View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositori Institucional de la Universitat Jaume I

Máster Universitario en Sistemas Inteligentes

Trabajo Final de Máster

Arquitecturas avanzadas distribuidas escalables e inteligentes

Autor:
Víctor Pérez Miralles

Tutor académico:
Luis Amable García Fernández

Fecha de lectura: 14 de Noviembre de 2016
Curso académico 2015/2016

Resumen

Este documento contiene la memoria del trabajo de fin de máster del Máster Universitario en Sistemas Inteligentes del curso 2015/2016. El proyecto tiene un ámbito documental para el estudio y evaluación de varios marcos de desarrollo para sistemas distribuidos basados en el paradigma de agentes.

El objetivo principal del proyecto consiste en realizar un estudio de las principales características y propiedades de las nuevas APIs (*Application Programming Interface*) en Java disponibles para el desarrollo de sistemas inteligentes distribuidos en el contexto de sistemas gestores de tráfico y transporte. Para conseguir este objetivo se plantea el análisis de dos de estas APIs: WADE y AMUSE; ambas basadas en la plataforma reconocida de forma internacional JADE.

Además del estudio de ambas tecnologías durante la prolongada fase de documentación, se han desarrollado una serie de programas de prueba tratando de explotar los puntos fuertes de cada uno y explorar de forma práctica sus diferentes características.

Como punto final se ha desarrollado un proyecto de gestión de tráfico urbano para la interacción eficiente entre vehículos en intersecciones. La sección de esta memoria destinada a resultados experimentales se ha centrado en éste último proyecto y en las simulaciones que se han llevado a cabo.

Palabras clave

Sistemas multiagente, JADE, gestión de tráfico, sistemas distribuidos inteligentes, intersecciones

Abstract

This document contains the master's thesis of the Master in Intelligent Systems for course 2015/2016.The project has research oriented goals regarding the study and assessment of distributed frameworks using the agent software paradigm.

The primary objective of this project is to evaluate the main features and properties of newly developed Java API (*Application Programming Interface*) for the development of intelligent distributed systems in the area of transportation and traffic management. To achieve this goal we propose the study of two such frameworks: WADE and AMUSE; both heavily based of the internationally established platform for multi-agent research JADE.

Besides researching on both frameworks during the lengthy review and documentation phase, a few software examples have been developed in order to better exploit their strengths in an attempt to explore their various characteristics.

As a closing chapter in this thesis, we have developed an standalone project for car trajectories management in traffic intersections using the multi-agent paradigm. The whole experimental results chapter is dedicated to this particular project and all the simulations that were created to better understand the behavior of cars navigating intersections.

Keywords

Multi-agent systems, JADE, traffic management, intelligent distributed systems, traffic intersection

Índice general

1. Introducción **13**

1.1. Contexto y motivación 14

1.2. Objetivos del proyecto 14

1.3. Entorno tecnológico 14

1.4. Planificación 15

1.5. Estructura de la memoria 16

2. Estado del arte **17**

2.1. Plataformas de desarrollo de SMA 17

2.2. Intersecciones de tráfico 18

2.3. Publicaciones relacionadas 18

3. JADE: Java Agent DEvelopment Framework **21**

3.1. Introducción a JADE 21

3.2. Arquitectura general 22

3.3. Agentes 22

3.4. Comportamientos 23

3.5. Mecanismo de comunicación 24

4. WADE: Workflows and Agents Development Environment **27**

4.1. Introducción a WADE 27
4.2. Características principales 28
4.2.1. Arquitectura general 28
4.3. Trabajando con Workflows 29
4.3.1. El meta-modelo de workflows 30
4.3.2. Wolf y Eclipse 30
4.3.3. Creación de workflows 30
4.3.4. Elementos básicos 30
4.3.5. Carga y ejecución de workflows 31
4.4. Ejemplos desarrollados 32
4.4.1. Tutorial de ensamblaje de juguetes 32
4.4.2. Caso práctico de vigilancia colaborativa 36
4.5. Plataformas que utilizan WADE 40
4.6. Conclusiones sobre WADE 40

5. AMUSE: Agent-based Multi-User Social Environment 41

5.1. Introducción a AMUSE 41
5.2. Características principales 41
5.2.1. Arquitectura general 42
5.2.2. Características de AMUSE 42
5.2.3. Agentes internos de AMUSE 43
5.3. Entorno de desarrollo 44
5.4. Ejemplos desarrollados 45
5.4.1. Tutorial M-Tris 45
5.4.2. Gestión autónoma de una intersección de tráfico 49

5.5. Conclusiones sobre AMUSE 50

6. Implementación 53

6.1. Introducción 53

6.2. Agentes 54

6.2.1. WorldAgent 54

6.2.2. IntersectionManagerAgent 54

6.2.3. RadarAgent 57

6.2.4. VehicleAgent 57

6.3. Funcionamiento del simulador 58

7. Análisis preliminar del sistema 61

7.1. Pruebas de la aplicación 61

7.1.1. Escenario 1: Vehículos con baja agresividad 62

7.1.2. Escenario 2: Vehículos con agresividad moderada 63

7.1.3. Escenario 3: Vehículos con alta agresividad 63

7.1.4. Resultados conjuntos y valoración 64

8. Conclusiones y trabajo futuro 67

8.1. Consecución de los objetivos del proyecto 67

8.2. Conclusiones personales y agradecimientos 68

8.3. Trabajo futuro y mejoras 68

A. Anexo A 73

Índice de figuras

3.1. Esquema de un sistema basado en la especificación de plataforma multiagente de la FIPA. 22

4.1. Diagrama de la arquitectura general de WADE. 28

4.2. Ejemplo sencillo de un workflow que representa el funcionamiento de una máquina de café. 32

4.3. Diagrama del protocolo FIPA-Request para la ejecución de un workflow y posterior recogida de resultados si fuera necesario. 33

4.4. Vista de la estructura de ficheros desde el explorador de proyectos de Eclipse. . . 33

4.5. Interfaz de usuario del proyecto de ensamblaje de juguetes 34

4.6. Workflow para el montaje de juguetes (`AssemblingToysWorkflow.Java`) 35

4.7. Interfaz mostrada por los agentes buscadores durante el proceso de recolección de partes. 36

4.8. Workflows que implementan el comportamiento de cada uno de los agentes. . . . 38

4.9. Prueba de ejecución de la plataforma mediante WADE y workflows. 39

4.10. Salida por consola de las etapas de carga, ejecución y evaluación de los workflows. 39

5.1. Visión general de la arquitectura de la plataforma AMUSE. 42

5.2. Pantalla principal del cliente de juego de la aplicación multijugador M-Tris. . . . 45

5.3. Diagrama representando la disposición de mesas de juego por salas. 46

5.4. Vista de la distribución del juego multijugador M-Tris desde el navegador de proyectos de Eclipse. 48

5.5. Vehículos en una intersección de tráfico típica. 49

5.6. Diagrama de la distribución de segmentos en una intersección. Los segmentos verdes son los de entrada a la intersección como simboliza la flecha blanca. Los segmentos azul claro son de salida, marcados con una flecha roja. En ambos casos las flechas denotan el sentido del tráfico. 50

6.1. Simulador con la intersección diseñada y esperando vehículos. Las líneas negras representan carriles y los cuadrados oscuros representan celdas transitables. . . . 55

6.2. Simulador en funcionamiento con un tiempo de generación de vehículos de 1500ms (1.5s). Como se puede ver el sistema es perfectamente capaz de tolerarlo. 59

6.3. Simulador en funcionamiento con un tiempo de generación de solo 500ms (0.5s), saturando por completo la intersección y generando colas en casi todos los carriles. 59

7.1. Gráfica de saturación con vehículos poco agresivos. Se observan diferencias marginales en el retraso inducido hasta generar aproximadamente un vehículo por segundo. 62

7.2. Gráfica de saturación con vehículos moderados. Sin grandes cambios sobre la utilización de una baja agresividad. 63

7.3. Gráfica de saturación con vehículos muy agresivos. Como se puede observar este tipo de conducta agresiva hace que la intersección se sature mucho antes que en los casos anteriores. 64

7.4. Gráfica con todos los escenarios de simulación superpuestos para apreciar mejor las diferencias. 64

Índice de cuadros

1.1. Desglose de tareas y su contribución temporal 16

A.1. Tabla de datos para el experimento con vehículos poco agresivos 73

A.2. Tabla de datos para el experimento con vehículos de agresividad moderada . . . 74

A.3. Tabla de datos para el experimento con vehículos de agresividad muy elevada . . 75

Capítulo 1

Introducción

Contents

1.1. Contexto y motivación 14
1.2. Objetivos del proyecto 14
1.3. Entorno tecnológico 14
1.4. Planificación 15
1.5. Estructura de la memoria 16

Cada día se deben administrar sistemas más complejos para la toma de decisiones, simulación de escenarios realistas o el modelado de estructuras sociales. La distribución de carga siempre ha sido uno de los grandes problemas de la computación de altas prestaciones, y uno de los paradigmas más prometedores se basa en el diseño de agentes que toman sus propias decisiones para la consecución de objetivos tanto locales como globales.

Desde telecomunicaciones, pasando por la simulación de interacción entre robots para su funcionamiento cooperativo, hasta la gestión de tráfico la filosofía de desarrollo basada en agentes ha demostrado su validez para abordar este tipo de problemas complejos. De ella se derivan distintas arquitecturas como la llamada red de contratos o la administración de recursos mediante subastas.

Además de como plataforma para el desarrollo de software también se postula como un candidato excelente para la investigación en el campo de la computación distribuida y la inteligencia artificial.

Una de las plataformas más extendidas para el desarrollo de proyectos mediante este paradigma es JADE[1], framework en lenguaje Java. JADE ofrece a los desarrolladores un conjunto de herramientas y mecanismos que facilitan enormemente la programación de soluciones basadas en agentes, además de una interfaz gráfica que muestra el estado del entorno, de sus agentes, y de las comunicaciones entre los mismos.

El principal foco de este proyecto es estudiar las características de dos marcos de trabajo

[1] http://jade.tilab.com/

relativamente nuevos, ambos descendientes directos de JADE, para su posible aplicación tanto a la investigación como a la mejora de proyectos existentes.

1.1. Contexto y motivación

La línea de investigación se relaciona fuertemente con varias asignaturas impartidas durante el periodo lectivo del máster en sistemas inteligentes, principalmente con las asignaturas Sistemas Multiagente (*SIU024*) y Robots móviles autónomos (*SIU016*).

La principal motivación tras el proyecto es realizar un análisis exhaustivo del estado actual de las nuevas plataformas para sistemas multiagente, estudiar sus principales características, poner a prueba sus potenciales nuevas aportaciones al campo de estudio y evaluar su viabilidad como herramientas de desarrollo e investigación. Éstas nuevas plataformas son WADE[2] y AMUSE[3], ambas diseñadas como ampliación y mejora de JADE.

1.2. Objetivos del proyecto

Los principales objetivos del trabajo de fin de máster son:

1. Analizar las características de la API WADE y adaptar a WADE el caso práctico de vigilancia colaborativa hecho originalmente en JADE.
2. Analizar las características de la API AMUSE, basada a su vez en WADE.
3. Estudiar la viabilidad de WADE y AMUSE para la resolución de conflictos de tráfico en intersecciones.
4. Implementar una pequeña aplicación que permita solucionar, total o parcialmente, el problema de las intersecciones de tráfico en vías urbanas.
5. Desarrollar la documentación necesaria para la comprensión de dicha aplicación.

1.3. Entorno tecnológico

Tanto el proceso de experimentación como los tutoriales y el proyecto final se han desarrollado en el entorno de Eclipse instalado en las máquinas del laboratorio bajo Windows 7 (64 bits). Además de ofrecer un entorno de desarrollo integrado, herramientas de prueba y un sinfín de plugins ha sido necesario utilizar Eclipse debido a la estrecha relación de las tecnologías estudiadas con este IDE.

[2] `http://jade.tilab.com/wadeproject/`
[3] `http://jade.tilab.com/amuseproject/`

El código se ha desarrollado principalmente en Java aunque se ha utilizado XML en aquellas partes que lo requerían. Java es un lenguaje de programación orientado a objetos que se ejecuta en máquinas virtuales lo que permite que sea totalmente independiente de la arquitectura sobre la que se esté ejecutando. XML (*eXtensible Markup Language*) es un lenguaje de marcado utilizado para el intercambio de información de forma legible, en el caso concreto de este proyecto para definir ficheros de configuración estandarizados.

La plataforma AMUSE, estudiada en la segunda mitad del proyecto, permite el almacenamiento de información en una base de datos de forma que el estado del programa pueda guardarse entre varias ejecuciones o en caso de fallo (persistencia). Por recomendación de sus creadores según se establece en la documentación oficial se ha utilizado la herramienta Hibernate[4] como gestor de base de datos para persistencia. *Hibernate* realiza un mapeado objeto-relacional para almacenar datos de un programa en Java orientado a objetos a una base de datos relacional corriente que emplea SQL.

1.4. Planificación

El desglose de tareas y su contribución al tiempo total dedicado al proyecto se puede ver en la tabla 1.1.

Tarea	**Horas**	**Objetivo**
Estudio y primera instalación de la API WADE	25 horas	Familiarizarse con la API WADE, basada en JADE
Desarrollo de casos de ejemplo WADE	-	-
- Tutorial “Booktrading”	15 horas	Desarrollar y comprender el tutorial “booktrading”
- Tutorial “Toys Assembler”	15 horas	Desarrollar y comprender el tutorial “toys assembler”
Adaptación del caso de seguimiento colaborativo utilizando WADE	30 horas	Replicar el funcionamiento del proyecto de vigilancia colaborativa de JADE utilizando WADE
Redacción de memoria sobre WADE	15 horas	Documentar el trabajo realizado y las características principales de WADE
Instalación y análisis de la API AMUSE	20 horas	Familiarizarse con la API AMUSE, basada en WADE
Desarrollo y estudio del tutorial “mTris”	20 horas	Completar el tutorial “mTris” y comprender su funcionamiento
Redacción de memoria sobre AMUSE	15 horas	Documentar el trabajo realizado y las características principales de AMUSE
Análisis de la viabilidad de AMUSE como plataforma de desarrollo para el problema de las intersecciones de tráfico	5 horas	Tomar una decisión sobre el estado de la API AMUSE como plataforma de investigación como contrapartida a JADE

[4] http://hibernate.org/

Proyecto intersecciones de tráfico	-	-
- Planificación y diseño	5 horas	Establecer el diseño básico a seguir y planificar su desarrollo
- Implementación	70 horas	Desarrollar del proyecto
- Documentación	5 horas	Documentar el funcionamiento
Redacción de la memoria del TFM	60 horas	Redactar la memoria del trabajo de fin de máster
TOTAL	300 horas	

Cuadro 1.1: Desglose de tareas y su contribución temporal

1.5. Estructura de la memoria

Esta memoria está dividida en 8 capítulos: *Introducción*, *Estado del arte*, *JADE*, *WADE*, *AMUSE*, *Implementación*, *Análisis preliminar del sistema* y, por último, *Conclusiones y trabajo futuro*.

Como es costumbre en la *Introducción* se trata de dar una primera impresión rápida de lo que trata el proyecto, cuales son sus objetivos y en qué entorno se debe desarrollar.

En el segundo capítulo, *Estado del arte*, se concreta mucho más sobre la naturaleza del proyecto y las tecnologías que que han estudiado entre otras cosas. Además, se menciona la aportación de algunos artículos y su relevancia dentro del trabajo de fin de máster.

Los siguientes tres capítulos se centran en el trabajo de desarrollo y análisis como tal, donde se estudian tres de las plataformas más importantes en la actualidad para el desarrollo de sistemas multiagente. Cada capítulo lleva el nombre de la plataforma que estudia y está dividido en lo que se han considerado las partes más importantes del estudio, incluyendo una exhaustiva descripción de cada uno y algunos ejemplos.

El sexto capítulo titulado *Implementación* contiene los detalles de la implementación realizada, del proceso seguido y del código desarrollado.

Tras terminar de documentar la implementación se pasa al capítulo destinado al *Análisis preliminar del sistema*, donde se exponen los resultados obtenidos en las diferentes simulaciones realizadas y un breve análisis sobre los mismos.

Por último, el octavo capítulo titulado *Conclusiones y trabajo futuro* contiene exactamente lo que su nombre indica. Documenta un corolario con las reflexiones y conclusiones alcanzadas tras la finalización del proyecto, así como un breve análisis de las líneas de trabajo que se podrían explorar en el futuro.

Capítulo 2

Estado del arte

Contents

2.1. Plataformas de desarrollo de SMA 17

2.2. Intersecciones de tráfico 18

2.3. Publicaciones relacionadas 18

En éste capítulo se introducirá el problema que se desea resolver con alguna de las plataformas investigadas, el problema de la gestión de intersecciones de tráfico. Además, se analizará el estado actual de las investigaciones estudiadas en diversas publicaciones.

2.1. Plataformas de desarrollo de SMA

A pesar de la novedad relativa de los sistemas de resolución de problemas basados en múltiples agentes, existen ya diversas plataformas que ofrecen herramientas para desarrollar aplicaciones reales mediante este paradigma.

Un caso concreto es el de Magentix 2[1], un framework de código abierto para la resolución de problemas reales (negocios, ingeniería, telecomunicaciones, etcétera) desarrollado por el departamento de sistemas informáticos y computación de la *Universitat Politècnica de València*. Sin embargo, el uso de frameworks desarrollados en grupos de investigación se limita, en la mayoría de los casos, al de la propia universidad de origen.

Por otra parte, también existen plataformas ampliamente reconocidas como JADE. Se trata de una plataforma de desarrollo de sistemas multiagente escrita completamente en Java, extensible, bien documentada, y utilizada por todo el mundo. Se ahondará más en la naturaleza y origen de JADE en su correspondiente capítulo dentro de esta memoria.

[1] http://gti-ia.upv.es/sma/tools/magentix2/index.php

2.2. Intersecciones de tráfico

Puesto que el principal foco del proyecto es realizar un estudio sobre dos nuevas plataformas para desarrollar sistemas multiagente se decidió dedicar las ultimas semanas en desarrollar un pequeño sistema de simulación de intersecciones de tráfico.

Las intersecciones de tráfico son aquellos puntos en los que se cruzan varias carreteras provenientes de distintas direcciones. La investigación en algoritmos y simulaciones que podrían optimizar el comportamiento de los vehículos permitiría no solo agilizar el tráfico, sino reducir el número de vidas que se pierden todos los años en accidentes de este tipo.

Atacar un problema tan complejo mediante el paradigma multiagente podría ser interesante, ya que el problema en sí consta de un escenario en el que varios vehículos (agentes) compiten o cooperan por el espacio que necesitan ocupar (recursos).

Tras la necesaria etapa de documentación sobre las nuevas tecnologías, los últimos capítulos de esta memoria se dedicarán a describir la simulación desarrollada, sus características y los resultados obtenidos con el objetivo de modelar una intersección sin semáforos que promueva la colaboración entre los vehículos para evitar sobre todo colisiones y en menor medida atascos.

2.3. Publicaciones relacionadas

Durante el desarrollo del proyecto se consultaron algunos artículos que ya atacaban, si bien no por medio del paradigma multiagente, el problema de las intersecciones de tráfico:

- En el artículo *Hybrid Petri net model of a traffic intersection in an urban network* [17] de hace una descripción del problema bastante exhaustiva y se propone una solución para la optimización del flujo de tráfico basada en un modelo híbrido para modelar su comportamiento.
- En el artículo *A Study of Single Intersection Traffic Signal Control Based on Two-player Cooperation Game Model* [11] se plantea el problema de las intersecciones de tráfico como si fueran un juego pero destina sus esfuerzos a controlar los semáforos para optimizar el flujo y evitar colisiones comunicando los vehículos uno a uno.

Además, también se pudo encontrar diversa bibliografía que atacaba el mismo problema directamente mediante el paradigma multiagente:

- El artículo *Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism* [6] se tiene mucha relevancia en la resolución del problema en cuestión. Establece las reglas básicas que se deben seguir, la casuística de las interacciones entre agentes, y describe en profundidad el sistema de reservas necesario.
- Profundizando en el artículo anterior y compartiendo autores principales, el artículo *Autonomous Intersection Management: Multi-Intersection Optimization* [10] expande el ámbito

del problema incluyendo como requisito la sincronización entre varias intersecciones gobernadas por el mismo sistema de reservas que describieron en su artículo del año 2004. Si bien la ampliación del problema no resulta relevante para este proyecto, también incluyen algunas mejoras y correcciones sobre el artículo original de modo que se consideró oportuno estudiar este artículo también.

- El artículo *Autonomous Intersection Management for Semi-Autonomous Vehicles* [1] introduce un nuevo protocolo para la gestión de intersecciones de tráfico con vehículos semi-autónomos. La principal motivación es que la transición hacia vehículos totalmente autónomos debe pasar necesariamente por una fase en la que el componente humano retiene algo de control. El desglose de las fases que intervienen en las interacciones entre los vehículos que hacen los autores es muy interesante.

- Por último, en el artículo publicado por el *MIT* titulado *Revisiting street intersections using slot-based systems*, los autores argumentan la necesidad de eliminar por completo el sistema tradicional de semáforos dando motivación más que suficiente a este proyecto. Al mismo tiempo, revisan el estado de resolución del problema y su aplicabilidad incluyendo reflexiones sobre algunos de los artículos citados anteriormente. Aporta una nueva forma de enfocar la resolución del problema aplicando las teorías clásicas de colas que resulta ser muy interesante.

Los artículos se han presentado en orden de lectura, independientemente de su fecha de publicación.

Capítulo 3

JADE: Java Agent DEvelopment Framework

Se ha decidido añadir este capítulo a modo de resumen para que el lector pueda comprender mejor el estudio realizado tanto sobre WADE como sobre AMUSE. Como consecuencia este capítulo es más corto que los siguientes y solo aborda el funcionamiento de JADE de forma general.

Contents

3.1. Introducción a JADE 21
3.2. Arquitectura general 22
3.3. Agentes 22
3.4. Comportamientos 23
3.5. Mecanismo de comunicación 24

3.1. Introducción a JADE

JADE [14] fue desarrollado en el *Telecom Italia Lab*, un laboratorio de investigación y desarrollo que forma parte del grupo Telecom Italia[1]. La primera versión de uso publico data de febrero del año 2000, y todavía sigue en pleno desarrollo.

Se trata de una plataforma de software que facilita el desarrollo de sistemas multiagente bajo los estándares de FIPA, proporcionando un entorno en el que los agentes pueden ser creados y ejecutados con facilidad. Además, proporciona mecanismos de comunicación, sincronización, una interfaz gráfica y un paquete de librerías con un largo catálogo de funcionalidades ya definidas.

FIPA (*Foundation for Intelligent Physical Agents*) es un organismo para el desarrollo y establecimiento de estándares de software utilizados en las aplicaciones multiagente. Los protocolos

[1] `http://www.telecomitalia.com/`

de interacción descritos por FIPA se consideran como estándar de facto en la comunicación entre agentes.

3.2. Arquitectura general

JADE es una plataforma de agentes distribuida, lo que significa que puede ejecutar distintas partes del mismo sistema en hosts diferentes. Solo una máquina Java se ejecuta en cada host y los agentes que residen en ella se organizan por contenedores.

Cada plataforma debe tener por lo menos un contenedor principal en el que se ejecutan dos agentes especiales:

- **Agent Management System** (AMS): Es el sistema de gestión de agentes que supervisa la plataforma. Es el punto de contacto para todos los agentes que necesitan interaccionar para acceder al DF y gestionar correctamente su ciclo de vida.
- **Directory Facilitator** (DF): Es el servicio de páginas amarillas donde los agentes se pueden registrar para ofrecer servicios o buscar entre los agentes que proveen cierto servicio.

Se puede ver un diagrama organizativo según la especificación de FIPA en la figura 3.1.

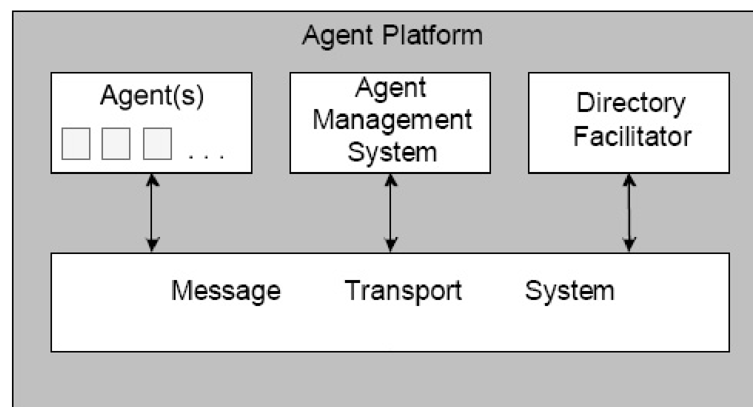

Figura 3.1: Esquema de un sistema basado en la especificación de plataforma multiagente de la FIPA.

3.3. Agentes

Los agentes en JADE deben crearse extendiendo la super clase `jade.core.Agent` y, por lo tanto, heredando comportamientos y funcionalidades predefinidas por la clase base que ofrece JADE. Normalmente cada agente implementa uno o varios servicios por medio de una serie de comportamientos; se definirá lo que es un comportamiento en el siguiente apartado.

La clase `Agent` proporciona métodos para realizar las tareas básicas de cualquier agente, es decir:

- El intercambio de mensajes mediante objetos de tipo `ACLMessage`.
- Gestión del ciclo de vida del propio agente: Cuándo activarse, entrar en suspensión o finalizar su ejecución definitivamente.
- Planear y ejecutar las tareas necesarias, incluso de forma alternada, para cumplir con los servicios que ofrece.

El ciclo de vida de todo agente es el propuesto originalmente por FIPA. Estos agentes pasan por diferentes estados definidos como:

- **INITIATED**: El agente ha sido creado pero no registrado en el AMS.
- **ACTIVE**: El agente se ha registrado y se le proporciona un nombre. En este estado ya puede comunicarse con otros agentes.
- **SUSPENDED**: El hilo en el que se ejecuta el agente está suspendido.
- **WAITING**: El agente está bloqueado esperando a cierto evento.
- **DELETED**: El agente ha finalizado su ejecución y su hilo ha desaparecido, ya no existe en el AMS.
- **TRANSIT**: El agente se encuentra en proceso de intercambio entre una plataforma y otra. Los mensajes que reciba en este estado se guardarán en un *buffer* y le serán remitidos cuando finalice el proceso de migración.

En general, podemos interpretar a un agente como la entidad que proporciona servicios que ofrece públicamente gracias al DF (páginas amarillas). Del mismo modo, cualquier agente puede realizar una búsqueda en el DF y utilizar los servicios que ofrecen el resto.

3.4. Comportamientos

Los comportamientos definen las tareas que ejecutan los agentes. Los agentes pueden añadirse comportamientos a sí mismos gracias al método `addBehaviour()` proporcionada por la clase base.

Aunque existen varios tipos de comportamientos todos deben implementar, como mínimo, dos métodos:

- `action()`: Se ejecuta cuando un evento activa al comportamiento.

- `done()`: se ejecuta en cada paso para comprobar si el comportamiento ha completado su tarea.

Entre otros, existen comportamientos tan variados como:

- `Behaviour`: Es la clase base, pero permite hacer cosas muy potentes como por ejemplo definir comportamientos en varios pasos guardando en una variable el último paso dado.
- `WakerBehaviour`: Ejecuta cierto bloque de código una sola vez en un tiempo especificado.
- `TickerBehaviour`: Ejecuta un bloque de código de forma periódica cada cierto tiempo establecido por programación.

A continuación se puede ver un pequeño fragmento de código correspondiente a un comportamiento de tipo `TickerBehaviour` y como se añade al agente que lo define.

```
Behaviour loop = new TickerBehaviour( this, 300 )
   {
      protected void onTick() {
         System.out.println("Looper:" + myAgent.getLocalName());
      }
   });

addBehaviour( loop );
```

Listing 3.1: Comportamiento sencillo de tipo TickerBehaviour

Lo primero que hace el código es crear un comportamiento de tipo `TickerBehaviour` pasándole como primer parámetro el agente que lo crea, y como segundo parámetro el tiempo en milisegundos entre cada activación (o *tick* en inglés).

Dentro de la definición implementamos el método `onTick` que se ejecutará cada vez que se despierte el comportamiento, es decir, cada 300 milisegundos. A modo de ejemplo, solo imprimimos una cadena de texto por la salida estándar con el nombre del agente que ejecuta el comportamiento.

Para terminar, en la última línea del código añadimos el comportamiento ya perfectamente definido al agente gracias al método `addBehaviour`.

3.5. Mecanismo de comunicación

Debido a que los desarrolladores de JADE han puesto mucho énfasis en facilitar el mecanismo de comunicación y su uso es bastante natural.

La base de toda comunicación entre agentes es ACL (*Agent Communication Language*). Los agentes pueden enviar mensajes mediante su método `send()` y recibirlos mediante `receive()`,

dichos mensajes extienden la clase `ACLMessage` y tanto el contenido del mensaje como el valor de los distintos campos que ofrece el lenguae los definen los propios agentes.

Una de las mecánicas a tener en cuenta es que la recepción de mensajes puede ser bloqueante si se usa el método `blockingReceive()`, lo que permite facilitar la sincronización cuando sea necesario pero debe usarse con cuidado ya que bloquea por completo al agente hasta que sucede el evento esperado.

Por supuesto, se podrían decir muchas más cosas sobre el paso de mensajes ya que JADE implementa soporte para ontologías, filtrado de mensajes por contenido o por cierto identificador, creación de plantillas tanto para su creación como recepción, y un largo etcétera. No obstante, se ha considerado que para la comprensión de éste trabajo no es necesario ahondar tanto, ya que tanto WADE como AMUSE automatizan bastante todo el proceso de comunicación entre agentes que en JADE debe programarse a mano.

Capítulo 4

WADE: Workflows and Agents Development Environment

Contents

4.1. Introducción a WADE **27**
4.2. Características principales **28**
4.2.1. Arquitectura general 28
4.3. Trabajando con Workflows **29**
4.3.1. El meta-modelo de workflows 30
4.3.2. Wolf y Eclipse 30
4.3.3. Creación de workflows 30
4.3.4. Elementos básicos 30
4.3.5. Carga y ejecución de workflows 31
4.4. Ejemplos desarrollados **32**
4.4.1. Tutorial de ensamblaje de juguetes 32
4.4.2. Caso práctico de vigilancia colaborativa 36
4.5. Plataformas que utilizan WADE **40**
4.6. Conclusiones sobre WADE **40**

4.1. Introducción a WADE

WADE es una plataforma software basada en JADE que proporciona soporte para la ejecución de tareas definidas mediante workflows. El componente principal de WADE es la clase agente `WorkflowEngineAgent` que extiende la clase `Agent` básica de JADE y proporciona métodos y herramientas para la ejecución de workflows mediante su programación en lenguaje Java.

Algunas funcionalidades extra incluyen la facilidad de configuración de las aplicaciones mediante ficheros XML y la integración completa con el plugin de eclipse WOLF que permite el

diseño de workflows utilizando una interfaz gráfica para luego poblar las actividades con código Java.

La última versión de WADE a fecha de redacción de esta memoria es la versión 3.5.0 publicada el 23 de Diciembre de 2015.

4.2. Características principales

En éste apartado se definen las características básicas de la plataforma WADE.

4.2.1. Arquitectura general

Plataforma

La plataforma WADE se distribuye en uno o varios sistemas huésped o "*host*", cada uno con uno o más contenedores además del contenedor principal (*Main Container*) que contiene el AMS (*Agent Management System*) y el DF (*Directory Facilitator*). El contenedor principal de cada host debe ser el primero en activarse para poder dar servicio a todos los demás.

Los agentes específicos de cada aplicación se distribuyen en los contenedores y al mismo tiempo se ejecutan componentes propios de WADE. Estos componentes son el proceso *Boot Daemon* (uno por host) que se encarga de activar los contenedores de cada host y el CFA (*Configuration Agent*) que se ejecuta en el contenedor principal y se encarga de interactuar con los Boot Daemon y controlar el ciclo de vida de la aplicación.

El diseño de la arquitectura se puede ver gráficamente en la 4.1, junto con algunos agentes específicos de WADE que se introducirán en el siguiente apartado.

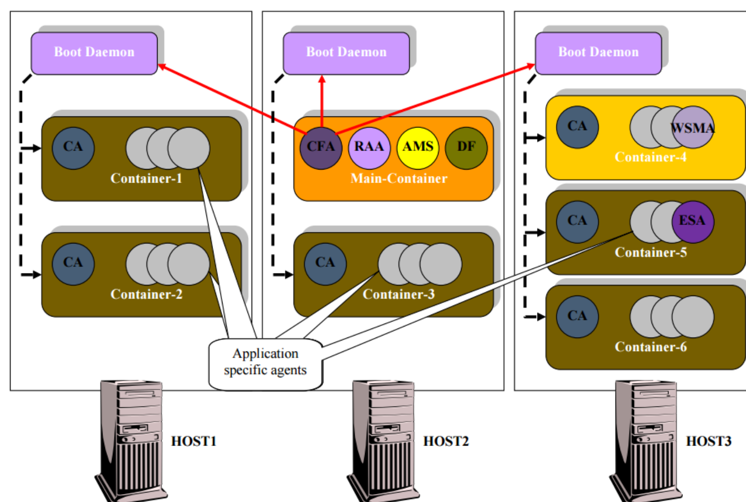

Figura 4.1: Diagrama de la arquitectura general de WADE.

Agentes WADE

Además de los componentes antes mencionados, WADE tiene algunos agentes específicos de la plataforma. Estos agentes son:

- **Runtime Allocation Agent** (RAA): Se activa en el contenedor principal. Se encarga de manejar las colecciones de agentes (*Agent Pools*) y de gestionar fallos en la ejecución específicos de los agentes.
- **Workflow Status Manager Agent** (WSMA): Se instancia en cualquier contenedor que necesite ejecutar workflows y se encarga de supervisar su ejecución.
- **Event System Agent** (ESA): Se puede ejecutar en cualquier contenedor y se encarga de manejar el sistema de eventos de WADE para que los workflows puedan suspenderse en espera de cierto evento concreto.

Una peculiaridad de WADE es que los agentes deben extender la clase `WadeAgentImpl` en lugar de `jade.core.Agent`. Esto permite registrar a los agentes automáticamente en el DF entre otras cosas que WADE hace por ti, como por ejemplo reiniciar agentes caídos o bloqueados automáticamente.

Tipos y roles

Para manejar los agentes WADE se guarda en un fichero de configuración (`wade/cfg/types.xml`) todas las clases que definen un tipo de agente. Cada tipo tiene un nombre, una clase asociada y un conjunto de propiedades que se aplican a todos los agentes de ese tipo. Por ejemplo si definimos un agente Sensor, deberíamos añadir al fichero `types.xml` una línea como la que se muestra a continuación:

```
<agentType description="Agente Sensor" className="monitoring.AgSensor" role="Workflow
    Executor" />
```

Listing 4.1: Registrar un agente en WADE

Donde el atributo `description` es el nombre que usaremos más adelante para instanciar el agente, `className` es la clase concreta que lo define y `role` es el rol del agente que le otorga ciertas características (en nuestro caso, es un agente que ejecuta workflows).

Este tipo de fichero de configuración permite definir tipos de agente y sus roles, proporcionando si es necesario un conjunto de atributos para cada uno. Por ejemplo, parámetros de entrada, atributos estáticos con cierta configuración, parámetros para el RAA, etcétera.

4.3. Trabajando con Workflows

En este apartado se define en más profundidad el concepto de workflow.

4.3.1. El meta-modelo de workflows

WADE no utiliza ningún estándar de definición de workflows formal. Esto se debe a que para explotar toda la funcionalidad de eclipse y WOLF los workflows se definen en Java siguiendo una serie de formalismos propios de WADE. Aún así, para no reinventar la rueda, la implementación se basa fuertemente en el estándar XPDL del Workflow Management Consortium[1].

Un workflow es un patrón repetible de actividades de negocio que permite automatizar cierta tarea compleja. Además, permite obtener una representación visual del ciclo de vida sin tener que ver una sola línea de código.

4.3.2. Wolf y Eclipse

WADE hace uso del plugin WOLF disponible para Eclipse. WOLF, además de ofrecer un conjunto de herramientas para la ejecución y depuración de aplicaciones WADE, permite diseñar workflows de forma visual para luego poblar cada actividad de código Java que desempeñe la tarea específica.

4.3.3. Creación de workflows

Para crear un workflow en WADE primero se debe crear un módulo que lo contenga, se puede hacer directamente con botón derecho sobre el proyecto y siguiendo los menús `WADE Tools >Add Module`. Una vez creado el módulo solo hay que hacer `New >Other >Wolf >Workflow`.

Los workflow pueden ser editados de dos formas, como código Java abriendo el fichero que lo contiene o mediante la interfaz gráfica con botón derecho sobre el workflow `WADE Tools >Open Workflow Editor`.

4.3.4. Elementos básicos

En este apartado se describen los elementos básicos que utiliza WADE para definir un workflow, sus tareas y la relación entre ellas. Además de la tarea principal y las tareas (una o varias) finales.

Actividades

Se entiende por actividad un conjunto de acciones que forman una etapa atómica del flujo de trabajo. Existen varios tipos de actividades:

- De ejecución

[1] http://www.wfmc.org/

- **Código**: Se especifica directamente mediante código Java.
- **Herramienta**: Invocan una o varias herramientas externas, generalmente identificadas como Aplicaciones.
- **Subflow**: Invocan un workflow y lo ejecutan.
- **Servicios web**: Invocan servicios web y gestionan la recogida de resultados.

- De sincronización

  - **WaitEvent**: Esta actividad detiene la ejecución del workflow hasta que el evento tiene lugar.
  - **WaitWebServices**: Detiene la ejecución del workflow hasta que recibe cierta llamada a un servicio web.
  - **SubflowJoin**: Por defecto los subflows se ejecutan de forma síncrona, es decir el workflow principal se detiene hasta que finaliza el subflow. También se pueden ejecutar de forma asíncrona, y el workflow principal continúa con su ejecución tras invocar el subflow. Cuando el workflow principal llega a un evento de tipo `SubFlowJoin` se detiene hasta que el subflow termina.

Transiciones

Las transiciones se encargan de conectar dos actividades y representan el orden de ejecución. Si una actividad tiene varias transiciones salientes se emplean condiciones para que el flujo de ejecución sepa cuál tomar. En la figura 4.2 se puede ver un ejemplo de workflow sencillo.

Herencia

WADE permite definir workflows a partir de otros. Ofrece métodos para eliminar actividades y transiciones concretas (`deregisterActivity()` y `deregisterTransition()`) de forma que puedes re-definir su funcionamiento o añadir más actividades en puntos intermedios.

Si se quiere insertar una actividad en un workflow heredado entre dos de las actividades originales solo es necesario des-registrar la transición que une a ambas, registrar la actividad nueva y luego registrar las dos nuevas transiciones que unirán la actividad de origen inicial con la nueva actividad y ésta misma con la actividad final.

4.3.5. Carga y ejecución de workflows

Para ejecutar un workflow WADE proporciona un protocolo estandarizado basado en el paso de mensajes según el estándar de FIPA. La figura 4.3 muestra el esquema de dicho protocolo.

WADE permite pasar argumentos a los workflows y también recoger resultados mediante la clase `WorkflowResultListener`.

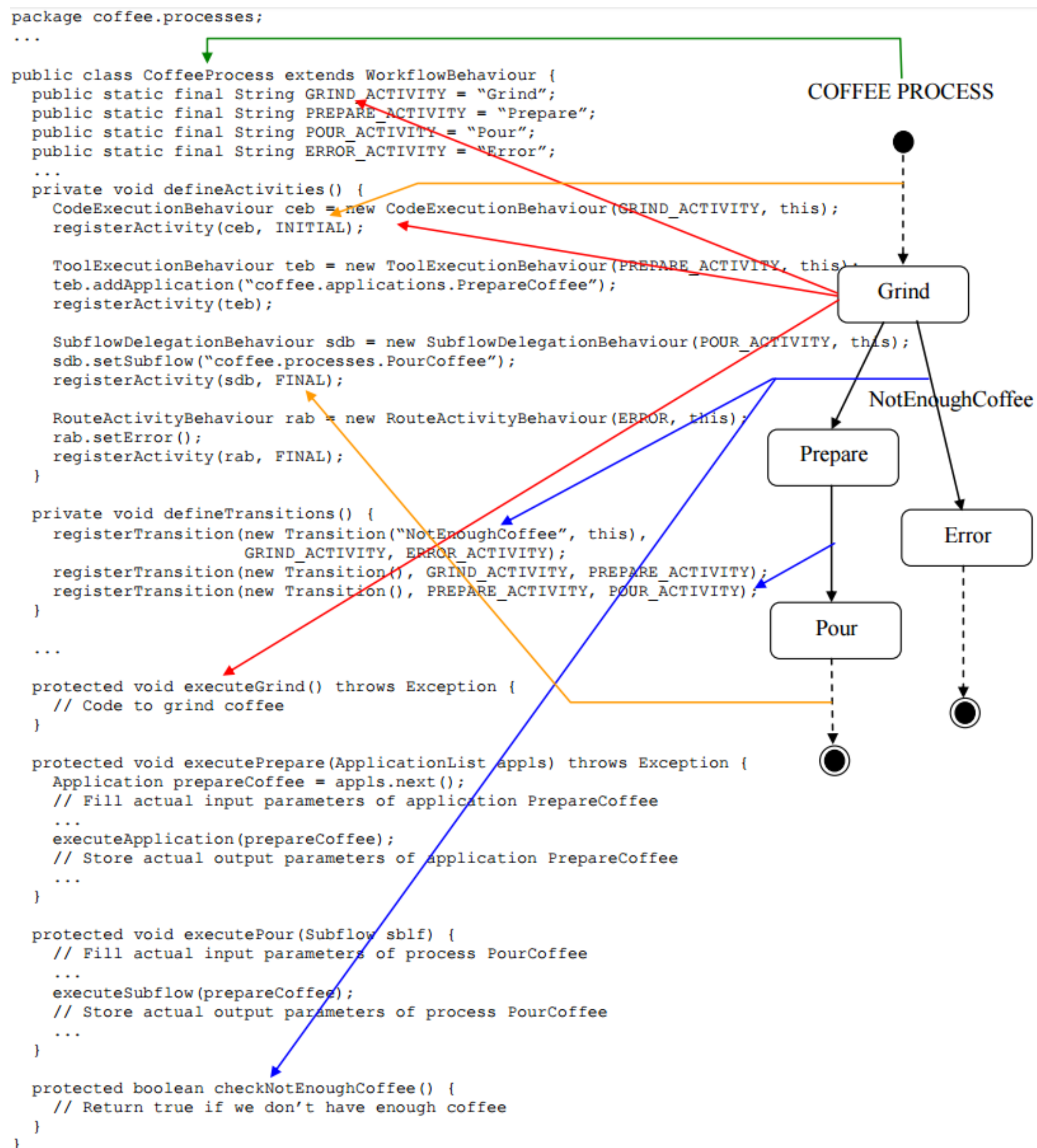

Figura 4.2: Ejemplo sencillo de un workflow que representa el funcionamiento de una máquina de café.

4.4. Ejemplos desarrollados

Esta sección contiene algunos comentarios sobre la aplicación que ofrece WADE a modo de tutorial y un caso práctico modelado con WADE a partir del proyecto para la asignatura *SIU024 Sistemas Multiagente* del Máster en Sistemas Inteligentes.

4.4.1. Tutorial de ensamblaje de juguetes

Este tutorial está disponible en la web oficial de WADE en la siguiente url:

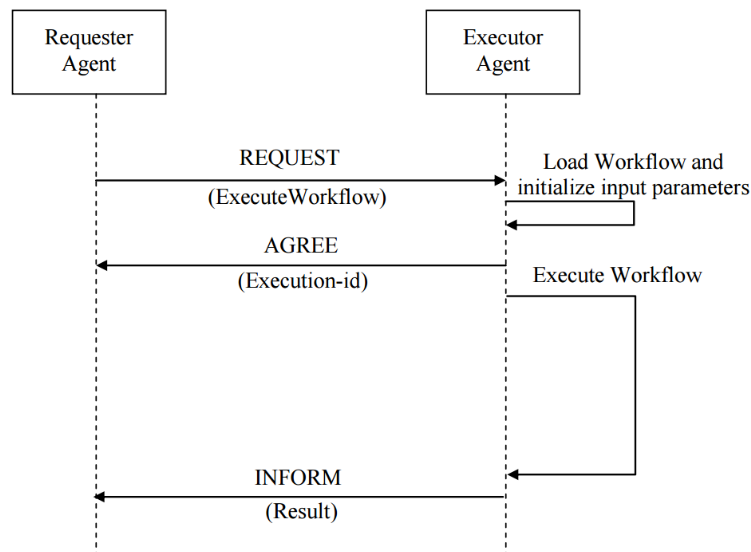

Figura 4.3: Diagrama del protocolo FIPA-Request para la ejecución de un workflow y posterior recogida de resultados si fuera necesario.

- Guía del tutorial[2]
- Código completo[3]

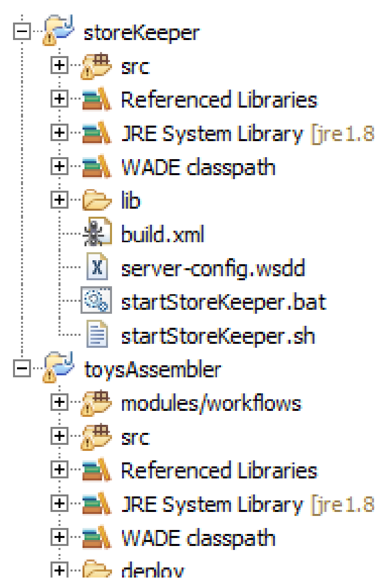

Figura 4.4: Vista de la estructura de ficheros desde el explorador de proyectos de Eclipse.

El tutorial está dividido en dos proyectos principales 4.4: Uno encargado de montar juguetes mediante un agente ensamblador y varios agentes buscadores, y otro proyecto que ofrece componentes (piezas de juguetes) a los agentes buscadores mediante un servicio web.

[2] http://jade.tilab.com/wade/doc/tutorial/WADE-Tutorial.pdf
[3] http://jade.tilab.com/dl.php?file=WADE-examples-3.5.0.zip

- `storeKeeper`: Contiene el servicio web, mantiene un almacén de piezas y las ofrece al agente ensamblador. No es un proyecto JADE y no se explica en más profundidad.
- `toysAssembler`: Contiene los agentes ensamblador y buscador que montan los juguetes.

El proyecto WADE en sí mismo es `toysAssembler`, además de las clases necesarias para mantener un listado de componentes define dos agentes.

Agente ensamblador

El agente ensamblador muestra la interfaz gráfica principal de la aplicación (*AssemblerAgentGui.Java*) 4.5 y define el propio agente (*AssemblerAgent.Java*).

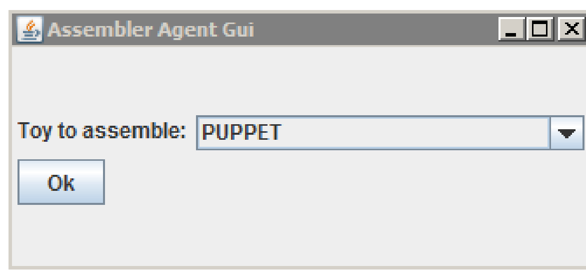

Figura 4.5: Interfaz de usuario del proyecto de ensamblaje de juguetes

Como en todo proyecto WADE la clase `AssemblerAgent` extiende la clase `WorkflowEngineAgent` que le permite ejecutar workflows. Inicializa y muestra la interfaz tras lo que añade un comportamiento de tipo `SubscriptionInitiator` para mantener una lista actualizada de todos los agentes buscadores que residen en el sistema.

El único método complejo que define el agente ensamblador se encarga de ejecutar un workflow y procesar sus resultados. Dicho workflow se encuentra en el paquete `modules/workflows` dentro del proyecto y su representación gráfica gracias al plugin WOLF se puede ver en la figura 4.6.

El workflow recibe un tipo de juguete a montar (marioneta o vagón de tren) y busca los componentes necesarios mediante agentes buscadores para luego montarlo.

El workflow se divide en las siguientes actividades:

- **Identify Toy Components**: Le pasa el tipo de juguete al catálogo y obtiene una lista de componentes necesarios.
- **Select Next Component-Set To Search**: Selecciona el siguiente componente a buscar de la lista obtenida en la actividad anterior. Esta actividad y todas las siguientes excepto la final se ejecuta una vez por cada tipo de componente (se puede ver claramente en la representación gráfica del workflow mediante una flecha de realimentación).

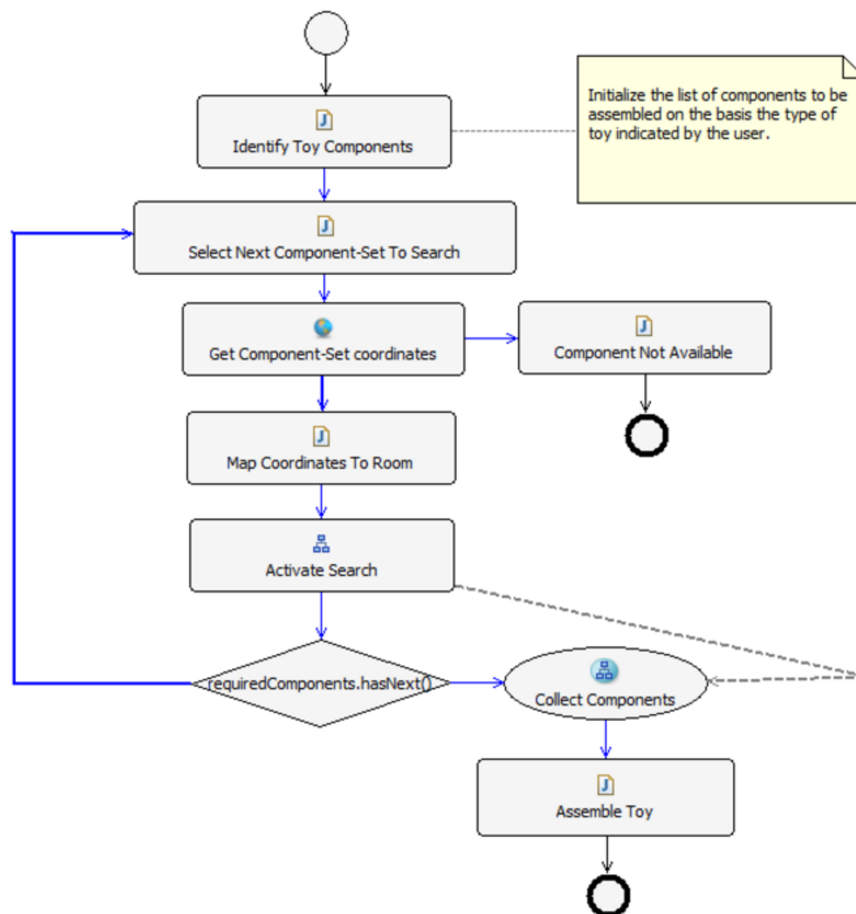

Figura 4.6: Workflow para el montaje de juguetes (`AssemblingToysWorkflow.Java`)

- **Get Component-Set coordinates**: Llama al servicio web publicado por "storeKeeper" para obtener la disponibilidad y coordenadas dentro del almacén de los componentes del tipo actual. Si no hay existencias el workflow termina y el montaje falla.

- **Map Coordinates To Room**: Obtiene las coordenadas del componente.

- **Activate Search**: Ejecuta un subflow (workflow anidado) que se encarga de simular el movimiento de un robot (el agente buscador) dentro del almacén hasta encontrar el componente (alcanzar las coordenadas obtenidas en la actividad anterior).

- **Collect Components**: Se trata de una actividad de sincronización (explicada en el apartado elementos básicos de ésta memoria) de tipo `SubflowJoin` que obtiene todos los componentes concretos encontrados por los agentes buscadores y los almacena en una lista.

- **Assemble Toy**: Por último, con todos los componentes necesarios en una lista se puede ensamblar el juguete.

Agente buscador

Son agentes buscadores muy sencillos que tienen una velocidad (*speed*) y un campo de búsqueda (*viewsize*). Su única función es esperar a ser invocados por el workflow de búsqueda desde la actividad `Activate Search` del workflow principal y mostrar una interfaz gráfica simulando dicha búsqueda.

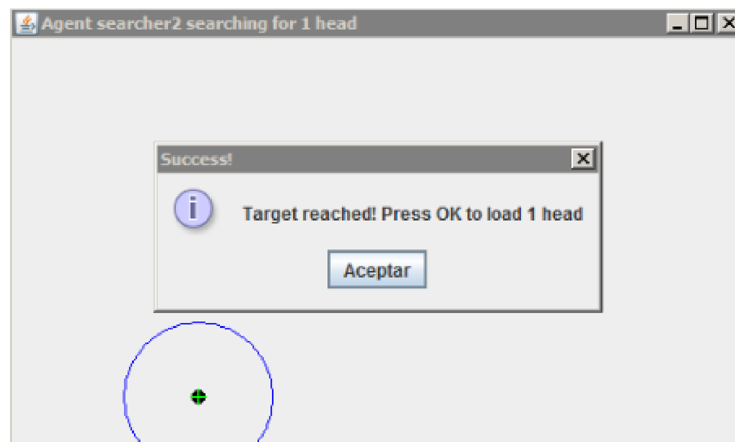

Figura 4.7: Interfaz mostrada por los agentes buscadores durante el proceso de recolección de partes.

4.4.2. Caso práctico de vigilancia colaborativa

Para profundizar más en el funcionamiento de los workflows se ha decidido adaptar el proyecto final de la asignatura SIU024 (*Sistemas Multiagente*) para que lleve a cabo su función mediante workflows.

El proyecto cuenta con un agente interfaz que muestra al usuario un mapa con el movimiento del resto de agentes. Uno o varios agentes sensores representados como barcos tratan de colaborar para seguir el movimiento de otros agentes móviles. Los agentes sensores tienen un rango limitado y deberán pedir ayuda si los móviles se alejan demasiado. El objetivo es que los agentes sensores sigan los movimientos de los agentes móviles mediante vigilancia colaborativa.

Los agentes

Los tipos de agente definidos son los siguientes (en el fichero `types.xml`):

```
<agentType description="Agente Interfaz" className="monitoring.AgInterfaz" role="Workflow
    Executor" />
<agentType description="Agente Movil" className="monitoring.AgMovil" role="Workflow
    Executor" />
<agentType description="Agente Sensor" className="monitoring.AgSensor" role="Workflow
    Executor" />
```

Listing 4.2: Tipología de agentes existente

Se han creado varias configuraciones para ejecutar cierto número de agentes sensores y móviles, cada tipo dentro de su contenedor propio. Un fichero de configuración tiene el siguiente aspecto:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<platform description="3 Moviles y 3 Sensores" name="Monitoring">
    <hosts>
        <host name="localhost">
            <containers>
                <container name="Interfaz-container">
                    <agents>
                        <agent name="interfaz" type="Agente Interfaz"/>
                    </agents>
                </container>

                <container name="Moviles-container">
                        <agents>
                        <agent name="movil1" type="Agente Movil"/>
                        <agent name="movil2" type="Agente Movil"/>
                        <agent name="movil3" type="Agente Movil"/>
                        </agents>
                </container>

                <container name="Sensores-container">
                        <agents>
                        <agent name="sensor1" type="Agente Sensor"/>
                        <agent name="sensor2" type="Agente Sensor"/>
                        <agent name="sensor3" type="Agente Sensor"/>
                        </agents>
                </container>
            </containers>
        </host>
    </hosts>
</platform>
```

Listing 4.3: Configuración de ejecución con tres agentes móviles y otros tres sensores

Este caso concreto tiene el agente interfaz, tres agentes móviles y otros tres agentes sensor.

El código del proyecto se puede descargar en la siguiente dirección url Proyecto de vigilancia colaborativa con workflows[4].

Los Workflows

Puesto que el sistema solo cuenta con tres tipos de agente se ha separado el funcionamiento de cada uno en un workflow separado.

El funcionamiento del primer workflow, correspondiente al agente interfaz, es aparentemente sencillo. Se encarga de inicializar los datos que necesita el agente, en este caso el canvas sobre el que dibujar, después pone en marcha los comportamientos que inicializan los agentes móviles

[4] https://drive.google.com/open?id=0BzxZ0r0Wy03eSV9rWUtVWG1OT00

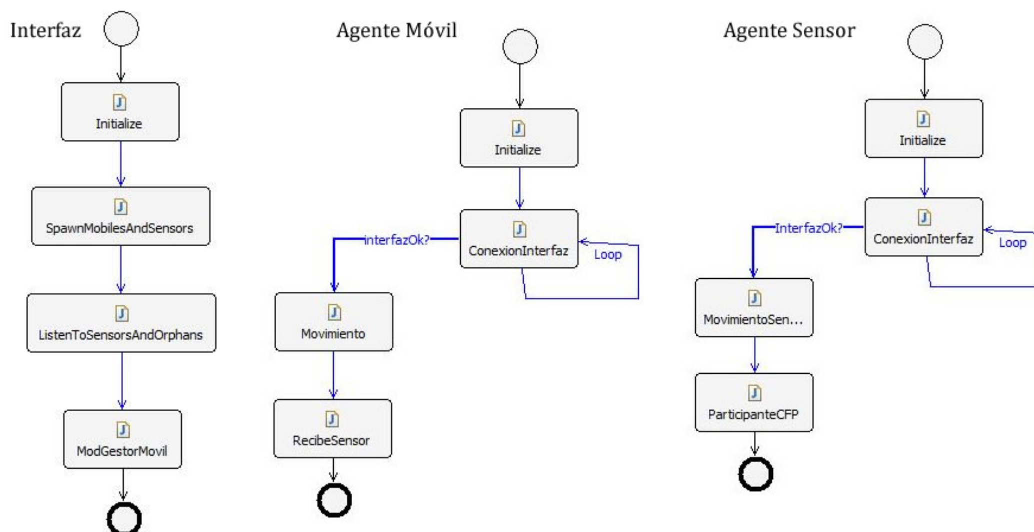

Figura 4.8: Workflows que implementan el comportamiento de cada uno de los agentes.

y sensores. Una vez todo está en marcha se mantiene en escucha a la espera de los mensajes de dichos agentes listo para recibir sus posiciones y actualizar la interfaz gráfica.

Aunque los workflows ejecutados por agentes móviles y sensores tienen una apariencia similar en el entorno de desarrollo gráfico de WADE, hacen cosas muy distintas una vez se han registrado con el agente interfaz.

Los móviles inician un comportamiento que los pone en marcha y que va variando su posición con cada paso, de forma aleatoria. Al mismo tiempo quedan a la espera de recibir un agente sensor que se encargará de transmitir su posición al agente interfaz.

Cuando un agente móvil se queda sin agente sensor, pasa a disposición del agente interfaz temporalmente hasta que, mediante un CFP (call For Proposal) se le pueda entregar a un nuevo agente sensor para su seguimiento. Este funcionamiento se implementa en los comportamientos `IniciadorCFP` y `ParticipanteCFP` que comparten tanto el agente interfaz como los agentes sensores y que ejecutan cuando es necesario.

Por otra parte los agentes sensores ejecutan mediante su workflow el comportamiento que les dota de movimiento (similar al de los agentes móviles) y un comportamiento que les pone a la escucha de cualquier petición de CFP que se pueda dar en el simulador, bien por parte de otros agentes sensores que piden ayuda tras perder a un móvil o por parte del agente interfaz.

Prueba de ejecución

Para probar el funcionamiento de la plataforma solo hay que iniciar el `Boot Daemon` seguido del `Main Container`, como en cualquier aplicación WADE 4.9. Despues se puede cargar una de las 4 configuraciones disponibles (por defecto *Monitoring*) y pulsar el botón de ejecución. Los pasos concretos han sido descritos en profundidad en el apartado de introducción a WADE de capítulos anteriores.

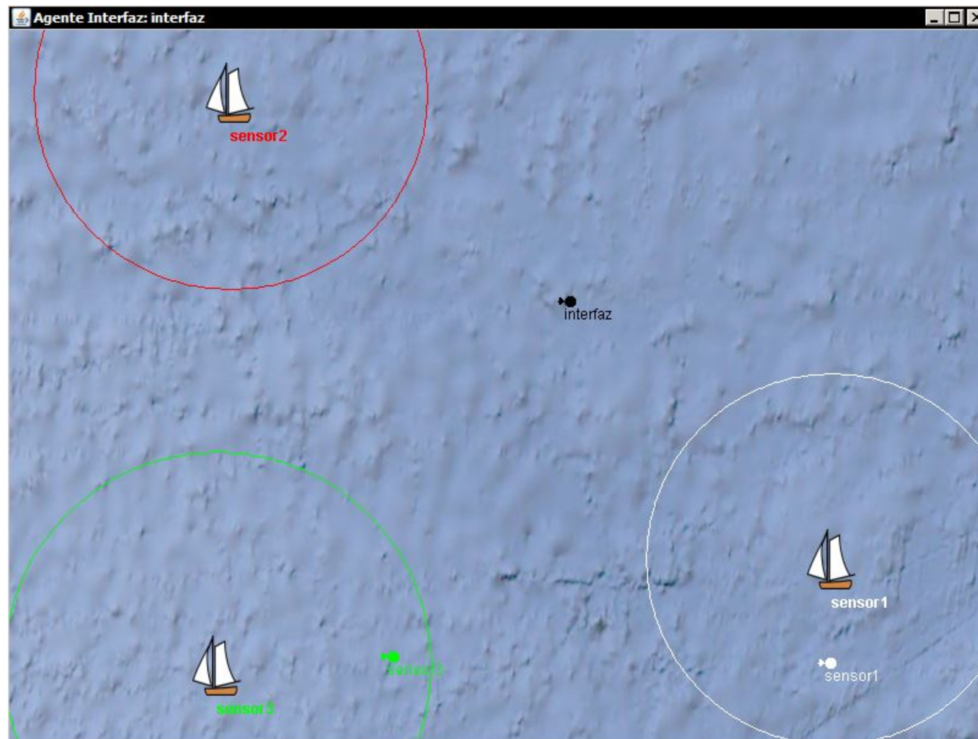

Figura 4.9: Prueba de ejecución de la plataforma mediante WADE y workflows.

Además del resultado visible mediante la interfaz gráfica, también se pueden consultar las consolas de salida donde se muestra información de la carga, ejecución y resultados de todos los workflows 4.10.

```
Workflow correctly loaded by performer movil1
Workflow execution for movil1 initialized at 1477238326041.
Workflow execution for movil3 initialized at 1477238326041.
[movil1]: Buscando agente interfaz...
[movil3]: Buscando agente interfaz...
[movil1]: Agente interfaz encontrado.
[movil3]: Agente interfaz encontrado.
2016-10-23 17:58:46.048 WARNING: Workflow status manager agents no
Workflow correctly loaded by performer movil2
2016-10-23 17:58:46.049 INFO: Agent movil2@MONITORING - Executor J
Workflow execution for movil2 initialized at 1477238326049.
[movil2]: Buscando agente interfaz...
[movil2]: Agente interfaz encontrado.
[interfaz]: Recibido nuevo movil -> movil1
[interfaz]: Recibido nuevo movil -> movil3
[interfaz]: Recibido nuevo movil -> movil2
2016-10-23 17:58:46.094 INFO: Agent interfaz@MONITORING - Executor
2016-10-23 17:58:46.095 INFO: Agent interfaz@MONITORING - Executor
Successfully executed workflow for executor interfaz
2016-10-23 17:58:46.190 WARNING: Workflow status manager agents no
Workflow correctly loaded by performer sensor1
2016-10-23 17:58:46.190 INFO: Agent sensor1@MONITORING - Executor
2016-10-23 17:58:46.191 WARNING: Workflow status manager agents no
Workflow execution for sensor1 initialized at 1477238326191.
Workflow correctly loaded by performer sensor2
2016-10-23 17:58:46.191 INFO: Agent sensor2@MONITORING - Executor
```

Figura 4.10: Salida por consola de las etapas de carga, ejecución y evaluación de los workflows.

4.5. Plataformas que utilizan WADE

No he podido encontrar ningún proyecto comercial que utilice WADE. De hecho, incluso las publicaciones en artículos son muy escasas:

- WADE: an open source platform for workflows and agents [5]
- Interactive workflows with WADE [2]
- WADE: a software platform to develop mission critical applications exploiting agents andworkflows [3]

Esto se debe tanto a su novedad como al hecho de que su uso se ha extendido más como plataforma de investigación en los campos de inteligencia artificial y computación distribuida.

4.6. Conclusiones sobre WADE

En líneas generales estoy satisfecho con el funcionamiento de WADE a pesar de que la documentación es muy escasa y los supuestos tutoriales no lo son. Solo veo potencial en aplicaciones muy grandes donde trabajar con workflows es una prioridad, bien porque trabajan muchas personas sobre el mismo proyecto o porque es conveniente que gente sin conocimientos en el ámbito de la programación JADE modele el flujo de trabajo general.

Para aplicaciones pequeñas trabajar con workflows ha demostrado, al menos en mi caso, dar más problemas que soluciones. Haces el doble de trabajo primero para modelar y luego para programar sus actividades y transiciones, por no hablar de que cosas que antes ejecutaba el propio agente ahora deben ser delegadas a la ejecución del workflow con todos los problemas que eso conlleva.

Las mejoras y novedades introducidas en WADE con respecto a JADE se centran en la ideología de trabajar con workflows. Se trata de una forma de trabajar no demasiado extendida y que por desgracia acarrea una serie de problemas derivados, principalmente, de la complejidad de la propia plataforma.

WADE se ha ideado para aplicaciones grandes o que requieran cierto nivel de abstracción para definir procesos de negocio en *Business Process Management*. Además, la falta de documentación por parte de los desarrolladores dificulta el aprovechamiento de todo el potencial que publicitan para WADE, tal vez debido a su novedad.

La portabilidad del proyecto también se vería mermada si se decidiera utilizar WADE como marco de trabajo ya que requiere, además de los ejecutables del proyecto, de una instalación completa de la plataforma y su posterior configuración para poder ponerlo en marcha.

Capítulo 5

AMUSE: Agent-based Multi-User Social Environment

Contents

5.1. Introducción a AMUSE **41**

5.2. Características principales **41**

5.2.1. Arquitectura general 42

5.2.2. Características de AMUSE 42

5.2.3. Agentes internos de AMUSE 43

5.3. Entorno de desarrollo **44**

5.4. Ejemplos desarrollados **45**

5.4.1. Tutorial M-Tris 45

5.4.2. Gestión autónoma de una intersección de tráfico 49

5.5. Conclusiones sobre AMUSE **50**

5.1. Introducción a AMUSE

AMUSE (*Agent-based Multi-User Social Environment*) es una plataforma basada en WADE que facilita el desarrollo de aplicaciones sociales distribuidas que involucran a usuarios que cooperan o compiten para alcanzar objetivos comunes o privados. El principal objetivo de AMUSE es proporcionar las herramientas necesarias para el desarrollo de juegos online multijugador basados en agentes.

5.2. Características principales

En éste apartado se definen las características básicas de la plataforma AMUSE.

5.2.1. Arquitectura general

La plataforma AMUSE se basa en WADE, cuyo foco principal es la gestión de procesos de negocio (*Business Process Management*), para el desarrollo de aplicaciones sociales y juegos multijugador. Permite tanto el desarrollo de aplicaciones cliente como de la parte del servidor, si fuera necesario, siguiendo una aproximación PaaS (*Platform as a Service*).

Como se puede ver en la figura 5.1, una aplicación completa tiene un número de clientes móviles Android y un servidor PaaS que se encarga de la conexión con la base de datos, la gestión de las salas de juego y de proporcionar un entorno a los desarrolladores mediante la consola de administración de AMUSE.

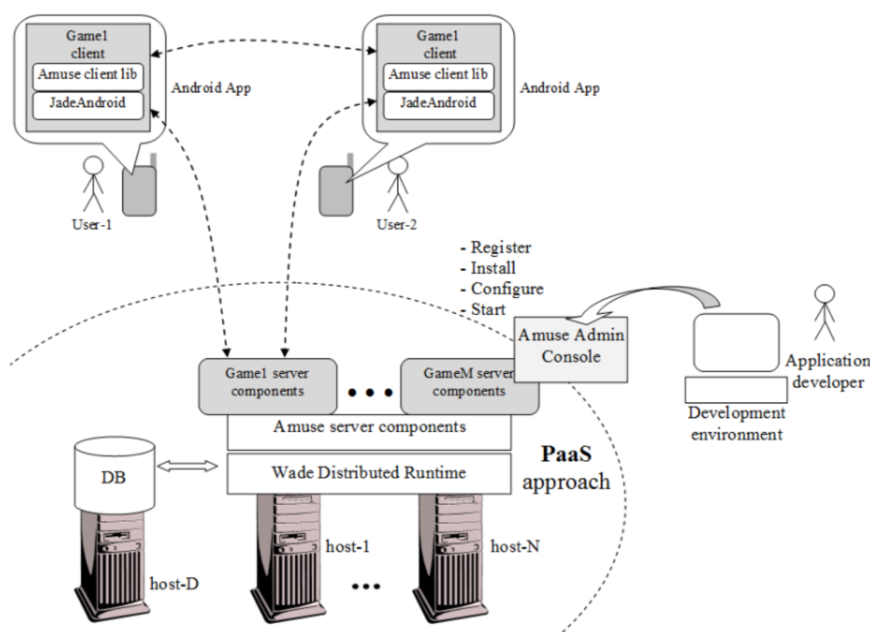

Figura 5.1: Visión general de la arquitectura de la plataforma AMUSE.

Todas las aplicaciones AMUSE cuentan, como mínimo, con algún cliente que proporciona la interfaz de usuario y cualquier lógica de la parte del cliente que sea necesaria. Los clientes hacen uso de las librerías `Amuse client library` para interactuar entre ellos y con el servidor (si hay alguno). En la versión actual solo están soportados los clientes de tipo Android.

Además, en muchos casos las aplicaciones necesitarán un servidor que ejecute ciertas partes de la lógica del juego y se encargue de tareas de coordinación y administración genéricas.

5.2.2. Características de AMUSE

AMUSE ofrece algunas características que no están estrictamente relacionadas con el desarrollo de videojuegos, las más importantes:

- **Application management**: Permite registrar, instalar, configurar, activar y administrar

aplicaciones. Se puede acceder mediante el panel de administración de AMUSE[1].

- **User management**: Permite el registro e identificación de los usuarios de la aplicación y algunas funciones básicas para la gestión de sus perfiles. La API (`com.amuse.client.features.core.UserManagementFeature`) permite obtener información de los usuarios por su nombre o también extraer usuarios aleatorios.

- **Clock synchronization**: Permite a distintos clientes de la aplicación ejecutar acciones al mismo tiempo, esto es especialmente importante para aplicaciones en tiempo real, donde el inicio de una partida debe darse al mismo tiempo para todos los usuarios.

- **Text message exchange**: Implementa el envío y recepción de mensajes entre los usuarios. La API necesaria se implementa por medio de la interfaz (`com.amuse.client.features.core.TextMessageFeature`). Si el receptor no está conectado el mensaje se almacena para su posterior envío.

- **P2P pipe management**: Permite conectar los clientes de dos usuarios para el intercambio de objetos. Se puede acceder mediante la API proporcionada por la interfaz (`com.amuse.client.features.core.PipeManagementFeature`).

Además, ofrece dos características específicamente diseñadas que permiten organizar y administrar partidas de juego.

- **P2P match coordination**: Accesible mediante la interfaz (`com.amuse.client.features.gaming.MatchCoordinationFeature`). Permite organizar partidas uno-a-uno con la posibilidad de que sean persistentes. La gestión de partidas se basa en invitaciones: El organizador de la partida invita al oponente especificando su nombre o pidiendo un oponente aleatorio a la plataforma AMUSE. Las aplicaciones desarrolladas usando esta aproximación no requieren de un servidor y por lo tanto se pueden construir usando solo clientes.

- **Centralized match coordination**: API implementada mediante la interfaz (`com.amuse.client.features.gaming.GamesRoomFeature`) que soporta la organización de partidas con 2 jugadores o más, que además puede unirse y abandonar partidas mientras están en marcha. Utiliza la metáfora de mesa de juego (*table metaphor*): El organizador crea una mesa a la que los jugadores se pueden unir, cuando se alcanza el número mínimo de jugadores la partida empieza. Las mesas de juego se organizan en salas (*rooms*) a las que los jugadores deben unirse antes de crear o entrar en las mesas de juego.

5.2.3. Agentes internos de AMUSE

AMUSE ofrece algunos agentes que trabajan internamente sin que el usuario lo sepa (también llamados agentes de tipo *backend*) y se encargan de tareas generales que facilitan el desarrollo de aplicaciones.

- **UMA** (*User Management Agent*): Gestiona los perfiles de usuario y las relaciones entre los mismos con una aproximación social.

[1] `http://amuse.dmi.unipr.it/amuseConsole`

- **GRA** (*Games Room Agent*): Se encarga de administrar el espacio de juego en aquellos juegos con interacción síncrona.
- **AMA** (*Application Management Agent*): Teniendo en cuenta la perspectiva PaaS, se encarga de gestionar los juegos y su ciclo de vida.
- **MTA** (*Match Tracer Agent*): Se encarga de proporcionar los servicios necesarios para juegos que necesitan persistencia y opciones de reinicio.

Finalmente, AMUSE ofrece un agente genérico **MMA** (*Match Manager Agent*) a cargo de la interacción entre la aplicación del cliente y los agentes *backend* antes mencionados. Es el único agente propio de la plataforma con el que los usuarios interactúan de forma abierta en la versión actual de AMUSE.

5.3. Entorno de desarrollo

La plataforma PaaS oficial se encuentra alojada en los servidores del departamento de matemáticas e informática[2] de la Universidad de Parma, los clientes se deben conectar a ella antes de explotar las características de AMUSE y la lógica de servidor. Por supuesto, es conveniente contar con una instalación de AMUSE local durante el desarrollo.

Para la instalación de este entorno local existen dos opciones:

- Instalar Wade, WOLF y AMUSE mediante los ficheros fuentes proporcionados en las webs oficiales de cada uno, lo que requiere además un proceso bastante tedioso de configuración (perfectamente detallado en las guías disponibles).
- Utilizar una máquina virtual[3] con todo instalado y configurado disponible en la web oficial de AMUSE. Se puede utilizar software de virtualización como VMware u Oracle VirtualBox.

Se se desea instalar todo de forma local como se describe en la primera opción, se deberán seguir las guías instalación y configuración disponibles en la web oficial.

- WADE Installation Guide [9]
- AMUSE Start-up Guide [8]

El proceso de instalación y configuración viene descrito paso a paso en ambas guías, quedando fuera del alcance de esta memoria.

[2] `http://amuse.dmi.unipr.it/`
[3] `http://jade.tilab.com/dl.php?file=amuseVM-1.0.zip`

5.4. Ejemplos desarrollados

En este apartado se describe el tutorial oficial de la plataforma AMUSE, el juego multijugador M-Tris[4].

5.4.1. Tutorial M-Tris

El juego M-Tris es una extensión del famoso tres en raya en el que participan 3 jugadores en un tablero de 6x7 casillas (ver figura 5.2). Cada jugador puede ver las casillas que ha marcado él con un círculo rojo y las casillas que han marcado los otros dos con una cruz negra (pero no puede distinguir quién de los dos tiene una casilla concreta). Cuando un jugador marca tres casillas consecutivas se anota un punto. Al terminar la partida el jugador con mayor puntuación gana, y en caso de empate no gana nadie.

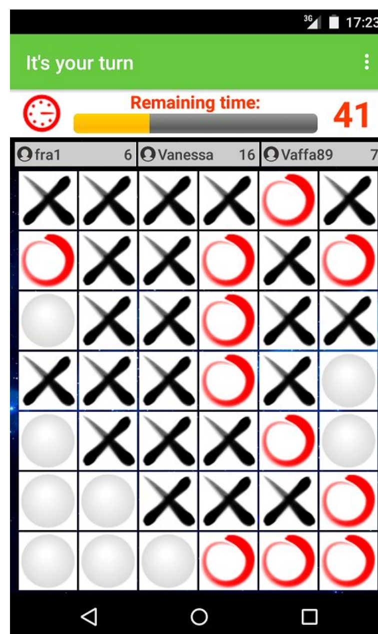

Figura 5.2: Pantalla principal del cliente de juego de la aplicación multijugador M-Tris.

Además de implementar la interfaz de usuario en el cliente y las lógicas de juego en el servidor, existen algunos problemas que se deben resolver:

- **Organización de partidas**: ¿Cómo coordinar a 3 jugadores para que jueguen en el mismo tablero?
- **Partidas de juego**: ¿Cómo aseguramos que los jugadores solo pueden marcar casillas en su turno? Y, ¿Cómo mostramos a cada jugador la casilla marcada en ese turno?
- **Generales**: Identificación de usuarios, mejor puntuación, gestión de amigos, como saber si un jugador está conectado o no, ...

[4]http://jade.tilab.com/amuseproject/amuse-tutorial

Aplicación del cliente

Como es habitual, el cliente Android se compone de un conjunto de actividades que implementan las pantallas necesarias y una clase de aplicación (`com.amuse.mtris.MTris`) que proporciona acceso a las características de la plataforma AMUSE.

Siguiendo la filosofía organizativa de AMUSE, las salas y sus mesas se organizan según se puede ver en el diagrama de la figura 5.3.

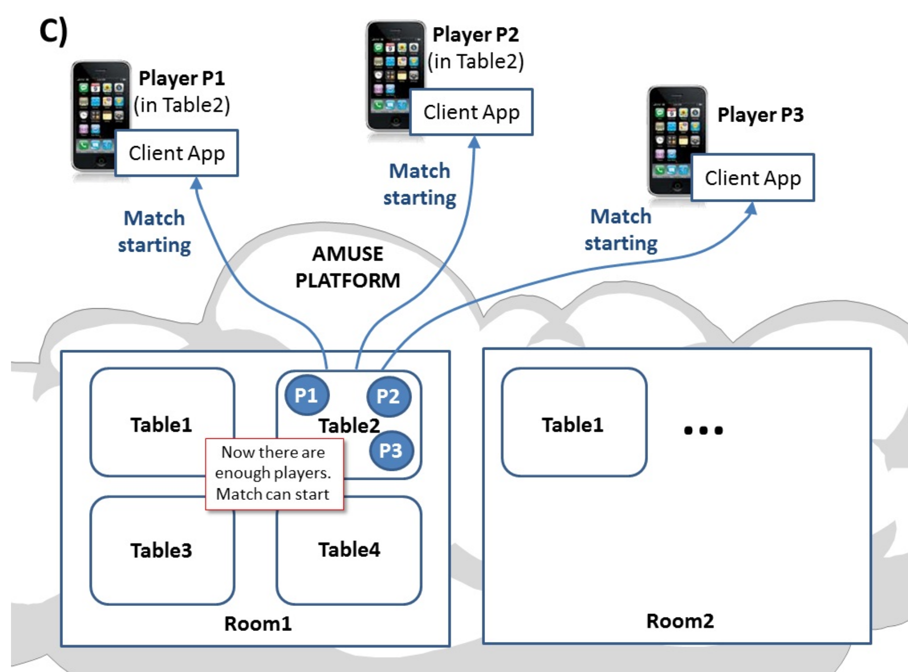

Figura 5.3: Diagrama representando la disposición de mesas de juego por salas.

Cabe destacar que, con el fin de simplificar las cosas, la implementación actual de M-Tris solo cuenta con una única sala (*Room1*) y diez mesas de juego creadas al inicio. No se permite a los jugadores crear nuevas mesas, solo unirse a ellas. Cuando una mesa alcanza el número mínimo de jugadores (por defecto tres) empieza una partida.

Todas las clases necesarias para la implementación del cliente Android están incluidas en el paquete (`com.amuse.mtris`):

- **StartupActivity**: Se lanza al inicio de la aplicación. Conecta al cliente con la plataforma AMUSE y permite al usuario introducir sus credenciales.
- **WelcomeActivity**: Se muestra cuando la conexión se ha establecido. Muestra los botones que permiten al usuario activar las otras actividades: `MainActivity`, `UserInfoActivity`, `FriendsActivity` y `AboutAmuseActivity`.
- **MainActivity**: Permite al usuario empezar una partida.
- **UserInfoActivity**: Permite al usuario administrar su perfil gracias a `UserManagementFeature`.

- **FriendsActivity**: Gestionar los amigos y ver su estado de conexión.
- **AboutAmuseActivity**: Una pequeña descripción sobre la plataforma AMUSE.

Aplicación del servidor

La parte del servidor implementa toda la lógica de salas y partidas, así como del estado de cada juego y las puntuaciones de los usuarios. Utiliza la `GamesRoomFeature` como una o varias instancias de un agente que extiende la clase `GamesRoomAgent`. Cada una de estas instancias implementa una sala de juego, como se ha mencionado con anterioridad solo se usa una por motivos de simplicidad.

Cada agente de tipo `GamesRoomAgent` mantiene cierto número de objetos de tipo `TableManager` que representan las mesas de juego que contiene la sala.

Las mesas de juego se pueden crear de forma automática mediante el método `createTable()` de la clase `GamesRoomAgent` o bien por petición de un usuario (mediante una llamada al método `createTable()` de la clase `Room`).

En la implementación actual de M-Tris se crean 10 mesas de juego en la única sala disponible automáticamente al inicio. Cuando se crea una mesa de juego es posible especificar el número mínimo de jugadores necesarios para que la partida empiece, así como su número máximo. También se puede prohibir a los usuarios que se unan a una partida en marcha.

Para terminar se crean las estructuras específicas de M-Tris:

- El objeto `Board`, que contiene la estructura de datos del tablero (en nuestro caso una lista bidimensional de enteros).
- El objeto `MTrisStatus` que contiene el tablero y la información relevante a la partida (jugadores, estado, ganador, etcétera.).

Ambas objetos se asocian a la mesa de juego mediante el método `setGameSpecificData()` de la clase `TableManager`.

Otros métodos relevantes para el funcionamiento del juego son:

- `handleMatchStartup()`: Se invoca en cuanto se alcanza el número requerido de jugadores en la mesa.
- `handleMove()`: Se invoca tras cada movimiento de un jugador.
- `handlePlayerJoined()`: Cuando un jugador se una a la mesa.
- `handlePlayerLeft()`: Cuando un jugador abandona la mesa.

Vista del proyecto

El proyecto, una vez importado en Eclipse, sigue una distribución de ficheros y carpetas como la que se puede ver en la figura 5.4.

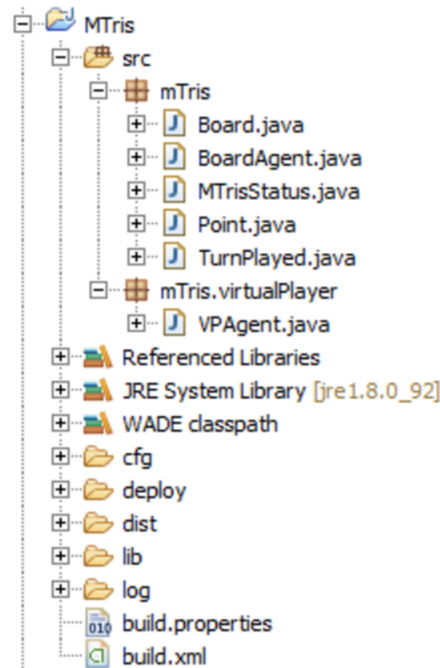

Figura 5.4: Vista de la distribución del juego multijugador M-Tris desde el navegador de proyectos de Eclipse.

Además de los ficheros de configuración y los directorios utilizados por WADE se distribuye en las siguientes clases:

- `Board`: Mantiene la estructura de datos del tablero de juego así como algunas funciones necesarias para su uso y mantenimiento.
- `BoardAgent`: Extiende `GamesRoomAgent` y se encarga de la gestión de las salas de juego. También añade los tipos de datos necesarios para realizar las jugadas (Point y TurnPlayed) que se usarán al realizar movimientos.
- `MTrisStatus`: Es el objeto pasado como `GameSpecificData` para realizar jugadas. Contiene el tablero, la lista de jugadores, las puntuaciones y el jugador del siguiente turno entre otras.
- `Point`: La estructura de datos de una casilla, simplemente valores `X` e `Y`, junto con sus *getters* y *setters*.
- `TurnPlayed`: Contiene la estructura de datos de un turno de juego. El punto marcado, la puntuación actual y el siguiente jugador en tirar.
- `VPAgent`: Extiende la clase `VirtualPlayerAgent` y la interfaz `GamesRoomFeature.Listener` y define la lógica asociada a un jugador máquina para poblar las salas en entornos de pruebas.

Además, dentro de la carpeta `cfg` se encuentra el fichero de configuración `application.xml`. Este fichero contiene los agentes que se instancia al inicio del servidor AMUSE, en nuestro caso un agente de tipo `GamesRoomAgent` (*room1*) y cinco agentes virtuales de tipo `VPAgent`.

5.4.2. Gestión autónoma de una intersección de tráfico

Se propone desarrollar una simulación mediante el paradigma multiagente del comportamiento autónomo negociado de los vehículos en las intersecciones sin regulación semafórica (ver figura 5.5).

La idea es utilizar el sistema de turnos que ofrece AMUSE para que los vehículos vean la intersección como un tablero de juego. De esta forma se puede plantear el problema como una partida de juego, donde los vehículos se mueven por turnos y compiten por el espacio físico de la intersección.

Figura 5.5: Vehículos en una intersección de tráfico típica.

Para simplificar el problema se representarán las intersecciones como un conjunto de carreteras (llamados segmentos) con varios carriles (*Lanes*) cada uno. A efectos de la simulación las intersecciones mantendrán referencias tanto los segmentos entrantes como a los salientes, pero los segmentos que se tendrán en cuenta a la hora de dirigir el tráfico son los entrantes.

Una vez un vehículo haya cruzado la intersección y se incorpore a un segmento de salida deberá empezar a tomar decisiones sobre su siguiente intersección, aquella en la que el actual segmento sea uno de entrada. Para ilustrar este comportamiento se puede ver la figura 5.6.

Por norma general un segmento vendrá definido por tener una intersección de entrada y otra de salida de manera que los vehículos puedan recibir una lista de intersecciones a visitar como su ruta aleatoria.

La viabilidad (o ausencia de la misma) para utilizar AMUSE como plataforma de desarrollo para esta aplicación se estudia en el siguiente apartado.

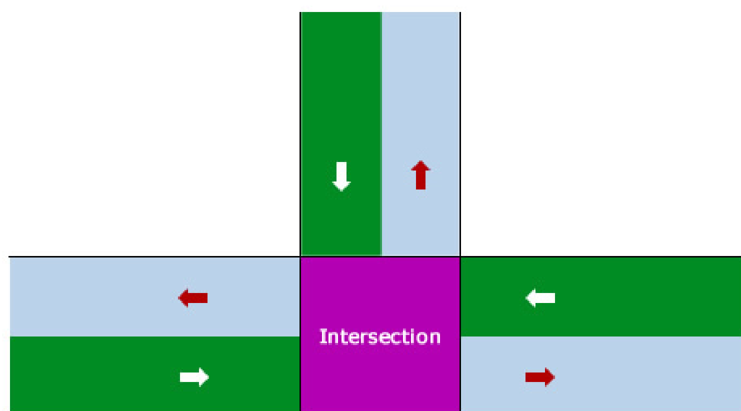

Figura 5.6: Diagrama de la distribución de segmentos en una intersección. Los segmentos verdes son los de entrada a la intersección como simboliza la flecha blanca. Los segmentos azul claro son de salida, marcados con una flecha roja. En ambos casos las flechas denotan el sentido del tráfico.

5.5. Conclusiones sobre AMUSE

AMUSE implementa muchas herramientas que hacen que el desarrollo de juegos multijugador basados en agentes sea mucho más sencillo. Ofrece una capa superior de abstracción a la programación puramente con JADE al ocultar todo el paso de mensajes y comportamientos propios de agentes JADE y ofrecer al usuario simples clases a extender y métodos a implementar.

Aunque sólo proporciona dos paradigmas de diseño de juegos multijugador: el puramente P2P (*solo clientes*) o el centralizado (*clientes y servidor*) mediante mesas de juego, se puede extender a otros en el futuro. Esto se debe a que proporciona muchas herramientas que no son específicas a juegos, entre otras: la gestión de usuarios, la sincronización de agentes o el intercambio de mensajes de texto.

Del mismo modo que WADE aporta muchas mejoras sobre JADE aparte del BPM y Workflows, AMUSE contiene gran cantidad de herramientas y comportamientos que agilizan el desarrollo no solo de juegos, sino de todo tipo de aplicaciones multiagente tanto comerciales como dedicadas a la investigación.

Incluso durante la etapa de realización de tutoriales se puso de manifiesto el temprano estado de esta plataforma para el desarrollo de juegos sociales multiagente.

Se encontraron varias características que no solo no funcionaban, sino que habían sido desactivadas en versiones anteriores por sus comportamientos inesperados. Un claro ejemplo de este tipo de imperfecciones quedó patente al intentar modificar las partidas para que permitiesen a los usuarios unirse en cualquier momento del juego. Tras incontables horas perdidas buscando una solución por nuestra parte resultó que la versión utilizada de AMUSE tenía problemas a la hora de gestionar las mesas de juego bajo estas circunstancias, a pesar de que la habilitación de este tipo de partidas forma parte de la configuración estándar de AMUSE.

Tras varios días en contacto con en el *Mailing List* oficial se dio parte del problema y los

desarrolladores consiguieron solucionarlo. Para más información se puede consultar la conversación[5].

Aunque la resolución de problemas mediante su planteamiento como partidas de juego parece prometedora, lo cierto es que la plataforma AMUSE no está lista para su uso en investigación y mucho menos para su uso comercial. Incluso sin tener en cuenta las características que aún están en desarrollo da la impresión de que algunas de las que se anuncian como funcionales no lo son tanto.

No obstante, es un buen candidato de cara al futuro y se deberá tener en cuenta en sucesivas versiones ya que desde su salida como plataforma de código abierto no ha dejado de recibir mejoras por parte del equipo de Tilab[6].

[5] http://jade.tilab.com/pipermail/jade-develop/2016q2/020419.html
[6] http://jade.tilab.com/

Capítulo 6

Implementación

Contents

6.1. Introducción **53**

6.2. Agentes **54**

6.2.1. WorldAgent 54

6.2.2. IntersectionManagerAgent 54

6.2.3. RadarAgent 57

6.2.4. VehicleAgent 57

6.3. Funcionamiento del simulador **58**

En éste capítulo se expondrán los detalles de implementación, justificándolos cuando sea necesario, junto con los diagramas explicativos necesarios para su comprensión.

6.1. Introducción

Conceptualmente, se desea resolver el problema de la intersección de tráfico mediante un sistema de reservas sin semáforos. Para ello se ha creado un agente `IntersectionManagerAgent` que entre otras cosas mantiene una instancia de la clase intersección, con toda la estructura de datos necesaria para su diseño y funcionamiento. Poniendo la vista en el futura se ha creado un agente `WorldAgent` que es el encargado de crear e iniciar a los agentes intersección, aunque en nuestra simulación solo necesitemos uno. Además, se han creado los agentes `RadarAgent` y `VehicleAgent`, el primero encargado de monitorizar la posición de todos los vehículos y de proporcionar a dichos vehículos información sobre el coche que les precede en la simulación (si es que hay alguno). Por su parte el agente `VehicleAgent` es el actor principal, se encarga de pedir un turno al agente `IntersectionManagerAgent`, procesar dicho turno y pedir una reserva y de desplazarse a la siguiente posición variando su velocidad cuando esto sea necesario.

6.2. Agentes

En ésta sección se describen los agentes que pueblan el sistema, se coordinan y toman decisiones.

6.2.1. WorldAgent

El agente `WorldAgent` se encarga de iniciar al agente `IntersectionManagerAgent` pasándole los parámetros de configuración necesarios. Una vez ha iniciado todas las intersecciones añade un comportamiento destinado a crear e iniciar agentes de tipo `VehicleAgent`, a los que pasa como parámetro un identificador de intersección aleatorio. En nuestra aplicación solo generamos un único agente intersección para poder centrarnos más en el sistema de reservas.

WorldVehicleSpawnBehaviour

El comportamiento al cargo de la generación de vehículos solo recibe un único parámetro, el retraso entre la creación de un agente y el siguiente. De ésta forma podemos controlar con facilidad la saturación que deseamos simular en la intersección, lo que nos permitirá realizar experimentos sobre qué nivel de tráfico puede soportar el sistema y exactamente cuándo se satura.

6.2.2. IntersectionManagerAgent

Por su parte, el agente `IntersectionManagerAgent` mantiene la estructura de datos necesaria para diseñar las características físicas de la intersección, es decir, los carriles por los que circulan los vehículos, las celdas que serán reservadas dentro de la propia intersección y las coordenadas dentro del mapa de cada uno de los elementos.

En cuanto a las interacciones de este agente con el resto, se implementan tres comportamientos:

- `IMVehicleTrackingBehaviour` encargado de escuchar peticiones de registro en la intersección por parte de los agentes vehículo.
- `IMTurnRotationBehaviour` destinado a iniciar la rotación de turnos que recibirán los vehículos, de que dichos turnos se ejecuten por completo en orden y también de establecer una espera mínima entre turnos si fuera necesario.
- `IMServeTurnBehaviour` que lanza un turno propiamente dicho, iniciando una comunicación de tipo Contract-Net con todos los agentes vehículo que tenga registrados.

El funcionamiento de estos comportamientos se describe en mayor profundidad en las siguientes secciones.

Estructura de datos de la intersección

La estructura de datos que se ha implementado para diseñar la intersección propiamente dicha es muy sencilla. Una intersección tiene carriles (o *Lanes* en inglés) y tiene un área centra con celdas dentro.

Los carriles se almacenan como una pareja de coordenadas `(x, y)` en el mapa, una coordenada inicial y otra final dotando al carril de dirección y sentido. Las celdas se almacenan como una coordenada `(x, y)` dentro del área central de la intersección, por simplicidad se ha utilizado una celda por cada carril entrante y saliente.

Los carriles se identifican por su posición de entrada o salida a la intersección y por su orientación cardinal dentro del mapa. Así tenemos cuatro grupos de carriles Norte, Sur, Este y Oeste donde, por ejemplo, el carril identificado por `N0` será el carril que entra a la intersección desde el norte, y el carril `W1` es el que sale por el oeste.

Estos identificadores se utilizarán más adelante para definir las rutas válidas dentro de la intersección, que estarán formadas por un carril de entrada, otro de salida y todas las celdas que sea necesario visitar para llegar de una a otra. Se puede ver el simulador con la estructura funcionando en la figura 6.1.

Figura 6.1: Simulador con la intersección diseñada y esperando vehículos. Las líneas negras representan carriles y los cuadrados oscuros representan celdas transitables.

IMVehicleTrackingBehaviour

Este es un comportamiento de tipo `CyclicBehaviour`, es decir se ejecuta en un bucle infinito esperando a recibir mensajes. El comportamiento recibirá mensajes de tipo `REQUEST` desde cada vehículo bien en el momento de su creación o cuando necesite ser eliminado por haber alcanzado su objetivo.

Cuando recibe una petición de registro responde al vehículo con unas coordenadas iniciales, su carril inicial y un carril aleatorio de destino para simular la ruta. En ese momento el vehículo ya aparece en la interfaz gráfica y recibe su primer turno para comenzar la simulación.

IMServeTurnBehaviour

Este comportamiento se añade cada vez que sea necesario lanzar un turno, en ese momento se inicia un `Contract-Net` con todos y cada uno de los vehículos registrados en ese momento.

El `Contract-Net` funciona de la siguiente forma:

1. Se envía un mensaje de tipo `CFP` (*Call for Proposal*) a los vehículos.
2. Los vehículos responden proporcionando el identificador de sus carriles inicial y final, su velocidad actual y un tiempo estimado de llegada.
3. En este momento el comportamiento intenta realizar una reserva, obtiene la ruta válida entre ambos carriles y trata de reservar las celdas involucradas durante el tiempo necesario en cada una.
4. Si la reserva se completa con éxito le envía las coordenadas de los carriles que debe visitar, de lo contrario responde con una negativa y el vehículo reducirá su velocidad para intentar reservar en el siguiente turno.
5. El comportamiento espera al mensaje `INFORM` de todos los vehículos con su nueva posición tras aplicar cualquier variación de velocidad que haya sido necesaria.
6. Lo último que se hace en cada turno es llamar al método encargado de limpiar las reservas que ya han pasado. Este método recorre todas las reservas y elimina por completo aquellas cuyo tiempo final ya ha pasado.

Es necesario concretar la naturaleza de las reservas. Cada celda mantiene una lista de las reservas vigentes, y cada una de ellas mantiene un tiempo inicial, un tiempo final y el identificador del vehículo que la pidió.

Esto hace que comprobar si una reserva es posible sea muy fácil, solo se debe mirar si el tiempo inicial o final caen dentro de otra reserva ya existente o si la cubre por completo.

6.2.3. RadarAgent

El agente radar mantiene una lista de todos los agentes vehículo junto con sus coordenadas e identificador. Añade un comportamiento llamado `RadarListenerBehaviour` que espera peticiones de los agentes vehículo y su funcionamiento es el siguiente.

1. El comportamiento espera a recibir un mensaje de tipo `REQUEST` de los vehículos, que contiene su identificador y las coordenadas de su posición actual y las de su objetivo actual.
2. El comportamiento actualiza la lista de vehículos con los nuevos datos.
3. Busca en la lista aquellos vehículos cuyas coordenadas objetivo son las mismas y calcula la menor distancia de aquel vehículo más cercano al nuestro y que esté por delante.
4. Para terminar envía esa distancia en un mensaje de tipo `INFORM` al vehículo que inició la comunicación pasa a procesar la siguiente petición.

6.2.4. VehicleAgent

El agente vehículo se moverá por la simulación transitando los carriles y las celdas desde una posición origen hasta su destino tratando de realizar una reserva que le permita completar su ruta.

Cada agente vehículo mantiene atributos y métodos que le ayudan a establecer: su velocidad actual, su velocidad máxima, los carriles de origen y destino junto a sus coordenadas, su ruta completa una vez tenga una reserva válida y la distancia al vehículo de delante.

Para funcionar cuenta con tres comportamientos principales: `VehicleRegisterBehaviour`, `VehicleRadarTrackingBehaviour` y `VehicleProcessTurnBehaviour`.

VehicleRegisterBehaviour

Este comportamiento se añade nada más el agente vehículo es creado por el agente `WorldAgent`. Se encarga de lanzar una petición dirigida al comportamiento `VehicleTrackingBehaviour` del agente `IntersectionManagerAgent` para su registro en la intersección. Como respuesta recibe los identificadores de su carril origen y destino y sus coordenadas iniciales.

La última acción que toma este comportamiento es añadir los dos comportamientos siguientes para que el vehículo pueda iniciar su ciclo de vida normal.

VehicleRadarTrackingBehaviour

`VehicleRadarTrackingBehaviour` se encarga, como su nombre indica, de lanzar peticiones

periódicamente enviando sus coordenadas actuales y las de su objetivo al agente `RadarAgent`; concretamente al comportamiento `RadarListenerBehaviour` de dicho agente.

Las características y funcionamiento concretos de esta comunicación se ha explicado con anterioridad cuando describimos el agente `RadarAgent` 6.2.3 de modo que no se repetirán aquí.

VehicleProcessTurnBehaviour

El funcionamiento del sistema de turnos se ha explicado ya en la sección destinada al comportamiento `IMServeTurnBehaviour` del agente `IntersectionManagerAgent`, no obstante en esta sección se hacen algunos apuntes desde la perspectiva de los agentes vehículo.

En cada turno se envían los identificadores de los carriles origen y destino que el agente vehículo recibió durante la etapa de registro, además de su velocidad actual y un tiempo estimado de llegada en número de turnos.

A continuación el agente espera a recibir una lista de coordenadas si su petición de reserva ha sido registrada en un mensaje de tipo `ACCEPT_PROPOSAL`, o un mensaje vacío de tipo `REJECT_PROPOSAL` si la reserva no se pudo completar.

En caso de que la reserva no se pudiera realizar el vehículo reduce su velocidad y pedirá una nueva reserva en el siguiente turno.

6.3. Funcionamiento del simulador

Con todos los elementos implementados se puede observar una instantánea del funcionamiento del simulador en la figura 6.2. Acortando el tiempo de cada turno se puede obtener un funcionamiento fluido hasta cierto límite para no saturar el sistema y que no se pierdan mensajes. Del mismo modo, reduciendo el tiempo de generación de vehículos se puede forzar esa saturación (ver figura 6.3).

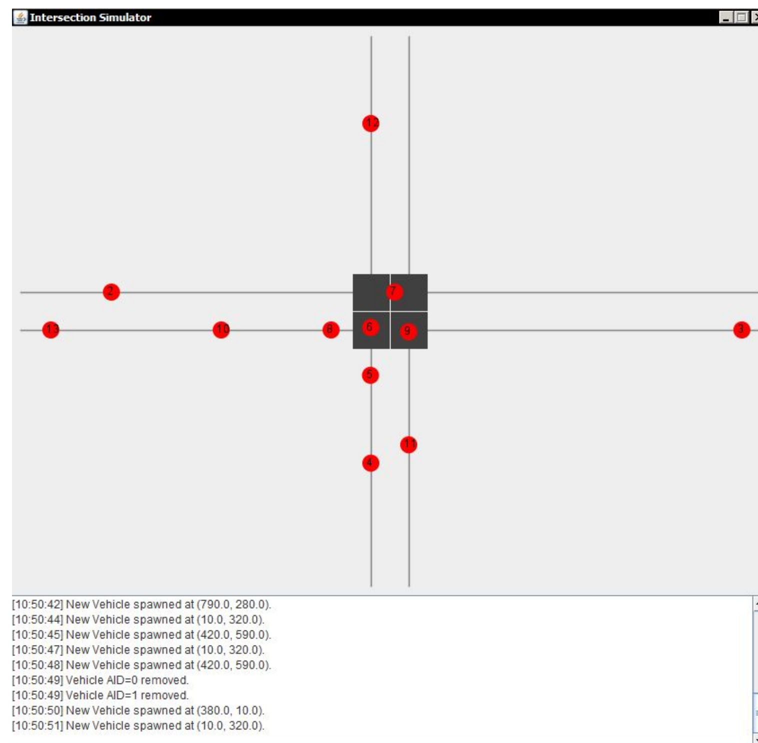

Figura 6.2: Simulador en funcionamiento con un tiempo de generación de vehículos de 1500ms (1.5s). Como se puede ver el sistema es perfectamente capaz de tolerarlo.

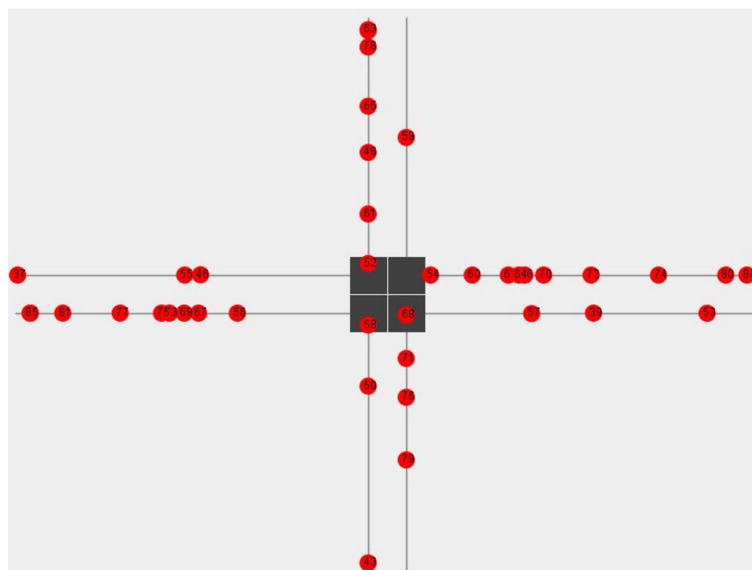

Figura 6.3: Simulador en funcionamiento con un tiempo de generación de solo 500ms (0.5s), saturando por completo la intersección y generando colas en casi todos los carriles.

Capítulo 7

Análisis preliminar del sistema

Contents

7.1. Pruebas de la aplicación 61

7.1.1. Escenario 1: Vehículos con baja agresividad 62

7.1.2. Escenario 2: Vehículos con agresividad moderada 63

7.1.3. Escenario 3: Vehículos con alta agresividad 63

7.1.4. Resultados conjuntos y valoración 64

En éste capítulo se presentan los resultados de las pruebas realizadas, con el objetivo de estudiar tanto el comportamiento de la intersección ante saturación como del retraso inducido en los vehículos.

7.1. Pruebas de la aplicación

Con el fin de analizar el comportamiento de la simulación bajo diferentes estados del tráfico, se han realizado una serie de experimentos preliminares modificando el período de generación de los vehículos, así como la agresividad que presentan a la hora de acelerar o decelerar bajo ciertas condiciones.

Durante la última etapa del proceso de desarrollo se descubrió un problema que podía dejar a toda la intersección bloqueada aun cuando aparentemente nadie la estaba transitando. Este problema se manifestaba cuando a varios vehículos se les denegaba una reserva mientras estaban a la misma distancia de la intersección. Como al denegar una reserva los vehículos deben perder velocidad, y lo hacían exactamente en la misma proporción, ambos entraban en un estado de bloqueo mutuo para todos los turnos futuros.

Para dar solución a este problema se decidió que tanto el factor de aceleración como el factor de frenado fueran aleatorios dentro de un rango pre-establecido. Ambos factores son el número por el que se multiplica la velocidad actual en caso de ser necesario frenar o acelerar, por este motivo el factor de frenado son valores por debajo de la unidad, y el de aceleración por encima.

Para establecer una medida de la saturación de la intersección se mide el retraso (`Delay` en inglés) sufrido por cada vehículo, en comparación a cómo podría haberla superado si nunca bajase su velocidad del límite permitido. Esto se consigue guardando los instantes de creación y borrado de los vehículos, así como su ruta (con el fin de calcular el total de píxeles recorridos).

Todas las pruebas que se ven a continuación se han ejecutado con un tiempo entre turnos de *120ms*. Los resultados mostrados en la siguiente gráfica representan en el eje vertical el retraso o *delay* y en el eje horizontal el número de vehículos por segundo.

7.1.1. Escenario 1: Vehículos con baja agresividad

El primer escenario establece un comportamiento poco agresivo para los vehículos. El factor de frenado oscila entre los valores `0.92` y `0.98`, mientras que el factor de aceleración entre `1.30` y `1.60`.

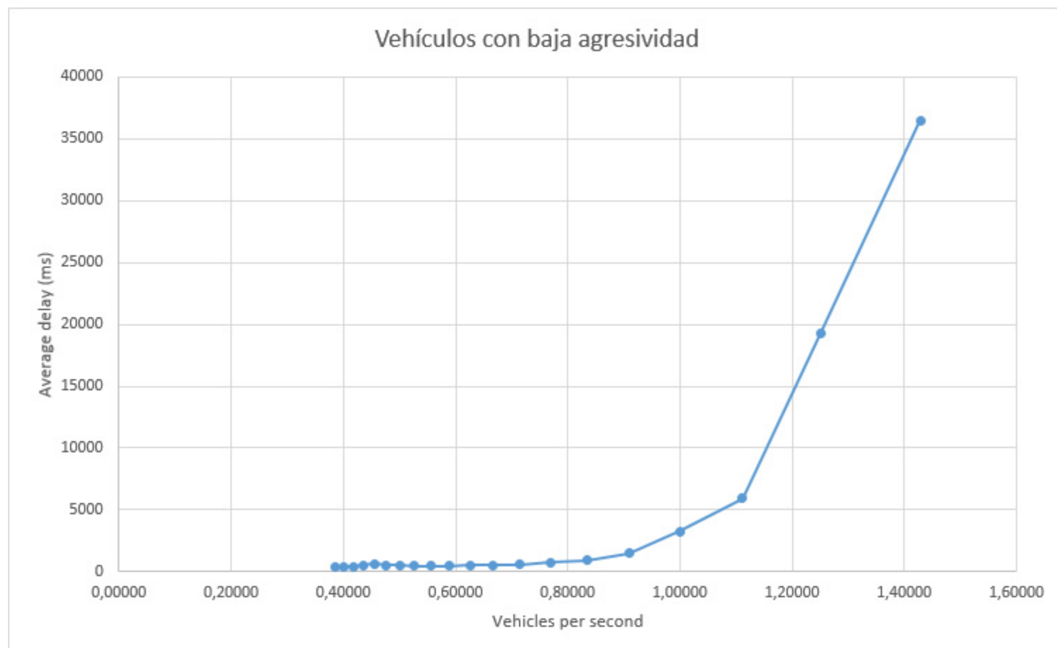

Figura 7.1: Gráfica de saturación con vehículos poco agresivos. Se observan diferencias marginales en el retraso inducido hasta generar aproximadamente un vehículo por segundo.

El comportamiento de la intersección es el habitual, el retraso inducido se mantiene hasta cierto punto en el que se dispara (aproximadamente cuando se genera un vehículo por segundo). Mencionar únicamente que el *delay* nunca llega a cero porque generar y destruir los vehículos conlleva cierto tiempo que puesto que afecta a todos los vehículos por igual se ha dejado en los cálculos.

7.1.2. Escenario 2: Vehículos con agresividad moderada

Para el segundo escenario se establece un comportamiento moderado para todos los vehículos. El factor de frenado oscila entre los valores `0.85` y `0.95`, mientras que el factor de aceleración entre `1.60` y `2.40`.

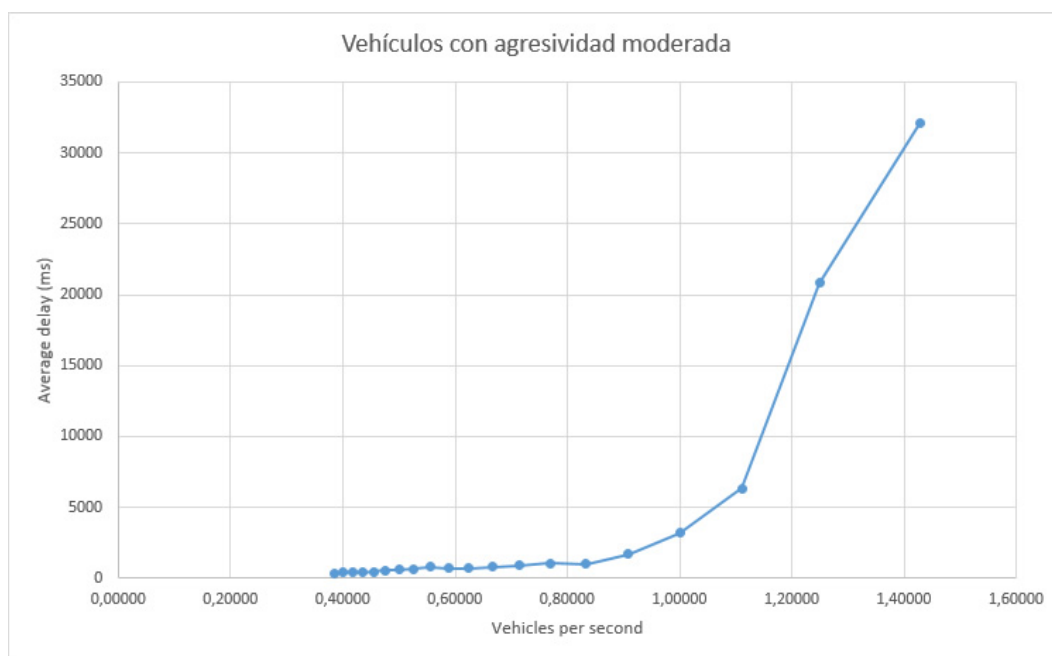

Figura 7.2: Gráfica de saturación con vehículos moderados. Sin grandes cambios sobre la utilización de una baja agresividad.

Se observan pocas diferencias con el comportamiento poco agresivo, durante las mucha simulaciones se ha observado que el retraso sufre oscilaciones más grandes que en el escenario anterior. Esto se debe a vehículos que por ciertas circunstancias reciben una cantidad inusual de reservas denegadas o bien ninguna en absoluto y pasan la intersección sin detenerse. Estos casos extremos parecen ser más habituales en este escenario que en el anterior.

7.1.3. Escenario 3: Vehículos con alta agresividad

El último escenario simulado establece un comportamiento extremadamente agresivo. El factor de frenado oscila entre los valores `0.50` y `0.85`, mientras que el factor de aceleración entre `1.40` y `2.80`.

Como es evidente observando la gráfica un comportamiento en el que los cambios de velocidad que sufren los vehículos son más bruscos de lo habitual no le hace ningún favor a la fluidez de la intersección. Incluso generando pocos vehículos por segundos se satura por completo, y lo hace mucho más rápido que en los escenarios anteriores.

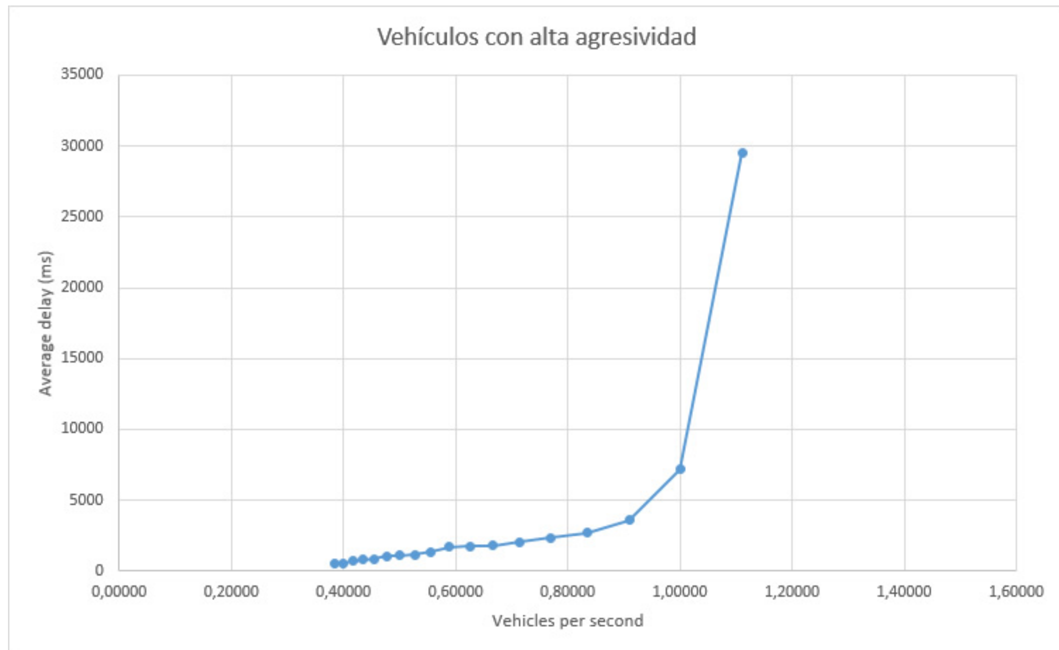

Figura 7.3: Gráfica de saturación con vehículos muy agresivos. Como se puede observar este tipo de conducta agresiva hace que la intersección se sature mucho antes que en los casos anteriores.

7.1.4. Resultados conjuntos y valoración

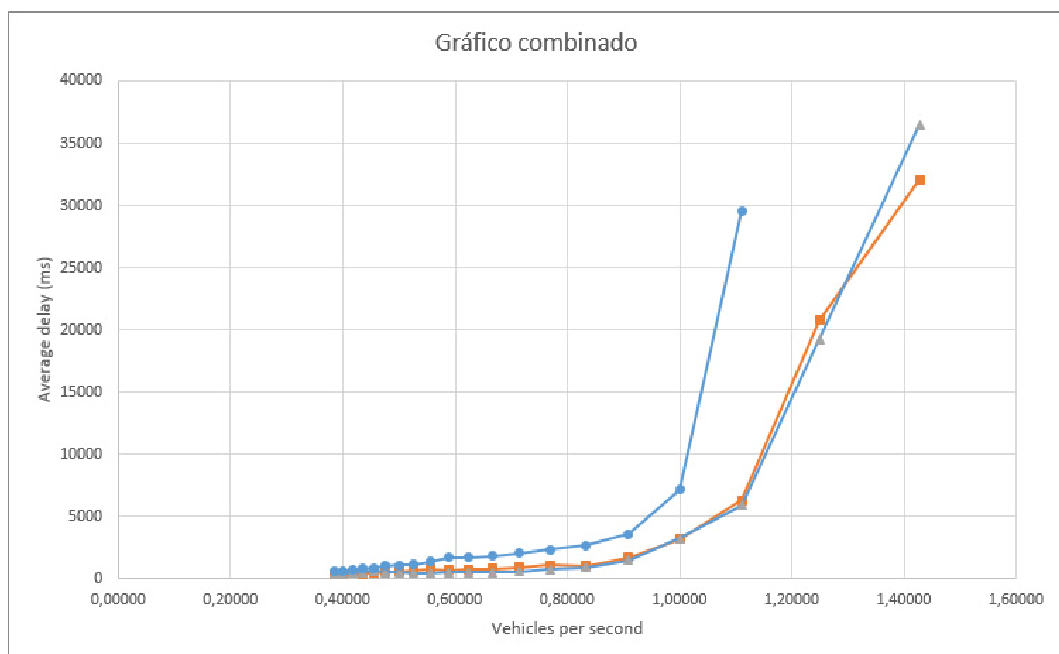

Figura 7.4: Gráfica con todos los escenarios de simulación superpuestos para apreciar mejor las diferencias.

Como valoración a los resultados obtenidos mencionar que aunque la intersección puede soportar cierto nivel de agresividad por parte de los vehículos, parece comportarse de forma

mucho más estable cuando los cambios de velocidad son lo más reducidos posible. También queda patente que en ningún caso puede un comportamiento muy agresivo mejorar el cálculo de reservas y trayectorias, haciendo que la intersección se desborde mucho más rápido y de forma más evidente.

Si se desea consultar la tabla de datos para todos los experimentos realizados se puede encontrar en el anexo A.

Capítulo 8

Conclusiones y trabajo futuro

Contents

8.1. Consecución de los objetivos del proyecto 67
8.2. Conclusiones personales y agradecimientos 68
8.3. Trabajo futuro y mejoras 68

Este artículo contiene las conclusiones a las que se ha llegado, la consecución de los objetivos previstos y las posibles mejoras que se podrían implementar en el futuro.

8.1. Consecución de los objetivos del proyecto

El proyecto está separado en dos bloques, uno de trabajo puramente destinado al análisis y documentación, y otro notablemente más pequeño dedicado a la implementación de un proyecto de simulación de tráfico.

Los objetivos concernientes a la primera parte se han completado con éxito, se ha realizado un estudio exhaustivo de las dos plataformas de desarrollo y se han podido estudiar y documentar sus características más relevantes. La conclusión tal vez no haya sido la que se proyectaba en el inicio del proyecto pero no obstante sigue siendo una conclusión válida.

Es evidente que tanto WADE como AMUSE se encuentran en una fase temprana de desarrollo y en opinión del autor de ésta memoria todavía necesitan refinarse. El hecho de que no existan aplicaciones comerciales ni con relevancia en el campo de la investigación es un claro indicativo de que no están listas.

En cuanto a los objetivos concernientes al segundo bloque, destinado a lo que es el proyecto práctico, se han cumplido los objetivos básicos al proporcionar un entorno de simulación del comportamiento del tráfico en intersecciones.

Se pueden definir unos objetivos de cara al futuro más amplios ya que las funcionalidades

implementadas en el proyecto son muy básicas.

8.2. Conclusiones personales y agradecimientos

Tanto JADE como WADE y AMUSE están siendo desarrollados por algunas de las personas más influyentes y respetadas en el campo de los sistemas multiagente. Las conclusiones a las que he llegado durante el proyecto no han pretendido reflejar una falta de respeto frente al excelente trabajo que realizan estas personas.

Dicho esto, para mi ha sido evidente desde el primer día que ni WADE ni AMUSE animan a nuevos investigadores a estudiar los entresijos de sus respectivas implementaciones. La documentación es escasa, y los artículos encontrados sobre ellos apenas se introducen temas distintos a los que ya se tratan en la propia documentación.

Los tutoriales permiten comprender mucho mejor como se han implementado las características novedosas de ambas plataformas pero, de nuevo, se proporciona escasa o nula documentación sobre partes muy importantes de los mismos.

No puedo evitar pensar que, en cierto modo, la dificultad añadida al proceso de investigación sobre dichas plataformas haya podido penalizar el desarrollo del proyecto. No obstante, ha sido un proceso muy interesante y creo que se han cumplido satisfactoriamente los objetivos del trabajo de fin de máster.

Agradecer a la *Universitat Jaume I* la beca de introducción a la investigación recibida, que ayudó a cimentar las bases sobre las que se sostiene este trabajo.

Me gustaría concluir agradeciendo al departamento de Ingeniería y Ciencia de los Computadores y en especial a mi tutor Luis Amable García Fernández la oportunidad brindada.

8.3. Trabajo futuro y mejoras

Como trabajo futuro me gustaría destacar algunos puntos importantes:

- El primero y principal problema que queda por resolver es la gestión de colisiones cuando dos trayectorias se cruzan pero no comparten ninguna celda en común. En el simulador este tipo de colisiones son frecuentes y evidentes, pero como las trayectorias no comparten celda para el sistema de reserva son perfectamente válidas. Queda como la mejora más importante de cara al futuro.
- Trasladar la implementación del proyecto de intersecciones de tráfico a AMUSE en el momento que la plataforma haya madurado lo suficiente. Todo el proceso de desarrollo ya se ha hecho pensando en el planteamiento de los cruces como una partida en un juego multijugador instanciado por agentes que colaboran y compiten entre ellos por los recursos disponibles, simulando así la filosofía de partida de juego que sigue AMUSE.

- Dotar a las intersecciones de más fidelidad, simulando un espacio físico real en el que los vehículos deben calcular trayectorias y calcular puntos de colisión para actuar en consecuencia. Esto dotaría al proyecto de las herramientas necesarias para empezar a realizar una investigación real sobre el comportamiento de los vehículos y permitiría, en última instancia, realizar predicciones y estudios mucho más útiles. Esto, por supuesto, parece quedar fuera del alcance de los créditos de un trabajo de fin de máster debido a su complejidad.

- Durante la fase de implementación se ha seguido una política FIFO a la hora de dar turno a los vehículos, simplemente se otorgan por orden de respuesta. Sería muy interesante y productivo diseñar otras políticas como por ejemplo: Aleatoria, basada en prioridad, los vehículos más cercanos primero, o una política que una varias de ellas.

- Añadir propiedades a los vehículos que permitan agruparlos, en lugar de ser todos iguales. Esto sería muy útil para simular vehículos prioritarios como ambulancias, coches de policía y bomberos, y otros vehículos de emergencia. Al mismo tiempo, dando importancia a unos vehículos sobre otros sería posible priorizar las reservas de unos frente a otros e incluso borrar aquellas de vehículos no prioritarios para facilitar el acceso a otros que sí lo son.

- Mejorar el proceso de comunicación entre los vehículos y la interfaz, recopilando mensajes y agrupándolos para su procesamiento por lotes. De esta forma se evitarían situaciones de colapso cuando se simula una cantidad irreal de vehículos simultáneamente.

Bibliografía

[1] Tsz-Chiu Au, Shun Zhang, and Peter Stone. Autonomous intersection management for semi-autonomous vehicles. *Handbook of Transportation. Routledge, Taylor & Francis Group*, 2015.

[2] Federico Bergenti, Giovanni Caire, and Danilo Gotta. Interactive workflows with wade. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on*, pages 10–15. IEEE, 2012.

[3] Giovanni Caire, Danilo Gotta, and Massimo Banzi. Wade: a software platform to develop mission critical applications exploiting agents and workflows. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, pages 29–36. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[4] Giovanni Caire, Marisa Porta, Elena Quarantotto, and Giovanna Sacchi. Wolf–an eclipse plug-in for wade. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2008. WETICE'08. IEEE 17th*, pages 26–32. IEEE, 2008.

[5] Giovanni Caire, Elena Quarantotto, and Giovanna Sacchi. Wade: an open source platform for workflows and agents. In *MALLOW*, 2009.

[6] Kurt Dresner and Peter Stone. Multiagent traffic management: A reservation-based intersection control mechanism. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 530–537. IEEE Computer Society, 2004.

[7] FIPA, IEEE Computer Society. Foundation for intelligent physical agents. http://www.fipa.org/. [Consulta: 5 de Mayo de 2016].

[8] Giovanni Caire. Amuse start-up guide. http://jade.tilab.com/amuse/doc/Amuse-Startup-Guide.doc. [Consulta: 6 de Junio de 2016].

[9] Giovanni Caire. Wade installation guide. http://jade.tilab.com/wade/doc/WADE-Installation-Guide.pdf. [Consulta: 5 de Julio de 2016].

[10] Matthew Hausknecht, Tsz-Chiu Au, and Peter Stone. Autonomous intersection management: Multi-intersection optimization. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4581–4586. IEEE, 2011.

[11] Tan Linglong, Zhao Xiaohua, Hu Dunli, Shang Yanzhang, and Wan Ren. A study of single intersection traffic signal control based on two-player cooperation game model. In

Information Engineering (ICIE), 2010 WASE International Conference on, volume 2, pages 322–327. IEEE, 2010.

[12] Remi Tachet, Paolo Santi, Stanislav Sobolevsky, Luis Ignacio Reyes-Castro, Emilio Frazzoli, Dirk Helbing, and Carlo Ratti. Revisiting street intersections using slot-based systems. *PloS one*, 11(3):e0149607, 2016.

[13] Telecom Italia SpA. Amuse - agent-based multi-user social environment. http://jade.tilab.com/amuseproject/. [Consulta: 17 de Julio de 2016].

[14] Telecom Italia SpA. Jade - java agent development framework. http://jade.tilab.com/. [Consulta: 2 de Mayo de 2016].

[15] Telecom Italia SpA. Wade - workflows and agents development environment. http://jade.tilab.com/wadeproject/. [Consulta: 13 de Julio de 2016].

[16] The Eclipse Foundation. Eclipse ide. http://www.eclipse.org/.

[17] C Renato Vázquez, Herman Y Sutarto, René Boel, and Manuel Silva. Hybrid petri net model of a traffic intersection in an urban network. In *2010 IEEE International Conference on Control Applications*, pages 658–664. IEEE, 2010.

[18] WFMC. The workflow management coalition. http://www.wfmc.org/. [Consulta: 4 de Mayo de 2016].

Anexo A

Anexo A

Tiempo de generación (ms)	Vehículos por segundo	Retraso medio (ms)
2600	0,38462	339
2500	0,40000	378
2400	0,41667	431
2300	0,43478	496
2200	0,45455	616
2100	0,47619	512
2000	0,50000	498
1900	0,52632	451
1800	0,55556	464
1700	0,58824	489
1600	0,62500	500
1500	0,66667	496
1400	0,71429	576
1300	0,76923	763
1200	0,83333	897
1100	0,90909	1468
1000	1,00000	3256
900	1,11111	5921
800	1,25000	19256
700	1,42857	36482

Cuadro A.1: Tabla de datos para el experimento con vehículos poco agresivos

Tiempo de generación (ms)	**Vehículos por segundo**	**Retraso medio (ms)**
2600	0,38462	378
2500	0,40000	421
2400	0,41667	462
2300	0,43478	412
2200	0,45455	469
2100	0,47619	564
2000	0,50000	592
1900	0,52632	645
1800	0,55556	786
1700	0,58824	689
1600	0,62500	725
1500	0,66667	793
1400	0,71429	919
1300	0,76923	1063
1200	0,83333	1012
1100	0,90909	1716
1000	1,00000	3205
900	1,11111	6334
800	1,25000	20806
700	1,42857	32069

Cuadro A.2: Tabla de datos para el experimento con vehículos de agresividad moderada

Tiempo de generación (ms)	**Vehículos por segundo**	**Retraso medio (ms)**
2600	0,38462	564
2500	0,40000	572
2400	0,41667	692
2300	0,43478	803
2200	0,45455	862
2100	0,47619	1032
2000	0,50000	1105
1900	0,52632	1168
1800	0,55556	1354
1700	0,58824	1685
1600	0,62500	1728
1500	0,66667	1821
1400	0,71429	2054
1300	0,76923	2340
1200	0,83333	2680
1100	0,90909	3603
1000	1,00000	7154
900	1,11111	29564
800	1,25000	-
700	1,42857	-

Cuadro A.3: Tabla de datos para el experimento con vehículos de agresividad muy elevada