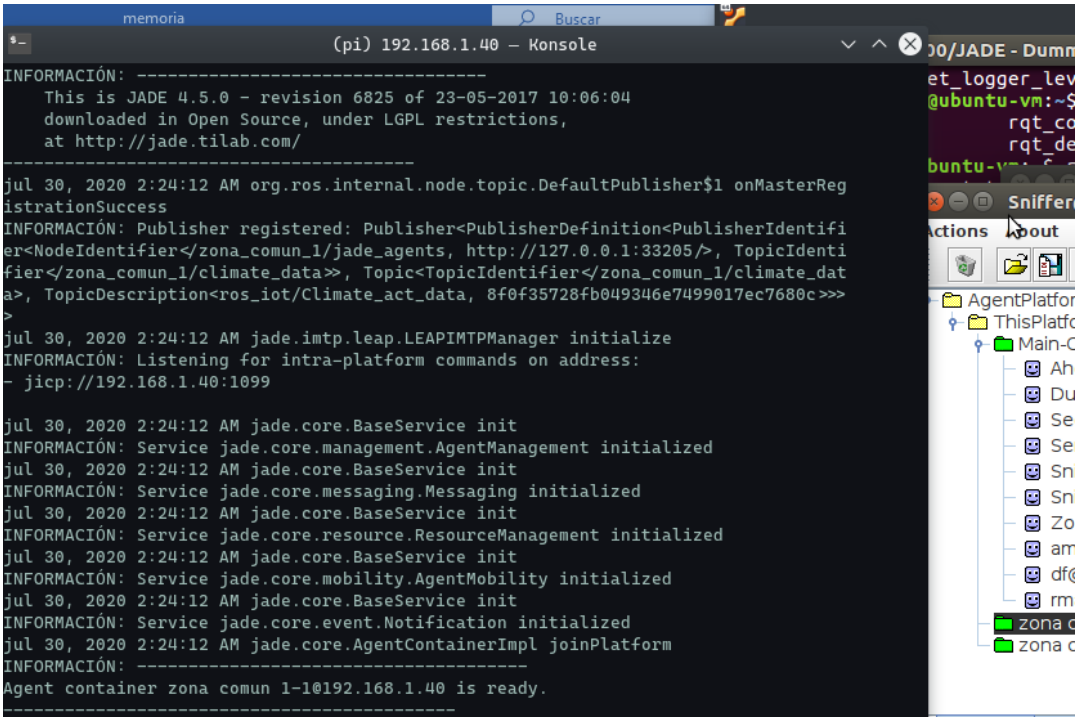


Figura 73 - Lanzamiento de zona desde Raspberry Pi, 1

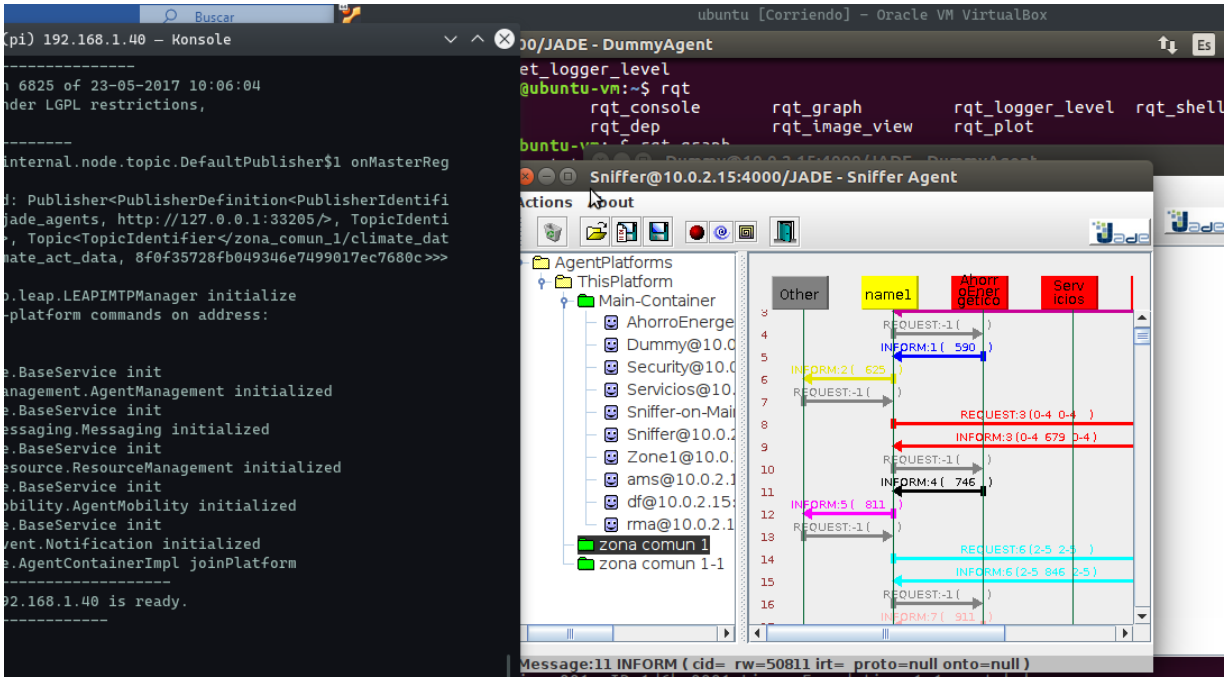


Figura 74 - Lanzamiento de zona desde Raspberry Pi, 2

Con esto, el proceso para el ordenador de sobremesa es exactamente el mismo, y una vez hemos lanzado el sistema al completo, vemos en la figura 75 que el grafo de ROS muestra las dos zonas con sus respectivos nodos:

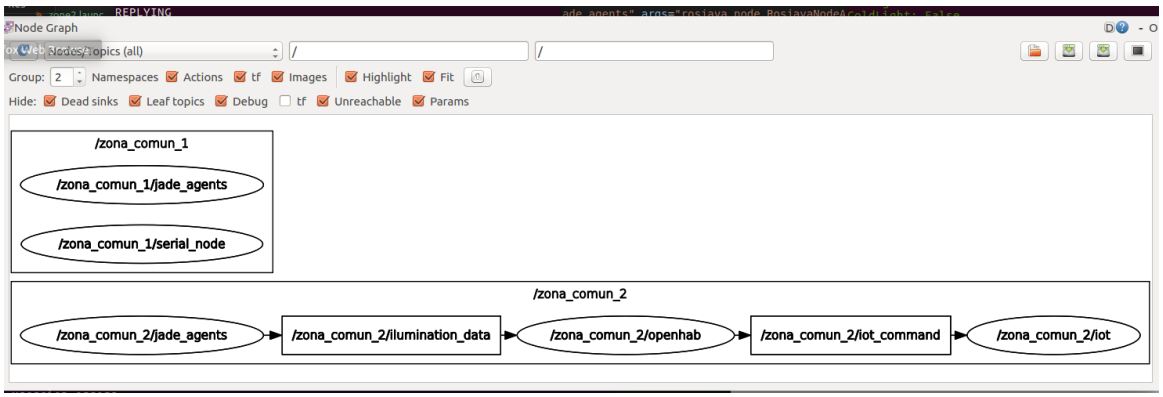


Figura 75 - Grafo ROS de las dos zonas separadas

Estos nodos corresponden con los explicados anteriormente, y demuestran como en efecto se crea el sistema al completo distribuido a través de la red.

4.3.3 Sistema 2: Única zona en Raspberry Pi con dos nodos con elementos IoT

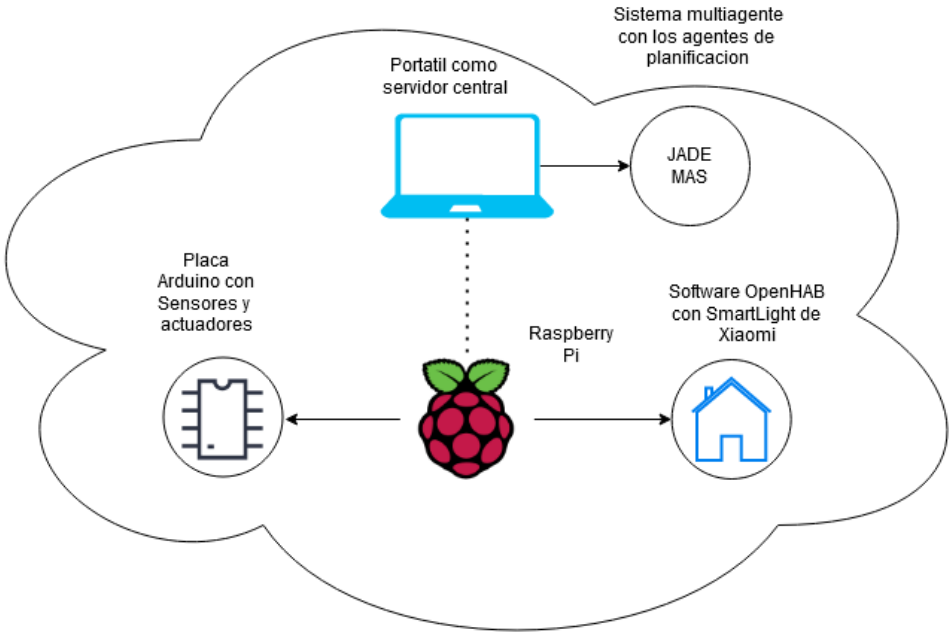


Figura 76 - Entorno de pruebas 2

En este segundo entorno el objetivo no es el de probar el sistema con diversos nodos y zonas, sino el probar como una única zona puede tener varios nodos de ROS que corresponden a diversos dispositivos IoT, y como coordina estos dispositivos para lograr el objetivo.

El archivo roslaunch que corresponde es el mostrado en la figura 77:

```
<launch>
    <group ns="zona_comun_2">
        <include file="$(find iot_bridge)/launch/iot.launch"></include>
        <node pkg="rosjava_pkg_a" type="execute" name="jade_agents" args="rosjava_node.RosjavaNodeArdHAB">
            <param name="agent" value="name1"/>
            <param name="zone_name" value="zona comun 2"/>
        </node>
        <node pkg="ros_iot" type="openhab.py" name="openhab" output="screen">
        </node>
        <node pkg="rosserial_python" type="serial_node.py" name="serial_node" args="/dev/ttyACM0">
        </node>
    </group>
</launch>
```

Figura 77 – Roslaunch de una zona con varios nodos IoT

Si observamos el grafo que genera ROS (figura 78), podemos ver como en esta ocasión todos los nodos existen dentro del mismo espacio de nombres, y son gestionados por un único agente:

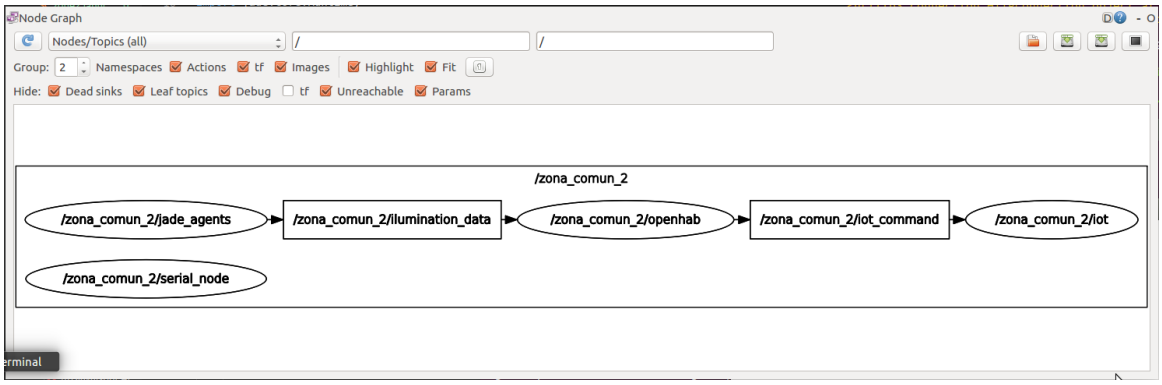


Figura 78 - Grafo de ROS con varios nodos en una zona

Por último, vemos como existen los dos topics y podemos ver por terminal como los mensajes se están transmitiendo por ambos topiocs por un mismo agente, demostrando así que es posible controlar varios tipos de nodos IoT con un único agente.

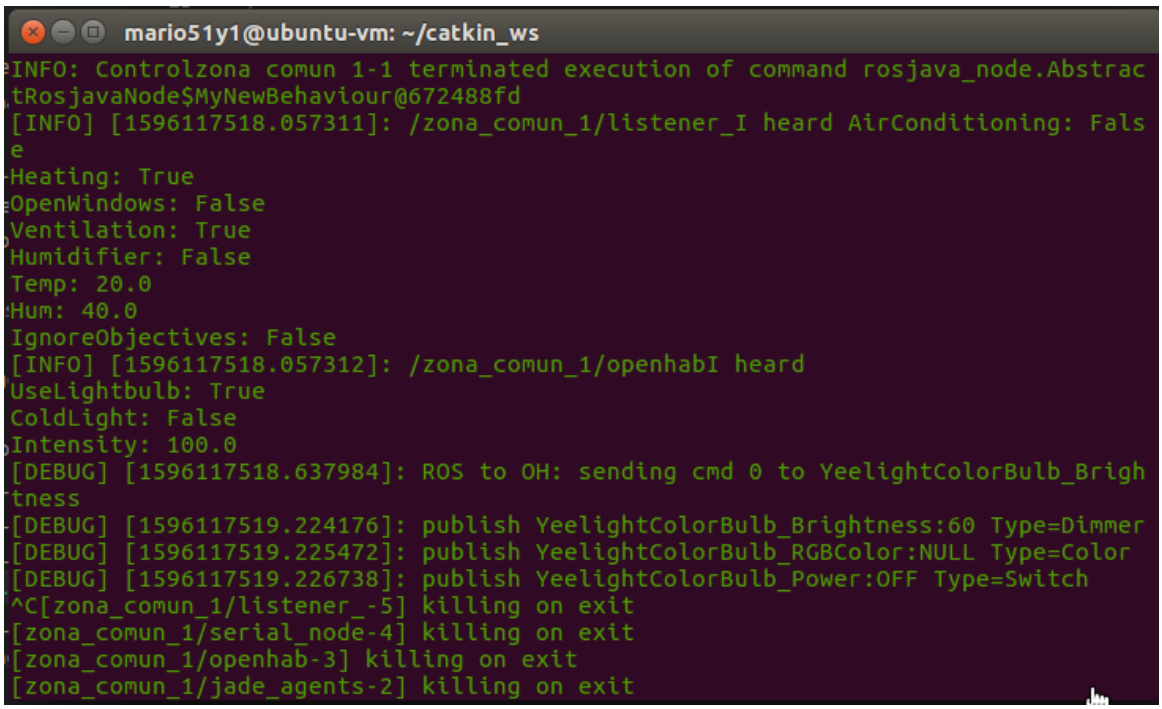


Figura 79 – Único nodo emitiendo mensajes por ambos topics

5. Conclusiones

Con este trabajo hemos cumplido con el objetivo general propuesto de otorgar a la tecnología de sistemas multiagentes una capa a bajo nivel, que en nuestro caso ha sido implementada con elementos IoT.

Esto lo hemos logrado gracias al middleware desarrollado que nos ha permitido integrar dos de las dos tecnologías más empleadas en cada ámbito, y aprovechar de esta manera los puntos fuertes de cada una de ellas. Con esto hemos conseguido cumplir con el primer objetivo específico y hemos logrado crear esa abstracción para emplear las tecnologías de Sistemas Multiagente e IoT.

Por un lado, la tecnología JADE nos permite crear una capa de alto nivel encargada de los objetivos más generales y abstractos, como por ejemplo la planificación de objetivos o la coordinación de los distintos elementos que componen el sistema.

Por otro lado, la capa de bajo nivel desarrollada en ROS nos ha permitido estandarizar el proceso de integración de elementos IoT gracias al sistema que nos ofrece de comunicación y separación en nodos de las distintas partes del sistema. También nos ha permitido añadir esta capa de bajo nivel que se encarga de actuar y percibir del entorno, que facilita la obtención de información de manera rápida y la creación de comportamientos reactivos que añaden valor a la parte de planificación.

Por último, mediante las pruebas que hemos realizado con el hardware especificado, hemos logrado demostrar la viabilidad de esta abstracción empleando entornos con dispositivos reales como son la placa Arduino y la bombilla de Xiaomi.

5.1 Problemas encontrados

El principal problema que se ha encontrado a la hora de desarrollar el proyecto ha consistido en la integración de todas las tecnologías empleadas a través del

middleware desarrollado. Este problema era el principal a resolver en este TFG y ha sido el que más tiempo y esfuerzo ha necesitado.

Pese a que tanto JADE como ROS nos proporcionan herramientas que han facilitado esta tarea, el desarrollo del middleware, así como la automatización del proceso de compilación y despliegue de los sistemas en conjunto, ha conllevado un desarrollo más complicado de lo esperado. Pese a todo, el resultado final se adecúa con los objetivos del proyecto, a pesar de por supuesto ser mejorable en ciertos aspectos que se detallarán en el siguiente punto.

La selección de MESSAGE para la parte de diseño del sistema multiagente ha sido debida a que esta metodología nos proporcionaba las siguientes ventajas: nos proporcionaba una metodología bien definida con los pasos a realizar y nos permitía usar agentes que pudiesen ir creándose y destruyéndose en el sistema en ejecución sin dar muchos problemas al usar el concepto de los roles. Pese a esto, el proceso de análisis y diseño ha sido más largo y difícil de lo esperado ya que, aunque la metodología está bien definida, se han encontrado pocos ejemplos del proceso completo con modelos y gráficos, que ha conllevado en un mayor esfuerzo para resolver dudas y problemas que fueron surgiendo al ir creando los modelos de cada uno de los pasos.

Centrándonos en la implementación y desarrollo del middleware, algunos de los problemas encontrados han sido los siguientes:

- Problemas para encontrar versiones del software compatibles entre sí, que permitiese tener todas las partes funcionales.
- Modificación del proceso de construcción por defecto del paquete de ROS creado con Rosjava para añadir compatibilidad con la herramienta Roslaunch

- Complejidad en el proceso de lanzamiento del sistema completo al tener que tratar con tantas tecnologías
- Necesaria configuración adicional en dispositivos hardware al tratar con diversos sistemas operativos y con máquinas virtuales distribuidas en la red. También ha sido necesario cambiar algunas partes ya implementadas al tratar con los ordenadores adicionales empleados (Raspberry Pi y Ordenador de Sobremesa) para poder lanzar los nodos correctamente.

5.2 Trabajo futuro

Una posible línea de trabajo podría ser la obtención de reglas de comportamiento y decisión de los agentes adaptados a este nuevo entorno. Esto se podría lograr mediante un estudio e implementación de dichas reglas de manera manual o intentar desarrollar un sistema de aprendizaje automático que ayude al agente a decidir sobre su actuación en según qué circunstancias del entorno. Es cierto que ya existe trabajo en el ámbito de los Sistemas Multiagentes sobre este proceso de obtención de reglas de comportamiento, pero al añadir estos elementos IoT en la ecuación sería interesante revisar y ampliar el conocimiento disponible.

Otra posible línea de trabajo sería la de ampliar la integración de elementos IoT con ROS a través de la creación de algún paquete que añadiese funcionalidad a ROS. Esto es debido a que, a pesar de que ROS permite el uso de elementos como placas Arduino, al tratar con otros dispositivos como la bombilla empleada en los sistemas de prueba existe una necesidad de conectar con software de terceros como es OpenHAB. Esto añade una dependencia en un software adicional, así como una necesidad extra de configuración, que podría solventarse con el desarrollo de un paquete para ROS que añadiese directamente esta interfaz común a todos los elementos IoT.

Finalmente, existe otra línea de trabajo posible es la propia mejora del software implementado. Por un lado, el proceso de despliegue y puesta en marcha se podría

intentar simplificar para facilitar su instalación en hardware nuevo. Se podría, por ejemplo, trabajar en empaquetar el código en contenedores Docker y lanzarlos con esta herramienta, eliminando así toda necesidad de instalación en la máquina final de software adicional. Por otra parte, también se puede trabajar en generalizar en mayor medida el middleware para facilitar al desarrollador en la medida de lo posible la implementación de nuevos sistemas.

Anexo

FIGURAS ANALISIS

Goals LVL 1

Simplificar tareas de limpieza

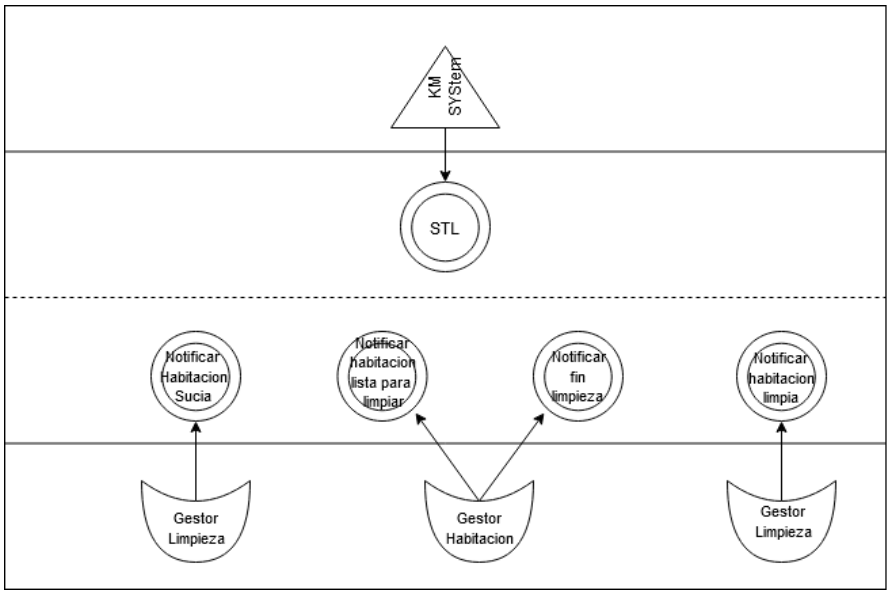


Figura l - Descomposicion goal STL

Mantener estado de seguridad

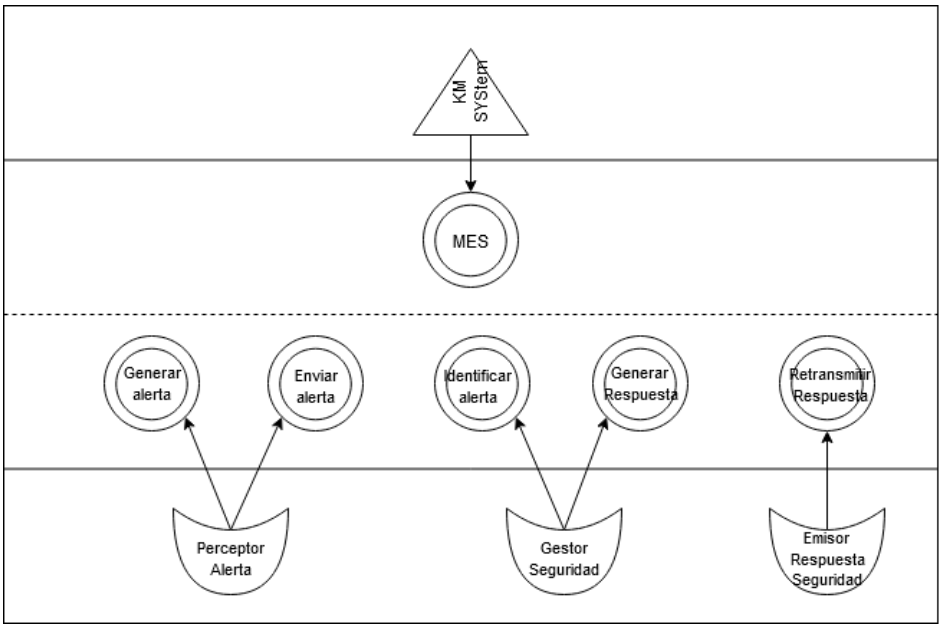


Figura 80 - Descomposicion goal MES

GOALS LVL 2

Goal Lvl 2: Regular Iluminacion

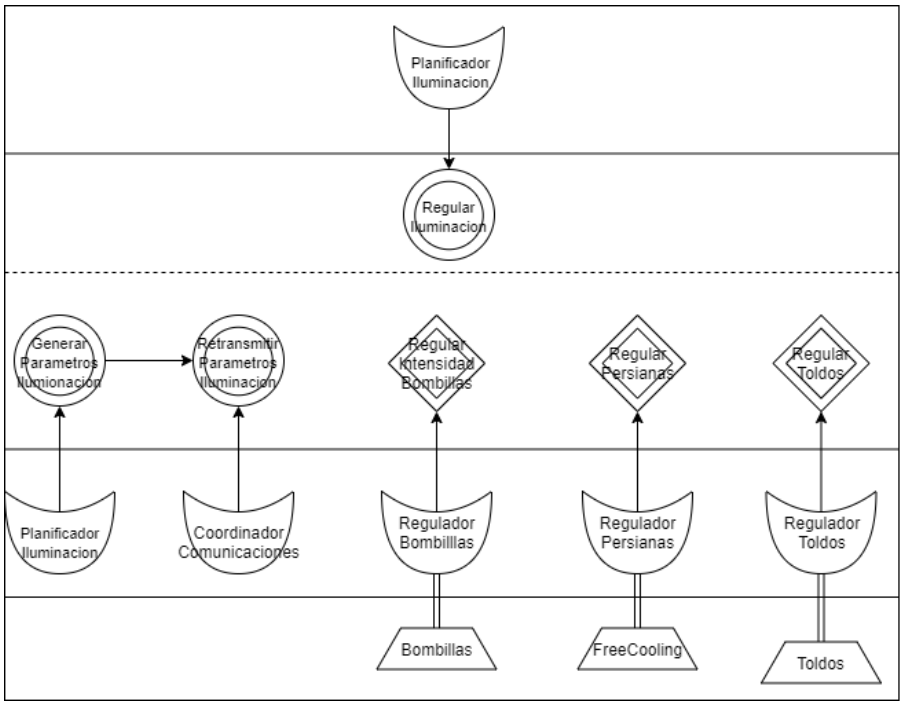


Figura 81 - Goal Regular Iluminacion

Goal Lvl 2: Regular Calderas

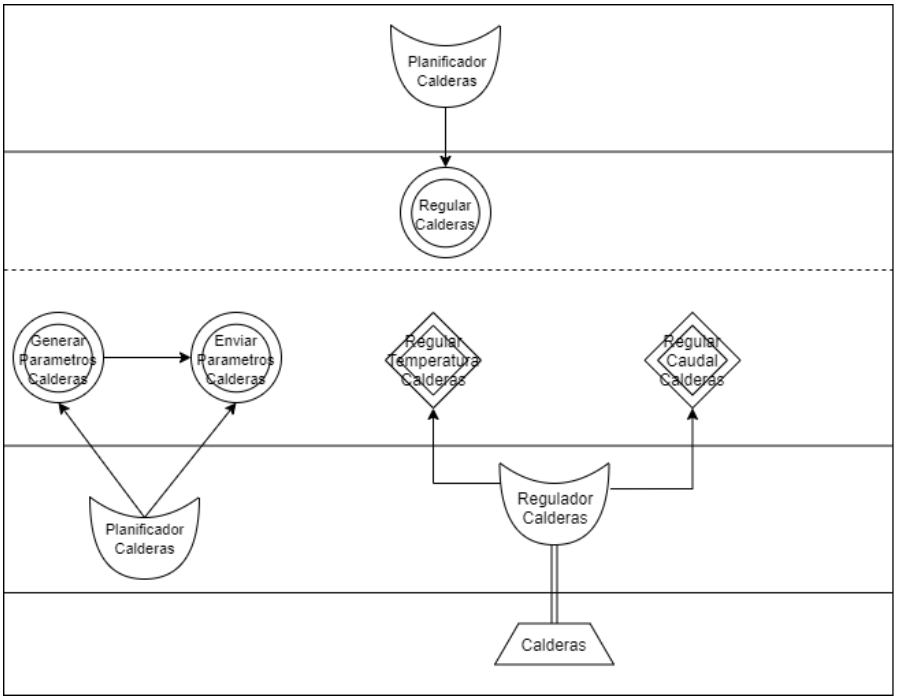


Figura 82 - Goal Regular Calderas

Goal Lvl 2: Regular Paneles Solares

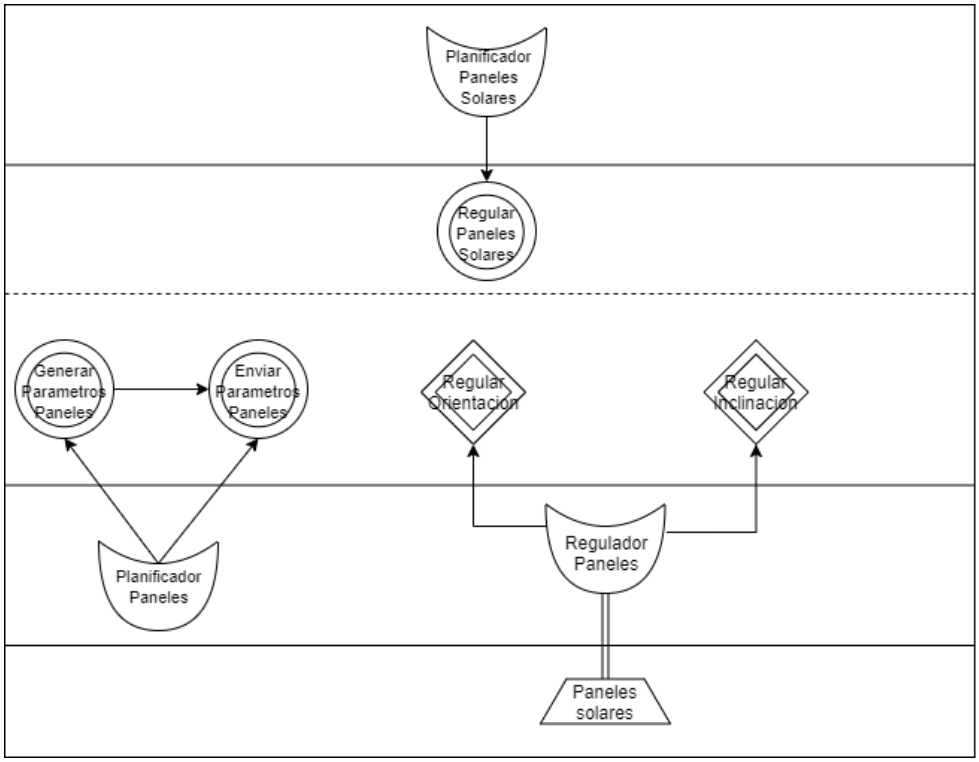


Figura 83 - Regular Paneles Solares

Goal Lvl 2: Regular Ascensores

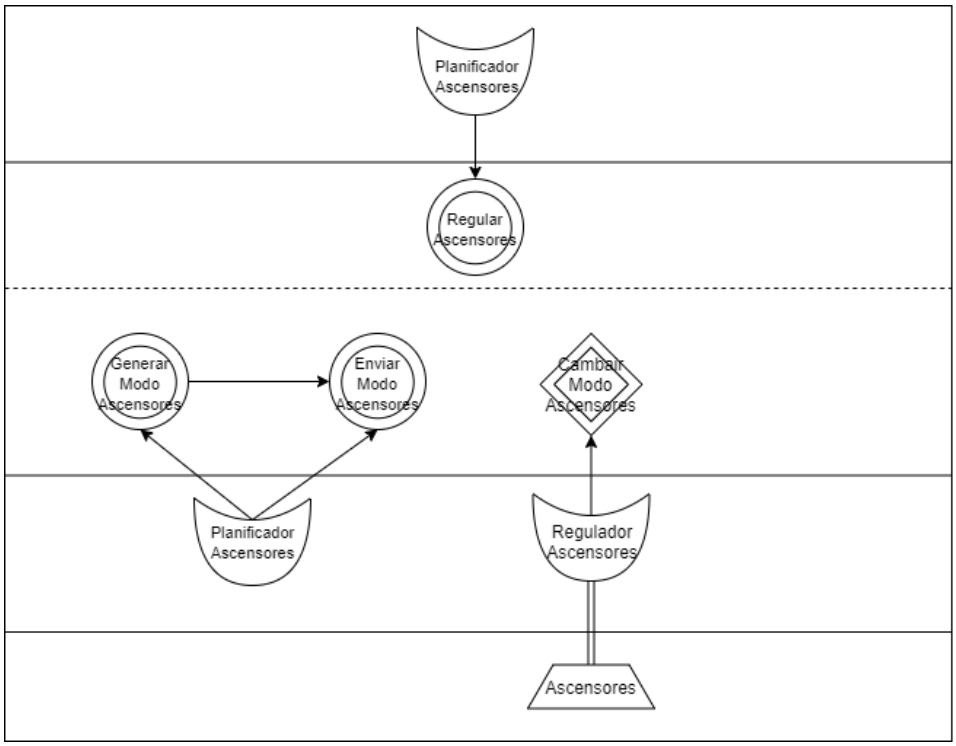
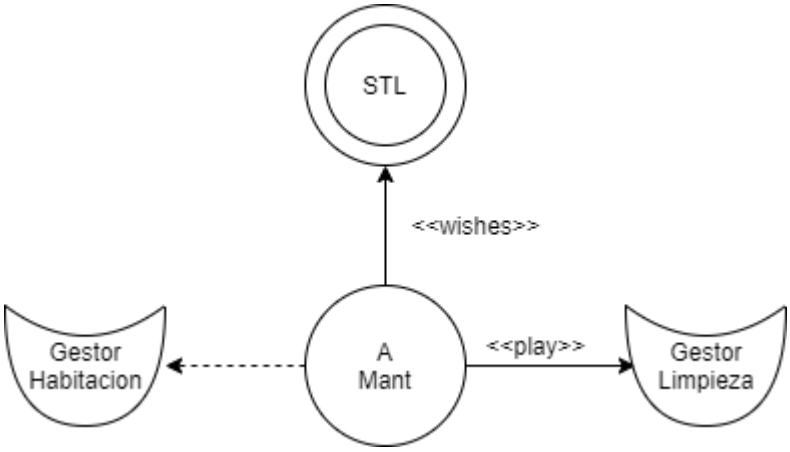


Figura 84 - Goal Regular Ascensores

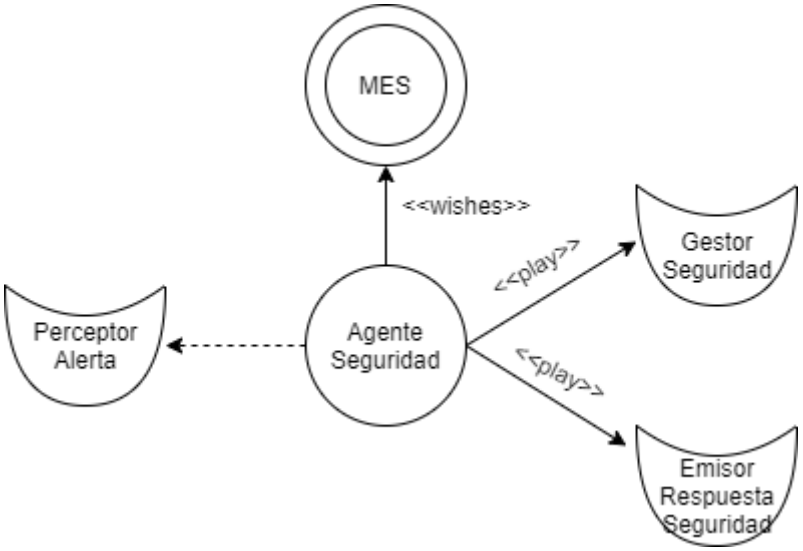
FIGURAS DISEÑO

Agentes

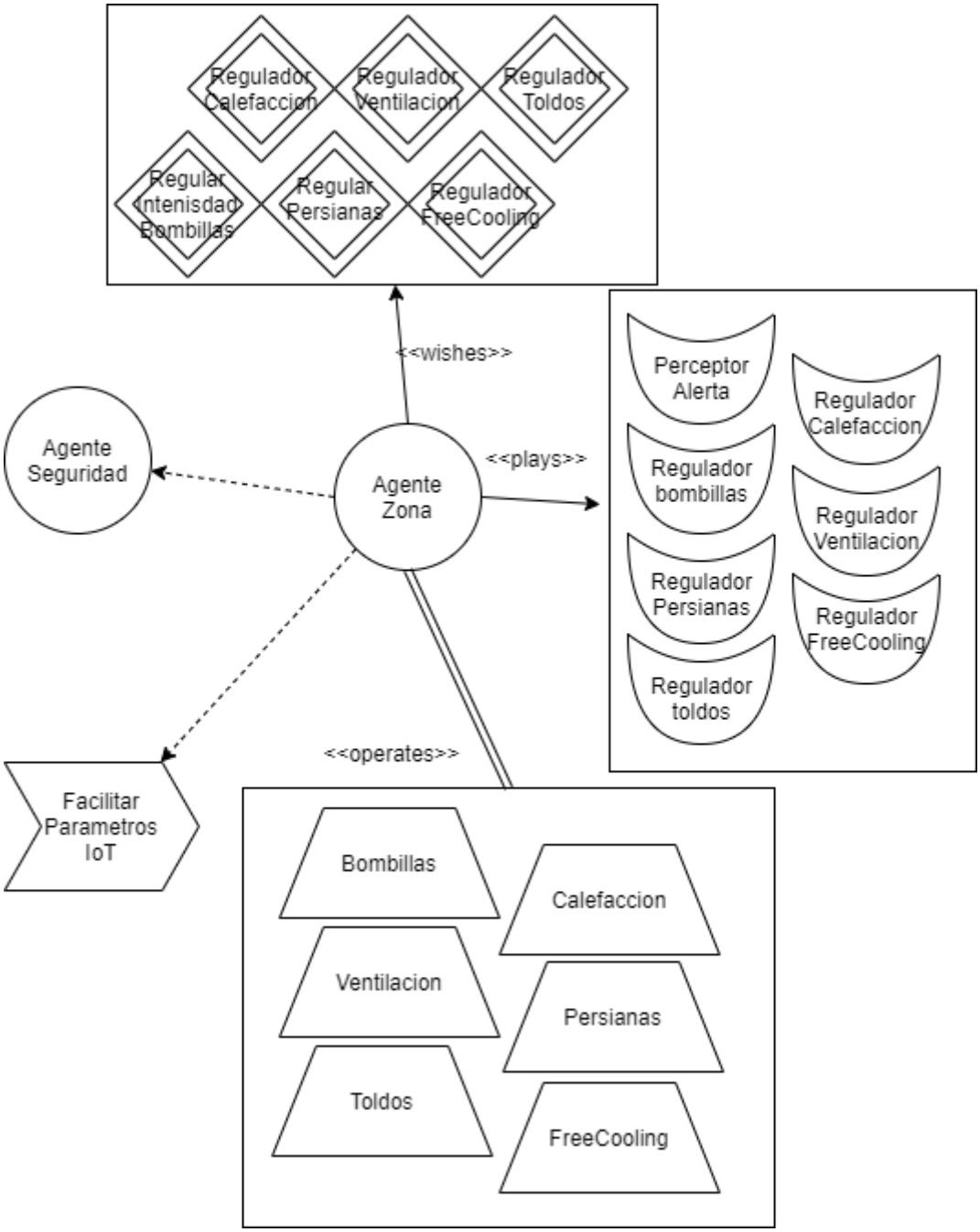
Mantenimiento



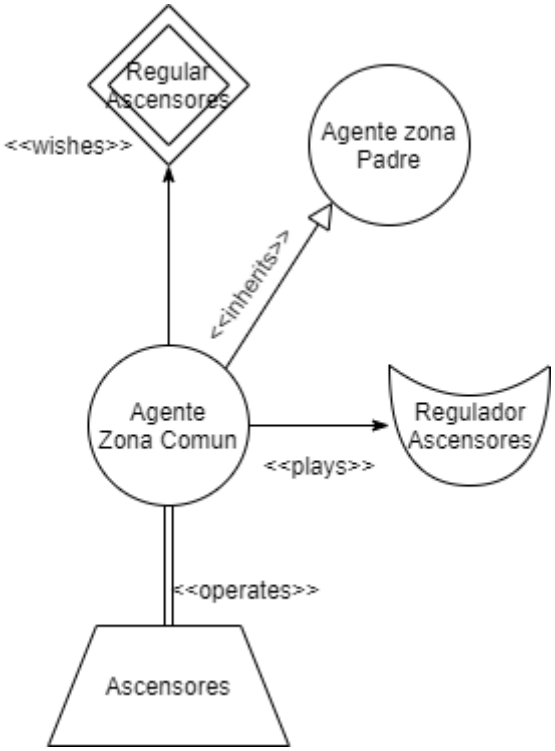
Seguridad



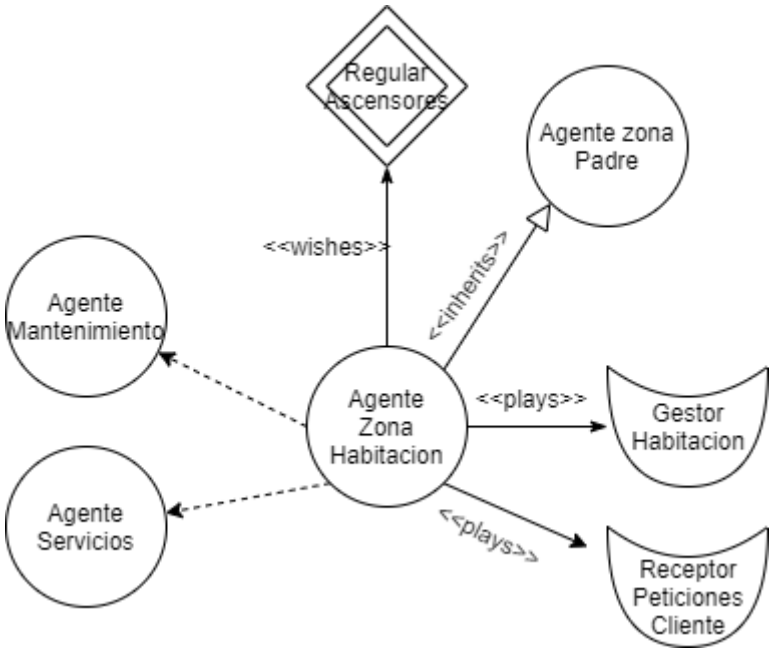
Agente de Control de Zona (padre)



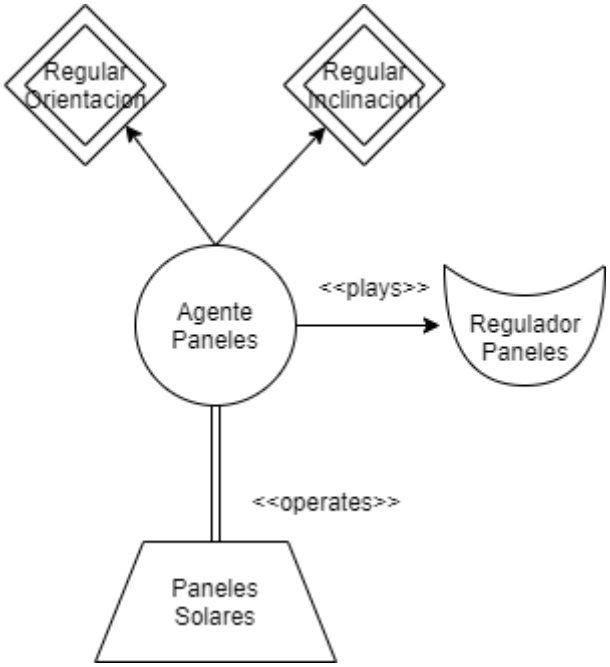
Agente de control de Zona (común)



Agente de control de Zona (habitación)

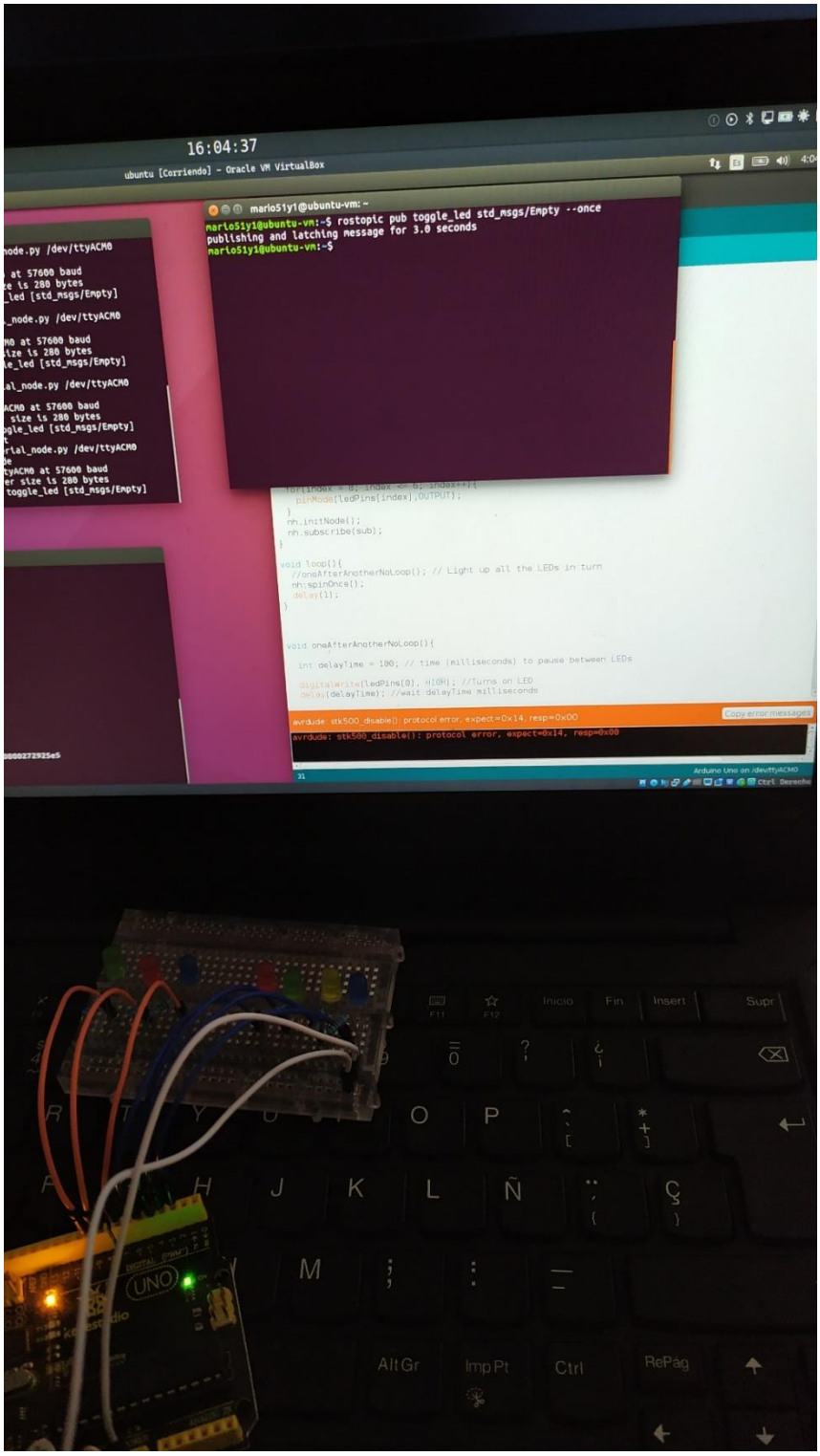


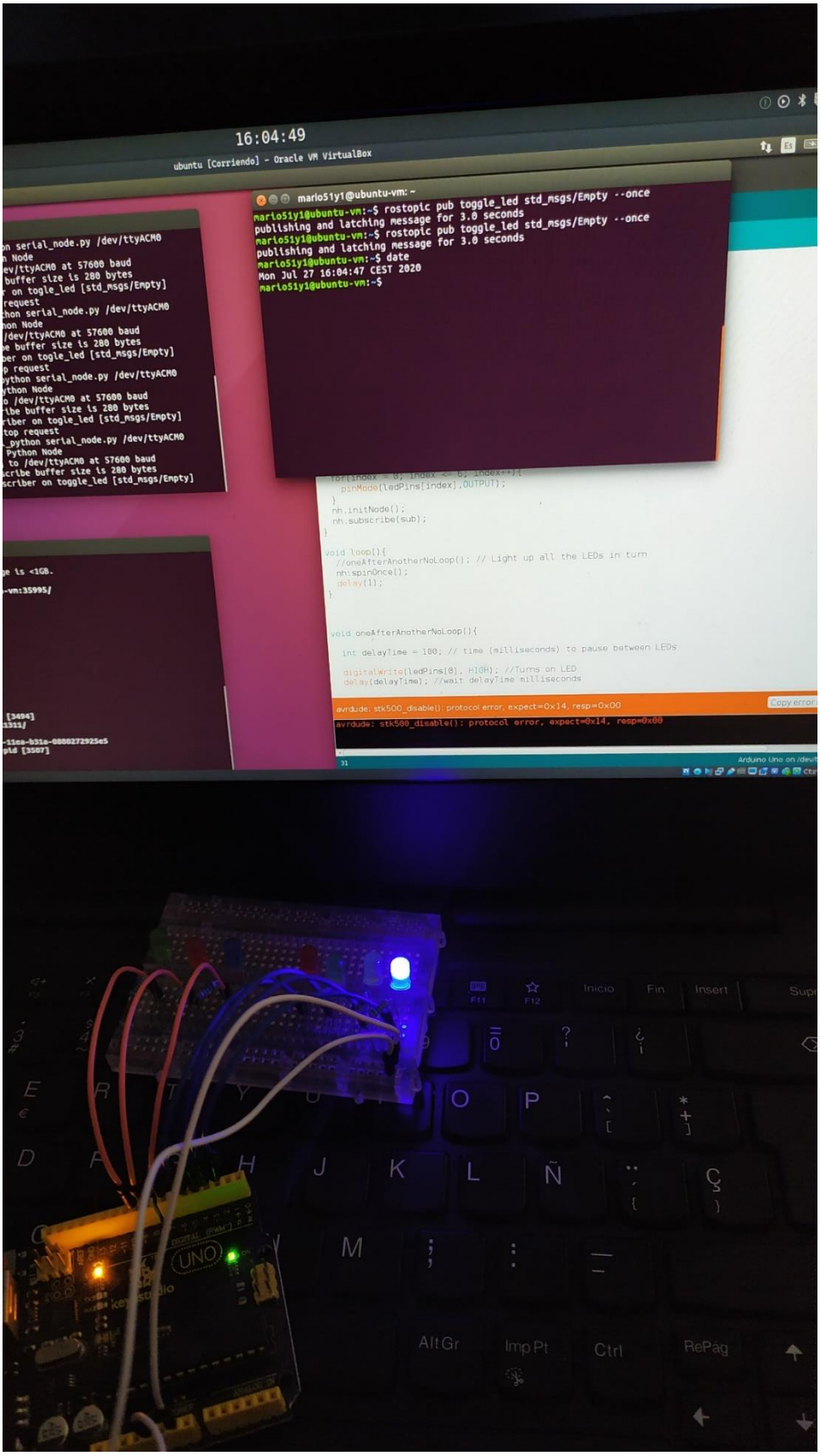
Agente de los paneles

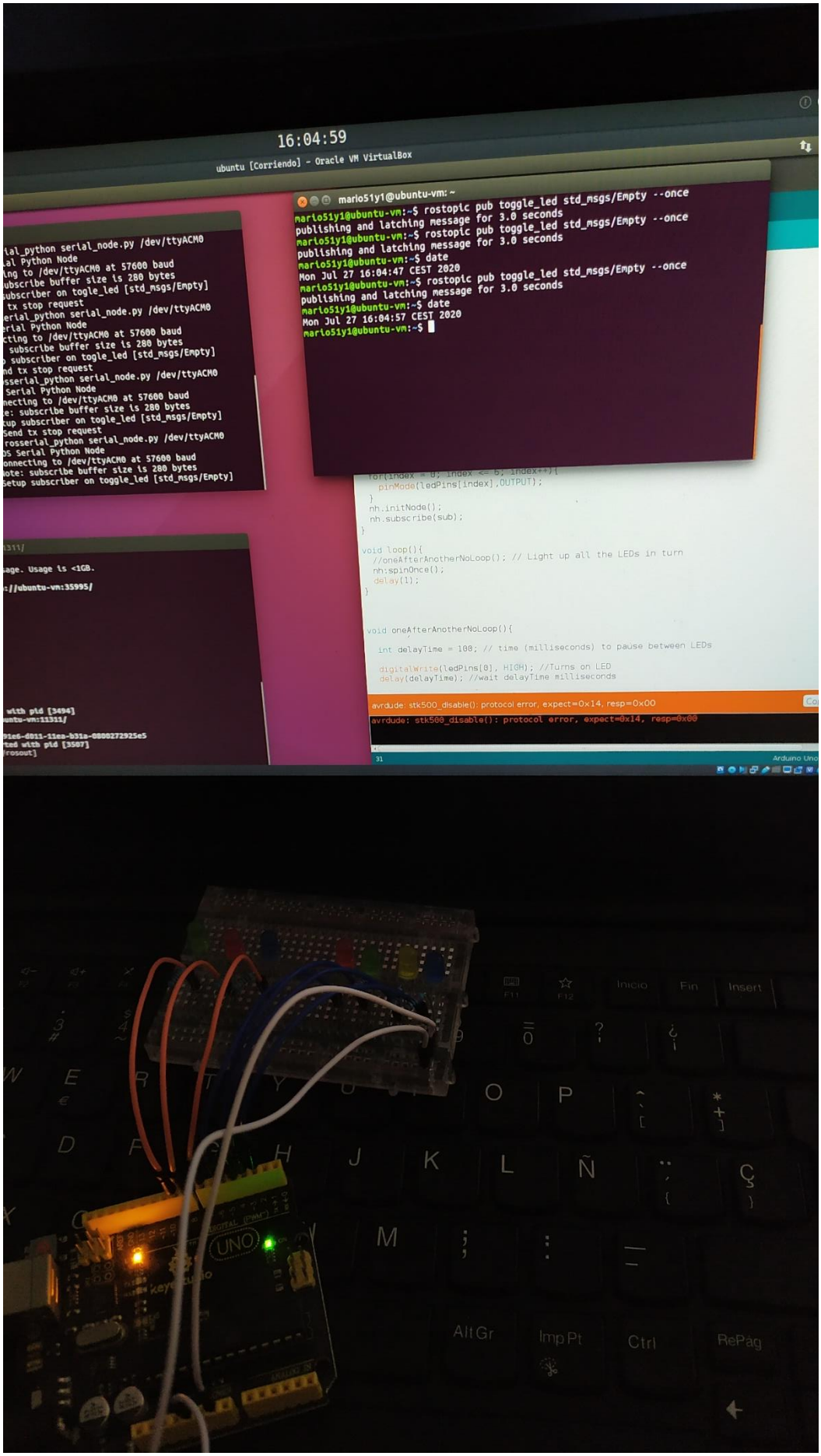


PRUEBAS

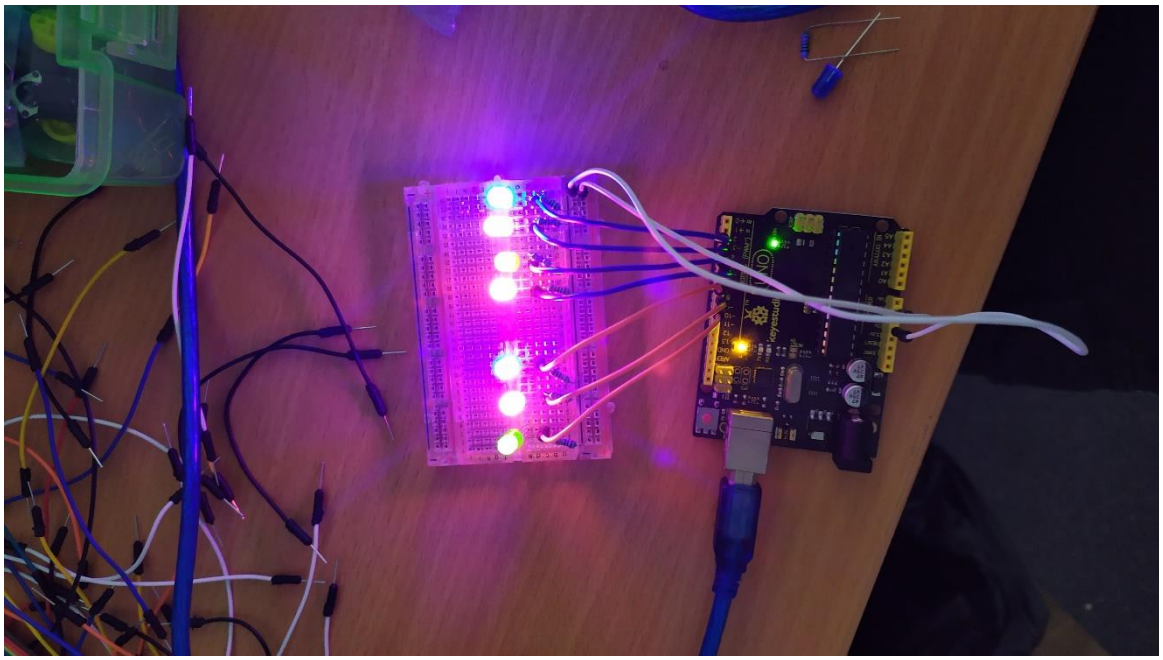
Fotografías funcionamiento de Arduino con ROS




16:04:49
ubuntu [Corriendo] - Oracle VM VirtualBox
mario51y1@ubuntu-vm: ~
mario51y1@ubuntu-vm:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
mario51y1@ubuntu-vm:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
mario51y1@ubuntu-vm:~$ date
Mon Jul 27 16:04:47 CEST 2020
mario51y1@ubuntu-vm:~$
void loop(){
//oneAfterAnotherNoLoop(); // Light up all the LEDs in turn
nh.spinOnce();
delay(1);
}
void oneAfterAnotherNoLoop(){
int delayTime = 100; // time (milliseconds) to pause between LEDs
digitalWrite(ledPins[0], HIGH); //Turns on LED
delay(delayTime); //wait delayTime milliseconds
avrdude: stk500_disable(): protocol error, expect=0x14, resp=0x00
UNO


16:04:59
ubuntu [Corriendo] - Oracle VM VirtualBox
mario51y1@ubuntu-vm: ~
mario51y1@ubuntu-vm:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
mario51y1@ubuntu-vm:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
mario51y1@ubuntu-vm:~$ date
Mon Jul 27 16:04:47 CEST 2020
mario51y1@ubuntu-vm:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
mario51y1@ubuntu-vm:~$ date
Mon Jul 27 16:04:57 CEST 2020
mario51y1@ubuntu-vm:~$
pinMode(ledPins[index],OUTPUT);
nh.initNode();
nh.subscribe(sub);
void loop(){
//oneAfterAnotherNoLoop(); // Light up all the LEDs in turn
nh.spinOnce();
delay(1);
void oneAfterAnotherNoLoop(){
int delayTime = 100; // time (milliseconds) to pause between LEDs
digitalWrite(ledPins[0], HIGH); //Turns on LED
delay(delayTime); //wait delayTime milliseconds
avrdude: stk500_disable(): protocol error, expect=0x14, resp=0x00
Arduino Uno
UNO
Inicio
Fin
Insert
AltGr
Imp Pt
Ctrl
RePag

Prueba de funcionamiento de LEDS



Bibliografía y referencias

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, *8*(3), 203-236. https://doi.org/10.1023/B:AGNT.0000018806.20944.ef

C. Arkin, R., & Balch, T. (1997). *AuRA: Principles and Practice in Review*.

Documentation—ROS Wiki. (s. f.). Recuperado 1 de marzo de 2020, de http://wiki.ros.org/

EDN - Connectivity options for the IoT - Brian Santo. (s. f.). Recuperado 29 de enero de 2020, de https://www.edn.com/connectivity-options-for-the-iot/

Evans, R., Kearney, P., Stark, J., Caire, G., Garijo, F. J., Sanz, J. J. G., Pavon, J., Leal, F., Chainho, P., & Massonet, P. (2001). *Methodology for Agent-Oriented Software Engineering*. 75.

Ferguson, I. A. (s. f.). TouringMachines: An architecture for dynamic, rational, mobile agents. *Mobile Agents*, 219.

FIPA ACL Message Structure Specification. (s. f.). Recuperado 4 de diciembre de 2019, de http://www.fipa.org/specs/fipa00061/SC00061G.html

FIPA Communicative Act Library Specification. (s. f.-a). Recuperado 4 de diciembre de 2019, de http://www.fipa.org/specs/fipa00037/SC00037J.html

FIPA Communicative Act Library Specification. (s. f.-b). Recuperado 13 de mayo de 2020, de http://www.fipa.org/specs/fipa00037/SC00037J.html#_Toc26729689

FIPA Request Interaction Protocol Specification. (s. f.). Recuperado 13 de mayo de 2020, de http://www.fipa.org/specs/fipa00026/SC00026H.html

Garijo, F. J., Gómez-Sanz, J. J., & Massonet, P. (2005). *The MESSAGE Methodology for Agent-Oriented Analysis and Design*. 33.

Gartner Says 8.4 Billion Connected «Things» Will Be in Use in 2017, Up 31 Percent From 2016. (s. f.). Gartner. Recuperado 30 de diciembre de 2019, de https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016

Giorgini, P., Kolp, M., Mylopoulos, J., & Pistore, M. (s. f.). *The Tropos Methodology: An Overview*. 20.

IEEE. (2015). *Towards a definition of the Internet of Things (IoT)*.

Internet of things. (2019). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Internet_of_things&oldid=932924161

Konolige, K., & Myers, K. (1996). *The Saphira Architecture for Autonomous Mobile Robots*. 29.

Memoria iot.pdf. (s. f.).

Mohammadi Zanjireh, M., & Larijani, H. (2015). *A Survey on Centralised and Distributed Clustering Routing Algorithms for WSNs* (Vol. 2015). https://doi.org/10.1109/VTCSpring.2015.7145650

Multi Agent Systems—Yoav.pdf. (s. f.).

Patel, K. K., Patel, S. M., & Scholar, P. (2016). *Internet of Things-IOT: Definition, Characteristics, Architecture, Enabling Technologies, Application & Future Challenges*. 10.

Phelan, D. (s. f.). *Amazon Admits Listening To Alexa Conversations: Why It Matters*. Forbes. Recuperado 31 de diciembre de 2019, de https://www.forbes.com/sites/davidphelan/2019/04/12/amazon-confirms-staff-listen-to-alexa-conversations-heres-all-you-need-to-know/

rosjava_core—Rosjava_core 0.1.6 documentation. (s. f.). Recuperado 9 de julio de 2020, de http://rosjava.github.io/rosjava_core/hydro/index.html

rosjava—ROS Wiki. (s. f.). Recuperado 9 de julio de 2020, de http://wiki.ros.org/rosjava

ROS.org | Powering the world's robots. (s. f.). Recuperado 12 de febrero de 2020, de https://www.ros.org/

Russel, S., & Norvig, P. (2004). *Inteligencia Artificial, un enfoque moderno* (2nd Edition).

Russell, S. J., Norvig, P., Corchado Rodríguez, J. M., & Joyanes Aguilar, L. (2011). *Inteligencia artificial: Un enfoque moderno*. Pearson Educación.

Sánchez Gea, S. (2019). *El framework ROS - Trabajo de Fin de Grado*. Universidad de Alicante.

Sempere Tortosa, M. L. (2013). *Agentes y enjambres artificiales: Modelado de comportamientos para sistemas de enjambre robóticos*. Universidad de Alicante.

Sistema Operativo Robótico. (2020). En *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/w/index.php?title=Sistema_Operativo_Rob%C3%B3tico&oldid=123477835

Sistema Operativo Robótico—Wikipedia, la enciclopedia libre. (s. f.). Recuperado 1 de marzo de 2020, de https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico

Skarmeta, A. G., Pujol, M., & Ramón, R. (s. f.). *Agentes inteligentes—Sistemas multiagentes y aplicaciones*.

Smart Home Statistics [2020]: Growth of IoT Devices. (s. f.). IPropertyManagement.Com. Recuperado 30 de diciembre de 2019, de https://ipropertymanagement.com/research/iot-statistics

Soriano Vigueras, Á. (2017). *INTEGRACIÓN DE SISTEMAS MULTI-AGENTE EN PLATAFORMAS EMBEBIDAS HETEROGÉNEAS CON RECURSOS LIMITADOS PARA TAREAS DE LOCALIZACIÓN Y COORDINACIÓN EN DETECCIÓN Y EVASIÓN DE COLISIONES EN ROBÓTICA MÓVIL.* [Universitat Politècnica de València]. https://doi.org/10.4995/Thesis/10251/86174

State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating. (s. f.). Recuperado 30 de diciembre de 2019, de https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

The JADE-Agentcities Tutorial. (s. f.). Recuperado 9 de julio de 2020, de https://www.cs.upc.edu/~luigi/frame0_files/JADE-tutorial/tutorial.html

Tutorials & Guides | Jade Site. (s. f.). Recuperado 23 de mayo de 2020, de https://jade.tilab.com/documentation/tutorials-guides/

Weiss, G. (1999). *Multiagent Systems—A modern approach to Distributed Artificial Intelligence*.

What is IoT architecture? (s. f.). Recuperado 12 de febrero de 2020, de https://www.avsystem.com/blog/what-is-iot-architecture/

Wooldridge, M. (2009). *An introduction to MultiAgent Systems* (Second).

Wooldridge, M. J., Weiss, G., & Ciancarini, P. (Eds.). (2002). *Agent-oriented software engineering II: Second international workshop, AOSE 2001, Montreal, Canada, May 29, 2001: revised papers and invited contributions*. Springer.

Wooldridge, M., & R. Jennings, N. (1995). *Intelligent agents: Theory and practice.*

Wooldridge, M., R. Jennings, N., & Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*.