Sistema Multi-Agente Para La Gestión Automática de Plazas De Aparcamiento En Parkings

Yoel Pérez Carrasco
Rubén Peña Baz
Pablo Alfonso López Fernández
David Nieto García
Cesar López Feo

Universidad de Salamanca. Plaza de los Caídos s/n. 37008 Salamanca, España

Resumen(Abstract) *In this work we design and implement a Multi-Agent System integrated with Eclipse, IDE, with which we have created a system for the management of multi-level parking areas. The aim is to facilitate the work of drivers and managers, improving the existing mechanisms and trying to solve the current shortcomings in this area. To this end, the state of the art in the area of Multi-Agent Systems and Virtual Organizations of agents, HashMaps traversal algorithms, is presented. As a result, an open Multi-Agent System has been obtained, a specific algorithm has been used for the routing of routes closest to the exit of each free parking space.*

1. Introducción

Actualmente, la búsqueda de aparcamiento en el centro de las ciudades es una tarea difícil dada la creciente cantidad de vehículos existentes para un espacio que se mantiene constante, o incluso que se va reduciendo. Además, estos espacios se encuentran regulados por zonas de aparcamiento.

En este proyecto se plantea la creación de un sistema multiagente que facilite a los conductores la gestión del pago en estas zonas de aparcamiento, permitiéndoles ahorrar tiempo y dinero. Tiempo, a la hora de tener que localizar una plaza de aparcamiento libre,o validar el ticket. Y dinero, ya que con el modelo de aplicación propuesto, el usuario no necesita indicar la cantidad de tiempo que va a estar aparcado su coche, de modo que el cobro será al salir,así será lo más barato posible,ya que de este modo solo pagará por el tiempo exacto en el parking.

La creación del sistema se plantea mediante la construcción de tres partes diferenciadas, basadas en el diseño de una organización virtual de agentes.El primer agente será el escáner del parking el cual recibirá peticiones de entrada o salida de parte de los clientes del parking,asociadas a la matrícula del vehículo. El segundo agente será el manager del aparcamiento el cual recibirá la matrícula del vehículo a través del agente escáner mencionado anteriormente,además el agente,realizará cuestiones sobre el estado de las plazas al tercer agente,el parking, para reservar,liberar o cobrar,es decir,contabilizar datos relativos a una plaza.El último agente será el parking, el cual comunicará al manager del parking información sobre las plazas,ya sea el estado (libre u ocupada) o el tiempo de uso de una plaza para su

cobro.

Una vez introducido el problema a resolver, en la sección 2. Estado del arte,se recogerán antecedentes sobre los sistemas multiagente,conceptos teóricos y artículos científicos relacionados.En el apartado 3,se expone un modelo propuesto para resolver dicho problema y la explicación de algoritmos utilizados. En el apartado 4 se presentan las conclusiones y resultados obtenidos del proyecto.

2. Estado del arte: antecedentes, otros artículos científicos relacionados, conceptos teóricos.

2.1. Agentes, Sistemas Multi-Agente y Organizaciones Virtuales.

Los agentes han sido empleados en numerosas áreas de la ciencia de la computación desde la segunda mitad del siglo XX (Inteligencia Artificial, Ingeniería del Software, Sistemas Distribuidos, Bases de Datos, etc.). Una de las primeras definiciones del término “agente” se debe a : Sistema computacional basado en hardware o (más habitualmente) basado en software que disfruta de las propiedades de autonomía, habilidades sociales, reactividad y proactividad.

Para unificar todas las definiciones existentes, se ha decidido definir el término agente como una entidad que posee las siguientes características: Autonomía, Situación, Reactividad, Pro-Actividad o Racionalidad, Habilidad social, Inteligencia, Organización, Movilidad y Aprendizaje.

Los agentes pueden colaborar entre ellos. En este momento surge el concepto de Sistema Multiagente (SMA), definido como “la unión de dos o más agentes colaborando en su trabajo con el objetivo de resolver un problema”.

Los sistemas multiagente poseen una computación asíncrona dado su carácter distribuido (descentralizado). No existe un control global, por lo que cada agente se centra en su conducta individual. Cada agente decide libre y dinámicamente, utilizando su autonomía y pro-actividad qué tareas debe efectuar y a quién asigna estas tareas [9], definiendo metas y planes que le permitan alcanzar sus objetivos.

Por otra parte, la analogía entre sociedades de agentes y sociedades humanas es clara. Por ello, aparece el concepto de Organización Virtual (OV), definido como un conjunto de individuos e instituciones que necesitan coordinar sus recursos y servicios dentro de unos límites institucionales". Una OV es un sistema abierto y eminentemente heterogéneo, formado por la agrupación y la colaboración de entidades donde existe una clara separación entre la forma y la función que define su comportamiento.

En la actualidad, existe un amplio abanico de investigaciones enfocadas a la construcción de organizaciones virtuales, a la simulación social basada en agentes y al estudio de su comportamiento.

2.2 Artículos Científicos Relacionados:

El artículo de investigación cuyos autores respaldan la buena reputación de la Universidad De Salamanca: Daniel Hernández Alfageme y Juan Manuel Corchado, basado en un Sistema Multiagente desplegado en un entorno Cloud Computing para gestión de zonas de aparcamiento, es donde hemos encontrado nuestra fuente de inspiración para la elaboración de este artículo y gestor mediante Agente Comunicados.

Enlace al artículo de investigación mencionado.

2.3 Entorno De Desarrollo Eclipse:

El entorno de desarrollo integrado (IDE) de Eclipse emplea plug-ins para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software. Adicionalmente a permitirle a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Python, permite a Eclipse trabajar con

lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet y Sistema de gestión de base de datos. La arquitectura plugin permite escribir cualquier extensión deseada en el ambiente, como sería Gestión de la configuración. Se provee soporte para Java y CVS en el SDK de Eclipse. Y no tiene por qué ser usado únicamente con estos lenguajes, ya que soporta otros lenguajes de programación.

La definición que da el proyecto Eclipse acerca de su software es: "una especie de herramienta universal - un IDE abierto y extensible para todo y nada en particular".

En cuanto a las aplicaciones clientes, Eclipse provee al programador con frameworks muy ricos para el desarrollo de aplicaciones gráficas, definición y manipulación de modelos de software, aplicaciones web, etc. Por ejemplo, GEF es un plugin de Eclipse para el desarrollo de editores visuales que pueden ir desde procesadores de texto wysiwyg hasta editores de diagramas UML, interfaces gráficas para el usuario, etc. Dado que los editores realizados con GEF "viven" dentro de Eclipse, además de poder ser usados conjuntamente con otros plugins, hacen uso de su interfaz gráfica personalizable y profesional.

El SDK de Eclipse incluye las herramientas de desarrollo de Java, ofreciendo un IDE con un compilador de Java interno y un modelo completo de los archivos fuente de Java. Esto permite técnicas avanzadas de refactorización y análisis de código. Mediante diversos plugins estas herramientas están también disponibles para otros lenguajes como C/C++ y en la medida de lo posible para lenguajes de script no tipados como PHP o Javascript. El IDE también hace uso de un espacio de trabajo, en este caso un grupo de metadatos en un espacio para archivos planos, permitiendo modificaciones externas a los archivos.

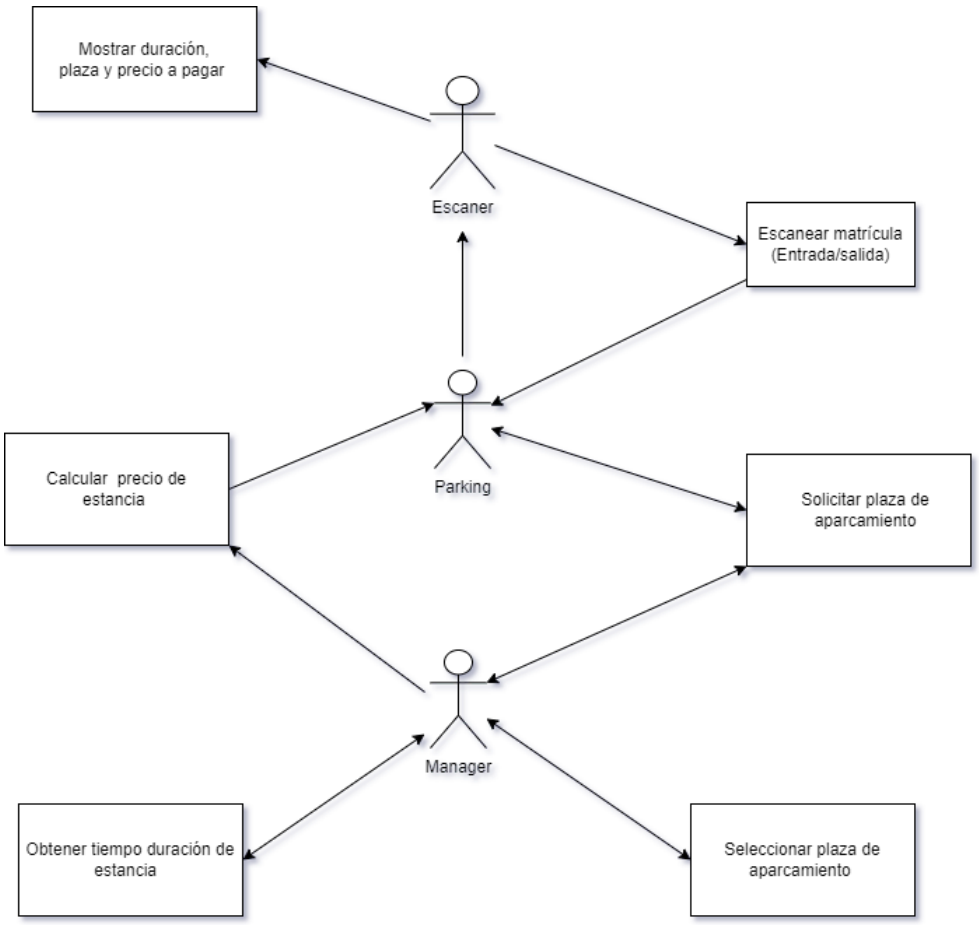
3. Modelo propuesto: explicación detallada del problema a resolver y de forma teórico-práctica cómo se resolverá. Arquitectura del sistema multiagente, protocolo de coordinación, colaboración o comunicación.

3.1 Problema:

El problema principal que intentamos resolver es la gestión simplificada de un parking de coches común, este sistema realiza las operaciones de entrada y salida del coche del parking mediante la recolección de datos gracias a un actor Escáner que capta mediante sensores la matrícula del coche, además, este sabe en qué parking se encuentra dado que cada escáner contiene un número de serie que se corresponde a cada uno de los escáneres de cada parking.

Otro Actor llamado gestor del parking solicita la plaza del aparcamiento al Actor Manager que se encarga de almacenar los datos del vehículo entrante y devolver la plaza y hora de llegada al gestor del parking así mostrandolos mediante una interfaz a través del escáner.

Una vez que el cliente desee sacar su vehículo del parking, el escáner detectará la matrícula de su coche y procederá a solicitar los datos sobre el cobro, la plaza y el tiempo que ha estado aparcado para mostrarlo nuevamente por el escáner.

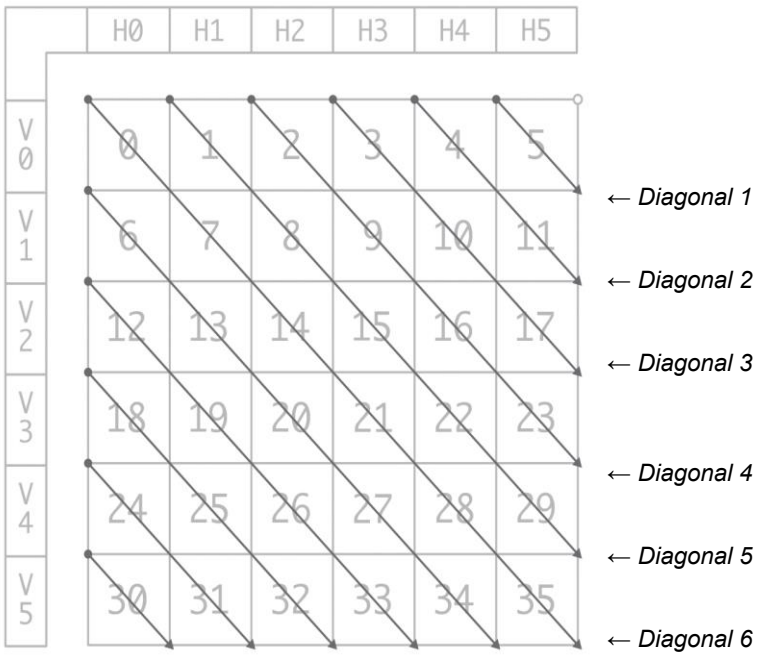


3.2 Optimización De Rutas Para Obtención De Plaza De Parking Más Próxima:

Para el guardado de los datos hemos utilizado un hashmap donde la clave es el nombre del parking y el campo un cubo de elementos de tipo Plaza. La primera dimensión de este cubo representa la planta, la segunda la fila y la tercera la columna de cada plaza.

El objetivo del algoritmo es garantizar que te se te va a otorgar la plaza libre más cercana a la salida en un tiempo eficiente. Se ha considerado que estar en una planta superior es siempre peor que en una planta inferior por lo que el algoritmo recorre el parking por las plantas.

Hemos considerado que la entrada se encuentra en una de las esquinas del parking, la esquina superior derecha en específico. Al inspeccionar cada planta realizamos un recorrido por diagonales empezando por la más cercana a la entrada para así garantizar que estamos siempre a la distancia mínima de esta.


H0
H1
H2
H3
H4
H5
V 0
V 1
V 2
V 3
V 4
V 5
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
← Diagonal 1
← Diagonal 2
← Diagonal 3
← Diagonal 4
← Diagonal 5
← Diagonal 6

4. Resultados y conclusiones: (explicación de la implementación realizada. Puede incluir código de Java pero de manera razonable, deben incluirse diagramas, esquemas, algoritmos o pseudocódigo.)

4.1 Resultados (Explicación con capturas de pantalla de código como función el mismo)

Dentro del código del proyecto encontramos diferentes paquetes como son entities, escaner, extra, manager y parking.

En el primero de estos paquetes mencionados encontramos dos clases. La primera de ellas, denominada *DatosPlaza.java,* donde encontramos las declaraciones, los constructores, los getters y los setters necesarios. Un ejemplo de ellos lo observamos en la figura 1.1.

```
public DatosPlazaDto(String texto, Integer planta, Integer fila, Integer columna, Date horaLlegada) {
        this.texto = texto;
        this.planta = planta;
        this.fila = fila;
        this.columna = columna;
        this.horaLlegada = horaLlegada;
}

public String getTexto() {
        return texto;
}
public void setTexto(String texto) {
        this.texto = texto;
}
```

Figura 1.1

En la segunda clase, dentro del paquete entities, denominada *Factura.java,* encontramos declaraciones, constructores, getters y setters; además de un método que utilizamos para devolver información. Esto lo podemos ver en la figura 1.2.

```
public FacturaDto(String texto ) {
        this.texto = texto;
        this.planta = null;
        this.fila = null;
        this.columna = null;
        this.horaLlegada = null;
        this.horaSalida = null;
        this.dias = null;
        this.horas = null;
        this.minutos = null;
        this.importe = null;
}


@Override
public String toString() {
        return "\n"+texto.toUpperCase() + "\n"
                        +"-----------------------------------------------\n"
                        +"Planta: "+ planta +"\n"
                        +"Fila: "+ fila +"\n"
                        +"Columna: "+ columna +"\n"
                        +"Hora de llegada: "+ horaLlegada +"\n"
                        +"Hora de salida: "+ horaSalida +"\n"
                        +"Total de dias (12€/dia): "+ dias + "\t\t-> subtotal: " + dias*12 + "€\n"
                        +"Total de horas (1€/hora): "+ horas +"\t\t-> subtotal: " + horas + "€\n"
                        +"Total de minutos(0,025€/minuto): "+ minutos +"\t-> subtotal: " + minutos*0.025 + "€\n"
                        +"Importe final: "+ importe +"€\n\n"
                        ;
}


public String getTexto() {
        return texto;
}
public void setTexto(String texto) {
        this.texto = texto;
}
```

Figura 1.2

En segundo lugar, encontramos el paquete denominado escáner, el cual solo consta

de una clase a la que hemos llamado *EscanerAgent.java.* En dicha clase encontramos el código correspondiente al agente escáner, quien se encarga de transmitir la solicitud de entrada o salida con la información necesaria (figura 2.1), así como de devolver los datos una vez realizada la reserva o la liberación de la plaza (figura 2.2).

```
Scanner scanner = new Scanner(System.in);
System.out.print("Introduzca si quiere entrar o salir del parking(entrar/salir): ");
String opcion=scanner.nextLine();
System.out.print("Introduzca matricula: ");
String matricula=scanner.nextLine();
System.out.print("Introduzca numero serie: ");
String serie=scanner.nextLine();


String info = serie + "/" + matricula;
ACLMessage peticion = new ACLMessage(ACLMessage.REQUEST);
peticion.setLanguage(new SLCodec().getName());
peticion.setEnvelope(new Envelope());
peticion.getEnvelope().setPayloadEncoding("ISO8859_1");
try {
        peticion.setContentObject((Serializable)info);
} catch (IOException e1) {
        e1.printStackTrace();
}

if(opcion.equalsIgnoreCase("entrar")) {
        peticion.addReceiver(Utils.buscarAgentes(this.myAgent, "entrada parking")[0].getName());
        peticion.setOntology("entrar");
        this.myAgent.send(peticion);
}else {
        peticion.addReceiver(Utils.buscarAgentes(this.myAgent, "salida parking")[0].getName());
        peticion.setOntology("salir");
        this.myAgent.send(peticion);
}
ACLMessage
msg=this.myAgent.blockingReceive(MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),
MessageTemplate.MatchOntology("ontologia")));
```

Figura 2.1

```
FacturaDto data=null;
try
{
        data = (FacturaDto) msg.getContentObject();
}catch (UnreadableException e){
        e.printStackTrace();
}
if(data.getHoraSalida()!=null) { //Devuelve los datos despues de haber liberado la plaza
        System.out.println(data.toString());
}else if(data.getPlanta()!=null) {//Devuelve los datos despues de haber reservado la plaza
        System.out.println("\n" + data.getTexto().toUpperCase());
        System.out.println("-----------------------------------------------");
        System.out.println("Planta: " + data.getPlanta());
        System.out.println("Fila: " + data.getFila());
        System.out.println("Columna: " + data.getColumna());
        System.out.println("Hora llegada: " + data.getHoraLlegada());
        System.out.println("\n");
}else { //En caso de no haber podido reservar o liberar
        System.out.println("\n" + data.getTexto().toUpperCase()+"\n");
}
```

Figura 2.2

A continuación, encontramos el paquete llamado extra, que está formado por una clase denominada *Utils.java,* y cuyo contenido lo constituyen tres métodos. El primero de estos permite buscar a todos los agentes que implementan un servicio de un tipo dado (figura 3.1); el segundo, envía un objeto desde el agente indicado a un agente que proporciona un servicio de un tipo dado (figura 3.2); y el tercero, que permite buscar los agentes que dan un servicio de un determinado tipo, devolviendo el primero de ellos (figura 3.3).

```
public static DFAgentDescription [] buscarAgentes(Agent agent, String tipo)
{
    //indico las caracter?sticas el tipo de servicio que quiero encontrar
    DFAgentDescription template=new DFAgentDescription();
    ServiceDescription templateSd=new ServiceDescription();
    templateSd.setType(tipo); //como define el tipo el agente coordinador tamiben podriamos buscar por nombre
    template.addServices(templateSd);

    SearchConstraints sc = new SearchConstraints();
    sc.setMaxResults(Long.MAX_VALUE);
    try
    {
        DFAgentDescription [] results = DFService.search(agent, template, sc);
        return results;
    }
    catch(FIPAException e)
    {
        //JOptionPane.showMessageDialog(null, "Agente "+getLocalName()+": "+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            e.printStackTrace();
    }

    return null;
}
```

Figura 3.1

```
public static void enviarMensaje(Agent agent, String tipo, Object objeto)
{
    DFAgentDescription[] dfd;
    dfd=buscarAgentes(agent, tipo);

    try
    {
        if(dfd!=null)
        {
            ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);

            for(int i=0;i<dfd.length;i++)
                            aclMessage.addReceiver(dfd[i].getName());

            aclMessage.setOntology("ontologia");
            //el lenguaje que se define para el servicio
            aclMessage.setLanguage(new SLCodec().getName());
            //el mensaje se transmita en XML
            aclMessage.setEnvelope(new Envelope());
                            //cambio la codificacion de la carta
                            aclMessage.getEnvelope().setPayloadEncoding("ISO8859_1");
            //aclMessage.getEnvelope().setAclRepresentation(FIPANames.ACLCodec.XML);
                    aclMessage.setContentObject((Serializable)objeto);
                    agent.send(aclMessage);
        }
    }
    catch(IOException e)
    {
        //JOptionPane.showMessageDialog(null, "Agente "+getLocalName()+": "+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        e.printStackTrace();
    }
}
```

Figura 3.2

```
protected static DFAgentDescription buscarAgente(Agent agent, String tipo)
{
    //indico las caracter?sticas el tipo de servicio que quiero encontrar
    DFAgentDescription template=new DFAgentDescription();
    ServiceDescription templateSd=new ServiceDescription();
    templateSd.setType(tipo); //como define el tipo el agente coordinador tamiben podriamos buscar por nombre
    template.addServices(templateSd);

    SearchConstraints sc = new SearchConstraints();
    sc.setMaxResults(new Long(1));

    try
    {
        DFAgentDescription [] results = DFService.search(agent, template, sc);
        if (results.length > 0)
        {
            //System.out.println("Agente "+agent.getLocalName()+" encontro los siguientes agentes");
            for (int i = 0; i < results.length; ++i)
            {
                DFAgentDescription dfd = results[i];
                AID provider = dfd.getName();

                //un mismo agente puede proporcionar varios servicios, solo estamos interasados en "tipo"
                Iterator it = dfd.getAllServices();
                while (it.hasNext())
                {
                    ServiceDescription sd = (ServiceDescription) it.next();
                    if (sd.getType().equals(tipo))
                    {
                        System.out.println("- Servicio \""+sd.getName()+"\" proporcionado por el agente "+provider.getName());

                        return dfd;
                    }
                }
            }
        }
        else
        {
            //JOptionPane.showMessageDialog(null, "Agente "+getLocalName()+" no encontro ningun servicio buscador", "Error", JOptionPane.INFORMATION_MESSAGE);
        }
    }
    catch(FIPAException e)
    {
        //JOptionPane.showMessageDialog(null, "Agente "+getLocalName()+": "+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            e.printStackTrace();
    }

    return null;
}
```

Figura 3.3

En penúltimo lugar, encontramos el paquete manager, que está constituido por un total de cuatro clases. Entre estas encontramos la clase denominada

CyclicBehaviourEntrada.java, que contiene el método encargado de transmitir la solicitud de reserva de plaza en un parking especificado (figura 4.1), la clase *CyclicBehaviourSalida.java,* que posee el método que transmite la solicitud de liberación de la plaza así como de proporcionar los datos necesarios para calcular el importe que el usuario debe realizar (figura 4.2), la clase *ManagerAgent.java*, que contiene código que se encarga de registrar dos servicios para reserva y liberación de las plazas del parking (figura 4.3) y la clase *ManagerParallelBehaviour.java,* que contiene un método para que un hilo ejecute un cyclic behaviour y el otro hilo el otro (figura 4.4).

```
	@Override
	public void action() {
		ACLMessage msg=this.myAgent.blockingReceive(MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),
MessageTemplate.MatchOntology("entrar")));

		try
		{
			//Numero de serie*matricula
			String[] datosEscaner = ((String) msg.getContentObject()).split("/");
			String parking = ManagerAgent.parkings.get(datosEscaner[0]);
			FacturaDto factura;
			if(parking!=null) {
				String datosPeticion = "reservar" +"/"+ parking +"/"+ datosEscaner[1];
				//Mirar el mapa y enviar peticion al agente parking
				ACLMessage peticion = new ACLMessage(ACLMessage.REQUEST);
				peticion.addReceiver(Utils.buscarAgentes(this.myAgent, "gestionar aparcamiento")[0].getName());
				peticion.setOntology("ontologia");
				peticion.setLanguage(new SLCodec().getName());
				peticion.setEnvelope(new Envelope());
				peticion.getEnvelope().setPayloadEncoding("ISO8859_1");
				peticion.setContentObject((Serializable)datosPeticion);
				this.myAgent.send(peticion);

				//La respuesta recibida la devuelves al agente que envio la peticion
				ACLMessage msg2=this.myAgent.blockingReceive(MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),
MessageTemplate.MatchOntology("ontologia")));
				factura = new FacturaDto((DatosPlazaDto)msg2.getContentObject());
			}else {//No existe parking buscado
				factura = new FacturaDto("Error -> no existe parking con numero de serie: "+datosEscaner[0]);
			}
			ACLMessage respuesta = new ACLMessage(ACLMessage.REQUEST);
			respuesta.addReceiver(msg.getSender());
			respuesta.setOntology("ontologia");
			respuesta.setLanguage(new SLCodec().getName());
			respuesta.setEnvelope(new Envelope());
			respuesta.getEnvelope().setPayloadEncoding("ISO8859_1");
			respuesta.setContentObject((Serializable)factura);
			this.myAgent.send(respuesta);
		}catch (UnreadableException e){
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}

	}
```

Figura 4.1

```
@Override
public void action() {
	ACLMessage msg=this.myAgent.blockingReceive(MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST), MessageTemplate.MatchOntology("salir")));

	try
	{
		String[] datosEscaner = ((String) msg.getContentObject()).split("/");
		String parking = ManagerAgent.parkings.get(datosEscaner[0]);
		FacturaDto factura;
		if(parking!=null) {
			String datosPeticion = "liberar" +"/"+ parking +"/"+ datosEscaner[1];
			//Mirar el mapa y enviar peticion al agente parking
			ACLMessage peticion = new ACLMessage(ACLMessage.REQUEST);
			peticion.addReceiver(Utils.buscarAgentes(this.myAgent, "gestionar aparcamiento")[0].getName());
			peticion.setOntology("ontologia");
			peticion.setLanguage(new SLCodec().getName());
			peticion.setEnvelope(new Envelope());
			peticion.getEnvelope().setPayloadEncoding("ISO8859_1");
			peticion.setContentObject((Serializable)datosPeticion);
			this.myAgent.send(peticion);


			//La respuesta recibida la devuelves al agente que envio la peticion
			ACLMessage msg2 = this.myAgent.blockingReceive(MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST), MessageTemplate.MatchOntology("ontologia")));

			DatosPlazaDto data = (DatosPlazaDto) msg2.getContentObject();

			if(data.getHoraLlegada()!=null) { //Liberacion exitosa
				Date horaSalida = new Date();
				Long tiempo = horaSalida.getTime() - data.getHoraLlegada().getTime(); //Seconds between dates
				Integer dias=(int) (tiempo/(24*60*60*1000));
				Integer horas=(int) (tiempo/(60*60*1000) - dias*24);
				Integer minutos=(int) ((tiempo/(60*1000)) - dias*24*60 - horas*60);
				Float importe = (float) (dias * 12.0 + horas * 1 + minutos * 0.025);

				factura = new FacturaDto(data,horaSalida, dias, horas, minutos, importe);
			}else {//Error a la hora de liberar
				factura = new FacturaDto(data.getTexto());
			}
		}else {//No existe el parking buscado
			factura = new FacturaDto("Error -> no existe parking con numero de serie" + datosEscaner[0]);
		}


		ACLMessage respuesta = new ACLMessage(ACLMessage.REQUEST);
		respuesta.addReceiver(msg.getSender());
		respuesta.setOntology("ontologia");
		respuesta.setLanguage(new SLCodec().getName());
		respuesta.setEnvelope(new Envelope());
		respuesta.getEnvelope().setPayloadEncoding("ISO8859_1");
		respuesta.setContentObject((Serializable) factura);
		this.myAgent.send(respuesta);
	}catch (UnreadableException e){
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}


}
```

Figura 4.2

```
public void setup() {

        //Registrar dos servicios para reserva y liberación de plazas del parking
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setName("entrada parking");
        sd.setType("entrada parking");
        sd.addOntologies("ontologia");
        sd.addLanguages(new SLCodec().getName());
        dfd.addServices(sd);
        sd = new ServiceDescription();
        sd.setName("salida parking");
        sd.setType("salida parking");
        sd.addOntologies("ontologia");
        sd.addLanguages(new SLCodec().getName());
        dfd.addServices(sd);
        try {
                DFService.register(this, dfd);
        } catch (FIPAException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        addBehaviour(new ManagerParallelBehaviour());
}
```

Figura 4.3

```
public ManagerParallelBehaviour()
{
        super();

        ThreadedBehaviourFactory threadedBehaviourFactory;

        threadedBehaviourFactory = new ThreadedBehaviourFactory();
        cyclicBehaviourEntrada = new CyclicBehaviourEntrada();
        addSubBehaviour(threadedBehaviourFactory.wrap(cyclicBehaviourEntrada));

        threadedBehaviourFactory = new ThreadedBehaviourFactory();
        cyclicBehaviourSalida = new CyclicBehaviourSalida();
        addSubBehaviour(threadedBehaviourFactory.wrap(cyclicBehaviourSalida));

}
```

Figura 4.4

En último lugar, encontramos el paquete que hemos llamado parking, el cual contiene cuatro clases, entre ellas la del tercer agente.

La primera de las clases es la denominada *CyclicReservar.java,* que posee los métodos encargados de la realización de la reserva o la liberación de las plazas del parking. Un ejemplo de esto es lo mostrado en la figura 5.1.1 y 5.1.2.

```
        @Override
        public void action() {
                //Recibes mensaje tipo*nombreParking*matricula
                ACLMessage msg=this.myAgent.blockingReceive(MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),
MessageTemplate.MatchOntology("ontologia")));

                DatosPlazaDto respuesta;
                try
                {
                String[] mensaje = msg.getContentObject().toString().split("/");
                if (mensaje[0].equals("reservar")) {
                        respuesta = reservarPlaza(mensaje[1],mensaje[2]);
                }else {
                        respuesta = liberarPlaza(mensaje[1],mensaje[2]);
                }

                ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);
        aclMessage.addReceiver(msg.getSender());
        aclMessage.setOntology("ontologia");
        aclMessage.setLanguage(new SLCodec().getName());
        aclMessage.setEnvelope(new Envelope());
                aclMessage.getEnvelope().setPayloadEncoding("ISO8859_1");

                aclMessage.setContentObject((Serializable)respuesta );
                this.myAgent.send(aclMessage);
                }
                catch (UnreadableException e)
                {
                e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
```

Figura 5.1.1

```
public DatosPlazaDto reservarPlaza(String nombre,String matricula) {


        Parking parking = this.parkings.get(nombre);
        for(int planta=0; planta<parking.getParking().length; planta++) {
                 Integer altura = parking.getParking()[0].length, anchura = parking.getParking()[0][0].length;

        for (
            // Recorre los inicios de cada diagonal en los bordes de la matriz.
            Integer diagonal = 1 - anchura; // Comienza con un número negativo.
            diagonal <= altura - 1; // Mientras no llegue a la última diagonal.
            diagonal += 1 // Avanza hasta el comienzo de la siguiente diagonal.
        ) {
            for (
                // Recorre cada una de las diagonales a partir del extremo superior izquierdo.
                Integer vertical = Math.max(0, diagonal), horizontal = -Math.min(0, diagonal);
                vertical < altura && horizontal < anchura; // Mientras no excedan los límites.
                vertical += 1, horizontal += 1 // Avanza en diagonal incrementando ambos ejes.
            ) {
                if(!parking.getParking()[planta][vertical][horizontal].getOcupada()) {
                        parking.getParking()[planta][vertical][horizontal].setOcupada(true);
                        parking.getParking()[planta][vertical][horizontal].setMatricula(matricula);
                        parking.getParking()[planta][vertical][horizontal].setHoraLlegada(new Date());
                        return new DatosPlazaDto("El aparcamiento ha sido reservado con exito", planta, vertical, horizontal,
                                parking.getParking()[planta][vertical][horizontal].getHoraLlegada());
                }
            }
        }
        }
        return new DatosPlazaDto("Todos los aparcamientos estan ocupados");
}
```

Figura 5.1.2

La segunda de las clases es la denominada *Parking.java,* que tiene un método encargado de crear las plazas del parking, además de el getter y setter correspondiente (figura 5.2).

```
public Parking(int plantas, int filas, int columnas) {
        parking= new Plaza[plantas][filas][columnas];
        for(int i=0; i<plantas; i++) {
                for(int j=0; j<filas; j++) {
                        for(int k=0; k<columnas; k++) {
                                parking[i][j][k] = new Plaza();
                        }
                }
        }
}

public Plaza[][][] getParking() {
        return parking;
}

public void setParking(Plaza[][][] parking) {
        this.parking = parking;
}
```

Figura 5.2

La tercera de las clases es la denominada *ParkingAgent.java,* cuyo método que la compone tiene la función de registrar un servicio para gestionar los aparcamientos (figura 5.3).

```
public void setup() {
        //Registrar un servicio para gestionar los aparcamientos
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setName("gestionar aparcamiento");
        sd.setType("gestionar aparcamiento");
        sd.addOntologies("ontologia");
        sd.addLanguages(new SLCodec().getName());
        dfd.addServices(sd);
        try {
                DFService.register(this, dfd);
        } catch (FIPAException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        addBehaviour(new CyclicReservar());
}
```

Figura 5.3

La cuarta y última clase del paquete parking se denomina *Plaza.java,* y contiene las declaraciones, los getters, los setters y el constructor necesario para su uso en el proyecto (figura 5.4).

```
public Plaza(){
        this.ocupada=false;
}


public String getMatricula() {
        return matricula;
}
public void setMatricula(String matricula) {
        this.matricula = matricula;
}
public Date getHoraLlegada() {
        return horaLlegada;
}
public void setHoraLlegada(Date horaLlegada) {
        this.horaLlegada = horaLlegada;
}
public boolean getOcupada() {
        return ocupada;
}
public void setOcupada(boolean ocupada) {
        this.ocupada = ocupada;
}
```

Figura 5.4

4.2. Conclusiones

La parte central del trabajo realizado ha sido la comunicación entre agente y la disciplina de identificación entre ellos utilizando hilos por cada una de las alternativas de entrada o salida.

Por otra parte la abstracción de datos almacenados en una matriz cúbica HashMap, nos crea una versatilidad a la hora de recopilar y extraer datos de ella que suaviza la

complicación del código. A la hora de simular un parking real con plantas y diferentes filas y columnas dentro de él, esta estructura cúbica lo simula a la perfección.

El algoritmo utilizado para recorrer de una forma diagonal la matriz HashMap de clases cúbica, es la parte que más nos costó de pensar y especificar para así encontrar la posición más cercana a la salida como si fuera una función de BackTracking.

4.2.1. Líneas futuras de investigación.

A continuación se enumeran algunas posibles líneas de investigación futuras para el presente trabajo:
Creación de algún método funcional para la obtención de plazas de parking más cercanas a la entrada de un método más limpio y depurado.

Creación de funciones de cobro remoto o algún método de suscripción mensual al servicio del parking para solo tener que registrarse y salir del complejo sin tener que pagar in situ, así facilitando la accesibilidad del parking de modo temporal.

Creación de técnicas de visualización interactivas para los datos mencionados anteriormente.

Información sobre plazas de aparcamiento libres.

Búsqueda de Nuevos Parkings Registrados.