UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

CENTRO UNIVERSITARIO UAEM VALLE DE MÉXICO

"ARQUITECTURA PARA SISTEMAS MULTIAGENTE EN APLICACIONES DISTRIBUIDAS USANDO TECNOLOGÍA JAVA"

TESIS

Que para obtener el Título de

INGENIERO EN SISTEMAS Y COMUNICACIONES

P r e s e n t a

C. Angeles Eloina Méndez García

Asesor: Dr. en C. Héctor Rafael Orozco Aguirre

Atizapán de Zaragoza, Edo. de Méx. Febrero 2015

CENTRO UNIVERSITARIO UAEM VALLE DE MÉXICO

UAEM | Universidad Autónoma del Estado de México

REGISTRO DE TEMA

Centro Universitario UAEM Valle de México

CUUAEMVM/SA/TITULACIÓN/329/14

Atizapán de Zaragoza, México, 17 de junio de 2014.

C. MÉNDEZ GARCÍA ANGELES ELOINA
Egresada de Ingeniería en Sistemas y Comunicaciones
PRESENTE

Por la presente, me permito comunicarle que el tema de su investigación por la modalidad de **Tesis**, bajo el título: **"ARQUITECTURA PARA SISTEMAS MULTIAGENTE EN APLICACIONES DISTRIBUIDAS USANDO TECNOLOGÍA JAVA"**, ha sido registrado en esta Subdirección Académica, y que el asesor que Usted propuso Dr. en C. Héctor Rafael Orozco Aguirre, también será notificado(a) por este medio para que se encargue de guiar su investigación.

Así mismo, le recuerdo que tiene usted un año a partir de esta fecha para presentar su trabajo final liberado por su asesor y revisores que posteriormente se le asignarán y que durante este período deberá presentar un informe cada dos meses, con el Visto Bueno de su Asesor, sobre el avance de su investigación en la oficina de Titulación de este Centro Universitario.

El trabajo de **Tesis** queda bajo la responsabilidad del egresado tanto en autoría como en su contenido, el cual deberá tener el nivel que se exige para la obtención de un Título Profesional.

ATENTAMENTE
PATRIA, CIENCIA Y TRABAJO
"2014, 70 Aniversario de la Autonomía ICLA-UAEM"

Centro Universitario
UAEM Valle de México
Subdirección Académica

LIC. PATRICIA ROJAS REYES
SUBDIRECTORA ACADÉMICA

c.c.p. Dr. en C. Héctor Rafael Orozco Aguirre
Expediente

PRR/GGB/gra*

Atizapán de Zaragoza, Estado de México a 4 de Julio del 2014

LIC. PATRICIA ROJAS REYES
SUBDIRECTORA ACADÉMICA
CENTRO UNIVERSITARIO UAEM VALLE DE MÉXICO
P R E S E N T E

Por la presente le informo que el pasante **Angeles Eloina Méndez García**, de la carrera de **Ingeniería en Sistemas y Comunicaciones**, con No. de cuenta **0824111**, presenta el trabajo de **TESIS: ARQUITECTURA PARA SISTEMAS MULTIAGENTE EN APLICACIONES DISTRIBUIDAS USANDO TECNOLOGÍA JAVA** , mismo que conforme a la Legislación Universitaria, ha sido **aprobado** por el que suscribe para los fines propios de titulación del interesado.

Sin más por el momento, reciba un cordial saludo.

ATENTAMENTE

Centro Universitario UAEM Valle de México
RECIBIDO
04 JUL 2014
TITULACIÓN

Dr. en C. Héctor Rafael Orozco Aguirre
ASESOR

TELS. 55-37-15-65-64
CORREO: rafilla.orozco@gmail.com

Atizapán de Zaragoza, Estado de México a 25 de Noviembre del 2014

LIC. PATRICIA ROJAS REYES
SUBDIRECTORA ACADÉMICA
CENTRO UNIVERSITARIO UAEM VALLE DE MÉXICO
P R E S E N T E

Por la presente le informamos que el pasante **Angeles Eloina Méndez García**, de la carrera de **Ingeniería en Sistemas y Comunicaciones**, con No. de cuenta **0824111**, presenta el trabajo de **TESIS: ARQUITECTURA PARA SISTEMAS MULTIAGENTE EN APLICACIONES DISTRIBUIDAS USANDO TECNOLOGÍA JAVA**, mismo que conforme a la Legislación Universitaria y a las observaciones dictaminadas en el preexamen, ha sido **aprobado** por los que suscribimos para los fines propios de titulación del interesado.

Sin más por el momento, reciba un cordial saludo.

ATENTAMENTE

Dr. En C. Victor Manuel Landassuri Moreno
REVISOR

ATENTAMENTE

Dr. En C. Oscar Herrera Alcantara
REVISOR

Centro Universitario UAEM Valle de México
RECIBIDO
20 NOV 2014
TITULACIÓN

UAEM | Universidad Autónoma del Estado de México

AUTORIZACIÓN DE IMPRESIÓN

Centro Universitario UAEM Valle de México

CUVM/SA/TITULACIÓN/694/14

Atizapán de Zaragoza, México, a 28 de noviembre de 2014.

C. MÉNDEZ GARCÍA ANGELES ELOINA
Egresada de Ingeniería en Sistemas y Comunicaciones
P R E S E N T E

Me permito comunicarle que se autoriza la impresión de su trabajo de titulación por la modalidad de Tesis, denominado **"ARQUITECTURA PARA SISTEMAS MULTIAGENTE EN APLICACIONES DISTRIBUIDAS USANDO TECNOLOGÍA JAVA"**, para obtener el título de Ingeniería en Sistemas y Comunicaciones, en virtud de que cuenta con los votos aprobatorios del Asesor y los Revisores asignados para este efecto, en apego a los lineamientos establecidos para la Evaluación Profesional.

Nota: No omito comentar que la impresión de tus empastados deberá coincidir con el título que en este documento se autorizó en términos de mayúsculas, minúsculas, acentos, comillas, paréntesis, etc.

A T E N T A M E N T E
PATRIA, CIENCIA Y TRABAJO
"2014, 70 Aniversario de la Autonomía ICLA-UAEM"

Centro Universitario UAEM Valle de México Subdirección Académica 2013-2017

LIC. PATRICIA ROJAS REYES
SUBDIRECTORA ACADÉMICA

c.c.p. Expediente
PRR/GGB/gra*

Centro Universitario UAEM, Valle de México
Blvd. Universitario s/n Predio San Javier Atizapán de Zaragoza, MéxicoTeléfono: (01 55) 58 27 03 61, Fax: 58 27 07 03
cuvm@uaemex.mx

DEDICATORIA

A Dios, por darme paciencia en los momentos de desesperación, valor para enfrentar los problemas que surgieron a lo largo de este camino, sabiduría para discernir en los momentos de tomar decisiones difíciles y su amor; gracias a ello, pude llegar a la meta.

A mi mamá, por darme la vida, confiar en mí y darme la oportunidad de experimentar momentos felices; como lo es concluir esta tesis. Por haber estado al pie del cañón cuando enfermé y estuve a punto de abandonar mis sueños, por ser mi amiga escucharme y aconsejarme, siempre encontraste la manera de sacar adelante a tu familia, eres mi ángel, te amo con toda mi alma.

A mi Papá, por darme la vida, ser mi ejemplo, apoyarme, aconsejarme, impulsarme y estar a mi lado a lo largo del camino para lograr la meta que me propuse; sin duda alguna, sin ti no lo habría logrado ya que siempre tuve tu apoyo incondicional preocupándote por dame lo mejor. Esta tesis es para ti con todo mi amor, es el primer retoño de todo lo que me has dado.

A mis hermanos Martha, Nelly y Jesús por ser mi felicidad, por hacerme reír con sus ocurrencias cuando me sentía frustrada, por enseñarme a ser una persona humilde y decidida a pesar de ser más pequeños que yo, alentándome a seguir adelante sin importar las circunstancias, los amo.

A mi novio Daniel, por el amor incondicional que me has dado desde el inicio de mi carrera, por tu apoyo no solo moral, por la paciencia, comprensión y consejos que me diste en los momentos de desesperación alentándome a seguir cuando sentía que era momento de rendirme, por hacerme reír cuando me sentía frustrada pero sobre todo por estar siempre a mi lado, te amo.

AGRADECIMIENTO

A mi institución que me ha brindado los cimientos para mi formación profesional, gracias a lo cual pude llegar al punto en el que me encuentro.

A mi asesor el Dr. Héctor Orozco por hacer posible la conclusión de esta tesis y ayudarme durante estos dos años en la investigación, influyendo con su inteligencia y sus conocimientos para que yo sea una mejor persona y esté preparada para los retos de la vida que iniciarán de ahora en adelante.

Al Dr. Felipe Castro por su apoyo e interés constante durante mi proceso de investigación, gracias por motivarme a seguir adelante, es una persona de un gran criterio y valor humano.

A los sinodales el Dr. Victor Landassuri y el Dr. Oscar Herrera quienes estudiaron, revisaron y aprobaron mi tesis, apoyándome para concluirla.

Resumen

Con el paso de los años, las necesidades de los humanos así como los problemas que surgen en las sociedades; conducen a las disciplinas científicas a generar métodos de simulación como apoyo en casos en los que si el humano interviniera lanzaría resultados poco exactos, desastres en su entorno e incluso daños a las personas mismas. De modo que los agentes inteligentes aquí juegan un papel importante, por lo tanto, se deben tener bases sólidas como son: el tipo de agente, sus características y reglas para el intercambio de mensajes al momento de realizar la interacción.

En este trabajo de tesis, se propone una arquitectura para aplicaciones de interoperabilidad entre agentes heterogéneos (es decir de diferentes tipos) programados en Java, para ello se emplearon sólo los siguientes tipos de agentes, tomados como los más sobresalientes: cognitivo, reactivo, híbrido, basado en metas, basado en su utilidad, basado en modelos y tipo Emoción-Creencia-Deseo-Intención (EBDI, por sus siglas en inglés Emotion-Belief-Desire-Intention); con los que se formó una jerarquía, agrupando los agentes en base a las características que los integran, posteriormente se programó cada uno de los agentes con el fin de que la arquitectura sea multiplataforma.

Además, para que la creación de los agentes así como la interoperabilidad entre ellos se efectúe satisfactoriamente se emplearon los estándares que determina FIPA (por sus siglas en inglés: The Foundation for Intelligent Physical Agents) ésta proporciona un ciclo de vida para los agentes, para lograr esto se utilizaron hilos de control empleados en una máquina de estados y de esta manera tener un control sobre un agente desde el momento de su creación hasta su destrucción; también la misma especifica la estructura de los mensajes que se intercambian en el momento

en que los agentes se comuniquen; y finalmente, especificaciones para envío y recepción de mensajes.

Finalmente, con estos estándares también se consultó el W3C (por sus siglas en inglés: World Wide Web Consortium) donde también especifica las relaciones entre los agentes, así como la estructura de los mensajes pero a nivel web con el fin de garantizar la interoperabilidad de los mismos.

Abstract

Through the years, the human needs and the problems that take place in society; drive the scientific disciplines to create simulation methods as a support in cases where the human intervention could generate less accurate results, disasters in the environment and even damage to other people. In this way, the intelligent agents play here an important role. Therefore, solid bases of knowledge are needed, such as: the type of agent, its characteristics and rules for the sending-receiving of messages at interaction time.

In this thesis, a Java programmed architecture for interoperability applications between heterogeneous agents is proposed, that is to say, different types of them. The following types of agents were taken as the most representative ones: cognitive agent, reactive agent, hybrid agent, based on goals agent, based on utility agent, based on models agent and EBDI agent, for its acronym: Emotion-Belief-Desire-Intention. All of them where grouped to form a hierarchy based on their features; subsequently each type of agent was implemented in order to have a multiplatform architecture.

In addition, in order to perform the creation and interoperability of the agents in a satisfactorily way, the standards determined by FIPA were used, for its acronym: The Foundation for Intelligent Physical Agents. This foundation provides a life cycle for agents. To achieve this, control threads were employed into a state machine to have control over an agent since its creation time until its destruction. Also this foundation specifies the structure of the messages that can be exchanged when agents are communicating each other and rules for the sending-receiving of them.

Finally, the W3C was also used, for its acronym: World Wide Web Consortium, which specifies the relationships between agents and the structure of the messages at a web level in order to ensure a good interoperability for them.

Contenido

Índice de Tablas

Índice de Figuras

Capítulo 1. Introducción

Actualmente, existen distintos paradigmas de simulación, en los cuales no siempre es posible aproximarse a los resultados esperados, que se requieren sean lo más parecidos a los que hay en la vida real al momento de realizar la simulación; ya que en dichas herramientas conforme aumenta el grado de complejidad de los ambientes se vuelve menos práctico su uso por la gran cantidad de operaciones matemáticas que son realizadas, por lo tanto, en esta cuestión no son muy eficientes ya que tardan mucho tiempo en entregar los resultados y no siempre son los requeridos. Estas son algunas de las razones por las cuales es preferible emplear otro tipo de herramienta, en este trabajo, se dan las razones por las cuales se defiende y es preferible hacer uso de los sistemas multiagente para simular casos de la vida cotidiana ya que arrojan mejores resultados en un menor tiempo y con más exactitud.

1.1 Antecedentes

Con el paso de los años, nuestro planeta se vuelve un entorno cada vez más complejo de entender y en el cual poder subsistir, debido a la amplia gama de situaciones que surgen a causa de las interacciones diarias entre los distintos miembros de las sociedades u organizaciones, dentro de ellas mismas. En efecto, la diversidad y el grado de los problemas que afectan al mundo es algo que debe preocupar a todos independientemente de la posición social, sexo, edad o religión; por lo tanto, hallar una solución a estos involucra diversas áreas de todos los campos de investigación.

Derivado de lo anterior, los investigadores en Inteligencia Artificial se han dado a la tarea de generar herramientas **[1, 2, 3]** para modelar y simular de una manera natural y detallada cada uno de los procesos e interacciones que se llevan a cabo

dentro de una sociedad u organización, ofreciendo así formas de encontrar soluciones a los problemas de la vida diaria.

De este modo, se puede obtener información valiosa ya que gracias a la simulación es posible observar y analizar, los pros y contras de una actividad determinada dando la oportunidad de generar alternativas para obtener efectos positivos y disminuir o eliminar todo efecto negativo **[4]**. Con ello, se pueden ofrecer soluciones pertinentes a distintos problemas como lo son los presentes en el mundo de los negocios, el mal flujo de personas dentro de una estación del metro, los efectos de un flujo vehicular lento en las ciudades, las malas estrategias de distribución de servicios, entre otras aplicaciones.

En base a diversos estudios, es posible afirmar que prácticamente todas las situaciones que existen en el planeta están relacionadas con el comportamiento de la humanidad, ya que por naturaleza los seres humanos realizan todo tipo de acciones influenciadas por su conducta que pueden generar un efecto positivo o negativo para ellos y su entorno. En consecuencia, debe ser necesario el poder encontrar y ofrecer diversas formas de entender a la sociedad humana para comenzar a proponer distintas estrategias de solución a los problemas sociales y ambientales de mayor prioridad. Una alternativa se halla presente en el uso de herramientas computacionales, donde es posible recurrir a técnicas de Inteligencia Artificial aplicada para tales efectos. Entre estas técnicas, el uso de sistemas multiagente ofrece de momento la mejor herramienta para modelar y simular toda situación presente en las sociedades y sus organizaciones.

No sólo en Inteligencia Artificial son importantes los agentes; sino también, para muchas otras áreas por ejemplo las sociales, políticas y económicas en las que se hace uso de agentes para cuestiones de simulación, predicción, optimización, entre muchos otros enfoques en los que se le puede dar uso a los mismos. Sin embargo, es importante seguir realizando investigación procurando agregar mejoras y nuevos

esquemas o paradigmas de simulación basados en agentes para obtener resultados mucho más confiables en los modelos de simulación.

En estos sistemas, un agente por lo general es usado para modelar a un ser humano y así poder simular su comportamiento para tener un mejor entendimiento de toda conducta que pueda exponer en el entorno donde se halle presente. Sin embargo, el modelado y simulación del comportamiento humano bajo estos sistemas no es una tarea sencilla y la mayor parte de las veces la complejidad de los mismos resulta en extremo mayor, debido a que es necesario conocer todos los elementos que pudieran estar presentes en una situación de la vida real para poder plasmarlos de manera artificial en un modelo computacional. De modo que, se pueden obtener grandes beneficios mediante el uso de estos sistemas como una herramienta confiable para representar en escalas diversas al mundo real, entre ellas resaltan situaciones donde se desee prevenir, predecir o mejorar, o bien, aquellas que si son efectuadas con experimentos en el campo real pueden resultar caras o incluso provocar pérdidas humanas como en el caso de batallas o ataque a grupos delictivos.

Ahora bien, para poder generar un sistema multiagente es de gran importancia elegir una arquitectura que se adecúe a las necesidades y requerimientos de las situaciones a ser modeladas y simuladas. Sin duda, en la actualidad existen diversas arquitecturas que ya tienen predeterminados los tipos de agentes que se pueden implementar, como bien lo son: arquitecturas deliberativas, arquitecturas reactivas, arquitecturas híbridas, entre otras; sin embargo, ninguna de ellas permite manejar a la vez la diversa gama de agentes que existen en la literatura y sólo se enfocan a uno o pocos de ellos. Así, las arquitecturas multiagente actuales limitan lo que se puede ver y hacer en una simulación. Esto sin tomar en cuenta que además la interoperabilidad en la mayor parte de ellas es sólo entre agentes del mismo tipo, lo cual limita o imposibilita el contar con interacciones en donde se necesite de distintos tipos interactuando a la vez, incluso el que los mismos se hallen en distintos ambientes o entornos.

Por otra parte, desde el momento en que se necesita de un sistema simulador para representar situaciones de la vida real **[5]**; como primer punto, se piensa en las características que tendrá la simulación en tal sistema; es decir, las características esenciales de los agentes, como son: sus metas y objetivos, junto con su conocimiento y sus habilidades cognitivas. Enseguida, se debe hacer un análisis minucioso sobre el entorno en el cual los agentes van a interactuar; ya sea que esté en constante cambio, al que se le va a conocer como dinámico o un ambiente que no sufrirá ningún cambio, al que se le llama estático. Posteriormente, se debe tomar en cuenta que tan complejas serán las interacciones y acciones de los agentes, así como que tantos procesos cognitivos van a intervenir en su toma de decisiones. Como paso final, se tiene que hacer la elección del software o tecnologías que logren adaptarse al cien por ciento o de la manera más aproximada a todo lo que anteriormente fue comentado; para ello, existen diversas plataformas, no obstante, ninguna de ellas cubre algunos aspectos que fueron mencionados anteriormente, (ver tabla 6.1) es decir, no se ajustan de manera satisfactoria a las expectativas necesarias para ofrecer una alternativa confiable como herramientas de simulación.

1.2 Definición del problema

Para poder modelar y simular un caso o situación de la vida real a través de agentes inteligentes, es necesario pensar en los diversos factores o elementos que serán requeridos, por ende, deberá elegirse la arquitectura que mejor se adapte a las necesidades dependiendo de qué tan complejo será cada agente, ya que estos pueden ir desde los agentes muy simples que interactúen en un entorno complejo hasta agentes capaces de razonar por sí mismos y aprender conforme van interactuando.

Ahora bien, lo que se pretende aportar es una nueva arquitectura de simulación de interacciones entre los siguientes tipos de agentes: reactivo, cognitivo, híbrido, basado en modelos, basado en metas, basado en su utilidad y Emoción-Creencia-Deseo-Intención (EBDI, por sus siglas en inglés: Emotion-Belief-Desire-Intention). En estas interacciones los agentes van a poder relacionarse sin problema alguno en

varios tipos de entornos, pero en solo uno de ellos a la vez. Así va a ser posible poder recrear situaciones en modelos más apegados a la vida real.

Por ejemplo, en un ambiente que simule cruceros y en donde se sabe existen conductores, semáforos y peatones, no todos los agentes que representan a estas entidades se comportarán por igual o similar; unos tendrán las necesidades de aprender, planear, experimentar entre otras características y otros únicamente necesitarán reaccionar ante una situación predeterminada. No obstante, aunque se estén realizando muchos avances en esta materia, de momento no se cuenta con algo que permita esa libertad en la gama de agentes que debe ser requerida.

De lo anterior, surge la siguiente interrogante: ¿Qué se puede hacer si ningún software, tecnología o producto existente cubre las características necesarias para ofrecer una alternativa confiable de modelado y simulación mediante sistemas multiagente?, la respuesta que se da a esta pregunta es la de crear una nueva arquitectura que contenga agentes heterogéneos capaces de comunicarse e interactuar, sin importar que sean simples o complejos. Basado en ello, en este trabajo se propone el crear una nueva arquitectura de red entre pares (P2P, por su nombre inglés: Peer to Peer) para desarrollo de aplicaciones basadas en agentes heterogéneos.

Cabe señalar, que lo que se quiere decir con el término agente heterogéneo, es que en dicha arquitectura se deben proporcionar agentes de diversos tipos, contemplando de momento los siguientes: cognitivos, reactivos, híbridos, basados en modelos, basados en metas, basados en utilidad y agentes de tipo Emoción-Creencia-Deseo-Intención (EBDI, por sus siglas en inglés Emotion-Belief-Desire-Intention).

Los agentes construidos en este trabajo van desde un agente abstracto o agente simple, el cual consta de las características básicas que debe tener todo agente como son la capacidad de percibir y tener acciones correspondientes a esas

percepciones, así como los agentes capaces de planear como son los cognitivos, hasta llegar a los de tipo EBDI.

Esta arquitectura será construida en el lenguaje de programación Java aprovechando de este modo todas las bondades que el mismo ofrece para crear soluciones mediante el paradigma de la programación orientada a objetos, resaltando que estas son multiplataforma y multihilo, lo cual permite trabajar de una manera más ordenada con aspectos de sincronización de procesos a nivel local y trabajo en red para comunicación remota y confiable entre diversas aplicaciones mediante sockets TCP/IP; mediante estos aspectos serán implementados los agentes y la comunicación entre ellos.

Del mismo modo, en este trabajo de tesis se implementarán los estándares que establece la Fundación para Agentes Físicos Inteligentes (FIPA, de su nombre en inglés: Foundation for Intelligent Physical Agents) **[6]**, los cuales son necesarios para lograr la comunicación en aplicaciones basadas en agentes heterogéneos, al seguir la estructura que deben tener los mensajes al momento de ser enviados, el servicio de transporte de mensajes, performativas (acciones o indicaciones de tipo petición/respuesta) las cuales sirven para especificar o describir lo que sucede con los mensajes, al tener agentes emisores y receptores.

Por último, también se emplean en conjunto con las especificaciones de FIPA los protocolos establecidos por el Consorcio de la Red Informática Mundial (W3C, de su nombre en inglés: World Wide Web Consortium) **[7]**, ya que ambos coinciden en algunas características para lograr el intercambio de mensajes entre los agentes. Sin embargo, en la plataforma que se propone en este trabajo, únicamente, se abarcará la parte del Modelo Arquitectónico Orientado a Mensajes, el cual consiste en definir la estructura que deberá tener un mensaje al momento de ser enviado; esta consta de: una dirección a la cual será enviado, una política de entrega, se debe definir a quién será enviado y quien lo envía, así como un mecanismo para asegurar que llegará completo y avisar en caso de que no haya sido posible el envío.

1.3 Delimitación

El desarrollo de la arquitectura se enfoca en la necesidad de requerir interacción de agentes heterogéneos, lo que implica modelar un agente abstracto como base, el cual va a contener las características principales que debe tener todo agente, para que a partir de él y usando mecanismos de herencia (generalización-especialización) se desarrollen los demás tipos de agentes con sus características en particular (cognitivo, reactivo, híbrido, basados en metas, basado en su utilidad, basado en modelos y EDBI).

Para el desarrollo de esta solución, la propuesta se apega a los estándares dictados por FIPA y los protocolos establecidos por el W3C, ya que éstos permiten la comunicación de los agentes bajo cualquier plataforma. De los estándares especificados por FIPA, serán cubiertos los siguientes: Lenguaje de Comunicación de Agentes (ACL, por sus siglas en inglés: Agent Comunication Lenguaje), Administrador de Agentes, Biblioteca del Acto Comunicativo, Protocolo del Transporte de Mensajes, Estructura del Mensaje ACL, y finalmente la validación del envío y recepción de los mensajes. Por otro lado lo establecido por el W3C será implementado sólo hasta el Modelo Arquitectónico Orientado a Mensajes.

La arquitectura que es propuesta en este trabajo de tesis, va dirigida a todos los desarrolladores que tengan la necesidad de generar soluciones basadas en agentes heterogéneos, lo cual exige emplear agentes de diferentes tipos para modelar las entidades que integrarán tales soluciones y de esta manera obtener resultados más cercanos a la vida real.

La arquitectura se desarrollará en el entorno de programación NetBeans para el lenguaje Java, siendo totalmente orientada a objetos, con ello se permitirá ver a las clases como plantillas de agentes, a partir de las cuales se generarán instancias, es decir, objetos dependiendo del tipo de agente que sea requerido en las simulaciones para las soluciones a ser modeladas.

1.4 Objetivos

En este trabajo de tesis, se cumplirá de manera satisfactoria con los siguientes objetivos:

1.4.1 Objetivo general

Crear una nueva arquitectura para aplicaciones de interoperabilidad que sea entre agentes inteligentes heterogéneos, es decir, que agentes de distintos tipos puedan interactuar dentro de una aplicación de manera local y remota, incluyendo soporte para escenarios de interacción de tipo Peer to Peer (P2P) siempre y cuando se cumpla con las especificaciones básicas que se indican en FIPA y el W3C para lograr el intercambio de los mensajes.

1.4.2 Objetivos específicos

De manera particular, se podrá verificar y cumplir con los siguientes objetivos:

- Generar la arquitectura de manera que cumpla con las expectativas de interacción entre los agentes de manera local y remota, incluyendo soporte para escenarios de interacción P2P, implementándola en el lenguaje Java.
- Implementar en la arquitectura los protocolos especificados por FIPA, así como el modelo de comunicación orientado a mensajes entre agentes inteligentes dado por el W3C.
- Crear paquetes de clases e interfaces para los tipos de agentes inteligentes: reactivo, cognitivo, híbrido, basado en metas, basado en modelos, basado en su utilidad y EBDI. Con los cuales, se podrán generar aplicaciones heterogéneas.

1.5 Hipótesis

Al verificar la arquitectura propuesta, se podrá constatar que es posible mejorar el desarrollo de sistemas que requieran de la interoperabilidad entre agentes heterogéneos cumpliendo con los estándares dados por FIPA y protocolos del W3C para que se pueda realizar el intercambio de mensajes de manera correcta y eficaz.

La arquitectura propuesta en este trabajo permitirá el intercambio de mensajes entre los siguientes tipos de agentes: reactivo, cognitivo, híbrido, basado en modelos, basado en metas, basado en su utilidad y EBDI, asegurando de esta manera una correcta comunicación al momento de relacionarse en su entorno; además, permitirá que cualquier agente tome el papel tanto de cliente (emisor) como de servidor (receptor), dando un comportamiento P2P y dependiendo de lo que demande la situación en la que se vea involucrado.

1.6 Motivación

El hecho de generar una arquitectura multiagente con comunicación heterogénea compromete a hacerlo de una manera segura, confiable y de calidad; esto se logrará por medio del uso de los estándares proporcionados por FIPA y W3C, usados por la mayoría de desarrolladores de arquitecturas para lograr la comunicación entre agentes, puesto que contienen:

- Protocolos para cuando sea necesario el intercambio de los mensajes incluyendo envío y recepción.
- Cuentan con lenguaje específico para la comunicación entre agentes.
- Tiene un administrador de agentes y de servicios parecido a un directorio, así como un protocolo para que un agente pueda solicitar realizar una acción o informar que la puede o no efectuar, incluso que ya la llevó a cabo.

- Un estándar para definir la estructura que deben llevar los mensajes; de modo que, como podemos ver nos da muchas opciones para lograr una buena comunicación.

Lo mencionado anteriormente son algunos de los beneficios que nos ofrece el uso de FIPA en conjunto con W3C, motivo suficiente para hacer uso de éstos y así cumplir con las expectativas que son requeridas en la nueva arquitectura que es propuesta en este trabajo de tesis.

1.7 Justificación

La razón por la cual se tomó la decisión de generar una nueva arquitectura es porque las existentes no siempre cumplen lo deseado o lo demandado por las necesidades del sistema que se desea crear, es decir, no cumplen los objetivos que requiere cubrir el desarrollador. Por lo tanto, a continuación se exponen algunos de los requerimientos que generalmente son considerados al momento de elegir una arquitectura para generar un modelo basado en agentes; los cuales, deben ser cumplidos por la arquitectura que en esta tesis se plantea, dichas características son:

- En primer lugar, la arquitectura debe ser capaz de modelar varios tipos de entidades ya sea una persona, institución e incluso cosas abstractas como modelos financieros.
- En segundo lugar, la arquitectura debe ser de uso libre ya que algunas necesitan que se pague por ellas para poder utilizar por completo sus bibliotecas de funciones o paquetes de clases, también la mayoría de desarrolladores prefieren que el lenguaje usado para programar sea Java ya que este es considerado como un lenguaje estándar para aplicaciones en red y además es multiplataforma.
- Se debe contar con una documentación clara de la arquitectura, la mayoría de los entornos de desarrollo en Java permiten la generación de la misma en

formato HTML a partir del código fuente. En este trabajo la arquitectura será implementada en NetBeans **[8]**.

- Por último, se debe permitir comunicación entre varios agentes para poder formar lo que en Inteligencia Artificial se conoce como población de agentes para diversos entornos o ambientes **[9]**.

1.8 Estructura de la tesis

El presente trabajo de tesis está organizado de la siguiente manera:

- En el capítulo 2, se da una explicación sobre qué es un agente, los componentes generales que forman su estructura así como los tipos de agentes que existen. Además, se incluyen los tipos de ambientes dentro de los cuales se pueden ver involucrados los agentes.
- Como extensión de lo visto en el segundo capítulo, en el capítulo 3 se explicará qué es un sistema multiagente, sus aplicaciones y las distintas arquitecturas que fueron estudiadas, así como una explicación sobre lo que es una arquitectura, un framework y una plataforma.
- La finalidad del W3C se verá en el capítulo 4, resaltando el mecanismo de trabajo y propósito que tiene este consorcio, así como, conceptos que son necesarios comprender al momento de crear aplicaciones para comunicación en red en base a protocolos Peer to Peer. También, se presentan los modelos arquitectónicos que son usados para el intercambio de mensajes entre agentes, la forma de solicitar y administrar recursos y servicios, sin descuidar las políticas necesarias para regular el comportamiento de los agentes al solicitarlos y hacer uso de ellos.
- En el capítulo 5, se hablará de manera clara y detallada acerca del propósito que tiene FIPA y los estándares que establece para aplicaciones basadas en el uso de agentes heterogéneos.
- La arquitectura propuesta será presentada en el sexto capítulo y en el mismo se detallarán casos de estudio para un mejor entendimiento de cómo debe ser

usada para crear soluciones personalizadas según las necesidades de las situaciones a ser modeladas y verificadas en las simulaciones que sean llevadas a cabo.

- En el capítulo 7, a manera de conclusiones se habla de los aportes, novedad y bondades que ofrece la arquitectura propuesta.
- El trabajo futuro a ser requerido para cumplir en su totalidad con lo que establece el W3C así como el agente interfaz, el agente de aprendizaje y el agente colaborativo es explicado en el octavo capítulo.
- Finalmente, si se desea tener acceso a la documentación o API (Javadoc) de la arquitectura propuesta y al contenedor .jar de los paquetes de clases e interfaces, se debe hacer una petición formal explicando las razones por correo electrónico a las siguientes direcciones: angie_mndz@hotmail.com o rafilla.orozco@gmail.com, esto es para su valoración y posible aprobación o rechazo.

Capítulo 2. Agentes Inteligentes

Actualmente, puede parecer sorprendente que a pesar de que se han estado empleando agentes durante ya varios años en muchas aplicaciones para diversas áreas, aún los científicos de la computación no puedan dar una definición concreta respecto a lo que es un agente, y esto en parte se debe al hecho de que tampoco se puede definir como tal lo que es inteligencia (la característica principal que describe a este tipo de entidades de software) ya que no hay parámetros que puedan decir con exactitud aquellos que deben estar presentes en todo razonamiento inteligente o proceso cognitivo asociado **[10, 11]**.

2.1 ¿Qué es un Agente Inteligente?

Con la finalidad de explicar y tener presentes las características que describen a un agente, en la tabla 2.1 **[10]** se presentan tres de las definiciones más aceptadas por las áreas afines a la Inteligencia Artificial y por la comunidad científica involucrada. En dicha tabla, al ir examinando tales definiciones y al llevar a cabo un análisis es posible observar que las mismas coinciden en varias de las características que deben estar presentes en estas entidades. Es por ello, que en el área de la Inteligencia Artificial un agente inteligente es considerado como una entidad software o sistema inteligente. Sin embargo, en este trabajo de tesis en base a las definiciones de varios autores **[11, 12, 13, 14]**, se propone la siguiente:

Agente Inteligente: es una entidad capaz de tomar decisiones de manera independiente o autónoma debido a las variaciones que surgen en su entorno dando como resultado el cumplimento de una o varias metas, esto lo logra por medio de sensores percibiendo información de entrada (como podrían ser archivos, paquetes a través de la red o los pulsos del teclado) y posteriormente gracias a esta información

los actuadores reaccionan conduciendo al agente a la toma de decisiones que da como resultado cambios en su entorno.

Autor y definición	Tipos
Maes **[12]**: *"una entidad autónoma que trata de satisfacer una serie de objetivos en un entorno dinámico complejo"*	Autónomo: cuando es capaz de decidir por él mismo cómo se realizarán las relaciones entre las percepciones y las acciones a realizar para de este modo logar sus metas, entonces se dice que es un agente autónomo. Adaptable: se dice que es adaptable porque puede mejorar su rendimiento a través del paso de tiempo, lo que quiere decir que satisface sus objetivos debido a su experiencia adquirida.
Wooldridge y Jennings **[13]**: *"un sistema de hardware de ordenador o software que tiene las siguientes propiedades:"*	Autónomo: agentes operan sin la intervención de terceros y estos tienen el control de sus acciones y el estado interno. Social: agentes interactúan con otros agentes (y los seres humanos potencialmente) a través de un lenguaje de comunicación o de la interfaz. Reactivo: los agentes perciben su entorno y éstos responden a sus cambios. Pro-activo: los agentes no simplemente actúan en respuesta a los estímulos percibidos del medio ambiente, sino que son capaces de exhibir un comportamiento dirigido por un objetivo o su propia iniciativa".
Franklin y Grasser **[14]**: *"un sistema que forma parte de un entorno en el que percibe y actúa a través del tiempo, de acuerdo con sus objetivos de alterar su futuro la percepción y las acciones".*	No establecen una clasificación de agentes.

Tabla 2.1: Definiciones y tipos de Agentes.

Ejemplos de los agentes inteligentes que cubren en su totalidad con la anterior definición se pueden encontrar en **[10]**, los cuales en un nivel básico siguen el

funcionamiento representado en la figura 2.1 **[11]**. En esta figura, otro de los conceptos importantes que se debe comprender es el saber que es una percepción, la cual se puede entender como el proceso de aprehensión de la información obtenida del entorno en cualquier instante de tiempo. De esta manera, conforme avanza el tiempo se va originando un conjunto distinto de percepciones formando un historial de las cosas que percibe el agente. Por otro lado, se puede especificar a cada uno de estos conjuntos una decisión, la cual dependiendo de la manera en la que sea elegida se verá reflejada en el funcionamiento del agente, haciéndole ver en algunos casos más inteligente que en otros, o bien, más dinámico que estático en casos en donde las mismas no sean previamente establecidas **[11]**.

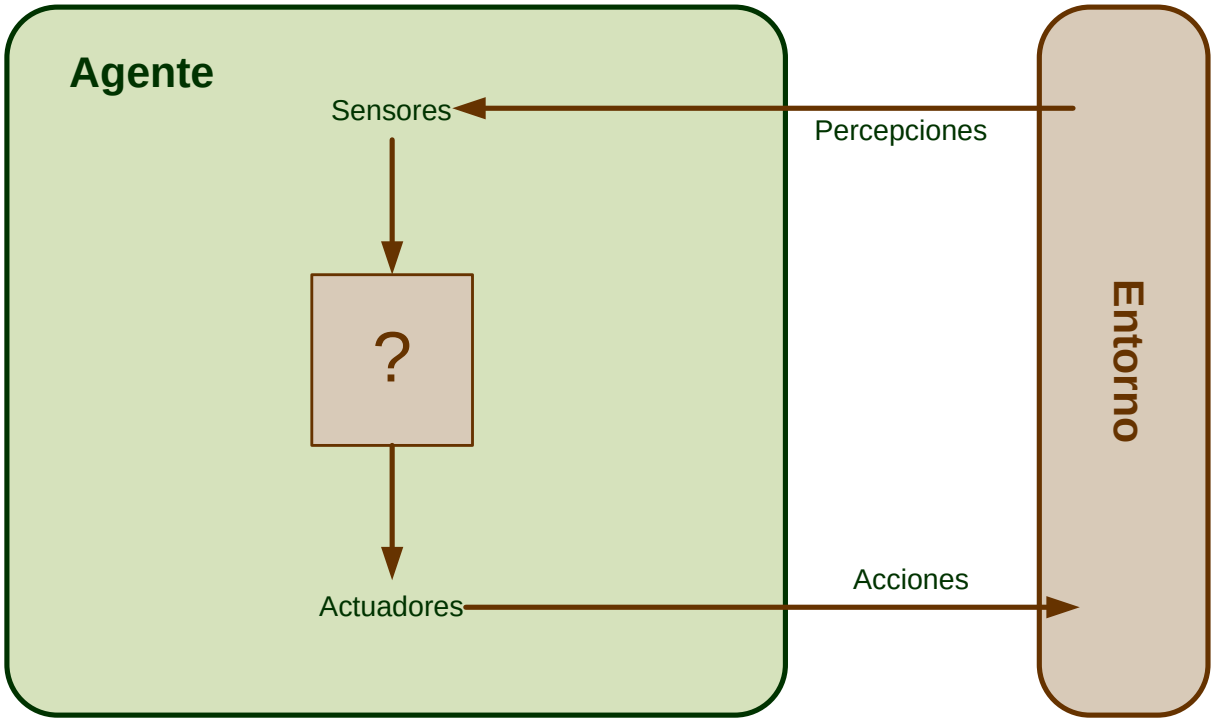

Figura 2.1: Elementos básicos de un Agente.

Otra parte fundamental en los agentes es lo que se conoce como función del agente, la cual es encargada de definir el comportamiento de éste, es decir, determinar qué acción le corresponde a cada percepción o conjunto de percepciones. Es esta

percepción la que será implementada por el programa de agente para expresar su comportamiento y ver los cambios generados en su entorno.

Existen algunas otras partes fundamentales dentro de un agente que sirven para dar un mejor resultado al momento de realizar la toma de decisiones, una de ellas es la estructura de un software agente la cual es la siguiente: los agentes perciben información de entrada a través de los sensores, posteriormente regresan una acción hacia los actuadores esto sucede en el caso de un agente simple. Por otro lado, si el agente es más complejo se tomará en cuenta el historial de percepciones antes de elegir una acción para enviarla a los actuadores (ver la figura 2.1).

La racionalidad dentro de un agente es un tema muy importante que no se pude dejar de lado en este campo de estudio que es la Inteligencia Artificial, la racionalidad de un agente se califica como el resultado correcto o incorrecto en base a algo llamado medida de rendimiento que es determinada por el programador del agente utilizando como datos para su evaluación el conocimiento del entorno, las entradas a través de los sensores y por último el resultado de las acciones tomadas.

2.2 Tipos de Agentes

Actualmente, no se cuenta con un consenso general que clasifique a los distintos tipos de agentes que existen. Sin embargo, a continuación se dará una descripción de los más sobresalientes.

2.2.1 Agente Reactivo

Sus acciones dependen de sus percepciones actuales sin tomar en cuenta un historial de estas. El hecho de que sea un agente simple no quiere decir que siempre se presentará en entornos básicos u ordinarios, incluso los entornos más complejos necesitan de este tipo de agentes. Esta clase de agentes cuenta con un tipo de

conexión encargada de enlazar el procesamiento de la información de entrada (a través de los sensores) y la reacción a la misma (por medio de los actuadores).
Rusell y Norving definen que a tal conexión se le llama *regla de condición::acción* y se escribe de la forma *si <condición> entonces <acción>* **[11]**. Esta regla es muy similar a la forma de pensar de los seres humanos al momento de tomar una decisión, generando analogías de este tipo antes de realizar una acción.

Ahora bien, este tipo de agente trabaja de la siguiente manera; cuenta con una función INTERPRETAR-ENTRADA encargada de realizar la descripción abstracta de lo que se ha percibido y la función REGLA-CORRESPONDENCIA que tiene como tarea retornar la primera regla del conjunto que coincida con la descripción dada. Por un lado, estos agentes tienen la ventaja de ser simples. Por el contrario, su inconveniente es tener inteligencia limitada, lo que significa que exclusivamente reaccionarán si el entorno es completamente observable, es decir, se conoce todo de él y lo qué se debe hacer ante cada posible situación (ver figura 2.2 **[11]**).

Cabe mencionar, que el hecho de ser simple tiene una desventaja causada por tener sus percepciones y acciones predefinidas, ya que si en algún momento se percibe alguna situación que no esté almacenada en la base de reglas del agente esto dará como resultado que el agente no sabrá cuál decisión tomar respecto a esa nueva percepción que no fue previamente definida. Esto puede causar problemas pero, para ello la Inteligencia Artificial puso una solución: si en algún momento se pierde el patrón respecto a la manera en que se esperaba recibir las percepciones, es decir, el entorno no es completamente observable, el agente entra en un bucle infinito del cual no podrá salir, la solución que se le da a esto es lanzar acciones aleatorias. En la figura 2.2 se muestra el esquema de un agente de este tipo.

En conclusión, este tipo de agentes realizan acciones predefinidas para cada configuración de sus sensores, gracias a los cuales se genera una representación interna de su entorno usando las reglas predefinidas. Las decisiones que toman están basadas únicamente en lo que perciben en el presente, lo cual indica que no

usan lo acontecido en el pasado (un historial de percepciones). Este tipo de agentes presenta las siguientes desventajas:

- Cada situación percibida en su entorno se debe definir en el sistema de reglas (percepción-acción), limitando al agente en tener aprendizaje continuo de los nuevos comportamientos.
- No tienen representación interna de su entorno, esto lo hace incapaz de razonar.
- No es capaz de planear a largo plazo.

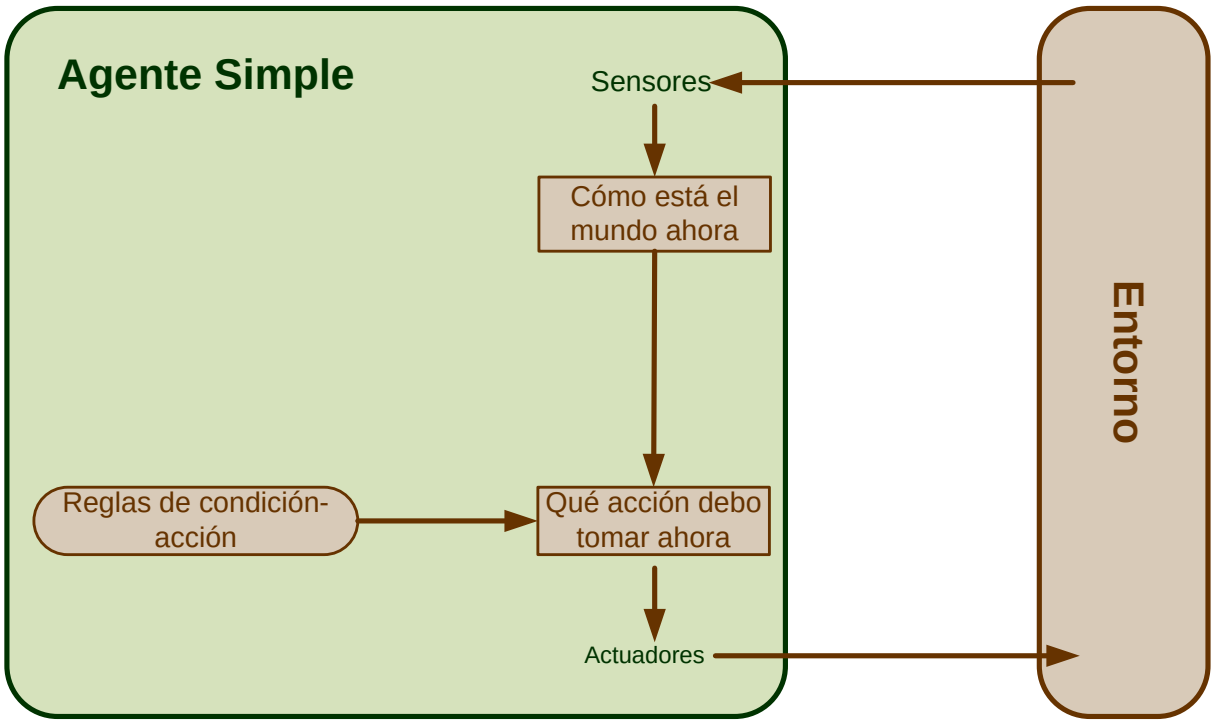

Figura 2.2: Representación de un Agente Reactivo.

2.2.2 Agente Racional o Cognitivo

Este tipo de agentes como se puede ver en la figura 2.3 **[11]** tiene una representación explícita y simbólica de su entorno y a partir de estas percepciones cada uno de ellos es capaz de tomar decisiones por medio del razonamiento lógico,

esto es posible por el conocimiento con el que ya cuenta (denominado creencias) y la información de entrada percibida por los sensores, dando como resultado su principal característica que es la capacidad de formular planes a largo plazo, así como lograr sus objetivos **[10]**. En este tipo de agentes se tienen los siguientes problemas:

- Dificultad al momento de representar todos los posibles estados en entornos complejos y dinámicos.
- Mantenimiento de la base de reglas en entornos dinámicos e imprevisibles.
- Falta de capacidad de respuesta de las aplicaciones en tiempo real.

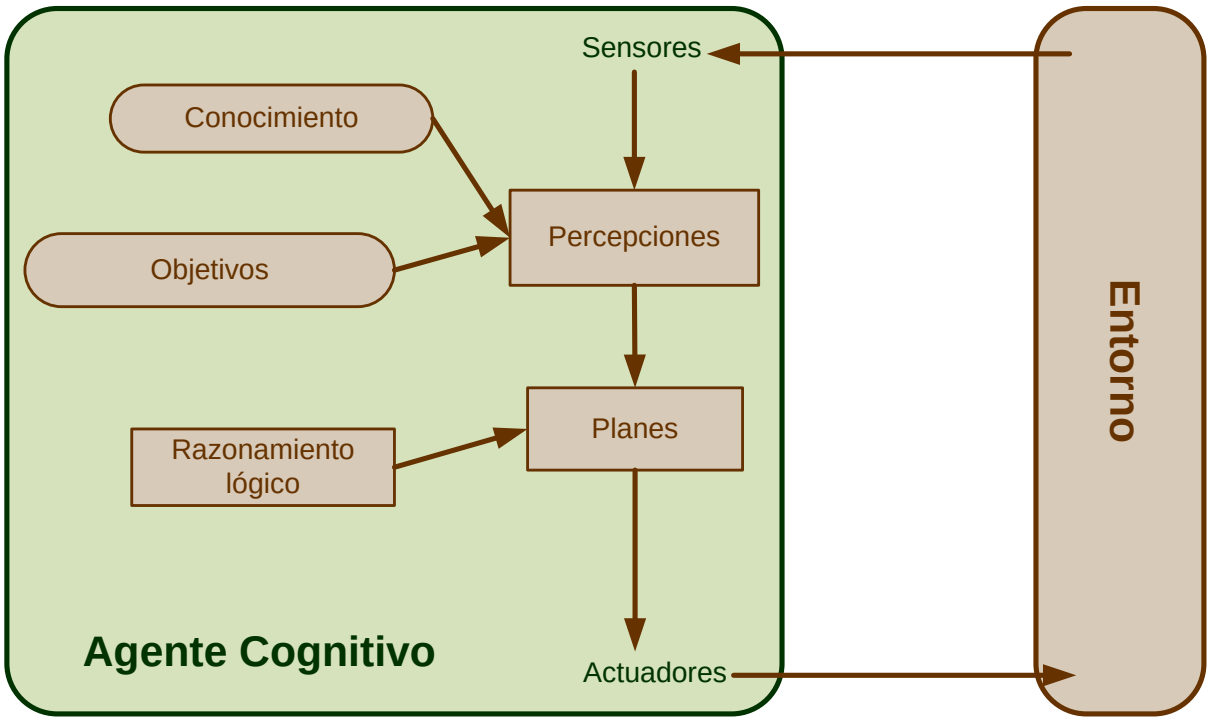

Figura 2.3: Representación de un Agente Cognitivo.

2.2.3 Agente Híbrido

Son agentes cognitivos con habilidades reactivas **[10]**, la combinación de las características de cada tipo antes mencionado les permite combinar sus cualidades

complementarias, limitando sus fallas en la figura 2.4 es posible visualizar cada una de las partes que integran este tipo de agente y cómo se relacionan esas partes.

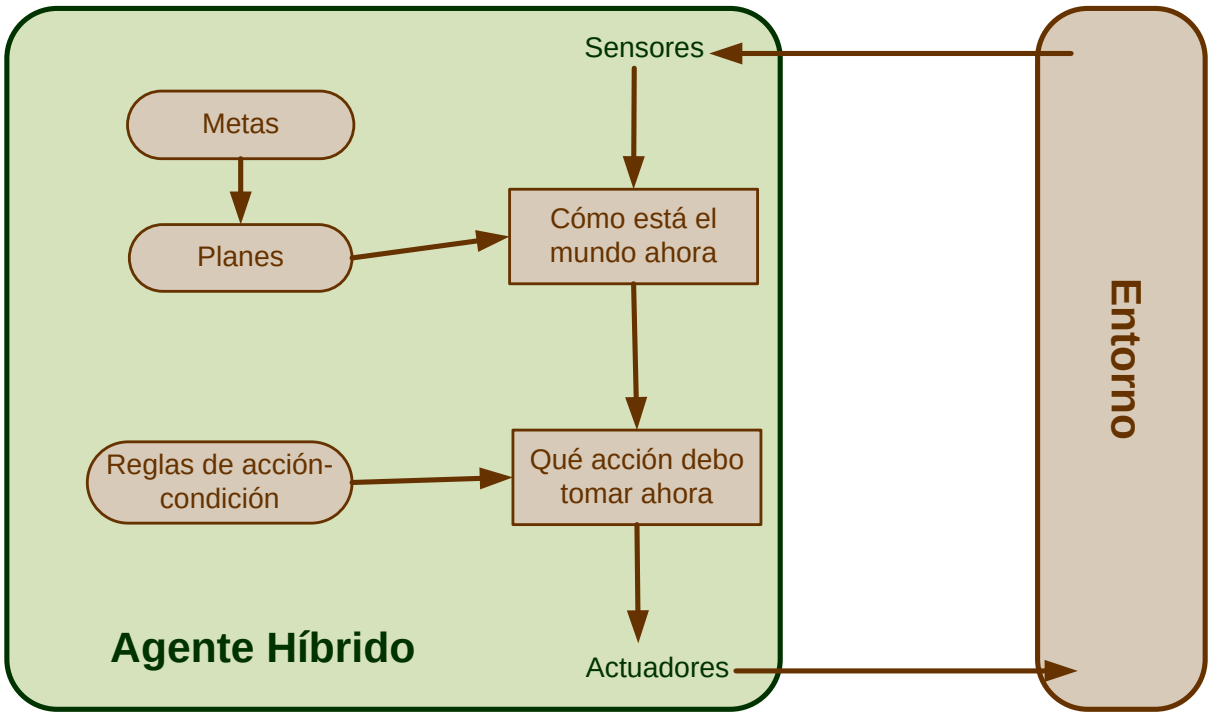

Figura 2.4: Representación de un Agente Híbrido.

2.2.4 Agente Basado en Modelos

Este tipo de agente al contrario del agente simple, cuenta con una mejor visibilidad puesto que tiene un estado interno que se va formando gracias al historial de percepción, de esta manera puede formular una visión más amplia en casos en los que no todo el ambiente pueda ser percibido por completo **[11]**. Esto es posible mediante la actualización de la información del estado interno, en primer término es necesaria información sobre cómo el mundo va evolucionando, en segundo término, se debe tener información sobre cómo las acciones del agente afectan al entorno; este conocimiento en Inteligencia Artificial es llamado: "cómo funciona el mundo", como resultado de ello se implementan circuitos booleanos o teorías científicas a lo que se le llama “modelo del mundo”, de esta razón se deriva el nombre del agente.

En la figura 2.5 **[11]**, se presenta la manera en la que este agente hace uso del estado interno al ir combinando la información contenida en éste con la percepción actual para generar una descripción actualizada dando como resultado un mejor estado actual, basándose en el modelo del agente sobre cómo funciona el mundo. La parte importante cuando se esté implementando el agente es la función actualiza-estado que fue la descrita anteriormente, encargada de crear la nueva descripción interna del estado.

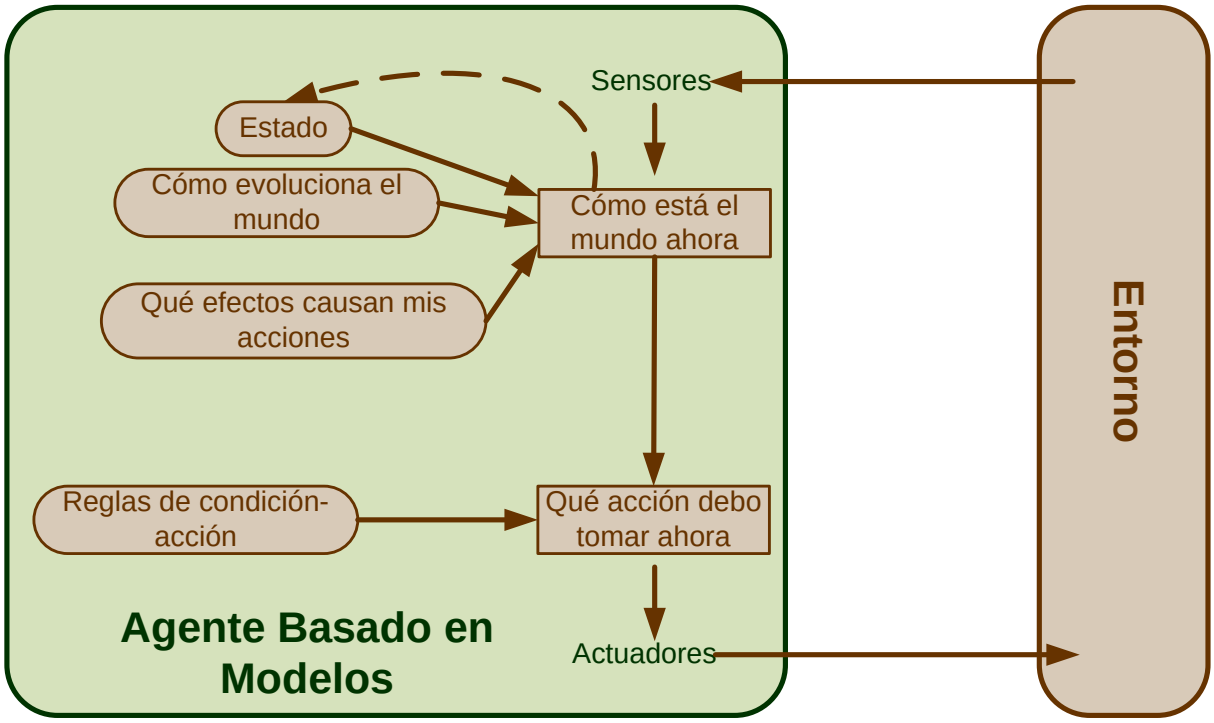

Figura 2.5: Representación de un Agente Basado en Modelos.

2.2.5 Agente Basado en Metas

La característica importante en este tipo de agente es que la decisión tomada depende de la meta como su nombre lo dice **[11]**, esto quiere decir que el agente no sólo necesita una descripción del estado actual, sino que también requiere que se le dé información respecto a la meta que se persigue. Para lograr esto es necesario que se describan las situaciones deseables en su entorno.

En ocasiones, la selección de acciones basándose en metas definidas es sencilla, es decir, hay momentos en los que se desea cumplir una meta y para lograrlo se hace uso de una sola acción. En otros casos, esta selección de acciones será muy complicada, ya que para lograr la meta será necesario pasar por una secuencia larga de acciones donde se considerará cómo es que afectaría la acción elegida en el futuro, esto se hace por medio de la búsqueda y planificación de estas secuencias para lograr llegar al objetivo planteado.

La figura 2.6 **[11]**, de manera clara establece que esta clase de agentes también tiene la característica de considerar el futuro como característica fundamental para la toma de decisiones, algunas de las preguntas que se plantearía un agente de este tipo antes de la elección de una decisión son: ¿Qué pasará si hago tal cosa u otra? y ¿Qué me hará feliz?

Otras ventajas es que tiene las capacidades de razonar y de ser flexible, pues el conocimiento que va adquiriendo así como las conductas pueden ser modificadas automáticamente conforme se actualiza dependiendo de las situaciones que se vayan presentando con el paso del tiempo para adaptarse a las nuevas condiciones derivadas de sus acciones en su entorno.

2.2.6 Agente Basado en su Utilidad

Para poder comprender los beneficios que tiene este tipo de agente sobre los demás **[11]**, se debe comparar con el agente basado en metas quien sólo se enfoca en proporcionar una manera de saber si el agente está feliz o no y cuáles son sus estados. Como mejora de lo antes mencionado el agente basado en utilidad hace uso de una medida de rendimiento que se encarga de comparar todos los estados del mundo permitiendo saber cuál estado realmente hace feliz al agente, a este resultado de felicidad se le conoce como “utilidad” (ver figura 2.7 **[11]**).

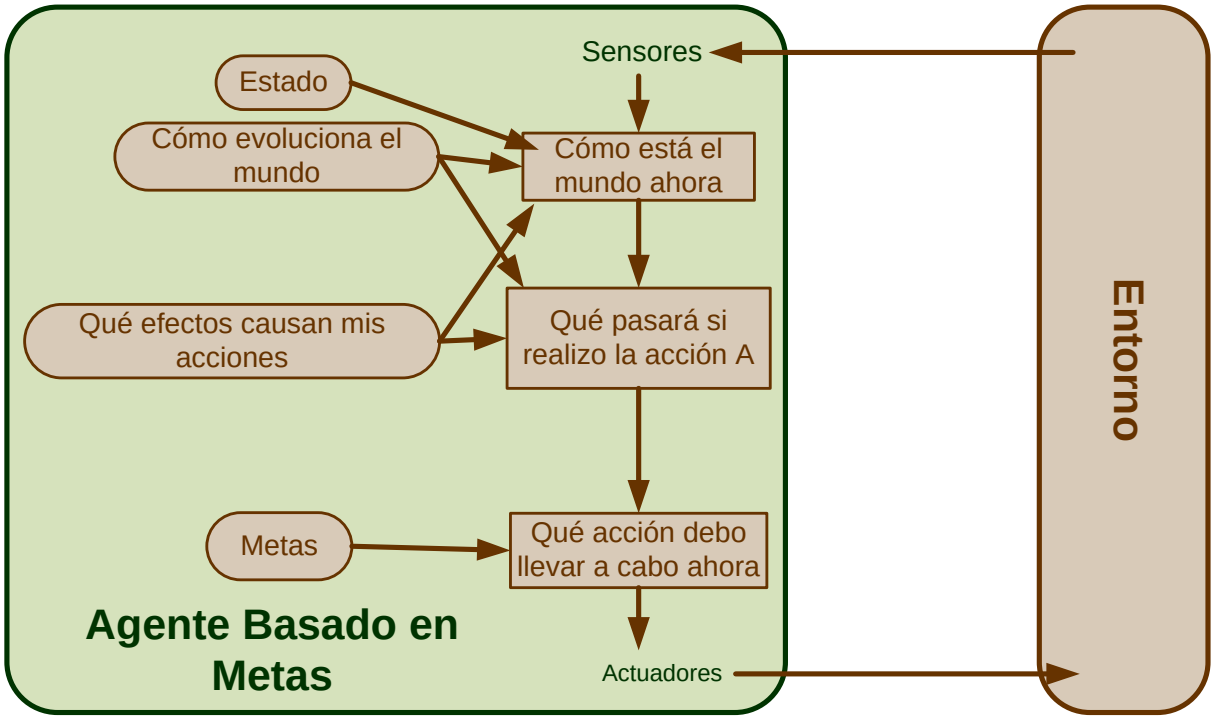

Figura 2.6: Representación de un Agente Basado en Metas.

La forma en la que trabaja este agente es la siguiente: la medida de rendimiento determina la utilidad para una secuencia de estados y así le será más sencillo el poder identificar las maneras convenientes para cumplir con su objetivo actual. Cuando la función de utilidad interna y la externa coinciden en sus resultados su utilidad será racional y proporcional a la utilidad externa. Esto indica que únicamente sería necesario manejar como única utilidad a la externa ya que es la que directamente influye de manera positiva en el resultado de evaluación de las acciones de los agentes lo cual claramente lo hará feliz, o dicho de otra manera mejorará su sensación interna de felicidad (mayor utilidad interna).

Los agentes basados en su utilidad tienen las ventajas de ser flexibles y la capacidad de aprender, por ejemplo cuando persigue objetivos contradictorios (el agente debe lanzar un resultado razonable) y sólo uno se puede realizar, la función de utilidad hace una compensación. Otro caso es cuando se tienen una serie de objetivos por

satisfacer, donde la utilidad se encarga de cumplir aquellos que conduzcan a un mayor éxito en el resultado.

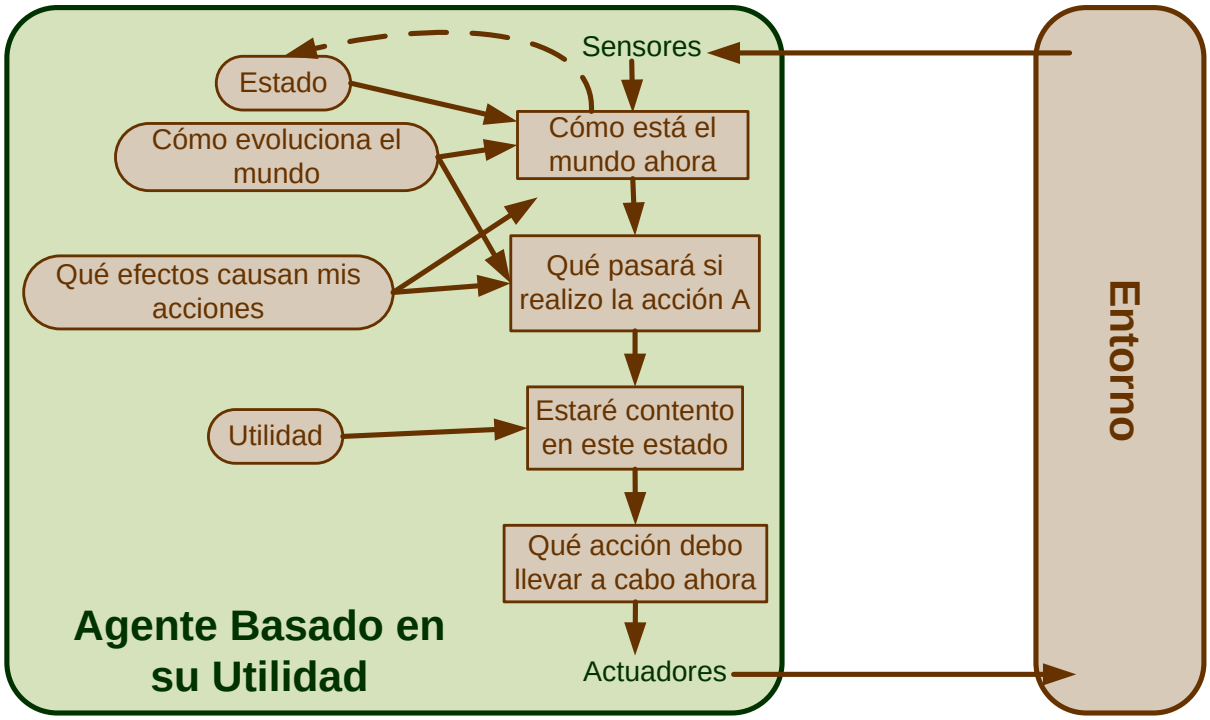

Figura 2.7: Representación de un Agente Basado en su Utilidad.

En términos prácticos, para este tipo de agente es deseable que se elija la acción que mejor maximice la utilidad esperada ante cada situación, lo cual resulta difícil en la mayor parte de los casos y en algunas ocasiones es más conveniente que se tome una acción que otorgue no necesariamente la mejor utilidad. Para ello, el agente lo logra tomando decisiones racionales (se denomina racional a las funciones con alto rendimiento) en base a un algoritmo que depende de la función de utilidad que se desea maximizar o bien la heurística que tenga implementada el agente.

2.2.7 Agente EBDI (Emotions-Beliefs-Desires-Intentions)

Para esta clase de agente, la manera en la cual llevan a cabo su proceso de toma de decisiones es por medio de la manipulación de estructuras de datos que representan sus emociones, creencias, deseos e intenciones ante cada situación que experimenten en su entorno (ver figura 2.8 **[15]**). Estos agentes son unos de los que mejor reflejan el razonamiento del ser humano y son convenientes para modelar simulaciones de conducta humana en donde sólo se pueda hacer uso de agentes inteligentes emocionales **[10]**.

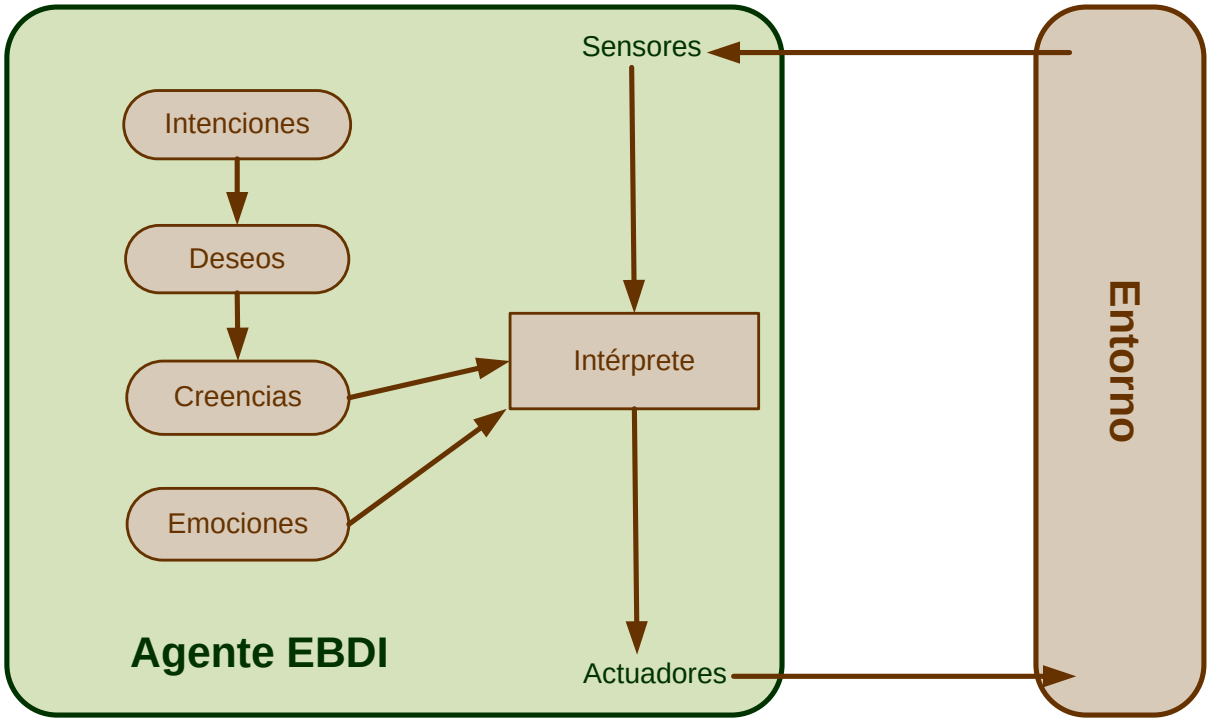

Figura 2.8: Representación de un Agente EBDI.

2.3 Tipos de Ambientes

El entorno **[11]** en el cual va a interactuar el agente es uno de los primeros pasos que se debe tomar en cuenta al momento de querer modelar un agente, para ser preciso, esto es requerido para poder especificar las características del agente y de

este decidir qué tipo es el que más conviene o mejor desempeño puede tener en el entorno modo lograr lo antes mencionado, por ejemplo, un agente racional, autónomo y con una respuesta de rendimiento maximizada ante un entorno de pronósticos de números de la lotería que pueden salir premiados en un futuro.

Entre las principales características que un agente necesita tener, se hallan el conocer su entorno y el tener sensores para obtener las percepciones. En base a estas percepciones el agente será capaz de formular su propia toma de decisiones a partir de las cuales van a funcionar sus actuadores. Por último, como resultado del cambio provocado en el ambiente y en base a su función de rendimiento de sus resultados se podrá validar el cumplimiento de cada uno de los objetivos que se esperaba fueran alcanzados.

La clasificación del entorno se da en base a las características antes mencionadas como se puede apreciar a continuación, los distintos tipos de entorno y sus diversas comparaciones se agrupan y explican de la siguiente manera:

2.3.1 Totalmente Observable vs Parcialmente Observable

Totalmente Observable: se presenta un entorno de este tipo cuando los sensores del agente detectan todas las situaciones de importancia que lo rodean, las cuales sirven en el momento de elegir una acción. Por otro lado, el grado de importancia de cada una de estas situaciones detectadas depende de la medida de rendimiento del agente. En estos casos, los agentes no necesitan de un estado interno ya que están al tanto de lo que sucede a cada instante en el ambiente.

Parcialmente Observable: un ambiente de este tipo puede presentarse por diversas causas como puede ser que los agentes no tienen suficientes sensores para detectar por completo las situaciones en su ambiente que es inexacto, posiblemente existen factores que alteran el funcionamiento de los sensores o en su defecto los sensores tienen deficiencias como podría ser poca capacidad de detección o de observación.

2.3.2 Agente Único vs Multiagente

Este entorno depende de cuántos agentes se encuentran dentro del ambiente, es decir, si únicamente participa un agente en el entorno se dice que es agente único, por el contrario si existen dos o más agentes se dice entonces que es un entorno multiagente. Para los entornos multiagente existen dos subtipos competitivos (todos están en constante lucha) y cooperativos (se coordinan para lograr objetivos en común).

2.3.3 Determinista vs Estocástico

Determinista: estos ambientes están presentes cuando el próximo estado del entorno está determinado por el estado actual y la o las acciones que realiza el agente.

Estocástico: también se le puede llamar no determinista ya que es lo contrario a ello. El entorno no es completamente observable, como consecuencia está presente la incertidumbre en los posibles resultados que se obtendrán de las acciones, su medida de desempeño debe marcar éxito para los agentes en todos los posibles resultados de acciones que se tomarán.

2.3.4 Episódico vs Secuencial

Episódico: se tiene que dividir cada experiencia del agente en episodios, cada episodio es formado por una percepción recibida a la que se le asigna una o más acciones; los episodios previos no influyen en la acción que se tomará en posteriores episodios.

Secuencial: contrario al episódico, en este entorno sí afectan las decisiones que se han tomado en el pasado, por lo que se dice que las decisiones tomadas a corto

plazo afectan a las decisiones tomadas a largo plazo. Este tipo de ambiente es más complejo que el episódico.

2.3.5 Estático vs Dinámico

Estático: para este tipo el agente no busca persistentemente cambios en el ambiente mientras está tomando una decisión y tampoco le es relevante el paso del tiempo.

Dinámico: cuando el entorno cambia constantemente mientras el agente está tomando una decisión se dice que es un ambiente de tipo dinámico. Como consecuencia, el agente se interesa por los cambios constantes en su ambiente, preguntándose frecuentemente qué es lo que quiere hacer y si no hay respuesta es tomado por el ambiente como que el agente no quiere hacer nada.

Semidinámico: cuando un entorno no cambia con el paso del tiempo pero el rendimiento del agente sí lo hace, se dice que el entorno es de tipo semidinámico.

2.3.6 Discreto vs Continuo

Discreto: este va dirigido al estado del ambiente, es decir cuando hay un número finito de percepciones, acciones y estados.

Continuo: al contrario del discreto este tiene un número infinito de estados, percepciones y acciones.

En la siguiente tabla **[11]** se muestran ejemplos de agentes que pueden pertenecer a uno o más tipos de ambiente.

A continuación, se dará una breve explicación del por qué los agentes contenidos en la tabla 2.2 pertenecen a cada uno de los entornos respectivamente:

Tipo de Entorno	Ejemplo de Agente
Estocástico	Taxi agente
Determinista	Aspiradora agente
Parcialmente observable	Aspiradora agente
Episódico	Robot clasificador
Secuencial	Ajedrez
Agente individual	Crucigrama
Multiagente	Juego de cartas

Tabla 2.2: Ejemplos de los tipos de Entornos en donde interactúan los distintos tipos de Agentes.

- El taxi agente pertenece a un entorno estocástico porque no se puede predecir el comportamiento del tráfico o algunas otras variables que puedan afectar el llegar al destino deseado.
- La aspiradora agente tiene un entorno de tipo determinista ya que el hecho de limpiar un área es una acción que determina la siguiente, la cual consiste en revisar las demás áreas hasta encontrar una sucia y entonces en ese momento se repite el ciclo de limpieza.
- Por otro lado, el entono de la aspiradora agente también puede considerarse como parcialmente observable porque con sólo un sensor de detección de suciedad el agente no puede saber si en otras áreas hay basura que limpiar. Por ende, el agente no tiene una vista completa de su ambiente.
- El entorno del robot clasificador es episódico ya que se encarga de buscar únicamente los objetos con características similares y los demás son desechados, lo que nos dice que esta clasificación no es influenciada por las acciones tomadas anteriormente.
- Un agente jugando ajedrez se encuentra en un entorno secuencial, con base en las acciones que se realicen actualmente se pueden afectar las posteriores, ya que si se realiza un mal movimiento se puede llevar al jugador a un jaque mate.

- Un agente resolviendo un crucigrama se desenvuelve en un entorno de único agente ya que como su nombre lo dice sólo un agente es necesario para resolverlo.
- El entorno de agentes jugando cartas es multiagente porque existen al menos dos agentes que interactúan en el juego.

Capítulo 3. Sistemas Multiagente

Como se ha venido mencionando desde el inicio de la investigación, los sistemas multiagente son sistemas en los cuales se puede hacer uso de los agentes para realizar simulaciones sobre algún problema o situación ya sea para predecir resultados que puedan mejorar o tratar de evitar algún momento o situación indeseable en la vida del hombre o en la naturaleza (evitar desastres). Estos sistemas son útiles en bastantes áreas como por ejemplo ciencias sociales, ciencias económicas, políticas predicción de desastres naturales, en el área de educación, por mencionar algunas ya que este tipo de herramientas tiene una amplia gama de aplicaciones.

3.1 Definición

Es un sistema formado por los siguientes componentes **[16]**: entorno o ambiente, objetos dentro del ambiente, agentes que son considerados elementos activos en el sistema, relaciones entre los elementos antes mencionados, acciones de los agentes que permiten observar e interactuar con los elementos del sistema, operadores que representan las acciones de los agentes y las reacciones de los agentes.

3.2 Aplicaciones

Actualmente, se han creado diferentes arquitecturas para resolver problemas por medio de agentes, algunas son hechas para resolver cualquier tipo de problema otras son para resolver problemas en específico, tal es el caso de la arquitectura que se explica a continuación.

La arquitectura FUSION **[17]** (Flexible User and ServIces Oriented multi-ageNt Architecture) tiene como objetivo permitir la comunicación e interacción de agentes para formar sistemas multiagente en entornos dinámicos, así como la comunicación e interacción de los usuarios con su entorno basándose en cumplir los requerimientos que exige la inteligencia ambiental, además de poder desarrollar sistemas multiagente distribuidos.

La arquitectura combina las tecnologías SOA (Arquitectura Orientada a Servicios) y Web Services para aprovechar las ventajas complementándose una con otra y de esta manera sus debilidades disminuirán, para cubrir las necesidades de los sistemas basados en Inteligencia Ambiental.

FUSION consta de las siguientes partes: aplicaciones, servicios, plataforma de agentes y protocolos de comunicación. La arquitectura permite que las aplicaciones distribuidas sean ejecutadas en diversos dispositivos, las peticiones y respuestas se administran por los agentes. Primero, los agentes analizan las peticiones solicitadas por las aplicaciones invocando de manera local o remota los servicios. Después, los agentes interpretan la petición al buscar un servicio que pueda procesar la petición. Enseguida, el servicio ejecuta la función y envía el resultado a la plataforma de agentes, la cual es quien envía el resultado a la aplicación que solicitó el servicio. La arquitectura hace posible todo este proceso de manera independiente al lenguaje de programación y el sistema operativo cumpliendo con el paradigma distribuido.

La arquitectura que a continuación se va a explicar **[18]**, es aplicada a un problema en específico ya que es para el área de medicina y servirá para detectar tres tipos de cáncer: pulmón, pecho e hígado. La implementación de esta arquitectura consta de un sistema de comunicación de audio y video interactivo en donde no habrá interacción entre el doctor y el paciente. La misma está compuesta por los siguientes agentes: un agente inicial encargado de realizar la interacción con el usuario (paciente), un agente coordinador quién será el intermediario entre el agente inicial y

el agente asistente y por último el agente asistente que realizará el procesamiento el problema.

La arquitectura se apoya del método clasificador de Naive Bayes para el proceso de toma de decisiones que realizan el agente inicial y el asistente, ya que este método es usado para realizar predicción bajo clasificación estadística.

Otro ejemplo de aplicación para un problema en específico, es la arquitectura de detección inteligente para el sensor de la NASA **[19]** y tiene como finalidad contribuir en la predicción del clima. Para ello, usa una red de sensores inteligentes funcionando de la siguiente manera: proporciona un grupo de agentes que operan naves y su comportamiento asegurando los objetivos de la misión de detección remota. Esta operación remota consta de dos partes: estaciones de trabajo en la tierra y un banco de pruebas compuesto de robots los cuales van a monitorear el clima enviando los datos través de WiFi.

El diseño de los agentes está basado en la combinación de estándares definidos por el OMG (Object Management Group) y FIPA. Para la implementación de los agentes se utiliza CORBA (Common Object Request Broker Architecture) para asegurar la interoperabilidad a través de plataformas de computación heterogéneas.

Por otra parte, se tiene el siguiente ejemplo de una arquitectura enfocada a un problema en particular, este es para la enseñanza por medio de un sistema capaz de cumplir las funciones que tiene un maestro o tutor **[20]**. El motivo de generar un sistema de este tipo es porque en caso de que un alumno no pueda tomar clases presenciales, como consecuencia las clases deben ser tomadas a distancia y la única herramienta que permite hacerlo es una computadora. El objetivo de generar una arquitectura que tenga la característica de enseñar es orientar al usuario o alumno en cualquier tema como por ejemplo, enseñar ecuaciones, ortografía, generar gráficas entre muchos otros temas que un agente sería capaz de explicar y resolver dudas. La arquitectura está formada por dos agentes que toman el papel un

cliente y un servidor y puede ser usado en situaciones para la solución de problemas o comunicación. A continuación, se puntualiza el proceso que se lleva a cabo en la arquitectura para la comunicación entre los agentes:

- Un agente en este caso el emisor genera un mensaje que será enviado a un agente receptor.
- El mensaje es transmitido a través de un canal de comunicación.
- Posteriormente, el receptor interpreta el mensaje y en conjunto con lo que fue percibido forma un patrón visual y auditivo entre otras sensaciones y procesos cognitivos como la imaginación para finalmente elaborar su concepción sobre el mensaje. De esta manera, cuando se desee enviar un mensaje el emisor podrá elegir de la lista de concepciones cuál es la más apta para la solución del problema.

Esta arquitectura utiliza únicamente agentes de un tipo que son de tipo Basado en Modelos, dividiendo la arquitectura en tres modelos que son: modelo de dominio, el cual contiene algunas características de la aplicación que será usada; modelo de comunicación, encargado de la interacción alumno-tutor (la máquina) formado a su vez por los diálogos y los materiales de aprendizaje para el alumno; finalmente, está el modelo de usuario que contiene información del usuario como su historial de interacción y datos del usuario (como por ejemplo nombre, sexo, nivel de progreso, etc.).

El último ejemplo que fue estudiado, es la arquitectura GeDA-3D **[21]** cuyo objetivo es lograr que un usuario sea capaz de decidir las características del agente así como del entorno en el que se va a desenvolver. Esto lo logra mediante la arquitectura de agentes la cual es capaz de generar autónomamente las conductas adecuadas para un entorno dinámico y de esta manera los agentes creados por el usuario actúen de manera adecuada a su ambiente.

La arquitectura GeDA-3D consta de las siguientes partes para lograr su funcionamiento: generar un método para la creación de los agentes que van a interactuar en el medio ambiente, los agentes son simulados por medio de un avatar a los cuales se les implantan expresiones faciales derivadas de su personalidad y emociones, también consta de un método para definir cómo afectan estas dos características (personalidad y emociones) al comportamiento, un lenguaje de comunicación, los protocolos para su interacción y finalmente con todas estas características propone formar una arquitectura para el desarrollo de sistemas basados en agentes.

Ahora bien, el lenguaje de comunicación entre agentes utilizado en este trabajo es el proporcionado por FIPA, el modelado de los agentes lo hace por medio de las redes de Petri ya que con esta herramienta es posible modelar todos los caminos posibles para lograr la comunicación (a esto en redes de Petri le llaman grafo completo), también utiliza álgebra de procesos para definir cómo se deberán realizar las metas de los agentes.

Los objetivos de esta arquitectura son: generar personalidad y estado emocional a los agentes, los objetivos a cumplir deben ser explicados detalladamente con cada una de las actividades a realizar así como su orden, definir el lenguaje con el que se comunicarán así como los protocolos, y por último, generar una base de conocimientos para que de esta manera le sea más fácil al agente el conocer su ambiente o entorno.

La arquitectura está integrada por un agente generador de entorno, sensores, acciones primitivas que son la base de las demás acciones a desencadenar, los efectores quienes envían las acciones que serán realizadas por los agentes al medio ambiente esperando el resultado de la acción para repetir ese proceso nuevamente, especifica la manera en la que se deberán llevar a cabo las metas por medio de álgebra de procesos, cuenta con una base de conocimientos la cual tiene dos funciones que son agregar oraciones y consultar las oraciones contenidas, como se

había mencionado anteriormente tiene personalidad y estado emocional, descripciones de postura para una representación en 3D del agente (el avatar) y su ubicación por medio de coordenadas 3D, interfaz de sistema de agentes encargada de solicitar y a su vez agregar una habilidad a la lista de comportamientos del agente, habilidades, repositorio de habilidades, un agente de transporte de mensajes que sigue el estándar FIPA, módulo de planificación que calcula la secuencia de acciones y el control de agente quien recibe y asigna el estado del agente así como sus habilidades y personalidad.

Finalmente, se puede concluir que aunque es una arquitectura muy completa de la cual es posible apoyarse ya que sigue estándares como FIPA respecto a la comunicación (así como también sigue las características básicas que se apegan al modelado y simulación de agentes como son sensores, percepciones y acciones), es semi-heterogénea y limita las posibilidades de creación de agentes y entornos.

3.3 Diferencia entre Arquitectura, Framework y Plataforma

A continuación, se revisarán el concepto y las características que tienen una arquitectura, un framework y una plataforma.

3.3.1 Definición de Arquitectura de Software

Clements dice **[22]**: es una manera de visualizar que componentes son los que integrarán un sistema, su comportamiento, sus intenciones y la manera de organizarse con el fin de lograr los objetivos planteados; así como especificar la estructura de los componentes que podrían ser:

- Protocolos para comunicación, sincronización y acceso a datos.
- Funcionalidad de los elementos.
- Distribución de los elementos.
- Composición de los elementos.

- Escalabilidad y rendimiento.

En el documento del estándar IEEE Std 1471-2000 **[23]** se define una arquitectura de software como la forma básica de organización de un sistema de acuerdo a los componentes que la integran, las relaciones entre ellos y su ambiente, su diseño y evolución.

Por otro lado, Les Bass **[24]** dice que una arquitectura es una abstracción del sistema, es un modelo pequeño con el cual es posible ver cómo se estructura el sistema y sus elementos.

En su trabajo Carlos Gómez **[25]** dice que son las partes fundamentales que forman una organización es decir: componentes, relaciones entre ellos y su ambiente junto con sus principios de organización.

De lo anterior, se concluye que el presente trabajo de tesis cumple con ser una arquitectura, ya que en ella se proporciona la estructura del software a partir del cual será posible el desarrollo de agentes inteligentes; esta estructura está integrada por un conjunto de agentes, cada agente se clasifica de acuerdo a las características que tiene cada uno, y finalmente, su relación depende de las características que comparten. Los componentes que la forman son:

- Protocolos proporcionados por FIPA y W3C donde es señalada la manera de comunicación, sincronización y registro de los agentes.
- Se explica la funcionalidad de cada una de las clases que integran la arquitectura.
- Distribución de los elementos.
- Cómo están compuestas cada una de las clases.

3.3.2 Elementos que integran un Framework

Según Eric Evans **[26]**, es considerado como una estructura con la finalidad de simplificar el trabajo a los programadores de sistemas ya que se encarga de problemas técnicos complejos y el único trabajo o la única tarea del desarrollador será realizar el diseño y funcionamiento de la aplicación, haciendo más fácil su desarrollo.

Por otro lado, Stephen T. Albien **[27]** dice que es una estructura donde es posible ver cómo se relacionan sus componentes, éste proporciona librerías de clases.

Javier Eguiluz en su libro **[28]** lo define como un conjunto de herramientas que permiten al desarrollador enfocarse en elementos específicos del sistema a diseñar ya que los elementos más básicos y especializados son proporcionados por el framework.

Una vez revisadas las definiciones anteriores se llegó a la conclusión de que un framework es una estructura que contiene un conjunto de elementos que son básicos pero complejos, permitiéndolos visualizar como librerías de clases, esto ayuda a facilitar y acelerar el desarrollo de un sistema ya que se reutiliza código que se especializará dependiendo de los requerimientos de cada aplicación.

3.3.3 Características que tiene una Plataforma

En **[29]** se dice que una plataforma es considerada como un entorno ya sea físico o lógico en el cual será posible ejecutar un conjunto de aplicaciones, en la actualidad la mayoría combinan una máquina (hardware) y un sistema operativo (software).

Una plataforma **[30]** tiene como propósito brindar diversos servicios a las aplicaciones dependiendo del tipo de cada aplicación para que éstas puedan ser ejecutadas de manera satisfactoria, dichos servicios son proporcionados de varias

maneras; por lo tanto, es necesario clasificar los diferentes tipos con los que una plataforma cuenta dependiendo de lo requerido; estos servicios pueden ser agrupados en cinco categorías:

Sistema Operativo: provee servicios fundamentales para que las aplicaciones funcionen, tal es el caso del sistema de archivos y requerimientos fundamentales para ejecutar código tales como programación.

Servicios de ejecución: se encargarán de proporcionar librerías para ejecutar software así como también para la creación de interfaces de usuario, comunicación con otro software, estructura sobre cómo ejecutar código, etc.

Servicios de datos: este da la posibilidad a las aplicaciones de almacenar y procesar datos; como por ejemplo un Sistema de Gestión de Bases de Datos.

Servicios en la nube: ofrece funciones que las aplicaciones pueden usar de manera remota, por ejemplo: uso de mapas, permitir a las aplicaciones buscar en internet o conectarse con otras aplicaciones.

Herramientas de desarrollo: ayuda a crear y mantener aplicaciones, puede ser desde editores de código hasta herramientas con soporte para escritura de código, pruebas, implementación y otros aspectos del proceso de desarrollo.

3.4 Arquitecturas para Aplicaciones Multiagente usando el lenguaje Java

Actualmente, existen muchas arquitecturas libres que proporcionan bibliotecas para los desarrolladores de dichos sistemas encargados de programar los agentes que se adecúen a las necesidades que exige el modelado. En estos sistemas se ven involucrados desde pocos a miles de agentes a quienes les definen una serie de reglas dependiendo del área a las que se está aplicando el sistema; como podrían

ser teorías científicas, modelos económicos, reglas matemáticas, comportamientos sociales, entre otras no menos relevantes.
Como se ha mencionado, es necesario realizar la elección de la arquitectura dependiendo de las necesidades que se tienen, las arquitecturas que contiene este apartado emplean el lenguaje de programación Java que es el mismo lenguaje utilizado para la arquitectura generada en este proyecto de tesis. A continuación, se expone un breve resumen sobre algunas de las arquitecturas que son utilizadas actualmente para la implementación de Sistemas Multiagente en Java.

3.4.1 Jade

JADE **[31]** es una tecnología que permite desarrollar aplicaciones multiagente basándose en la filosofía de la arquitectura Peer to Peer (P2P), por lo tanto esta permite la distribución ya sea de manera móvil o fija de los recursos, datos, aplicaciones y todos los demás objetos que compongan la aplicación.

Otra característica que proporciona JADE con su paradigma Peer to Peer es que los agentes funcionan de manera dinámica ya que dependiendo de los requerimientos de su entorno éstos podrán funcionar de dos formas, ya sea como cliente o como servidor sin importar si su comunicación es cableada o inalámbrica.

A continuación, en la figura 3.1 **[31]** se presenta el esquema de las diferentes maneras en las que JADE permite la comunicación de los agentes dentro de su plataforma, es decir, en dónde se integrará una vez formado el sistema multiagente.

JADE fue desarrollada en Java y sus principios son los siguientes: interoperabilidad ya que podrá relacionarse con otros agentes siempre y cuando cumplan con los mismos estándares, uniformidad y portabilidad porque cuentan con APIs que podrán ser usadas para cualquier entorno ya que son independientes de la versión, fáciles de usar puesto que hacen que los programadores utilicen sólo lo que necesiten y lo demás lo ignoren. Otra característica importante es que puede ser utilizado tanto en

plataformas complejas como en sencillas con capacidad de memoria limitada como J2ME (Java Micro Edition).

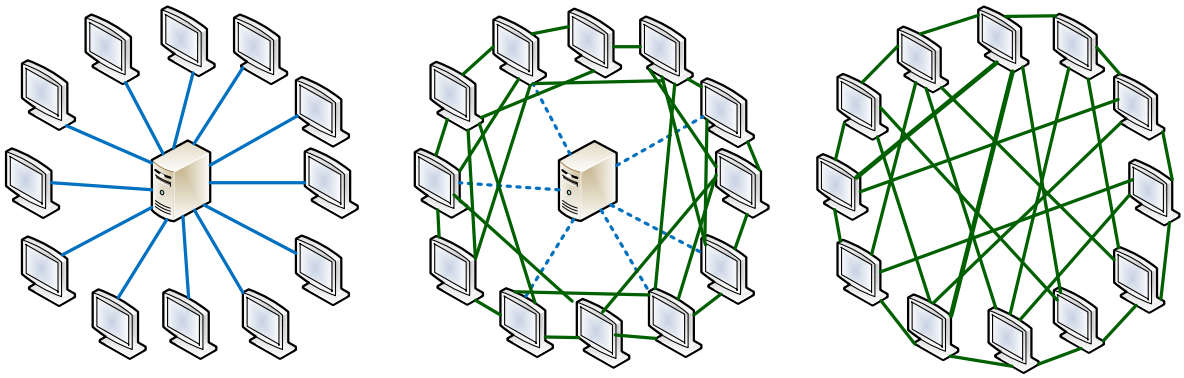

Figura 3.1: Tipos de comunicación que permite JADE.

En la figura 3.2 **[31]** es posible ver la manera en la que JADE organiza su arquitectura, en la primera capa se puede ver el Sistema MultiAgente, posteriormente el contenedor de agente y por último, la tecnología de Java que se utilizará dependiendo de los dispositivos que serán utilizados.

Las principales funciones de JADE son las siguientes:

- Cada agente tiene un nombre.
- Cada agente proporciona servicios que puede registrar o modificar.
- Su comunicación es asíncrona.
- Utiliza el lenguaje ACL (Agent Communication Language, de su nombre en inglés) definido por FIPA
- Tiene patrones de tareas específicas ya definidas.
- Puede ejecutar múltiples tareas (paralelamente).
- Tiene movilidad de código y estado de ejecución (ejecutar el código en diferentes ubicaciones remotas sin necesidad de instalar el código en cada host en cuestión).
- Distribución de carga de trabajo en tiempo de ejecución.

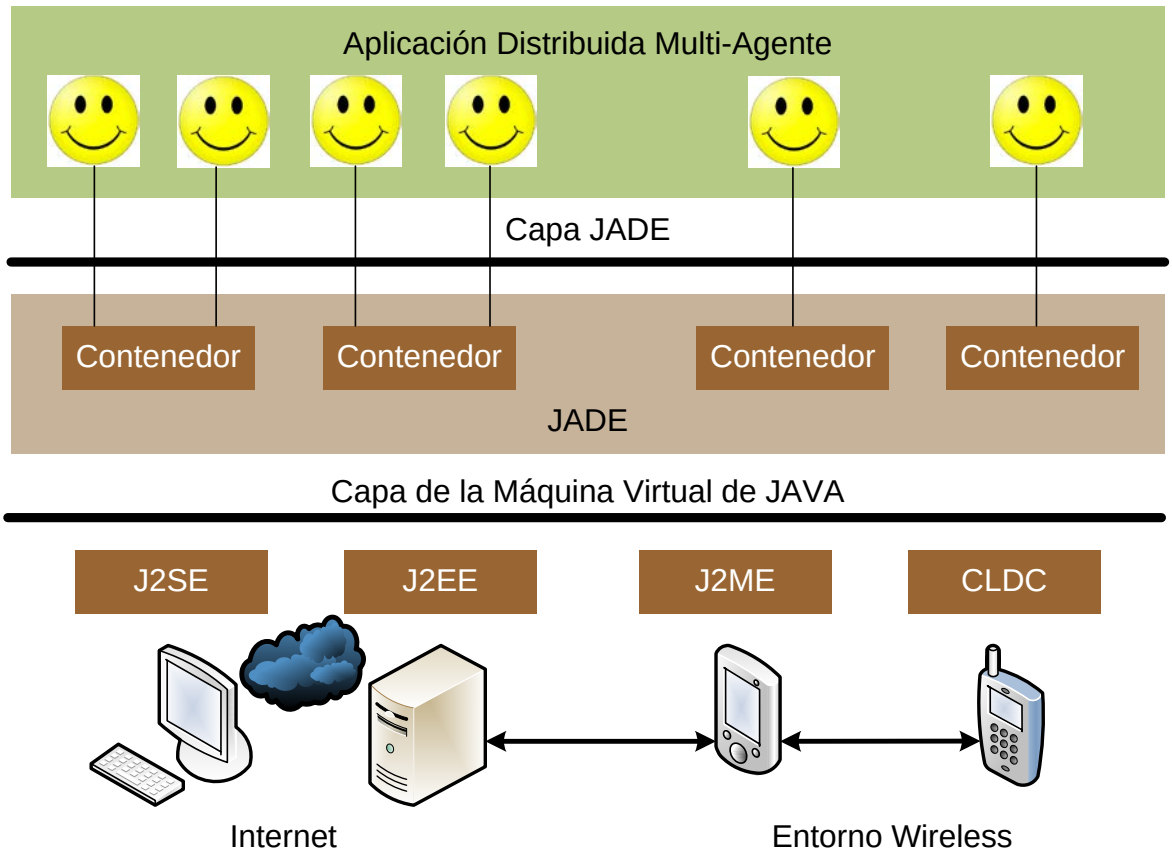

Figura 3.2: Capas de la implementación de JADE.

3.4.2 Jack

JACK **[32]** es una aplicación para desarrollo de sistemas autónomos, pero a diferencia de JADE su entorno de desarrollo es gráfico, por medio del cual se puede diseñar, implementar y monitorear de agentes tipo BDI (por su nombre en inglés, Belief-Desire-Intention). Esta arquitectura tiene como prioridad cubrir el paradigma sobre la autonomía en los sistemas, el cual pretende que el sistema sea capaz de relacionarse con agentes o con humanos y a su vez debe razonar independientemente sin la ayuda del humano.

JACK tiene las siguientes ventajas: puede ser ejecutado en cualquier sistema operativo sólo es necesario tener habilitado Java, es ligero por lo tanto puede

utilizarse en dispositivos con poca capacidad (es decir hardware de gama baja) así como también en servidores, es capaz de ejecutar miles de agentes, tiene un modelo simple de comunicación con otros agentes y su principal diferencia con otras arquitecturas es que tiene una interfaz gráfica de usuario.

Ahora bien, para entender la manera en la que trabaja esta arquitectura, a continuación se dará una breve explicación sobre las partes en las que JACK divide el paradigma de agente de tipo BDI. De modo que como primer paso permite a los programadores construir eventos, éstos son generados como respuesta a las percepciones externas dando como resultado un cálculo interno conocido como planes quienes tienen la tarea de definir un conjunto de pasos a seguir, dependiendo del plan que sea elegido de la lista de planes será lo que formará la siguiente intención para finalmente generar las creencias del agente, éstas serán elegidas de acuerdo a lo que más se asimile a la verdad misma y la de su entorno.

Los atributos de JACK son los siguientes: es autónoma es decir los agentes funcionan sin necesidad de ser operados por un humano enfocándose en cumplir los objetivos para los que fueron diseñados y finalmente deben ser capaces de adaptarse dinámicamente a los cambios en su entono, es compacta y eficiente lo que quiere decir que las aplicaciones que en ella se desarrollan son ligeras por lo tanto pueden ejecutarse en hardware con especificaciones limitadas, puede realizar especificación rápida (tiene elementos gráficos como arrastrar y soltar) y es resistente (puede dar soluciones alternativas a problemas inesperados en tiempo real).

Esta arquitectura no sólo permite generar agentes de tipo BDI sino que también consta del paradigma de trabajo en equipo el cual consiste en facilitar el modelado de agentes que requieran comportarse coordinadamente para con diversos agentes que formen un grupo para realizar tareas, esto funciona mediante algo a lo que JACK llama etapa de propagación de creencias a los integrantes del equipo colocando un

filtro para determinar que creencias serán necesarias propagar, cuales no y especificar a qué equipo de agentes serán enviadas.

Para concluir, a través de los datos proporcionados anteriormente sobre la arquitectura JACK es posible ver que tiene grandes ventajas y beneficios ofreciendo varias opciones de modelado para la simulación de situaciones reales de la vida cotidiana permitiendo hacer uso de agentes BDI, agentes colaborativos (agentes en equipo) pero no proporciona una base para utilizar agentes de otros tipos como por ejemplo; reactivos, híbridos o basados en su utilidad entre otros tipos más que existen y fueron explicados en el capítulo anterior.

3.4.3 AnyLogic

Es una herramienta de simulación **[3]** en la que se modelan Eventos Discretos basados en agentes y Simulación de Eventos Dinámicos, en ella el modelado de la simulación es gráfico haciendo uso de tres metodologías: Sistema Dinámico usado para las estrategias que se utilizan en solución de problemas, Eventos Discretos usado para las operaciones y el Modelado Basado en Agentes donde la persona que modela se enfoca en el comportamiento de los objetos que conformarán su simulación donde cada uno de estos será un agente. Las metodologías descritas anteriormente pueden utilizarse individualmente si el problema sólo requiere de las características de uno en particular, pero en su defecto si el problema es muy complejo será necesario combinar las tres metodologías.

3.4.4 MadKit

Es una plataforma multiagente libre programada en Java **[33]**, proporciona paquetes de clases que pueden ser probadas en cualquier entorno de desarrollo (estos paquetes incluyen documentación), también proporciona tutoriales donde se explica con ejemplos sencillos sobre cómo se pueden usar las librerías, la manera de

comunicación que emplea la arquitectura es Peer to Peer, permite usar aparte de Java algún otro lenguaje de Script, está bajo la licencia GPL.

3.4.5 Mason

Es una herramienta de simulación en java **[34]**, proporciona librerías para simulación multiagente, de igual manera que las anteriores está desarrollada en Java y puede funcionar en aplicaciones para visualizar agentes 2D y 3D, es posible generar tablas, gráficos y flujos de datos pero para lograr esto también proporciona paquetes específicos, proporciona un libro en el cual explica cómo se usan las librerías y es libre. Otra cosa muy importante que brinda esta arquitectura es su soporte a través del correo electrónico, al cual se informa y pide orientación respecto a dudas o problemas en particular. En la figura 3.3 **[34]** es posible ver la manera en la que trabaja Mason.

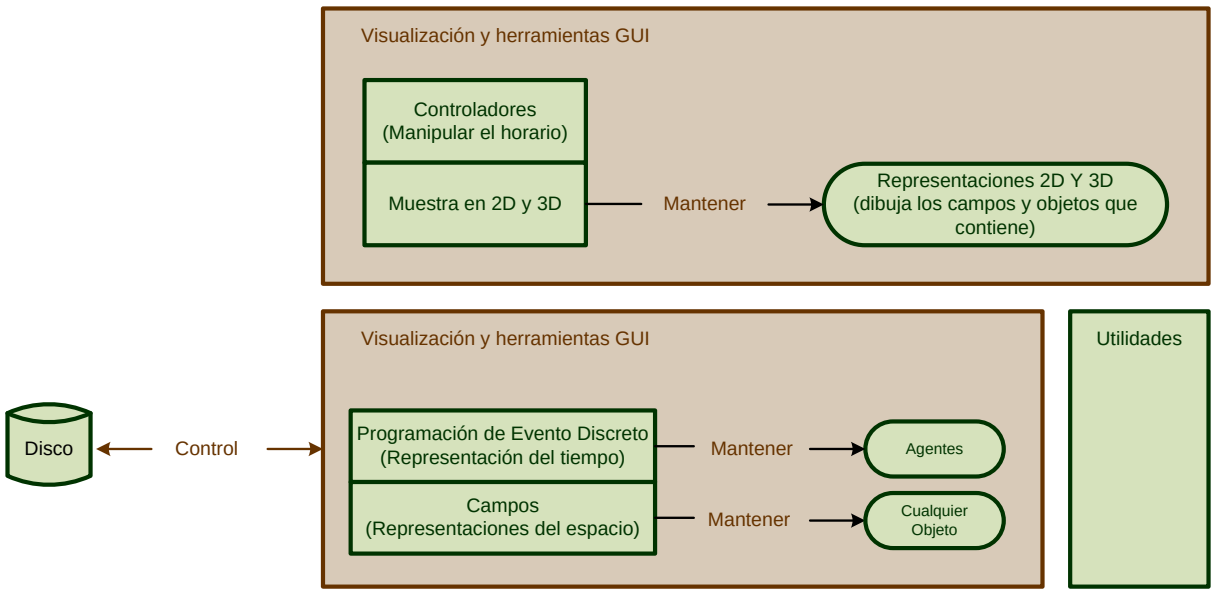

Figura 3.3: Elementos básicos del modelo y visualización de capas MASON.

3.4.6 StarLogo

Esta plataforma fue diseñada especialmente para ser usada por estudiantes y ayuda en el diseño de sistemas descentralizados **[35]**, es decir, sistemas en donde se requiera que las colonias o poblaciones de agentes sean modeladas de manera que no haya un agente principal que guie a todos, sino que todos los agentes se guíen por sí solos sin la necesidad de tener un jefe que les diga qué hacer.

Otra característica que se destaca en StarLogo es su interfaz gráfica de manera que consta de botones, barra de menús y comandos (para el uso de estos comandos existe un manual que contiene los que pueden ser usados). Algunas de las funciones que permite la barra de menús son: brindar información, mostrar resultados que lanza el proyecto creado.

Así mismo, proporciona ejemplos paso a paso para comenzar a familiarizarse con la interfaz y de esta manera aprender a hacer proyectos, otra manera de obtener información sobre cómo se debe usar e incluso obtener referencias es por medio de su correo y su foro de discusiones para obtener concejos útiles, entre muchas ventajas derivadas de los temas que se abordan.

Capítulo 4. Consorcio de la Red Informática Mundial

Existen aplicaciones en la web que funcionan con agentes, por lo tanto es necesario para estos casos tener reglas en casos de envío de solicitudes de servicios, o bien, de respuestas de atención a tales solicitudes para realizar una o más tareas dentro de un sistema multiagente. Es por ello, que en este capítulo se presentan una serie de conceptos en los que se explican las características y los elementos necesarios para llevar a cabo la comunicación dentro de un ambiente web, con base en los protocolos establecidos por el Consorcio de la Red Informática Mundial (W3C, de su nombre en inglés: World Wide Web Consortium).

4.1 Propósito

El W3C es un grupo de trabajo encargado de definir la arquitectura de servicios web. Este tiene como tarea la de identificar los elementos fundamentales a nivel mundial de los servicios de red que se requieren para garantizar la interoperabilidad entre servicios web, así como determinar las relaciones de sus componentes. La arquitectura de servicios web tiene los siguientes objetivos principales:

- *Interoperabilidad:* los servicios web están diseñados para realizar interacción entre una máquina y otra.
- *Extensibilidad, seguridad e integración web*.
- *Implementación:* la manera en la que se implementa el servicio web (el cual es considerado como abstracto) es por medio de un agente.
- *Administración de servicios web.*

4.2 Definición de Servicio Web

Un servicio web es un sistema de software diseñado para favorecer la interoperabilidad, la comunicación de máquina a máquina se documenta por medio de un documento escrito en el Lenguaje de Descripción de Servicios Web (WSDL) **[7]**, el cual es un formato procesable por máquina, en este se definen los formatos de los mensajes, tipos de datos, protocolos de transporte y se especifica las ubicación del agente proveedor.

Los servicios están formados por los siguientes elementos:

- *Agente:* software o hardware que envía y recibe mensajes.
- *Servicio:* recurso que proporciona servicios.
- *Entidad prestadora:* persona u organización que proporciona un agente para implementar un servicio en particular.
- *Entidad solicitante:* persona u organización que hace uso de un servicio web de una entidad del proveedor. La interacción se realiza mediante dos agentes, el prestador y el solicitante para intercambiar mensajes.

Para lograr el intercambio de mensajes tanto la entidad solicitante y como la prestadora deben estar de acuerdo en la semántica del intercambio de los mensajes.

4.3 Modelos Arquitectónicos

Estos modelos se encargan de explicar y resumir los elementos necesarios, así como las relaciones que forman para lograr la interoperabilidad entre agentes que integran la arquitectura de servicios web. Ahora bien, se procederá a explicar cada uno de los componentes para comprender la importancia que tienen aunque en algunos casos no sea necesario usar todos. En este trabajo de tesis, sólo fue implementado y tomado en cuenta el primero de ellos.

4.3.1 Modelo Orientado a Mensajes

Es el proceso que se lleva a cabo para generar el envío y recepción de mensajes, también está encargado de la relación entre los remitentes y los receptores de los mensajes así como de los mecanismos que se utilizan para el envío de estos (ver figura 4.1 **[7]**). Para enviar un mensaje es necesario saber una dirección, un protocolo de transporte de mensajes, un emisor, un receptor de mensajes y políticas para transportar el mensaje.

En la siguiente figura, se plasma que para el envío o recepción de los mensajes es necesario que se tenga información precisa, como es el caso de una dirección que describe cómo y dónde se debe enviar el mensaje, en este caso es un localizador de recursos uniforme (URL, por su nombre en inglés Uniform Resource Locator). Así como también se deben tener políticas de envío las cuales son reglas bajo las cuales se van a enviar dichos mensajes.

Los mensajes también tienen una estructura para su envío o recepción y es la siguiente:

- *Envolvente:* encapsula las partes que componen el mensaje, con esta es posible localizar información para el direccionamiento del mensaje.
- *Cabecera:* contiene información auxiliar sobre el mensaje para su enrutamiento, como por ejemplo, la cabecera HTTP.
- *Cuerpo del mensaje:* es la estructura que representa el contenido que el emisor (agente proveedor) del mensaje va a entregar al receptor (agente receptor). Un ejemplo de esto pueden ser los parámetros codificados que se encuentran dentro del URL.

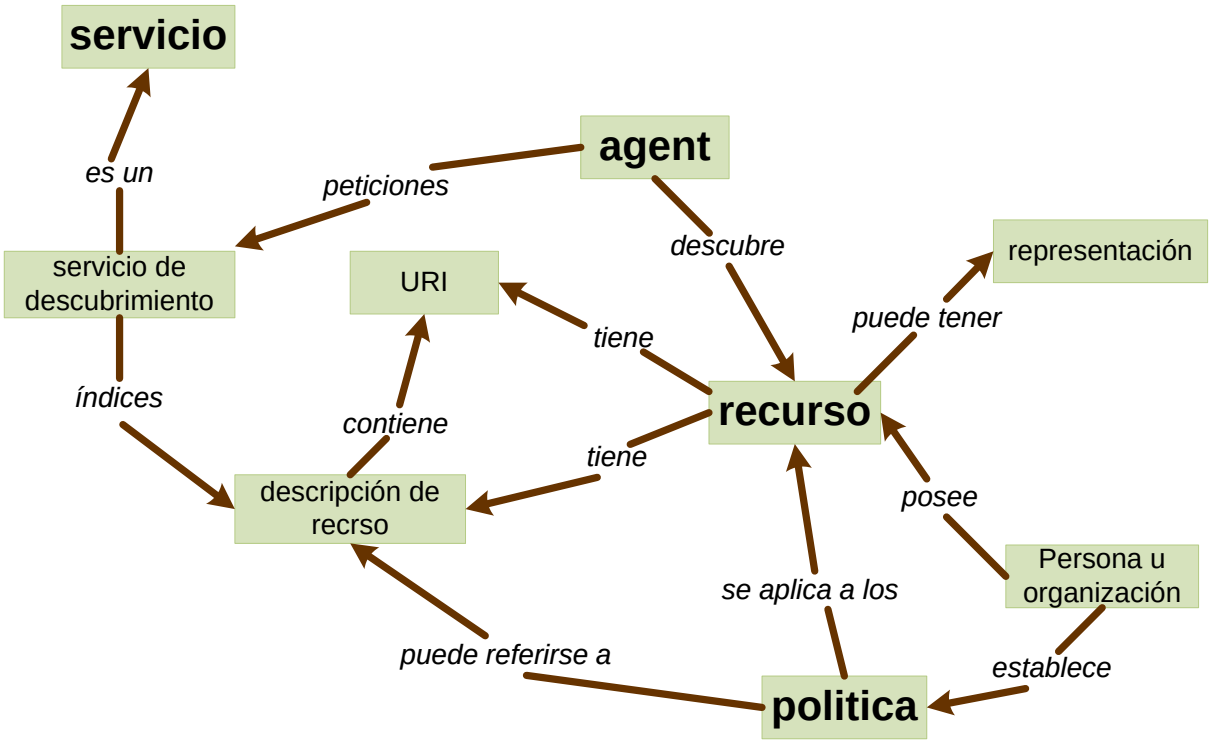

Figura 4.1: Modelo Orientado a Mensajes.

4.3.2 Modelo Orientado a Servicios

Este modelo se encarga de explicar la relación que existe entre el agente y el servicio, es decir, las acciones que se llevan a cabo para enviar y recibir el mensaje hasta llegar al objetivo que se desea. Su estructura se visualiza en la figura 4.2. **[7].** Este modelo tiene elementos que le ayudan a un mensaje a llegar a su meta y son:

- *Agente:* es un software en ejecución que maneja servicios web para solicitar o realizar un servicio.
- *Coreografía:* determina la secuencia y condiciones de los agentes al momento de intercambiar los mensajes para lograr su objetivo.
- *Objetivo del estado:* son las acciones relacionadas con los servicios a realizar.
- *Agente proveedor:* agente que proporciona un servicio en este caso es el envío de mensajes.

- *Entidad proveedor:* persona u organización que ofrece un servicio brindado por el agente proveedor.
- *Agente solicitante:* agente que invoca al agente proveedor para pedir un servicio.
- *Entidad solicitante:* persona u organización que necesita un servicio.
- *Servicio:* es una acción encargada de realizar una tarea.
- *Descripción de servicios:* sirve para encontrar de manera fácil un servicio, contiene detalles como comportamiento que tendrá el servicio, tipos de datos, protocolos de transporte y dirección.
- *Interfaz de servicios:* contiene tipos de mensaje y los patrones de intercambio de mensajes.
- *Servicio intermediario:* este realiza la ruta que debe seguir un agente ya sea solicitante o proveedor para llegar a otro agente.
- *Rol de servicio:* es el intermediario entre los servicios y las tareas.
- *Semántica de servicio:* son todos los acuerdos, efectos y requerimientos para que exista una buena relación entre la entidad proveedora y la solicitante.
- *Servicios de tareas:* son el conjunto de tareas necesarias para cubrir el servicio solicitado.

4.3.3 Modelo Orientado a Recursos

Como su nombre lo dice este modelo está dirigido a las propiedades de los recursos **[7]**, para ello es necesario explicar antes lo que esto quiere decir. Un recurso es cualquier información que tenga un identificador, una descripción (dato legible por máquina que permite que los recursos sean descubiertos) y una representación (dato que describe el estado del recurso, como por ejemplo, todos los valores que se proporcionan en una dirección URL al momento de enviar un GET o un POST).

Este modelo funciona de la siguiente manera; como primer punto se necesita un servicio de descubrimiento que es el encargado de publicar las descripciones de los recursos conociendo cada una de sus funciones que puede realizar, como

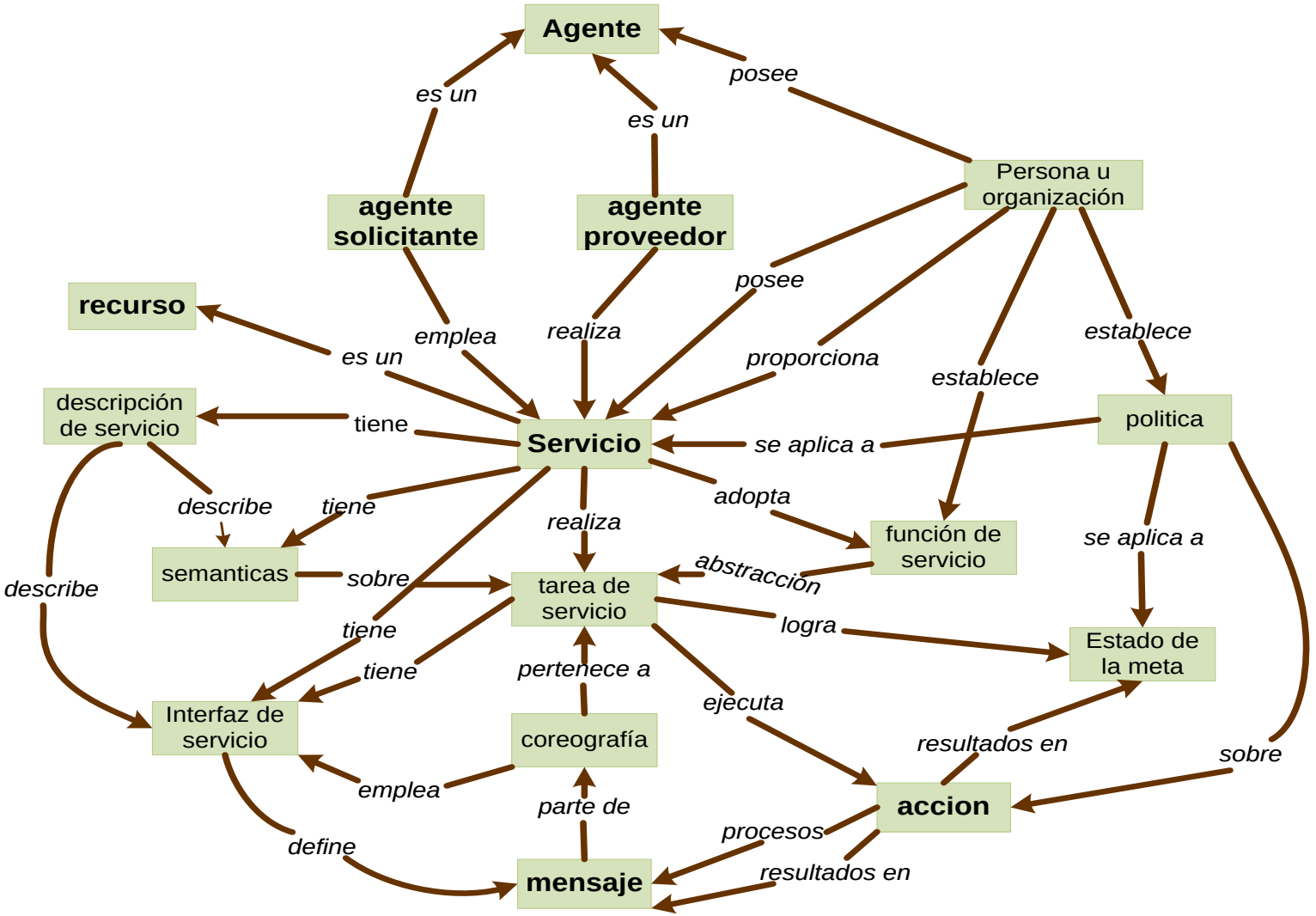

Figura 4.2: Modelo Orientado a Servicios.

consecuencia de esto, al agente solicitante le es más fácil buscar un agente proveedor que pueda solucionar su tarea para lograr su objetivo.

En la figura 4.3 **[7]** se muestra que están representados todos los elementos del modelo orientado a recursos así como las relaciones que forman entre sí.

4.3.4 Peer to Peer

Para este tipo de arquitectura, los servicios web son denominados nodos en una red y trabaja de la siguiente manera: el agente solicitante consulta a sus vecinos buscando un servicio web adecuado que pueda realizar las tareas que necesita, si alguno de estos nodos coincide con lo demandado por el agente solicitante este responde, de lo contrario, esta consulta se propaga con sus pares vecinos a través de toda la red cubriendo una serie de saltos y formando algo parecido a una malla hasta encontrar el servicio que cubra sus expectativas.

Estos nodos se contactan directamente mediante un índice de servicios web, el cual se actualiza constantemente. En el proceso que se lleva a cabo dentro de esta actualización, existe algo llamado latencia que es la diferencia de tiempo entre la actualización del servicio web y la actualización de la descripción **[7]**.

Esta arquitectura tiene ventajas que son las que se mencionan a continuación:

- Proporciona confiabilidad porque tiene rendimiento en los costos.
- Cada nodo es responsable de proporcionar recursos mediante la propagación a sus pares.
- Los nodos pueden funcionar como agentes proveedores y como agentes solicitantes.

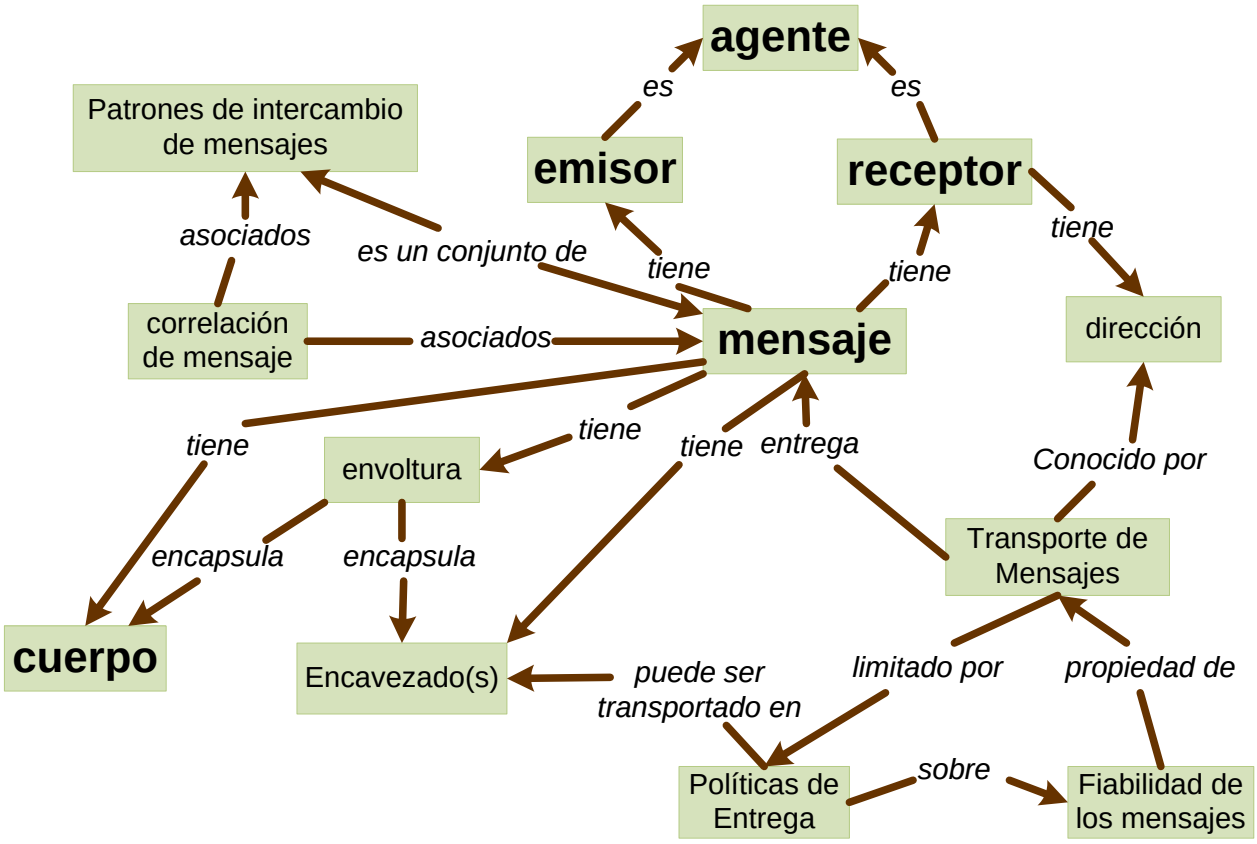

Figura 4.3: Modelo Orientado a Recursos.

Ahora se mencionarán las desventajas:

- No predice el trayecto de propagación.
- Los nodos proporcionan información constantemente que puede que no sea de interés para el nodo par.
- Lo anterior provoca ineficiencia y sobrecarga en la arquitectura, lo cual puede tener remedio, o bien, tratamiento la mayor parte de las veces.

Capítulo 5. Fundación para Agentes Físicos Inteligentes

Así como existen reglas para lograr la comunicación de los agentes dentro de un entorno web también hay especificaciones que deben seguir las aplicaciones para asegurar la interoperabilidad entre los mismos. En el presente capítulo, se van a tratar los diferentes estándares existentes para todos los posibles estados en los que un agente puede verse involucrado.

5.1 Propósito

La Fundación para Agentes Físicos Inteligentes (FIPA, de su nombre en inglés Foundation for Intelligent Physical Agents) es una organización encargada de desarrollar especificaciones para lograr la interoperabilidad entre agentes y aplicaciones basadas en estos. Estas especificaciones consisten en un conjunto de estándares que definen las características que deben cumplir las plataformas que administran los sistemas multiagente, algunas de estas características son: creación, destrucción, registro, localización y comunicación entre agentes.

Las especificaciones también tienen como objetivo definir los servicios que son esenciales para la interacción y comunicación entre los agentes, algunos de estos servicios son el Servicio de Transporte de Mensajes, Sistema de Administración de Agentes, Directorio Facilitador y el Canal de Comunicación para Agentes, entre otros. Todos estos servicios tienen un agente específico para ser utilizado, por lo tanto, la comunicación entre ellos es por medio de otro estándar establecido por FIPA que es el Lenguaje de Comunicación para Agentes (ACL, de su nombre en inglés Agent Communication Language) **[6]**.

5.2 Estándares

Ahora bien, ya que se ha explicado que es FIPA y se ha dado un breve resumen sobre algunos elementos que integran a un agente, se explicará más detalladamente cada uno de esos elementos y el papel que juegan dentro de los estándares.

5.2.1 Arquitectura Abstracta

Es un estándar encargado de especificar las partes básicas con las que debe contar toda arquitectura para la construcción de un sistema de agentes que se reglamente bajo las especificaciones de FIPA. Estas reglas incluyen tanto la construcción de agentes como el empleo de servicios necesarios para lograr la comunicación entre ellos, como por ejemplo: lenguajes para la interoperabilidad entre agentes y servicios de agente.

Esta especificación se planteó con el fin de generar una arquitectura abstracta concreta, cabe mencionar que no se puede implementar como tal, sino que su única función es señalar las bases para el desarrollo de las arquitecturas que requieran ser compatibles con FIPA. Las arquitecturas generadas en base a esta deben incluir: mecanismos para registro de agentes, descubrimiento de agentes y transferencia de mensajes. La descripción de los elementos abstractos (es decir los elementos que forman a la arquitectura) se denomina realización.

En la figura 5.1 **[6]** se muestra cómo se podrían usar las características que establece FIPA al implementar el Protocolo Ligero de Acceso a Directorios (LDAP, de su nombre en inglés Lightweight Directory Access Protocol). Sin embargo, no sólo existe esta manera de usar la arquitectura ya que no se limita a usar sólo sus elementos, también se le pueden agregar muchas otras características siempre con el fin de que la arquitectura logre su objetivo.

Con base en la figura 5.1 uno de los propósitos principales de esta arquitectura es proporcionar interoperabilidad y reusabilidad, lo que quiere decir que es necesario detectar los elementos que se van a codificar. En caso de que haya dos diferentes enfoques se deben extraer los elementos similares para programarlos en uno sólo, el cual debe servir para ambos enfoques.

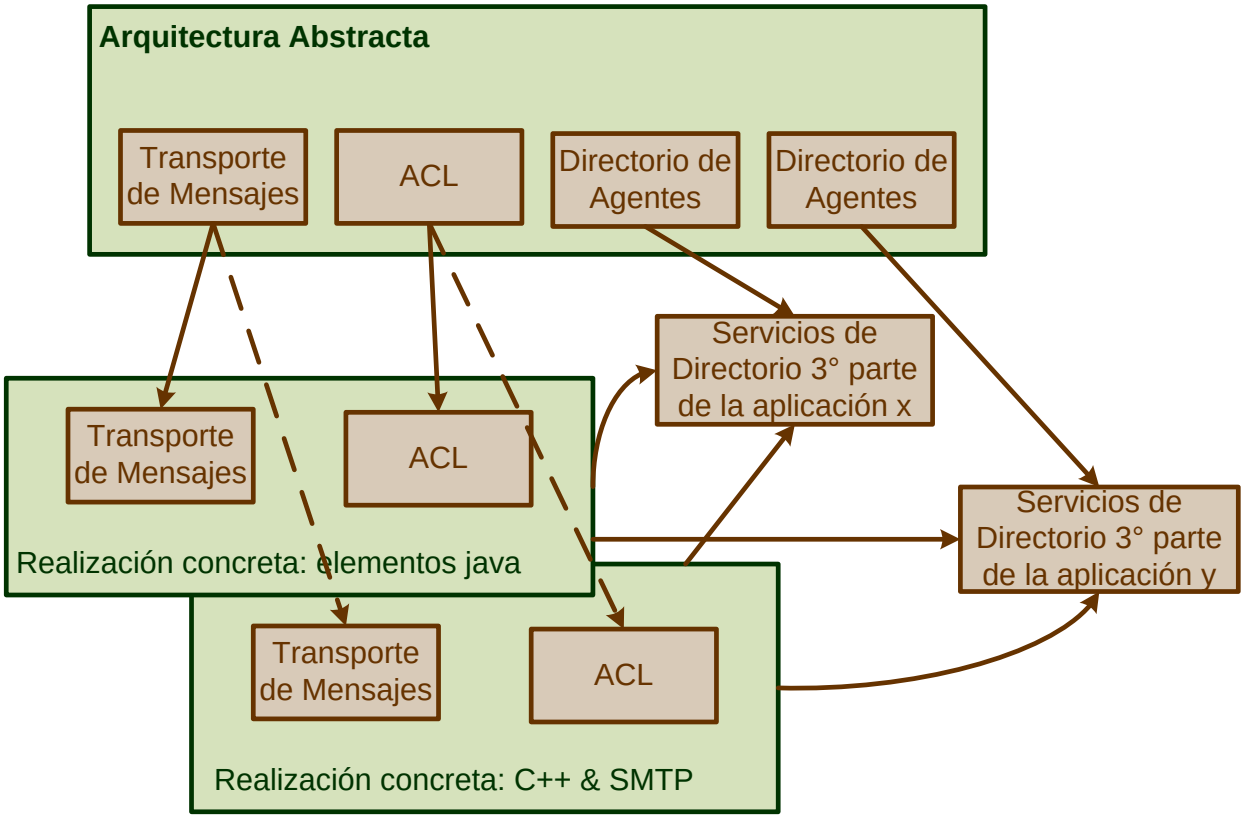

Figura 5.1: Implementación de la Arquitectura Abstracta en un caso concreto.

Los elementos que forman la arquitectura abstracta son los siguientes: agentes, de los cuales se derivan elementos que lo integran para su funcionamiento y estos son iniciar un agente, servicio de directorio de agentes, registrar un agente, descubrir un agente, servicio de directorio de servicios, mensajes de agente, estructura de mensajes, transporte de mensajes, envío de mensajes a otros agentes, validación de mensajes y encriptación de los mismos.

En conclusión, esta arquitectura proporciona los elementos mínimos necesarios para el buen funcionamiento a nivel abstracto, ya que define cómo localizar y cómo comunicar un agente con otro, a su vez es posible extenderla ya que no se limita a sólo los elementos definidos por esta sino que se le pueden agregar más aparte de los especificados por la arquitectura misma.

5.2.2 Contenido Específico del Lenguaje

Es una especificación la cual define la sintaxis para la semántica del lenguaje FIPA, esta se sugiere, aunque no es obligatorio usarlo como lenguaje de contenido para el lenguaje ACL. Dentro de este lenguaje se usa algo llamado fórmulas bien formadas, las cuales son expresiones que darán como respuesta un verdadero o un falso como valor booleano.

5.2.3 Especificación para el Soporte de Aplicaciones Nómadas

Esta especificación surge como necesidad y como resultado de la rápida evolución de la tecnología que está al alcance de la mayoría de personas en el mundo, ya que se pensó en dispositivos inalámbricos y móviles no dejando de lado que existen dispositivos con bajos recursos como un Organizador Personal o una Agenda Electrónica de Bolsillo (PDA, de su nombre en inglés Personal Digital Assistant) que también deben ser contemplados.

Para obtener el funcionamiento esperado de este estándar; el agente debe realizar varias tareas como: son la Selección de un Protocolo de Transporte de Mensajes y Conexión de Transporte de Mensajes para la comunicación entre agentes, elegir una representación para el ACL, apoyo a los agentes de aplicación para adaptar los datos y comunicación de estos agentes. El intercambio de mensajes queda representado en la figura siguiente:

A continuación, para una mejor comprensión de la figura 5.2 **[6]** se tienen las definiciones de las abreviaciones usadas en dicha figura:

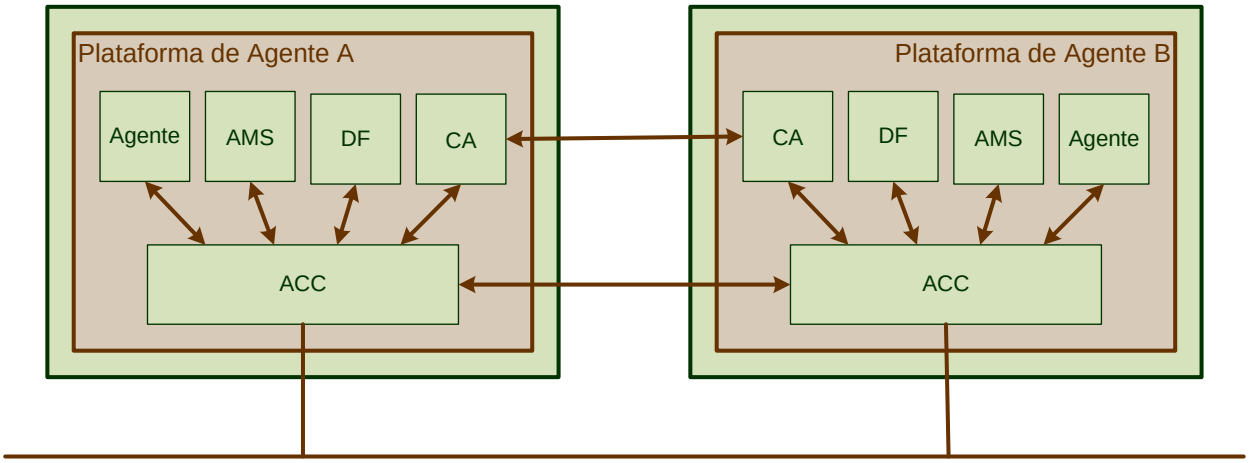

Figura 5.2: Negociación de los Agentes de Control sobre un Protocolo de Transporte de Mensajes.

AMS de sus siglas en inglés *Agent Management System* (en español significa Sistema Administrador de agentes).
DF por sus siglas en inglés *Directory Facilitator* (Directorio Facilitador).
CA de su nombre en inglés *Control Agent* (Agente de Control).
ACC por sus siglas en inglés *Agent Communication Channel* (Canal de Comunicación para los Agentes)

Tiene funciones que sirven para el monitoreo y el control de la calidad de los servicios mencionados anteriormente lo que quiere decir que hay un agente para cada uno, el cual se encargará de realizar esta evaluación (Agente de Monitoreo y Agente de Control).

Las actividades que realiza el Agente de Monitoreo son recopilar datos del rendimiento en intervalos fijos, tiene un repositorio para almacenar los datos recopilados, realiza un análisis de estos datos y finalmente los envía al Agente de Control si es que lo solicita. El Agente de Control tiene el poder de establecer, cerrar,

suspender, activar entre otras cosas a la Conexión de Transporte de Mensajes. Estos son los elementos que integran el estándar, los cuales son necesarios para tener una óptima comunicación de los agentes en entornos donde los usuarios se mueven constantemente.

5.2.4 Especificación del Administrador de Agentes

Este se encarga de proporcionar los estándares para los agentes que se encuentran dentro del sistema, se establece una relación entre el Sistema Administrador de Agentes, el cual ayuda a la Plataforma de Agentes en operaciones como: establecer la creación, registro, ubicación, comunicación, migración, suspensión, ejecución, gestión, invocar, ejecutar y así como eliminar a los agentes.

Está formada por componentes que representan las capacidades, estos son: un Identificador de Agente encargado de diferenciar a cada agente dentro del universo de agentes, un Directorio Facilitador en donde los agentes pueden registrar sus servicios o en su defecto consultar para saber cuáles servicios son ofrecidos por otros agentes, Sistema de Administración del Agente encargado de supervisar el uso y acceso a la plataforma de agentes (sólo hay uno por cada plataforma), un Servicio de Transporte de Mensajes que es el método de comunicación entre agentes de distintas plataformas, la plataforma de agentes que son todos los recursos hardware y software. Por lo tanto, la plataforma contiene todos los agentes que van a interactuar (ver figura 5.3 **[6]**).

Cabe destacar que al mencionar plataforma no quiere decir que todos los agentes estarán contenidos en ella, sino que puede haber distintas plataformas, para las cuales no es necesario que se encuentren en el mismo host. Por consiguiente, el objetivo de este administrador es definir cómo se llevará a cabo la comunicación entre los agentes. El último componente necesario para este estándar es el software, el cual viene a representar a todo código ejecutado por los agentes.

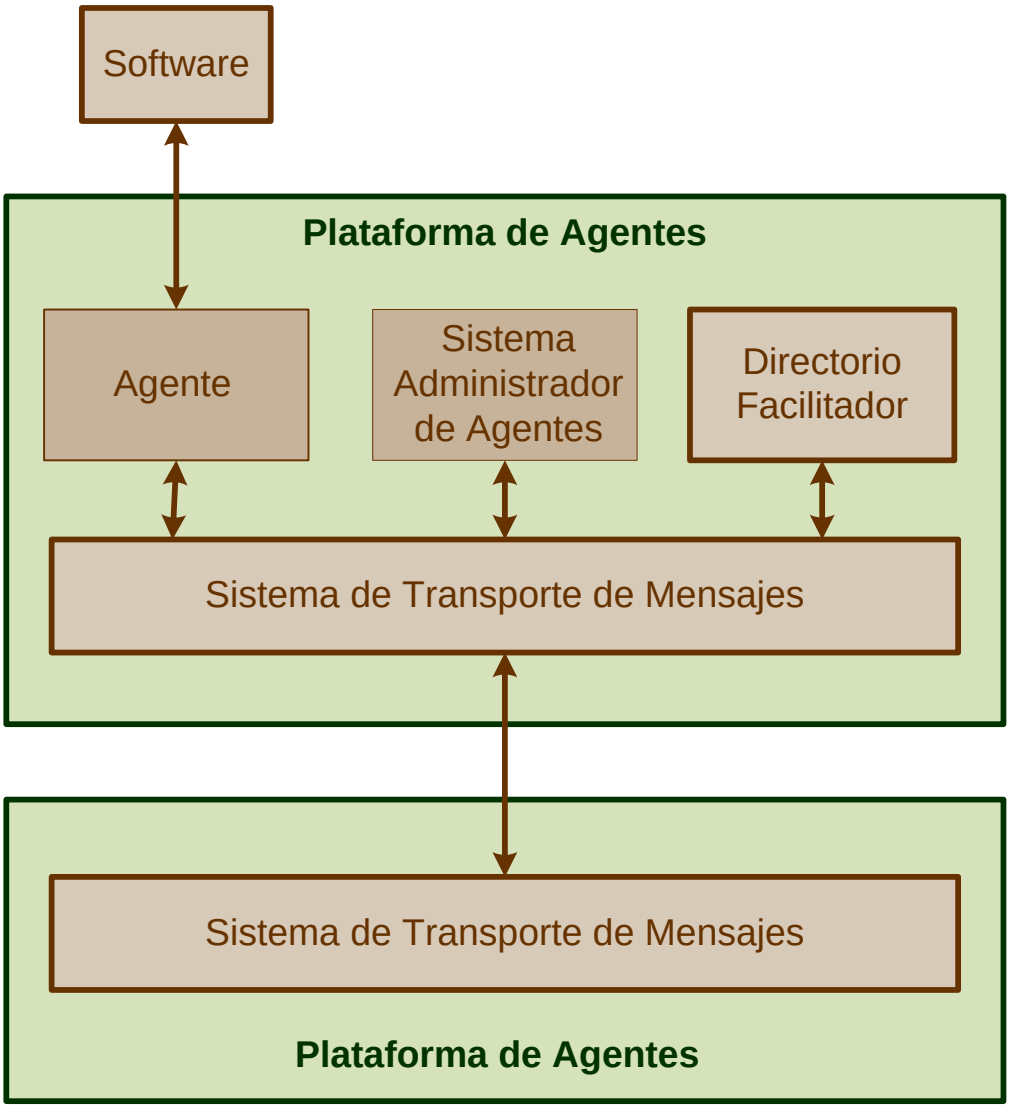

Figura 5.3: Modelo de referencia del Administrador de Agentes.

5.2.5 Especificación de la Solicitud del Protocolo de Interacción

Esta solicitud permite a un agente pedir a otro realizar una acción y funciona de la siguiente manera: el agente procesa la solicitud y decide si tomarla o dejarla, posteriormente, en caso de haber rechazado la solicitud se dice que el rechazo es verdadero por el contrario el acuerdo es aceptado y se convierte en verdadero por último los agentes deben comunicar la decisión elegida, las respuestas para esto pueden ser:

- *Falló* (por sus siglas en inglés failure): si falló al llenar la solicitud.
- Informe hecho (Inform-done de sus siglas en inglés): si se logró realizar la solicitud y sólo se quiere señalar lo que se hizo.
- *Informar resultado* (Inform-result de sus siglas en inglés): esto se realiza en caso de que se desee informar tanto lo que se hizo como la notificación de los resultados.

La figura 5.4 **[6]** es la representación por medio del Lenguaje Unificado de Modelado (UML, de su nombre en inglés Unified Modeling Language) de la solicitud del protocolo de interacción. La manera en la que se organizan las interacciones que se realizan con este protocolo es a través de un identificador de conversación, el cual no debe ser nulo. Esto es para que cada agente lleve un control sobre sus mensajes, conociendo los mensajes individuales y su historial de conversación.

5.2.6 Especificación de la Consulta del Protocolo de Interacción

Este estándar permite que un agente solicite realizar una acción en otro agente y funciona de la siguiente manera: el iniciador solicita a un participante realizar alguna acción y tiene dos tipos de solicitudes que puede hacer que son las siguientes (ver figura 5.5 **[6]**):

- *Consulta-si* (*Query–if,* de su nombre en inglés): este tipo de consulta se utiliza cuando el iniciador quiere consultar si una posición es verdadera o falsa.
- *Consulta-referencia* (*Query-ref,* de su nombre en inglés): se usa cuando el identificador necesita consultar algún objeto. Tiene el mismo mecanismo que cuando se solicita un protocolo de interacción, el cual consiste en realizar la consulta, posteriormente, se dará una respuesta de rechazo o aceptación mediante un true o un false, y finalmente, se puede dar un informe sobre lo que se hizo.

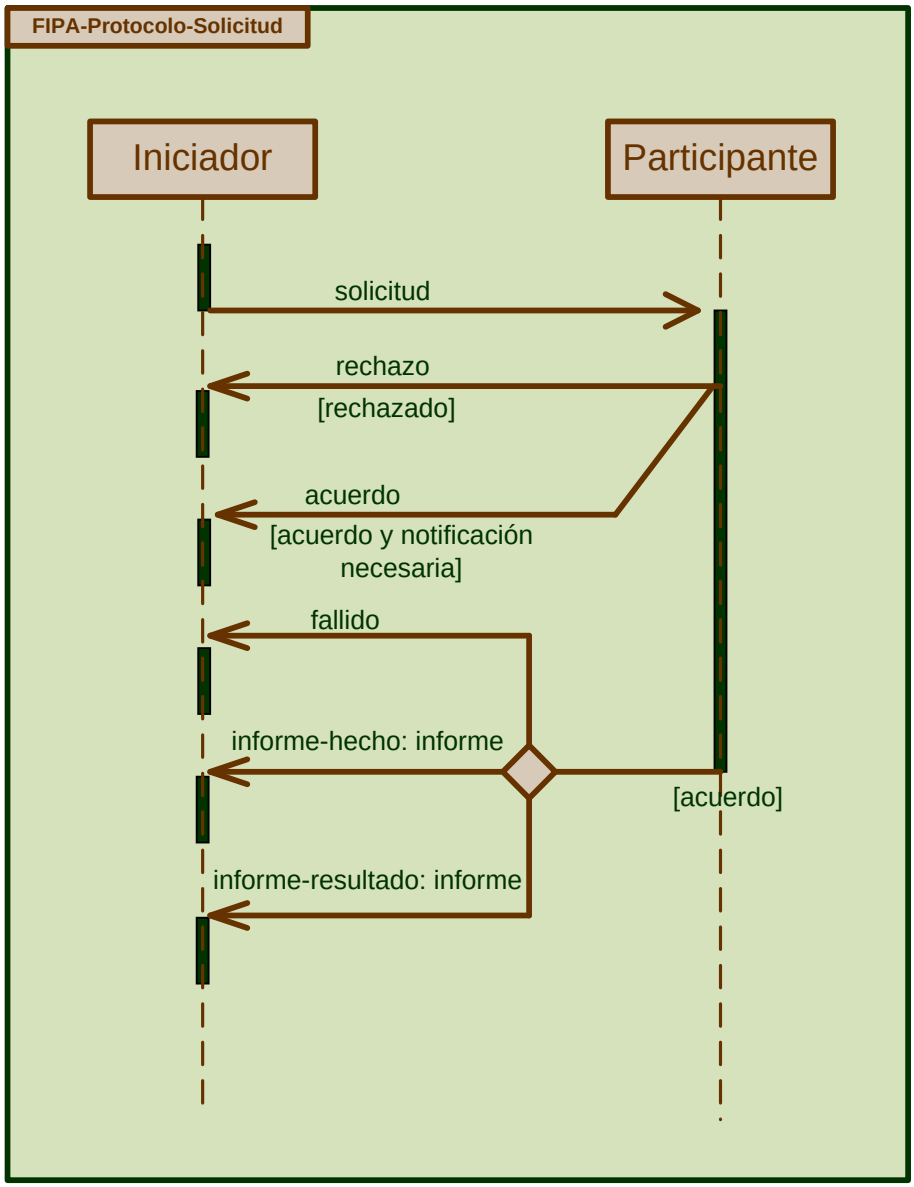

Figura 5.4: Protocolo de Interacción Cuando Solicita.

5.2.7 Especificación del Protocolo de Interacción Cuando Solicita

Esta especificación se muestra en la figura 5.6 **[6]**, la cual se encarga de las operaciones referentes a cuando un agente solicita que el receptor realice alguna acción siempre y cuando una precondición que se le da sea verdadera. La especificación funciona de la siguiente manera:

- El agente iniciador utiliza la acción request-when para solicitar realizar una acción cuando una condición se cumple.
- Si el agente solicitado no rechaza la solicitud, se espera a que la condición ocurra.
- Después cuando la condición ocurre, se intenta realizar la acción.
- Una vez realizada la acción se notifica al solicitante si esta falló o resultó exitosa.

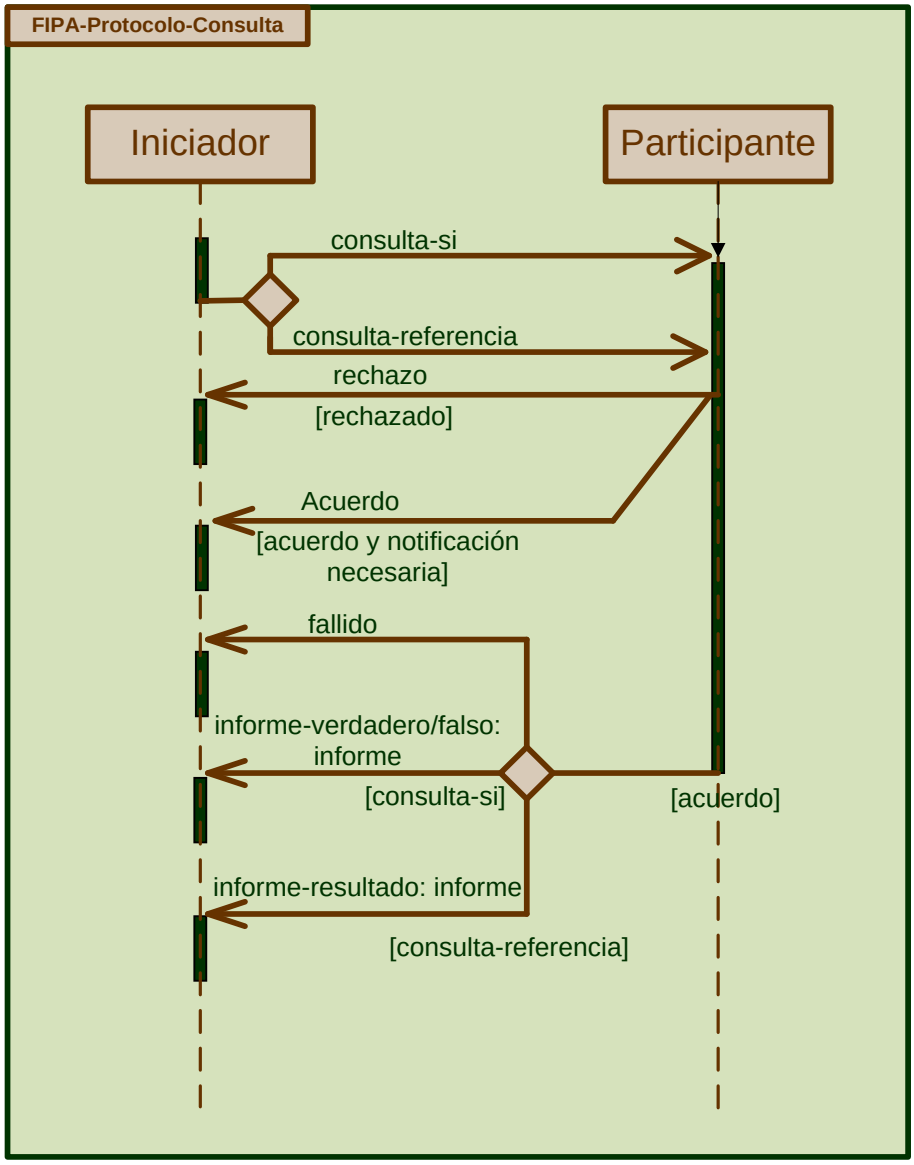

Figura 5.5: Protocolo de Interacción Cuando Consulta.

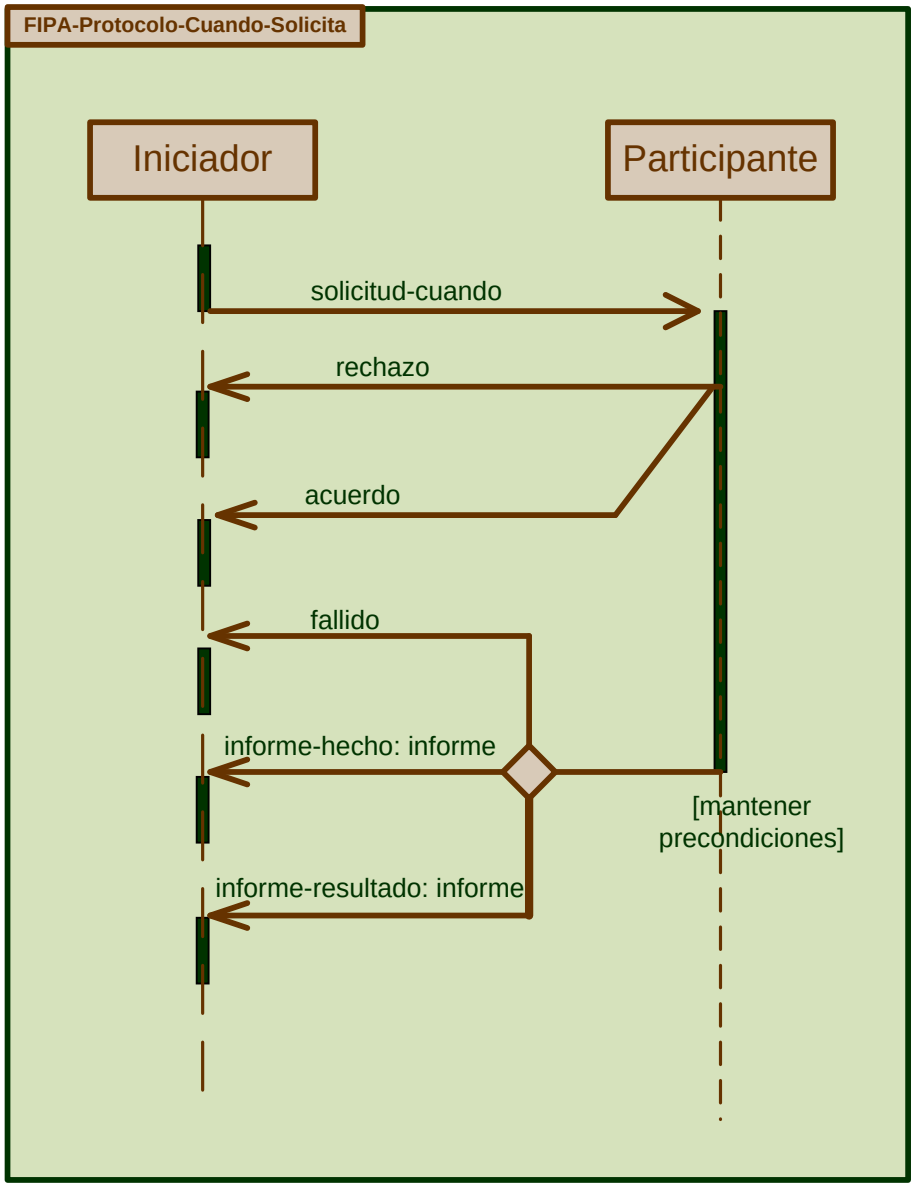

Figura 5.6: Protocolo de Interacción Cuando Solicita.

5.2.8 Especificación del Protocolo de Interacción Intermediación

Este protocolo proporciona un agente intermediario encargado de ofrecer servicios para hacer más fácil la interacción entre los agentes de la plataforma, esto es posible ya que el agente intermediario conoce los requerimientos y capacidades que tienen

los agentes. El uso de este protocolo hace más simple las tareas de interacción en las plataformas multiagente.

5.2.9 Especificación de la Biblioteca de Acto Comunicativo

El estándar determina la estructura que debe tener la biblioteca del acto comunicativo definiendo la semántica para cada acto. Tiene tres objetivos principales los cuales son enumerados a continuación:

- Asegura la interoperabilidad.
- Facilita reutilización de actos comunicativos.
- También el estándar se encarga de proporcionar un proceso para que por medio de él se le pueda dar mantenimiento a la biblioteca del acto comunicativo.

Cuando un Acto comunicativo es utilizado por un agente, también se ve obligado a usar la semántica definida para este. La sintaxis que debe cumplir el acto comunicativo es la siguiente:

- El nombre debe identificar el acto.
- No debe ser igual al de cualquier otro acto de la biblioteca.
- Debe ser una palabra en inglés o una abreviatura.

Los actos comunicativos que forman la biblioteca son los siguientes:

- *Propuesta aceptada (Accept Proposal):* es la acción de aceptar una propuesta previamente enviada para realizar una acción.
- *Acuerdo (Agree):* esta acción permite acordar realizar una posible acción en el futuro.
- *Cancelar (Cancel):* tiene la finalidad de informar que un agente ha decidido o no realizar alguna acción que ya había sido acordada.

- *Llamada de propuestas (Call for Proposal):* acción de convocar propuestas para llevar a cabo una acción.
- *Confirmar (Confirm):* en esta acción el emisor está encargado de informar al receptor si una proposición dada es verdadera.
- *No confirmado (Disconfirm):* en esta acción el emisor tiene la tarea de informar al receptor si una proposición dada es verdadera.
- *Falla (Failure):* es la acción que realiza un agente advirtiendo que una acción fue intentada pero falló.
- *Informar (Inform):* es la acción de informar que una proposición es verdadera.
- *Informe si (Inform If):* acción que toma el agente para informar al recipiente si un proposición es verdadera o no.
- *Informe Referencia (Inform Ref):* es una acción donde el remitente informa al receptor que a un objeto le corresponde una descripción.
- *No entendido (Not Understood):* es la acción que realiza el receptor de informar al emisor que no ha entendido el mensaje que le envió.
- *Propagar (Propagate):* la finalidad de esta performativa es como su nombre lo define, propagar el mensaje por los nodos, esto se realiza de la siguiente manera; el emisor envía el mensaje a un receptor definido y éste a su vez lo propaga a través de la red, con la restricción de que se propague como si este fuera enviado directamente del emisor al receptor y no de receptor a receptor.
- *Proponer (propose):* es cuando se propone una acción pero definiéndole condiciones para realizarla.
- *Proxy:* esta acción consta de que el emisor y el receptor elijan agentes que se encarguen de enviar mensajes adjuntos a ellos.
- *Consulta Si (Query*-If): es la acción que toma un agente de preguntar a otro si una proposición dada es verdadera.
- *Consulta* Ref (Query-Ref): es la acción que toma un agente de informar al agente solicitante sobre el objeto u objetos que corresponden a la descripción.
- *Rechazar (Refuse):* esta acción sirve para rechazar una acción y explicar la razón por la cual fue rechazada.

- *Rechazar la Propuesta (Reject Proposal):* es la acción de rechazar una propuesta para realizar alguna otra acción mientras se realiza una negociación.
- *Solicitud (Request):* el emisor solicita al receptor realizar alguna acción.
- *Solicitud Cuando (Request When):* el emisor solicita al receptor realizar alguna acción cuando alguna posición dada se convierte en verdadera.
- *Solicitud Siempre que (Request Whenever):* es cuando el emisor solicita al receptor realizar alguna acción, tan pronto como alguna proposición se convierta en verdadera y después de eso cada vez que la proposición sea verdadera de nuevo; como consecuencia, si la proposición se convierte en falsa, la acción se repetirá hasta que la proposición se convierta de nuevo en verdadera.
- *Suscribe (Subscribe):* es la acción de solicitar algo, teniendo que notificar al emisor el valor de una referencia para notificar siempre que el objeto identificado por la referencia cambie.

Estos actos comunicativos son algunos de los especificados por FIPA y que ayudan a la comunicación de los agentes, no es obligatorio usarlos, ya que el diseñador puede generar su propia librería y esto funciona como lo muestra la figura 5.7 **[6]**.

5.2.10 Especificación de la Estructura del Mensaje ACL

Esta especificación se encarga de la estructura de los mensajes ACL, es decir, define los parámetros contenidos en los mensajes entre agentes y de esta manera se pueda realizar la comunicación de manera clara y precisa, si en un momento el agente no reconoce o no procesa los parámetros que son enviados dentro de esta estructura, se tiene la opción de responder con el mensaje de *no-understood*. La estructura de los mensajes es la siguiente:

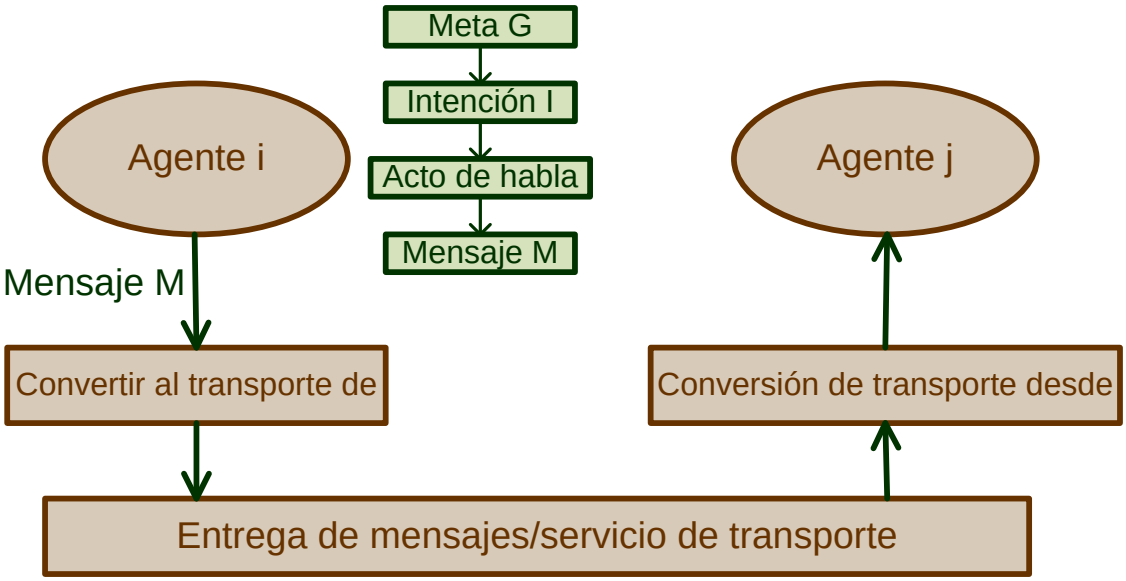

Figura 5.7: Intercambio de Mensajes entre dos Agentes.

- *Frame:* nombre de la entidad que representa una instancia.
- *Ontología:* es el nombre de la base de conocimiento.
- *Parámetros:* algunos de ellos pueden ser emisor, receptor, protocolo, identificador de conversación, respuesta a, entre otros parámetros especificados por el acto comunicativo.
- *Descripción:* explicación en lenguaje natural de las semánticas.
- *Valores reservados:* valores constantes asociados a los parámetros.
- *Tipos de acto comunicativo:* performativo como tipo de acción comunicativa.
- Participantes en la comunicación: emisor, receptor o responder a.

5.2.11 Especificación del Servicio de Transporte de Mensajes para Agentes

Este estándar define un modelo para el transporte de mensajes entre agentes, dicho modelo está formado por tres niveles que son los siguientes:

- *Protocolo de Transporte de Mensajes (MTP, de su nombre en inglés Message Transport Protocol):* realiza la transferencia de mensajes entre dos Canales de Comunicación entre Agentes (ACC).
- *Servicio de Transporte de Mensajes (MTS, de su nombre en inglés Message Transport Service):* este servicio lo ofrece la Plataforma de Agentes, el cual tiene la función de transportar mensajes ACL ya sea dentro de la misma Plataforma de Agentes o diferentes Plataformas de Agentes.
- *El Lenguaje de Comunicación para Agentes (ACL, de su nombre en inglés Agent Communication Language):* éste es la representación del contenido de los mensajes que serán transmitidos por el agente a través del MTP y el MTS.

Los niveles anteriormente mencionados se pueden ver en la figura 5.8 **[6]**:

Este servicio de transporte puede funcionar de tres maneras:

- La primera, es que el agente emisor envíe el mensaje a su ACC local para que posteriormente éste envíe el mensaje al ACC correcto, una vez que lo reciba lo entregará al receptor.
- La segunda, es que el agente emisor envíe el mensaje directamente al ACC de la plataforma remota en donde se localiza el agente receptor.
- La tercera y última, es que el agente emisor envía directamente el mensaje al agente receptor, pero para ello será necesario el uso de un mecanismo de comunicación directo. Este modo de comunicación no lo proporciona FIPA.

5.2.12 Especificaciones Generales

Aparte de las especificaciones mencionadas anteriormente, existen más estándares que ayudan a la interoperabilidad entre los agentes ya sea a administrar tareas y agentes, optimizar funciones, gestionar acciones, proporcionar directorios de servicios o proponer acciones para realizar; estas tareas son posibles gracias a los siguientes estándares:

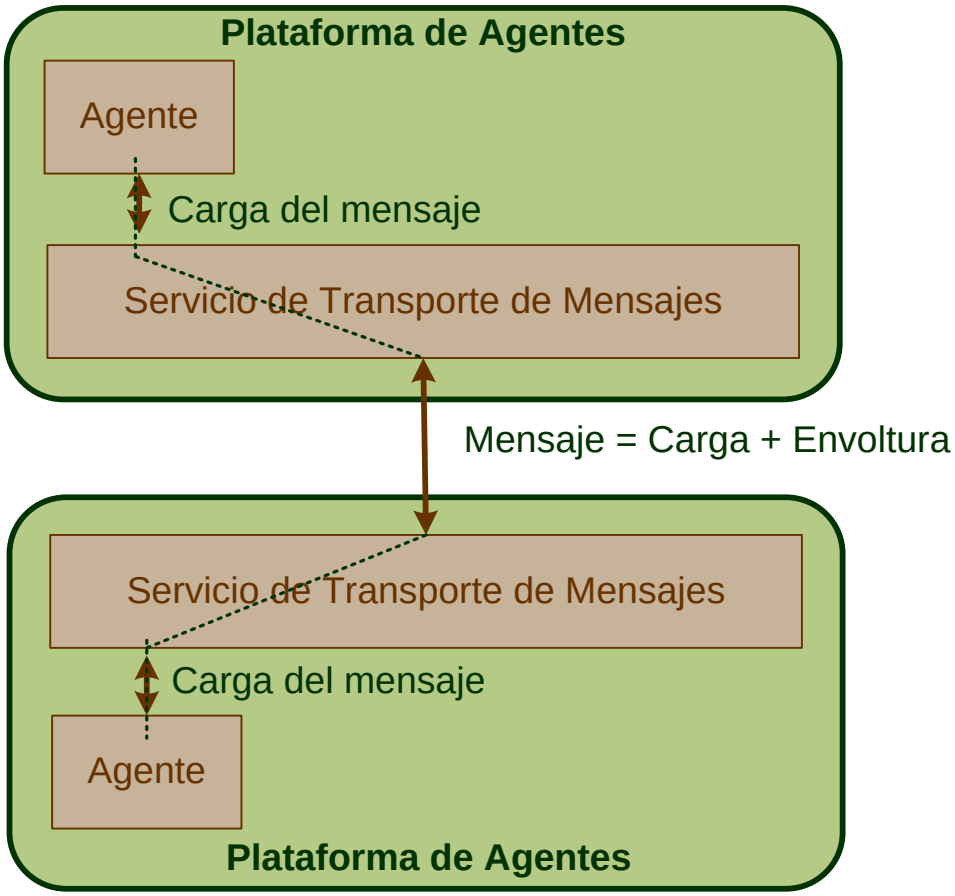

Figura 5.8: Modelo de referencia del Transporte de Mensajes.

- Especificación del Contrato Net del Protocolo de Interacción.
- Especificación de la Interacción del Contrato Net del Protocolo de Interacción
- Especificación del protocolo de interacción FIPA recluta.
- Especificación del protocolo de interacción FIPA subscribe.
- Especificación del protocolo de interacción FIPA.

Todos los protocolos mencionados en este capítulo son usados por la mayoría de arquitecturas multiagente para lograr la interoperabilidad de estos junto con los recursos y servicios necesarios para brindar transparencia y seguridad en la interfaz con la que va a interactuar el usuario final.

Capítulo 6. Una Arquitectura Heterogénea para Aplicaciones Multiagente en Java

El contenido de este capítulo, se enfoca en mostrar y explicar de manera clara y detallada el desarrollo de cada una de las partes que forman la arquitectura de agentes propuesta. Además, presenta los casos de estudio que se implementaron para la validación y el conocimiento respecto a cómo se debe usar la misma en el desarrollo de sistemas multiagente. De este modo, se puede verificar que los objetivos planteados en la sección 1.4 se cumplen de manera satisfactoria.

6.1 Propuesta

En la figura 6.1, se puede apreciar la estructura de la arquitectura que se propone en este trabajo de tesis. Para contar con un mejor entendimiento de lo que representa esta figura, se deben conocer y comprender los siguientes aspectos:

- Como primer punto, tenemos un conjunto de máquinas en red, las cuales es posible que cuenten con sistemas operativos diferentes. Cabe mencionar, que el hecho anterior no afecta a la arquitectura ya que el lenguaje que se utilizó es Java y por naturaleza es multiplataforma, lo que permitirá la ejecución de agentes en cualquier sistema operativo sin provocar algún conflicto, es decir, no se presenten situaciones de incompatibilidad.
- Como segundo punto se tiene la aplicación que será ejecutada en las máquinas, esta aplicación tiene de manera abstracta un repositorio de agentes donde se almacenarán todos los agentes, quienes tomarán decisiones dependiendo de las variables que tengan como podrían ser: el entorno, sus percepciones, su estado interno si es que lo tiene, objetivos, metas, creencias, etc.

- Finalmente, los agentes dentro del repositorio estarán formados principalmente por sensores, efectores y un conjunto de acciones. Dado que la arquitectura es de tipo P2P, los agentes podrán tomar en cualquier momento el papel de clientes (solicitantes), o en su defecto, de servidores (prestadores de servicios), es decir, según sea conveniente.

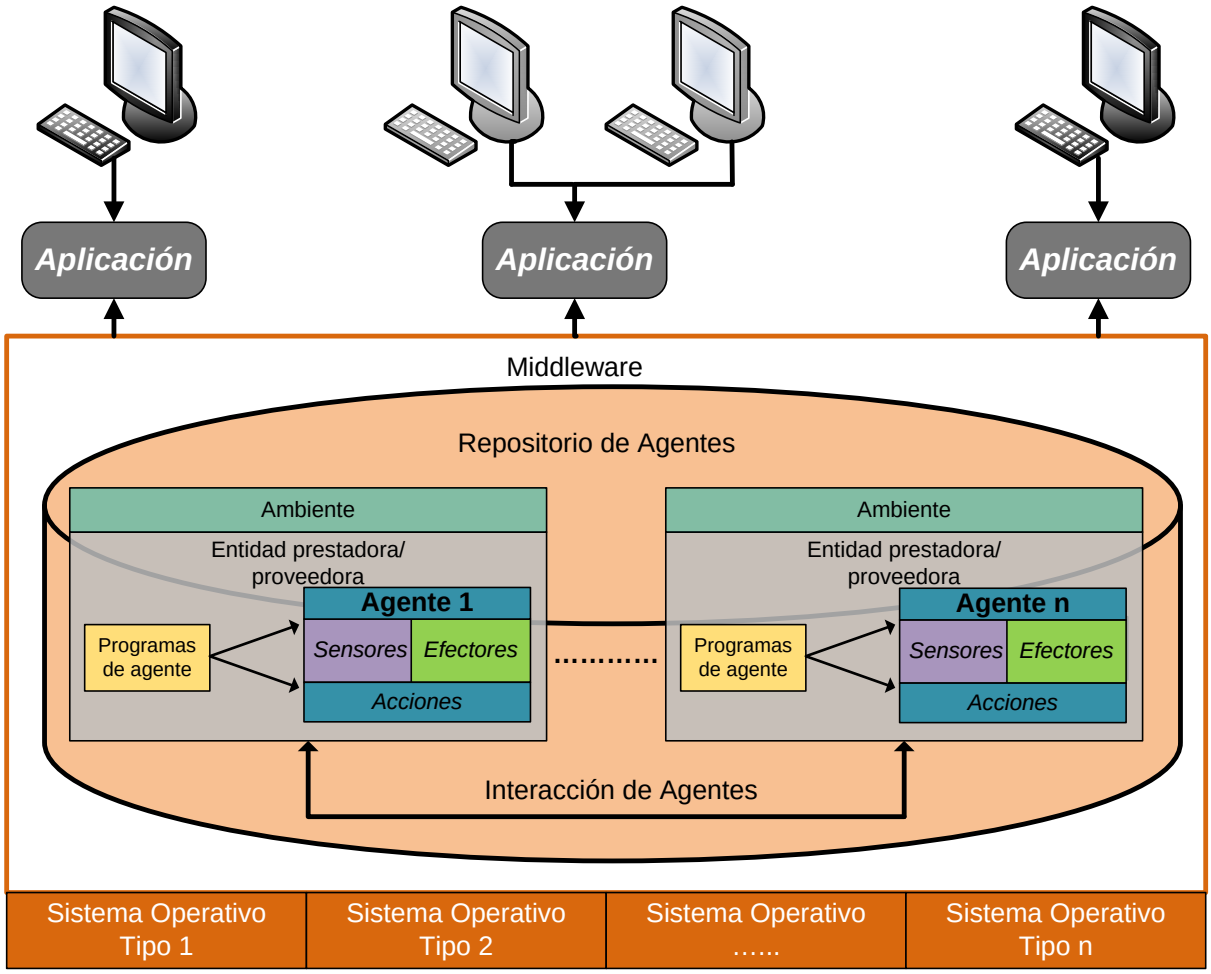

Figura 6.1: Propuesta de la Arquitectura.

6.2 Jerarquía de Agentes

En las figuras 6.2 y 6.3, es posible visualizar la manera en la que los agentes están organizados jerárquicamente dependiendo de las características que tiene cada tipo,

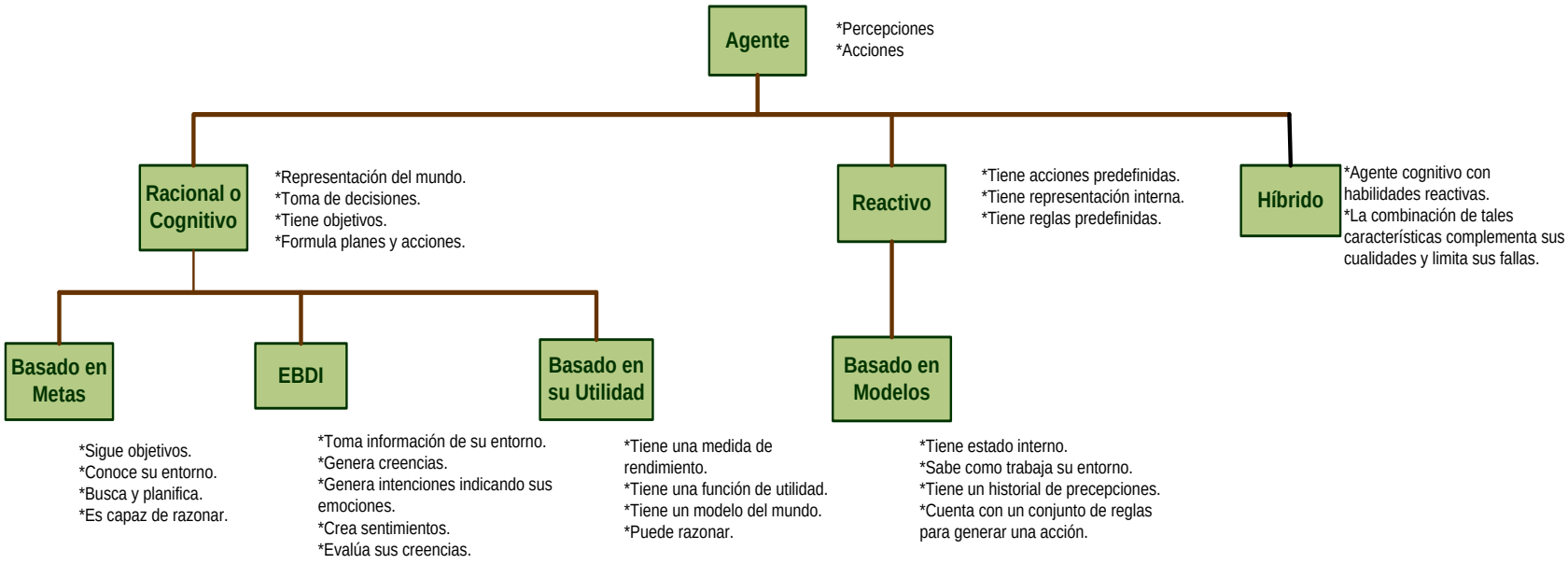

Figura 6.2: Jerarquía de Agentes.

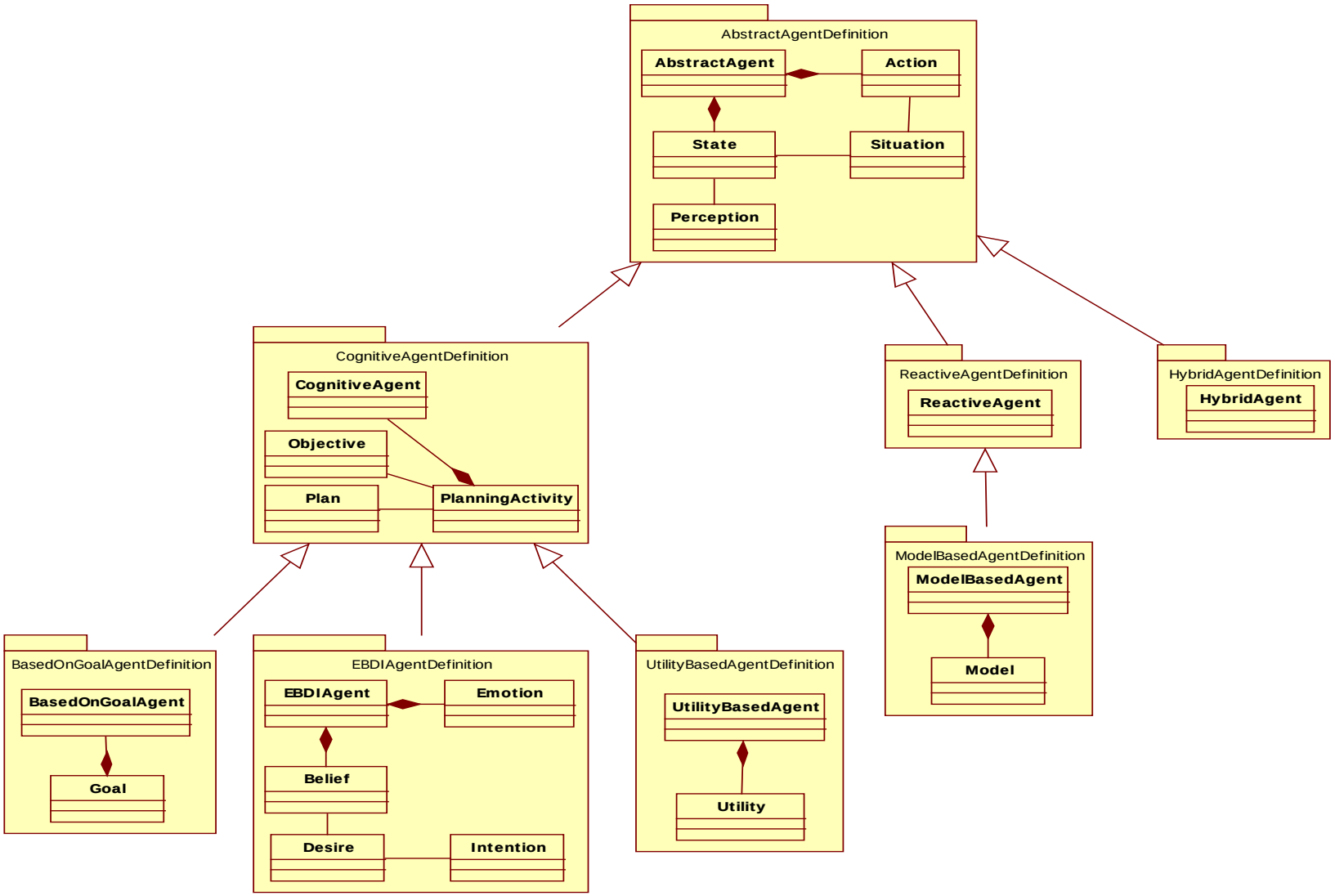

Figura 6.3: Relación de los paquetes que forman la Arquitectura de Agentes.

iniciando en el primer nivel con el *Agente Abstracto* que representa las características generales con las cuales debe contar todo agente; éstas, principalmente son: que debe ser capaz de percibir su entorno y en base a esto ejecutar acciones en el mismo. En el segundo nivel, se tienen los siguientes tres tipos de agentes:

- *Agente Cognitivo:* el cual tiene representación del ambiente, cumple sus objetivos y ejecuta planes para lograrlos.
- *Agente R*eactivo: tiene acciones y reglas predefinidas así como una representación interna del ambiente.
- *Agente Híbrido:* que está formado por características tanto cognitivas como reactivas.

En el tercer nivel, se encuentran los siguientes tipos de agentes:

- *Agente Basado en Metas:* el cual tiene las mismas características que el cognitivo con la diferencia de que este es capaz de razonar.
- *Agente Basado en su Utilidad:* también comparte las mismas características que el cognitivo, agregando que este tiene una medida de rendimiento para valorar los estados, una función de utilidad y también puede razonar.
- *Agente Basado en Modelos*: este tiene las mismas características que el reactivo con la diferencia de que este primero cuenta con un historial de percepciones y sabe cómo trabaja su entorno.
- *Agente EDBI:* agrega las habilidades de generar creencias, deseos, intenciones y todo esto es influenciado por sus emociones.

6.3 Diagramas de Clases

A continuación, se puede visualizar por medio de una serie de diagramas de clases los cuales fueron modelados con ayuda de la herramienta StarUML **[36],** cada una de las partes que forman la arquitectura de agentes, así como las relaciones con sus

características que darán forma al comportamiento dependiendo del tipo de agente al que pertenecen.

Para cada agente, sus características también son representadas como clases con el fin de implementar el paradigma orientado a objetos el cual es la reutilización de código, esto es porque existe un atributo que posiblemente tenga que ser utilizado por más de un tipo de agente por ejemplo; las acciones son usadas por las clases Situation y AbstractAgent; así la clase Action podrá ser relacionada con estas dos clases sin necesidad de volver a escribir el código.

En seguida, se desglosa detalladamente cada una de las clases (agentes y atributos) junto con sus características y la función que tiene cada una dentro de la arquitectura, la cual especifica el comportamiento de cada agente dependiendo del tipo que sea, estos son: Abstracto, Cognitivo, Basado en Metas, Basado en su Utilidad, EBDI, Reactivo, Basado en Modelos e Híbrido.

6.3.1 Clase AbstractAgent

Su finalidad es tener una clase que contenga las características básicas que debe tener todo agente independientemente del tipo, estas características son: un método setup() encargado de poner al agente en estado vivo, iniciando así la ejecución de sus comportamientos y acciones por medio de un hilo de control ya que de esta manera será posible la ejecución de los agentes en diferentes máquinas de forma remota o distribuida y de forma local o concurrente (ver figura 6.4).

Existen otros métodos que forman el ciclo de vida del agente, éstos fueron creados en base a como lo determina FIPA y son: doActivate(), consiste en cambiar el estado de espera o suspendido del agente por activo; doSuspend(), encargado de cambiar un estado de activo o en espera a suspendido; doWait(), el cual tiene dos opciones esperar un intervalo de tiempo o esperar por tiempo indefinido; doWake(), en caso de

haber estado en espera o suspendido; takeDown() para terminar un agente que fue creado y finalmente doDelete() un agente de la lista.

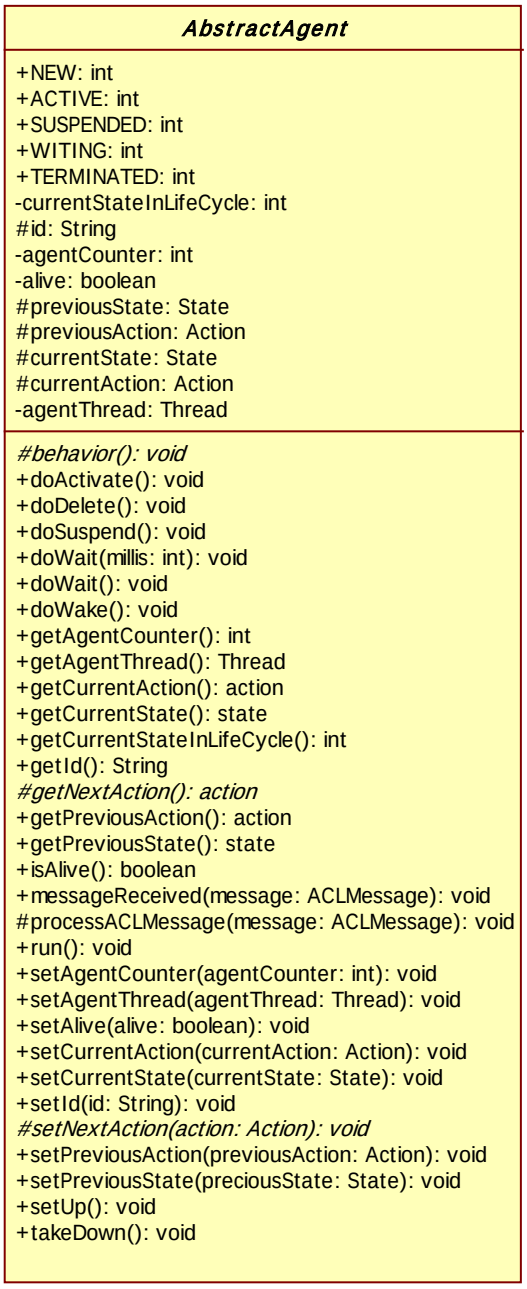

Figura 6.4: Clase AbstractAgent.

El resto de los métodos consiste en un conjunto de acciones y estados que determina la manera que agente deberá ejecutarlos, señala tanto la acción como el

estado anterior, actual y siguiente completando de esta manera las características generales que deben cumplir todos los agentes.

En la figura 6.5 es posible ver la descripción de la clase dada en los párrafos anteriores, ésta contiene las relaciones de la clase AbstractAgent con sus características que son: Action, Perception, Situation y State. Como primer término tenemos la clase Action formada por un número de identificación y nombre; en segundo término está la clase Perception que almacena datos percibidos del ambiente o del agente que corresponden a un nombre de percepción y su valor percibido.

Finalmente, está la clase Situation encargada de como su nombre lo dice registrar una situación en la que se encuentra involucrado un agente, las partes que lo forman son un estado y un conjunto de acciones. El décimo y doceavo atributos son de tipo la clase State quien formará los posibles estados en los que podría estar un agente, integrados por una lista de percepciones.

6.3.2 Clase CognitiveAgent

Ahora bien, como se puede ver en la figura 6.6 el CognitiveAgent es una clase que se forma de objetos con características de un agente racional por medio de sus constructores. En primer lugar tiene una representación de su entorno, esto es posible gracias a que lo hereda de la clase AbstractAgent a través del estado formado por percepciones, posteriormente se tiene su principal característica que es planear a largo plazo; esto lo hace por medio de la clase PlanningActivity (en la figura 6.6 se podrá ver con más detalle sobre las características de esta clase), una vez que formula el plan deberá seguirlo para lograr su objetivo.

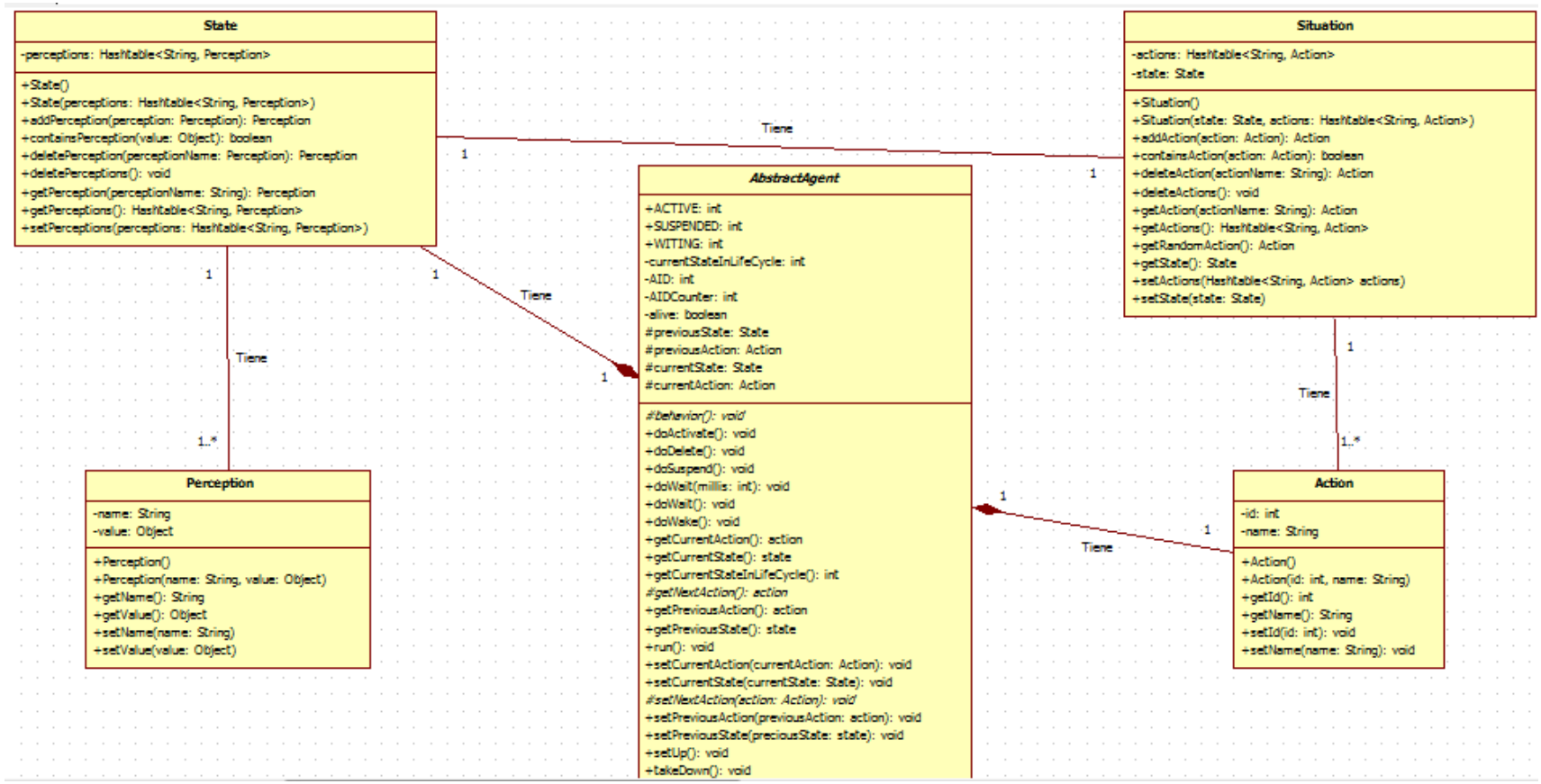

Figura 6.5: Clase AbstractAgent y las relaciones con sus atributos.

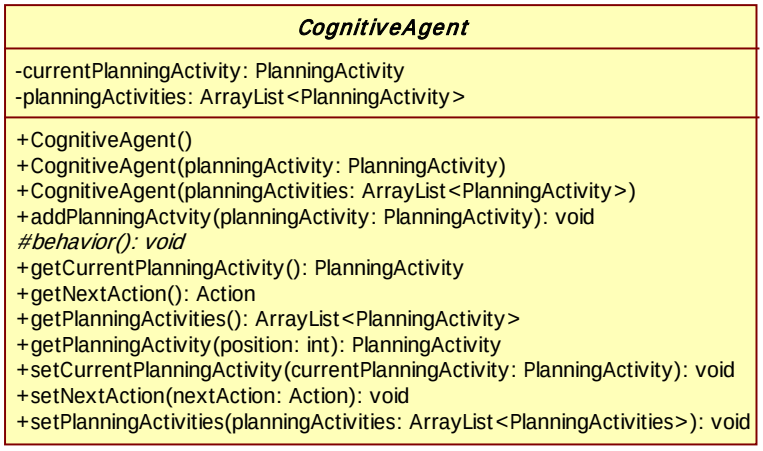

Figura 6.6: Clase CognitiveAgent.

Del mismo modo, en la siguiente figura se pueden visualizar los atributos que forman a la clase CognitiveAgent, generados como clases debido a que será necesario reutilizar este código para otros agentes que también requieren de una actividad de planeación, evitando de esta manera volver a escribir el mismo código.

En consecuencia, la clase PlanningActivity como se puede ver en la figura 6.7 está integrada por una lista de planes y una lista de objetivos que se relacionan entre sí, también debe ser necesario especificar cuál es el plan actual que está ejecuntado el agente. A su vez una clase Plan es una secuencia de acciones donde se debe definir cuál será la primera y cuál la segunda actividad. Finalmente, PlanningActivity se relaciona con la clase Objective donde registra las características de un objetivo que son: identificador, nombre y la definición del objetivo.

6.3.3 Clase BasedOnGoalAgent

Con respecto a este tipo de agente, la característica que destaca es como su nombre lo define: persigue una o varias metas, lo que quiere decir que las decisiones y acciones que tome dependerán de la meta. Esto da como resultado, que el estado actual heredado de la clase CognitiveAgent no será suficiente, por lo tanto a cada

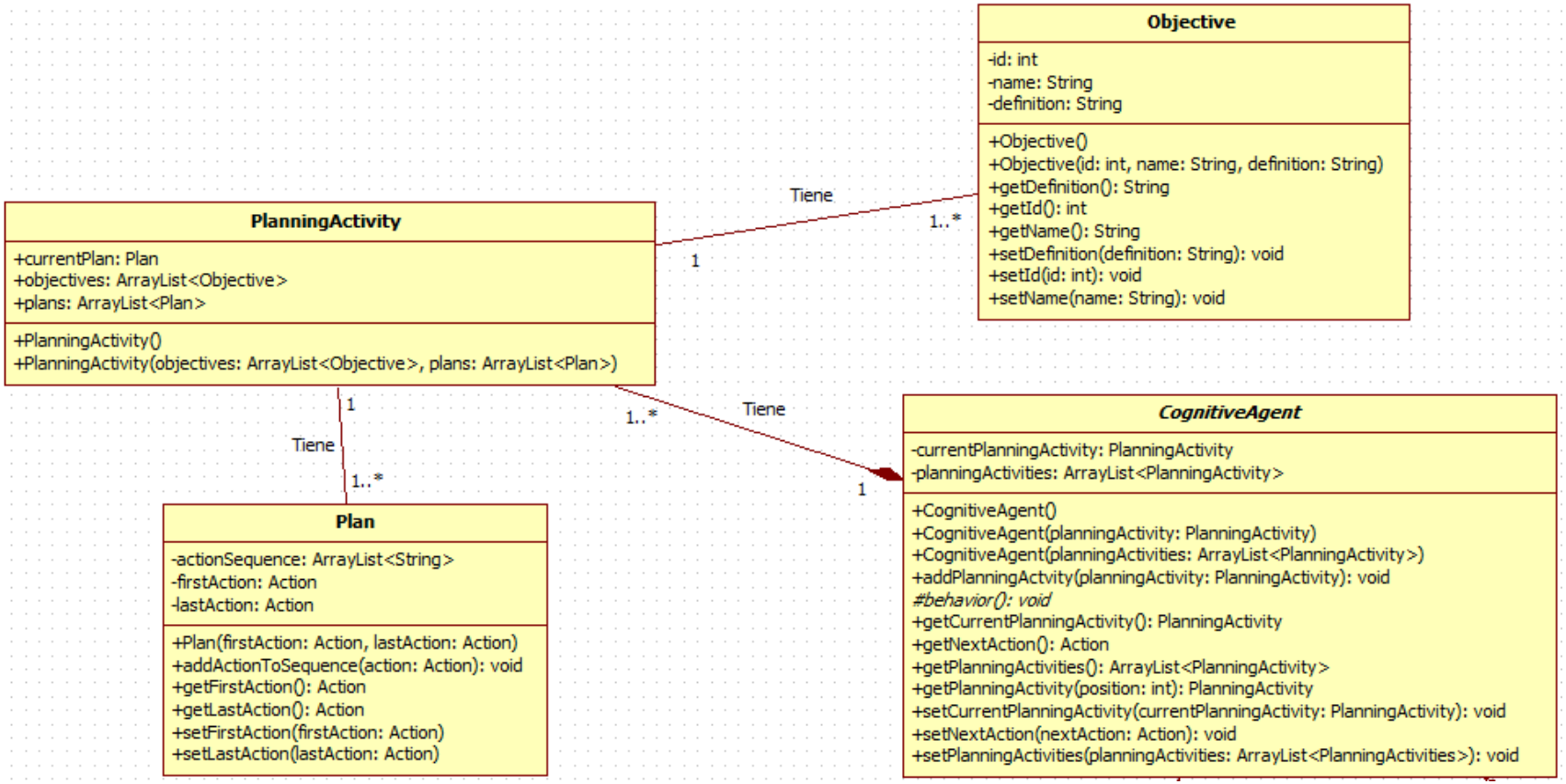

Figura 6.7: Relación de la Clase CognitiveAgent con sus atributos.

actividad de planeación se le asignará una o varias metas, para que el agente cumpla con lo que desea.

En la figura 6.8 se muestra la clase encargada de realizar lo mencionado en el párrafo anterior, como se puede ver almacena en una lista las metas que persigue, especificando la meta que se ejecuta actualmente.

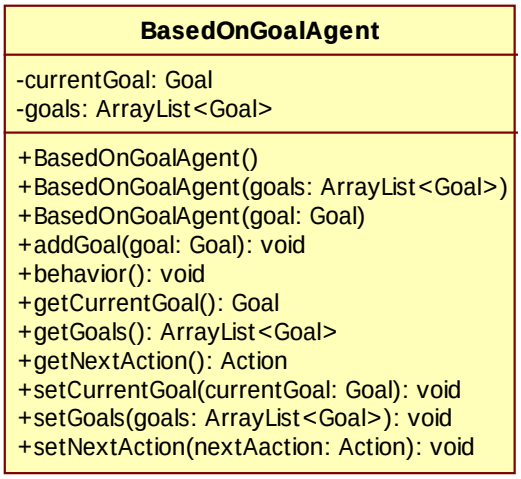

Figura 6.8: Clase BasedOnGoalAgent.

Con respecto a lo explicado sobre la clase BasedOnGoalAgent; a continuación, en la figura 6.9 es posible ver cómo están compuestas las metas de las que se habló anteriormente, las cuales están relacionadas directamente con el agente ya que es su característica primordial. La clase Goal está formada por el nombre de la misma y la actividad de planeación que le corresponde a esa meta, recordando que una actividad de planeación está compuesta por planes y objetivos, los cuales serán pasados a la clase PlanningActivity por medio del constructor de la clase. Hay que destacar que la clase Goal es: un conjunto de planes y objetivos que se deben cumplir para lograr un fin perseguido.

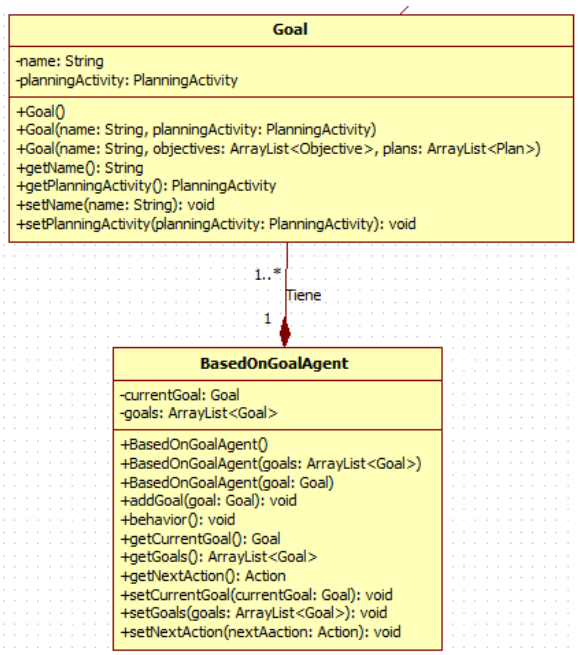

Figura 6.9: Clase BasedOnGoalAgent y las relaciones con sus atributos.

6.3.4 Clase EBDIAgent

Ahora, toca el turno de la clase EBDIAgent la cual tiene características muy particulares, un tanto más complicadas puesto que éste no sólo tiene información de su entorno, sus acciones y metas; sino que también tiene emociones y creencias influenciadas por lo que desea e intenta hacer.

Como se puede apreciar en la figura 6.10 un agente de este tipo va a registrar el conjunto de creencias y deseos que tiene por medio de una lista, evaluando constantemente esas creencias que ha formado; por consiguiente, las acciones que se realicen como producto de las creencias darán emociones al agente.

Ahora bien, en la figura 6.11 se puede apreciar claramente cómo es que una creencia del agente va a desencadenar un deseo, que a su vez es influenciado por una intención. En principio el agente toma información de su entorno por medio de los sensores, una vez que se ha registrado todo lo que fue posible percibir, el agente

va a generar creencias. Para ello, se tiene la clase Belief donde registra un identificador, nombre y el valor de la creencia, la cual puede ser verdadera o falsa y una lista de deseos que genera el agente en base a esta creencia.

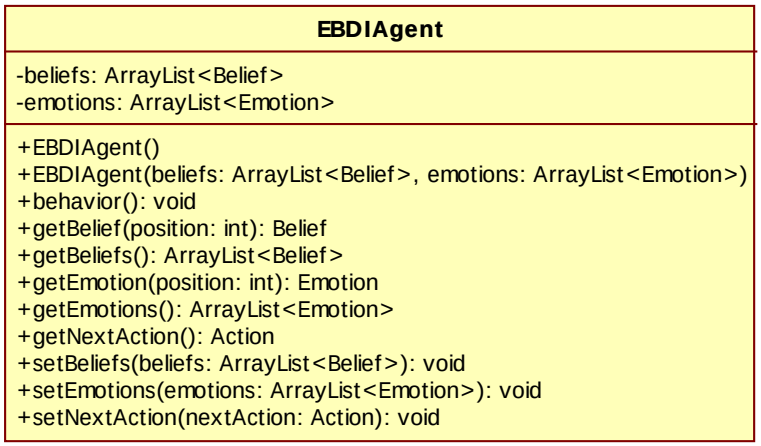

Figura 6.10: Clase EBDIAgent.

Posteriormente, se tiene la clase Desire, donde el agente registra también un identificador y nombre de lo que desea, junto con una lista de las intenciones que tiene para con este deseo. Finalmente, se tiene la clase Intention la cual está dirigida por metas que debe cumplir el agente.

6.3.5 Clase UtilityBasedAgent

Contrario al Agente Basado en Metas que solo se enfoca en cumplir una meta, el Agente Basado en Utilidad implementa una medida de rendimiento, con esta permite saber si el agente está satisfecho con las acciones que se le han asignado para realizar. En la figura 6.12 se puede ver que el constructor de esta clase va a requerir una lista de planeación de actividades, (vale la pena recordar que una planeación de actividades está compuesta por planes y objetivos) con la finalidad de que se sigan una serie de planes y de esta manera cumplir uno o varios objetivos.

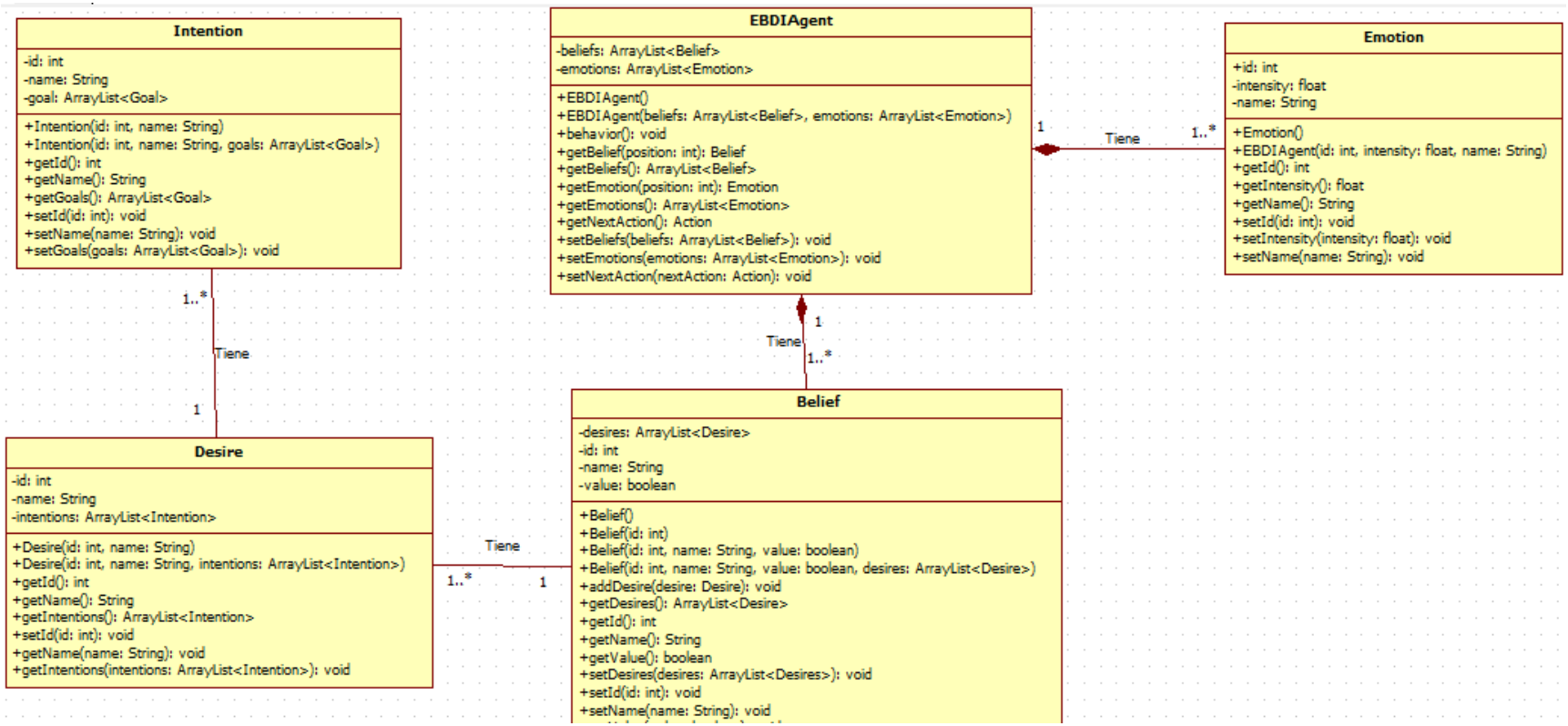

Figura 6.11: Clase EBDIAgent y las relaciones con sus atributos.

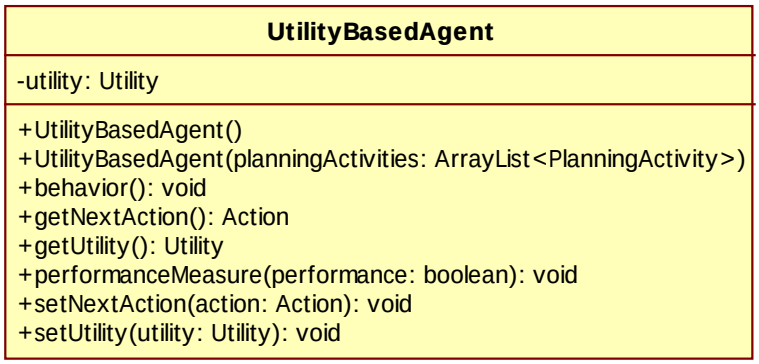

Figura 6.12: Clase UtilityBasedAgent.

Ahora bien, no sólo es importante cumplir los objetivos sino también tener conocimiento sobre el beneficio de elegir ciertas acciones; es por ello, que esta clase tiene un método para valorar la acción que se desea agregar; es decir si la medida de rendimiento para esta acción fue buena se asignará un true y como consecuencia la acción será agregada en la lista de acciones de la utilidad, pues quiere decir que satisface las necesidades del agente.

A continuación, se revisará la manera en que trabaja el atributo utility (ver figura 6.13), donde es posible ver que se encuentra directamente relacionado el agente con su atributo (es decir la clase Utility), el cual almacenará los resultados de la medida de rendimiento que realizó el método performanceMeasure(performance: boolean) en la clase UtilityAgent, esto es debido a la acción que haya sido calificada por el agente como buena; será agregada a la lista de acciones de la utilidad y se va a premiar con puntos positivos a dicha acción para que así el agente confirme la utilidad de la acción que realizó.

Para finalizar, la medida de rendimiento se encarga de determinar la utilidad para una secuencia de estados y así identificar las maneras convenientes para cumplir el objetivo.

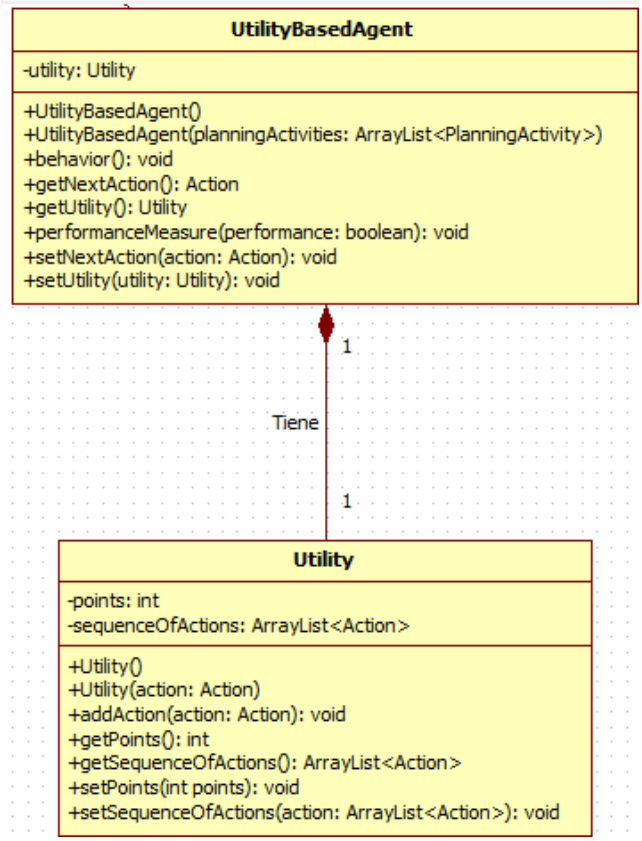

Figura 6.13: Clase UtilityBasedAgent y las relaciones con sus atributos.

6.3.6 Clase ReactiveAgent

El objetivo principal de clase ReactiveAgent es realizar acciones predefinidas para cada percepción, este agente es muy sencillo, por lo cual no fue necesario agregar atributos extra a los que ha heredado de su súper clase.

Como se puede ver en la figura 6.14 el agente reactivo se va a encargar de registrar sus percepciones detectadas por sus sensores, lo que dará como resultado que el agente ejecute una acción en base al estado en el que se encuentre y como resultado de esto se obtendrá una situación. De esta forma nos damos cuenta que va a ir guardando los estados y situaciones actuales de los cuales tendrá la oportunidad de elegir la acción que realizó en caso de volverse a presentar una percepción anteriormente detectada.

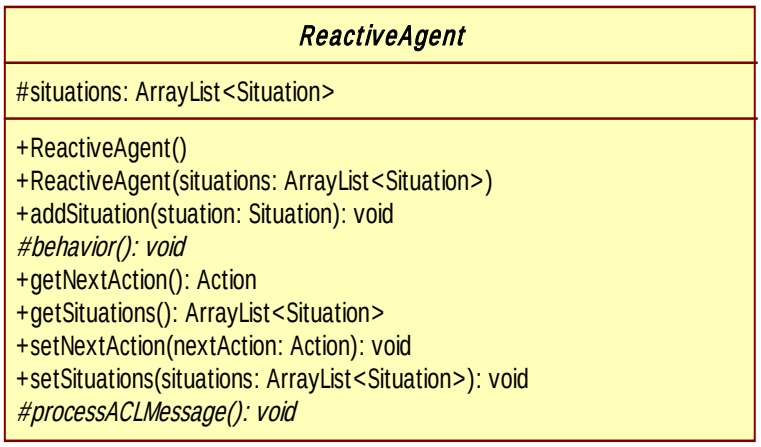

Figura 6.14: Clase ReactiveAgent.

6.3.7 Clase ModelBasedAgent

Es preciso recordar que este agente tiene un estado interno (ver figura 6.15), que se va formando gracias a la lista de estados y de esta manera se tiene una visión respecto a lo que no pueda ser percibido en ese momento junto con la lista de situaciones que registra las acciones correspondientes a cada percepción.

Esto es posible mediante la actualización de la información del estado interno, para ello se tiene el método getActionFromEvolution(), encargado de proporcionar la acción a ejecutar correspondiente al estado actual teniendo en cuenta que a esto se le conoce como modelo del mundo (ver figura 6.16).

Otra parte fundamental para este tipo de agente es cuando se actualiza su estado, en la arquitectura lo denominamos estado resultante, como se puede ver en la figura de abajo la clase Model relacionada directamente con el agente, registra el estado percibido y el estado resultante una vez que se haya puesto en marcha la acción.

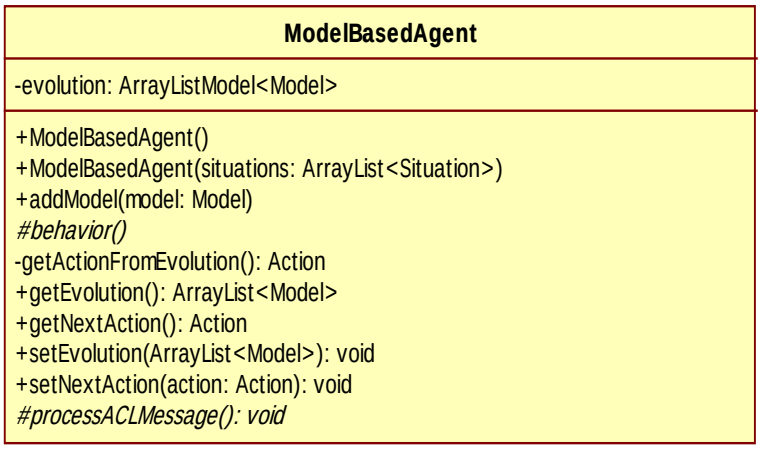

Figura 6.15: Clase ModelBasedAgent.

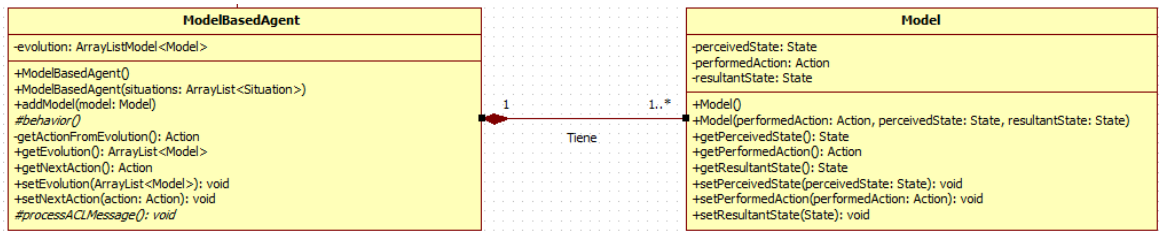

Figura 6.16: Clase ModelBasedAgent y las relaciones con sus atributos.

6.3.8 Clase HybridAgent

Antes que nada, vale la pena recordar que un Agente Híbrido es un agente cognitivo con habilidades reactivas, de manera que para crear agentes con estas características se decidió generar una lista de actividades de planeación donde como se comentó en otras clases está formada por planes y objetivos, cumpliendo con ello su parte cognitiva, por otro lado con la creación de una lista de situaciones se está cumpliendo la parte reactiva donde el tener una situación quiere decir que ya se tienen definidos estados y acciones que se llevarán a cabo en caso de haber percibido algo el agente.

La parte reactiva también se ve implementada en el método getNextActionForCurrentSituation() ya que busca si una situación existe, si es así regresará la acción a realizar, de lo contario no regresa nada porque quiere decir que el agente la desconoce. Para finalizar la clase de la figura 6.17 representa como está integrado el agente descrito en el párrafo anterior.

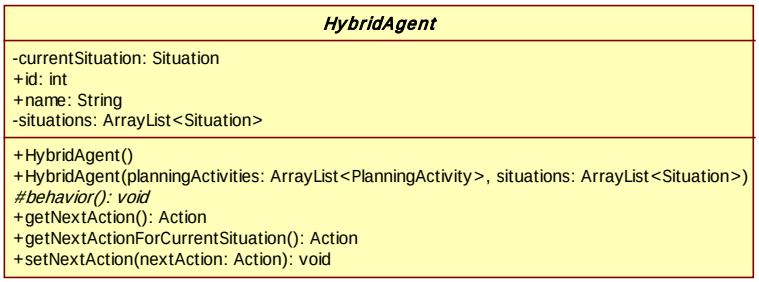

Figura 6.17: Clase HybridAgent.

6.4 Diagramas de Clases que cumplen lo establecido por FIPA Y W3C

En este apartado se revisarán las clases que formarán lo especificado por los estándares FIPA y W3C, así como también los atributos y comportamientos que integran cada una de estas clases para que se pueda lograr la creación e interacción de los agentes.

6.4.1 Clase ACLMessage

Esta es la clase encargada de formar los mensajes de acuerdo a la estructura que determina FIPA la cual incluye los siguientes parámetros como obligatorios: emisor, receptor y performativa (ver figura 6.18). Como se puede ver en el diagrama, el constructor ACLMessage() es el encargado de recibir los parámetros de las instancias creadas a partir de esta clase.

Los demás parámetros referentes a la estructura del mensaje son opcionales, por lo tanto si es necesario agregar uno o saber lo que contiene, solo será posible a través

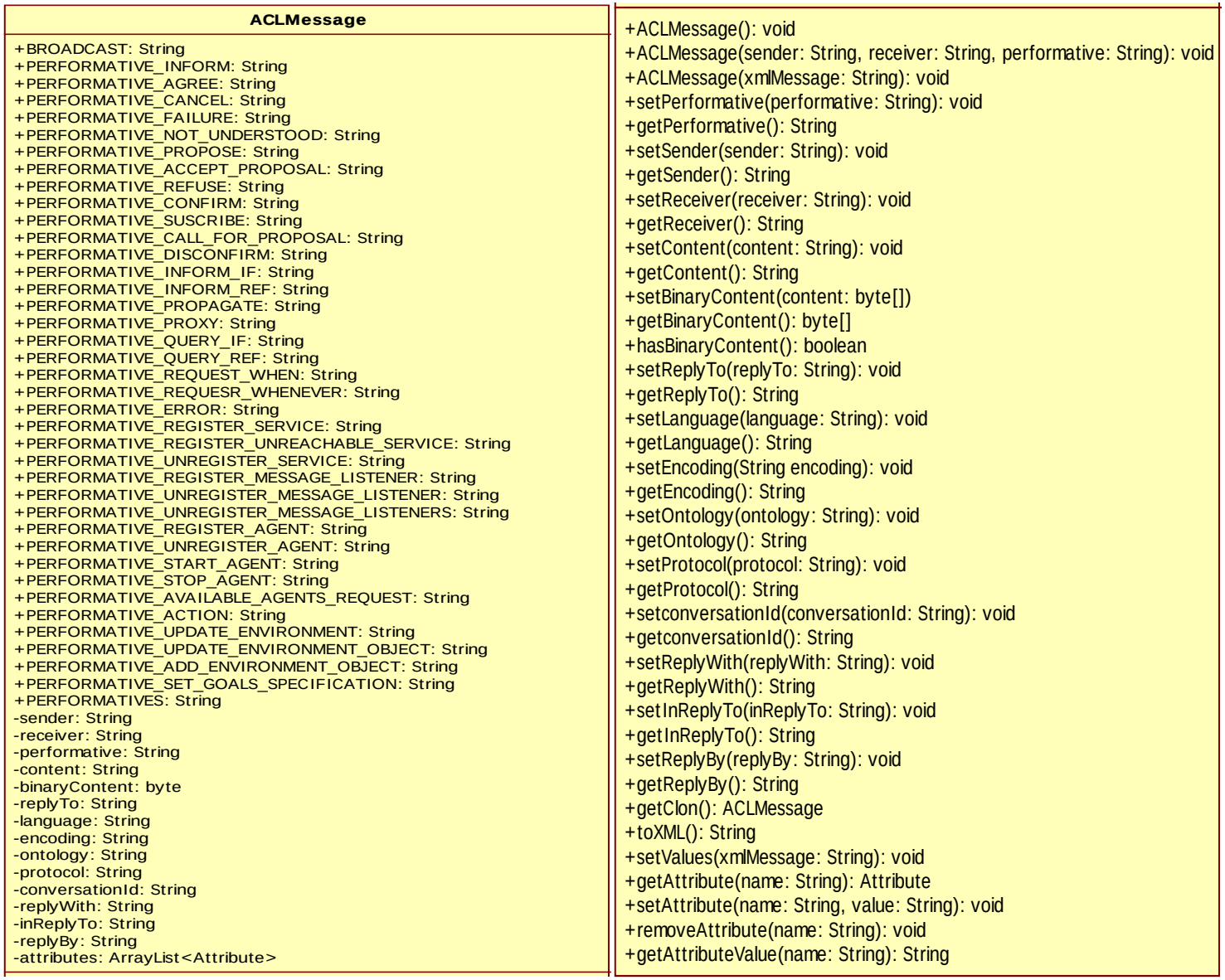

Figura 6.18: Clase ACLMessage.

de sus métodos get() y set(); esos parámetros son: respuesta para, contenido del mensaje, descripción del contenido y control de conversación.

Por otro lado, en la parte de los atributos es posible ver todas las performativas señaladas en el estándar FIPA llamado Librería del Acto Comunicativo, recordando que las performativas son la descripción de las acciones que va a realizar el agente por medio de una palabra referente a la acción.

6.4.2 Clase ACLMessageReceiver

Esta clase tiene como finalidad formar un receptor de acuerdo a las características especificadas tanto por FIPA como por W3C; por medio de esta, es posible llevar un control del número de agentes que tomen el papel de receptores (denominado así por FIPA) o proveedores (denominado así por W3C) al momento de iniciar la interacción entre ellos, ya que cuando se instancia esta clase se crea la conexión y así mismo se va almacenando el número de receptores que van a participar en la comunicación, esto es con el fin de llevar un control sobre el número de agentes que pueden proporcionar información a los agentes emisores.

Por lo tanto, como es posible ver en la figura 6.19 hay métodos para poner al receptor como oyente de las peticiones, saber el estado de la conexión la cual puede ser local o remota, el método run() ejecuta un hilo que pone al receptor a escuchar las conexiones entrantes en caso de haber en ese momento espera hasta que haya una, addReceiver() agrega un receptor a la lista cada vez que se genera uno y finalmente se tienen los métodos terminate() y removeTerminatedReceivers() que sirven para terminar uno o varios receptores respectivamente.

6.4.3 Clase ACLMessageSender

Esta clase al igual que la anterior, tiene la finalidad de formar un emisor de acuerdo a las características especificadas tanto por FIPA como por W3C; por lo tanto al

momento de crear una instancia de esta clase, los datos requeridos son la dirección IP donde se encuentra el receptor al que será enviado el mensaje y el puerto por donde se van a escuchar.

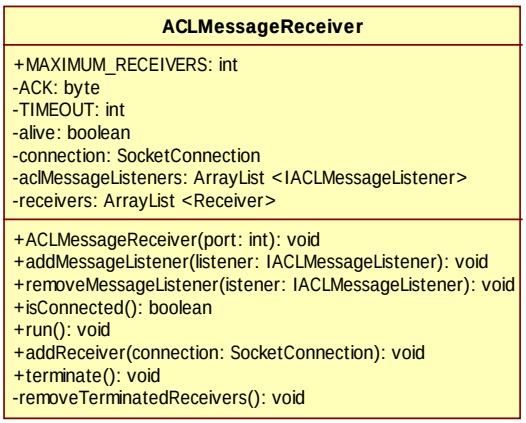

Figura 6.19: Clase ACLMessageReceiver.

Por otro lado, como se puede ver en la figura 6.20 hay otros métodos con los que será posible saber si se realizó la conexión correctamente, verificar si la conexión sigue abierta para poder comunicarse con un receptor, un método send() para enviar el mensaje, encargado de verificar si hay conexión y si no es así la crea para poder enviar el mensaje al receptor especificado y otro para cerrar la conexión.

6.4.4 Clase DeliveringMessageException

Esta clase, tiene como objetivo lanzar una excepción en caso de que surjan problemas al momento de que el mensaje está siendo enviado, es decir en caso de que existan errores causados porque los parámetros solicitados al enviar un mensaje no son los indicados, en la figura 6.21 se puede ver la clase en UML.

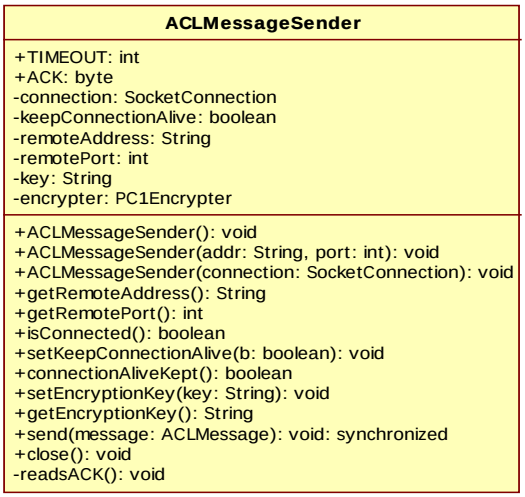

Figura 6.20: Clase ACLMessageSender.

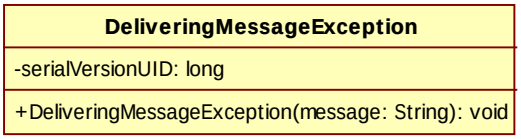

Figura 6.21: Clase DeliveringMessageException.

6.4.5 Clase ExceededReceiversException

La función que tiene esta clase, es la de lanzar una excepción en caso de que el número de receptores exceda el máximo permitido; es decir si se quiere agregar un receptor más a la lista y dicha lista ya tiene treinta receptores registrados, entonces la excepción es lanzada (ver figura 6.22).

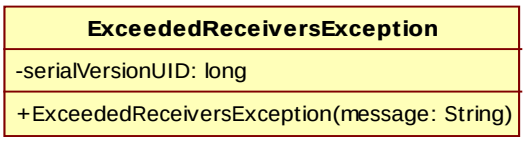

Figura 6.22: Clase ExceededReceiversException.

6.4.6 Interfaz IACLMessageListener

Dentro de esta clase existe un solo método llamado messageReceived() que es implementado en la clase MessageTransportService donde el método puede enviar los mensajes de acuerdo a dos situaciones, la primera permite difundirlo, por medio de su identificador; es decir, enviarlo a varios receptores registrados, la otra es enviarlo a un solo receptor por medio de su identificador (ver figura 6.23).

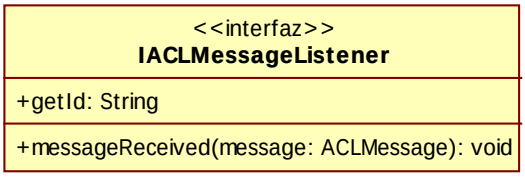

Figura 6.23: Clase IACLMessageListener.

6.4.7 Clase MalFormedMessageException

Esta clase es útil para la clase ACLMessage debido a que hay momentos en los cuales es necesario validar si los mensajes han sido formados correctamente de acuerdo a las estructuras especificadas por FIPA y W3C, en caso de no cumplir con lo determinado es cuando surge esta excepción (ver figura 6.24).

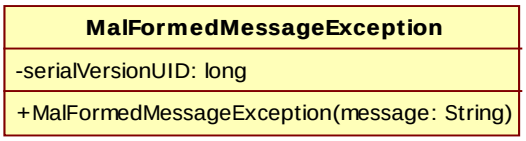

Figura 6.24: Clase MalFormedMessageException.

6.4.8 Clase MessageTransportService

Al generar instancias de este tipo, se deben proporcionar datos para lograr el intercambio de mensajes entre los agentes, son los siguientes: el identificador del

Servicio de Transporte de Mensajes el cual está formado por la dirección IP donde se ubica el receptor y el puerto, así como también existe la posibilidad de agregar un oyente (ver figura 6.25).

Figura 6.25: Clase MessageTransportService.

La clase está formada por métodos para verificar si el Servicio de Transporte de Mensaje fue iniciado, así como también da la posibilidad de detenerlo o suspenderlo cuando sea necesario; por otro lado también hay métodos para crear un oyente, es decir que un agente se ponga en estado de esperar (escuchar) alguna solicitud

(mensaje) que sea propagada por los demás oyentes que estén registrados, así como también hay opciones para eliminar y buscar dicho oyente. Y finalmente cuenta con un método que será el encargado de enviar el mensaje a su respectivo receptor.

6.4.9 Clase UnknownReceiverException

Esta clase (ver figura 6.26) es útil para la clase ACLMessageSender y la clase MessageTransportService, ya que cuando se utiliza su método send() el cual sirve para enviar a un receptor el mensaje formulado; se realiza una búsqueda de dicho receptor con el que se desea realizar la comunicación, en caso de no encontrarlo es cuando será necesario emplear esta clase la cual lanzará una excepción avisando que no existe el receptor al cual se le quiere enviar el mensaje.

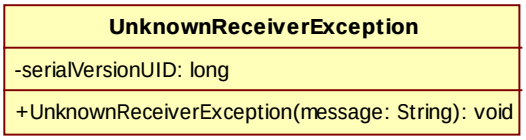

Figura 6.26: Clase UnknownReceiverException.

6.4.10 Clase Packet

Clase útil para la creación de paquetes con la información que será intercambiada por los agentes al momento de realizar la interacción, como es posible ver en el diagrama (figura 6.27) por medio de esta clase podrá especificarse el tamaño del paquete que podrá ser de un máximo de 64 KB, el código asignado a ese paquete, así como su contenido.

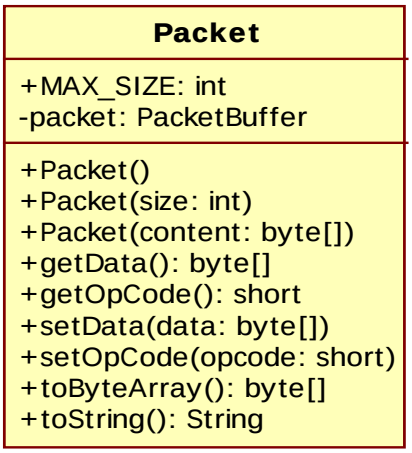

Figura 6.27: Clase Packet

6.4.11 Clase PacketBuffer

La clase PacketBuffer sirve para empaquetar y almacenar en un espacio de memoria la información contenida en los mensajes que se enviarán entre los agentes. Los tipos de datos que puede empaquetar son: int, byte, char, doublé, float, long y short (ver figura 6.28).

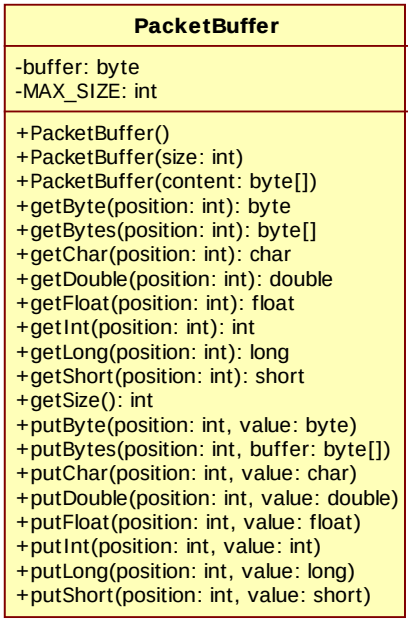

Figura 6.28: Clase PacketBuffer

6.5 Comparativa con otras Arquitecturas

Ahora bien, con la finalidad de resaltar de manera breve y concisa las características que brinda la arquitectura heterogénea aquí propuesta; en la tabla siguiente, se muestra una comparativa con base en las siguientes características: existencia de documentación (API), tutoriales que expliquen cómo crear aplicaciones multiagente, interfaz gráfica para hacer más visual y agradable el proceso de creación de los agentes y de sus comportamientos generales, cumplimiento o apego a lo que dicen FIPA y W3C, y el permitir contar con agentes de diversos tipos (heterogéneos).

Arquitectura	API	Tutoriales	Interfaz gráfica	FIPA	W3C	Peer to Peer	Agentes Heterogéneos
JADE	Sí	Sí	No	Sí	No	Sí	No
JACK	Sí	Sí	Sí	No	No	No	BDI y colaborativos
AnyLogic	Sí	Sí	Sí	No	No	No	No
MaDKit	Sí	Sí	No	No	No	Sí	No
MASON	Sí	Sí	No				
StarLogo	Sí	Sí	Sí	No	No	No	No
Arquitectura heterogénea propuesta en esta tesis	Sí	Sí, de momento en los casos de estudio presentados	No, pero en un futuro sería conveniente	Sí	Sí	Sí	Sí (reactivo, cognitivo, híbrido, basado en metas, basado en su utilidad, basado en modelos y EBDI)

Tabla 6.1: Comparativa de algunas Arquitecturas Multiagente basadas en Java contra la propuesta en esta tesis.

6.6 Casos de Estudio

En esta sección, se describirá cómo funciona la arquitectura heterogénea utilizando para ello casos de estudio que fueron implementados a manera de ejemplos, con el fin de demostrar la forma en la que se usan cada una de las clases y métodos que fueron empleados.

6.6.1 Simulación de un Escenario Cliente (cognitivo) - Servidor (reactivo)

Este caso de estudio, tiene como finalidad probar los beneficios que proporciona la arquitectura heterogénea que se plantea en este trabajo de tesis, ya que será posible ver la interacción entre agentes de diferente tipo, un reactivo que en este caso es el servidor y dos cognitivos que serán los clientes; así como también la comunicación de manera remota o distribuida ya que los tres agentes son ejecutados en diferentes computadoras.

En este ejemplo, se destaca el uso de las clases ReactiveAgent, CognitiveAgent, MessageTransportService y ACLMessage; con el fin de describir cómo se usan sus métodos y constructores. Los dos tipos de agentes usados son suficientes para demostrar que es posible crear escenarios en la arquitectura para que coexistan agentes de diferentes tipos, ya que todos son derivados de la clase AbstractAgent que es la clase padre, por lo tanto todos presentan las siguientes características que son: su ciclo de vida que funciona por medio de un hilo de control, con el cual es posible tener un seguimiento del agente, desde su creación hasta su destrucción. El uso de la clase ACLMessage es para el proceso de envío y recepción de mensajes permitiendo así la comunicación entre los agentes.

6.6.1.1 Agente Server

```
package Test;

import ACLMessage.ACLMessage;
import ACLMessage.MessageTransportService;
```

```
import ACLMessage.Packet;
import ACLMessage.PacketBuffer;
import AbstractAgentDefinition.Action;
import AbstractAgentDefinition.ReactiveAgentDefinition.ReactiveAgent;
import AbstractAgentDefinition.Situation;
import java.util.ArrayList;
import java.util.Random;

public class Server extends ReactiveAgent {
    private MessageTransportService messageTransportService;
    private ACLMessage request;
    private ACLMessage response;
    private boolean requestState;
    private int resultValue;
    Situation currentSituation;
```

Como se puede ver en el código de arriba, Server hereda de la clase ReactiveAgent lo que quiere decir que este agente tendrá un comportamiento de tipo reactivo. Los atributos messageTransportService, request y response servirán a lo largo de su ciclo de vida para lograr una interacción con los agentes clientes, es decir, para el envío y recepción de los mensajes entre él y ellos.

El atributo requestState, tiene la función de validar que la petición fue formulada correctamente, si es así entonces se empaqueta para enviarse al cliente. Así mismo, el atributo resultValue almacena el resultado de la operación que se realizó. Finalmente, currentSituation lleva el control de la situación actual que se está ejecutando en ese momento.

```
    public Server(){
        super();
        ClientServerEnvironment.init();
        updateSituation();
    }
```

Este constructor sólo tiene la invocación a dos métodos, la primera ClientServerEnvironment.init() tiene la función de inicializar la situación en la que se encontrará por primera vez el servidor. La segunda updateSituation() como su nombre lo dice va a actualizar la situación del agente.

```
    public Server(String id, MessageTransportService
messageTransportService,
            ArrayList <Situation> situations) {
```

```
        super(situations);
        ClientServerEnvironment.init();
        updateSituation();
        this.id = id;
        this.messageTransportService = messageTransportService;
        this.messageTransportService.addMessageListener(this);
        currentSituation = situations.get(3);
        currentState = currentSituation.getState();
        currentAction = getNextAction();
    }
```

Este constructor recibe los parámetros que formarán al agente, es decir el id con el cual se identificará cada agente, el servicio de transporte de mensajes para poder lograr la interacción con otros agentes y la lista de posibles situaciones en las que se podrá encontrar el agente. Así como el anterior constructor, este también inicializa la primera situación en la que se encontrará el agente que es “esperando por una petición”. Además, asigna el id al agente, inicializa el servicio de transporte de mensajes, especifica la situación actual y por medio de esta el estado actual y la acción actual. Cabe recordar que la situación está formada por un estado y una lista de acciones pertenecientes a ese estado.

```
    private void updateSituation() {
        currentSituation =
situations.get(ClientServerEnvironment.serverSituation);
        currentState = currentSituation.getState();
        System.out.println("Current situation: " +
ClientServerEnvironment.serverSituation);
        currentAction = getNextAction();
        System.out.println("Current action: " + currentAction.getName());
    }
```

Este método actualiza la situación dependiendo de lo que se le vaya asignando al atributo currentSituation, que es lo que se encuentra en el atributo estático serverSituation de la clase ClientServerEnvironment, es decir, dependiendo de la situación en la que se encuentre es cómo se determinará el estado actual y la situación actual.

```
    public void behavior() {
        while(true) {
            performAction(currentAction);
            updateSituation();
        }
    }
```

Con este método, se valida que el comportamiento del agente es infinito y mientras exista una petición se va a realizar una acción previamente definida invocando al método performAction(currentAction) y al mismo tiempo con el método updateSituation() se actualiza la situación.

```
    public void performAction(Action action) {
        if(currentAction.getId() ==
ClientServerEnvironment.ACTION_WAIT_REQUEST) {
            doWait();
        }
        else if(currentAction.getId() ==
ClientServerEnvironment.ACTION_RECEIVE_REQUEST) {
            ClientServerEnvironment.serverSituation =
ClientServerEnvironment.PROCESSING_REQUEST;
        }
        else if(currentAction.getId() ==
ClientServerEnvironment.ACTION_PROCESS_REQUEST) {
            processRequest();
        }
        else if(currentAction.getId() ==
ClientServerEnvironment.ACTION_GENERATE_REPLY) {
            generateReply();
        }
        else {
            sendReply(response);
        }
    }
```

Este método recibe como parámetro una acción que debe ser la acción actual, en caso de que el id de la acción sea igual a un valor ya predefinido entonces se ejecuta el comportamiento reactivo correspondiente a esa acción, de lo contrario si el id de la acción recibida no es igual a ninguno de los valores, entonces es momento de que el agente envíe la respuesta al cliente al invocar al método sendReply().

```
    public void processRequest() {
        int number1, number2;
        byte[] binaryContent;
        Packet packet;
        PacketBuffer packetBuffer;
        binaryContent = request.getBinaryContent();
        packet = new Packet(binaryContent);
        packetBuffer = new PacketBuffer(packet.getData());
        doWait(3000);
        resultValue = -1;
        if(packetBuffer.getInt(0) == 2 && packetBuffer.getInt(4) == 1
                && packetBuffer.getInt(12) == 1) {
            requestState = true;
            number1 = packetBuffer.getInt(8);
```

```
            number2 = packetBuffer.getInt(16);
            System.out.println("numeros: " + number1 + ", " + number2);
            switch(packet.getOpCode()) {
                case Operations.ADD:
                    resultValue = number1 + number2;
                break;
                case Operations.SUBSTRACT:
                    resultValue = number1 - number2;
                break;
                case Operations.MULTIPLY:
                    resultValue = number1 * number2;
                break;
                default: //case DIVIDE:
                    resultValue = number1 / number2;
                break;
            }
        }
        else {
            requestState = false;
        }
        ClientServerEnvironment.serverSituation =
ClientServerEnvironment.GENERATING_REPLY;
    }
```

Con la ayuda de este método se va a procesar la petición; es decir, una vez que se recibe el paquete de datos que envía el cliente al momento de hacer una solicitud (petición) se va a verificar que los parámetros enviados por el cliente sean los correctos para la operación requerida; validando el número de parámetros y el código de operación; si la validación es correcta, entonces se realiza la operación solicitada, la cual puede ser ya sea una suma, resta, multiplicación o división, de lo contrario el estado de la petición se pone en falso para que el servidor cambie nuevamente a estado de espera hasta que el cliente formule nuevamente una petición.

```
    private void generateReply() {
        String errorMessage;
        Packet packet;
        PacketBuffer packetBuffer;
        response = new ACLMessage();
        response.setSender(getId());
        response.setReceiver(request.getSender());
        response.setPerformative("action");
        if(requestState) {
            packetBuffer = new PacketBuffer(8);
            packetBuffer.putInt(0, 1);
            packetBuffer.putInt(4, resultValue);
            packet = new Packet(10);
            packet.setOpCode(Operations.SUCCESS);
        }
        else {
```

```
            errorMessage = "The request is malformed";
            packetBuffer = new PacketBuffer(8 + errorMessage.length());
            packetBuffer.putInt(0, 2);
            packetBuffer.putInt(4, errorMessage.length());
            packetBuffer.putBytes(8, errorMessage.getBytes());
            packet = new Packet(10 + errorMessage.length());
            packet.setOpCode(Operations.FAIL);
        }
        packet.setData(packetBuffer.getBytes(0));
        response.setBinaryContent(packet.toByteArray());
        ClientServerEnvironment.serverSituation =
ClientServerEnvironment.SENDING_REPLY;
    }
```

Este método se encarga de empaquetar la respuesta que se enviará al cliente; como primer paso formula la respuesta, especificando quien envía el mensaje que en este caso es el servidor así como también establece el receptor a quien va dirigido y la performativa señala lo que va a realizar el servidor. Como segundo paso, si el estado de la petición es verdadero entonces se empaqueta la respuesta para ser enviada al cliente, de lo contrario se manda un mensaje de error ya que quiere decir que los parámetros enviados fueron erróneos. Finalmente, se actualizan los datos del paquete a enviar y se establece la situación actual la cual cambia a “enviar la respuesta”.

```
    public void sendReply(ACLMessage response) {
        try {
            messageTransportService.sendMessage(response);
            ClientServerEnvironment.serverSituation =
ClientServerEnvironment.WAITING_REQUEST;
        } catch(Exception e) {}
    }
```

Este método recibe como parámetro el mensaje que se ha procesado y empaquetado para finalmente ser enviado al cliente, por lo tanto, la situación actual del servidor vuelve a estar en espera hasta que otro agente cliente envíe nuevamente una petición.

```
    public void processACLMessage(ACLMessage message) {
        request = message;
        ClientServerEnvironment.serverSituation =
ClientServerEnvironment.RECEIVING_REQUEST;
        doWake();
    }
}
```

Con este método se recibe como parámetro el mensaje de petición que envía el cliente, asignándolo al atributo request y después se actualiza la situación a “petición recibida” lo que quiere decir que el agente debe despertarse para empezar el proceso de respuesta al cliente.

6.6.1.2 Agente Client

```
package Test;

import ACLMessage.ACLMessage;
import ACLMessage.MessageTransportService;
import ACLMessage.Packet;
import ACLMessage.PacketBuffer;
import AbstractAgentDefinition.Action;
import AbstractAgentDefinition.CognitiveAgentDefinition.CognitiveAgent;
import AbstractAgentDefinition.CognitiveAgentDefinition.Objective;
import AbstractAgentDefinition.CognitiveAgentDefinition.Plan;
import AbstractAgentDefinition.CognitiveAgentDefinition.PlanningActivity;
import java.util.ArrayList;
import java.util.Random;

public class Client extends CognitiveAgent {
    private MessageTransportService messageTransportService;
    private Plan currentPlan;
    private Objective currentObjective;
    private Action nextAction;
    private int numberOfLearnedOperations;
    private boolean[] parameters;
    private ACLMessage reply;
```

Como es posible ver en el código de arriba, este agente cliente va a tener un comportamiento de tipo cognitivo por lo tanto tiene que heredar de la clase CognitiveAgent. Los atributos declarados, son útiles para crear un servicio de transporte de mensajes para la comunicación entre los agentes, así como también para saber cuál es el plan actual, el objetivo actual, la acción siguiente a ser ejecutada, llevar un registro del número de operaciones aprendidas, almacenar los parámetros que enviará el agente cliente al agente servidor, y finalmente, una respuesta donde se asignará el valor que es recibido del agente servidor.

```
    public Client(){
        super();
    }
```

Este constructor sólo tiene la función de inicializar a un objeto cliente, ya que se usará en caso de declarar a un objeto sin parámetros.

```
    public Client(String id, MessageTransportService
messageTransportService,
            ArrayList <PlanningActivity> planningActivities) {
        super(planningActivities);
        this.id = id;
        this.messageTransportService = messageTransportService;
        this.messageTransportService.addMessageListener(this);
        currentPlan = currentPlanningActivity.getCurrentPlan();
        numberOfLearnedOperations = 0;
        parameters = new boolean[3];
        currentAction =
currentPlanningActivity.getCurrentPlan().getFirstAction();
    }
```

Por medio de este constructor, se inicializa un objeto cliente con parámetros que son: el identificador del mismo, un servicio de mensajes para la comunicación con el servidor y las actividades de planeación que debe cumplir para lograr sus objetivos. También se especifica el plan actual a ejecutar, se inicializan las operaciones aprendidas en cero ya que hasta ese momento no ha aprendido a realizar ninguna operación matemática, así como también se determina la primera acción derivada de la actividad de planeación.

```
    private void tryRequest() {
        int numberOfValues;
        Random randomValue;
        ACLMessage request;
        byte[] binaryContent;
        Packet packet;
        PacketBuffer packetBuffer;
        String errorMessage;
        nextAction = new
Action(ClientServerEnvironment.ACTION_GENERATE_REQUEST, "generate
request");
        setNextAction(nextAction);
        doWait(3000);
        request = new ACLMessage();
        request.setSender(getId());
        request.setReceiver("server");
        request.setPerformative("action");
        randomValue = new Random();
        do {
            numberOfValues = randomValue.nextInt(3);
        } while(parameters[numberOfValues] == true);
        parameters[numberOfValues] = true;
        System.out.println("number of values” + numberOfValues);
```

```
        switch(numberOfValues) {
            case 0:
                packetBuffer = new PacketBuffer(4);
                packetBuffer.putInt(0, 0);
            break;
            case 1:
                packetBuffer = new PacketBuffer(12);
                packetBuffer.putInt(0, 1);
                packetBuffer.putInt(4, 1);
                packetBuffer.putInt(8, randomValue.nextInt());
            break;
            default:
                packetBuffer = new PacketBuffer(20);
                packetBuffer.putInt(0, 2);
                packetBuffer.putInt(4, 1);
                packetBuffer.putInt(8, randomValue.nextInt());
                packetBuffer.putInt(12, 1);
                packetBuffer.putInt(16, randomValue.nextInt());
            break;
        }
        packet = new Packet(2 + packetBuffer.getSize());
        if(currentPlan.getFirstAction().getName().equals("initialize add"))
{
            packet.setOpCode(Operations.ADD);
        }
        else if(currentPlan.getFirstAction().getName().equals("initialize
substraction")) {
            packet.setOpCode(Operations.SUBSTRACT);
        }
        else if(currentPlan.getFirstAction().getName().equals("initialize
multiply")) {
            packet.setOpCode(Operations.MULTIPLY);
        }
        else {
            packet.setOpCode(Operations.DIVIDE);
        }
        packet.setData(packetBuffer.getBytes(0));
        request.setBinaryContent(packet.toByteArray());
        nextAction = new
Action(ClientServerEnvironment.ACTION_SEND_REQUEST, "send request");
        setNextAction(nextAction);
        try {
            messageTransportService.sendMessage(request);
            nextAction = new
Action(ClientServerEnvironment.ACTION_WAIT_REPLY, "wait reply");
            setNextAction(nextAction);
            doWait();
            nextAction = new
Action(ClientServerEnvironment.ACTION_PROCESS_REPLY, "process reply");
            setNextAction(nextAction);
            binaryContent = reply.getBinaryContent();
            packet = new Packet(binaryContent);
            packetBuffer = new PacketBuffer(packet.getData());
            if(packet.getOpCode() == Operations.SUCCESS) {
                System.out.println("reply: " + packetBuffer.getInt(0)
                    + "," + packetBuffer.getInt(4));
                currentObjective.setReached(true);
```

```
            }
            else {
                if(packetBuffer.getInt(0) == 2) {
                    errorMessage = new String(packetBuffer.getBytes(8));
                    System.out.println("Error: " + errorMessage);
                }
            }
        } catch(Exception e) {System.out.println(
                "Error al enviar petición: " + e.getMessage());
        }
    }
```

Este método como su nombre lo dice, intenta crear la petición que será enviada al servidor; declarando variables para: almacenar y generar el número de valores enteros que serán enviados al servidor para realizar la operación aritmética, una variable request de tipo ACLMessage para generar la petición, un arreglo para almacenar el contenido binario que será enviado como paquete por medio de las variables packet y packetBuffer. Posteriormente, se agrega una nueva acción al plan actual que es “generando petición”; para después formar la petición donde se debe especificar: el nombre que identifica a este cliente, determinar a quién será enviada la petición que es el servidor y la performativa que se está realizando.

Después, se genera aleatoriamente el número de parámetros que serán enviados para realizar la petición, éstos podrán ser cero, uno o dos números enteros, éstos serán validados para empaquetarlos dependiendo del número que se esté intentando enviar al servidor. Posteriormente, se elige la operación aritmética a realizar dependiendo de la acción en la que se encuentre en ese momento el agente cliente, luego se empaquetan los números que serán enviados y cambia ahora la acción actual a “enviar petición”.

Ahora por medio del servicio de transporte de mensajes se enviará la petición al servidor y entonces es momento de hacer cambio de acciones, ya que ahora estará esperando por una respuesta. Por lo tanto, el ciclo de vida del agente cambia a espera por medio de método doWait() y así se quedará en ese estado hasta que llegue una respuesta y se despierte; entonces volverá a cambiar su acción a “procesar respuesta”, entonces se desempaqueta el mensaje de respuesta enviado

por el servidor, se valida que fue recibido exitosamente y por lo tanto se confirma que el objetivo fue cumplido, si no fue exitoso se envía un mensaje de error.

```
    private int performPlan() {
        int i;
        currentAction = currentPlan.getFirstAction();
        System.out.println("Current action client: " +
currentAction.getName());
        for(i = 0; i < 3; i++) {
            parameters[i] = false;
        }
        i = 0;
        while(!currentObjective.isReached()) {
            tryRequest();
            i++;
        }
        nextAction = currentPlan.getLastAction();
        setNextAction(nextAction);
        System.out.println("Current action client: " +
currentAction.getName());
        numberOfLearnedOperations++;
        System.out.println("The current plan was performed in " + i + "
trials");
        return i;
    }
```

Este método cada vez que se ejecuta, imprime la acción actual que se encuentra ejecutando el cliente; así también se encarga de iniciar el aprendizaje ya que va registrando los parámetros que ya fueron enviados por el cliente y si ya intentó con parámetros que no fueron factibles para generar la operación aritmética no lo volverá a intentar con esos, puesto que ya aprendió que no es posible realizar la petición. También va a intentar formular la petición hasta que el objetivo sea cumplido. Finalmente, una vez cumplido el objetivo se muestra que el agente terminó de aprender a formular peticiones de manera correcta, así como el número de intentos que hizo el agente hasta lograr generar una petición de la forma requerida por el servidor.

```
    public void behavior() {
        int i;
        System.out.println("Current action client: " +
currentAction.getName());
        i = 0;
        while(numberOfLearnedOperations < 4) {
            currentPlan = currentPlanningActivity.nextPlan();
            currentObjective = currentPlanningActivity.nextObjective();
```

```
            i += performPlan();
        }
        currentPlan = currentPlanningActivity.getPlans().get(0);
        nextAction = currentPlan.getLastAction();
        setNextAction(nextAction);
        System.out.println("Current action client: " +
currentAction.getName());
        System.out.println("The learning task was performed in " + i + "
trials");
        setAlive(false);
```

En este método, se va mostrando que operación aritmética se realiza en ese momento y cuando se concluye con ella entonces se cambia al siguiente plan que sería la siguiente operación aritmética para ejecutar su lista de acciones correspondientes, así como también imprimir la acción en la que se encuentre en ese momento el plan.

```
    public void processACLMessage(ACLMessage message) {
        nextAction = new
Action(ClientServerEnvironment.ACTION_RECEIVE_REPLY, "received reply");
        setNextAction(nextAction);
        reply = message;
        doWake();
    }
}
```

La función que tiene este método, es recibir como parámetro el mensaje de respuesta por parte del cliente, por lo tanto, se modifica la acción a “respuesta recibida”, se recibe el mensaje y se despierta al agente para que realice la acción de procesar la respuesta recibida y finalmente mostrar lo que en es este caso es el resultado de la operación aritmética.

6.6.1.3 Ejecución del EscenarioTestClientServer

```
package Test;

import ACLMessage.DeliveringMessageException;
import ACLMessage.MessageTransportService;
import ACLMessage.UnknownReceiverException;
import AbstractAgentDefinition.Action;
import AbstractAgentDefinition.CognitiveAgentDefinition.Objective;
import AbstractAgentDefinition.CognitiveAgentDefinition.Plan;
import AbstractAgentDefinition.CognitiveAgentDefinition.PlanningActivity;
import AbstractAgentDefinition.Perception;
```

```
import AbstractAgentDefinition.Situation;
import AbstractAgentDefinition.State;
import java.io.IOException;
import java.util.ArrayList;

public class TestClientServer {
    public static void main(String[] args) {
        MessageTransportService messageTransportService;
        Client2 client;
        Server2 server;
```

Esta es la clase TestClientServer y será la principal, ya que se encargará de instanciar objetos de tipo cliente y servidor a los cuales se les proporcionará los parámetros necesarios para que ambos puedan comenzar la interacción e intercambio de mensajes gracias al objeto messageTransportService.

```
        Action action0 = new
Action(ClientServerEnvironment.ACTION_WAIT_REQUEST,
                "Wait request");
        Action action1 = new
Action(ClientServerEnvironment.ACTION_RECEIVE_REQUEST,
                "Receive request");
        Action action2 = new
Action(ClientServerEnvironment.ACTION_PROCESS_REQUEST,
                "Process Request");
        Action action3 = new
Action(ClientServerEnvironment.ACTION_GENERATE_REPLY,
                "Generate reply");
        Action action4 = new
Action(ClientServerEnvironment.ACTION_SEND_REPLY,
                "Send reply");
```

Las cuatro declaraciones que se pueden ver arriba inicializan objetos de tipo Action con cada una de las posibles acciones que podrá ejecutar el agente servidor ya que es el que funcionará de manera reactiva, por lo tanto, se deben tener ya predefinidas las acciones que puede desarrollar, las cuales son: “esperar petición”, “recibir petición”, “procesar petición”, “generar respuesta” y “enviar respuesta”.

```
        State state0 = new State();
        state0.addPerception(new Perception("Waiting request",
                ClientServerEnvironment.WAITING_REQUEST));
        State state1= new State();
        state1.addPerception(new Perception("Receiving request",
                ClientServerEnvironment.RECEIVING_REQUEST));
        State state2 = new State();
        state2.addPerception(new Perception("Processing request",
                ClientServerEnvironment.PROCESSING_REQUEST));
```

```
State state3 = new State();
state3.addPerception(new Perception("Generating reply",
        ClientServerEnvironment.GENERATING_REPLY));
State state4 = new State();
state4.addPerception(new Perception("Sending reply",
        ClientServerEnvironment.SENDING_REPLY));
```

Las anteriores declaraciones inicializan objetos de tipo State con cada uno de los posibles estados en los que se puede encontrar el agente servidor ya que es el que funcionará de manera reactiva por lo tanto debe tenerlos ya predefinidos, estos son: “esperando petición”, “recibiendo petición”, “procesando petición”, “generando respuesta” y “enviando respuesta”.

```
Situation situation0 = new Situation();
situation0.setState(state0);
situation0.addAction(action0);
Situation situation1 = new Situation();
situation1.setState(state1);
situation1.addAction(action1);
Situation situation2 = new Situation();
situation2.setState(state2);
situation2.addAction(action2);
Situation situation3 = new Situation();
situation3.setState(state3);
situation3.addAction(action3);
Situation situation4 = new Situation();
situation4.setState(state4);
situation4.addAction(action4);
```

Las declaraciones de arriba, al igual que para las acciones y los estados, inicializan objetos de tipo Situation con cada una de las posibles situaciones por las que pasará el agente servidor, dando así un comportamiento reactivo; recordando que las situaciones están formadas por un estado y una o varias acciones para dicho estado.

```
ArrayList <Situation> situations = new ArrayList <Situation>();
situations.add(situation0);
situations.add(situation1);
situations.add(situation2);
situations.add(situation3);
situations.add(situation4);
```

Esta estructura de datos en forma de arreglo de elementos de tipo Situation tiene la finalidad de crear una lista de las situaciones en las que se puede encontrar el agente en un momento dado y es útil ya que a la hora de pasar los parámetros al

constructor del servidor, recibirá las situaciones, pero sólo a manera de lista para un mejor control sobre la búsqueda de situaciones durante la ejecución del agente.

```
        Plan generalPlan = new Plan(new Action(-1, "initialize learning"),
                new Action(10, "finalize learning"));
        Plan addPlan = new Plan(new Action(-1, "initialize add"),
                new Action(10, "finalize add"));
        Plan substractPlan = new Plan(new Action(-1, "initialize
substract"),
                new Action(10, "finalize substract"));
        Plan multiplyPlan = new Plan(new Action(-1, "initialize multiply"),
                new Action(10, "finalize multiply"));
        Plan dividePlan = new Plan(new Action(-1, "initialize divide"),
                new Action(10, "finalize divide"));
        Objective generalObjective = new Objective(0,
                "learn to form requests",
                "request arithmethic operations");
        Objective addObjective = new Objective(1,
                "learn to form add request",
                "request add operation");
        Objective substractObjective = new Objective(1,
                "learn to form substract request",
                "request substract operation");
        Objective multiplyObjective = new Objective(1,
                "learn to form multiply request",
                "request multiply operation");
        Objective divideObjective = new Objective(1,
                "learn to form divide request",
                "request divide operation");
```

En el código anterior, se instancian objetos de tipo Plan y Objective los cuales serán empleados por el agente cliente, ya que estos son parte de las características que integran a un agente de tipo cognitivo, sin dejar de lado que un plan está formado por una acción inicial, una acción final y a cada plan le corresponde un objetivo que se debe cumplir.

```
        ArrayList <Plan> plans = new ArrayList<Plan>();
        plans.add(generalPlan);
        plans.add(addPlan);
        plans.add(substractPlan);
        plans.add(multiplyPlan);
        plans.add(dividePlan);

        ArrayList <Objective> objectives = new ArrayList<Objective>();
        objectives.add(generalObjective);
        objectives.add(addObjective);
        objectives.add(substractObjective);
        objectives.add(multiplyObjective);
        objectives.add(divideObjective);
```

```
        PlanningActivity clientPlanActivity = new
PlanningActivity(plans,objectives);
```

Con el código antes mostrado, se instancian dos estructuras de datos en forma de arreglos de elementos de tipo Plan y Objective, el arreglo de tipo Plan contiene los planes que tendrá el agente cliente y son: el plan general para iniciar el aprendizaje, plan para hacer una suma, una resta, una multiplicación y una división; y el arreglo de tipo Objective registra los objetivos que se deben cumplir para satisfacer las necesidades del agente. Estos dos arreglos sirven para formar la actividad de planeación, la cual lleva un registro de relación plan-objetivo, lo que quiere decir, que a cada plan le corresponde un objetivo por cumplir.

```
        ArrayList<PlanningActivity> planningActivities = new
ArrayList<PlanningActivity>();
        planningActivities.add(clientPlanActivity);

        try {
            messageTransportService =
MessageTransportService.getInstance();
            messageTransportService.addRemoteMessageTransportService(
                                                              "127.0.0.1",
6000);
            server = new Server2("server", messageTransportService,
                                 situations);
            client = new Client2("client", messageTransportService,
                    planningActivities);
        } catch(IOException
                | NullPointerException
                | UnknownReceiverException
                | DeliveringMessageException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Para finalizar, se genera una lista de actividades de planeación para que éstas puedan ser enviadas al constructor del cliente; posteriormente, se agrega el servicio de transporte de mensajes, el cual contiene la IP de la máquina donde se encuentra el agente con el cual se realizará la interacción; así como también, el puerto por el cual se va a escuchar. Después, se crean los agentes de tipo Sever y Client donde se debe especificar el identificador de cada agente, con quien se realizará la

comunicación en el servicio de transporte de mensajes, las situaciones en caso del servidor y las actividades de planeación para el agente cliente.

6.6.1.4 Salida en Consola

En las siguientes imágenes, se puede ver el resultado de la interacción entre el agente los agentes cliente y el agente servidor; cabe mencionar, que dichos agentes se encuentran en diferentes máquinas lo que quiere decir que la arquitectura cumple con el aspecto de ser distribuida.

En la figura 6.29, se puede apreciar el comportamiento del agente servidor, la primer acción desde el momento que comienza a ejecutarse es “esperar por una petición” ahí se mantendrá hasta que algún cliente localizado en otra máquina le envíe un mensaje de petición, entonces es cuando cambia la acción a “recibe petición”, una vez que la recibe cambia a “procesa petición”, es decir, desempaqueta los datos que se le están enviando, que en este caso son dos números enteros, enseguida genera la operación matemática que se le haya solicitado cambiando su acción a “generar respuesta”, después, se empaqueta el resultado de la operación matemática y se envía, por lo tanto, cambia el estado a “enviar respuesta”, y finalmente, la acción del servidor vuelve a “esperar petición”. Con esto, está repitiendose el mismo ciclo antes mencionado en cuanto llegue otra petición ya sea por el mismo cliente o por otro diferente.

Como se muestra en la figura 6.30 la salida que tiene un cliente, donde la primer acción que realizará es “iniciar el aprendizaje” para poder formular peticiones de manera correcta, ya que el agente inicialmente no sabe como hacerlo; por lo tanto, desconoce qué debe proporcionar al menos dos enteros para que el servidor pueda realizar alguna de las operaciónes aritméticas que ya tiene definidas, las cuales son sumar, restar, multiplicar y dividir.

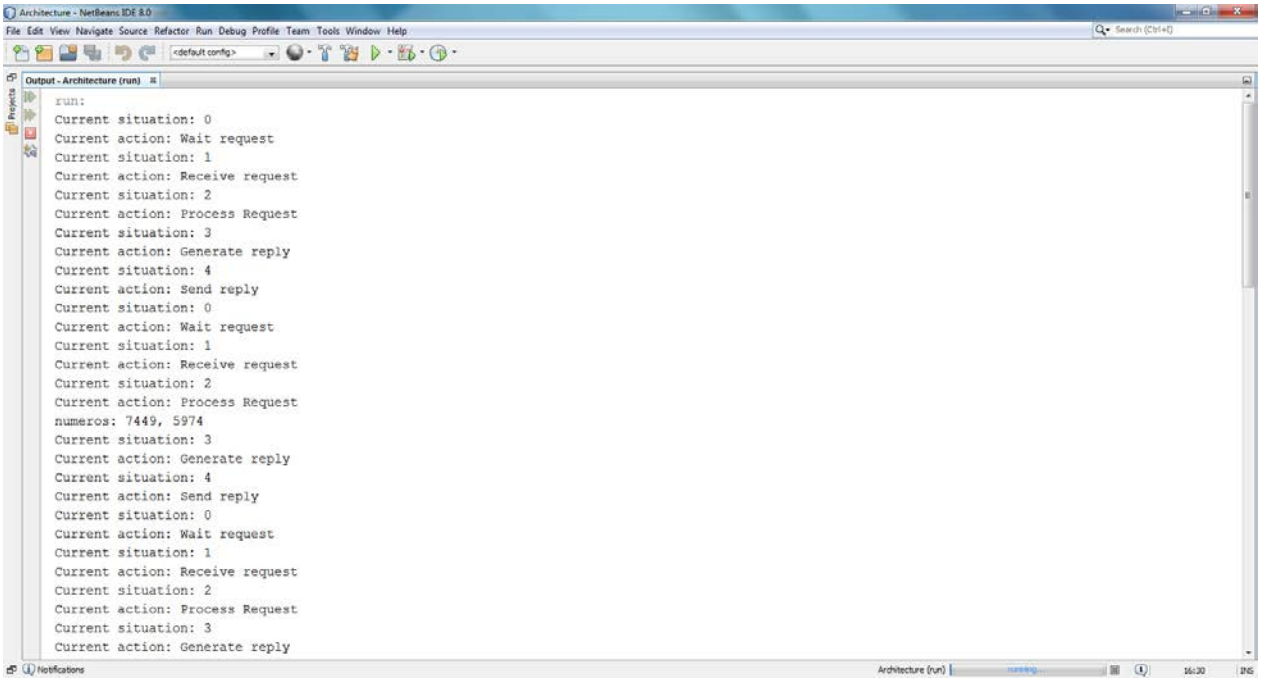

Figura 6.29: Ejemplo de Salida en Consola del Agente Server.

Seguido de lo anterior, en la misma figura 6.30, el agente intenta formular la petición para cada una de las operaciónes aritméticas, la primera petición es para realizar una suma enviando dos enteros al servidor por lo tanto este le responde de manera afirmativa ya que pudo recibir los parámetros necesarios para poder realizar la operación.

Enseguida, tiene que formular una petición para la resta lo que quere decir que ahora va a aprender a formular peticiones para realizar esta operación, el agente decide enviar cero parámetros a lo que el servidor le responde que “la petición esta mal formada” ya que no puede realizar una operación con cero números, con esto inicia el aprendizaje del agente cliente ya que aprende que enviar una petición al servidor para realizar una resta con cero parámetros no es posible y no lo volverá a hacer, ahora formula una petición con un parámetro y el servidor le vuelve a responder que no es correcto; ahora el cliente aprende que no puede realizar peticiones con un parámetros, finalmente, formula la petición con dos enteros y entonces el servidor le

responde enviando el resultado de la operación aritmética y aprende que así si es posible.

Con esto el cliente, se da cuenta que cumplió su objetivo de “aprender a formular peticiones para realizar una resta”, así termina el aprendizaje para esa operación y ahora toca el turno de aprender a hacer peticiones para otras nuevas que son una multiplicación y una división; una vez que termina de aprender a formular peticiones para todas las operaciones el cliente informa que “ha terminado de aprender”.

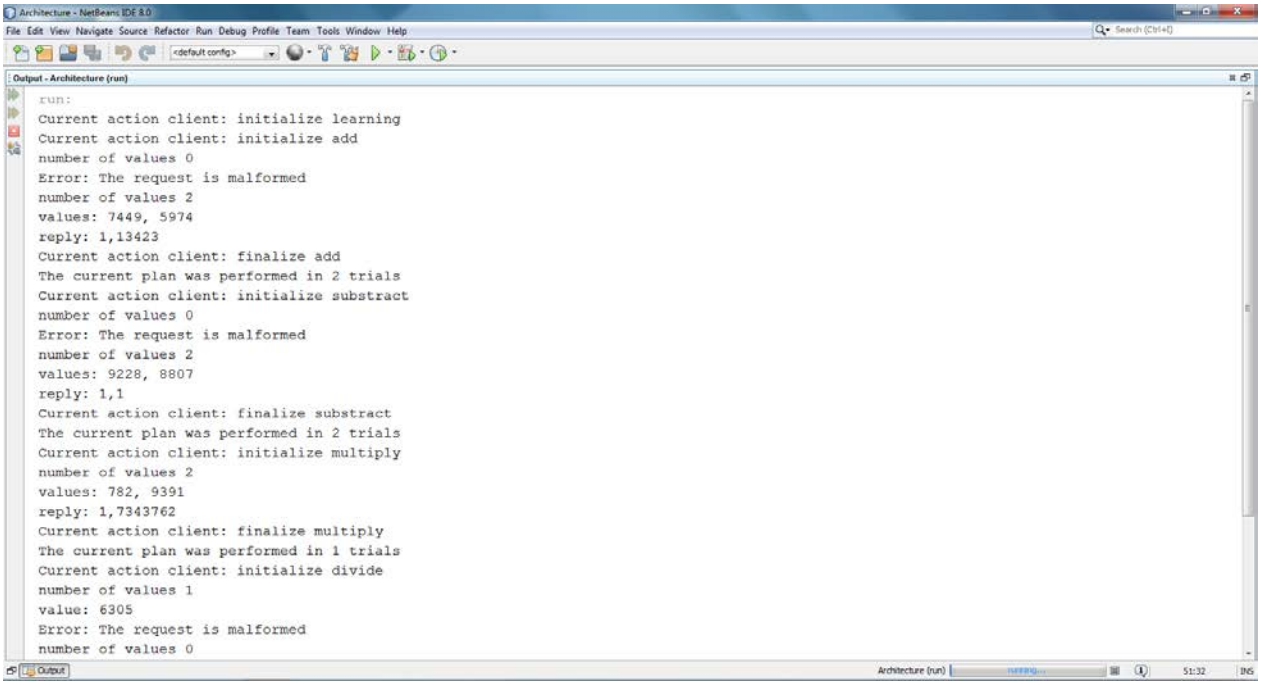

Figura 6.30: Ejemplo de Salida en Consola del Agente Client.

6.6.2 Simulación de una Aspiradora en un escenario de nueve celdas.

En este ejemplo, se utilizó la clase ReactiveAgent, ya que a un Agente Aspiradora se le proporcionará un comportamiento de tipo reactivo en base a ciertas reglas predefinidas para las situaciones que perciba de su entorno. Es decir, para cada situación la aspiradora va a poder tener sólo 2 percepciones, que son: primeramente, el estado de la celda en que se encuentra, dado por dos valores sucia o limpia; en seguida, en qué número de celda se localiza. Posteriormente, buscará cada situación

en el conjunto de reglas preestablecidas para realizar una posible acción que le corresponda. En la figura 6.32, se puede apreciar que el agente va a poder moverse únicamente hacia cuatro máximas posibles direcciones: arriba, abajo, izquierda y derecha, dependiendo de la celda en donde se encuentre.

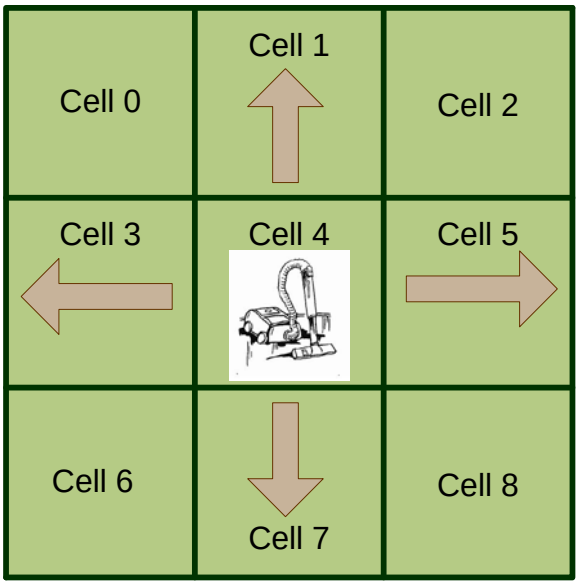

Figura 6.31: Estados y posibles acciones para un Agente Aspiradora.

A continuación, se describe la codificación del agente.

6.6.2.1 Escenario VacuumEnvironment3x3

```
package Test;

import java.util.Random;

public class VacuumEnvironment3x3 {
     public final static int NUMBER_OF_LOCATIONS = 9;
     public final static int NUMBER_OF_STATE_VALUES = 2;
```

Esta clase, en su mayoría está formada por atributos que le serán de gran utilidad al agente para evaluar sus percepciones y decidir acciones para todas sus posibles situaciones. El atributo NUMBER_OF_LOCATIONS es el número de celdas en las que se dividió el área que el agente limpiará, el atributo

NUMBER_OF_STATE_VALUES contiene el número de estados que puede tener cada una de las ubicaciones o celdas.

```
public final static int LOCATION_0 = 0;
public final static int LOCATION_1 = 1;
public final static int LOCATION_2 = 2;
public final static int LOCATION_3 = 3;
public final static int LOCATION_4 = 4;
public final static int LOCATION_5 = 5;
public final static int LOCATION_6 = 6;
public final static int LOCATION_7 = 7;
public final static int LOCATION_8 = 8;
```

Estos nueve atributos representan el número de celdas o ubicaciones en las que se dividió el área que se limpiará.

```
public final static int ACTION_SUCK = 0;
public final static int ACTION_MOVE_LEFT = 1;
public final static int ACTION_MOVE_RIGHT = 2;
public final static int ACTION_MOVE_UP = 3;
public final static int ACTION_MOVE_DOWN = 4;
```

Los atributos que arriba se pueden apreciar, se generaron con el fin de representar cada una de las acciones que puede ejecutar el agente, dependiendo de la situación en la que se encuentre y el estado de la celda.

```
public static int currentLocation;
public static int currentLocationState;
```

El atributo currentLocation contendrá el número de celda o ubicación en la que se encuentra el agente. Por otro lado, se tiene que el atributo currentLocationState contiene el estado actual de la celda, ya sea limpia o sucia.

```
public static class LocationState {
    public final static int CLEAN = 0;
    public final static int DIRTY = 1;
}
```

La clase anterior, contiene los valores de los posibles estados que se le pueden asignar a las celdas, ya sea sucia o limpia.

```
    public static void init() {
        Random r = new Random();
        currentLocation = r.nextInt(NUMBER_OF_LOCATIONS);
        currentLocationState = r.nextInt(NUMBER_OF_STATE_VALUES);
    }
```

Finalmente, se tiene al método init(), que tiene la función de inicializar la ubicación del Agente Aspiradora, dándole el número de la celda en donde se ubica y el estado en el que se encuentra la misma.

6.6.2.2 Agente VacuumCleaner

```
package Test;

import ACLMessage.ACLMessage;
import AbstractAgentDefinition.Action;
import AbstractAgentDefinition.ReactiveAgentDefinition.ReactiveAgent;
import AbstractAgentDefinition.Situation;
import java.util.ArrayList;
import java.util.Random;
```

La clase VacuumCleaner será la encargada de recibir los valores que tendrán las acciones, percepciones, estados y situaciones; así como también, contiene el comportamiento que podría tener el agente dependiendo de la situación en la que se encuentre.

```
public class VacuumCleaner extends ReactiveAgent {
    public VacuumCleaner() {
        super();
    }

    public VacuumCleaner(ArrayList<Situation> situations) {
        super(situations);
    }
```

En la parte de arriba, se pueden ver dos constructores, con el primer constructor es posible generar un objeto de esta clase sin parámetros y el segundo recibe como parámetros las situaciones, recordando que las situaciones están formadas por acciones y estados, los estados a su vez están formados por percepciones.

```
    public void clean() {
        VacuumEnvironment3x3.init();
```

El método clean() tiene la función de ejecutar el método init(), que se encuentra en la clase VacuumEnvironment3x3; recordando que sirve para proporcionarle al agente el número de celda que limpiará, dicho de otra manera su ubicación, además del estado de esta.

```
    public void behavior() {
        Situation currentSituation;
        doWait();
        System.out.println("Sale del dowake");
        while(true) {
            currentSituation =
situations.get(VacuumEnvironment3x3.currentLocation * 2 +
VacuumEnvironment3x3.currentLocationState);
            currentState = currentSituation.getState();
            System.out.println("Current location: " +
VacuumEnvironment3x3.currentLocation);
            System.out.println("Current location state: " +
VacuumEnvironment3x3.currentLocationState);
            currentAction = getNextAction();
            System.out.println("Current action: " +
currentAction.getName());
            if(currentAction.getId() == VacuumEnvironment3x3.ACTION_SUCK)
{
                VacuumEnvironment3x3.currentLocationState =
VacuumEnvironment3x3.LocationState.CLEAN;
            }
            else {
                if(currentAction.getId() ==
VacuumEnvironment3x3.ACTION_MOVE_LEFT) {
                    VacuumEnvironment3x3.currentLocation -= 1;
                    currentState.getPerceptions().get("current
location").setValue(VacuumEnvironment3x3.LocationState.CLEAN);
                }
                else if(currentAction.getId() ==
VacuumEnvironment3x3.ACTION_MOVE_RIGHT) {
                    VacuumEnvironment3x3.currentLocation += 1;
                    currentState.getPerceptions().get("current
location").setValue(VacuumEnvironment3x3.LocationState.CLEAN);
                }
                else if(currentAction.getId() ==
VacuumEnvironment3x3.ACTION_MOVE_UP) {
                    VacuumEnvironment3x3.currentLocation -= 3;
                    currentState.getPerceptions().get("current
location").setValue(VacuumEnvironment3x3.LocationState.CLEAN);
                }
                else if(currentAction.getId() ==
VacuumEnvironment3x3.ACTION_MOVE_DOWN) {
                    VacuumEnvironment3x3.currentLocation += 3;
                    currentState.getPerceptions().get("current
location").setValue(VacuumEnvironment3x3.LocationState.CLEAN);
                }
                VacuumEnvironment3x3.currentLocationState = new
Random().nextInt(VacuumEnvironment3x3.NUMBER_OF_STATE_VALUES);
```

```
        }
        doWait(5000);
```

En el método behavior(), se crea una variable donde se almacenará la situación actual y se pone en espera el hilo del agente, para que posteriormente, se haga la elección de la acción a ejecutar. Enseguida, se inicializa la variable currentSituation con el valor de la situación actual; esto se determina gracias al estado actual y al número de celda que fueron asignadas al agente, por medio del método init() en la clase VacuumEnvironment3x3. El estado actual se obtiene de la situación que se formó.

Por otro lado, a la variable currentAction se le asigna el valor de la acción que se va a ejecutar, este proceso se lleva a cabo gracias al método getNextAction() que se hereda y especializa de la clase ReactiveAgent, este método lanza una acción aleatoria de entre las posibles acciones que se tienen dependiendo de cada situación.

Por último, se ejecutará la acción de limpiar en caso de que este sucia la celda, o bien, moverse a la izquierda, a la derecha, hacia arriba o hacia abajo según sea el caso del valor que contenga la variable currentAction.

6.6.2.3 Ejecución del Escenrario TestVacuumCleaner

```
package Test;

import AbstractAgentDefinition.Action;
import AbstractAgentDefinition.Perception;
import AbstractAgentDefinition.Situation;
import AbstractAgentDefinition.State;
import java.util.ArrayList;
```

Como se podrá apreciar a continuación, la clase principal TestVacuumCleaner crea objetos de todos los posibles estados y situaciones en los que se puede encontrar el Agente Aspiradora, así como también pasa los parámetros necesarios al constructor de la clase VacuumCleaner.

```
public class TestVacuumCleaner {

    public static void main (String args[]){
        State state0 = new State();
        state0.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_0));
        state0.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.CLEAN));

        State state1 = new State();
        state1.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_0));
        state1.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.DIRTY));

        State state2 = new State();
        state2.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_1));
        state2.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.CLEAN));

        State state3 = new State();
        state3.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_1));
        state3.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.DIRTY));

        State state4 = new State();
        state4.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_2));
        state4.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.CLEAN));

        State state5 = new State();
        state5.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_2));
        state5.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.DIRTY));

        State state6 = new State();
        state6.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_3));
        state6.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.CLEAN));

        State state7 = new State();
        state7.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_3));
        state7.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.DIRTY));

        State state8 = new State();
        state8.addPerception(new Perception("current location",
                             VacuumEnvironment3x3.LOCATION_4));
        state8.addPerception(new Perception("current location state",
                             VacuumEnvironment3x3.LocationState.CLEAN));
```

```
State state9 = new State();
state9.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_4));
state9.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.DIRTY));

State state10 = new State();
state10.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_5));
state10.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.CLEAN));

State state11 = new State();
state11.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_5));
state11.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.DIRTY));

State state12 = new State();
state12.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_6));
state12.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.CLEAN));

State state13 = new State();
state13.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_6));
state13.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.DIRTY));

State state14 = new State();
state14.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_7));
state14.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.CLEAN));

State state15 = new State();
state15.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_7));
state15.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.DIRTY));

State state16 = new State();
state16.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_8));
state16.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.CLEAN));

State state17 = new State();
state17.addPerception(new Perception("current location",
                     VacuumEnvironment3x3.LOCATION_8));
state17.addPerception(new Perception("current location state",
                     VacuumEnvironment3x3.LocationState.DIRTY));
```

Desde la instancia state0 hasta la instancia state17 se forman todos los estados en los que se podrían encontrar las celdas; un estado está formado por una o muchas percepciones, en este caso a cada estado le corresponden dos percepciones que son: el número de ubicación o celda y el estado en el que se encuentra, que puede ser limpia o sucia.

```
        Situation situation0 = new Situation();
        situation0.setState(state0);
        situation0.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_RIGHT,
                                        "move right"));
        situation0.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_DOWN,
                                        "move down"));

        Situation situation1 = new Situation();
        situation1.setState(state1);
        situation1.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                        "suck"));

        Situation situation2 = new Situation();
        situation2.setState(state2);
        situation2.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_LEFT,
                                        "move left"));
        situation2.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_RIGHT,
                                        "move right"));
        situation2.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_DOWN,
                                        "move down"));

        Situation situation3 = new Situation();
        situation3.setState(state3);
        situation3.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                        "suck"));

        Situation situation4 = new Situation();
        situation4.setState(state4);
        situation4.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_LEFT,
                                         "move left"));
        situation4.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_DOWN,
                                         "move down"));

        Situation situation5 = new Situation();
        situation5.setState(state5);
        situation5.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                         "suck"));

        Situation situation6 = new Situation();
```

```
        situation6.setState(state6);
        situation6.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_RIGHT,
                                          "move right"));
        situation6.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_UP,
                                         "move up"));
        situation6.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_DOWN,
                                          "move down"));

        Situation situation7 = new Situation();
        situation7.setState(state7);
        situation7.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                         "suck"));

        Situation situation8 = new Situation();
        situation8.setState(state8);
        situation8.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_LEFT,
                                         "move left"));
        situation8.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_RIGHT,
                                         "move right"));
        situation8.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_UP,
                                         "move up"));
        situation8.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_DOWN,
                                         "move down"));

        Situation situation9 = new Situation();
        situation9.setState(state9);
        situation9.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                         "suck"));

        Situation situation10 = new Situation();
        situation10.setState(state10);
        situation10.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_LEFT,
                                         "move left"));
        situation10.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_UP,
                                         "move up"));
        situation10.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_DOWN,
                                         "move down"));

        Situation situation11 = new Situation();
        situation11.setState(state11);
        situation11.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                         "suck"));

        Situation situation12 = new Situation();
        situation12.setState(state12);
        situation12.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_RIGHT,
```

```
                                        "move right"));
        situation12.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_UP,
                                        "move up"));

        Situation situation13 = new Situation();
        situation13.setState(state13);
        situation13.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                        "suck"));

        Situation situation14 = new Situation();
        situation14.setState(state14);
        situation14.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_LEFT,
                                        "move left"));
        situation14.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_RIGHT,
                                        "move right"));
        situation14.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_UP,
                                        "move up"));

        Situation situation15 = new Situation();
        situation15.setState(state15);
        situation15.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                        "suck"));

        Situation situation16 = new Situation();
        situation16.setState(state16);
        situation16.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_LEFT,
                                        "move left"));
        situation16.addAction(new
Action(VacuumEnvironment3x3.ACTION_MOVE_UP,
                                         "move up"));

        Situation situation17 = new Situation();
        situation17.setState(state17);
        situation17.addAction(new Action(VacuumEnvironment3x3.ACTION_SUCK,
                                        "suck"));
```

Desde la instancia situation0 hasta la instancia situation17 se forman las situaciones en las que se puede ver involucrado el Agente Aspiradora; una situación está formada por una o muchas percepciones y una o muchas acciones, el porqué de esta estructura es debido a que: si la aspiradora se encuentra en la celda cero únicamente podrá moverse o a la izquierda o abajo, a los demás lados no será posible porque se encuentra con pared es decir con los límites del área por limpiar, este es el motivo por el que las acciones varían dependiendo de la celda en donde

se encuentre el agente pues no en todas las celdas se podrá mover a la derecha, izquierda arriba o abajo según sea el caso.

```
ArrayList<Situation> situations = new ArrayList<Situation>();
situations.add(situation0);
situations.add(situation1);
situations.add(situation2);
situations.add(situation3);
situations.add(situation4);
situations.add(situation5);
situations.add(situation6);
situations.add(situation7);
situations.add(situation8);
situations.add(situation9);
situations.add(situation10);
situations.add(situation11);
situations.add(situation12);
situations.add(situation13);
situations.add(situation14);
situations.add(situation15);
situations.add(situation16);
situations.add(situation17);
```

Como se puede apreciar en las líneas de arriba, se está haciendo una lista de todas las situaciones; esta se hizo con el fin de que cuando se crea el objeto aspiradora se le proporcionen todas las situaciones.

```
VacuumCleaner vacuumCleaner3x3 = new VacuumCleaner(situations);
vacuumCleaner3x3.clean();
System.out.println("Invoca a dowake");
vacuumCleaner3x3.doWake();
```

Por último, se crea el objeto vacuumCleaner3x3 que representa al Agente Aspiradora, al cual se le pasará, como parámetro de su constructor, la lista de situaciones para su entorno. Para luego, inicializar los atributos currentLocation y currentLocationState por medio del método clean() de la clase VacuumCleaner y enseguida se despierta el hilo que pone en ejecución al Agente Aspiradora.

6.6.2.4 Salida en Consola

Como es posible ver en la figura 6.32, una vez ejecutada la clase TestVacuumCleaner el agente se ubicó al inicio en la celda 3 y al encontrarla limpia

decidió moverse hacia arriba, es decir, a la celda 0. A esta ubicación, se le asignó como estado aleatorio el valor de limpio, y así, el agente tuvo que moverse hacia su derecha, ubicándose en la celda 1. Esta celda se encontraba sucia y la acción ejecutada fue la de limpiar. Finalmente, el agente siguió de manera sucesiva moviéndose entre celdas y limpiando aquellas que estuvieran sucias.

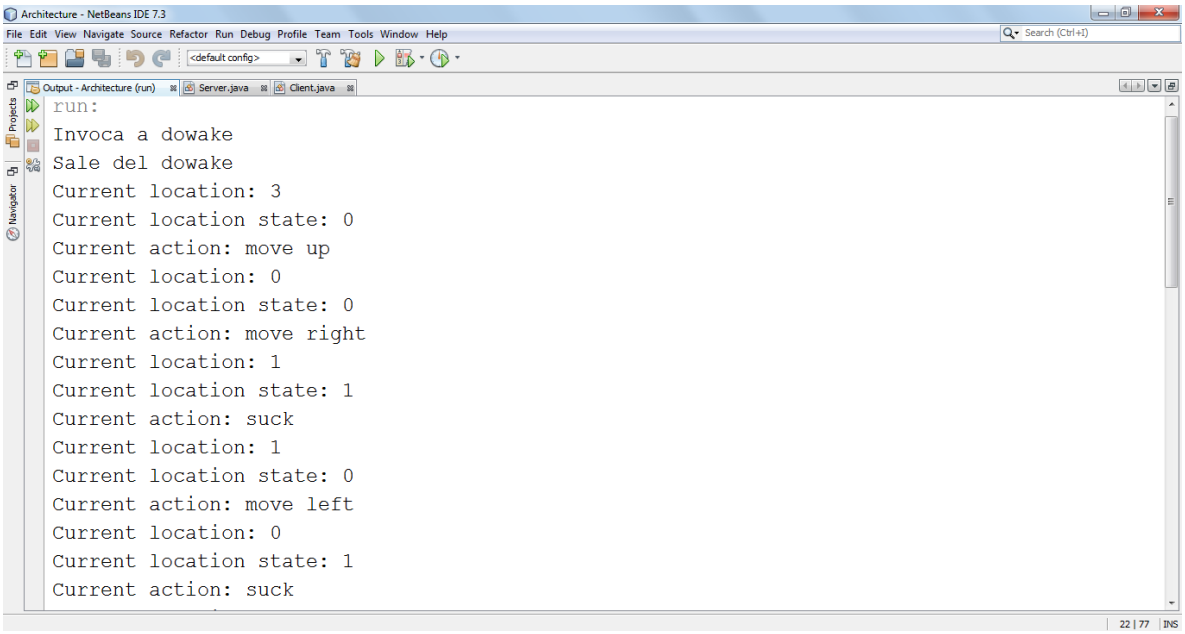

Figura 6.32: Salida en Consola del Agente Aspiradora.

6.6.3 Simulación de un Escenario Peer to Peer

Por medio del caso de estudio que se revisará a continuación, se podrá comprobar la funcionalidad Peer to Peer que tiene la arquitectura presentada en este capítulo. Para ello, fue necesario programar dos agentes nombrados PeerToPeerAgent1 y PeerToPeerAgent2, los cuales sostendrán una pequeña conversación con un máximo de cuatro preguntas hechas por el Agente PeerToPeerAgent1 al Agente PeerToPeerAgent2. De esta manera, el Agente PeerToPeerAgent1 tomará al inicio el papel de cliente y el agente PeerToPeerAgent2 el de servidor. Sin embargo, en cualquier momento el Agente PeerToPeerAgent2 puede tomar el papel de cliente y luego regresar al de servidor, con lo cual el Agente PeerToPeerAgent1 debe cambiar

su papel para seguir con la comunicación, es decir, volverse un servidor y luego regresar al papel de cliente.

6.6.3.1 Agente PeerToPeerAgent1

```
package Test;

import ACLMessage.ACLMessage;
import ACLMessage.DeliveringMessageException;
import ACLMessage.MessageTransportService;
import ACLMessage.UnknownReceiverException;
import AbstractAgentDefinition.AbstractAgent;
import AbstractAgentDefinition.Action;
import java.io.IOException;
```

Esta clase dará el comportamiento al Agente PeerToPeerAgent1, para lograrlo se utilizan las siguientes clases: ACLMessage para formar mensajes, la clase MessageTransportService para envío y recepción de los mismos, las clases DeliveringMessageException y UnknownReceiverException que indican los casos en que surja algún error al momento de enviar el mensaje, y finalmente, las clases AbstractAgent y Action para formar al agente.

```
public class PeerToPeerAgent1 extends AbstractAgent {
    private MessageTransportService messageTransportService;
    private ACLMessage message;
    private int counter;
```

Este agente es simple, ya que hereda de la clase AbstractAgent lo que quiere decir que gracias a las propiedades del padre se crea un hilo para controlarlo desde que se crea hasta que termine la conversación y después es terminado. Los atributos que aquí se declaran son de tipo MessageTransporService para realizar el intercambio de mensajes entre los agentes, ACLMessage para formar el mensaje según lo especifica FIPA y un atributo de tipo int que llevará el conteo del número de preguntas que se le harán al Agente PeerToPeerAgent2.

```
    public PeerToPeerAgent1(String id, MessageTransportService
messageTransportService) {
```

```
        super();
        this.id = id;
        this.messageTransportService = messageTransportService;
        this.messageTransportService.addMessageListener(this);
        counter = 0;
        setNextAction(new Action(1, "generate request"));
    }
```

Por medio del constructor, se recibirán los parámetros necesarios para permitir la creación del agente asignándole un identificador y los valores que solicita el Servicio de Transporte de Mensajes para que por medio de él se pueda comunicar con el Agente PeerToPeerAgent2. También, se inicializa el contador encargado de llevar el número de las preguntas que se le hace al Agente PeerToPeerAgent2; así como asignar la primera acción que realizará la cual es: “generar una petición”, es decir, realizarle una primera pregunta al Agente PeerToPeerAgent2.

```
    public void setNextAction(Action action) {
        previousAction = currentAction;
        currentAction = action;
    }

    public Action getNextAction() {
        return currentAction;
    }
```

Como se puede observar en la parte de arriba, hay dos métodos, el primero se usa para establecer la acción que acaba de ser ejecutada y la siguiente acción a ser realizada por el agente y el segundo devuelve esta última acción actual que debe ser ejecutada.

```
    public void behavior() {
        if(getNextAction().getId() == 1 && counter < 4) {
            System.out.println("Current action: " +
currentAction.getName());
            doWait(3000);
            generateRequest();
        }
        else {
            setAlive(false);
        }
    }
```

El método behavior(), es heredado de la clase AbstractAgent e implementado en esta clase; como su nombre lo dice, dentro de este método dependiendo se determina el comportamiento del agente. Este funciona de la siguiente manera: lo primero que hace es validar que la acción actual sea “generar una petición” y que no se hagan más de cuatro preguntas, posteriormente se espera tres segundos para generar la petición.

```
        public void generateReply() {
        double randomValue;
        String reply;
        ACLMessage aCLReplyMessage;
        if(getNextAction().getId() == 5) {
            System.out.println("Current action: " +
currentAction.getName());
            aCLReplyMessage = new ACLMessage();
            aCLReplyMessage.setSender(getId());
            aCLReplyMessage.setReceiver(message.getSender());
            aCLReplyMessage.setPerformative("action");
            randomValue = Math.random();
            if(randomValue > 0.0 && randomValue <= 0.33) {
               reply = "Because I want to be your friend";
            }
            else if(randomValue > 0.33 && randomValue <= 0.66) {
               reply = "I just want to know about you";
            }
            else {
               reply = "It is a simple question for you";
            }
            aCLReplyMessage.setContent(reply);
            currentAction = new Action(2, "send message");
            sendMessage(aCLReplyMessage);
        }
    }
```

Arriba tenemos al método generateReply() cuya funcionalidad es; como su nombre lo dice, generar una respuesta en caso de que el Agente PeerToPeerAgent2 haya formulado una pregunta como respuesta a una pregunta que se la haya realizado. Primero se declaran dos variables, una para generar un valor aleatorio y la otra para almacenar la respuesta que se enviará al Agente PeerToPeerAgent2 y también se instancia un objeto de tipo ACLMessage para poder formar el mensaje que será enviado. Después, se validará que la acción actual sea “generar respuesta” para entonces poder formar el mensaje con los parámetros especificados por FIPA y

aleatoriamente enviar una respuesta al Agente PeerToPeerAgent2. Ahora la acción actual cambia a “enviar mensaje”, por lo tanto se invoca el método sendMessage().

```
public void generateRequest() {
    double randomValue;
    String content;
    ACLMessage aCLRequestMessage;
    aCLRequestMessage = new ACLMessage();
    aCLRequestMessage.setSender(getId());
    aCLRequestMessage.setReceiver("p2pAgent2");
    aCLRequestMessage.setPerformative("action");
    randomValue = Math.random();
    if(randomValue > 0.0 && randomValue <= 0.25) {
        content = "What is your name?";
    }
    else if(randomValue > 0.25 && randomValue <= 0.5) {
        content = "How old are you?";
    }
    else if(randomValue > 0.5 && randomValue <= 0.75) {
        content = "Which is your favourite color?";
    }
    else {
        content = "Tell me about your hobbits";
    }
    aCLRequestMessage.setContent(content);
    setNextAction(new Action(2, "send message"));
    sendMessage(aCLRequestMessage);
    counter++;
}
```

A continuación, se explicará el método generateRequest(), donde se va a formular la pregunta que será enviada al Agente PeerToPeerAgent2. Empezando por declarar dos variables, la primera para generar un valor aleatorio y la segunda para almacenar la respuesta que se enviará al Agente PeerToPeerAgent2. También, se instancia un objeto de tipo ACLMessage para poder formar el mensaje que será enviado. Posteriormente, se formará el mensaje con los parámetros especificados por FIPA, para después generar un valor aleatorio y con ello determinar la pregunta que será enviada al Agente PeerToPeerAgent2. Finalmente, se cambia la acción actual a “enviar mensaje” para así invocar al método sendMessage() y aumentar en uno el contador de preguntas realizadas.

```
public void processMessage() {
    if(getNextAction().getId() == 4) {
```

```
            System.out.println("Current action: " +
currentAction.getName());
            if(message.getContent().equals("Why do you want to know it?")
               || message.getContent().equals("Is there any reason to
answer your question?")
               || message.getContent().equals("Tell me your real intention
of knowing about me")) {
                System.out.println("I received a request instead of a reply
and is: " + message.getContent());
                currentAction = new Action(5, "generate reply");
                generateReply();
            }
            else {
                System.out.println("The reply is: " +
message.getContent());
                setNextAction(new Action(1, "generate request"));
            }
        }
    }
```

El método processMessage() procesará el mensaje que llegue, validando que la acción actual sea “procesar mensaje” de manera que verifica si el contenido del mensaje enviado por el Agente PeerToPeerAgent2 es una pregunta; si es así, la acción actual cambia a “generar una respuesta” y se invoca al método generateReply(). De lo contrario, quiere decir que el contenido del mensaje corresponde a una respuesta; con ello, se muestra el contenido del mensaje y el método setNextAction() vuelve a la primera acción que es “generar una petición”.

```
    public void sendMessage(ACLMessage message) {
        int id;
        id = currentAction.getId();
        if(getNextAction().getId() == 2) {
            System.out.println("Current action: " +
currentAction.getName());
            try {
                messageTransportService.sendMessage(message);
                if(getPreviousAction().getId() == 1) {
                    setNextAction(new Action(3, "wait for a reply"));
                    doWait();
                    processMessage();
                }
                else {
                    setNextAction(new Action(1, "generate request"));
                }
            } catch(DeliveringMessageException | UnknownReceiverException
                    | IOException | NullPointerException e) {
                System.err.println("Error: " + e.getMessage());
            }
        }
```

Este método sirve para enviar el mensaje una vez que se ha formado, lo que hace primero es validar si la acción actual es “enviar mensaje”; si es así, por medio de la acción previa verifica si lo que generó fue una petición, entonces cambia la acción actual a “esperar por una respuesta” y el estado del agente se pone en espera por medio del método doWait() hasta que le responda el Agente PeerToPeerAgent2, una vez que llega el mensaje se invoca al método processMessage(). Por el contrario, si la acción previa no fue “generar una petición” quiere decir que fue “generar respuesta” lo que quiere decir que, la acción actual cambia a “esperar una petición”.

```
    public void processACLMessage(ACLMessage message) {
        if(getNextAction().getId() == 3) {
            System.out.println("Current action: " +
currentAction.getName());
            this.message = message;
            setNextAction(new Action(4, "process message"));
            doWake();
        }
    }
}
```

El método processACLMessage() se hereda de la clase AbstractAgent y se implementa en esta clase, como se puede ver valida que la acción actual sea “esperar respuesta”, asigna el mensaje recibido a un objeto de tipo ACLMessage, actualiza la acción por “procesar mensaje” y despierta al Agente PeerToPeerAgent1, recordando que cuando se envió la petición, se puso en estado de espera.

6.6.3.2 Agente PeerToPeerAgent2

```
package Test;

import ACLMessage.ACLMessage;
import ACLMessage.DeliveringMessageException;
import ACLMessage.MessageTransportService;
import ACLMessage.UnknownReceiverException;
import AbstractAgentDefinition.AbstractAgent;
import AbstractAgentDefinition.Action;
import java.io.IOException;
```

Esta clase dará el comportamiento al Agente PeerToPeerAgent2, para lograrlo se utilizan las siguientes clases: ACLMessage para formar mensajes, la clase

MessageTransportService para envío y recepción de los mismos, las clases DeliveringMessageException y UnknownReceiverException que indican los casos en que surja algún error al momento de enviar el mensaje, y finalmente, las clases AbstractAgent y Action para formar al agente.

```
public class PeerToPeerAgent2 extends AbstractAgent {
    private MessageTransportService messageTransportService;
    private ACLMessage message;
    private int counter;
```

Ahora bien, como este agente es simple hereda de la clase AbstractAgent lo que quiere decir que gracias a las propiedades del padre, se crea un hilo para controlar al agente desde que se crea hasta que finaliza la conversación. Los atributos que arriba se definen son de tipo MessageTransporService para realizar el intercambio de mensajes entre los agentes, ACLMessage para formar el mensaje según lo especifica FIPA y un atributo de tipo int que llevará el conteo del número de respuestas que se le darán al Agente PeerToPeerAgent1.

```
    public PeerToPeerAgent2(String id, MessageTransportService
messageTransportService) {
        super();
        this.id = id;
        this.messageTransportService = messageTransportService;
        this.messageTransportService.addMessageListener(this);
        counter = 0;
        setNextAction(new Action(1, "wait for a request"));
    }
```

A través del constructor, se recibirán los parámetros necesarios para permitir la creación del agente especificando su identificador y los valores que requiere el Servicio de Transporte de Mensajes; para que, por medio de él se pueda comunicar con el Agente PeerToPeerAgent1. La variable counter se inicia y tiene la función de llevar el registro de las respuestas que le envía al Agente PeerToPeerAgent1; también, por medio del método setNextAction() es asignada la primera acción que realizará, la cual es: “esperar una petición” es decir esperar una pregunta hecha por el Agente PeerToPeerAgent1.

```
    public void setNextAction(Action action) {
        previousAction = currentAction;
        currentAction = action;
    }
    public Action getNextAction() {
        return currentAction;
    }
```

En el código que se encuentra en la parte de arriba hay dos métodos, el primero se usa para establecer la acción que acaba de ser ejecutada y la siguiente acción a ser realizada por el agente y el segundo devuelve esta última acción actual que debe ser ejecutada.

```
    public void behavior() {
        int id;
        id = getNextAction().getId();
        if((id == 1 || id == 5) && counter < 4) {
            System.out.println("Current action: " +
currentAction.getName());
            doWait();
            processMessage();
        }
        else {
            setAlive(false);
        }
    }
```

El método behavior(), se hereda de la clase AbstractAgent y es implementado en esta clase; como su nombre lo dice, con este método se determina el comportamiento que tendrá el agente dependiendo de lo que el programador requiera. Por lo tanto, se programó para que trabaje de la siguiente manera: lo primero que hace es verificar que la acción actual sea “esperar una petición” ó “esperar una respuesta” y que no haya enviado más de cuatro respuestas, posteriormente se espera por tiempo indefinido hasta que llegue un mensaje por medio del método processACLMessage() para finalmente invocar al método processMessage() y de esta manera procesar el mensaje recibido, por otro lado cuando no esté esperando una petición o una respuesta el agente cambia su estado de vivo a false para que sea destruido.

```
    public void generateReply() {
        String reply;
```

```
        ACLMessage aCLReplyMessage;
        if(getNextAction().getId() == 6) {
            System.out.println("Current action: " +
currentAction.getName());
            aCLReplyMessage = new ACLMessage();
            aCLReplyMessage.setSender(getId());
            aCLReplyMessage.setReceiver(message.getSender());
            aCLReplyMessage.setPerformative("action");
            if(message.getContent().equals("What is your name?")) {
                reply = "I do not have a name, I am an agent";
            }
            else if(message.getContent().equals("How old are you?")) {
                reply = "I am very young, I was born as an intelligent
entity";
            }
            else if(message.getContent().equals("Which is your favourite
color?")) {
                reply = "I have not eyes, any color is fine for me";
            }
            else if(message.getContent().equals("Tell me about your
hobbits")) {
                reply = "It depends on my behavior";
            }
            else {
                reply = "I know it now, Anything else you want to know";
            }
            aCLReplyMessage.setContent(reply);
            setNextAction(new Action(4, "send message"));
            sendMessage(aCLReplyMessage);
        }
    }
```

En las líneas de código de arriba se puede observar que, el método generateReply() tiene la función de generar una respuesta en caso de que el Agente PeerToPeerAgent1 haya formulado una petición (en dicha petición formula una pregunta). Primeramente, se declara una variable para almacenar la respuesta que será enviada al Agente PeerToPeerAgent1, posteriormente, se declara un objeto de tipo ACLMessage para poder formar el mensaje que será enviado. Después, se hará una comparación entre del contenido del mensaje recibido con las posibles preguntas que pudo haber formulado el Agente PeerToPeerAgent1; finalmente, la acción actual cambia a “enviar mensaje”, por lo tanto, es invocado el método sendMessage().

```
    public void generateRequest() {
        double randomValue;
        String request;
        ACLMessage aCLRequestMessage;
```

```
        if(getNextAction().getId() == 3) {
            System.out.println("Current action: " +
currentAction.getName());
            aCLRequestMessage = new ACLMessage();
            aCLRequestMessage.setSender(getId());
            aCLRequestMessage.setReceiver(message.getSender());
            aCLRequestMessage.setPerformative("action");
            randomValue = Math.random();
            if(randomValue > 0.0 && randomValue <= 0.33) {
               request = "Why do you want to know it?";
            }
            else if(randomValue > 0.33 && randomValue <= 0.66) {
               request = "Is there any reason to answer your question?";
            }
            else {
                request = "Tell me your real intention of knowing about
me";
            }
            aCLRequestMessage.setContent(request);
            setNextAction(new Action(4, "send message"));
            sendMessage(aCLRequestMessage);
        }
    }
```

En los siguientes dos párrafos, se explicará el método generateRequest(), donde se va a formular la pregunta que será enviada al Agente PeerToPeerAgent1. Empezando por declarar dos variables, la primera para generar un valor aleatorio y la segunda para almacenar la respuesta que se enviará al Agente PeerToPeerAgent1; también, se define un objeto de tipo ACLMessage para poder formar el mensaje que será enviado. Posteriormente, se formará el mensaje con los parámetros especificados por FIPA, para después generar un valor aleatorio y con ello determinar la pregunta que será enviada al Agente PeerToPeerAgent1, ahora se cambia la acción actual por “enviar mensaje” para finalmente invocar el método sendMessage().

```
    public void sendMessage(ACLMessage message) {
        int id;
        if(getNextAction().getId() == 4) {
            System.out.println("Current action: " +
currentAction.getName());
            try {
                messageTransportService.sendMessage(message);
                if(getPreviousAction().getId() == 3) {
                    setNextAction(new Action(5, "wait for a reply"));
                }
                else {
                    setNextAction(new Action(1, "wait for a request"));
```

```
                }
            } catch(DeliveringMessageException | UnknownReceiverException
                    | IOException | NullPointerException e) {
                System.err.println("Error: " + e.getMessage());
            }
        }
    }
```

El método que se tiene en las líneas de arriba sirve para enviar el mensaje una vez que se ha formado, lo que hace primero es validar si la acción actual es “enviar mensaje”; si es así, por medio de la acción previa verifica si lo que generó fue una petición, entonces cambia la acción actual a “esperar por una respuesta”, de lo contrario, quiere decir que la acción previa fue “generar petición” por lo tanto, la acción actual cambia a “esperar por una petición”.

```
    public void processMessage() {
        if(getNextAction().getId() == 2) {
            System.out.println("Current action: " +
currentAction.getName());
            if(getPreviousAction().getId() == 1) {
                if(Math.random() < 0.5) {
                    System.out.println("I will reply with a request");
                    setNextAction(new Action(3, "generate request"));
                    generateRequest();
                }
                else {
                    setNextAction(new Action(6, "generate reply"));
                    generateReply();
                }
                counter++;
            }
            else {
                System.out.println("The reply is: " +
message.getContent());
                setNextAction(new Action(1, "wait for a request"));
            }
        }
    }
```

Con el método processMessage(), se decidirá qué hacer con el mensaje que haya llegado al método processACLMessage() validando que la acción actual sea “procesar mensaje”. Después, se verifica si la acción previa era “esperando petición” y ahora por medio de un valor aleatorio se decide si responde a la petición realizada por el Agente PeerToPeerAgent1 o genera una petición es decir una pregunta, dando un comportamiento P2P; si es así, la acción actual cambia a “generar una petición” y

se invoca al método generateRequest(). De lo contrario, quiere decir que el agente decidió responder a la petición; por lo tanto, el método setNextAction() actualiza la acción a “generar respuesta” e invoca al método generateReply(). Posteriormente, se incrementa el contador de respuestas, finalmente, si la acción actual no era “esperar por una petición” se imprime el contenido del mensaje y la acción actual del agente vuelve a “esperar petición”.

```
    public void processACLMessage(ACLMessage message) {
        this.message = message;
        setNextAction(new Action(2, "process message"));
        doWake();
    }
}
```

Para finalizar con esta clase, en la parte de arriba se encuentra el método processACLMessage() heredado de la clase AbstractAgent e implementado aquí, ya que por medio él se recibe el mensaje que envía el Agente PeerToPeerAgent1, como se puede ver se recibe el mensaje y se asigna el mismo al atributo message que es de tipo ACLMessage, luego, se actualiza la acción por “procesar mensaje” y se despierta al agente. Cabe mencionar, que cuando se envió la petición el agente se puso en estado de espera.

6.6.3.3 EscenarioTestPeerToPeer

```
package Test;

import ACLMessage.DeliveringMessageException;
import ACLMessage.MessageTransportService;
import ACLMessage.UnknownReceiverException;
import java.io.IOException;
```

Esta es la clase principal, encargada de poner en ejecución la conversación entre los dos agentes de tipo PeerToPeerAgent1 y PeerToPeerAgent2, respectivamente. Estos serán llamados p2pAgent1 y p2pAgent2. Para lograr su interacción, se utilizan las clases: MessageTransportService para envío y recepción de mensajes, las clases DeliveringMessageException, UnknownReceiverException y IOException para casos donde surja algún error al momento de iniciar el intercambio de mensajes.

```
public class TestPeerToPeer {
    public static void main(String[] args) {
        MessageTransportService messageTransportService;
        PeerToPeerAgent1 p2pAgent1;
        PeerToPeerAgent2 p2pAgent2;
        try {
            messageTransportService =
MessageTransportService.getInstance();
            messageTransportService.addRemoteMessageTransportService(
                    "127.0.0.1", 6000);
            p2pAgent1 = new PeerToPeerAgent1("p2pAgent1",
messageTransportService);
            p2pAgent2 = new PeerToPeerAgent2("p2pAgent2",
messageTransportService);
        } catch(IOException | NullPointerException
                | UnknownReceiverException | DeliveringMessageException e)
{
            System.err.println(e.getMessage());
        }
    }
}
```

Dentro del método main(String[] args) se definen variables de tipo MessageTransportService, PeerToPeerAgent1 y PeerToPeerAgent2, para que posteriormente en el bloque try se creen instancias del transporte de mensajes para la comunicación entre los dos agentes p2pAgent1 y p2pAgent2. Finalmente, se tienen las posibles excepciones que puedan surgir en caso de algún error.

6.6.3.4 Salida en Consola

A continuación, en las figuras 6.33 y 6.34 será posible visualizar las salidas en pantalla con los resultados de la conversación para cada agente, cabe destacar que cada uno se ejecutó en una máquina distinta por ello las preguntas y respuestas aparecen en pantallas diferentes. La figura 6.33 pertenece al agente llamado p2pAgent1, es decir el que inicialmente toma el papel de cliente, de manera que la primer acción que realiza es “generar una petición”, la segunda “enviar el mensaje”, la tercera “esperar una respuesta” por parte del agente nombrado como p2pAgent2 y la tercera que es “procesar el mensaje” donde va a identificar qué tipo de mensaje recibió, el cual puede ser de dos tipos: una respuesta a la pregunta o una petición en su lugar.

Como es posible ver en la figura 6.33, el Agente p2pAgent2 respondió con una petición donde pregunta “¿para qué quieres saberlo?”, lo que quiere decir que ahora el agente p2pAgent1 elegirá ser servidor ya que tiene que proporcionar una respuesta a la petición que recibe. Después envía la respuesta, para después poder formular una pregunta y el agente p2pAgent2 le responda; de manera que, este ciclo se repite hasta cumplir las cuatro preguntas que tiene permitido realizar el agente p2pAgent1.

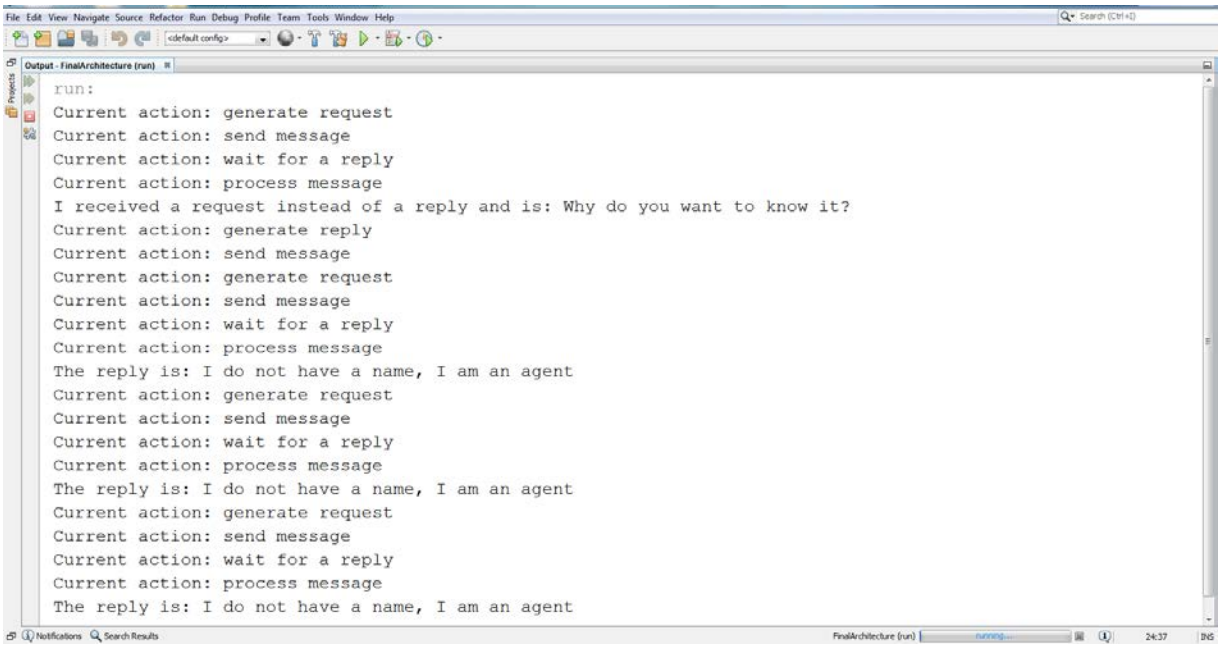

Figura 6.33: Ejemplo de Salida en Consola del Agente p2pAgent1.

En relación con la figura anterior, está la figura 6.34 que muestra el resultado de ejecutar al agente p2pAgent2 quién inicialmente tiene el papel de servidor, es decir, quien responderá a las preguntas del agente p2pAgent1. La primera acción que realiza es “esperar una petición”. Una vez que le llega un mensaje de pregunta decidirá si va a responder la pregunta, o en su defecto, si cambia su papel de servidor por el de cliente y responde con una petición de pregunta y por lo tanto, espera recibir una respuesta. Esto quiere decir, que cambia la acción actual a

“generar petición”, después envía el mensaje, espera una respuesta, y finalmente, recibe el mensaje que es la respuesta a su petición.

Por último, como recibe una respuesta, ahora su acción actual cambia nuevamente a “esperar una petición” hasta que le haga una pregunta el agente p2pAgent1, le llega una petición, procesa el mensaje, genera una respuesta, la envía y así se repite el ciclo hasta completar las cuatro preguntas que le puede hacer el agente p2pAgent1.

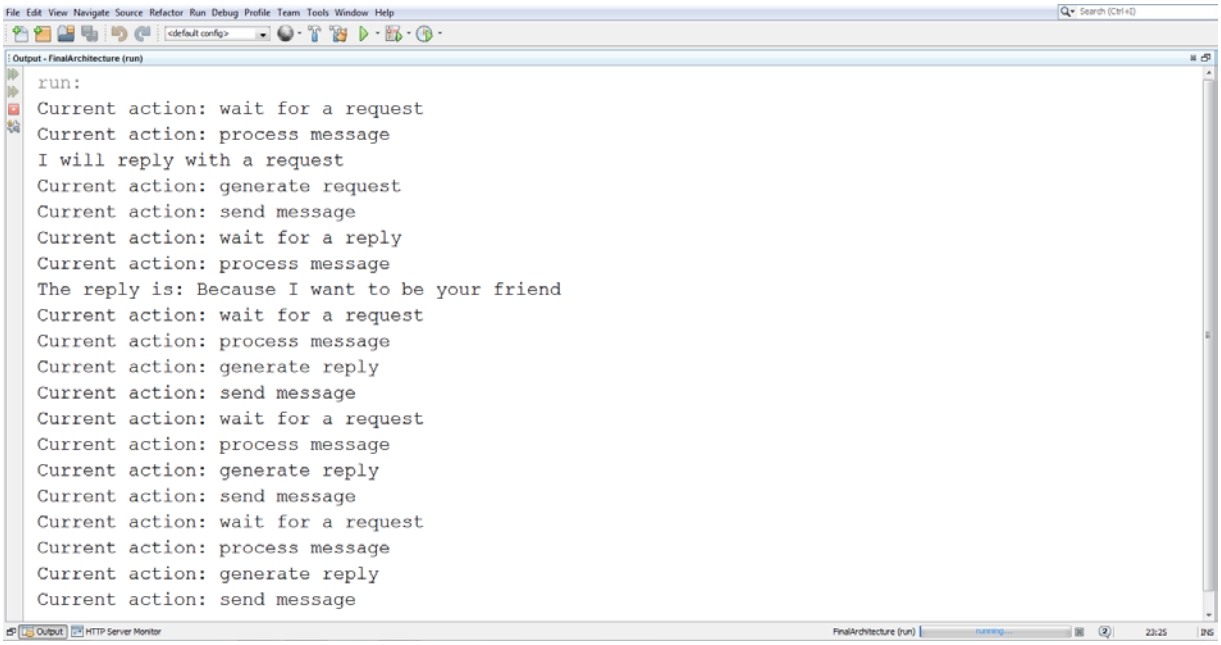

Figura 6.34: Ejemplo de Salida en Consola del Agente p2pAgent2.

Capítulo 7. Conclusiones

Para los desarrolladores que se dedican a generar simulaciones de situaciones sociales usando al lenguaje Java, resulta de gran utilidad el tener a la mano uno o más paquetes de clases e interfaces que les ahorren el trabajo de programar desde el inicio a cada agente que requieran; dependiendo del tipo que les resulte más apropiado, así como también disponer de un mecanismo de comunicación que permita que estos interactúen en su entorno. Es por ello que en este trabajo, se propuso una nueva arquitectura de agentes heterogéneos, en la cual se cumplió de manera satisfactoria con cada uno de los siguientes objetivos que fueron especificados en la sección 1.4 de este trabajo de tesis:

1. Se generó la arquitectura cumpliendo con las expectativas de un entono P2P bajo el esquema del modelo cliente-servidor, implementándola en el lenguaje Java. Haciendo notar que, en el último caso de estudio (ver sección 6.6.3) se comprobó que es posible mediante esta arquitectura, que los agentes puedan comunicarse aun cuando estén alojados en máquinas distintas y a su vez tomar el papel tanto de cliente como de servidor según sus necesidades en determinado momento en su entorno.

2. Se implementaron en la arquitectura los protocolos especificados por FIPA, así como el modelo de comunicación orientado a mensajes entre agentes inteligentes dado por el W3C.

    a. Para cumplir este objetivo, se programó una serie de clases para cubrir los estándares proporcionados por FIPA, estas clases fueron las siguientes: AbstractAgent, DeliveringMessageException, ExceededReceiversException, MalFormedMessageException y UnknownReceiverException donde se cumple la “Especificación para Administración de Agentes” (Agent Management

Specification, de su nombre en inglés). Esta especificación dice que todo agente tiene un ciclo de vida, el cual fue implementado por medio de hilos de control; así como también un registro de los agentes que se van creando y finalmente las excepciones en caso de que ocurra algún error.

b. Con la clase ACLMessage, ACLMessageReceiver y ACLMessageSender se cubrió la “Especificación para la Estructura del Mensaje ACL” (ACL Message Structure Specification, de su nombre en inglés). Esta especificación es encargada de proporcionar las partes que forma y debe cumplir un mensaje para poder ser enviado de un agente a otro. En la clase ACLMessage se incluyeron performativas o acciones realizables, recordando que son aquellas posibles que puede llevar a cabo un agente en determinado momento en su entorno, las performativas se encuentran en la “Especificación de la Biblioteca del Acto Comunicativo” (Communicative Act Library Specification, de su nombre en inglés).

c. Ahora bien, por medio de la clase MessageTransportService y la interfaz IACLMessageListener se cubre la “Especificación para el Servicio de Transporte de Mensajes para Agentes” (Agent Message Transport Service Specification, de su nombre en inglés). Por medio de estas se cubrió el Modelo Orientado a Mensajes dado por el W3C que en conjunto con FIPA dan la estructura que deben tener los mensajes para que puedan ser enviados a través de la web.

3. Se crearon varios paquetes de clases e interfaces para los siguientes tipos de agentes inteligentes: reactivo, cognitivo, híbrido, basado en metas, basado en modelos, basado en su utilidad y EBDI. Con los cuales, se generaron casos de estudio, en donde se incluyó uno que demuestra una interacción heterogénea.

a. Ahora bien, para lograr el cumplimiento de este último objetivo; como primer paso, se realizó la clasificación de los agentes inteligentes más destacados.

Una vez teniendo la clasificación junto con las características de cada tipo de agente, se diseñó una jerarquía donde se relacionaron y agruparon los agentes dependiendo de las características que estos comparten y que los distinguen, como lo fueron: el historial de percepciones, reglas de acción condición, objetivos, planes, etc.

b. En el primer caso de estudio, el servidor tuvo un comportamiento de tipo reactivo y los agentes clientes uno de tipo cognitivo. Ahora bien, con ello se demostró que pueden interactuar agentes heterogéneos (un agente servidor reactivo y dos agentes clientes cognitivos) y a su vez pueden comunicarse aun cuando estén alojados en máquinas distintas. Gracias a que se trabajó con hilos es posible que dos clientes hagan peticiones al mismo tiempo al servidor y éste les responda de manera satisfactoria sin ningún problema conforme las vaya recibiendo. Cabe señalar, que fue suficiente demostrar la funcionalidad de la arquitectura solo con estos dos tipos de agentes puesto que todos los demás heredan del mismo agente padre principal que es la clase AbstractAgent; lo que quiere decir, que las características básicas con las que cuenta todo agente son heredadas de esta clase padre y mediante especializaciones de herencia se fueron creando los demás tipos.

c. En el segundo caso de estudio, se empleó un agente aspiradora de tipo reactivo que estuvo integrado por acciones y percepciones, mismas que formaron sus estados y situaciones en entorno de nueve celdas. Este caso de estudio, se programó con la finalidad de dar un ejemplo sobre lo tedioso que puede llegar a ser el trabajar con agentes reactivos, sobre todo en escenarios donde el número de estados y situaciones es de un tamaño grande, lo cual puede aligerarse más no siempre hacerlo tan preciso como lo es usando un agente de tipo cognitivo como reemplazo.

Las ventajas que brinda esta arquitectura son las siguientes:

- Es multiplataforma debido a que se utilizó el lenguaje java para programarla.

- Los agentes se pueden ubicar en una sola máquina, de manera local; o en varias, de manera remota.
- Debido al uso de sockets TCP/IP para el servicio de transporte de mensajes, hay confiabilidad en el envío-recepción de los mismos.
- Pueden interactuar agentes de diferentes tipos y el ciclo de vida de los agentes también puede manipularse de acuerdo a lo requerido en los escenarios que sea necesario implementar.

Capítulo 8. Trabajo Futuro

Como parte de la continuación de este trabajo de tesis, se debe completar la implementación de los modelos arquitectónicos faltantes del W3C ya que de momento sólo se implementó el modelo orientado a mensajes y los restantes por ser llevados a cabo son:

- Modelo de servicios: el cual tiene la función de explicar las relaciones entre un agente y los servicios y las peticiones que realiza es decir la acciones que tiene que realizar el agente cuando envía o recibe un mensaje.
- Modelo de recursos: se enfoca en las características esenciales y relevantes de los servicios a lo que se le llama recursos, como podría ser la URL de una página web.
- Modelo de políticas: se encarga de las restricciones pertenecientes al servicio web con el fin de entregar un servicio seguro y de calidad.

Por otro lado, también en la parte de los agentes hace falta implementar al menos otros tres tipos de agentes:

- Agente interfaz: que tiene la capacidad de interactuar con los usuarios proporcionándole asistencia multimedia por medio de una o más interfaces gráficas de usuario.
- Agente de aprendizaje **[11]:** que como su nombre lo dice aprende haciendo mejoras respecto a su historial de acciones.
- Agente colaborativo **[37]:** el cuál va a poder trabajar en conjunto con uno o más agentes para alcanzar objetivos complejos que son difíciles o imposibles de alcanzar por un solo agente.

Referencias

[1] “Java Agent DEvelopment Framework” (JADE). [Fecha de consulta: 19 Abril 2013] Disponible en: http://jade.tilab.com/

[2] Jack Autonomous Decision-Making Software. [Fecha de consulta: 22 Abril 2013] Disponible en: http://www.agent-software.com.au/products/jack/#.UYLdy7Vg9AF

[3] Any Logic “Multimethod Simulation Software” The only simulation tool that supports Discrete Event, Agent Based, and System Dynamics Simulation. [Fecha de consulta: 2 Mayo 2013] Disponible en: http://www.anylogic.com/

[4] Remondino, M. (2004). Analysis of Agent Based Paradigms for Complex Social Systems Simulation. *Universita di Torino*.

[5] Macal, C. M., & North, M. J. (2010). Tutorial on agent-based modeling and simulation. *Journal of Simulation*, *4*(3), 151-162.

[6] The Foundation for Intelligent Physical Agents (FIPA) [Fecha de consulta: 25 Enero 2013] Disponible en: http://www.fipa.org/

[7] World Wide Web Consortium (W3C) [Fecha de consulta: 15 Marzo 2013] Disponible en: http://www.w3.org/TR/ws-arch/#id2268743

[8] NetBeans. [Fecha de consulta: 15 Agosto 2013] Disponible en: https://netbeans.org/

[9] Tobias,R., Hofmann, C.(2004). Evaluation of free Java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation,* vol. 7.

[10] Orozco Aguirre, H.R., (2010). Making Conscious Virtual Humans with Personality an Emotional Intelligence. (Tesis de doctorado). Centro de Investigación y Estudios Avanzados del I.P.N., Unidad Guadalajara.

[11] Russell, S., Norving, P., (2010). Artificial Intelligence a Modern Approach (3ª Ed.). Prentice Hall Series in Artificial Intelligence.

[12] P. Maes. Modeling adaptive autonomous agents. Artifcial Life, 1:135 {162, 1994}.

[13] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. Knowledge Engineering Review, 10:115 {152, 1995}.

[14] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. Pages 21-35. Springer-Verlag, 1996.

[15] Fuente: [Imagen de las partes que componen un agente EBDI]. Recuperado el 3 de Enero de 2014, de: https://www.google.com.mx/search?q=diagrama+de+un+agente+EBDI&biw=1366&bih=657&tbm=isch&tbo=u&source=univ&sa=X&ei=axMPVMO5M8yI8gG2m4DgDA&ved=0CBoQsAQ#tbm=isch&q=diagrama+de+un+agente+BDI&facrc=_&imgdii=_&imgrc=LtMjewtAdtBxOM%253A%3BkhknjWOzml5NeM%3Bhttp%253A%252F%252Fetherpad.proyectolatin.org%252Fup%252Ff2c189f597b25b08330d8f93016f9e7d.jpg%3Bhttp%253A%252F%252Fescritura.proyectolatin.org%252Finteligencia-artificial%252Farquitectura-de-agentes%252F%3B960%3B720

[16] Alkhateeb, F., Al Maghayreh, E., & Doush, I. A. (2011). Multiagent Systems–Modeling, Interactions, Simulations and CASE studies.

[17] Tapia Martínez, D. (2009). *Arquitectura Multiagente para Entornos de Inteligencia Ambiental.* (Tesis doctoral, Universidad de Salamanca). Recuperado de

http://gredos.usal.es/jspui/bitstream/10366/76358/1/DIA_Tapia_Martinez_DI_Arquitectura_multiagente.pdf

[18] Chaw, E. E. (2013). Naïve Bayesian Learning based Multi Agent Architecture for Telemedicine. *International Journal of Innovation and Applied Studies*, 2(4), 412-422.

[19] Suri, D., Howell, A., Schmidt, D., Biswas, G., Kinnebrew, J., Otte, W., Shankaran, N. (2006). A Multi-Agent Architecture provides Smart Sensing for the NASA Sensor Web. *IEEE Xplore Digital Library,* paper #1198, Versión 1.

[20] Abed Al-Asadi, M. A, Al-Asadi, Y. A., Al-Asadi, H. A. (2012). Architectural Analysis of Multi-Agents Educational Model in Web-Learning Environments. *Journal of Emerging Trends in Computing and Information Sciences*, 3(6), 930-935. http://cisjournal.org/journalofcomputing/archive/vol3no6/vol3no6_13.pdf

[21] Zúñiga Gallegos, F., (2007). Suitable Behaviors in Dynamic Virtual Environments (Tesis de doctorado). Centro de Investigación y de Estudios Avanzados del I.P.N., Unidad Guadalajara.

[22] Clements, P. C. (1996, March). A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design* (p. 16). IEEE Computer Society.

[23] Maier, M. W., Emery, D., & Hilliard, R. (2001). Software architecture: Introducing IEEE standard 1471. *Computer*, *34*(4), 107-109.

[24] Bass, L., Clements, P., & Kazman, R. Software Architecture in Practice. 1998.

[25] Gómez Gómez, C.H. (2010). Togaf y Zachman framework. Universidad de Caldas, Manizales, Colombia.

[26] Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[27] Albin, S. T. (2003). *The art of software architecture: design methods and techniques* (Vol. 9). John Wiley & Sons.

[28] Eguíluz Pérez, J. (2009). CSS Avanzado. *Editorial www.librosweb.es*.

[29] Groussard, T. (2012). *JAVA 7: Los fundamentos del lenguaje Java*. Ediciones ENI.

[30] Chappell, D. (2011, December). *What is an application platform?* San Francisco, California.

[31] Bellifemine, F., Caire, G., Poggi, A., Rimassa, G. (2003). *JADE a White Paper.* Torino, Italia.

[32] AOS Group. *JACK An Agent Infrastructure for Providing the Decision-Making Capability Required for Autonomous System.* [Fecha de consulta: 15 Julio 2013] Disponible en: http://www.aosgrp.com/downloads/JACK_WhitePaper_UKAUS.pdf

[33] MadKit [Fecha de consulta: 23 Julio 2013] Disponible en: http://www.madkit.org/

[34] Luke, S., Cioffi-Revilla, C., Panait, L., & Sullivan, K. (2004, May). Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop* (Vol. 8).

[35] **StarLogo** [Fecha de consulta: 5 Agosto 2013] Recuperado de: **http://education.mit.edu/starlogo/**

[36] StarUML [Fecha de consulta: 15 Agosto 2013] Recuperado de: http://staruml.sourceforge.net/en/

[37] *Teoría de la decisión.* Bogotá: Universidad Nacional de Colombia. Recuperado de http://www.virtual.unal.edu.co/cursos/ingenieria/2001394/docs_curso/capitulo1/leccion1.3.htm