UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Algoritmo distribuido para la asignación de tareas en un equipo de robots NAO

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Fábregues de los Santos, Luis

Tutor: Julián Inglada, Vicente Javier

2014-2015

Resumen

En los últimos años se ha avanzado considerablemente en la robótica móvil y, actualmente, los robots permiten realizar cada vez más tipos de acciones y con mayor precisión. Gracias a estas mejoras, se han podido desarrollar mejores aplicaciones automáticas para ayudar a gente con problemas para desplazarse o comunicarse. También ha sido posible crear sistemas inteligentes que integran interfaces robóticas para el cuidado de personas mayores.

Pese a estos hechos, el enfoque de la mayoría de proyectos no se centra en el consumo doméstico. En la actualidad, las aplicaciones robóticas en este campo están orientadas a usos muy específicos como limpiar el polvo o preparar comida y son incapaces de ayudar en otros campos. Pese a que dichas actividades son de utilidad, existe la posibilidad de que este tipo de tareas se acaben realizando por solo una unidad robótica de propósito general o un conjunto de ellas. Por otro lado, cada vez más empresas ponen a la venta líneas de robots de propósito general que son asequibles para los consumidores, haciendo más viable la posibilidad anteriormente comentada.

Sin embargo, desarrollar un proyecto capaz de distribuir, comunicar y realizar tareas del hogar y de ayuda a personas impedidas sería imposible en el marco de desarrollo. Un prototipo de dicho proyecto y una simplificación podría ser recoger y depositar objetos. Se propone crear un algoritmo que distribuya tareas de recogida y entrega de objetos a realizar entre varios robots Nao que se comunicarán mediante la plataforma de agentes inteligentes SPADE.

Palabras clave: distribución de tareas, robótica móvil, agentes inteligentes, inteligencia artificial,

Tabla de contenidos

1 Introducción

1.1 Robótica móvil y la IA

En los últimos años se ha visto un aumento incremental en la complejidad de los productos robóticos gracias al esfuerzo de una gran cantidad de desarrolladores alrededor del mundo. Esto significa que los robots pueden recrear comportamientos más parecidos a los humanos, se desplazan mejor y su capacidad de comunicación es bastante buena.

Ya no es difícil encontrar noticias relacionadas con los avances de la robótica en una cantidad enorme de campos: aplicaciones industriales, medicina, cara al público en diferentes empresas, actividades lúdicas, etc.

Gran parte del esfuerzo se centra en conseguir robots móviles que sean más rápidos y puedan hacer más variedad de movimientos, en muchos casos para simular actividades humanas como pueden ser correr y saltar. La mayoría de estos avances logran que cada vez se reduzcan más las diferencias entre comportamientos humanos y robóticos, haciendo muy prometedor el futuro de la robótica.

En parte unida a este campo de investigación, es normal encontrar el concepto de inteligencia artificial. Esto se debe a que en muchos casos se quiere de una inteligencia por detrás de la interfaz robótica que dictamine las acciones que realice.

Pese a todo, la inteligencia artificial no sigue obligatoriamente el mismo camino que la robótica ya que sus campos de aplicación también son muy grandes. Desde sistemas de consulta hasta jugadores de ajedrez, pasando por piezas de software capaces de controlar la trayectoria de un cohete espacial.

El desarrollo de sistemas utilizando estos dos conceptos ha sido siempre muy popular y objetivo de investigación durante muchos años. El mero hecho de contar con una entidad que sea capaz de moverse y tomar decisiones guiada por sus propias decisiones ya suscita un gran interés. Si a eso le sumamos el enorme abanico de aplicaciones en el mundo real, desde coches que se conducen solos hasta robots de limpieza autónomos, resulta de elevado interés su estudio.

1.2 Aplicaciones de ayuda a mayores y discapacitados

Las aplicaciones de ayuda a las personas con algún impedimento físico y/o mental han sido uno de los focos en el desarrollo de sistemas en los últimos años. Actualmente, el envejecimiento de la población y la ayuda a personas con una discapacidad han hecho que las interfaces de cara al público se adapten a las necesidades de las personas con problemas para interaccionar con ellas.

Por ejemplo, un país que es un gran parangón de aplicaciones de ayuda es Japón. El país asiático cuenta con dos conceptos clave: una población profundamente envejecida y un desarrollo robótico de vanguardia a nivel mundial. No es raro ver noticias sobre los maravillosos avances en los exoesqueletos que permiten a ancianos y discapacitados pasear con normalidad, por su propio pie. Este tipo de herramientas llevan asociadas unas poderosas y precisas estructuras mecánicas que ayudan al movimiento, pero también requieren de un complejo sistema software que controle las acciones.

En concreto, la empresa japonesa *Cyberdyne* tiene uno de los proyectos más avanzados en este campo: el exoesqueleto conocido como *Hal*. Este sistema es capaz de identificar las señales cerebrales del usuario cuando tiene intención de andar, interpretar dichas señales, moverse acorde e informar al cerebro de que el movimiento se ha realizado correctamente. El proyecto ya ha sido utilizado para realizar tareas de rescate de catástrofes, permitir andar a personas con ciertas discapacidades y ayudar al movimiento de extremidades a gente con dificultades para hacerlo. [1]

También se están implantando androides para el cuidado de enfermos y ancianos, un sector que está ganando muchas mejoras en poco tiempo. En el año 2014, seis ancianos europeos comenzaron a utilizar un sistema informático específicamente diseñado para cuidar a gente de la tercera edad por la universidad de Örebo en Suecia. Este sistema se conforma de varios aparatos de control de salud y una interfaz robótica conectada a un sistema central que monitoriza las actividades y el bienestar del anciano del que están al cargo. [2]

Cabe considerar que mucha de esta tecnología es experimental y requiere de mayor complejidad para poder adaptarse a un número mayor de situaciones. En la mayoría de casos, las necesidades de cada anciano y sus costumbres son muy variopintas y requiere de los desarrolladores adaptar su producto al

usuario objetivo. También hay que considerar que no todo el mundo se fía de estar al cargo de un sistema informático y eso es una barrera que se tendrá que superar antes de poder comercializarlos.

Sin embargo, el desarrollo de estas aplicaciones sigue siendo escaso respecto al peso que debería tener en los campos de la investigación tecnológica e informática. Estos sistemas son complicados de crear y perfilar, sin embargo se debería considerar un esfuerzo extra a la hora de implantar estas tecnologías a la ayuda de los más necesitados.

2 Objetivos

El objetivo de este trabajo consiste en realizar una aplicación distribuida que planifique, asigne y controle el proceso de recogida de objetos mediante el uso de uno o más robots NAO. Este trabajo puede comprender una serie de subobjetivos que se describirán a continuación, agrupados por su prioridad en el desarrollo.

2.1 Subobjetivos

En primer lugar, el esfuerzo del proyecto se centrará en crear un algoritmo que planifique la recogida de objetos, que pueden ser cajas, juguetes, utensilios, etc. Este algoritmo deberá recibir una serie de posiciones en las que se pueden encontrar dichos objetos, una lista de posiciones iniciales para los robots que van a realizar dichas operaciones y la posición de un contenedor en el que depositar dichos objetos durante la ejecución. Cuando termine el proceso de cómputo, el algoritmo debería devolver una lista para cada robot con los objetos que deberán recoger y depositar en el contenedor. Dichas listas estarán ordenadas para minimizar el recorrido que el robot deba realizar.

El segundo apartado versará sobre el desarrollo de un sistema de coordinación basada en comunicación de agentes inteligentes. Cada robot será representado por uno de estos agentes y el sistema de comunicación se dedicará a intercambiar mensajes entre ellos. En un primer momento se plantea que los robots no sean en sí un agente, si no que tengan asignado un agente que lo controle. También se comenzará el trabajo con la intención de desarrollar un sistema centralizado con un nodo coordinador. Este agente recibirá las posiciones iniciales de los robots, las posiciones de los objetos disponibles y ejecutará el algoritmo de distribución. Una vez obtenga los resultados, asignará cada lista al robot correspondiente.

Finalmente, en la tercera sección del proyecto comprenderá las instrucciones necesarias para hacer que el robot físico se desplace a recoger los objetos y dejarlos en el lugar correspondiente. Esta función se integrará en el código de los agentes inteligentes asignados a cada robot y, cuando reciban la lista de objetos, los propios agentes ejecutarán las órdenes necesarias para

desplazar al robot con el objetivo de que recoja los objetos de forma que complete su tarea.

3 Contexto

3.1 Nao

Nao es una línea de robots programables y autónomos puestos a la venta por una empresa francesa conocida como *Aldebaran Robotics* en el año 2008. Estos robots tienen forma humanoide, cuentan con dos cámaras HD, cuatro micrófonos, un sonar, dos emisores y receptores de infrarrojos, una unidad de medición inercial, nueve sensores táctiles y ocho sensores de presión. También cuenta con *Wi-Fi* y conexión *Ethernet*. Estos robots son controlados por un sistema operativo basado en *Linux* llamado *NAOqi* que hace uso de los recursos anteriormente mencionados.

Esta herramienta física será interesante ya que cuenta con una forma de desplazamiento bastante realista y sensores que permitirán que el propio robot evalúe su posición en un sistema de coordenadas cartesiano. También permiten ser accedidos mediante conexión *Wi-Fi* para darles órdenes sin necesidad de cargar fragmentos de código en la propia memoria del robot, cosa que agilizará las pruebas.

Ilustración 1: Robot Nao

En lo referente al desarrollo de código para estos robots, *Aldebaran Robotics* ofrece una *API(Application Programming Interface)* para diferentes lenguajes de programación en la que se pueden encontrar órdenes de todo tipo para controlar una gran cantidad de aspectos del robot. Por ejemplo, las órdenes de desplazamiento y localización del robot será indispensables para el desarrollo de esta tarea y se encuentran disponibles en la *API* que ofrece la empresa distribuidora de Nao.

Aparte de lo que ya ha sido comentado, también cabe remarcar que la propia empresa ofrece un conjunto *software* pensado para el desarrollo en sus

robots. Este paquete incluye las librerías del Nao para cada lenguaje y una herramienta llamada *Choregraphe*. Esta herramienta será de incontable ayuda en el transcurso de la implementación ya que nos permite realizar pruebas con un robot simulado y obtener retroalimentación de forma bastante precisa.

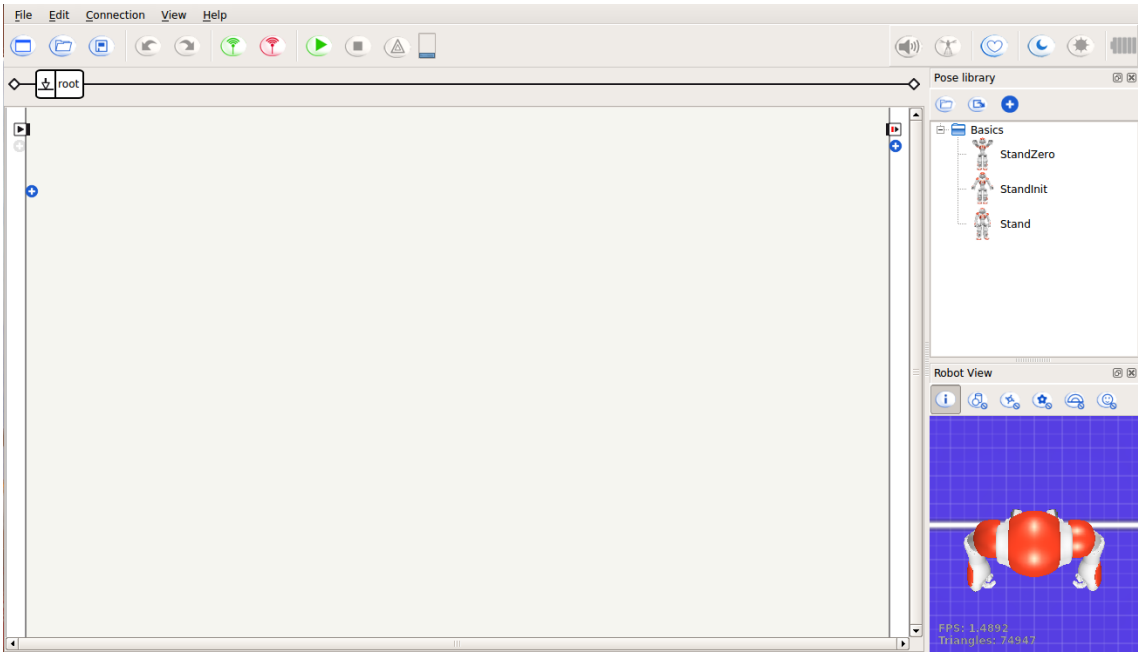


Ilustración 2: Herramienta Choregraphe

Esta herramienta permite interaccionar tanto con un robot físico mediante conexión *wifi* como con un robot virtual para dar órdenes o para comprobar su estado. Existen varias secciones que se pueden visualizar en la interfaz:

- En la parte inferior derecha, hay una representación virtual del estado del robot que es de gran utilidad para comprobar la efectividad de las órdenes ejecutadas o el estado del robot físico.
- En la parte superior derecha se pueden observar una serie de posiciones básicas que sirven para volver al robot a una posición inicial y controlada.
- Para finalizar, existe un panel en blanco en la parte izquierda que sirve para colocar módulos que contienen fragmentos de códigos con instrucciones para ejecutar acciones en el robot.

Como nota adicional, existen más simuladores de sistemas robóticos. Se quiere hablar en particular de *webots*, un *software* muy útil que tiene módulos

para emular varios tipos de robots. Además de ello, es capaz de gestionar más de un robot simulado al mismo tiempo, cosa que hubiera sido de especial utilidad a la hora de probar el proyecto.

Lamentablemente este *software* es de pago y, a falta de recursos, no se pudo hacer uso de él. *Choregraphe* también es de pago, sin embargo permite la simulación nativa de robots Nao y además su periodo de prueba es más largo.

3.2 Planificadores de rutas y/o tareas

Desde que se descubrió la potencia de cálculo de los computadores, uno de los problemas más demandados ha sido el de planificar una serie de tareas de forma óptima. Existen varias formas de resolver estos problemas: se pueden probar todas la soluciones posibles hasta dar con la mejor, plantear un algoritmo de ramificación y poda, resolver una serie de ecuaciones matriciales asociadas a una serie de inecuaciones planteadas por el problema, un algoritmo de programación dinámica, etc.

Esto es notorio en el mundo de las empresas en las que se tienen que distribuir recursos, planificar la producción o ajustar los horarios de descanso del personal. Son tareas de planificación que cada vez se confían más a algoritmos computacionales para optimizarlas y, en muchos casos, son simples sistemas lineares.

Por ejemplo, hace unos años se utilizaron aplicaciones de resolución de cálculos lineales para planificar actividades empresariales relacionadas con la agricultura y la pesca en el continente africano. No solo se utilizaron para aprovechar de la mejor manera posible los limitados recursos de los que disponían, si no también para maximizar beneficios y satisfacer la demanda que había en aquel momento. [3]

El caso concreto de repartir una serie de tareas entre un número de individuos es conocidos como “task allocation” y es un tema de investigación en el entorno científico actual. Usualmente utilizan conceptos como la organización de las hormigas o abejas para determinar la mejor forma de distribuir un conjunto de tareas. Por lo general, sin disponer de un sistema jerarquizado de distribución del trabajo, se plantean diferentes formas de distribuir las tareas y se evalúa cual de ellas es la mejor.

Teniendo en cuenta este concepto, el algoritmo desarrollado sigue la estructura de ramificación y poda iterativa. Este tipo de algoritmos se suelen usar para encontrar caminos óptimos o para distribuir tareas, por tanto pareció acertado para realizar la resolución de esta parte del problema. Se hubiera podido utilizar muchos de los algoritmos anteriormente mencionados, pero teniendo en cuenta que los criterios de optimalidad del problema pueden cambiar en el transcurso del desarrollo se optó por la opción anteriormente mencionada.

Los algoritmos que hacen uso de ramificación y poda construyen y evalúan estados hasta hallar la mejor solución. La diferencia con algoritmos de búsqueda exhaustiva (buscan la mejor solución en todo el espacio de soluciones) es que no se generan todas las soluciones posibles, ya que utilizan sistemas para puntuar cada estado. Sólo el estado que tenga mayor puntuación será evaluado y, en caso de no ser una solución, ampliado.

Su estructura es sencilla, ya que se basa en ir dando pasos para construir una solución de forma iterativa y utilizando el mejor candidato en cada momento. Cuando se elige al mejor candidato se le añaden todas las posibles opciones que tiene y cada opción formará un nuevo candidato. Luego se elegirá al mejor de dichos candidatos hasta encontrar la mejor solución.

Usualmente los algoritmos de ramificación y poda usan un esquema común que posteriormente suele ser ampliado con diferentes mejoras o módulos opcionales. Este es el denominador común de estos algoritmos:

```
Clase RamificaciónYPodaEsquemaBásico:
  Función es_completo(e):
  // Determina si un estado es completo.
  Función seleccionar(A):
  // Selecciona el elemento más prometedor de A.
  Función ramificar(e):
  // Ramifica el estado e.
  Función podar(x):
  // Poda el contenido de x.
  Función solucionador():
   A := EstadoInicial ;
   Mientras no (longitud(A)== 1 y es_completo(A[0])) hacer:
      s := seleccionar(A);
      Eliminar s de A;
      A := podar(A + ramificar(s)))
   return A.pop()
```

Por su naturaleza genérica, este esquema deja muchas incógnitas planteadas: qué estado es completo, cómo se selecciona el elemento más prometedor de un conjunto de estados, cómo ramificar un estado y cómo podar el contenido de un conjunto de estados. Lamentablemente, esas cuestiones son muy dependientes del planteamiento del problema y de los pormenores de la implementación.

Sin embargo, este esquema puede servir para aclarar cómo funciona la ejecución del algoritmo para que posteriormente se comprenda mejor las decisiones tomadas y los detalles de las tareas planteadas en el proyecto.

3.3 Agentes autónomos y sistemas multiagentes

Dentro de entornos en los que se habla de inteligencia artificial es bastante común acabar hablando del concepto de agente inteligente. Un agente inteligente es una entidad que se encuentra dentro de un entorno y actúa dentro del mismo de forma que sus acciones maximicen sus resultados. Por ejemplo, si

tuviéramos un agente que invirtiera en bolsa, dicho agente tendría algoritmos para tomar las mejores decisiones posibles.

En otras palabras, es un sistema que toma decisiones de forma autónoma. En el caso del desarrollo de este proyecto, estos agentes tomarán la forma de piezas de *software* que serán diseñadas con la idea de que decidan qué acciones realizar considerando el estado de su entorno.

Las aplicaciones de los agentes inteligentes son bastante amplias, cualquier lugar en el que se tomen decisiones en entornos, sobretodo distribuidos, utilizando información del mismo puede ser un buen lugar para hacer uso ellos.

Se han utilizando tanto para controlar los movimientos de sistemas que se autoregulan para desplazarse de una forma más precisa o para permitir que una entidad forme parte de una negociación y busque los mejores resultados según los parámetros que le hayan sido implantados.

Los sistemas multiagentes son entornos en los que actúan dos o más agentes inteligentes. Son de especial interés ya que las acciones de un agente afectan el entorno del resto de agentes y dichos agentes tomarán decisiones respecto a decisiones de otros agentes, haciendo que muchas acciones tomen información creada por entidades no humanas.

Las aplicaciones de los entornos multiagentes son también muy numerosas y han logrado grandes resultados. Un campo bastante peculiar en el que aplicar estos sistemas es en las subastas de pescado. Un comprador de pescado entrena a un agente inteligente para no tener que personarse físicamente en el lugar de la subasta y poder estar presente en varias subastas simultáneamente. Los agentes entrenados por cada comprador humano interaccionarán en la puja y tratarán de obtener los mejores resultados posibles. [4]

3.4 Python y SPADE

Para poder trabajar de forma cómoda con las plataformas que ofrece *Aldebaran Robotics,* uno de los lenguajes soportados oficialmente por la API es Python, que será utilizado en este proyecto dada su buena integración con el entorno de simulación.

Python es un lenguaje de programación interpretado de propósito general que tiene una cantidad creciente de librerías, dando lugar a amplias aplicaciones. Existen varias librearías para crear aplicaciones web, realizar cálculos científicos y para realizar aplicaciones gráficas. Está siendo utilizado en la actualidad por Google, Yahoo, CERN y NASA.

Su naturaleza de *script* también es bastante útil a la hora de realizar pruebas en un entorno físico al que solo se puede acceder un tiempo limitado ya que se pueden modificar pequeños detalles sin tener que compilar todo el sistema en cada modificación que se haga.

El sistema que se seleccionó como acertado para la comunicación entre agentes en este proyecto fue SPADE. Es una plataforma multiagente que habilita de forma simple la interacción entre agentes inteligentes basada en la tecnología de mensajería instantánea XMPP. Este último concepto es fundamental ya que los mensajes que intercambian los agentes se agrupan en conversaciones, cosa que permite separar e identificar el contenido de la información transmitida.

Además de esto, SPADE ofrece un par de utilidades para crear un sistema multiagente. Existe a disposición del desarrollador una librearía para Python que ayuda a la creación de agentes y a habilitar los métodos necesarios para recibir y enviar mensajes en el sistema. También dispone de una interfaz web para gestionar la plataforma y los agentes que estén conectados a ella, además de mantener un histórico de las conversaciones. [5]

Hablando con más detalle de la librería, permite usar modelos de agentes que cuentan con mecanismos para conectar a la plataforma, un emisor de mensajes y un conjunto de comportamientos básicos para el agente. Como se puede observar, estas características favorecen bastante la tarea de la implementación ya que dan hecha una parte bastante grande del proceso de crear un sistema multiagente.

4 Planteamiento y diseño del sistema

En esta sección se comentará tanto la descripción del problema como las partes componentes del mismo y cómo han sido implementadas.

4.1 Descripción del problema

Tras comentar el estado de la robótica móvil en el apartado 3.1, se plantea una situación que fácilmente se podría encontrar en la vida real y en la que se basará el desarrollo de este proyecto.

En el caso de personas con discapacidad o niños es planteable la situación en la que se disponga de varios robots en una habitación que tienen que realizar una serie de tareas para ayudar a dicha persona. Estas tareas pueden ser tanto recoger objetos, como abrir una ventana, realizar alguna interacción con un electrodoméstico, abrir la puerta, etc. El objetivo del sistema a desarrollar será un sistema que distribuya el trabajo entre la cantidad de robots de los que se disponga de tal forma que los trabajos que realice cada robot sean los más adecuados frente a varios criterios. Realizar un sistema de tal complejidad está fuera del alcance de este proyecto, por tanto el trabajo se centrará en la recogida de objetos.

Se plantea el siguiente problema: se dispone de R robots, de B objetos y de C contenedores. Cada uno de estos elementos tendrá su posición determinada en un sistema cartesiano. El objetivo del proyecto consistirá en distribuir la tarea de recoger los B objetos entre los R robots de forma que dichos robots depositen los objetos en cualquiera de los C contenedores en el mínimo número de pasos, considerados como coordenadas enteras en el sistema teórico y como decímetros en el sistema virtual y real.

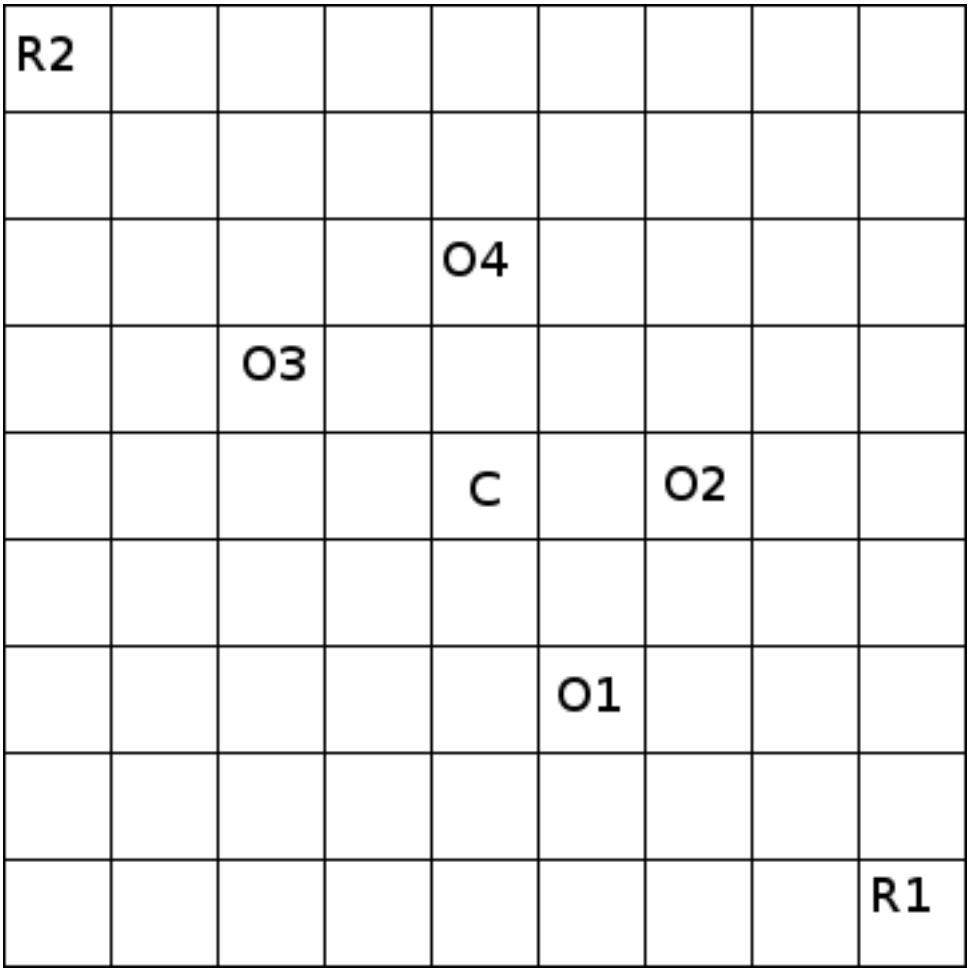


Ilustración 3: Ejemplo distribución de pruebas

Para ello se desarrollará el proyecto en tres partes: algoritmo de planificación, comunicación y coordinación de agentes e implementación del movimiento. En los siguientes apartados se describirá con mayor detalle cada una de estas secciones.

El sistema constará de un número N de robots (reales o simulados), cada robot tendrá asignado un agente que realizará las comunicaciones necesarias para distribuir las tareas. También se usará un agente maestro que reunirá los datos de los robots y ejecutará el algoritmo de planificación para distribuir las tareas, por tanto creando un sistema multiagente. Por último, cada agente incluirá un algoritmo que, dada la lista de objetos a recoger, ordenará los desplazamientos necesarios para completar la tarea.

Como se puede observar se requiere desarrollar un sistema con comunicación y coordinación entre agentes inteligentes, distribución de tareas de forma óptima, el desarrollo de un algoritmo de planificación y la implementación de la serie de órdenes necesarias para que cualquier cantidad de robots físicos o virtuales recoja los objetos de la forma más precisa posible.

4.2 Algoritmo de planificación

El problema a desarrollar requiere de un algoritmo que sea capaz de distribuir objectos de forma equitativa entre uno o varios robots. Además de eso, la distribución debería ser la óptima respecto a varios factores que se considerarán posteriormente. Como se comentó en apartado 3.2, se utilizará un algoritmo de ramificación y poda para resolver esta parte del desarrollo.

Para facilitar la representación de los datos se crearon dos clases, objetos y robots:

- La clase de objetos engloba a los contenedores donde se depositarán los objetos que se deben llevar y dichos objetos. Consta simplemente de la posición en el sistema de coordenadas cartesiano, a la espera de ser ampliado con diversos atributos que puedan tener dichas entidades. Es probable que en un futuro se deban especificar dimensiones de un objeto, la clase de objeto que es o contiene, el peso de dichos objetos, etc.
- La clase de robots tiene exactamente los mismos atributos pero se ha querido separar de la clase de objetos ya que es bastante probable que se añadan atributos, bastante diferentes a los que se han considerado en la clase de objetos, como el estado de batería, la orientación del robot, la cantidad de objetos que ha cargado, etc.

Los argumentos del algoritmo de distribución de objetos serán instancias de dichas clases, contenidas en tres listas diferentes según sean objetos, contenedores o robots. Una particularidad que se debe guardar por razones de consistencia es que las posiciones de los contenedores no pueden coincidir con las posiciones de los objetos, ya que si ambas posiciones coincidieran dicho objeto se podría considerar como entregado.

Dado que este desarrollo se centra en repartir objetos, es natural que las soluciones sean formadas por dichos objetos. Una solución contendrá una lista

de objetos para cada robot. Cada lista contendrá los objetos que le toca recoger a un robot en el orden que minimiza una puntuación que se comentará posteriormente.

Durante la ejecución se mantendrá una cola con prioridad de estados, siendo los estados prioritarios aquellos que tengan una mejor puntuación. En cada iteración del algoritmo, el estado con mejor puntuación saldrá de la cola y se evaluará. En caso de ser una solución, la ejecución acabará y devolverá dicha solución. En otro caso, se ampliará añadiendo un nuevo objeto a dicho estado. Posteriormente se evaluará y entrará en la cola de estados activos.

Dadas las características del problema, el mejor estado al que podemos optar es el que menor puntuación tenga. Esto se debe a que la puntuación de cada estado estará basada en distancias que deben ser recorridas por los robots y las restricciones del problema ordena que se minimice dicho valor. Esto es bastante adecuado sabiendo que las colas con prioridad en Python (los *heap)* usan ordenaciones de menor a mayor, por tanto el primer elemento que se extrae por defecto de esta estructura será el estado con menor puntuación.

Un estado se considera completo si cada objeto a recoger está asignado a un robot. Cuando todos los objetos están dentro de la lista de un robot, ese estado será la solución del sistema si sale de la cola con prioridad en la siguiente iteración del algoritmo.

Los estados serán una tripleta que contendrán: la puntuación del estado actual, la cantidad de objetos que ya han sido considerados y el conjunto de listas de objetos que ya se han distribuido.

Se añade un fragmento de pseudocódigo que clarificará el funcionamiento de una primera aproximación al algoritmo:

```
Funcion solver ( Robots, Contenedor, Objetos ):

  ordenar(Objetos);
  crear lista puntuaciones;
  crear cola estados;
  Por cada objeto en Objetos:
    añadir(distancia(objeto, Contenedor)*2, puntuaciones);
  Fin Por
  estadoInicial := (suma(puntuaciones), Objetos, [])
  añadir(estadoInicial, estados);
  Mientras estados no esté vacía:
    estadoActual := extraer(estados);
    Si estadoActual[2] está vacía:
      devolver estadoActual[3];
    Si no:
      Por cada objeto en estadoActual[2]:
        Por cada robot en Robots:
          nuevoOrden := estadoActual[3];
          añadir(objeto, nuevoOrden[robot]);
          nuevosObjetos := estadoActual[2];
          quitar(objeto, nuevosObjetos) ;
          nuevoEstado := (2*evaluarEstado(nuevoOrden)+
longitud(nuevosObjetos), nuevosObjetos, nuevoOrden);
          añadir(nuevoEstado, estados);
        Fin Por
      Fin Por
    Fin Si
  Fin Mientras
Fin Funcion
```

A continuación se pasará a describir algunas de las funciones que es están usando en el código que se ha escrito:

1. ordenar: ordenará la lista de objetos según su proximidad al contenedor, primero las más alejadas, con la distancia euclidiana.
2. distancia: es un método que devuelve la distancia entre dos puntos usando la distancia euclidiana.

Es momento de comentar uno de los puntos fundamentales de la creación de un algoritmo de ramificación y poda que es la evaluación de estados. Este apartado tiene un especial interés ya que permitirá que el algoritmo se encamine de forma más eficiente a la solución, ahorrando de esta forma tiempo de cómputo y espacio en memoria.

En muchos algoritmos de ramificación y poda se utilizan funciones suplementarias a la evaluación del estado para asignar una puntuación más acorde al potencial de dicho estado. Esto tiene una gran utilidad ya que guía los

estados más rápidamente a una solución. Esta primera versión del algoritmo usa una función de cota optimista que se basa en considerar las cajas que quedan por distribuir.

Una cota optimista es una función que evalúa los datos que aún no se han tenido en cuenta y calcula una puntuación que será mejor (menor en este caso) o igual que la puntuación real que se podría conseguir evaluando dicho estado. Disponer de estos valores es de extremada utilidad ya que guían la solución más rápidamente hacía mejores estados.

En el caso anterior, se considera una cota optimista ya que simplemente considera que los objetos restantes se pueden recoger con solo realizar el movimiento equivalente a una unidad del espacio euclidiano.

Considerando ahora la forma de puntuar, una de las aproximaciones más simples que se puede adoptar es usar la distancia total que se recorrería usando el estado que se está analizando. El cálculo de esta distancia se hará usando el método que parecía en el pseudocódigo anterior, la distancia euclidiana.

Para obtener la puntuación del estado que se está analizando, aparte de sumarle la cota optimista antes mencionada, se utilizó un método para obtener la distancia total que se recorrería utilizando la planificación del estado actual. En este caso, el cálculo se realiza de la siguiente forma:

```
Funcion evaluarEstado ( Estado, Robots, Contenedor ):
  resultado := 0;
  i := 0;
  Mientras i < longitud(Estado) hacer:
    j := 0;
    Mientras j < longitud(Estado[i]) hacer:
      Si i == 0:
        resultado := resultado + distancia(Robots[i], Estado[i][j]);
        resultado := resultado + distancia(Estado[i][j], Contenedor);
      Si no:
        resultado := resultado + distancia(Contenedor, Estado[i][j])*2;
      Fin Si:
      j := j + 1;
    Fin Mientras
    i := i + 1;
  Fin Mientras
  Devolver resultado
Fin Funcion
```

Existen varias peculiaridades en este algoritmo. Se recuerda que la variable *Estado,* en este caso, contiene una lista para cada robot con los objetos que debe recoger. También cabe destacar que solo se considera la posición inicial del robot en el primer movimiento y resto de ellos toman como posición de salida el propio contenedor.

Es fácil de ver porqué ocurre esto: para recoger el primer objeto, el robot se desplaza desde su posición inicial hasta la posición del objeto y lo recoge; una vez hecho esto, se desplaza al contenedor para depositarlo. A partir de ese momento, el robot se desplazará para recoger un objeto desde el contenedor, lo recogerá, volverá al contenedor y repetirá este ciclo hasta terminar de recoger todos los objetos que tenga asignados.

Una vez analizado este algoritmo, se pueden extraer una serie de conclusiones respecto a los resultados que proporciona. La puntuación de cada estado depende de los pasos recorridos y tratará de encontrar la distribución de tareas que menos distancia sea capaz de recorrer para recoger los objetos.

A pesar de que este resultado es interesante a la hora de planificar la ruta más corta y parece ser el objetivo de la distribución, si se plantea su aplicación en un sistema real, se puede observar que minimizar la distancia recorrida no proporciona el resultado que se esperaría. La mejor distribución del trabajo sería aquella en la que los robots, como conjunto, terminaran lo antes posible. Por tanto, la forma de puntuar debe de tener más que ver con el tiempo que con la distancia.

Esto es algo que se puede observar a la vista de los resultados que devuelve el algoritmo de ramificación y poda teniendo en consideración el método de puntuación explicado: la forma de puntuar el estado considera que los objetos más cercanos al contenedor que a un robot no pueden ser el primer objeto recogido por un robot; también ocurre que cuando varios robots alcanzan el contenedor solo uno de ellos recoge los objetos restantes mientras que el otro no hace nada más, acción que tiene sentido respecto al número de pasos pero no respecto al tiempo total de ejecución.

La tarea de medir la cantidad de tiempo que dedica un robot en desplazarse a cierta posición es más complicada ya que requeriría simular y medir el tiempo de ejecución dentro de un algoritmo de distribución, comportamiento nada recomendado para un algoritmo que debería ejecutarse en el menor tiempo posible.

Para conseguir un valor equivalente al tiempo que le cuesta al conjunto de robots recoger todos los objetos, se planteó seguir utilizando las distancias euclidianas entre robot-objeto y objeto-contenedor pero modificando el algoritmo de puntuación anteriormente mencionado. Ahora se considerará solo la mayor distancia que recorre un solo robot.

Este nuevo paradigma hará que la puntuación ignore la distancia que recorra el resto de robots ya que esa distancia se recorrerá simultáneamente y será menor o igual a la que haya sido seleccionada por el algoritmo. De esta forma se dispondrá de un algoritmo que dará prioridad a aquellos estados cuya distancia recorrida sea menor.

A vistas de este nuevo enfoque, se planteó un nuevo método de puntuación de estados que contempla los ligeros cambios que se introducen en dicho algoritmo y que se expresan a continuación:

```
Funcion evaluarEstado2 ( Estado, Robots, Contenedor ):
 crear lista numérica resultados de tamaño longitud(robots) ;
 i := 0;
 Mientras i < longitud(Estado) hacer:
  j := 0;
  Mientras j < longitud(Estado[i]) hacer:
   Si i == 0:
    resultado[i] := resultado[i] + distancia(Robots[i], Estado[i][j]);
    resultado[i] := resultado[i] + distancia(Estado[i][j], Contenedor);
   Si no:
        resultado[i] := resultado[i] + distancia(Contenedor, Estado[i]
[j])*2;
   Fin Si:
   j := j + 1;
  Fin Mientras
  i := i + 1;
 Fin Mientras
 Devolver max(resultado)
Fin Funcion
```

Se puede comprobar que el algoritmo funciona de una forma muy similar al presentado anteriormente, solo que ahora se calcula por separado la distancia que debe recorrer cada robot y se coge el mayor valor dentro de estas distancias. De esta forma se premia que la máxima distancia recorrida sea la menor posible y no solo se puntúa mejor estados que recorran poca distancia general, si no que también se premiará aquellos estados que distribuyan el trabajo de forma que la operación se realice en menos tiempo.

Respecto a la cota optimista, utilizando este método de puntuación de estados ya no es viable. Cuando se consideraba la distancia total recorrida, era una medida consistente con el problema: recoger y depositar una caja en el contenedor debe costar, al menos, una unidad de desplazamiento (teóricamente 2).

Con la nueva forma de puntuar ya no se puede garantizar la consistencia de la cota optimista. Supongamos que llegamos a un estado en el que queda solo un objeto por distribuir y dicho objeto puede ser recogido sin incrementar la puntuación del estado. A la hora de considerar la puntuación de dicho estado junto a su cota optimista, obtendremos una mejor puntuación de la que puede obtener y eso rompe las condiciones de la cota optimista, por tanto dejaría de ser consistente.

Finalizando ya la explicación sobre el desarrollo del algoritmo, hubo una medida adicional que mejoró su funcionamiento. Se suele incorporar a este tipo de algoritmos una funcionalidad adicional se basa en almacenar el mejor estado completo que se alcanzado. Considerando las restricciones propias de este tipo de problemas, ningún estado con una puntuación peor o igual a la puntuación de almacenada puede alcanzar una puntuación mejor que la almacenada.

Por ese motivo, los estados con peores puntuaciones al almacenado no hace falta que entren en la cola de estados. De esta forma se reduce el tamaño de la cola y el número de inserciones, haciendo que el algoritmo ocupe menos memoria. Finalmente, se añade un fragmento de pseudocódigo con la función mejorada:

```
Funcion solver2 ( Robots, Contenedor, Objetos ):

  ordenar(Objetos);
  bestYet := ∞;
  crear lista puntuaciones;
  crear cola estados;
  Por cada objeto en Objetos:
    añadir(distancia(objeto, Contenedor)*2, puntuaciones);
  Fin Por
  estadoInicial := (suma(puntuaciones), Objetos, [])
  añadir(estadoInicial, estados);
  Mientras estados no esté vacía:
    estadoActual := extraer(estados);
    Si estadoActual[2] está vacía:
      devolver estadoActual[3];
    Si no:
      Por cada objeto en estadoActual[2]:
        Por cada robot en Robots:
          nuevoOrden := estadoActual[3];
          añadir(objeto, nuevoOrden[robot]);
          nuevosObjetos := estadoActual[2];
          quitar(objeto, nuevosObjetos) ;
          nuevoEstado := (2*evaluarEstado(nuevoOrden)+
longitud(nuevosObjetos), nuevosObjetos, nuevoOrden);
           Si nuevoEstado[1] < bestYet:
             añadir(nuevoEstado, estados);
             Si estadoActual[2] está vacía:
               bestYet := nuevoEstado[1];
             Fin Si
           Fin Si
        Fin Por
      Fin Por
    Fin Si
  Fin Mientras
Fin Funcion
```

4.3 Comunicación y coordinación de agentes

Una vez descrito el algoritmo de distribución de tareas, se pasa a comentar cómo se ha creado el sistema de comunicaciones entre agentes. Para comenzar, se dispondrá de un agente para cada robot y dicho agente se comunicará con el resto para ejecutar el algoritmo del apartado anterior.

Cada uno de estos agentes deberá informar al agente maestro de varios datos referentes al robot que tiene asignado:

- El nombre del robot es uno de esos datos y se usa con motivos de depuración del código.
- La posición inicial también se debe comunicar para dar información indispensable al algoritmo de distribución como ya se ha visto anteriormente.
- El robot debe informar al nodo maestro de los objetos que percibe. A nivel de la implementación del prototipo, estos datos están integrados en el código ya que las tareas de percibir objetos y su localización son bastante complejas como para formar un proyecto independiente.

Las tareas de comunicación se pueden realizar de una forma centralizada o descentralizada. Para el segundo enfoque solo harán falta los agentes de los que ya se dispone mientras que el primer enfoque requerirá un agente adicional que servirá como nodo maestro.

Las funciones de este nodo maestro son complementarias a las del resto de agentes. Este nodo registrará los datos de los agentes antes mencionados y, una vez estos datos sean recolectados (para todos los agentes de los que se esperaba recibir datos), el nodo maestro ejecutará el algoritmo de distribución de tareas. Cuando tenga las listas de objetos para cada robot, las enviará en un mensaje a cada agente para que sean ejecutadas.

Independientemente del esquema que comunicaciones que se seleccione, los agentes asociados a un robot suelen tener el mismo comportamiento siempre que no sean seleccionados como nodo maestro. En este caso, el comportamiento sigue este esquema simple:

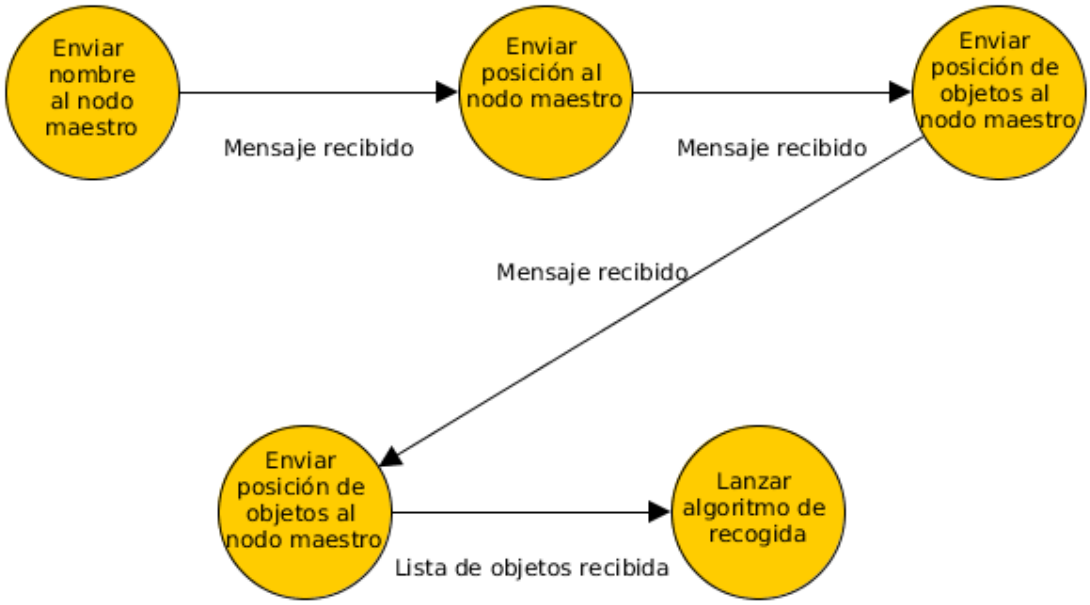


Ilustración 4: Comportamiento general de un agente-robot

Ahora se pasará a describir cada una de las estructuras de comunicación y se dará una breve descripción de los mensajes que se intercambian en el sistema desarrollado.

4.3.a Esquema centralizado

El esquema centralizado se apoya en un agente que no tiene asignado ningún robot que recibirá los datos del resto de agentes, distribuirá las tareas y los coordinará para realizar las acciones de recogida de cajas. Los agentes-robot dependerán del nodo central para recibir datos e indicaciones sobre cuando deben de hacer sus acciones. Este enfoque consigue que las comunicaciones entre agentes se simplifiquen y que los agentes encargados de controlar los robots sean más simples. A continuación se presenta un esquema sobre la organización de los agentes.

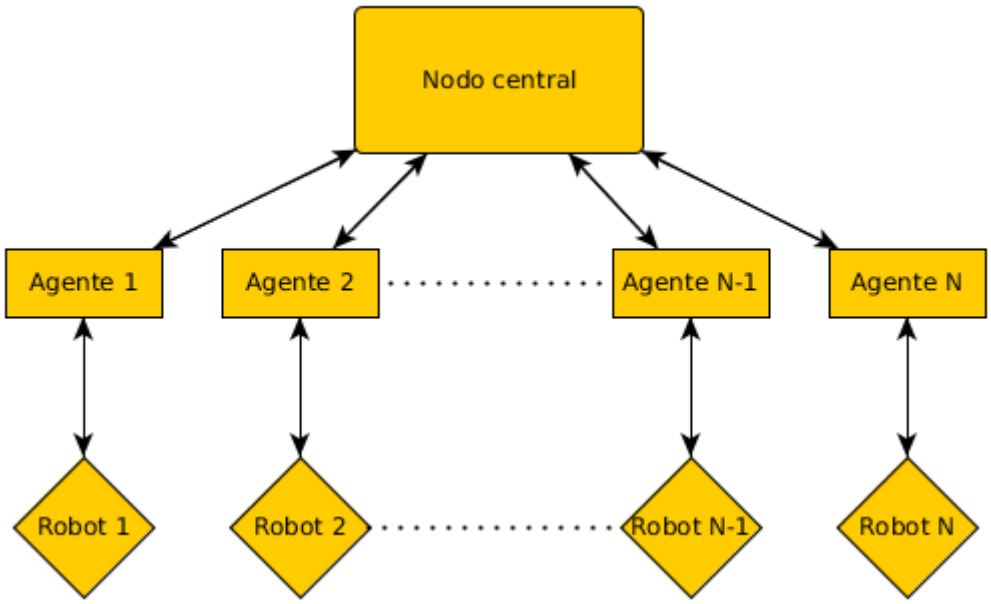
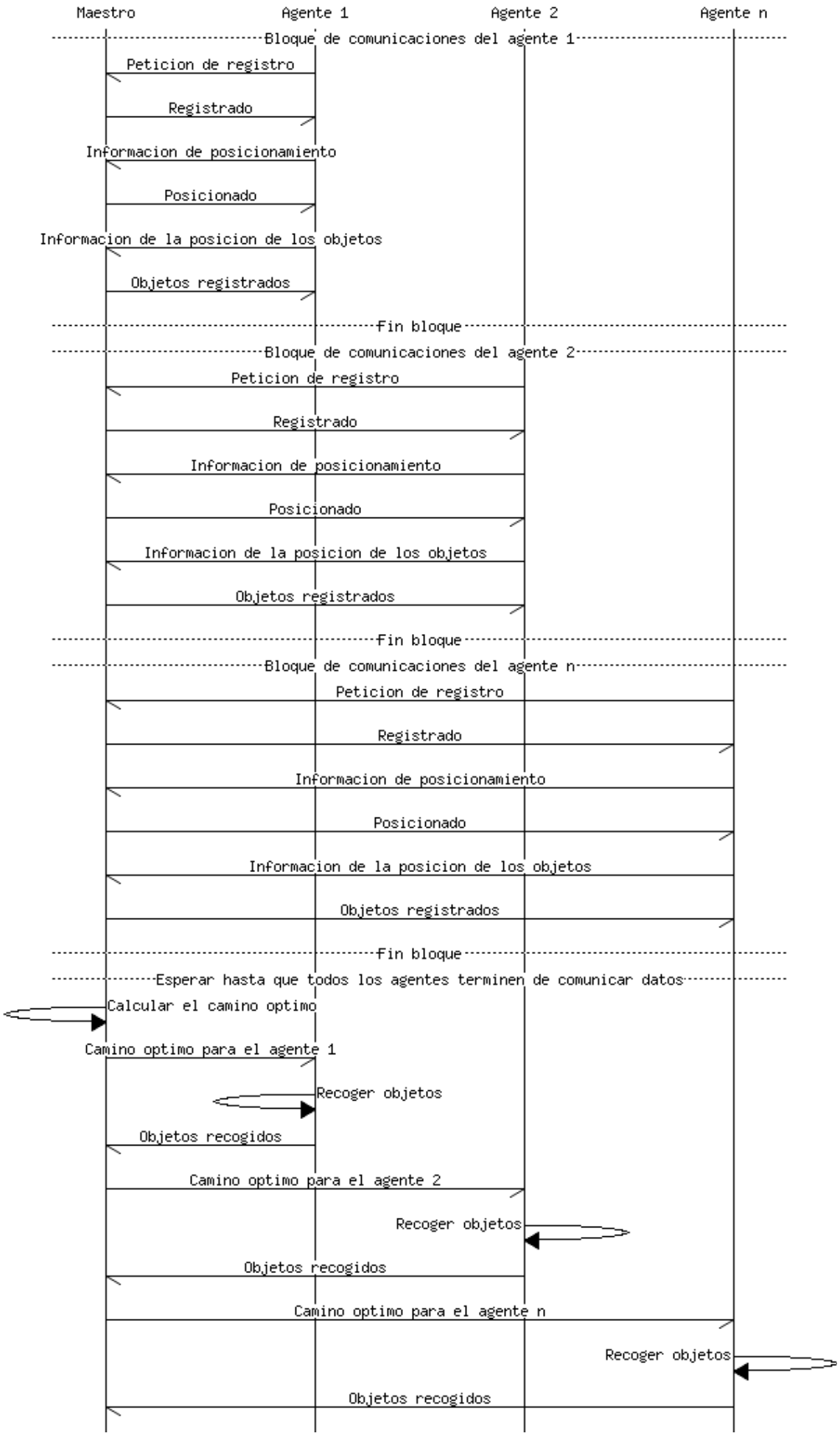


Ilustración 5: Esquema de comunicación centralizada

Las comunicaciones incluyen series de mensajes que ayudarán al nodo central a recopilar el conocimiento necesario para lanzar el algoritmo de planificación y distribuir el trabajo. Como los mensajes son cuantiosos y su información es muy dispar, se incluye un diagrama de mensajes que trata de explicar dichos intercambios:


Maestro
Agente 1
Agente 2
Agente n
Bloque de comunicaciones del agente 1
Peticion de registro
Registrado
Informacion de posicionamiento
Posicionado
Informacion de la posicion de los objetos
Objetos registrados
Fin bloque
Bloque de comunicaciones del agente 2
Peticion de registro
Registrado
Informacion de posicionamiento
Posicionado
Informacion de la posicion de los objetos
Objetos registrados
Fin bloque
Bloque de comunicaciones del agente n
Peticion de registro
Registrado
Informacion de posicionamiento
Posicionado
Informacion de la posicion de los objetos
Objetos registrados
Fin bloque
Esperar hasta que todos los agentes terminen de comunicar datos
Calcular el camino optimo
Camino optimo para el agente 1
Recoger objetos
Objetos recogidos
Camino optimo para el agente 2
Recoger objetos
Objetos recogidos
Camino optimo para el agente n
Recoger objetos
Objetos recogidos

Como se observa en el diagrama, los agentes deben enviar mensajes de registro, posicionamiento e información sobre los objetos que conocen al nodo maestro. Esto se hace antes de distribuir cualquier trabajo y se esperará a que todos los agentes hayan comunicado sus datos, por tanto el nodo maestro conoce el número de agentes que se tienen que registrar de antemano.

Gracias a la plataforma SPADE se pueden distinguir el tipo de mensajes fácilmente utilizando identificadores de conversación. Por lo general los agentes lanzarán primero un mensaje de petición al nodo maestro para registrarse que contendrá el nombre del agente. Cuando el nodo central reciba dichos datos se pasará a enviar información sobre la posición del robot y los objetos de los cuales se conoce su posición. Esta información, al tener un formato conocido, se pasará en formato de cadena de texto.

El sistema tiene una peculiaridad a la hora de distribuir el trabajo y es que los datos relativos a la ruta de recogida de objetos sólo se transmiten al siguiente agente cuando el agente anterior ha terminado de recoger los objetos. Esto ha sido planteado de dicha forma para evaluar la precisión de los robots más fácilmente y evitar choques, a pesar de que dicho comportamiento no sea idóneo a la hora de hacer un modelo realista.

4.3.b Esquema descentralizado

El sistema de comunicaciones centralizado es bastante simple cuando los agentes sólo necesitan comunicarse con el nodo que distribuirá el trabajo. De esta forma reciben su lista de objetos y ejecutan las órdenes correspondientes sin depender de lo que ocurra con el resto de robots. Esta aproximación es simple pero carente de realismo en este tipo de casos ya que pueden ocurrir fallos o replanificaciones durante la ejecución de las acciones.

Gracias al enfoque descentralizado, las comunicaciones se realizarán independientemente de agentes maestros entre todos los agentes. Por ejemplo, cuando un robot sufra una avería el enfoque descentralizado ayuda a que el resto de agentes sepan donde está y eviten pasar por ahí. Esto es de especial ayuda cuando existen tareas que son excluyentes entre sí ya que si, por ejemplo, dos robots deben acceder al mismo objeto deberían ser capaces de coordinarse para no hacerlo al mismo tiempo.

Con esta idea en mente se desarrolló un prototipo del sistema descentralizado. A pesar de que la intención del desarrollo sea que el sistema nunca sea dependiente de un solo nodo, existe una discrepancia de la cual es difícil escapar y ese problema es que el algoritmo debe de ser llamado con información completa sobre los robots y los objetos.

Si se suma eso al hecho de que sólo se dispondrá de dos robots físicos a la hora de realizar pruebas, este prototipo de sistema descentralizado se resume en relocalizar el agente controlador a uno de los agentes de los robots. El objetivo de este cambio aparentemente menor es que los robots puedan comunicarse sin depender de un nodo central, siendo uno de ellos un agente que concentrará los datos para ejecutar el algoritmo. Una vez realizada esta operación, los agentes se comunicarán de la misma forma que lo harían en un sistema puramente descentralizado.

Los procesos de elección de nodos centrales suelen ser complejos y requieren una gran cantidad de planificación. Existen muchas opciones para realizar esta tarea: por orden de nombre, por menor carga de trabajo, por un sistema de votaciones, etc. Debido a esta complejidad y a que solo se dispondrá de dos unidades robot físicas, la elección de líder se hace por imposición dentro del propio código.

Se añade el siguiente diagrama que tratará de aclarar la jerarquía de agentes:

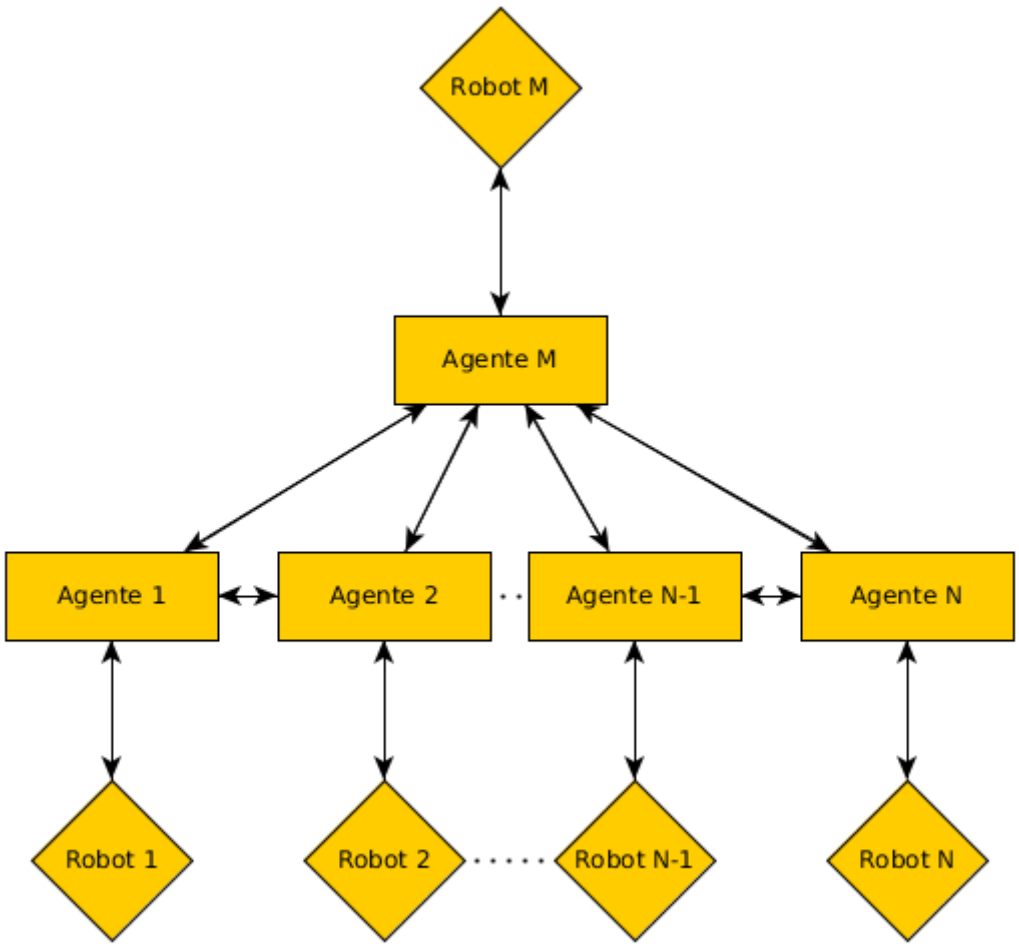
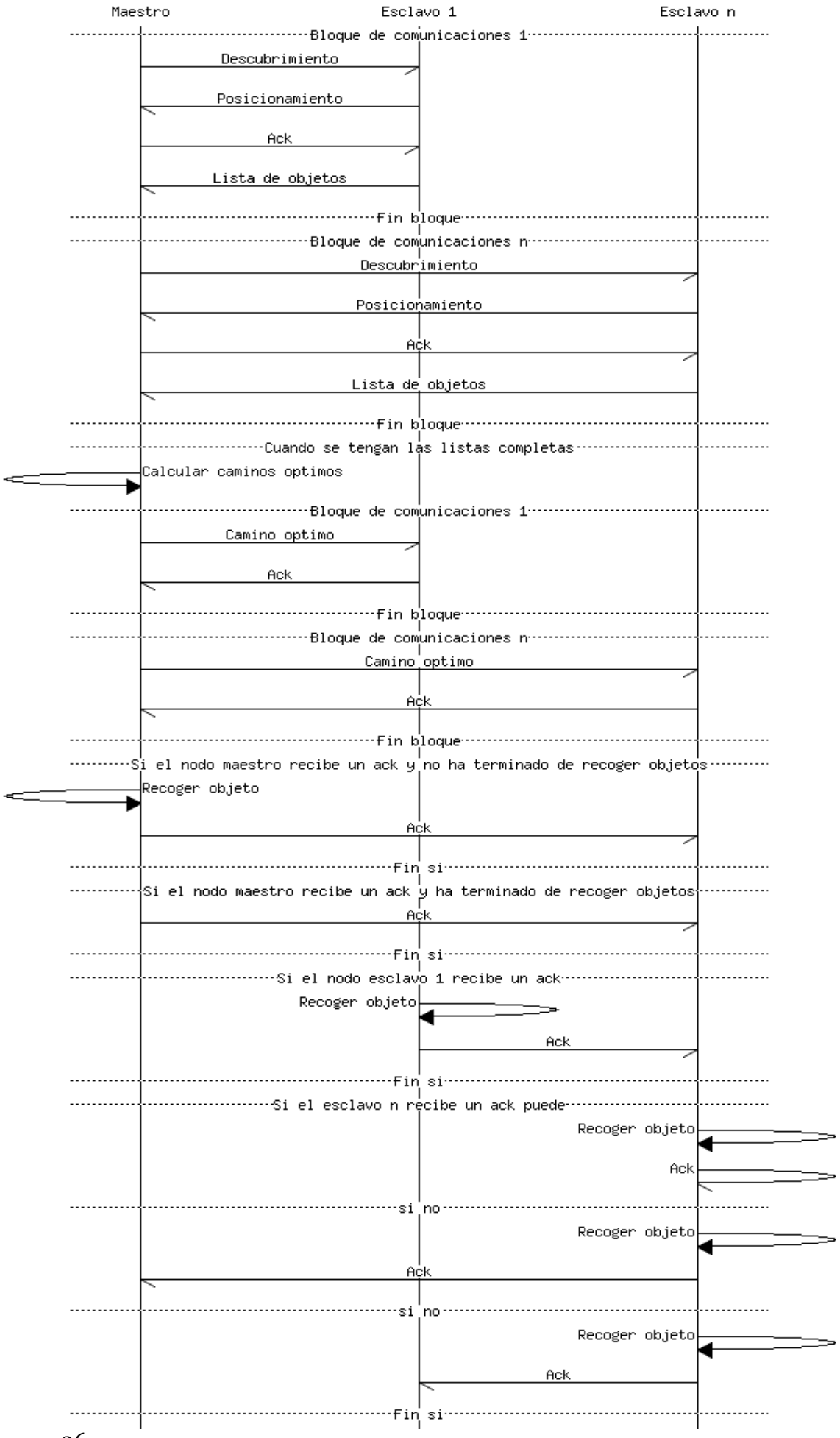


Ilustración 6: Esquema descentralizado

Como se puede observar, hay un nodo (en este caso el Agente M) que claramente se comunicará con todos a la hora de recibir información y distribuir el trabajo. Una vez hecho esto, los agentes se pasarán una serie de mensajes que determinarán cuando se deben de desplazar para recoger objetos y completar tareas. También se puede ver que los agentes que no son el maestro solo se comunicarán con sus agentes vecinos durante este periodo de tiempo para pasar dichos mensajes.

A continuación se presenta un diagrama que tratará de explicar los mensajes transmitidos a la hora de organizar las tareas:


Maestro
Esclavo 1
Esclavo n
Bloque de comunicaciones 1
Descubrimiento
Posicionamiento
Ack
Lista de objetos
Fin bloque
Bloque de comunicaciones n
Descubrimiento
Posicionamiento
Ack
Lista de objetos
Fin bloque
Cuando se tengan las listas completas
Calcular caminos optimos
Bloque de comunicaciones 1
Camino optimo
Ack
Fin bloque
Bloque de comunicaciones n
Camino optimo
Ack
Fin bloque
Si el nodo maestro recibe un ack y no ha terminado de recoger objetos
Recoger objeto
Ack
Fin si
Si el nodo maestro recibe un ack y ha terminado de recoger objetos
Ack
Fin si
Si el nodo esclavo 1 recibe un ack
Recoger objeto
Ack
Fin si
Si el esclavo n recibe un ack puede
Recoger objeto
Ack
si no
Recoger objeto
Ack
si no
Recoger objeto
Ack
Fin si

Tal y como ve puede apreciar en el diagrama de intercambio de mensajes dispondremos de un agente designado como maestro que mandará mensajes de descubrimiento a todos los agentes que tiene registrados. Se escogerá un nodo como maestro de forma forzada (sin que los agentes decidan cual debe de serlo) ya que solo se planea trabajar con dos robots y dicho intercambio de mensajes sería innecesario. También se supone que para el nodo maestro las direcciones del resto de agentes son conocidas.

Una vez el resto de agentes han recibido el mensaje inicial, enviarán sus datos al nodo maestro de una forma similar a cómo se vio en el esquema centralizado. Habiendo hecho esto, el agente maestro realizará los primeros movimientos necesarios para recoger sus objetos. Cuando haya terminado de hacerlo, enviará al siguiente agente un mensaje avisándole de que puede continuar sus acciones.

Esta última medida está hecha por dos motivos: el primero está determinado por las necesidades de evitar choques a la hora de realizar las pruebas con los robots físicos y el segundo es porque este tipo de comunicación puede ser muy útil en un entorno realista donde unas acciones se deban realizar a mitad de la ejecución de otras o quizás en substitución de las mismas.

4.4 Implementación del movimiento

A continuación se describe cómo fue posible implementar las órdenes de movimiento de los robots. Tras un análisis de la API de Nao se encontraron unas cuantos módulos que fueron útiles para implementar la recogida de objetos.

Dentro del paquete *ALMotionProxy* se pueden encontrar la gran mayoría de órdenes que mueven al robot. Sin embargo, la tarea de recoger objetos reales se antoja cercana a imposible ya que hay que coordinar articulaciones a nivel de movimiento matricial y hacer uso de la percepción del robot para ajustar el agarre. Esta parte se desestimará de la implementación, siendo el principal enfoque solo el desplazamiento.

La primera de las instrucciones que puede parecer la más simple es *move*, pero su utilización es incómoda y compleja: los argumentos que requiere son la velocidad de avance en el eje de las X, la velocidad de avance en el eje de las Y y

la velocidad de giro en el sentido de las agujas del reloj. Este tipo de instrucción sería bastante complicada de utilizar ya que dependería mucho del error de movimiento del robot y sería necesario hacer cálculos complejos para cada movimiento que el robot debe realizar.

La instrucción *moveTo* es mucho más cercana al funcionamiento que se buscaba para este tipo de acciones de movimiento ya que acepta como argumentos: desplazamiento en el eje X, desplazamiento en el eje Y y giro respecto a su posición inicial. Esta instrucción es bastante potente ya que simplemente requerirá calcular los desplazamientos en cada eje y el giro que el robot debería hacer para posicionarse con la orientación adecuada.

Por último, existe un conjunto de instrucciones llamadas cartesianas. A pesar de que este conjunto parezca el más adecuado para el sistema desarrollado resulta ser que su complejidad es incluso mayor que la de la instrucción *move*. Las instrucciones cartesianas utilizan matrices para posicionar y mover cada una de las articulaciones del robot mediante matrices de transformación. Este enfoque se descartó por razones obvias de dificultad.

Ya que el movimiento hizo uso de la instrucción *moveTo* cabe comentar que su uso fue sencillo pero requirió cálculos específicos, sobretodo para el giro del robot. Para ello se desarrolló un método que toma como argumentos las coordenadas actuales y las coordenadas objetivo del robot y calcula, utilizando combinaciones de arco-tangentes, el ángulo correcto que debe girar.

Con el punto inicial y el final se puede obtener el vector de desplazamiento. Comprobando el signo de cada coordenada del vector se debe aplicar una fórmula diferente para que el ángulo sea el correcto. También se debe considerar que cuando una de las coordenadas del vector es cero también se tendrán que realizar ciertas consideraciones para evitar excepciones matemáticas.

Se adjunta un pequeño fragmento de pseudocódigo que trata de dar una intuición sobre cómo funciona este método:

```
Función calcularAngulo(x1, y1, x2, y2):
 Si y1 == y2:
  Si x2 – x1 < 0:
    Devolver  π+angRel;
  Fin Si
  Devolver angRel;
 Fin Si
 Si x1 == x2:
  Si y2 – y1 < 0:
    Devolver – π/2+angRel;
  Fin Si
  Devolver π/2+angRel;
 Fin Si
 Si x2-x1 < 0:
  Si y2-y1 < 0:
    Devolver – π+atan((y2-y1)/(x2-x1))+angRel;
  Fin Si
  Devolver  π+atan((y2-y1)/(x2-x1))+angRel
 Fin Si
 Devolver atan((y2-y1)/(x2-x1)+cos(angRel))*angRel
Fin Función
```

En líneas generales, este método contempla una serie de casos específicos que han de ser resueltos mediante cálculos diferentes al convencional y los trata. Esta función recibe como argumentos la posición de salida y la posición objetivo del robot para devolver el ángulo que se debe girar.

Existe una constante en este fragmento que se denomina *angRel* que contiene la orientación del robot en radianes. Los robots físicos no siempre son inicializados en la misma posición ni apuntando al mismo punto cardinal, por tanto deben de ser considerados estos datos dentro del código. No es lo mismo

que el robot comience mirando al frente que de espaldas y estas cosas se tienen que considerar en el código.

Un pequeño apunte sobre la instrucción *moveTo* es que el tercer argumento no admite ángulo mayores a π o menores a -π. En caso de recibir un argumento de este tipo, el robot muestra comportamientos erráticos que pueden variar entre girar mal o directamente no realizar un giro.

Los resultados de la ejecución del algoritmo fueron, posiblemente, contrarios a la intuición de lo que esta instrucción debería hacer. Cuando se indican coordenadas en x e y el robot se moverá considerando dicho vector de desplazamiento. Cuando se especifica un ángulo de giro, el robot acabará su desplazamiento habiendo girado lo indicado. Esto da lugar al uso de esta instrucción de tres formas: especificando solo coordenadas del vector de desplazamiento, especificando solo giro y especificando los tres argumentos.

Aunque durante las pruebas virtuales se evaluaron los resultados de utilizar esta instrucción de maneras diferentes, se pueden adelantar las dos formas de utilizarlas que se probaron.

La primera opción que se utilizó fue la de hacer girar primero al robot y luego avanzar la distancia necesaria para llegar a la posición objetivo. De esta forma los robots realizan giros más precisos y se detecta con facilidad si el movimiento no es el esperado. En el siguiente fragmento de pseudocódigo se puede observar cuales son las instrucciones necesarias para recoger un objeto:

```
Alpha := calcularAngulo(robot.x, robot.y, objeto.x, objeto.y);
moveTo(0, 0, Alpha);
waitUntilMoveIsFinished();
despX := robot.x – objeto.x;
despY := robot.y – objeto.y;
moveTo(sqrt(despX*despX+despY*despY), 0, 0);
waitUntilMoveIsFinished();
```

Tal y como se puede comprender, primero se calcula el ángulo y se ordena al robot que gire. Una vez haya terminado de girar, se ordenará un desplazamiento en el eje X (en la dirección que encara el robot) siendo el desplazamiento el valor propio del vector de desplazamiento. La instrucción

waitUntilMoveIsFinished() sirve para comprobar si el movimiento que se ha ordenado ha terminado. Si no se usara esta instrucción, el robot recibiría varias instrucciones de movimiento antes de terminar el actual, dando lugar a comportamientos no deseados.

En caso de la segunda opción, el mecanismo de movimiento es mucho más sencillo, pero en la práctica resultó más impreciso y más difícil de comprobar cual había sido la fuente del error. Pese a todo, el movimiento en los ejes de coordenadas incorpora el giro, haciendo de esta forma de uso la más elegante y cercana a la realidad. El siguiente fragmento de pseudocódigo demuestra también cómo se recoge un objeto con este paradigma:

```
Alpha := calcularAngulo(robot.x, robot.y, objeto.x, objeto.y);
despX := robot.x – objeto.x;
despY := robot.y – objeto.y;
moveTo(despX, despY, alpha);
waitUntilMoveIsFinished();
```

Siguiendo con el resto de la codificación, un tipo de instrucción totalmente necesaria en este tipo de sistemas es una que obtenga la posición donde el robot cree que está. Esto es vital a la hora de corregir errores ya que el robot puede expresar cuanto considera que se ha desplazado y contrastar dichos datos con los esperados.

La API de Nao ofrece dos instrucciones que cumplen estas exigencias. El módulo *ALMotion* tiene la instrucción *getRobotPosition* que simplemente devuelve un vector con tres elementos, que son la posición en el eje X, en el eje Y y su giro respecto al eje X en radianes. Curiosamente, se puede encontrar una instrucción con el mismo nombre en el módulo *ALLocation* y devuelve valores en el mismo formato que la primera que se comentó.

Pese a lo que pueda parecer, las dos instrucciones no funcionan exactamente igual. Ambas instrucciones no devuelven los mismos valores cuando son llamadas una después de otra, pero los valores se diferencian bastante poco. Su uso también tiene unas cuantas diferencias que se comentan a continuación.

La instrucción *getRobotPosition* de *ALMotion* tiene un requisito para su funcionamiento y es especificar respecto a qué parte del cuerpo del robot se realizarán las mediciones. Esto es un dato importante ya que ciertas partes del cuerpo del robot no están en el centro del robot (piernas y brazos) y otras partes tienen orientaciones diferentes a la del resto del cuerpo (cabeza).

Para utilizar la instrucción que se encuentra en el módulo *ALLocation* es necesario usar antes la orden *learnHome,* cuya utilidad es estimar la posición inicial del robot para moverse acorde a ella. Para ello, el robot mueve la cabeza de una lado a otro y de arriba a abajo, luego gira, hace lo mismo y se vuelve a orientar en la posición en la que se quedó cuando la orden fue utilizada. Esta instrucción viene dada por la API y es difícil de saber qué valores obtiene durante la ejecución, aunque seguramente estén relacionados con el uso de balizas.

A la hora de la implementación final, la instrucción de posicionamiento de ALMotion fue elegida como la indicada. Esto se debe a que la del módulo *ALLocation* tarda un tiempo bastante largo en posicionarse y hizo que las pruebas se ralentizaren.

Por lo general, se hará uso de este tipo de instrucciones después de recoger un objeto y después de depositarlo en un contenedor para evaluar el error que había cometido el robot al desplazarse.

Finalmente, cabe comentar que los desplazamientos que realiza el robot físico no suelen ser precisos. Es bastante común que la ejecución termine con errores alrededor de dos centímetros. Este detalle es bastante importante ya que el método que determina el ángulo a girar depende en gran medida de la precisión del robot.

5 Pruebas y evaluación

5.1 Algoritmo

Una ventaja con la que que se contó a la hora de probar los resultados de la implementación es la buena modularidad incluso siendo un proyecto de poca envergadura. La parte más diferenciada del proyecto es el propio algoritmo de distribución de objetos, cosa que resulta bastante adecuada la hora de comprobar si sus resultados son los esperados o por el contrario el algoritmo incurre en algún tipo de error a la hora de ser ejecutado.

Las pruebas del algoritmo se realizaron tanto considerando los datos del problema general a tratar como muestras aleatorias que se le introdujeron al algoritmo para comprobar su eficiencia y corrección.

Las comprobaciones iniciales se hicieron con los datos de pruebas del sistema en general. Es una disposición de objetos bastante representativa y que comprueba de forma fácil si el algoritmo funciona en líneas generales. Este caso de prueba consta de dos robots y seis objetos. Si se visualiza como una cuadrícula, los dos robots están localizados en las esquinas diagonales opuestas. Los objetos se localizan de tal forma que cuatro de ellos están en posiciones equidistantes del contenedor y dos de ellos se encuentran más cerca de cada uno de los robots.

Tras ejecutar varias veces el algoritmo siempre obtenemos un resultado en el que las cajas cercanas a los robots son recogidas las primeras y las que se encuentran equidistantes al contenedor se recogen sin un orden en particular, pero de forma equitativa entre los dos robots.

En lo que al resto de pruebas se refiere, se distribuyeron aleatoriamente una cantidad de robots y objetos por la cuadrícula. Se limitó esa cantidad de objetos y robots a un máximo de cinco por motivos de claridad.

Uno de los mayores problemas para probar este enfoque es la incertidumbre de las soluciones al problema. En la mayoría de casos es difícil asegurar que una solución es la mejor posible en este tipo de problemas porque depende mucho de la distribución global de los objetos. Una solución puede

parecer bastante adecuada dados los datos, pero en realidad no serlo ya que coger un objeto más distante a la posición inicial del robot podría dar mejores resultados.

Para comprobar la corrección de los resultados, se realizaron 100 ejecuciones con datos aleatorios. Siendo que la función de puntuación es bastante clara (minimizar el desplazamiento que hace el robot que más se desplaza) se consideró difícil incurrir en errores, a pesar de todo muchas de las ejecuciones se comprobaron en papel.

En esta parte del desarrollo se encontró el error de la cota optimista sobre la segunda función de puntuación. Algunos casos de prueba (no muchos) daban valores ligeramente diferentes para el mismo caso de prueba, dando la función que no consideraba la cota mejores soluciones pero peores resultados a nivel de ejecución.

Ya que el algoritmo sigue una estructura medianamente simple, se creyó bastante oportuno localizar una serie de variables que dieran información sobre cómo se ha comportado el algoritmo durante su ejecución. Para ello se decidió contar el número de estados creados, estados que entraron en la cola, estados que salieron de la cola y el tamaño máximo de la cola.

En un primer lugar se compararán los resultados del algoritmo sin cota optimista, con ella y también el algoritmo con almacenamiento explícito de la puntuación de la mejor solución. La versión con cota optimista se incluye a pesar de que sus resultados no son óptimos, pero siguen siendo interesantes para versiones más rápidas que no requieran precisión perfecta. La tabla que sigue compara los resultados obtenidos:

	Sin cota	Con cota	Con Alm. Expl.
Iteraciones	1594	1078	1594
Max. Tam. Cola	10479	7168	1949
Extracciones	1595	1079	1595
Creaciones	12075	8248	12075
Inserciones	12075	8248	3253
Tiempo(ms)	973	648	904

Ilustración 7: Comparación entre algoritmos

Como los datos indican, existe una mejora significativa usando una cota optimista muy simple. Las mejoras son cercanas a un 40% en todos los campos y vienen dadas gracias a la información adicional que añade considerar las cajas sin distribuir, guiando de esta forma la solución de forma más eficiente. Sin embargo, como ya se demostró en el apartado 4.2.

Por otro lado, la versión del algoritmo sin cota optimista y con almacenamiento explícito de la puntuación de la mejor solución hasta el momento hace claras mejoras respecto al algoritmo sin cota en el tamaño máximo de la cola de estados y en la cantidad de inserciones del 81,4% y del 73% respectivamente.

Utilizando ahora el último algoritmo comentado, se muestra un pequeño análisis de su comportamiento respecto a la talla del problema. Con diversas cantidades de objetos frente a diversas cantidades de robots, el problema devuelve un resultado óptimo con el número de extracciones que se indica para cada combinación:

	Un objeto	Dos objetos	Tres objetos	Cuatro objetos	Cinco objetos	Seis objetos
Un robot	2	3	10	40	198	1166
Dos robots	2	3	16	83	695	6508
Tres robots	2	4	14	127	998	9971
Cuatro robots	2	4	18	119	1458	15711
Cinco robots	2	4	20	166	1818	23123
Seis robots	2	4	26	203	3683	31503

Ilustración 8: Comportamiento del algoritmo frente a diferentes tallas del problema

La cantidad de extracciones se consideró una forma interesante de medir el problema ya que es cercana al número de iteraciones y proporcional al número de creaciones.

Como se puede observar en los resultados, el número de extracciones crece con la talla general del problema de forma exponencial. Mirando más a fondo los factores que condicionan el número de extracciones, se puede observar que el número de robots aumenta de forma menos significativa este valor que el número de objetos.

Es fácil ver que la mayor carga del algoritmo viene a la hora de obtener todas las combinaciones posibles de robots para cada objeto. Cuando la talla de los objetos sea grande, la carga del algoritmo también será grande.

Tras observar dichos datos y a vistas de que, salvo error, el algoritmo da soluciones óptimas se considera que la implementación del algoritmo ha sido un éxito. Esto en parte es debido a que su complejidad ha tenido que ser restringida por la necesidad de dedicar tiempo al resto de secciones, pero seguramente este algoritmo asiente las bases para dichas ampliaciones futuras.

No se puede ignorar el hecho de que, por ejemplo para un robot y varias cajas, se consideran más estados de los necesarios a pesar de que a simple vista se comprueba que no sería necesario. Esto se basa en una serie de casos base que el algoritmo no considera y que dentro del desarrollo del proyecto no se presentan habitualmente. De cualquier manera, el objetivo del desarrollo del algoritmo era prepararlo no solo para devolver resultados óptimos, si no para que sea sencillo añadirle más factores en el futuro.

5.2 Simulación

Tras realizar una serie de comprobaciones iniciales con los agentes para determinar si se comunicaban correctamente, las pruebas virtuales con el sistema completamente montado son ciértamente complejas.

Antes de pasar al entorno simulado, los movimientos del robot fueron comprobados con la salida standard. Cuando el robot fuera a realizar una acción, la consola imprimiría por pantalla cual sería dicha acción y su supuesto resultado. Este método se utilizó hasta que se pulieron la mayoría de errores en las primeras implementaciones.

Las pruebas con el robot virtual necesitaban tomar un matiz diferente. Hubiera sido de interés contar con un entorno simulado lo más realista posible, donde se pudieran colocar objetos y varios robots para poder comprobar con precisión los resultados del algoritmo. Este lujo estaba lejos de la realidad y las pruebas se realizaron en un entorno bastante más limitado.

El mayor problema viene de la imposibilidad de tener dos robots en el mismo entorno simulado. La herramienta *Choregraphe* está diseñada para dar órdenes simples a un robot real mediante conexión en red o para comprobar los resultados de scripts en un robot virtual.

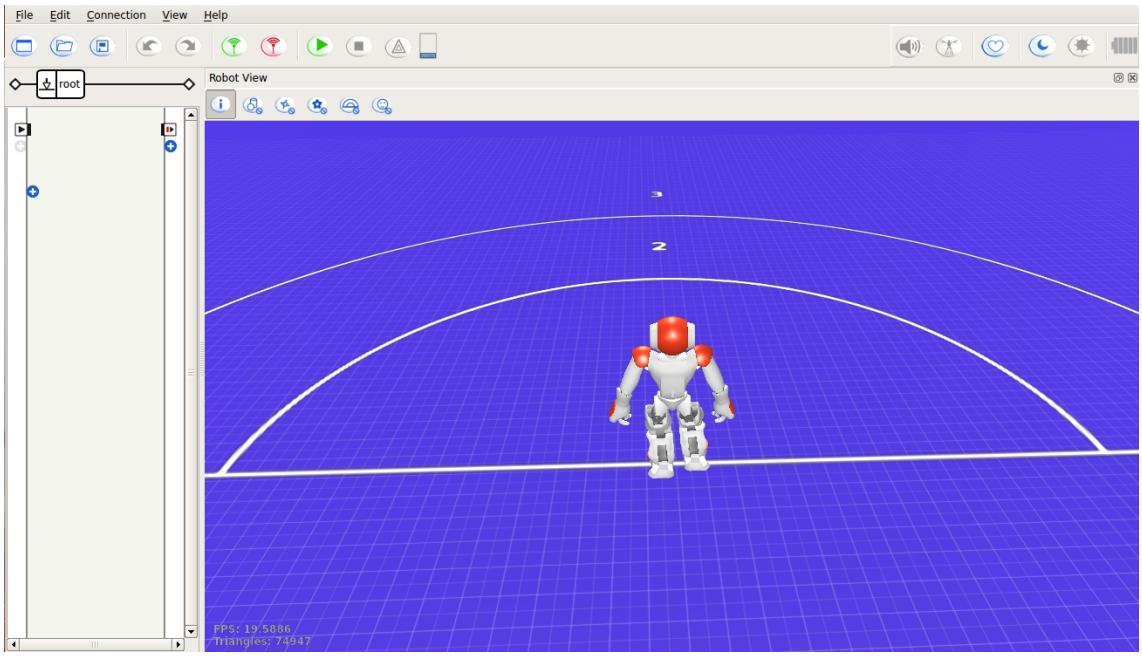


Ilustración 9: Robot Nao caminando en una simulación

Sin embargo los requisitos del proyecto hacían bastante atractiva la opción de poder simular dos robots. Esta carencia fue clave en el desarrollo ya que desencadenó la necesidad de tomar medidas complementarias.

Para hacer posibles estas pruebas se desarrollaron una serie de scripts complementarios que emulaban a un robot virtual. Dichos scripts eran totalmente necesarios para comprobar que las comunicaciones funcionaban como se pretendía y para utilizar un entorno de pruebas lo más cercano a la realidad.

Estos scripts simplemente integraban la funcionalidad completa de comunicación, pero no conectaban con ningún robot y, por tanto, no transmitían órdenes de movimiento. Para las versiones distribuidas del sistema fue necesario emular el tiempo que le costaría a un robot desplazarse con llamadas de tipo *sleep* que detienen la ejecución del script un determinado número de segundos.

Gracias a este limitado pero efectivo entorno de pruebas se pudieron localizar fácilmente problemas de implementación y comunicaciones para cortar en la medida de lo posible la cantidad de pruebas que se realizarían en los robots físicos tal y como se comentará en la siguiente sección.

Uno de los primeros problemas en emerger gracias a esto fue la falta de precisión que tienen los movimientos de los robots. Incluso en un entorno simulado, los robots no se desplazan exactamente la distancia que se les indica. Este error de posicionamiento en el mundo virtual se ve mermado en importancia comparado con los errores de posicionamiento en el mundo real que se verán en el siguiente apartado, pero dieron las primeras pautas a seguir a la hora de trabajar con resultados imprecisos.

También se solucionaron una gran cantidad de errores relacionados con los giros del robot y errores de desplazamiento dados por las instrucciones de la API de Nao. En ciertas ocasiones se tuvo que crear errores de forma intencionada para simular con más éxito el entorno real, haciendo de esta formas las pruebas virtuales mucho más robustas.

5.3 Pruebas físicas

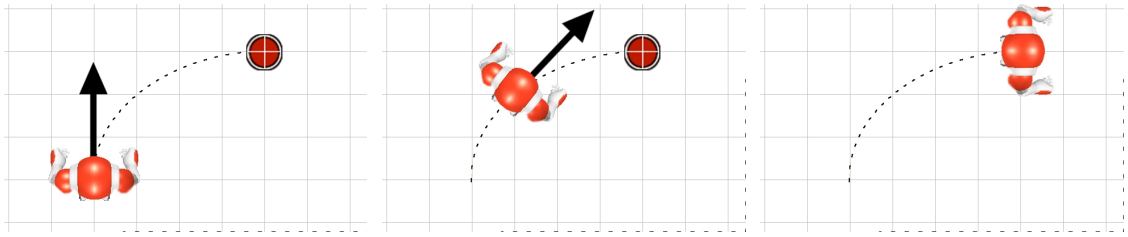
Las pruebas físicas que se realizaron utilizando a ambos robots físicos de los que dispone el departamento tenían su serie de peculiaridades. Al no poder probar los algoritmos instantáneamente en un robot físico, en la mayoría de casos las sesiones de pruebas se agotaban probando diversas variaciones del mismo problema. Al carecer de disponibilidad total, los resultados eran apuntados para ser corregidos para la siguiente sesión. De ese modo las pruebas sobre los robots físicos avanzaron.

El mayor problema a la hora de enfrentarse a pruebas físicas es sin duda la falta de precisión. Los robots *Nao* no giran de forma precisa y tampoco consiguen desplazarse con una precisión adecuada a las posiciones cartesianas que se le indican.

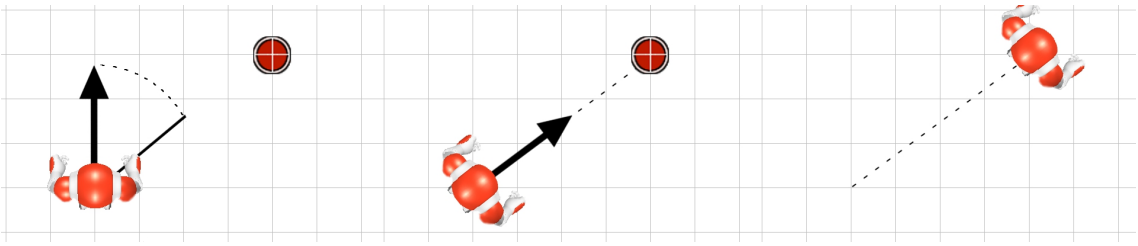
Cabe remarcar que los robots tienen una percepción de la realidad diferente entre sí. Muchas veces indicaban que estaban en la posición correcta cuando se habían desviado notablemente. Esto se pensó que puede deberse a las características físicas del propio robot, falta de percepción de la realidad (al no contar con balizas o cámaras) o a las características del suelo.

Estas dificultades hicieron que se tomaran líneas diferentes a la hora de usar las órdenes de las que dispone la API de Nao para los desplazamientos del robot. El cambio más importante que se introdujo fue respecto a la instrucción

moveTo. En un primer momento se utilizaban sus tres argumentos, el desplazamiento en ambos ejes y el giro a realizar. Esta forma de utilizar la instrucción daba lugar a desplazamientos más elegantes, pero con ciertos valores el robot hacía movimientos imprevistos.



Para subsanar dicho error se utilizó el *moveTo* de forma diferente, haciendo girar primero al robot y luego indicando que se desplazara solo hacia delante. Este conjunto de instrucciones conseguía hacer que el robot primero se girara hasta quedar frente a la dirección en la que se debía desplazar y después desplazarse. Este nuevo enfoque hizo que el movimiento fuera más preciso.



Posteriormente se descubrió que el desplazamiento en ambos ejes de coordenadas y giro incorporado se debía utilizar considerando el giro inicial del robot. El robot se desplaza con vectores relativos a la posición actual del robot por tanto, si no está orientado en el ángulo cero respecto al eje X, no se desplazará en las coordenadas absolutas, si no en las relativas al giro que tiene el propio robot cuando comienza el desplazamiento. Una vez se descubrió esto, el vector de desplazamiento se debió de modificar para realizar una transformación que corrigiera el vector.

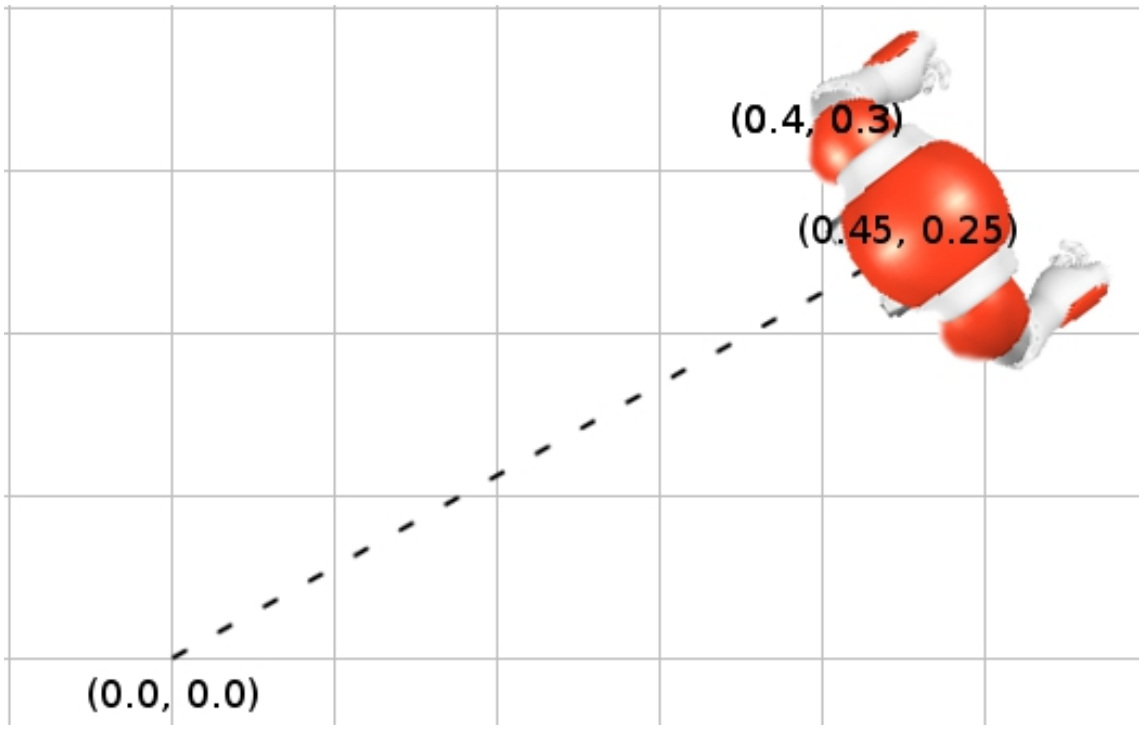
Sin embargo esto no fue suficiente. Por ejemplo, supongamos que se le indica al robot que debe desplazarse (1.0, 1.0) o, lo que es lo mismo, desplazarse un metro en el eje X y un metro en el eje Y. Una vez se lanza dicha instrucción el robot puede quedar, probablemente, fuera de dicha posición.

Supongamos que esa posición errónea sea (0.9, 1.1). Una vez sabemos esto, si se le pregunta al robot la posición que tiene, la respuesta puede variar. En diversas ocasiones el robot considera que se ha desplazado perfectamente y, dentro de nuestro ejemplo, el robot diría que se encuentra en la posición (1.0, 1.0) a pesar de que en la realidad esto no sea cierto.

También es posible que al llamar a la instrucción *getRobotPosition* su resultado sea también una posición errónea. A partir de ahora se hablará del error que comete el robot en el entorno físico como error real y del error que el robot considera que ha cometido como error virtual.

Solucionar el error real se descartó por su enorme complejidad. En general, las aplicaciones que requieren el uso preciso de robots utilizan mecanismos de percepción para ver el objetivo y desplazarse hacía el, corrigiendo los errores que pueden surgir a medida que el propio robot percibe desvíos en su trayectoria.

También se hace uso de cámaras cenitales que pueden ayudar al robot a orientarse. Al no ser objetivo del trabajo este tipo de implementaciones, se decidió trabajar alrededor de este problema en lugar de dedicar recursos a solucionarlo. Para dar una idea de cómo son estos errores, se incluye el siguiente diagrama:



• Posición Inicial: (0.0, 0.0) • Posición Objetivo: (0.4, 0.3) • Posición Final: (0.45, 0.25) • Posición percibida: (0.47, 0.32)	• Error real: (0.05, 0.05) • Error virtual: (0.02, 0.07)

El error virtual es más fácilmente subsanable. Este error se podría haber ignorado, ya que en la mayoría de casos no contiene información relevante sobre la posición del robot, pero se consideró que corregirlo conseguiría ayudar a que el siguiente movimiento fuera más preciso. Para ello se creó un sencillo bucle de reposicionamiento para que el propio robot se desplace el espacio necesario para acabar en la posición que él considera acertada.

Por último, el sistema tiene la necesidad de prevenir choques entre los robots cuando recogen objetos. Esto es un factor muy importante ya que cuando los robots chocan pierden la información sobre la posición en la que se encuentran. Existía una necesidad imperativa de evitar choques entre robots ya que creaba errores en el resto de las pruebas.

Se evitaron los choques implementando un sistema de turnos entre los robots de forma que, cuando un robot fuera a dejar un objeto en el contenedor, el otro robot debe de estar fuera del contenedor. Esto evitó la necesidad de implementar la percepción del robot para evitar choques entre robots y el coste de replanificar el movimiento.

De esta forma se hizo fácil comprobar que los robots se mueven de la forma esperada sin tener que reiniciar el experimento cada vez que los robots colisionan. Pese a todo, los robots pueden llegar a colisionar si el vector de desplazamiento coincide con la posición que ocupa otro robot.

6 Conclusiones

Se planteó desarrollar un sistema que, dados unos objetos, sus posiciones y un contenedor, hiciera uso de un algoritmo de planificación para distribuir el trabajo entre varios robots. Una vez las tareas fueran distribuidas, se debía utilizar un sistema de comunicación entre agentes inteligentes que transmitiera el resultado del algoritmo a cada agente controlador de un robot. Dicho agente utilizaría esta información para ordenar al robot los movimientos que debía realizar, además de comprobar en qué posición acababa el robot tras ejecutar cada acción.

La parte del desarrollo que corresponde al algoritmo de planificación se desarrolló con éxito. Se creó un algoritmo de ramificación y poda que obtiene el orden de recogida de objetos que se complete en el menor tiempo global posible. Además, el algoritmo tiene integradas ciertas mejoras que agilizan el tiempo de cómputo sin hacer que la solución deje de ser óptima.

El sistema de comunicación entre agentes fue desarrollado con éxito. Los agentes son capaces de registrarse en el nodo central y enviar la información necesaria para planificar la recogida de objetos. De la misma forma, existen canales de comunicación para transmitir los resultados de la ejecución del algoritmo. También son capaces de mantener un orden de turnos en el cual los robots intercalan sus acciones y esperan a que terminen las acciones del resto para moverse.

El conjunto de órdenes que permite que los robots se muevan se exploraron e implementaron con un buen nivel de eficacia dadas las restricciones. El sistema hace que los robots se desplacen con una precisión aceptable, que alcancen las posiciones de los objetos que deben recoger, se desplacen al contenedor y vuelvan a su posición inicial una vez terminado el trabajo. Como contenido adicional, se añadió un sistema de reposicionamiento que permite mover al robot cuando se considera que no está en la posición indicada para tratar de alcanzarla.

Los puntos planteados en los objetos del proyecto han sido cumplidos, además de añadir diversos contenidos adicionales. Por esta razón se considera que el proyecto en conjunto ha sido realizado con éxito

6.1 Trabajo futuro

El algoritmo de planificación puede ser sujeto a diversas mejoras que añadirían funcionalidades adicionales o mejorarían las actuales:

- Aceptar nuevos objetos en tiempo real y replanificar las órdenes.

El hecho de volver a planificar las listas de objetos a recoger requiere volver a lanzar el algoritmo de planificación con información adicional. En un primer lugar se deberá evaluar la solución de la que ya se dispone para poder usarla de mecanismo de poda. En segundo lugar, utilizar partes de la solución de la que ya se dispone para agilizar el proceso de creación de un nuevo estado. En tercer y último lugar se deberá añadir también el estado inicial para garantizar que todas las opciones son consideradas.

Además de lo comentado, habría que adaptar los mecanismos de comunicación de los agentes para aceptar a tiempo real modificaciones en las listas. También tendría que añadirse algún tipo de interfaz para el usuario y permitir que dichos datos se transmitan al nodo central para lanzar de nuevo el algoritmo.

- Controlar si un robot ha quedado inutilizado y replanificar las órdenes.

Es poco realista suponer que el robot jamás será detenido bajo ningún motivo una vez haya comenzado a desplazarse. Una vez pase esto, habría que disponer de algún mecanismo para recuperarse del error, considerar si es necesario replanificar y hacerlo en el caso que lo sea. Además de eso, habría que habilitar mecanismos para comprobar si el robot sigue indispuesto y qué instrucciones no ha podido ejecutar.

- Prevenir choques entre robots calculando el cruce de trayectorias.

Mientras que evitar los choques es una medida totalmente necesaria, se puede también hablar de prevención de choques. A la hora de planificar se podría calcular si dentro de la ruta que se está analizando existen choques entre sus vectores de desplazamiento. Cuando se detecten estos casos se podría penalizar los choques en la puntuación del estado para que se traten de evitar en la práctica.

- Añadir valores adicionales al robot que condicionen los resultados (nivel de batería, mejores capacidades para realizar ciertas tareas, etc).

El proyecto está planificado para que se consideren datos adicionales del robot de forma sencilla. Sería necesario especificar dichos campos dentro de la clase y crear una función que considerara los valores que tuviera cada robot. Pese a que esto parece sencillo, existe un problema significativo y viene a la hora de comprobar la corrección del algoritmo. Cuando se añaden factores como la batería al cálculo del camino se hace considerablemente más complicado comprobar si el resultado es el más indicado.

- Añadir la posibilidad de que exista más de un contenedor para depositar los objetos.

Como se ha podido comprobar en las descripciones del algoritmo, el desarrollo se centra principalmente en el uso de un solo contenedor. En un principio, la infraestructura del algoritmo de planificación debe de ser capaz de controlar sistemas con más de un contenedor modificando la función de puntuación. Una vez más, esta es una modificación que hace complejo evaluar si los resultados son los adecuados.

También hay varios detalles que se pueden perfilar en el conjunto de instrucciones que controlan el movimiento del robot:

- Hacer que el robot realmente coja objetos y los deposite.

Uno de los elementos más discutibles en el desarrollo del proyecto probablemente sea que el robot no recoge y deposita objetos físicamente, si no que se mueve a las posiciones que estos ocupan y no hacen nada más. Pese a que sería muy deseable contar con esta funcionalidad, existe una gran dificultad en conseguir que el robot perciba objetos, haga los desplazamientos necesarios para recogerlo y consiga depositarlo. Existen instrucciones que pueden controlar estos movimientos, pero posiblemente esto diera para un nuevo proyecto entero.

- Hacer uso de balizas y/o mecanismos externos de posicionamiento.

Se puede afirmar de forma general que la precisión de estos robots es mala si no se cuenta con algún sistema externo de posicionamiento para el robot. Las balizas son consideradas de ayuda en estos casos, ya que el robot teóricamente es capaz de orientarse usándolas. También es considerable pero mucho más complejo usar una cámara cenital para que el propio robot determine su posición.

7 Referencias

[1]Cyberdyne. Empresa de desarrollo robótico. Fecha de consulta: 30 junio 2015. Disponible en: http://www.cyberdyne.jp/english/products/HAL/

[2]Motherboard. Web de noticias científicas online. Fecha de consulta: 30 junio 2015. Disponible en: http://motherboard.vice.com/read/robots-are-caring-for-elderly-people-in-europe

[3]Igwe KC, Onyenweaku CE, Nwaru JC. Application of Linear Programming to SemiCommercial Arable and Fishery Enterprises in Abia State, Nigeria. Management Journals. International Journal of Economics and Management Sciences (IJEMS). 2011;1(1):75-81. Disponible en: www.managementjournals.org

[4]Cuni, G., Esteva, M., Garcia, P., Puertas, E., Sierra, C., & Solchaga, T. MASFIT: Multi-Agent System for FIsh Trading. Disponible en: http://www.iiia.csic.es/files/pdfs/1182.pdf

[5]SPADE. Python Package Index, sitio oficial de consulta de paquetes de Python. Disponible en: https://pypi.python.org/pypi/SPADE

8 Anexos

Repositorio de libre acceso con el código desarrollado durante el proyecto: https://github.com/Pazaak/NaoPickUp