Universitat de les Illes Balears

escola politècnica superior

Treball Final de Grau

GRAU D'ENGINYERIA ELECTRÒNICA INDUSTRIAL I AUTOMÀTICA

Implementación de mecanismos de asignación de tareas basado en subastas para sistemas multi-robot

DAVID GARCÍA ORTIZ

Tutor
José Guerrero Sastre

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, 6 de septiembre de 2017

ÍNDICE GENERAL

ACRÓNIMOS

ROS Robot Operating System

SA Sense and Act

TCP Transmisión Control Protocol

IP Internet Protocol

SDC Sistemas De Coordenadas

UDP User Datagram Protocol

RESUMEN

Este proyecto tiene como objetivo principal la implementación de un mecanismo de asignación de tareas basado en subastas para sistemas multi-robot. Dicho mecanismo se basa en la negociación entre robots, los cuales pujarán por las tareas para conseguir que se les asigne. Estas tareas consisten en unas coordenadas objetivo que los robots deben alcanzar. Por un lado, se necesita un sistema de comunicación para implementar el algoritmo de subastas y, por otro lado, se necesita un mecanismo de navegación para alcanzar las coordenadas objetivo.

La asignación de tareas basadas en subastas se caracteriza por dos figuras: subastadores y pujadores. A cada tarea se le asignará, mediante el algoritmo de elección de líder implementado, un único líder. Los líderes actuarán como subastadores y su función es decidir que pujadores (no líderes) pueden apoyarle en la ejecución de su tarea. Para poder participar en una tarea los pujadores realizan una oferta, conocida en subastas como puja, al subastador (líder) de dicha tarea y éste seleccionará qué robots le apoyarán en función del valor de la puja.

Primero se ha implementado un algoritmo de evitación de obstáculos por campos de potencial permitiendo al robot alcanzar las coordenadas de forma segura. Posteriormente, se ha desarrollado los mecanismos de comunicación, los cuales están basados en *sockets*, para poder implementar los algoritmos de asignación de tareas basados en subastas.

El programa se ha realizado utilizando el lenguaje C++. Como nexo de comunicación se ha utilizado *Robotics Operating System* (ROS) que consiste en un *framework* que hace la función de interfaz interactiva entre *software* y *hardware*. El simulador utilizado para la realización de las pruebas ha sido Stage que consiste en una plataforma de simulación en dos dimensiones capaz de cargar mapas y especializada en robótica móvil.

Las primeras pruebas se realizaron sobre el simulador Stage. Tras el éxito de estas pruebas se procedió a validarlas sobre el robot real Pioneer 3DX disponible en el laboratorio de robótica.

CAPÍTULO 1

INTRODUCCIÓN

1.1 Motivación y contextualización

Uno de los campos que más avances está experimentando es el ámbito de la robótica. Se tienen muchas expectativas en ella y cada día se integra en más áreas. La robótica puede encontrarse desde en máquinas cotidianas y sencillas hasta en las que poseen la tecnología más puntera del momento. Esto se debe a que algunas de sus cualidades son inalcanzables para el propio ser humano. Los robots pueden ofrecer algunas virtudes muy superiores: fuerza, velocidad, no sufren cansancio, memoria, precisión, etc. La robótica móvil se ha convertido en un campo muy versátil y con un gran futuro en la actualidad. Se está integrando en sectores como el transporte, la construcción, navegación espacial, agricultura o medicina.

Este trabajo se centra en los llamados sistemas multi-robot donde varios robots trabajan bajo un objetivo común. El trabajo en equipo permite que se pueda llevar a cabo tareas en un periodo de tiempo más reducido o que se pueda realizar misiones, las cuales serían imposibles para un robot sólo. En sistemas multi-robot surge una problemática inexistente en sistemas de un único robot: la asignación de tareas. Al haber varios robots trabajando a la vez emerge la complejidad de la interacción de comportamientos simples en cada robot y se tiene que implementar de tal forma que colaboren y no interfieran en sus respectivas tareas.

Esta colaboración puede ser de dos tipos: robótica de enjambres (*swarm*) donde no se necesita necesariamente una comunicación entre robots porque cada uno tiene su tarea y no interfiere directamente en la de los demás. La otra es la denominada robótica basada en subastas (*auction*) donde es imprescindible la comunicación multi-robot debido a que la asignación de tareas se negocia entre ellos mismos. Los mecanismos de subastas se basan en un robot o varios que actúan como subastadores y el resto que hacen de pujadores. Los pujadores envían una oferta, conocida como puja, a los subastadores para solicitarles participar en una tarea. Los subastadores en función del valor de la puja deciden si permitirle participar o no. El valor de la puja depende de los intereses y requisitos que impongan los subastadores. El mecanismo implementado en

este proyecto encargado de la asignación de las tareas multi-robot se basa en algoritmos de subastas (*auction*) propuestos inicialmente por el tutor de este proyecto [1].

Este proyecto se basaba en el entorno Player/Stage, entorno que ya no tiene soporte actualmente. Por ello, surge la necesidad de adecuar los algoritmos a nuevos entornos de desarrollo cómo es *Robotics Operating System* (ROS).

ROS se trata de un entorno de desarrollo que actúa como interfaz de control haciendo de nexo entre el *software* implementado y el *hardware* del robot. El simulador usado en este proyecto para la realización de las pruebas es el Simulador Stage. Éste funciona en dos dimensiones y se caracteriza por su flexibilidad de creación de mapas. El robot utilizado es el Pioneer 3DX. Éste es un robot muy común en prácticas universitarias gracias a su diseño y a la multitud de funciones que ofrece para sus dimensiones y su *hardware*.

1.2 Objetivos

El objetivo principal de este proyecto es la implementación de un mecanismo de asignación de tareas basado en subastas para sistemas multi-robot adaptado al entorno de trabajo ROS. Estas tareas consisten en unas coordenadas que los robots tienen que alcanzar. El mecanismo de asignación de tareas basado en subastas se rige por la negociación entre robots donde éstos pueden ejecutar una de estas dos funciones: ser subastador o pujador. Cada tarea tendrá un subastador, el cual decidirá que pujadores pueden participar en cada tarea en función de la oferta, conocida como puja, que le hagan éstos. El valor de la puja depende de la cercanía del robot pujador a las coordenadas objetivo.

Para ello, se han tenido que alcanzar los siguientes subobjetivos:

- Implementación algoritmo evitación de obstáculos para navegar hasta las coordenadas objetivo.
- Sistema de comunicación basado en *sockets* para la asignación de tareas por subastas.
- Implementación programa monitor para el envío de las coordenadas de las tareas a los robots.

1.3 Estructura

Esta memoria se encuentra formada por siete capítulos y un anexo con todo el código desarrollado. El actual, capítulo 1, se trata de la introducción. En el siguiente, capítulo 2, se explica todo el *hardware* del Pioneer 3DX. En el capítulo 3 se desarrolla todo el funcionamiento del sistema ROS y del simulador Stage. Como continuación se encuentra el capítulo 4 donde se exponen los algoritmos implementados para explicarlos detalladamente en el capítulo 5. En el capítulo 6 se explican las pruebas realizadas en Stage y en el robot real. Por último, este proyecto finaliza con el capítulo 7 de conclusión y se propone una serie de mejoras y propuestas para futuros trabajos.

CAPÍTULO 2

INFRAESTRUCTURA HARDWARE

En este capítulo se tratan todas aquellas características físicas y componentes del robot, Pioneer 3DX, que se utilizará durante este proyecto.

2.1 Pioneer 3DX

El robot utilizado es el Pioneer 3DX (figura 2.1), el cual consiste en una base robótica diseñada por la compañía MobileRobots Inc. (ActivMedia Robotics). Se trata en un robot pensado para interior. Su estructura consta de tres ruedas, sensores de ultrasonidos, codificadores de rueda, sistema de alimentación basado en baterías, motores y un microcontrolador del *firmware* de AROS y el Pioneer SDK como paquete de desarrollo de *software*. Estos componentes son los que vienen integrados en el modelo básico, pero se le pueden añadir una gran variedad de sensores y dispositivos [2].

2.1.1 Tracción

Este tipo de base robótica no está pensada para trabajar en terreno irregular, sino para uno uniforme, preferiblemente zonas interiores o cubiertas. Es decir, no está diseñado para soportar las condiciones ambientales adversas, ni para terrenos que no hayan sido adaptados previamente. El robot tiene tres ruedas, dos de las cuales son motrices y la otra es de giro libre (rueda loca). El diámetro de las ruedas motrices es de 19 centímetros y la de giro libre es de 6 centímetros. Todas las ruedas son lisas y de neumático blando, ya que al no tener que circular por terreno irregular no necesita características de ruedas todoterreno. Las ruedas motrices son de tracción independiente y se encuentran a cada lado del robot y la rueda de giro libre se encuentra en la parte posterior. Debe de existir la tracción independiente porque el método de giro se basa en éste. Al funcionar cada rueda motriz por separado se puede conseguir direccionar del robot con solo dos ruedas, sin necesidad de que el robot posea control de dirección [2].

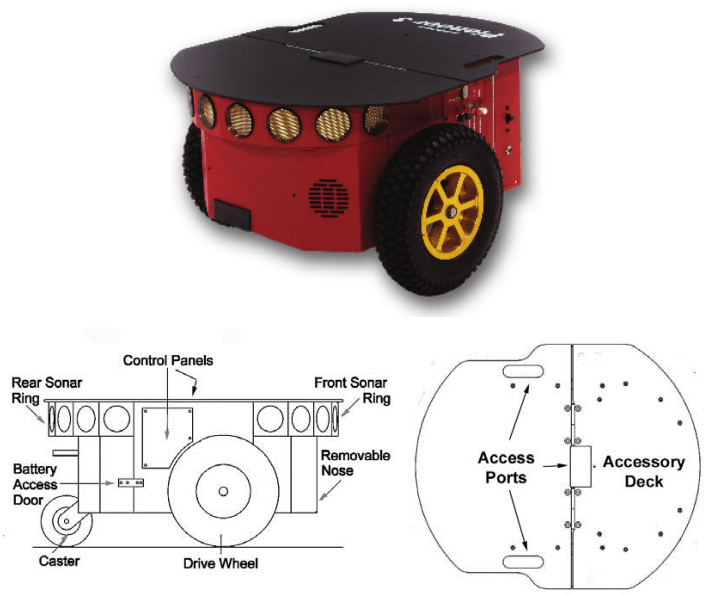

Figura 2.1: Robot Pioneer 3DX [2].

2.1.2 Sensores de ultrasonidos

El robot dispone de 16 sensores de ultrasonidos de 25 Hz de frecuencia de lectura: ocho de ellos en la parte frontal y ocho en la parte trasera. Estos sensores están colocados estratégicamente formando una especie de anillo, con el objetivo de que no existan ángulos muertos y sea capaz de detectar cualquier obstáculo próximo manteniendo al robot siempre consciente de los 360 grados de su entorno [2].

2.1.3 Posicionamiento

El posicionamiento del Pioneer 3DX se rige mediante la aplicación de odometría. Para ello, dispone de encoders angulares situados en los ejes de las ruedas motrices. La odometría consiste en determinar la posición de un vehículo a partir de información sobre la rotación de las ruedas.

Los encoders son sensores que pueden medir las revoluciones que se producen en un eje. Para conocer esta rotación estos sensores tienen una resolución determinada, la cual se mide en pulsos/revolución. Por lo tanto, conociendo cuantos pulsos ha rotado el eje puede saberse el ángulo rotado. Por ejemplo, si el encoder corresponde a una precisión de 360 pulsos/revolución la resolución sería de 1 grado. Es decir, cada pulso que avance será un grado rotado. Las características de los encoders del Pioneer 3DX pueden observarse en la tabla 2.1 [3].

El robot considera que se encuentra en las coordenadas (0,0) cuando se enciende y a partir de ese momento controla su desplazamiento aplicando la odometría gracias a la información obtenida de los encoders. Para ello, es necesario determinar el desplazamiento lineal y angular. Como en las características del robot ya se encuentra la conversión de pulsos/revolución a pulsos/mm es sencillo saber el avance a través

Resolución (pulsos/rev)	500
Resolución (pulsos/mm)	76
Relación de transmisión	19.7

Cuadro 2.1: Características Encoders.

de los pulsos y poder determinar las coordenadas del robot tomando su punto inicial como la posición (0,0).

2.1.4 Sistema de alimentación

El factor alimentación siempre es un punto muy importante en cualquier dispositivo que no esté conectado permanentemente a una fuente de energía estable. Por ello, se necesitan las mejores prestaciones energéticas manteniendo óptimas las condiciones físicas.

El Pioneer 3DX obtiene su alimentación de 2 baterías de plomo de 3 kg cada una. Éstas proporcionan hasta 252 W/h y una autonomía de 2 horas. Aportan un peso considerable teniendo en cuenta las dimensiones del robot, pero una de las características de este tipo de robots es que pueden soportar una carga considerable, en relación a su tamaño, sin afectar significativamente a su eficiencia, ni a su funcionamiento.

2.1.5 Motores

Los motores juegan un papel imprescindible en cualquier robot móvil. Este papel es más destacable en los Pioneer 3DX, ya que no solo el sentido de avance del robot depende de ellos, sino, también, su dirección. Los motores utilizados son de corriente continua. Hay un motor conectado a cada rueda motriz encargándose de ofrecer la tracción y la dirección al robot.

2.1.6 Conexión ordenador-Pioneer

La conexión del ordenador al Pioneer 3DX se realiza mediante puerto serie y un entorno *software* específico. Este entorno se basa en ARIA/AROS. Aria lleva a cabo las funciones de arquitectura de control. Consiste en un entorno de código abierto en C++, cuyo fin es dar robustez a la interfaz de cliente dispuesta por el programa del usuario. Aria trabaja ligado al entorno de trabajo *Robotics Operating System* (ROS) en la interacción a bajo nivel con el *hardware*. Para ello, utiliza un nodo de ROS, llamado ROSAria. Para comunicarse con el firmware AROS del microcontrolador Hitachi H8S/2357 del Pioneer se utiliza el bus RS-232 y un protocolo de comunicaciones Transmisión Control Protocol (TCP)-Internet Protocol (IP). En la figura 2.2 se aprecia como funciona la comunicación PC-Pioneer [4].

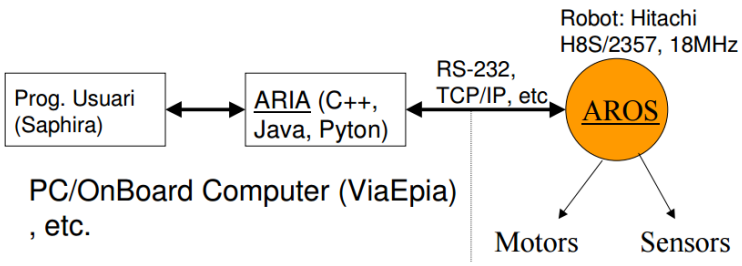

Figura 2.2: ARIA/AROS [4]

CAPÍTULO 3

INFRAESTRUCTURA SOFTWARE: ROS Y STAGE

En este capítulo se explican los elementos de ROS y del simulador Stage, cómo funcionan, por qué se ha optado por ellos.

3.1 Robotics Operating System

ROS son las siglas de *Robotics Operating System*. Fue desarrollado en 2007 por el Laboratorio de Inteligencia Artificial de Stanford. En 2008 se trasladó al instituto de investigación Willow Garage con la colaboración de más de 20 instituciones y dejó de ser un proyecto experimental pasando a ser un proyecto que empezaba a consolidarse gracias al equipo experimentado que se hizo cargo de él. Esto permitió que fuese creciendo exponencialmente dando lugar a la primera versión estable y estandarizada en 2009. ROS es de código libre y se encuentra bajo los términos de licencia BSD (Berkeley Software Distribution). *Robotics Operating System* (ROS) se trata de un entorno de trabajo que provee librerías y diversas herramientas con el fin de proporcionar la ayuda que necesitan los programadores del campo de la robótica y así facilitar la integración de los mecanismos programados en el robot físico. Para ello, ROS interactúa a bajo nivel con el *hardware* del robot haciendo de intérprete del lenguaje de programación de alto nivel del programa para conseguir que los dispositivos, periféricos y motores del robot respondan de la forma deseada. Es decir, hace la función de sistema operativo proporcionando una manera sencilla de conectar el *software* con el *hardware*. Además, ROS también dispone de una serie de funciones: la comunicación entre procesos, gestión de paquetes, multitud de librerías y herramientas de optimización, las cuales consiguen acelerar el ritmo de desarrollo de *software* y permite agilizar el trabajo del programador. La versión de ROS utilizada es ROS Kinetic ejecutado sobre GNU/Linux, con Ubuntu 16.04 como distribución [5].

ROS ha sido elegido debido a la flexibilidad y la sencillez que ofrece a la hora de desarrollar *software* en robótica. Dispone de una gran comunidad que ha ido creando

una serie de repositorios que permiten la reutilización de código. Además, al ser código abierto puede hacerse cambios y mejoras del trabajo de otros y volver a subirlo para que todo el mundo pueda acceder a él. Todo esto, ligado a la gran versatilidad que ofrece en relación al uso de un lenguaje u otro, su adaptación a los simuladores y la facilidad de implementación sobre cualquier tipo de robot permite que no se necesite un nivel experto para adentrarte en la robótica y que los programadoras dispongan de multitud de herramientas para aumentar su productividad.

ROS se utiliza en la gran mayoría de los proyectos del ámbito de la robótica a causa de su diseño modular y distribuido. Esto permite que se pueda escoger que es útil para el proyecto que se está desarrollando e implementar solo esa parte. Es decir, su característica modular permite reutilizar solo el código que se necesita sin tener problemas de no haber implementado otras funciones inútiles para ese comportamiento que se está programando. La naturaleza distribuida hace referencia al hecho de que cualquier usuario puede realizar programas y ponerlos al abasto de todo el mundo generando lo que se llama paquetes. Estos paquetes permiten que cualquiera pueda acceder a funciones sencillas, pero laboriosas, ahorrándole una cantidad de tiempo considerable.

El desarrollo de *software* ha estado creciendo significativamente desde que ROS se asentó (2009) gracias a la existencia de una comunidad desarrolladora de nuevas funciones, algoritmos, mecanismos y comportamientos. Además, el hecho de que ROS tenga un equipo importante y experimentado dirigiéndolo le ha otorgado una gran fiabilidad y calidad. Hasta el punto de convertirse en la principal herramienta de desarrollo de *software* en robótica en el ámbito industrial, docente y en investigación. Estos hechos han provocado que entornos como Player/Stage hayan dejado de tener soporte y hayan sido substituidos por ROS, motivo que ha dado lugar a la propuesta de este proyecto por parte del tutor.

3.2 Principios básicos de ROS

ROS tiene una serie de niveles de conceptos: nivel del sistema de archivos, nivel de computación gráfica y nivel de comunidad.

Nivel del sistema de archivos

Paquetes: los paquetes consisten en los módulos primarios de la organización de *software* de ROS. Un paquete puede constar de nodos de ROS, bibliotecas y librerías, archivos de configuración o cualquier tipo de información organizada eficazmente y dependiente de ROS. Es decir, los paquetes son la unidad más básica que se puede construir y es gracia a éstos que es modular y de fácil implementación.

Nivel gráfico de computación

Maestro: el Maestro o ROS Master es el encargado del registro de nombres permitiendo encontrar los distintos elementos que componen el nivel gráfico de compu-

tación y la conexión entre ellos. Es decir, es imprescindible para que cada elemento sea consciente del resto de elementos existentes y así puedan comunicarse o invocarse.

Nodos: los nodos son aquellos procesos encargados de realizar las tareas de cálculo y computación. Debido al carácter modular se tiene tendencia a crear muchos nodos en un mismo sistema, encargándose cada nodo de uno o varios módulos siempre que éstos contribuyan en la misma tarea.

Servidor de parámetros: es un servicio que pertenece al ROS Master y cuya función es la de almacenar los datos en una ubicación central.

Mensajes: los mensajes son el sistema de comunicación entre nodos. Éste es, simplemente, una información estructurada que puede contener los datos de cualquier primitiva (integer, float, boolean, etc.) o, más complejos, pudiéndose enviar strings, arrays o estructuras.

Tópicos: el funcionamiento de la comunicación por mensajes en ROS se basa en la metodología publicador/suscriptor. Esto significa que un nodo para enviar un mensaje a otro ha de publicar este mensaje en el tópico que se desea qué lo reciba y cuando un nodo quiere recibir la información que obtiene un tópico deberá suscribirse a él. Es decir, se lleva a cabo un enrutamiento de los mensajes permitiendo que al programar puedas elegir a qué tópicos van a ir los mensajes que se publiquen y de que tópicos se leerá la información suscribiéndote a ellos. Este enrutamiento se produce con éxito gracias a los nombres de los tópicos. La ventaja que ofrece el método de publicador/suscriptor es el hecho de que cada publicador y cada suscriptor no es consciente del resto. Por lo tanto, permite que pueda haber varios para un mismo tópico, con un objetivo distinto cada uno, sin ningún tipo de interferencia entre ellos.

Servicios: los servicios son procesos que permiten llegar donde el modelo publicador/suscriptor no llega. El modelo publicador/suscriptor es muy flexible, pero tiene un problema debido a que es de sentido único. Como consecuencia de este hecho puede que no tenga la reactividad necesaria para algunas de las peticiones. Por ello, surgieron los servicios que se consisten en un par de estructuras encargadas de abarcar esta función. Una estructura se ocupa del envío del mensaje de solicitud y la otra de la respuesta. Es decir, el nodo es el encargado de la creación de un servicio identificado por un nombre y este servicio es usado por un cliente para efectuar la solicitud de mensaje y esperar la respuesta [6].

En la figura 3.1 se aprecia un ejemplo gráfico del funcionamiento de ROS. Hay 4 nodos y 2 tópicos. EL nodo 1 se encuetra publicando un mensaje en ambos tópicos. El nodo 2 se encuentra publicando un mensaje en el tópico 2 y, a su vez, es subscriptor del tópico 1. Los nodos 3 y 4 son subscriptores de los tópicos 1 y 2, respectivamente. Además, estos nodos han creado un servicio donde el nodo 4 ha solicitado este servicio y el nodo 3 le ha respondido.

3.3 Simulador Stage

3.3.1 Introducción del simulador

Stage es un simulador de *software* libre, bajo los términos de GNU (General Public License) creado por Player Project y consta del soporte de una gran comunidad. Se trata de un simulador caracterizado por ser una herramienta de *software* capaz de cargar mapas en dos dimensiones y de poder introducir en él prácticamente múltiples tipos

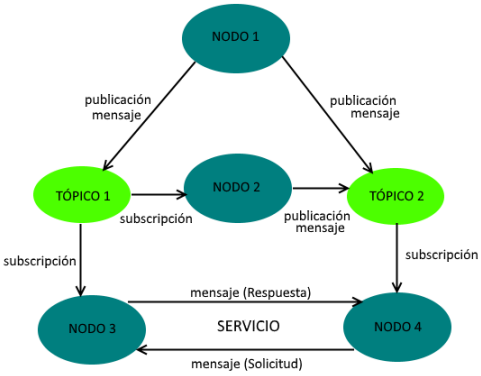

Figura 3.1: Ejemplo del funcionamiento de ROS.

de robots. Además, se permite la interacción de los robots con el medio simulado y con los objetos creados en él. Fue concebido con la finalidad de simulaciones multi-robot. Además, ofrece modelos simples para evitar un exceso de carga computacional con el fin de conseguir un sistema eficaz y de reacción real a coste de perder realismo gráfico. Es decir, intenta conseguir el mejor balance entre realismo y un sistema lo suficientemente rápido para realizar simulaciones multi-robot con grandes poblaciones [7].

Stage ha sido elegido debido a su enfoque multi-robot, a su compatibilidad total con ROS y a la flexibilidad que ofrece de desarrollo de mapas y robots. Este proyecto se centra en un programa de interacción de varios robots en un mismo medio. Por ello, es necesario que las simulaciones estén adaptadas a ellos y sean lo más realistas posibles. Así que la característica multi-robot y la rapidez de ejecución juegan un papel prioritario a la hora de elegir el simulador.

Desde que comenzó a desarrollarse Stage fue pensado para que existiese una compatibilidad completa con ROS. Este planteamiento ha supuesto una gran ventaja, ya que permite poder ejecutar las pruebas sin problemas de incongruencias o errores. Este hecho ha provocado que haya acabado siendo el simulador en más alta consideración para la realización de este proyecto.

Otra de las características de este simulador es que permiten cargar mapas de bits cuyo diseño es muy simple y ocupa poca cantidad de memoria. Esto concede al usuario poder desarrollar mapas complejos con un un simple programa de dibujo. Además, permite introducir robots con múltiples características con la misma facilidad.

También existen otros simuladores que se consideraron utilizar para este proyecto por sus características similares a Stage. En un principio se pretendía utilizar ARGOS que consiste en un simulador multi-robot en 3 dimensiones. Se descartó debido a que se necesita un programa para enlazarlo con ROS y no se consiguió que este programa funcionase correctamente. Una vez determinado que no se podría utilizar ARGOS se decidió utilizar otro simulador llamado Gazebo. Éste es muy similar a ARGOS debido a que también está especializado en sistemas multi-robot y es en 3 dimensiones. Sin

embargo, tampoco se encuentra integrado en ROS y se descartó al no conseguir establecer la conexión a este entorno de trabajo. Por lo tanto, se acabó eligiendo Stage que resolvía todos los problemas encontrados con el resto de simuladores y proporcionaba características similares.

3.3.2 Principios básicos de Stage

Utiliza un sistema de coordenadas levógiro. Es decir, los giros hacia la izquierda los considera positivo y negativos hacia la derecha. Su eje de coordenadas es el habitual, las abscisas (eje X) aumentan hacia la derecha y las ordenadas (eje Y) hacia arriba.

Sus mundos son ficheros bitmap, los cuales consisten en imágenes png. Esto permite que cualquier usuario puede crear un mapa personalizado con una herramienta de dibujo o con imágenes de Internet.

La creación de robots se basa en modelos. Estos modelos pueden ser sensores como sonar, infrarrojos o láser. También actuadores, bases de robots móviles, pinzas, parachoques, etc . Además, se pueden crear objetos o añadir planos.

La mayoría de robots u objetos pueden encontrarse en las librerías. Estas son archivos de tipo .inc que hace la función de una clase de programación donde puede llamar a los métodos que hay dentro de esa clase, pero en este caso se llama a robots u objetos [8].

Este simulador se encuentra ligado directamente a ROS. Para ello, ROS dispone de un paquete específico para Stage llamado stage_ros. Éste utiliza el nodo stageros, cuya función es la de enlazar el simulador a ROS. Este nodo permite localizar los modelos Stage de tipo sensores, actuadores, posicionamiento y cámara y mapearlos en ROS[9].

En el cuadro 3.1 puede verse un ejemplo de algunos de los tópicos de Stage mapeados en ROS. Se aprecia el tópico cdm_vel, el cual corresponde al tópico encargado de gestionar el movimiento del robot. El siguiente es el tópico de odometría, llamado odom. Se observa que el nombre del robot aparece dentro del propio nombre del tópico indicando así a que robot pertenece dicho tópico.

/robot_1/cmd_vel
/robot_1/odom

Cuadro 3.1: Tópicos de Stage mapeados en ROS.

El simulador Stage presenta una serie de limitaciones que han afectado al planteamiento del proyecto. La principal es el hecho de que el simulador no soporta más de un sensor por robot. Esto provoca que no se puedan utilizar los ultrasonidos y obliga a buscar alguna solución para detectar los obstáculos. Esta alternativa es el sensor láser disponible en el simulador que tiene un rango de 180 grados. Éste substituirá a los 8 sensores de ultrasonidos frontales detectando cualquier obstáculo que se encuentre delante del robot. Esto resuelve la limitación de Stage, pero en las pruebas reales habrá que volver a modificar el programa porque en éstas sí se utilizarán los ultrasonidos del Pioneer 3DX. Además, en el robot real se utilizará el nodo de ARIA de ROS, llamado ROSAria, para poder establecer la conexión PC-Pioneer 3DX comentada en el capítulo 2.

El simulador Stage presenta una serie de limitaciones que hay que tener en cuenta en la implementación. La limitación principal es el hecho de que solo admite un sensor por robot. Esto impide que puedan utilizarse los ultrasonidos y hay que buscar una alternativa. Como solución se ha optado por utilizar un sensor láser, cuyo rango es igual a 180 grados. Por lo tanto, puede sustituir a los 8 sensores de ultrasonidos del frontal del robot.

En la figura 3.2 puede verse un ejemplo de cómo son los mapas de bits que emplea Stage y de como se representa a los Pioneer (círculo azul y círculo rojo). También se aprecia que de los robots salen unos triángulos verdes. Éstos son los sensores de ultrasonidos, los cuales sí que aparecen, pero Stage no permite utilizarlos por la limitación ya comentada.

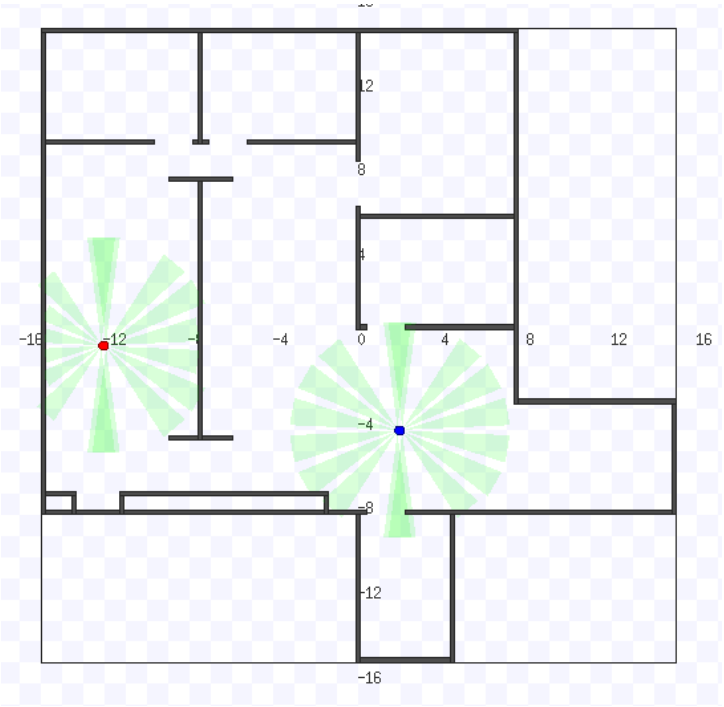

Figura 3.2: Ejemplo de mapa en el simulador Stage.

CAPÍTULO 4

ALGORITMOS DE NAVEGACIÓN Y ASIGNACIÓN DE TAREAS

En este capítulo se explican los algoritmos implementados y cómo funcionan. Entre ellos se encuentran el algoritmo de evitación de obstáculos y navegación y los algoritmos de asignación de tareas basados en subastas.

4.1 Navegación: Algoritmos de evitación de obstáculos

Como primer objetivo del proyecto se tiene que conseguir implementar un mecanismo de navegación, ya que las tareas que ejecutarán se basan en esto: alcanzar unas coordenadas objetivo.

Este algoritmo de navegación tiene que permitir que el robot llegue a las coordenadas objetivos con la mayor efectividad y seguridad posible. Para ello, ha de implementarse un algoritmo capaz de afrontar situaciones complicadas como sería un entorno dinámico con obstáculos e impedimentos. Por lo tanto, se ha de crear un algoritmo capaz de evitar aquello que se interponga en el camino del robot y aun así consiga llegar a las coordenadas deseadas. Es decir, se necesita un algoritmo de evitación de obstáculos para poder navegar con seguridad. Existen tres tipos de tipos de arquitecturas: Jerárquica, Reactiva e Híbrida. Su estructura puede observarse en la figura 4.1.

La arquitectura jerárquica se basa en 3 factores: visualización del entorno y creación de un mapa, planificación de la ruta a seguir en función del mapa y alcanzar el objetivo siguiendo esta planificación. Esto supone una serie de ventajas y de desventajas. Las ventajas son que al construirse un mapa se conoce el entorno permitiéndole planificar una trayectoria más eficaz. La desventaja es que se puede necesitar una cantidad de tiempo y de memoria elevada para realizar el mapeo y la planificación. Además, en entornos dinámicos el mapa no es estático así que tendría que rehacerse el mapa y la planificación con frecuencia.

Todo esto puede suponer un tiempo demasiado elevado para algunos entornos excesivamente dinámicos donde se tiene que reaccionar con la mayor celeridad posible

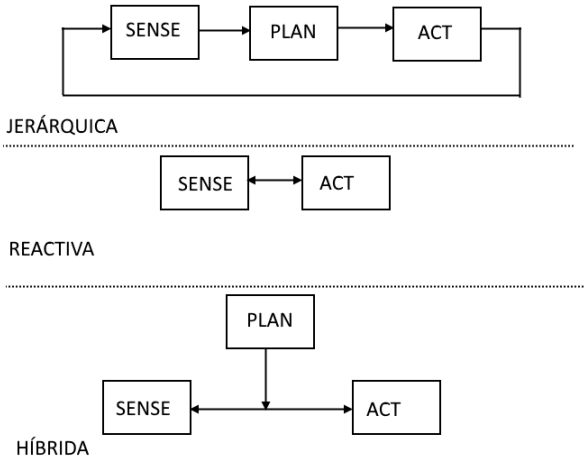

Figura 4.1: Comparación de las distintas arquitecturas.

para preservar la integridad física de la plataforma robótica. Es decir, para entornos estáticos, donde el mapa no cambiará, la mejor opción es seguir los tres pasos anteriores, pero en entornos muy dinámicos, donde el mapa cambia a medida que pasa el tiempo, es arriesgado mantener al robot haciendo cálculos por un tiempo elevado, ya que puede producirse la situación de que el entorno cambie y el robot no esté preparado para afrontar este cambio.

Para entornos dinámicos y entornos en los que se necesite una arquitectura simple se utiliza , sobre todo, arquitecturas reactivas. Esto quiere decir, detectar y reaccionar o, como se conoce en robótica, "Sense and Act"(Sense and Act (SA)). Esta arquitectura se basa en la eliminación de la fase de mapeo y planificación de manera que su comportamiento únicamente dependa del valor de los sensores en cada instante de tiempo. Algunos de los algoritmos reactivos de evitación de obstáculos más conocidos son: "Bug Algorithm.º, el utilizado en este proyecto, llamado campos de potencial [10].

La gran ventaja de esta arquitectura (SA) es el bajo coste computacional y la capacidad casi instantánea de reacción ante un obstáculo. Además, no se necesita una gran capacidad de memoria. Por todo ello, se ha elegido este tipo de arquitectura.

Por último, existen una serie de arquitecturas híbridas que intentan combinar las arquitecturas reactivas y las jerárquicas con el objetivo de conseguir reunir las ventajas y reducir las limitaciones de ambas. Es decir, se intentar obtener la efectividad de la arquitectura jerárquica y la reactividad de la reactiva. En la última década se esta haciendo grandes avances en este campo gracias a los nuevos procesadores que disponen una capacidad y velocidad de cómputo inmensa permitiendo planificaciones elaboradas casi instantáneas.

4.1.1 Evitación de obstáculos por campos de potencial

Este tipo de evitación de obstáculos es una de las más sencillas conceptualmente. Básicamente consiste en la generación de vectores encargados de indicar al robot la

ruta a seguir. Para ello, se considera al robot como una partícula que se encuentra sometida a distintas de fuerzas de atracción y repulsión. La coordenada objetivo ejerce una fuerza de atracción en el robot y los obstáculos de repulsión, representadas en forma de vector. Es decir, funciona tal como si el robot fuese una partícula eléctrica, la cual se atrae por las de signo opuesto (objetivo) y se repele por las del mismo signo (obstáculos). En la figura 4.2 puede verse un ejemplo de cómo funcionan los campos de potencial.

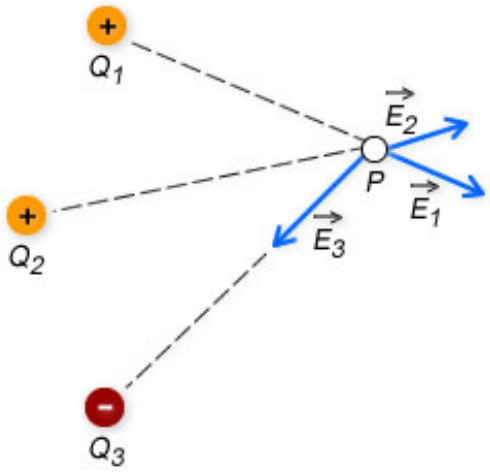

Figura 4.2: Ejemplo de campos de potencial sobre carga eléctrica.

Este algoritmo se ha planteado de tal forma que existen dos tipos de comportamientos: comportamiento de evitación obstáculos y el de navegación hacia el objetivo, definidos por un vector obstáculo y objetivo, respectivamente. El vector objetivo es, simplemente, un vector que comienza en el robot con dirección y sentido hacia las coordenadas a alcanzar. Por lo tanto, si solo estuviese éste seguiría una trayectoria directa hacia el objetivo. El vector obstáculo es la suma de todos los vectores generados por los obstáculos. Estos vectores funcionan a la inversa que el vector objetivo. Salen del obstáculo en dirección y sentido al robot. Es decir, se genera un vector que indica una ruta que se aleja del obstáculo. Una vez se tienen ambos vectores se suman y el vector resultante es la trayectoria a seguir. En la figura 4.3 puede verse un ejemplo del funcionamiento de este algoritmo.

Es decir, para determinar la ruta a seguir solo se ha de realizar la suma de todos lo vectores y el vector resultante indicará la ruta. Éste es el funcionamiento básico de este tipo de sistema de evitación de obstáculos. Pero existen más factores a tener en cuenta al generar los vectores. Se ha de definir el módulo de los vectores de tal manera que siempre se le dé mayor prioridad al vector obstáculo, ya que la seguridad del robot es lo primordial. Como consecuencia de este hecho, hay que generar vectores obstáculos con módulo proporcionalmente superior al vector objetivo para conseguir que cuando el robot se aproxime a una zona de peligro de colisión el vector obstáculo resultante sea considerablemente mayor que el objetivo y así, al sumarlos, el vector final se aleje

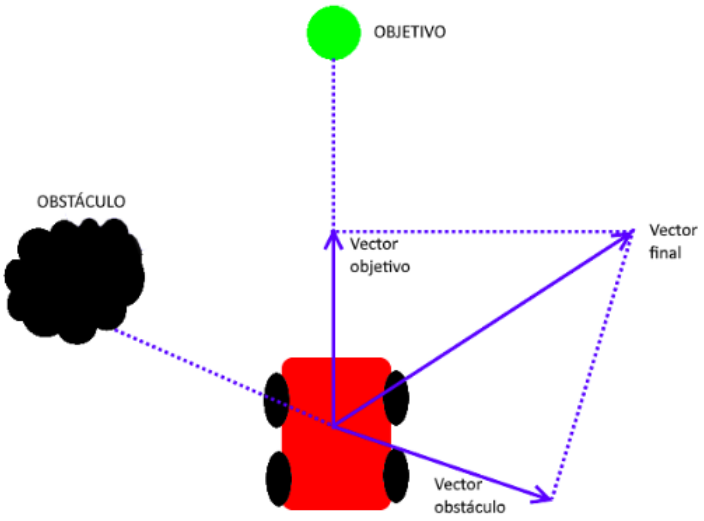

Figura 4.3: Funcionamiento campos de potencial.

del obstáculo. Para ello, se suelen introducir lo que se denomina "pesos"que se trata de una constante por la cual se multiplica cada vector y según el valor de ésta se da más prioridad a un vector u otro. En la ecuación 4.1 se aprecia esta suma de vectores y, también, aparece una constante w_obj que se multiplica por el vector objetivo y otra, llamada w_obs, multiplicada por el vector obstáculo correspondientes a los "pesos"de cada vector.

$$\vec{V} = w_{obj} \cdot \vec{V}_{objetivo} + w_{obs} \cdot \vec{V}_{obstáculo} \tag{4.1}$$

En la figura 4.4 se puede apreciar la diferencia existente entre un algoritmo con los vectores bien proporcionados (izquierda) y uno que no lo está (derecha). Se observa como en la figura de la izquierda el vector resultante se aleja considerablemente del objetivo. En cambio, en la figura que los vectores no se encuentran bien proporcionados (derecha) no se aleja lo suficiente y la ruta supondría que el robot colisionaría con el obstáculo. Estos ejemplos demuestran la gran importancia del factor de los "pesos", ya que puede suponer la obtención de un algoritmo seguro o uno peligroso.

Cabe destacar que este algoritmo de evitación de obstáculos no es completo. Esto significa que si existe camino puede ser que este algoritmo no lo encuentre. Normalmente se produce cuando se encuentra con obstáculos demasiado grandes situados entre el robot y el objetivo. Esto provoca que el vector obstáculo tenga la misma dirección, pero distinto sentido que el vector objetivo. Por lo tanto, el vector resultante no permitirá que esquive el obstáculo, solamente le hará retroceder.

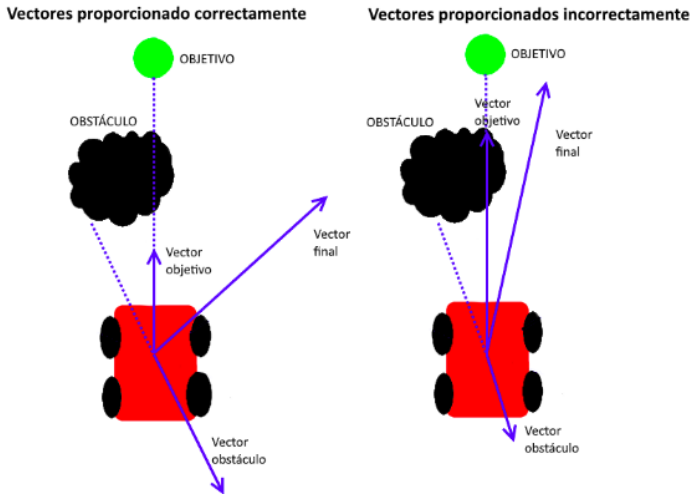

Figura 4.4: Comparación entre vectores proporcionados de forma correcta e incorrecta.

4.2 Asignación de tareas basada en subastas

4.2.1 Funcionamiento general

Una vez programado el algoritmo de evitación de obstáculos se procede a implementar los algoritmos correspondientes al objetivo principal del proyecto: algoritmo de asignación de tareas para sistemas multi-robot basado en subastas. Las tareas se trata de coordenadas y son el producto a subastar.

La asignación de tarea basada en subastas consiste en un algoritmo donde es necesario que exista la figura de subastador y la de pujador. En este proyecto el subastador consiste en un robot que actúa como dirigente de una tarea (solo un subastador por tarea) decidiendo si los pujadores pueden participar en dicha tarea. Los pujadores son todos aquellos robots que no son subastadores y para que se les asigne una tarea deben solicitársela al subastador. La forma de solicitarla es mediante una puja. La puja consiste en una oferta, la cual tendrá más valor o menos en función de las condiciones que imponga el subastador. En este proyecto el valor de la puja aumenta cuanto más cercano se encuentre el robot de las coordenadas de la tarea.

En este caso las tareas consisten en alcanzar unas coordenadas objetivo. Cada objetivo tendrá un líder, que actuará como subastador. Los que no sean líderes serán los pujadores. El líder es el primero en navegar hacia las coordenadas de la tarea. Una vez alcance su objetivo envía una solicitud de apoyo a los pujadores. Éstos solicitan apoyarle mediante la puja. En ese momento el subastador (líder) decide los robots que le prestarán apoyo en función del valor de sus pujas.

Para conseguir este funcionamiento se han implementado dos tipos algoritmos de asignación de tareas: uno de elección de líder y otro de elección de los apoyos del líder. El primero no es estrictamente un algoritmo basado en subastas, ya que, para ello, se necesita que haya un robot que actúe de subastador y otros que actúen de pujadores. Sin embargo, en este algoritmo no existen estas figuras, sino que todos actúan a la vez de subastadores y de pujadores. Esto significa que todos consultan con el resto si pueden

ser líder de una tarea. Para ello, envían una puja de la tarea que quieren liderar y entre todos ellos comparan las pujas y deciden en conjunto quiénes serán los líderes. Como el valor de la puja depende de la cercanía, el líder de cada tarea será el robot que se encuentre más cercano a ella. Solamente podrá haber un líder por tarea. En el apartado 4.2.2 se encuentra explicado en detalle. El segundo algoritmo sí se basa estrictamente en las subastas, ya que en éste los líderes, elegidos en el algoritmo anterior, actúan de subastadores y el resto de robots hacen el papel de pujadores, apartado 4.2.3.

Estos algoritmos han sido propuestos por el tutor con el fin de conseguir un mecanismo capaz determinar qué robot realizará cada una de las tareas de la misión. Para ello, existen una serie de fases: primero los robots realizan peticiones al subastador o al resto de robots, en caso de que no haya subastador, pujando por la tarea a la que quieren optar. En este instante se emplea un periodo de tiempo para que el subastador o los subastadores analicen las pujas recibidas. Según esta puja se determina quiénes optarán a cada una de las tareas y se le informa de ello.

4.2.2 Elección de los líderes de la misión

A continuación se procede a explicar el algoritmo de elección de los líderes. Cada tarea, es decir, cada coordenada objetivo solo puede disponer de un único líder, el cual será el robot principal a la hora de llevar a cabo la misión. Esto quiere decir que ese robot será el que se haga cargo desde un primer momento de encaminarse hacia las coordenadas. Además, los líderes serán subastadores en el momento de aplicar la asignación de las tareas de apoyo explicadas en la siguiente sección (apartado 4.2.3).

La elección del líder se lleva a cabo mediante un algoritmo donde todos los robots intercambian mensajes y acaban decidiendo que robot será el líder de cada tarea. Esta decisión se lleva a cabo intercambiando las coordenadas objetivo a las que quiere optar cada robot. Todos los robots conocen que tarea quiere liderar el resto y la puja que hace cada uno por ella. Conociendo cada robot el interés del resto se deciden los líderes según la proximidad de cada uno a las coordenadas de las tareas.

Cada robot recibe el mensaje y analiza si algún robot quiere ir al mismo punto que él. En caso de que así sea, si el otro robot se encuentra más cerca que él deja de optar a ser líder de tarea y pasa a la siguiente (segunda tarea más cercana) y si estas nuevas coordenadas también coinciden con las de otro robot más cercano que él pasa al siguiente. Y así hasta que se acaben las tareas o este robot sea el más cercano y se convierta en el líder de esa misión. Se aprecia claramente como ningún robot actúa de subastador a la hora de decidir los líderes, sino que todos son subastadores y pujadores para acabar todos llegando a la misma conclusión gracias a que conocen toda la información.

Una vez establecido que líderes se convierten en subastadores de sus tareas y se dirigen hacia las coordenadas. Los que no son líderes son pujadores y se quedan en su posición de inicio ejecutando el algoritmo de 'Elección de los apoyos de los líderes'. En el listado de código 4.1 puede verse en pseudocódigo el funcionamiento de este algoritmo.

```
1 Petición tarea t;
2 tiempo_inicio = tiempo_ahora();
3 while((tiempo_ahora() - tiempo_inicio) < tiempo_evaluar_peticiones)
```

```
{
    if(nuevo_mensaje)
    {
        if(tarea ya tiene líder)
        {
            Petición siguiente tarea;
        }
        else if(otro robot pide la misma tarea)
        {
            if(el otro robot se encuentra a menos distancia de la tarea)
            {
                Petición siguiente tarea;
            }
            else if(el otro robot se encuentra a la misma distancia)
            {
                if(ID robot menor que el ID del otro)
                {
                    líder de la tarea;
                }
                else
                {
                    Petición siguiente tarea;
                }
            }
        }
    }
}
```

Listing 4.1: Algortimo elección de líder.

4.2.3 Elección de los apoyos de los líderes de la misión

Una vez ha acabado el algoritmo de elección de líder y éstos han comenzado su misión, los robots restantes comienzan a ejecutar este algoritmo. El presente algoritmo se basa, como su nombre indica, en ofrecer su apoyo a los robots líderes en la ejecución de su tarea. Esto significa que estos robots pueden llegar a ser partícipes de alguna de las tareas.

Ayudar al robot líder significa ir hacia el objetivo de la misión cuando el robot líder ya haya llegado a ella y haya solicitado ayuda. Cuando el robot líder de la misión llega al objetivo evalúa la situación y pide ayuda a todos los demás. Todos recibirán el mensaje, pero solo evaluarán ofrecerle su ayuda los robots que no sean líderes, es decir, los pujadores. En este momento entra en juego el algoritmo de este apartado.

El algoritmo funciona de tal forma que los robots que se encuentran ejecutándolo están a la espera de recibir una petición de ayuda. Cuando ésta llega envían su puja de ayuda al robot líder que la ha solicitado. El robot líder, actuando como subastador, evalúa las pujas recibidas y opta por escoger a los robots que han hecho una puja mayor. En este caso el valor de la puja consiste en la cercanía a las coordenadas: cuanto más cercano se encuentre el robot mayor es la puja. Una vez que ha decidido quiénes le proporcionarán la ayuda, les envía un mensaje de confirmación.

Los robots que respondieron a la solicitud de ayuda esperan una cantidad de tiempo determinado en espera de la respuesta. Si ésta no llega seguirán en espera de otras peticiones de apoyo, pero si llega se ponen en marcha hacia el objetivo. El robot líder

los esperará y una vez lleguen al objetivo los robots de apoyo la tarea se habrá realizado con éxito y pueden ir a sus posiciones de reposo. En el listado de código 4.2 se aprecia en pseudocódigo el funcionamiento de este algoritmo.

```
1  Petición del líder para apoyo en su tarea;
2  tiempo_inicio = tiempo_ahora();
3  while((tiempo_ahora() - tiempo_inicio) < tiempo_evaluar_peticiones)
4  {
5      if(número robots que han pujado < número máximo de robots de apoyo)
6      {
7          for(robot_id = 0; robot_id < número robots que han pujado; robot_id++)
8          {
9              mensaje de confirmación(robot_id);
10         }
11     }
12     else
13     {
14         ordenar los robots según el valor de su puja;
15         for(robot_id = 0; robot_id < número máximo de robots de apoyo; robot_id
               ++)
16         {
17             mensaje de confirmación(robot_id);
18         }
19     }
20 }
```

Listing 4.2: Algoritmo de elección de apoyo.

CAPÍTULO 5

IMPLEMENTACIÓN SOFTWARE

En este capítulo se explica cómo se han implementado los algoritmos del capítulo anterior (Capítulo 4). Para ello, conviene remarcar, una vez más, en qué consisten las tareas de este proyecto: coordenadas objetivo que deben ser alcanzadas por los robots. Y mencionar los apartados de la implementación que se explican en este capítulo:

- Algoritmo de evitación de obstáculos y navegación.
- Caracteríticas del sistema de comunicación basado en *sockets*.
- Programa monitor y los tipos mensajes existentes.
- Algoritmo de subastas.

5.1 Evitación de obstáculos y navegación

El Pioneer 3DX dispone de ultrasonidos, por lo tanto, la detección de obstáculos debería realizarse con ellos. Sin embargo, no es posible usarlo en el simulador, ya que Stage tiene limitaciones y no permite que cada robot tenga más de un sensor. Esto implica que no se puedan utilizar los ultrasonidos en las pruebas con Stage y se tenga que utilizar un sensor láser adaptado para que simule la función de los ultrasonidos. En cambio, en las pruebas con el robot real sí se utilizarán los ultrasonidos.

El sensor láser es capaz de cubrir 180 grados. Éste se encuentra situado sobre el robot orientado hacía delante. Es decir, cubre la misma área que cubrían los 8 ultrasonidos frontales del Pioneer. Su funcionamiento consiste en estar activo en todo momento y cuando un obstáculo entra dentro dentro de su alcance detectarlo. La manera que tiene de detectarlos es mediante los rayos que se reflejan en el obstáculo. El ángulo en el que se reciba el rayo reflejado determinará dónde se encuentra el obstáculo.

A pesar de que la información proviene de un sensor láser se ha de implementar el programa como si se usasen los sensores de ultrasonidos. Para ello, se ha de dividir el rango del sensor láser en 8 sectores, uno por cada sensor de ultrasonidos frontal del

robot, tal como se aprecia en la figura 5.1. Esto permite simular que la información proveniente de cada sector es de un sensor distinto. Por lo tanto, como el rango es de 180 grados y esta información debería obtenerse de 8 sensores de ultrasonidos se divide el rango en 8 regiones, tal como viene definido en la Ecuación 5.1.

$$ángulo\_región = \frac{rango\_láser\_simulado}{número\_total\_sensores\_ultrasonidos} \tag{5.1}$$

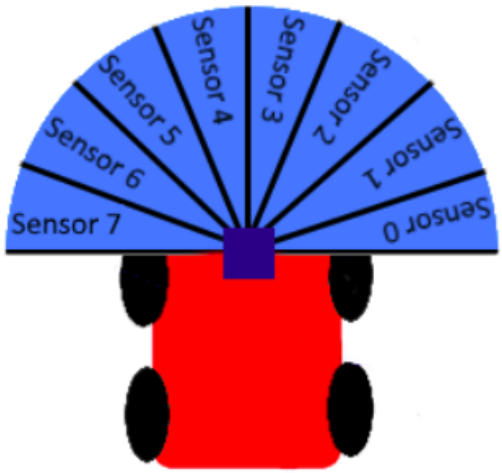

Figura 5.1: Rango del sensor láser dividido en regiones.

Para obtener los datos del láser hay que subscribirse al tópico del láser proveniente al simulador Stage. Para realizar una suscripción ha de conocerse el nombre del tópico. Éste se llama robot_ más el ID del robot más /base_scan1. Por ejemplo, si el nombre del robot fuese el robot_1 el tópico sería: robot_1/base_scan_1. En el listado de código 5.1 se aprecia la creación de un subscriptor: se genera un string con el nombre del tópico y, posteriormente, se subscribe a este tópico mediante el método de ROS nh.subscribe.

```
1    // subscribe to robot's laser scan topic "robot_X/base_scan"
2    string sonar_scan_topic_name = "robot_";
3    sonar_scan_topic_name += id;
4    sonar_scan_topic_name += "/base_scan_1";
5    ros::Subscriber sub = nh.subscribe(sonar_scan_topic_name, 1, scanCallback);
```

Listing 5.1: Subscriptor al tópico del láser.

ROS dispone de una clase, llamada LaserScan, la cual se encarga de gestionar la información recibida del subscriptor del sensor láser. La función principal de este programa es rellenar un *array* con el valor de distancia de los obstáculos que se encuentre.

Este *array* se llama ranges y dispone de 180 posiciones: una para cada ángulo de su rango. Es decir, ranges es un *array* de 180 posiciones donde la posición cero corresponde al ángulo 0, la uno al ángulo 1 y así sucesivamente. La distancia de los obstáculos que encuentre se guardarán en una posición u otra del *array* en función de su ángulo respecto al robot. Para todos aquellos ángulos que no detecte ningún obstáculo el valor de distancia guardado no será cero, sino el alcance máximo del láser. En la tabla 5.1 puede verse un ejemplo de como funciona este *array* viendo un valor de distancia para cada posición sin importar si ha encontrado obstáculo o no. Si consideramos que el alcance máximo del sensor es de 7 se aprecia claramente que para la posición 0 y 1 (ángulo 0 y 1) ha encontrado obstáculo porque los valores de distancia están muy alejados de 7. Sin embargo, en la posición 2 no ha encontrado obstáculo y, por ello, el valor de distancia es tan próximo al valor máximo.

Ángulo	Distancia del obstáculo
0	1.502
1	0.774
2	6.987
...	...

Cuadro 5.1: Ejemplo array ranges.

Una vez establecidas las regiones y subscrito al láser hay que proceder al siguiente paso. Este paso es el de obtener un único resultado por cada región, ya que hay que simular que solo existe un sensor por cada región y éste no puede devolver dos informaciones distintas. Así que se escoge el valor de distancia más pequeño de las posiciones del *array* ranges correspondientes a cada área. Además, se guarda el ángulo (posición del *array*) del valor de distancia escogido, ya que con la distancia y el ángulo puede conocerse la posición del obstáculo.

En este momento ya se dispone de la capacidad de obtener 8 distintos valores, uno por sector. Ahora, es el turno de analizar estos valores, ya que hay que determinar cuáles son los que interesan. Esto significa establecer una distancia mínima a partir de la cual se tiene en consideración el resultado. Esto es porque solo interesa la detección de obstáculos cuando estos se encuentran cercanos al robot, pero suficientemente lejos como para que éste tenga tiempo de reaccionar. Además, así se descartan todos aquellos valores de distancia correspondientes a posiciones donde no ha detectado nada.

A medida que se va eligiendo qué resultados son interesantes hay que comenzar a generar los vectores, pero hay una problemática. El problema es que el láser no tiene el mismo Sistemas De Coordenadas (SDC) que el simulador Stage. Si el robot se encuentra con orientación 0 grados, el láser abarca el primer y cuarto cuadrante respecto al SDC de Stage. Por lo tanto, como el láser tiene un rango de 180 grados, su primer cuadrante correspondería con el cuarto de Stage y su segundo con el primero del simulador. En la figura 5.2 se aprecia con mayor claridad la diferencia entre el SDC del sensor láser y el del simulador Stage.

Por lo tanto, si no se tratan los resultados habrá una incongruencia de los ángulos obtenidos en referencia al sistema de coordenadas establecido por Stage. Entonces hay

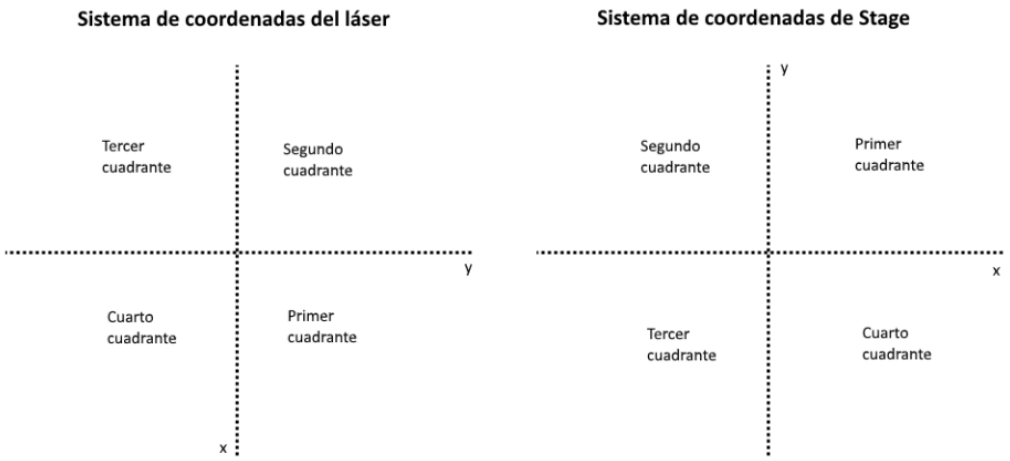

Figura 5.2: Comparación entre sistema de coordenadas del láser y el de Stage.

que hacer una rotación del SDC del láser simulado para que coincida con el de Stage. Se aprecia, con claridad, que el SDC del sensor láser está rotado 90 grados en sentido horario respecto al SDC de Stage. Por lo tanto, para que los SDC coincidan solo hay que restarle 90 a los ángulos obtenidos por el láser para pasarlos al cuadrante anterior.

Otro factor importante es que el láser siempre devolverá ángulos del primer y segundo cuadrante sin importarle la orientación del robot. Por lo tanto, ahora que los ángulos del láser se encuentran referenciados respecto el SDC del simulador Stage hay que tener en cuenta la orientación del robot para determinar dónde se encuentra el obstáculo. Para ello, se le suma a la orientación del robot el ángulo del obstáculo. Éste es el funcionamiento debido a que tanto el robot como el obstáculo están referenciados según el SDC del simulador. Por consiguiente, la suma resultante también será respecto el SDC de Stage.

En la figura 5.3 puede observarse dos imágenes que muestran cómo afecta la orientación del robot. Si el obstáculo se encuentra en el primer cuadrante respecto al SDC del láser al pasarlo al cuarto cuadrante respecto al SDC de Stage tendrá un valor negativo. Por lo tanto, al sumarlos el ángulo del obstáculo será menor que la orientación del robot, tal como se aprecia en la imagen de la derecha. Sin embargo, en el otro caso el obstáculo se encuentra a un ángulo mayor que la orientación del robot, imagen izquierda.

Una vez todos los datos se encuentran referenciados por un mismo SDC y los ángulos se encuentran donde corresponden, es el momento de generar el vector obstáculo resultante. Para ello, se calcula la componente x y la componente y del ángulo aplicando coseno del ángulo para la x y seno para la y. Ésta es la forma de sacar el vector unitario de un ángulo. Ahora toca darle un módulo y aplicarle el "peso". Como módulo se ha optado por una fórmula bastante común en mecanismos de evitación de obstáculos que consiste en la reflejada en la ecuación 5.2.

$$|\vec{V_{obs}}| = \frac{MAX_{DIST} - distancia}{MAX_{DIST}} \tag{5.2}$$

Entonces se ha multiplicado cada componente por el módulo y por el "peso"(w_obs).

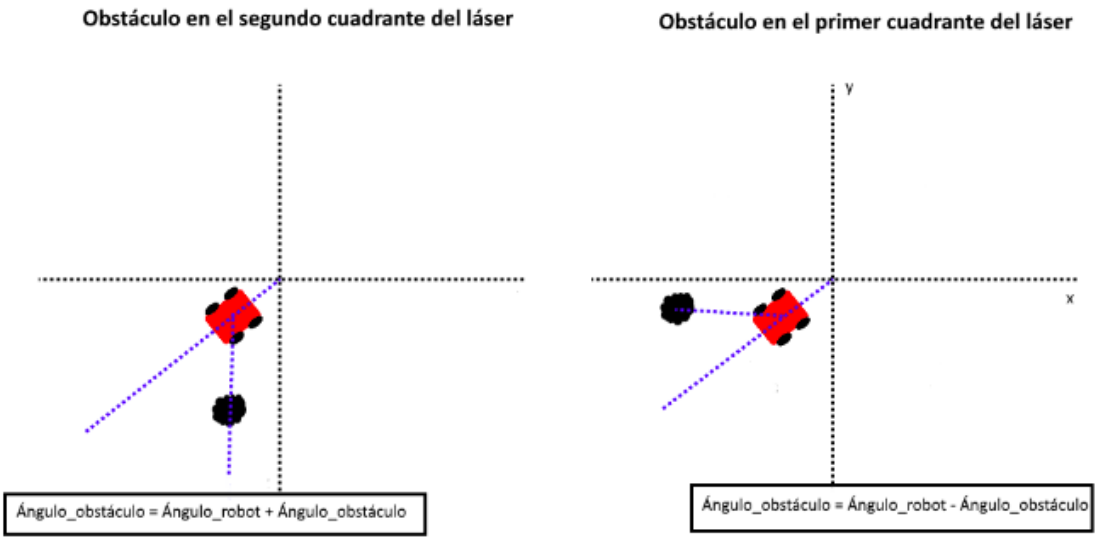

Figura 5.3: Cálculo del ángulo del obstáculo según la orientación del robot.

Así se obtiene el vector, pero éste ahora mismo apunta hacia el obstáculo. Por lo tanto, no alejará al robot de él, sino al revés. Por ello, se multiplica por -1 cada componente para cambiarle el sentido al vector y así conseguir que se aleje del obstáculo. Todo esto está reflejado en las ecuaciones 5.3.

$$\begin{aligned} X_{obs} &= (-1) \cdot cos(ángulo\_obstáculo) \cdot w_{obs} \cdot |\vec{V_{obs}}| \\ Y_{obs} &= (-1) \cdot cos(ángulo\_obstáculo) \cdot w_{obs} \cdot |\vec{V_{obs}}| \end{aligned} \tag{5.3}$$

Estos vectores obstáculos calculados se van sumando para obtener el vector obstáculo final. Una vez se ha evaluado todo el rango del láser se suma este vector al vector objetivo para obtener el vector final definitivo indicador de la ruta a seguir. Se ha seguido este procedimiento porque no ha de calcularse el vector final definitivo hasta que no se haya terminado de calcular el vector final obstáculo, ya que podría faltar información sobre los obstáculos presentes.

En el listado de código 5.2 se refleja en pseudocódigo como funciona el cálculo de los vectores obstáculo explicado. Además, aparecen las variables obstacle_x y obstacle_y que son las variables locales que van sumando los vectores obstáculos y las variables obstacle_vector_x y obstacle_vector_y son las variables globales, cuyo valor solo cambia cuando se ha calculado el vector obstáculo final. Se aprecia como el valor de las variables globales se modifica al salir del while. Es decir, se modifica cuando ha acabado de analizar todo el rango del láser y cuando ya ha calculado todos los vectores obstáculos.

```
1 while(dentro del rango del sensor láser)
2 {
3     for(cada ángulo del sector)
4     {
5         distancia_obstáculo = valor más pequeño de esta sección;
```

```
6       }
7       if(distancia_obstáculo < distancia_mínima)
8       {
9           rotar sistema de coordenadas del láser;
10
11          recalcular ángulo obstáculo teniendo en cuenta la orientación del robot;
12
13          x = cálculo componente x del vector obstáculo;
14          y = cálculo componente y del vector obstáculo;
15
16          obstacle_x += x;
17          obstacle_y += y;
18      }
19      obstacle_vector_x = obstacle_x; //variable global
20      obstacle_vector_y = obstacle_y; //variable global
21  }
```

Listing 5.2: Cálculo del vector obstáculo.

En este momento ya se detectan los obstáculos y se calculan los vectores, pero, todavía, no se ha implementado que la función de navegación que dirija al robot a las coordenadas objetivo. Para establecer esta ruta se sigue utilizando campos de potenciales. El método se basa en sumar el vector obstáculo final un vector objetivo. Este vector objetivo es, simplemente, un vector en dirección y sentido a las coordenadas objetivo.

Esta suma de vectores permite establecer una ruta segura, ya que cuando no haya obstáculos se dirigirá directamente hacia el objetivo y cuando sí haya obstáculos calculará una ruta que se aleje de él y a su vez intente acercarse al objetivo. Sin embargo, para poder seguir esta ruta se ha de establecer un método eficaz de cálculo de vectores y de rotación.

El cálculo de los vectores obstáculos ya ha sido explicado en el apartado 4.1.1. Solo falta aclarar la generación del vector objetivo y cómo se establece la ruta a seguir. Para obtener el vector objetivo se tiene que saber tanto las coordenadas del robot como las coordenadas del objetivo. Entonces, se necesita tener la posición actualizada del robot.

Para obtener la posición del robot hay que subscribirse al tópico de la odometría. Este tópico se llama robot_ más su ID más /odom. Es decir, si fuese el robot_1 sería: robot_1/odom. En el listado de código 5.3 puede verse como se realiza la subscripción, la cual sigue el mismo procedimiento que la subscripción del sensor láser explicada que aparece en el listado de código 5.1.

```
1  string odom_topic_name = "robot_";
2  odom_topic_name += id;
3  odom_topic_name += /odom;
4  ros::Subscriber odom_sub = nh.subscribe(odom_topic_name, 1000, chatterCallback);
```

Listing 5.3: Subscripción a la odometría.

Este tópico permite conocer coordenadas del robot gracias a la utilización de la odometría. Dichas coordenadas se encuentran referenciadas respecto al SDC de Stage. Además, también permite conocer la orientación del robot.

Para procesar la información proveniente de la odometría se utiliza la clase, llamada Odometry, la cual permite acceder a la información del tópico: coordenadas y orienta-

ción. Las coordenadas las devuelve en forma de x,y. Sin embargo, para la orientación devuelve los cuaterniones. Los cuaterniones consisten en un sistema númerico capaz de describir las orientaciones y las rotaciones de objetos en las tres dimensiones. En este caso se trabaja en un simulador de dos dimensiones, por lo tanto, la orientación la devuelve mediante el cuaternión w y z. Hasta ahora se ha estado trabajado con ángulos respecto un SDC cartesiano, por consiguiente, se tiene que utilizar la función atan2. La función atan2 corresponde a la tangente inversa con rango ampliado entre [-π,π]. Por lo tanto, mediante la Ecuación 5.4 se consigue pasar la orientación del robot en cuaterniones a una orientación respecto a un SDC cartesiano [11].

$$\theta = 2 * atan2(orientation_z, orientation_w) \tag{5.4}$$

La forma de calcular la orientación mediante atan2 devuelve la orientación entre (-π, π). Esto significa que para los ángulos de los dos primeros cuadrantes el valor va de 0 a π. Sin embargo, en los otros dos cuadrantes los valores van de 0 a -π, donde 0 correspondería con los 360 grados y -π con los 180 grados.

Llegado a este punto se pasa el vector objetivo a unitario y se le aplica su "peso". Se ha pasado a unitario para que sea más sencillo construir un vector obstáculo proporcionalmente más grande a él para evitar el problema de vectores mal proporcionados reflejado en la figura 4.4.

El algoritmo de evitación de obstáculos ya es capaz de calcular la ruta a seguir. Sin embargo, le falta una función fundamental para que funcione: orientar el robot adecuadamente para que pueda seguirla. Para orientarlo se calcula primero el ángulo a rotar. Calcular el ángulo es, simplemente, hacer el arcotangente de la componente y del vector final partido la componente x (5.5).

$$\theta = atan((V_y)/(V_x)) \tag{5.5}$$

El arcotangente proporciona el ángulo en el que se encuentra la ruta a partir del vector. A pesar de ello, existe un problema porque los ángulos de algunos cuadrantes los posiciona en otros. A causa de esto, se debe realizar una transformación de estos ángulos erróneos. Esta transformación consiste en pasarlo al cuadrante que corresponde sumando o restándole al número Pi el ángulo obtenido. Una vez los ángulos se encuentren en los cuadrantes pertinentes ya se podrá orientar el robot.

El funcionamiento de la rotación consiste en rotar el robot sobre sí mismo consultando su orientación para detenerlo cuando ésta sea la deseada. La manera de obtener un movimiento depende de dos factores: el publicador y el objeto de la clase Twist. Primero se ha de crear un publicador, ya que éste será el que permita enviar las órdenes al tópico encargado del movimiento. Este publicador hay que enlazarlo al tópico del robot, el cual se llama robot_ más el ID del robot más /cmd_vel. Una vez se dispone del publicador se crea el objeto de la clase Twist. Twist es una clase de ROS que permite el tipo de desplazamiento que se desea: angular y lineal. Para el movimiento angular el objeto dispone de un atributo en referencia a Roll, otro a Picht y otro a Yaw. Por lo tanto, otorgándole valor a estas variables se indica que tipo de rotación que se desea. En cambio, para el movimiento lineal dispone de variables que permiten elegir en que eje de coordenadas se desea el desplazamiento. En el listado de código 5.4 se encuentra la creación del publicador cmd_vel_pub. Se aprecia que se genera un string con el nombre del tópico y luego se emplea el método nh.advertise de ROS para enlazar el

publicador al tópico. A continuación, se genera el objeto de la clase Twist, llamado vel_msg. Se aprecia que se le da un valor 0 a todas las variables de movimiento lineal y de movimiento angular menos a la variable z de movimiento angular (angular.z), la cual corresponde a la variable de Yaw y así rote paralelo al plano y sobre sí mismo.

```
1  // cmd_vel_topic = "robot_X/cmd_vel"
2  string cmd_vel_topic_name = "robot_";
3  cmd_vel_topic_name += id;
4  cmd_vel_topic_name += "/cmd_vel";
5  cmd_vel_pub = nh.advertise<geometry_msgs::Twist>(cmd_vel_topic_name, 10);
6
7  geometry_msgs::Twist vel_msg;
8
9  //Variables movimiento angular
10 vel_msg.linear.x =0;
11 vel_msg.linear.y =0;
12 vel_msg.linear.z =0;
13
14 //Variables movimiento lineal
15 vel_msg.angular.x = 0;
16 vel_msg.angular.y = 0;
17 vel_msg.angular.z =abs(angular_speed);
18
19 cmd_vel_pub.publish(vel_msg);
```

Listing 5.4: Publicador del movimiento.

En el listado de código 5.5 se aprecia en pseudocódigo cómo funciona el programa de rotación. Se crea el objeto de la clase Twist (vel_msg), luego se calcula la orientación de la ruta a seguir y se guarda en la variable orientacion_ruta, se asigna un valor de rotación a la variable angular equivalente a Yaw, posteriormente entra en un while donde se publica el movimiento y del cual no saldrá hasta que la orientación del robot no sea la misma que la de la ruta y una vez sale se pone a cero el valor de la variable angular equivalente a Yaw y se publica para que deje de rotar.

```
1  void rotate_robot()
2  {
3  creación del objeto de la clase Twist.h llamado vel_msg;
4
5  orientacion_ruta = calculo_angulo_ruta(vector_definitivo_ruta);
6
7  vel_msg.angular.z = velocidad_rotación; //Asignar valor a Yaw para rotar paralelo
       al plano
8
9  while(orientación_robot =! orientación_ruta)
10 {
11     publicar_movimiento(vel_msg);
12 }
13
14 vel_msg.angular.z = 0;
15 publicar_movimiento(vel_msg); //Parar robot
```

Listing 5.5: Función encargada de la rotación del robot.

5.2 Sockets y características de los mensajes

Para implementar los algoritmos de subastas en sistemas multi-robot primero hay que crear un sistema de comunicación entre los propios robots. Para ello, se ha utilizado un sistema basado en *sockets* que ha sido proporcionado por el tutor del proyecto.

Los *sockets* se tratan de mecanismos de comunicación que hacen de nexo entre dos o más computadoras para establecer un puente mediante el cual puedan intercambiar cualquier tipo de flujo de datos. El *socket* se define por la dirección IP del ordenador, un número de puerto y un protocolo de comunicaciones (User Datagram Protocol (UDP)/TCP). Para este intercambio de paquetes implementan una arquitectura de cliente-servidor. El cliente será en emisor de los mensajes y el servidor el receptor. Para que el cliente consiga hacer llegar al servidor los mensajes es necesario que conozca su dirección IP, su puerto y que utilice el mismo protocolo de comunicación. Para hacer servir los sockets se ha utilizado una librería de C++, proporcionada por el profesor, llamada RcSocket [12].

El programa consta de tres hilos de ejecución: el hilo principal que ejecuta el main, el hilo servidor y el hilo que analiza los mensajes. Estos tres hilos se ejecutarán en paralelo e interactúan entre ellos. Esto es lo que se llama concurrencia de procesos y obliga a tener en cuenta una serie de regiones críticas. Estas regiones críticas consisten en que dos o más hilos quieran acceder al mismo recurso compartido. Éstos no pueden acceder al recurso en el mismo momento, por lo tanto, deben coordinar su ejecución. Existen diversos mecanismos de coordinación como los semáforos, mailbox o exclusión mútua (mutex), el cual se ha utilizado en este proyecto. Ésta consiste en evitar el acceso a más de un hilo a las secciones críticas, entendiendo como secciones críticas los fragmentos de código con recursos compartidos. Para ello, se usa la función mutex, la cual se sitúa justo al entrar en estas secciones críticas para que cuando entre un hilo en ella se bloquee y no pueda entrar ningún hilo más. En el listado de código 5.6 puede verse la utilización de la función mutex en el servidor para proteger la variable headMessage que es de uso compartido.

```
1 //pthread_cond_signal(&conditionMessageExec);
2 headMessage = (headMessage + 1) %RC_MAX_BUFFER_MESSAGE;
3 pthread_mutex_unlock(&mutexMessage);
```

Listing 5.6: Exclusión mutua.

Por lo tanto, se ha de crear una arquitectura cliente-servidor para poder establecer la comunicación. Para ello, se ha creado un hilo de ejecución que actúa como servidor y cuya función es la de recibir todos aquellos mensajes que vayan destinados a él. Este hilo servidor se crea como muestra el listado de código 5.7. Utiliza la función serverFunction() del programa RcSocket, la cual realiza las funciones de recibir los mensajes y guardarlos.

```
1 //Servidor
2 if (pthread_create(&threadServer, (pthread_attr_t*) NULL, serverFunction, (char*)
      NULL) != 0)
3 {
4 printf("Error en el servidor.\n");
5 return (-1);
```

```
6 }
```

Listing 5.7: Creación del hilo de ejecución servidor.

Por otra parte, para enviar mensajes a otros robots ha de usarse la función sendMessage() implementada en RcSocket. Esta función tiene tres parámetros: el primero es el mensaje a enviar (*array* de *char*), el segundo el *host* del robot al que se le va a enviar (*string*) y el tercero el número de puerto (*string*). En el listado de código 5.8 puede verse un ejemplo donde se envía un mensaje que dice hola, al robot con dirección IP Robot_1 y número de puerto 6656.

```
1 mensaje = "hola";
2 host_socket_robot = "Robot_1";
3 puerto_socket_robot = "6656";
4
5 RcSocket::sendMessage(mensaje, host_socket_robot, puerto_socket_robot);
```

Listing 5.8: Envío de mensaje utilizando sockets.

La función servidor implementada se llama serverFunction() y asegura que se reciban todos los mensajes destinados al robot sobre el que se ha implementado. Ahora, es el momento de analizar los mensajes.

En un mismo mensaje se encuentran datos distintos que hay que evaluar por separado. Por lo tanto, hay que dividir el mensaje en segmentos donde cada segmento contenga solo un tipo de información. Esto es lo que se llama en programación parsear un mensaje. Para poder realizarse tiene que haberse definido algún tipo de estructura de mensaje para poder saber dónde se encuentra cada dato.

Estos mensajes constan de un tipo de estructura especial definida por el signo del dolar ($) al principio y al final del mensaje para indicar donde comienza y finaliza y con el signo de punto y coma (;) como separador de las distintas informaciones que hay en el mensaje. En el cuadro 5.2 se encuentra la estructura de los mensajes.

mensaje = $tipo_mensaje;coord_x_objetivo;coord_y_objetivo;robot_id$

Cuadro 5.2: Estructura mensaje.

Este parseo podría realizarse en el mismo servidor después de haber recibido cada mensaje, pero supone un problema importante. El problema consiste en que si mientras se está parseando un mensaje llega otro, este otro no se guardará porque el servidor estará ocupado. Por lo tanto, existe una región crítica aquí que no permite mantener al servidor mucho tiempo ocupado con el análisis de cada mensaje porque podría suponer la pérdida de información. Es por este motivo que el programa servidor lo único que hace es guardar los mensajes.

Entonces, se ha de crear otra función encargada de analizar los mensajes y parsearlos. Para ello, se ha creado otro hilo de ejecución, llamado messageAuctionFunction, cuya función es ir cogiendo los mensajes de la estructura que rellena el servidor y analizarlos, uno a uno, guardando cada dato de cada mensaje en variables globales para así no perder ningún tipo de información. Para poder separar toda la información de cada mensaje se usa una función, llamada parse_message(). Esta función usa la

herramienta strtok de C++ para separar la información debido a que permite establecer que delimitador separa cada dato (punto y coma) y guardar esta información por separado.

5.3 Programa monitor y tipos de mensajes.

Se ha creado un programa llamado monitor cuya función es enviar a los robots las coordenadas objetivos y las de reposo. La comunicación se ha llevado a cabo mediante la utilización del programa proporcionado por el tutor basado en *sockets*. Dicho programa obtiene los puntos y las características de los *sockets* (dirección IP y puerto) de un archivo de configuración como el que se observa en el cuadro 5.3.

```
#Fitxer amb alguns paràmetres
#En funció de la missió en concret que heu de fer servir
aquest fitxer s'haurà de modificar
NumRobots=5

#Tareas
Num_goals=2
Goal1=(8,6)
Goal2=(0,-10)

#Posicion de reposo
Sort0=(7,-7)
Sort1=(0,10)
Sort2=(-5,0)
Sort3=(2,2)
Sort4=(6,0)

#Parametros comunicacion sockets
RemoteServerRobot0=localhost
RemoteServerRobot1=localhost
RemoteServerRobot2=localhost
RemoteServerRobot3=localhost
RemoteServerRobot4=localhost

RemotePortRobot0=8669
RemotePortRobot1=8670
RemotePortRobot2=8671
RemotePortRobot3=8672
RemotePortRobot4=8673
```

Cuadro 5.3: Estructura archivo de configuración.

Primero se envían los puntos objetivos y, a continuación, las coordenadas de la posición de reposo de cada robot. Los tipos de mensajes y su significado pueden verse

en la tabla 5.4.

ID Mensaje	Estructura	Dirección
1	$1;objetivo_x;objetivo_y;robot_id$	Monitor –>Robots
2	$2;reposo_x;reposo_y;robot_id$	Monitor –>Robots
3	$3;objetivo_x;objetivo_y;robot_id;distancia$	Robot –>Resto Robots
4	$4;objetivo_x;objetivo_y;robot_id$	Robot Líder –>Resto Robots
5	$5;objetivo_x;objetivo_y;robot_id$	Líder –>Resto Robots
6	$6;objetivo_x;objetivo_y;robot_id;distancia$	Robots No Líderes–>Robot Líder
7	$7;objetivo_x;objetivo_y;robot_id$	Robot Líder –>Robots No Líderes
9	$9;objetivo_x;objetivo_y;robot_id$	Cualquier dirección.

Cuadro 5.4: Tipos de mensajes.

- El mensaje de tipo 1 lo envía el programa monitor a todos los robots. Su información consiste en las coordenadas de los objetivos. Se envía un mensaje por cada punto.
- El mensaje de tipo 2 lo envía el programa monitor a todos los robots. Su información consiste en las coordenadas de las posiciones de reposo. Se envía un mensaje por cada punto.
- El mensaje de tipo 3 lo envía cada robot al resto. Su infomación consiste en las coordenadas del objetivo al que opta para ser líder.
- El mensaje de tipo 4 lo envía el robot líder de una tarea al resto de robots. Actúa de baliza informativa de su liderazgo sobre una tarea e indica de qué tarea se trata.
- El mensaje de tipo 5 lo envía el robot líder al resto de robots una vez llega a las coordenadas de su objetivo. Consiste en una petición de apoyo y transmite las coordenadas de su objetivo.
- El mensaje de tipo 6 es de los robots no líderes a los líderes que han enviado un mensaje de tipo 5. Se trata de un mensaje mostrando su interés de servir de apoyo y, para ello, envían la distancia a la que se encuentran del objetivo en el que se necesita su ayuda.
- El mensaje de tipo 7 es de los robots líderes que ya hicieron una petición de apoyo hacia algunos de los que le contestaron la petición con un mensaje de tipo 6. Es un mensaje para confirmar que aceptan el apoyo.
- El mensaje de tipo 9 es un mensaje que envia el monitor después de haber enviado todos los mensajes de tipo 1 y de tipo 2 y, también, es un mensaje que envían los robots de apoyo cuando han llegado a su objetivo. Son de carácter informativo indicando que se ha finalizado la tarea que se estaba realizando.

5.4 Algoritmos de subastas

5.4.1 Algoritmo de elección de líder

La función implementada encargada de establecer los líderes de misión se llama leader_algorithm(). Este algoritmo se centra en hacer peticiones al resto de robots para ser líder de una tarea. El motivo de tener preferencia por una tarea u otra es la cercanía: mayor interés por las tareas más cercanas. Por lo tanto, antes de empezar a ejecutar la función de elección de líder se tienen que evaluar los puntos objetivos recibidos por el programa monitor para establecer el orden de preferencia basado en la distancia. Esto se realiza con el programa order_tasks(). Esta función lo que hace es analizar los puntos en base a las coordenadas iniciales del robot y según la distancia rellenar un *array*, llamado task_coords, indexándolos de más cercano a más lejano.

Una vez se tienen los objetivos ordenados se comienza a ejecutar el algoritmo de elección de líder. Éste se define por el hecho de que cada robot cuando entra en él envía un mensaje al resto con las coordenadas del objetivo de mayor preferencia, su ID y la distancia a la que se encuentra de ese objetivo. Este mensaje se genera con la función leader_message() y es de tipo 3 (tabla 5.4).

Una vez se ha enviado el mensaje se quedan escuchando por mensajes de otro robot. Si reciben un mensaje de otro robot y determinan que es del mismo tipo (mensaje de tipo 3) y que las coordenadas del mensaje son las mismas que las que él ha solicitado procede a comparar la distancia al objetivo del otro robot con la suya. Si el otro robot se encuentra más cerca deja de querer ser líder de ese objetivo y procede a optar al siguiente con mayor preferencia. Si le vuelve a pasar que otro robot solicita el mismo y se encuentra más cerca pasa al siguiente. Sigue este procedimiento hasta que encuentra un objetivo o se acaban las tareas.

En caso de que un robot reciba un mensaje de otro, de tipo 3 (tabla 5.4), y que éste haya solicitado las mismas coordenadas que él y se encuentre a la misma distancia se queda el liderazgo el robot con menor ID.

La última situación posible es que la tarea ya tenga líder. En este caso pasaría a solicitar la siguiente tarea al recibir el mensaje de tipo 4 (tabla 5.4).

5.4.2 Algoritmo de elección de apoyo del líder

Este algoritmo se ejecuta de dos formas distintas dependiendo de si el robot es líder o no. Es decir, dependiendo de si actúa como subastador o como pujador.

Una vez decididos los líderes de las tareas, éstos se encaminan al objetivo y el resto se quedan esperando en su posición de inicio un mensaje de solicitud de apoyo. Se mantendrán esperando hasta que los líderes lleguen a su objetivo. Éste el momento en el que se empieza a ejecutar el algoritmo de elección de apoyo del líder. Esta función se divide en dos partes: una que es la que ejecuta el líder y otra que la ejecutan los no líderes.

El funcionamiento del algoritmo se basa en que cuando el robot líder o subastador llega al objetivo envía un mensaje de tipo 5 (tabla 5.4) a todos los robots donde indica las coordenadas del objetivo y su ID. Este tipo mensaje es el que están esperando los robots pujadores. Se encarga la función auction_petition() de construir el mensaje y enviarlo. Al enviarlo el líder se queda un periodo de tiempo esperando alguna respuesta.

Los pujadores recibirán la solicitud de apoyo y contestan al robot emisor con un mensaje de tipo 6 donde envían las coordenadas del objetivo del líder, su ID y la distancia a la que se encuentra de ellas mediante la función request_auction(). Este mensaje es su puja para participar en la tarea. Una vez enviado se quedan durante un intervalo de tiempo a la espera de la confirmación de apoyo por parte del subastador.

El líder que estaba esperando los mensajes de apoyo en respuestas a su solicitud evalúa las respuesta recibidas. El primer paso es determinar que estos mensajes van dirigidos hacia él. Así que mira que los mensajes sean del tipo correcto, tipo 6 (tabla 5.4), y si lo son es seguro que son para él, ya que los mensajes de este tipo son de respuesta, por lo tanto, solo los recibe el robot que solicitó ayuda. Si éste es el caso, guarda el ID de los robots que han pujado y la distancia a la que se encuentran.

Una vez analizado todos los mensajes, el líder, actuando como subastador, ha de determinar a qué robots les confirmará el apoyo. Para ello, se ha de cumplir una condición que no permite al líder recibir ayuda de más de un número determinado de robots. Si han respondido el mismo número o menos robots que su límite no hay ningún problema, ya que basta con confirmar a todos los robots. Pero si hay más robots hay que decidir cuáles serán los robots que le proporcionen apoyo. La decisión de qué robots serán los elegidos se hace en función de la puja, la cual depende de la cercanía de éstos al objetivo. Por tanto, se debe ir enviando el mensaje de confirmación del más cercano al más lejano hasta que se haya llegado al límite de robots de apoyo. Para realizar la confirmación se envía un mensaje de tipo 7 (tabla 5.4) a los robots seleccionados. La función encargada de este mensaje es accept_message().

Los robots no líderes que estaban esperando la confirmación evalúan los mensajes recibidos. Si han recibido algún mensaje de confirmación, mensaje de tipo 7 (tabla 5.4), comienzan su camino hasta el objetivo. En caso de que no hayan recibido nada. Vuelven a entrar en espera de alguna otra petición de ayuda.

Los robots líderes se mantienen esperando que lleguen los robots de apoyo. Y cuando estos llegan se da la tarea por cumplida y todos vuelven a sus posiciones de reposo.

CAPÍTULO 6

PRUEBAS EXPERIMENTALES

En este capítulo se tratarán todos los aspectos que se han tenido en cuenta en la realización de las pruebas en el simulador Stage y en el robot real. Además, se explican las modificaciones realizadas para adaptar el programa al robot real Pioneer 3DX.

6.1 Pruebas simuladas

6.1.1 Creación de un mapa en Stage

Stage es un simulador habitual en el desarrollo de aplicaciones en robótica por la facilidad que ofrece de creación y manipulación de mapas.

El mapa se construye mediante un archivo world. En este archivo se implementa todo aquello que se quiera que aparezca en el mapa que ha de cargar el simulador. Para ello, hay infinidad de librerías y una estructura de creación de mapas bien definida por el soporte de Stage. Estas características y el hecho de que exista una comunidad detrás que, continuamente, sube paquetes y librerías nuevas agregando nuevos elementos permite simplificar el trabajo de la creación de mapas complejos. Además, Stage dispone de la capacidad de poder cargar archivos .png y usarlos como elemento, objeto o como un mapa en sí. Esto permite que puedas crear tu propio mapa con un programa de dibujo.

El mapa utilizado en las pruebas consiste en un círculo con obstáculos cuadrados de color negro distribuidos de manera aleatoria por toda su área y unas zonas verdes que son las coordenadas de las misiones a realizar. Éste puede verse en la figura 6.1.

Los robots utilizados son los Pioneer 3DX, pero como el simulador funciona en dos dimensiones la librería los llama 2DX. Estos se encuentran implementados en una librería llamada pioneer.inc. En la figura 6.2 se aprecia como representa el simulador a estos robots. Se puede ver como tienen los sensores de ultrasonidos (triángulos verdes) y el sensor láser que es el semicírculo azul en la zona delantera del Pioneer.

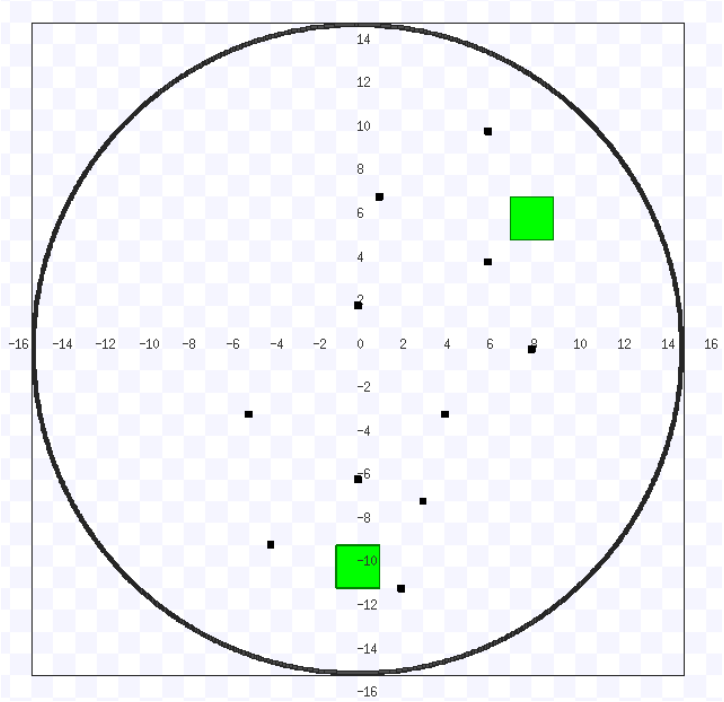

Figura 6.1: Mapa creado para las pruebas.

6.1.2 Realización de una misión en Stage

La misión de las pruebas simuladas consiste en dos tareas objetivo y 5 robots. Primero se asigna un líder para cada tarea aplicando el algoritmo de asignación de líderes. Una vez decidido el líder de cada objetivo comienza a navegar cada uno hacia el suyo. Cuando llegan al objetivo solicitan, apoyo al resto de robots, y aplican el algoritmo de selección de apoyos para determinar qué robots les prestarán ayuda. En el momento que los apoyos han sido confirmados éstos se dirigen hacia las coordenadas de la tarea en la que participan. Una vez llegan todos los apoyos se da la misión por finalizada con éxito y se dirige cada robot a su posición de reposo.

Para la realización de la misión se ha distribuido los Pioneer por el mapa tal como se ve en la figura 6.3. Se ha creado cada Pioneer con un color distinto para facilitar diferenciarlos y cada uno tiene un nombre distinto: el azul se llama robot_0, el rojo robot_1, el lila robot_2, el amarillo robot_3 y el rosa robot_4.

Para llevar a cabo las pruebas se carga en cada robot el programa. Estos se mantendrán en espera hasta que el programa monitor no les haya enviado los puntos objetivos y los puntos de reposo. Una vez les ha enviado los puntos comienza la misión.

Primero se determina quiénes serán los líderes de las dos tareas, como ya se ha comentado. Como son solo dos únicamente habrá dos líderes y el resto se mantendrán a la espera para apoyar cuando se les solicite. Los robots líderes serán los más cercanos. En esté caso son los robots: rosa (robot_4) y azul (robot_0).

Los líderes se ponen en marcha y comienzan a dirigirse cada uno hacia su objetivos mientras van esquivando los obstáculos por el camino. En la figura 6.4 puede verse como ambos robots se dirigen a su objetivo. Y en la 6.5 se aprecia a los líderes en los

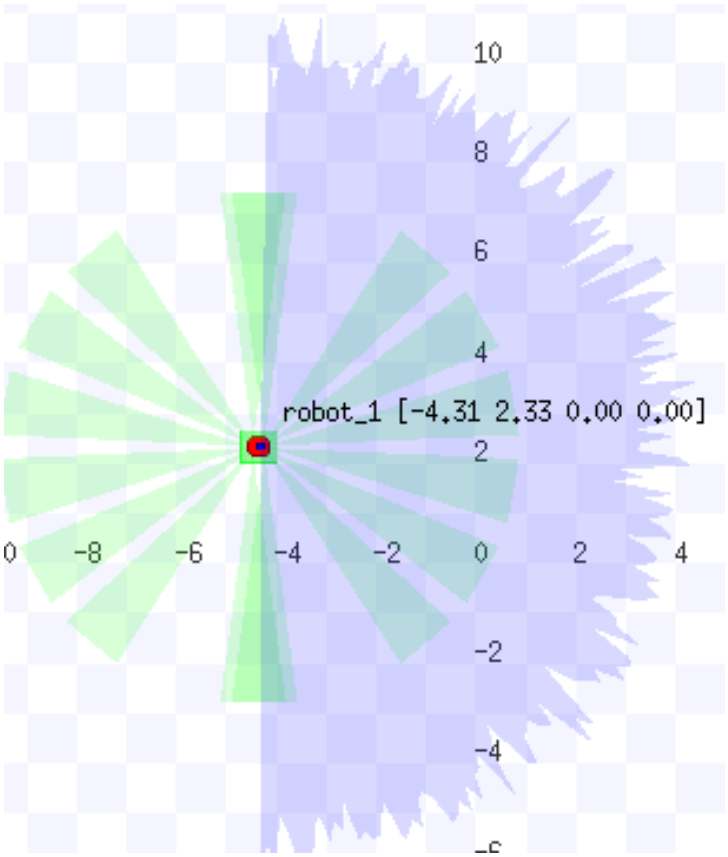

Figura 6.2: Pioneer 3DX en Stage.

objetivos.

El robot_4 (rosa) llegará antes a su objetivo, ya que se encuentra más cerca que el robot_0 (azul) y, además, se encuentra mejor orientado y tiene que rotar menos. Esto significa que será el primero en pedir apoyo.

Cuando llega el robot rosa a su objetivo envía una solicitud de apoyo a todos los robots. El robot azul ignora el mensaje, ya que él ya es líder. Los robots que responde son el amarillo, rojo y lila, es decir, los que están en espera y son pujadores.

El número máximo de robots de apoyo que puede haber se ha establecido en 2. Por lo tanto, de las tres respuestas que ha recibido el rosa, actuando como subastador, solo podrá elegir a dos. Estos dos son los más cercanos (puja mayor) y en este caso se trata del rojo y del amarillo. Es decir, robot_4 (rosa) solo confirma a estos dos para que le apoyen.

En el instante que los robots de apoyo se percatan de que han recibido la aceptación del líder. Se encaminan hacia el objetivo. En la figura 6.5 se aprecia como estos robots se dirigen a apoyarlo. En cambio, el robot lila como no ha recibido la confirmación se queda en su posición esperando otra solicitud. Esta solicitud llega cuando el robot azul llega a su objetivo. Éste no tiene que decidir qué robots pueden apoyarle debido a que solo le responde el robot lila. Los otros robots ya tienen tareas asignadas. El robot lila recibe la confirmación y se dirige al objetivo como se refleja en la figura 6.6.

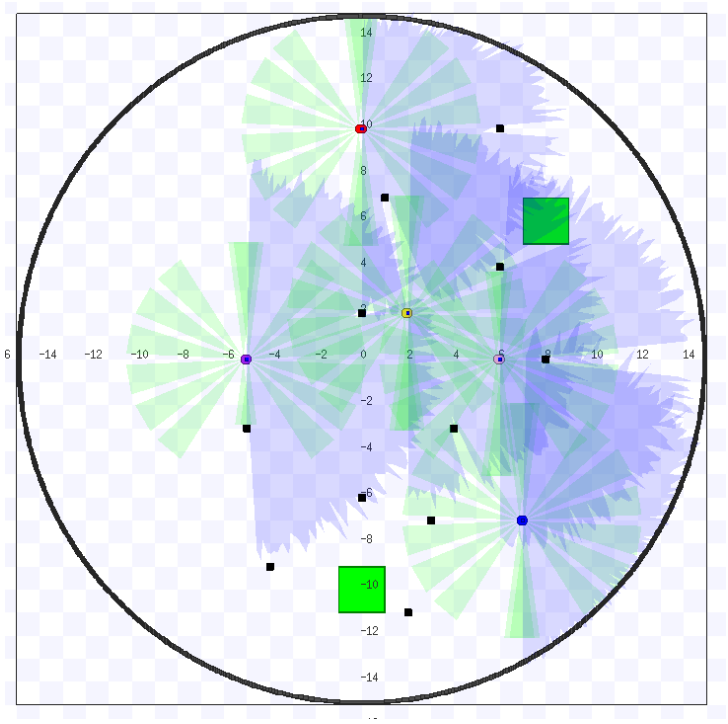

Figura 6.3: Posición de los Pioneer en el mapa.

Cuando llegan los robots de apoyo a las coordenadas se considera la tarea finalizada y se dirige cada uno hacia su posición de reposo, figura 6.7

6.2 Pruebas con el robot real

6.2.1 Preparación del Pioneer 3DX

El primer paso para que el programa funcione en el robot real es instalar Aria. Aria consiste en una serie de librerías que permiten controlar el Pioneer 3DX, pero no se utiliza directamente, sino que se utiliza el nodo que tiene implementado para ROS, llamado ROSAria. ROSAria se trata de un nodo provisto con el fin de interactuar con el *hardware* del Pioneer creando los tópicos necesarios para acceder al control de velocidad y aceleración, odometría y sensores. Por ello, todos los nombres de los tópicos del robot real comienzan por RosAria [13].

Una vez se tiene instalado Aria se procede a comprobar el nombre de los tópicos del Pioneer 3DX para enlazar los publicadores y subscriptores. En la tabla 6.1 se encuentra el nombre de los tópicos de Stage comparado con los del Pioneer. Se aprecia que no coinciden, por lo tanto, hay que modificarlos en el código.

La diferencia principal entre las simulaciones y las pruebas con el robot real se encuentra en los sensores, ya que en el robot real se usan los ultrasonidos y en las simulaciones se ha utilizado un sensor láser. Para poder utilizar los ultrasonidos, primero, hay que subscribirse a su tópico (RosAria/sonar) y, segundo, incluir el programa de ROS capaz de gestionarlos. Esta clase se llama PointCloud. Esta clase se ocupa de todas

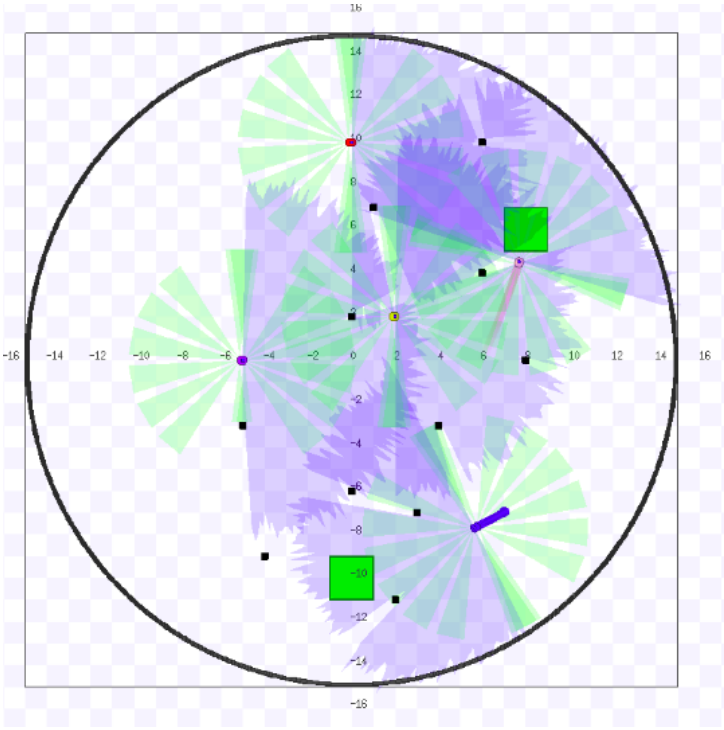

Figura 6.4: Pioneer dirigiénose al objetivo.

Tipo	Stage	Pioneer 3DX
Velocidad	nombre_del_robot/cmd_vel	RosAria/cmd_vel
Odometría	nombre_del_robot/cmd_vel	RosAria/pose
Sensores	nombre_del_robot	RosAria/sonar

Cuadro 6.1: Conversión tópicos Stage a tópicos del Pioneer 3DX.

las funciones relacionadas con los ultrasonidos y consta de una estructura, llamada points, donde va guardando los obstáculos que detecta cada uno de los sensores en forma de coordenadas x e y.

Estas coordenadas de los obstáculos son las que permitirán construir los vectores de repulsión que en Stage se generaba con la información que proporcionaba el sensor láser. Sin embargo, no pueden usarse tal como se encuentran debido a que el SDC del robot no es el mismo que el de Stage toma por defecto. Por lo tanto, se ha de rotar el SDC del Pioneer debido a que el programa está implementado en función del SDC del simulador. En la figura 6.8 puede observarse la comparación entre ambos sistemas de coordenadas.

Para rotar el SDC del Pioneer 3DX para que coincida con el del simulador Stage se ha de rotar 90 grados en sentido horario el sistema de coordenadas del robot real. Para conseguir esta transformación se ha aplicado las ecuaciones de la matriz de rotación 6.2. Las rotaciones de un SDC se rigen por la fórmula 6.3, la cual, desarrollada, da lugar a las ecuaciones 6.1. Al tener que rotar 90 grados el resultado de la transformación será

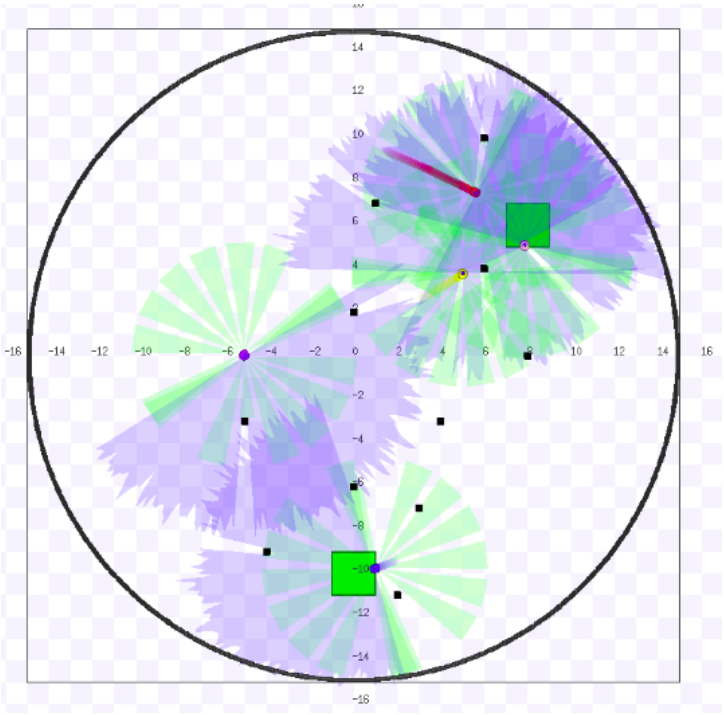

Figura 6.5: Pioneer en sus respectivos objetivos y apoyo del robot rosa en camino.

$$R(\theta) = \begin{bmatrix} Cos\theta & -Sin\theta \\ Sin\theta & Cos\theta \end{bmatrix}$$

Cuadro 6.2: Matriz de rotación.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} Cos\theta & -Sin\theta \\ Sin\theta & Cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Cuadro 6.3: Fórmula rotación vectores columna.

que el valor de la x según el sistema de referencia de Stage será la y negativa del sistema del Pioneer y la coordenada y según Stage será la x del sistema de coordenadas del robot.

$$\begin{aligned} x' &= xCos\theta - ySin\theta \\ y' &= xSin\theta - yCos\theta \end{aligned} \tag{6.1}$$

En el listado de código 6.1 se aprecia como se realiza esta rotación del SDC referenciando las coordenadas de los obstáculos (estructura points) respecto al SDC de Stage.

```
1 x = (-1)*(scan -> points[i].y);
2 y = scan -> points[i].x;
```

Listing 6.1: Rotación SDC del robot par que coincida con el SDC de Stage.

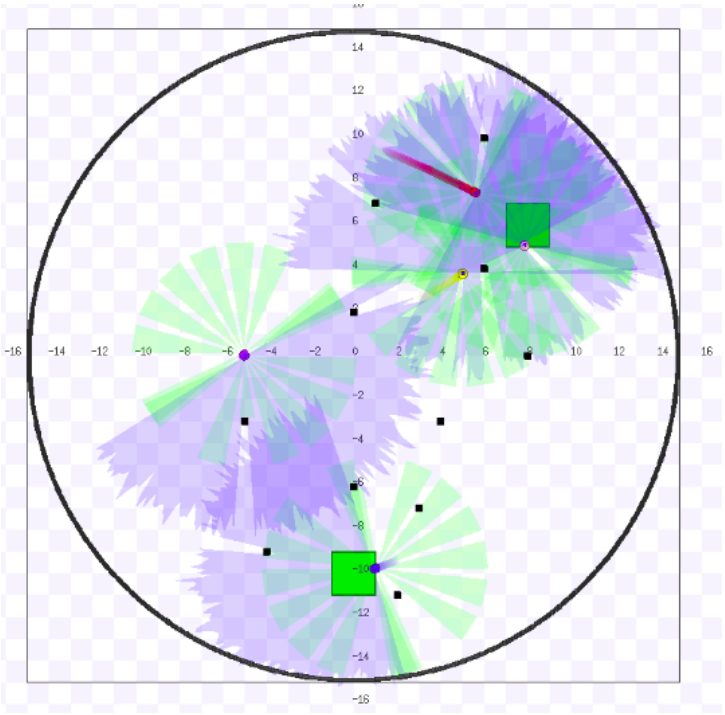

Figura 6.6: Robot lila apoyando a líder.

En este instante las coordenadas de los obstáculos ya se encuentran correctamente referenciadas, pero el programa está implementado de tal forma que se espera recibir ángulos, no coordenadas. A causa de este hecho, se ha de calcular el ángulo de cada vector aplicando el arcotangente (5.5). Los resultados para los ángulos de los sensores 5,6,7 y 8 corresponden a ángulos positivos del primer cuadrante, como corresponde. En cambio, los sensores del 1 al 4 dan ángulos negativos que se encuentran en el cuarto cuadrante y deberían ser del segundo. Por esta razón debe de trasladarse al cuadrante correspondiente sumándole Pi.

Después de trasladar el ángulo al cuadrante correspondiente ya se dispone de la misma información que proporcionaba el sensor láser del simulador. Por lo tanto, se puede aplicar el sistema de cálculo de vectores implementado. Sin embargo, siguen existiendo más factores a tener en cuenta para las pruebas con el robot real. Éstos se deben al hecho de que en un simulador todas las condiciones son perfectas, pero la realidad no es así.

Se ha tenido que aumentar la distancia de detección de obstáculos a 1 metro debido a que el Pioneer no reacciona a la misma velocidad que el robot del simulador. Además, cualquier pequeño desperfecto que tenga el Pioneer 3DX afecta, como podría ser ruedas deshinchadas, batería baja, etc. El entorno es otro factor importante que puede influir en el funcionamiento. Por ejemplo, un suelo que tenga pendiente, baches o que haya viento. Todos éstos son factores a tener en cuenta en el robot real que no aparecían en las simulaciones.

Otro cambio ha sido el de los pesos, ya que se ha tenido que aumentar el peso vector de los obstáculos para aumentar la reactividad debido a que no es tan rápido, ni

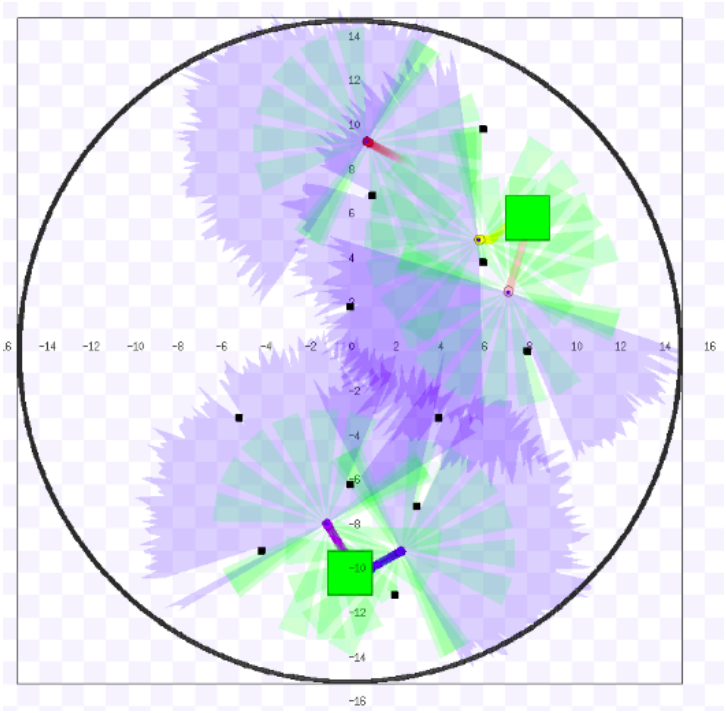

Figura 6.7: Misión realizada con éxito.

preciso como el robot simulado. Aumentando los pesos se evita que pueda acercarse tanto a los obstáculos.

El factor velocidad ha tenido que modificarse para que el Pioneer fuese más lento y así asegurarse su seguridad. Esto ha hecho que en las pruebas el robot no se mueva con demasiada celeridad, pero, como son pruebas, ha de establecerse un modo de funcionamiento seguro hasta que esté bien testeado y se confirme su fiabilidad.

Las pruebas tienen que hacerse en espacios abiertos. Si el robot se encuentra en habitaciones pequeñas con las paredes muy cerca puede suponer un problema de çrosstalk path". Esto significa que la señal del ultrasonido no se refleja en dirección y sentido hacia el sensor, sino que se refleja en otro ángulo distinto. Esto puede suponer que la señal nunca llegue al sensor y no sepa que hay un obstáculo o que llegue demasiado tarde y piense que el obstáculo se encuentra más lejano de lo que está en realidad. O, incluso, que esta señal la reciba otro sensor. En la figura 6.9 puede observarse como se producen estos errores [14].

6.2.2 Prueba de evitación de obstáculos

La prueba realizada de evitación de obstáculos consiste en hacer ir a un robot a un objetivo encontrándose impedimentos por el camino que debe sortear y una vez llegue al objetivo volver a su posición de reposo de la misma manera.

El robot comienza en las coordenadas (0,0) con una orientación de 0 grados. El programa se encuentra ejecutándose a la espera de que el monitor le envíe las coordenadas del objetivo y las de reposo.

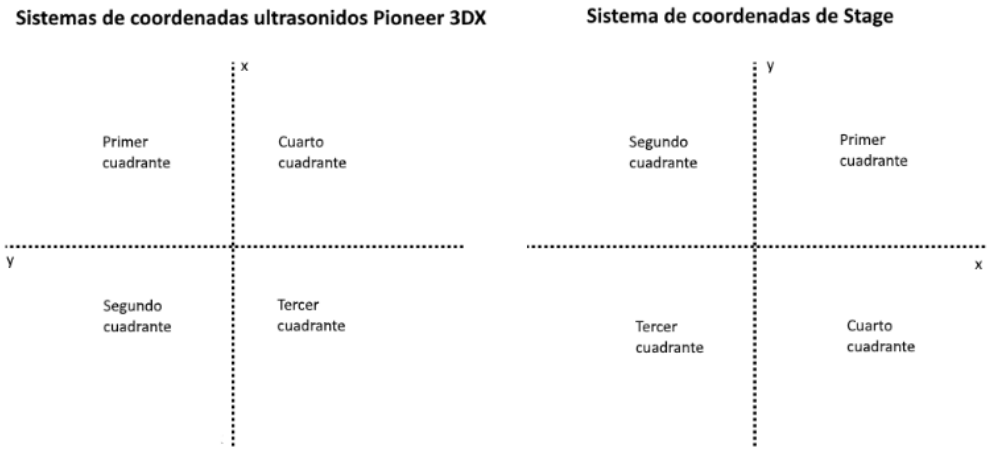

Figura 6.8: Comparación entre el sistema de coordenadas del Pioneer 3DX y el del simulador Stage.

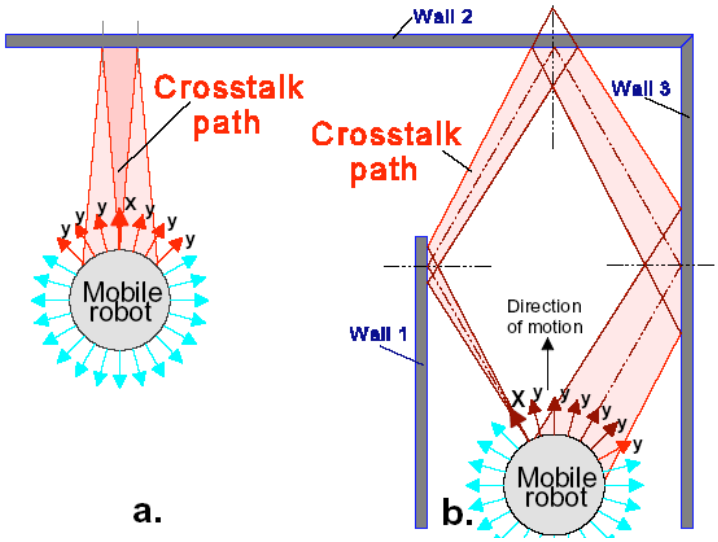

Figura 6.9: Errores de crosstalk path [14].

Una vez recibe los puntos del monitor calcula la ruta y se orienta hacia ella. El objetivo se encuentra en las coordenadas (0,5). Por lo tanto, el Pioneer tiene que rotar 90 grados en sentido antihorario para encontrarse mirando a estas coordenadas.

Una vez ha rotado avanza. Por el camino se encuentra varias cajas de cartón que tiene que ir esquivando como si fuesen obstáculos. En las figuras 6.10, 6.11 y 6.12 se aprecia como el robot se dirige al punto objetivo y va esquivando los obstáculos para llegar a él.

Una vez llega al objetivo la misión se habrá cumplido y se dirige a las coordenadas de reposo. En la figura 6.13 el robot ya se encuentra en las coordenadas (0,5) y en la figura 6.14 el Pioneer se reorienta para ir hacia las coordenadas de reposo, las cuales en

Figura 6.10: Pioneer 3DX orientado y dirigiéndose al objetivo.

este caso son (0,0).

A la vuelta vuelve a esquivar una serie de obstáculos. Al llegar a las coordenadas de reposo el programa termina de ejecutarse y se considera que la misión se ha realizado con éxito. En la figura 6.15 el robot ya ha alcanzado las coordenadas de reposo y ha finalizado su misión.

6.2.3 Prueba de los algoritmos de subastas

Esta prueba no ha conseguido llevarse a cabo con el robot real debido al hecho de que no se consiguió crear un entorno capaz de permitir la comunicación entre robots. Se intentó llevar a cabo utilizando un hub, pero no se consiguió redireccionar los mensajes para que llegasen a su destino.

Sin embargo, cada robot por separado funcionaba y el sistema de sockets también, ya que el programa monitor conseguía enviar los mensajes al robot en el que se ejecutaba. El problema surgía en la comunicación entre robots.

Figura 6.11: Pioneer 3DX encarando un obstáculo.

Figura 6.12: Pioneer 3DX esquivando un obstáculo.

Figura 6.13: Pioneer 3DX en el objetivo.

Figura 6.14: Pioneer 3DX dirigiéndose a las coordenadas de reposo.

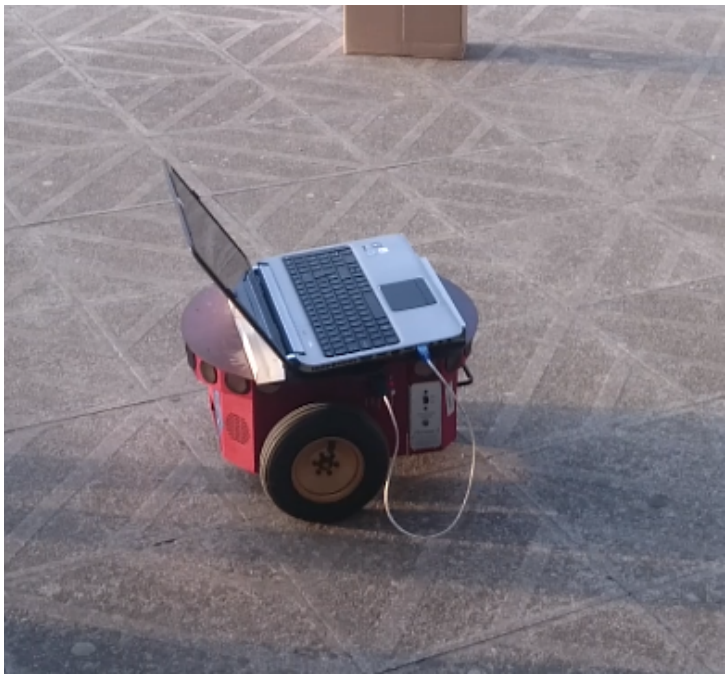

Figura 6.15: Misión completada con éxito.

CAPÍTULO 7

CONCLUSIÓN Y FUTUROS TRABAJOS

7.1 Conclusión

Este proyecto se centra en la implementación de algoritmos de asignación de tareas basados en subastas para sistemas multi-robots. Este objetivo justifica la importancia de los mecanismos de comunicación y asignación de tareas, sin desatender el hecho de que en la robótica móvil es primordial la navegación. Sin embargo, el propósito principal de este trabajo ha sido demostrar cómo funciona la asignación de tareas basados en subastas y la importancia arraigada a este tipo de algoritmos.

Primero se tuvo que establecer un mecanismo que permitiese llegar al robot sin peligro a su objetivo. Se implementó un algoritmo de campos de potencial, el cual es ampliamente utilizado y su diseño es matemático. Esto supuso que el algoritmo en sí no supusiese problemática. La dificultad apareció al tenerse que adaptar este algoritmo a las limitaciones del simulador.

Además, esta problemática se incrementó al tener que adaptar el programa para que funcionase en los robots reales. Los hándicaps de trabajar con robots reales son que no puedes controlar todas las variables que afectan al éxito de la misión. Se puede adecuar el programa para que el funcionamiento sea lo más similar al funcionamiento de la simulación. También se puede intentar paliar estos factores externos al máximo para que afecten en la menor medida. Sin embargo, el estado del *hardware* del robot, las condiciones meteorológicas, las características del entorno, la durabilidad de las baterías, etc. Todos estos factores provocan conflictos, cuyos efectos son muy complicados de predecir y corregir.

Después de muchas pruebas se determinó que el Pioneer 3DX real no conseguía responder a la misma velocidad que se le exigía debido a sus características físicas y a las del entorno. Esto supuso que se redujese la velocidad lineal y de rotación. Se aumentó la distancia a la que se detectan los obstáculos para paliar su menor reactividad. Es decir, su eficiencia disminuyó en las pruebas reales en comparación a la simulación.

Además, el hecho de usar ultrasonidos supone que pueda haber problemas de *crosstalk path* y que se tenga problemas en la localización de los obstáculos. Por lo

tanto, éstos deben encontrarse de tal forma que el rebote de los ultrasonidos lleguen al propio sensor. No puede usarse obstáculos cuyas características supongan una mala reflexión y confundan al robot.

Una vez se consiguió que el robot llegase, sin riesgos, al objetivo se implementaron los algoritmos de comunicación y asignación de tareas por subastas.

Estos algoritmos, también, son ampliamente utilizados y su estructura se encuentra bien definida. La problemática en estos mecanismos surge en el tratamiento de mensajes, regiones críticas y la capacidad de procesamiento del robot. Al trabajarse con varios hilos es difícil establecer todos los casos que pueden producirse y se necesita realizar muchas pruebas para encontrar la manera más eficaz y segura de comunicación. En estos algoritmos también influyen en gran medida factores externos relacionados con *hardware* e interferencias en la señal.

A pesar de todos estos inconvenientes que no se pudieron prever hasta la realización de las pruebas se ha obtenido un programa robusto que no ha necesitado cambios importantes en su estructura para pasarlo al robot real. Los resultados, tanto en las pruebas simuladas como en las pruebas con el Pioneer 3DX, han sido satisfactorios. Se ha tenido problemas en la comunicación con el robot real que han impedido la comunicación entre los robots, pero se ha debido a factores externos a los implementados en este proyecto.

7.2 Trabajos futuros

Este proyecto se define por dos tipo de algoritmos: evitación de obstáculos y asignación de tareas basadas en subastas. En ambas ramas se puede realizar mejoras, añadir nuevas funciones y tratar errores.

Mejora algoritmo de evitación de obstáculo

- Añadir mapeo y planificación al algoritmo de evitación de obstáculos buscando una arquitectura híbrida.
- Añadir funciones de otros algoritmos para mejorar la planificación de la ruta y reducir las limitaciones de los campos de potencial.

Mejora algoritmo de asignación de tareas por subastas.

- Tratamiento de errores y pérdidas de los mensajes.
- Funciones que aseguren el éxito de la misión aunque un robot no responda, reciba la información incorrectamente o se produzca un error de cualquier tipo.

APÉNDICE A

CODIGO C++

A.1 mundo.world

Mapa creado para las pruebas realizadas en el simulador Stage.

```
# Author: David

include "pioneer.inc"
include "map.inc"
include "objects.inc"
include "sick.inc"

window
(
size [ 1000 1000 ]
center [0.000 0.000]
rotate [ 0.000 0.000 ]
scale 55.000
show_data 1 # make sure we can see the effect of the controller
)

floorplan
(
name "rink"
size [30.000 30.000 1.000]
pose [0.000 0.000 0.000 0.000 ]
bitmap "bitmaps/rink.png"
)

floorplan2
(
name "verde"
size [2.000 2.000 0.000]
pose [8.000 6.000 0.000 0.000 ]
color "green"
```

```
)

floorplan2
(
name "verde2"
size [2.000 2.000 0.000]
pose [0.000 -10.000 0.000 0.000 ]
color "green"
)

floorplan
(
name "obstacle1"
size [0.300 0.300 1.000]
pose [8.000 0.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle1"
size [0.300 0.300 1.000]
pose [6.000 4.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle2"
size [0.300 0.300 1.000]
pose [0.000 -6.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle3"
size [0.300 0.300 1.000]
pose [-4.000 -9.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle4"
size [0.300 0.300 1.000]
pose [-5.000 -3.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle5"
size [0.300 0.300 1.000]
pose [4.000 -3.000 0.000 0.000 ]
color "black"
)
```

```
floorplan
(
name "obstacle6"
size [0.300 0.300 1.000]
pose [0.000 2.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle7"
size [0.300 0.300 1.000]
pose [1.000 7.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle8"
size [0.300 0.300 1.000]
pose [6.000 10.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle9"
size [0.300 0.300 1.000]
pose [3.000 -7.000 0.000 0.000 ]
color "black"
)

floorplan
(
name "obstacle9"
size [0.300 0.300 1.000]
pose [2.000 -11.000 0.000 0.000 ]
color "black"
)

pioneer2dx
(
name "robot_0"
pose [ 7.000 -7.000 0.000 0.000 ]
color "blue"
sicklaser
(
# plug the ../examples/ctrl/lasernoise.cc module into this laser
ctrl "lasernoise"

alwayson 1 # don't wait for a subscriber
)
)

pioneer2dx
(
```

```
name "robot_1"
pose [ 0.000 10.000 0.000 0.000 ]
color "red"
sicklaser
(
# plug the ../examples/ctrl/lasernoise.cc module into this laser
ctrl "lasernoise"

alwayson 1 # don't wait for a subscriber
)
)

pioneer2dx
(
name "robot_2"
pose [ -5.000 0.000 0.000 0.000 ]
color "purple"
sicklaser
(
# plug the ../examples/ctrl/lasernoise.cc module into this laser
ctrl "lasernoise"

alwayson 1 # don't wait for a subscriber
)
)

pioneer2dx
(
name "robot_3"
pose [ 2.000 2.000 0.000 0.000 ]
color "yellow"
sicklaser
(
# plug the ../examples/ctrl/lasernoise.cc module into this laser
ctrl "lasernoise"

alwayson 1 # don't wait for a subscriber
)
)

pioneer2dx
(
name "robot_4"
pose [ 6.000 0.000 0.000 0.000 ]
color "pink"
sicklaser
(
# plug the ../examples/ctrl/lasernoise.cc module into this laser
ctrl "lasernoise"

alwayson 1 # don't wait for a subscriber
)
)
```

A.2 robot_parameters.txt

Archivo de configuracion con los datos necesarios para el funcionamiento de la navegacion, evitacion de obstaculos de los robots y los algoritmos de subastas.

```
#Robots
NumRobots=5
MaxNumRobotsAuction=2
LinealSpeed=0.2
AngularSpeed=5
AngleSensor=22
RangeLaser=180
Dmin=1.0
Threshold=0.05
DminGoal=1.0

#Pesos dels comportaments
Wgoal=1.0
Wobstacle=3.0

#Parametros comunicacion sockets
RemoteServerRobot0=localhost
RemoteServerRobot1=localhost
RemoteServerRobot2=localhost
RemoteServerRobot3=localhost
RemoteServerRobot4=localhost

RemotePortRobot0=8669
RemotePortRobot1=8670
RemotePortRobot2=8671
RemotePortRobot3=8672
RemotePortRobot4=8673
```

A.3 communication_parameterts.txt

Archivo de configuracion con los datos de los sockets (direcciones IP y puertos) y las coordenadas de las tareas y las de reposo.

```
NumRobots=5

#Tareas
Num_goals=2
Goal1=(8,6)
Goal2=(0,-10)

#Posicion de reposo
Sort0=(7,-7)
Sort1=(0,10)
Sort2=(-5,0)
Sort3=(2,2)
Sort4=(6,0)
```

```
#Parametros comunicacion sockets
RemoteServerRobot0=localhost
RemoteServerRobot1=localhost
RemoteServerRobot2=localhost
RemoteServerRobot3=localhost
RemoteServerRobot4=localhost

RemotePortRobot0=8669
RemotePortRobot1=8670
RemotePortRobot2=8671
RemotePortRobot3=8672
RemotePortRobot4=8673
```

A.4 RcSocket.cc

Libreria del siistema de sockets otorgado por el tutor del proyecto.

```
/*
Library to manage Sockets
Original Source Code: Ignasi Furio

ArSocketLab.cc
Creation Date : 18/08/2005
Modifications: -
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "RcSocket.h"

#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
#endif

u_short portbase = 0;

//extern int errno;
//extern char *sys_errlist[];

int RcSocket::sock;
pthread_mutex_t RcSocket::mutexSock;

RcSocket::RcSocket() {}

/*
_____________________________________________________
```

```
convert a string to  a IP address
------------------------------------------------------------
*/
u_long RcSocket::inet_addr(char *hostIP) {
char a1, a2, a3, a4;
if (sscanf(hostIP, "%c.%c.%c.%c", &a1, &a2, &a3, &a4) == 4) {
return a1 << 3 + a2 << 1 + a3 << 1 + a4;
} else return INADDR_NONE;
}

/*------------------------------------------------------------------
*  connectsock - assigna i connecta un socket usant UDP o TCP  (CLIENT)
* ------------------------------------------------------------------
*/

int RcSocket::connectSocket(char *host, char *service, char *protocol) {
struct hostent *phe; //punter a informacio del host
struct servent *pse; //Servei associat al port
struct protoent *ppe;
struct sockaddr_in sin;
int s, type;

bzero((char *) &sin, sizeof(sin));
sin.sin_family =  AF_INET;

if (pse = getservbyname(service, protocol))
sin.sin_port = pse->s_port;
else if ((sin.sin_port = htons((u_short) atoi(service))) == 0)
printf("No es possible accedir al servei \"%s\". \n", service);

if (phe = gethostbyname(host))
bcopy(phe->h_addr, (char *)&sin.sin_addr, phe->h_length);
else if ((sin.sin_addr.s_addr = RcSocket::inet_addr(host)) == INADDR_NONE)
printf("no es possible accedirt al host \"%s\". \n", host);



if ((ppe = getprotobyname(protocol)) == 0)
printf("No es possible accedir al protocol \"%s\". \n", protocol);

if (strcmp(protocol, "udp") == 0)
type = SOCK_DGRAM;
else
type = SOCK_STREAM;

s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
printf("No es pot crear el socket\n");

if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
printf("No es connecta a %s. %s\n", host, service);
else
printf("Socket Connected to %s.%s\n", host, service);
return s;
}

/*------------------------------------------------------------------
```

```
passiveSocket - Crea un socket com a servidor
qlen: longitut maxima de la cua del servidor
------------------------------------------------------------------
*/

int RcSocket::passiveSocket(char *service, char *protocol, int qlen) {
struct servent *pse;
struct protoent *ppe;
struct sockaddr_in sin;
int s, type;

bzero((char *)&sin, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;

if (pse = getservbyname(service, protocol))
sin.sin_port = htons(ntohs((u_short)pse->s_port)+portbase);
else if ((sin.sin_port = htons((u_short) atoi(service))) == 0)
printf("No es possible accedir al servei \"%s\". \n", service);

if ((ppe = getprotobyname(protocol)) == 0)
printf("No es possible accedir al protocol \"%s\". \n", protocol);

if (strcmp(protocol, "udp") == 0)
type = SOCK_DGRAM;
else
type = SOCK_STREAM;

s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
printf("No es pot crear el socket\n");

if (bind(s, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
printf("No es pot bloquejar el port %s \n", service);
return (-1);
}
if ((type == SOCK_STREAM) && (listen(s, qlen) < 0)) {
printf("No es pot esperar pel portr %s \n", service);
return (-1);
}

return s;
}

int RcSocket::connectSocketUDP ()
{
struct sockaddr_in address;
int s;

/* Se abre el socket UDP (DataGRAM) */

if (s = socket(AF_INET, SOCK_DGRAM, 0) == (-1)) {
return -1;
}

/* Se rellena la estructura de datos necesaria para hacer el bind() */
address.sin_family = AF_INET;              /* Socket inet */
```

```
address.sin_addr.s_addr = htonl(INADDR_ANY);     /* Cualquier direccion IP */
address.sin_port = htons(0);                     /* Dejamos que linux eliga el
    servicio */

/* Se hace el bind() */
if (bind (s, (struct sockaddr *)&address, sizeof(s)) == -1) {
//close(s);
return -1;
}

/* Se devuelve el Descriptor */
return s;
}


int RcSocket::fillAddress(char *host, char *port, struct sockaddr_in *sin)
{
struct hostent *hp;
if (sin == NULL) { printf("Sin es null\n"); return -1;}

bzero(sin, sizeof(sin));
sin->sin_family = AF_INET;
//fsin.sin_addr.s_addr = RcSocket::inet_addr("127.0.0.1");
hp = gethostbyname(host);

if (hp == NULL) printf("hp es NULLL->%i, %s\n", h_errno, host);
/*while (hp == NULL) {
hp = gethostbyname("localhost");
printf("Try again\n");
}*/
//if (hp == NULL) printf("hp es NULLL2->%i\n", h_errno);
bcopy((char *)hp->h_addr,
(char *)&(sin->sin_addr),
hp->h_length);
sin->sin_port = htons(atoi(port));
return 0;
}

int RcSocket::fillAddress(struct in_addr addr, unsigned short port, struct
    sockaddr_in *sin)
{
struct hostent *hp;
if (sin == NULL) return -1;
bzero(sin, sizeof(sin));
sin->sin_family = AF_INET;
sin->sin_addr = addr;
sin->sin_port = htons(port);
return 0;
}

bool RcSocket::sendMessage(char *message, char *host, char *port) {
struct sockaddr_in fsin;
//int sock= socket(AF_INET, SOCK_DGRAM, 0);
RcSocket::fillAddress(host, port, &fsin);
int n = sendto(RcSocket::sock, message, strlen(message), 0, (struct sockaddr*)&
    fsin, (socklen_t)sizeof(struct sockaddr_in));
if (n < 0) {
```

```
printf("Error Sendto \n");
return false;
}
return true;
}

bool RcSocket::sendMessage(char *message, struct in_addr addr, unsigned short
    port) {
struct sockaddr_in fsin;
int n;
//int sock= socket(AF_INET, SOCK_DGRAM, 0);

RcSocket::fillAddress(addr, port, &fsin);

pthread_mutex_lock(&RcSocket::mutexSock);
n = sendto(RcSocket::sock, message, strlen(message), 0, (struct sockaddr*)&fsin,
    (socklen_t)sizeof(struct sockaddr_in));
pthread_mutex_unlock(&RcSocket::mutexSock);

if (n < 0) {
printf("Error Sendto \n");
return false;
}
//close(sock);
return true;
}

void RcSocket::initRcSocket() {
RcSocket::sock= socket(AF_INET, SOCK_DGRAM, 0);
pthread_mutex_init(&RcSocket::mutexSock, NULL);
}
```

A.5 monitor.cpp

Programa monitor encargado de enviar las coordenadas objetivo y de reposo a los robots.

```
/*
* monitor.cpp
*
*      Author: David
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <string>
#include <string.h>
#include <fstream>
#include <iostream>
#include <math.h>
#include <strings.h>
#include <sstream>
```

```
#include <algorithm>
#include <iomanip>

/*Comunicacions*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <pthread.h>

/*Robotica*/
#include "RcSocket.h"

using namespace std;

#define FILE_NAME "communication_parameters.txt"

/*
     ********************************************************************************
     */

struct _parameters
{
    int NumRobots; //Numero de robots
    int NumGoals;

    char RemoteServerRobot[40][40];
    char RemotePortRobot[40][40];

    char RemoteServerRobot0[256];
    char RemoteServerRobot1[256];
    char RemotePortRobot0[256];
    char RemotePortRobot1[256];

    double GoalCoords[40][40];
    double SortCoords[40][40];
};

void printParameters(struct _parameters data)
{
    cout << "Parametros del robot: " << endl;
    cout << "\tNumero de robots: " << data.NumRobots << endl;
    cout << "\tNumero de tareas: " << data.NumGoals << endl;
    cout << "\tObjetivo_1: X= " << data.GoalCoords[0][0] <<  " Y= " << data.
        GoalCoords[0][1] << endl;
    cout << "\tObjetivo_2: X= " << data.GoalCoords[1][0] <<  " Y= " << data.
        GoalCoords[1][1] << endl;
    //cout << "\tSort: X= " << data.Sort_x <<  " Y= " << data.Sort_y << endl;
}

struct _parameters robotParams; //Estructura donde se guardan los parametros

unsigned short readRobotConfFile(char fileName[256], struct _parameters *data)
{
    string lineF;
    ifstream dataF(fileName);
```

```
char auxLine[256];
unsigned short i;

if (!dataF.good()) {
cout << "El fitxer: " << fileName << " no es pot obrir\n";
return 1;
}

i=0;
while (getline(dataF,lineF,'\n')) { //Fins el final del fitxer
i++;
remove_if(lineF.begin(), lineF.end(), ::isspace); //S'eliminen els espais en
    blanc

//Linea de comentari
if (sscanf(lineF.c_str(), "#%255s",auxLine))
continue;

//Parametros
if (sscanf(lineF.c_str(), "NumRobots=%i",&data->NumRobots)==1)
continue;

//Objetivos
if (sscanf(lineF.c_str(), "Num_goals=%i",&data->NumGoals)==1)
continue;
if (sscanf(lineF.c_str(), "Goal1=(%lf,%lf)", &data->GoalCoords[0][0], &data->
    GoalCoords[0][1])==2)
continue;
if (sscanf(lineF.c_str(), "Goal2=(%lf,%lf)", &data->GoalCoords[1][0], &data->
    GoalCoords[1][1])==2)
continue;

//Posicion de reposo
if (sscanf(lineF.c_str(), "Sort0=(%lf,%lf)",&data->SortCoords[0][0], &data->
    SortCoords[0][1])==2)
continue;
if (sscanf(lineF.c_str(), "Sort1=(%lf,%lf)",&data->SortCoords[1][0], &data->
    SortCoords[1][1])==2)
continue;
if (sscanf(lineF.c_str(), "Sort2=(%lf,%lf)",&data->SortCoords[2][0], &data->
    SortCoords[2][1])==2)
continue;
if (sscanf(lineF.c_str(), "Sort3=(%lf,%lf)",&data->SortCoords[3][0], &data->
    SortCoords[3][1])==2)
continue;
if (sscanf(lineF.c_str(), "Sort4=(%lf,%lf)",&data->SortCoords[4][0], &data->
    SortCoords[4][1])==2)
continue;

//Sockets
if (sscanf(lineF.c_str(), "RemoteServerRobot0=%255s",&data->RemoteServerRobot
    [0])==1)
continue;
if (sscanf(lineF.c_str(), "RemotePortRobot0=%255s", &data->RemotePortRobot
    [0])==1)
continue;
if (sscanf(lineF.c_str(), "RemoteServerRobot1=%255s",&data->RemoteServerRobot
```

```
        [1])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemotePortRobot1=%255s", &data->RemotePortRobot
        [1])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemoteServerRobot2=%255s",&data->RemoteServerRobot
        [2])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemotePortRobot2=%255s", &data->RemotePortRobot
        [2])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemoteServerRobot3=%255s",&data->RemoteServerRobot
        [3])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemotePortRobot3=%255s", &data->RemotePortRobot
        [3])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemoteServerRobot4=%255s",&data->RemoteServerRobot
        [4])==1)
    continue;
    if (sscanf(lineF.c_str(), "RemotePortRobot4=%255s", &data->RemotePortRobot
        [4])==1)
    continue;

    cerr << "Error de syntaxis a la linea " << i << ": "<< lineF << endl;
    return i;
    }
    return 0;
}

/*************************************************************************/

int main(int argc, char **argv)
{
    char dataFileName[256] = FILE_NAME;

    if (!readRobotConfFile(dataFileName, &robotParams)) {
    printParameters(robotParams);
    } else {
    cout << "No se ha podido leer el fichero." << endl;
    }

    char messageAux[256];
    RcSocket::initRcSocket();

    string message;
    string robot_id;
    string goal_x;
    string goal_y;
    string sort_x;
    string sort_y;
    stringstream convert;
    while(1)
    {
    getchar();

    for(int j = 0;j<robotParams.NumGoals;j++)
```

```
    {
        convert<<robotParams.GoalCoords[j][0];
        goal_x = convert.str();
        convert.str(string());

        convert<<robotParams.GoalCoords[j][1];
        goal_y = convert.str();
        convert.str(string());

        for(int i = 0;i<robotParams.NumRobots;i++)
        {
            convert<<i;
            robot_id = convert.str();
            convert.str(string());

            message = "$1;";
            message += goal_x;
            message += ";";
            message += goal_y;
            message += ";";
            message += robot_id;
            message += "$";

            cout<<endl<<":     "<<message<<endl;

            strcpy(messageAux, message.c_str());

            RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[i],
                robotParams.RemotePortRobot[i]);
        }
    }

        for(int i = 0;i<robotParams.NumRobots;i++)
        {
            convert<<robotParams.SortCoords[i][0];
            sort_x = convert.str();
            convert.str(string());

            convert<<robotParams.SortCoords[i][1];
            sort_y = convert.str();
            convert.str(string());

            convert<<i;
            robot_id = convert.str();
            convert.str(string());

            message = "$2;";
            message += sort_x;
            message += ";";
            message += sort_y;
            message += ";";
            message += robot_id;
            message += "$";

            cout<<endl<<":     "<<message<<endl;

            strcpy(messageAux, message.c_str());
```

```
            RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[i],
                robotParams.RemotePortRobot[i]);
        }

        for(int i = 0;i<robotParams.NumRobots;i++)
        {
            convert<<i;
            robot_id = convert.str();
            convert.str(string());

            message = "$9;";
            message += robot_id;
            message += "$";

            cout<<endl<<":    "<<message<<endl;

            strcpy(messageAux, message.c_str());

            RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[i],
                robotParams.RemotePortRobot[i]);
        }
    }
}
```

A.6 mision.cpp

Programa principal con el cual se han realizado las pruebas.

```
/*
* mision_sockets.cpp
*
*      Author: David
*/

/*GENERAL LIBRARIES*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctime>
#include <unistd.h>
#include <string>
#include <string.h>
#include <strings.h>
#include <fstream>
#include <iostream>
#include <math.h>
#include <sstream>
#include <iomanip>
#include <algorithm>
#include <map>

/*COMMUNICATIONS*/
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <pthread.h>

/*SOCKETS*/
#include "RcSocket.cc"

/*ROS LIBRARIES*/
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "sensor_msgs/LaserScan.h"
#include "nav_msgs/Odometry.h"
#include "std_msgs/String.h"

using namespace std;

#define RC_MAX_BUFFER_MESSAGE 45
#define TAMANY_BUFFER 1024
#define FILE_NAME "robot_parameters.txt"

/*
    ********************************************************************************
    */

/*Move robot*/
void chatterCallback(const nav_msgs::Odometry::ConstPtr& odom);
void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan);

bool direction_of_rotation(double relative_angle);
bool need_rotate(double goal_x, double goal_y);
bool move_goal(double goal_x, double goal_y);

void principal();
void calculate_goal_vector(double coord_goal_x, double coord_goal_y);
void rotate_robot(double angular_speed, double goal_x, double goal_y);
void calculate_final_vector(double goal_x, double goal_y);

double angle_to_rotate(double goal_x, double goal_y);
double distance_between(double coord_x, double coord_y);
double degrees2radians(double angle_in_degrees);

/*Communication*/
void *serverFunction(void *arg);
void *messageAuctionFunction(void *arg);
void parse_message();
void order_tasks();
void leader_message(double coord_x, double coord_y);
void auction_message(double coord_x, double coord_y);
void ping_leader_message(double coord_x, double coord_y);
void task_completed(double coord_x, double coord_y, int robot_id_auction);
void accept_request(double coord_x, double coord_y, int robot_id_auction);
void request_auction(double coord_x, double coord_y, int robot_id_auction);
void auction_petition(double coord_x, double coord_y);

bool leader_auction_algorithm(int task_leader);
```

```
int leader_algorithm();

/*
     ********************************************************************************
     */

const double PI = 3.14159265359;

int robot_id;

bool danger = false;

double current_position_x;
double current_position_y;
double current_angle;
double obstacle_vector_x;
double obstacle_vector_y;
double goal_vector_x;
double goal_vector_y;
double final_vector_x;
double final_vector_y;

double task_coords[40][40];
double sort_x;
double sort_y;

int task_leader;
double petition_leader_task[40][40];
double petition_auction_task[40][40];
double request_auction_task[40][40];
int petition_leader_task_index = 0;
int petition_auction_task_index = 0;
int request_auction_task_index = 0;
int num_tasks = 0;

//Message
int task_id_mes;
int robot_id_mes;

double coord_x_mes;
double coord_y_mes;
double distance_mes;

bool new_message = false;
bool auction_ok = false;

//Publisher
ros::Publisher cmd_vel_pub;
ros::Publisher velocity_publisher;

//Mutex
pthread_mutex_t mutexMessage = PTHREAD_MUTEX_INITIALIZER; //OK
pthread_mutex_t mutexState = PTHREAD_MUTEX_INITIALIZER; //OK

//Message
struct messageType {
sockaddr_in   senderAddr;
```

```
char  message[TAMANY_BUFFER];
};

struct messageType buffMessages[RC_MAX_BUFFER_MESSAGE];
int headMessage = 0;
int queueMessage = 0;
bool monitorFinish = false;
bool auctionFinish = false;
bool auctionRequest = false;

/*
    ********************************************************************************************
    */

struct _parameters
{
int AngleSensor; //Angulo del sensor
int RangeLaser; //Rango del laser
int NumRobots; //Numero de robots
int MaxNumRobotsAuction;

double LinealSpeed; //Velocidad lineal
double AngularSpeed; //Velocidad angular
double Dmin;    //Distancia minima
double GoalDmin; //Distancia minima al objetivo
double Threshold; //Constante theta threshold
double wGoal; //Pes anar cap a objectiu
double wObstacle; //Pes evitar obstacle

char RemoteServerRobot[40][40];
char RemotePortRobot[40][40];
};

void printParameters(struct _parameters data)
{
cout << "Parametros del robot: " << endl;

cout << "\tNumero de robots: " << data.NumRobots << endl;
cout << "\tNumero maximo de robots para ayudar: " << data.MaxNumRobotsAuction <<
    endl;
cout << "\tVelocidad lineal: " << data.LinealSpeed << endl;
cout << "\tVelocidad angular: " << data.AngularSpeed << endl;
cout << "\tAngulo del sensor: " << data.AngleSensor << endl;
cout << "\tRango del laser: " << data.RangeLaser << endl;
cout << "\tDistancia minima para detectar el obstaculo: " << data.Dmin << endl;
cout << "\tDistancia minima al objetivo: " << data.GoalDmin << endl;
cout << "\tConstante threshold: " << data.Threshold << endl;

cout << "\twObjectiu: " << data.wGoal << endl;
cout << "\twObstacle: " << data.wObstacle << endl;
}

struct _parameters robotParams; //Estructura donde se guardan los parametros

unsigned short readRobotConfFile(char fileName[256], struct _parameters *data)
{
string lineF;
```

```
ifstream dataF(fileName);
char auxLine[256];
unsigned short i;

if (!dataF.good()) {
cout << "El fitxer: " << fileName << " no es pot obrir\n";
return 1;
}

i=0;
while (getline(dataF,lineF,'\n')) { //Fins el final del fitxer
i++;
remove_if(lineF.begin(), lineF.end(), ::isspace); //S'eliminen els espais en
    blanc

//Linea de comentari
if (sscanf(lineF.c_str(), "#%255s",auxLine))
continue;

//Parametros
if (sscanf(lineF.c_str(), "NumRobots=%i",&data->NumRobots)==1)
continue;
if (sscanf(lineF.c_str(), "MaxNumRobotsAuction=%i",&data->MaxNumRobotsAuction)
    ==1)
continue;
if (sscanf(lineF.c_str(), "LinealSpeed=%lf",&data->LinealSpeed)==1)
continue;
if (sscanf(lineF.c_str(), "AngularSpeed=%lf",&data->AngularSpeed)==1)
continue;
if (sscanf(lineF.c_str(), "AngleSensor=%i",&data->AngleSensor)==1)
continue;
if (sscanf(lineF.c_str(), "RangeLaser=%i",&data->RangeLaser)==1)
continue;
if (sscanf(lineF.c_str(), "Dmin=%lf",&data->Dmin)==1)
continue;
if (sscanf(lineF.c_str(), "DminGoal=%lf",&data->GoalDmin)==1)
continue;
if (sscanf(lineF.c_str(), "Threshold=%lf",&data->Threshold)==1)
continue;

//Pesos
if (sscanf(lineF.c_str(), "Wgoal=%lf",&data->wGoal)==1)
continue;
if (sscanf(lineF.c_str(), "Wobstacle=%lf",&data->wObstacle)==1)
continue;

//Sockets
if (sscanf(lineF.c_str(), "RemoteServerRobot0=%255s",&data->RemoteServerRobot[0])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemotePortRobot0=%255s", &data->RemotePortRobot[0])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemoteServerRobot1=%255s",&data->RemoteServerRobot[1])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemotePortRobot1=%255s", &data->RemotePortRobot[1])
```

```
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemoteServerRobot2=%255s",&data->RemoteServerRobot[2])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemotePortRobot2=%255s", &data->RemotePortRobot[2])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemoteServerRobot3=%255s",&data->RemoteServerRobot[3])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemotePortRobot3=%255s", &data->RemotePortRobot[3])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemoteServerRobot4=%255s",&data->RemoteServerRobot[4])
    ==1)
continue;
if (sscanf(lineF.c_str(), "RemotePortRobot4=%255s", &data->RemotePortRobot[4])
    ==1)
continue;

cerr << "Error de syntaxis a la linea " << i << ": "<< lineF << endl;
return i;
}
return 0;
}

int main(int argc, char **argv)
{

    char dataFileName[256] = FILE_NAME;

    if (!readRobotConfFile(dataFileName, &robotParams)) {
    printParameters(robotParams);
    } else {
    cout << "No se ha podido leer el fichero." << endl;
    }

    if (argc < 2) {
    ROS_ERROR("Debes especificar el ID del robot.");
    return -1;
    }

    char *id = argv[1];
    robot_id = atoi(id);

    if (robot_id < 0 ) {
    ROS_ERROR("El ID del robot no es correcto.");
    return -1;
    }
    cout<<endl<<"Robot "<<robot_id<<" en movimiento."<<endl;

    // Create a unique node name
    string node_name = "move_robot_";
    node_name += id;

    ros::init(argc, argv, node_name);
```

```
    ros::NodeHandle nh;

    // cmd_vel_topic = "robot_X/cmd_vel"
    string cmd_vel_topic_name = "robot_";
    cmd_vel_topic_name += id;
    cmd_vel_topic_name += "/cmd_vel";
    cmd_vel_pub = nh.advertise<geometry_msgs::Twist>(cmd_vel_topic_name, 10);

    // subscribe to robot's laser scan topic "robot_X/base_scan"
    string sonar_scan_topic_name = "robot_";
    sonar_scan_topic_name += id;
    sonar_scan_topic_name += "/base_scan_1";
    ros::Subscriber sub = nh.subscribe(sonar_scan_topic_name, 1, scanCallback);

    // subscribe to robot's laser scan topic "robot_X/odom"
    string odom_topic_name = "robot_";
    odom_topic_name += id;
    odom_topic_name += "/odom";
    ros::Subscriber odom_sub = nh.subscribe(odom_topic_name, 1000,
        chatterCallback);

    pthread_t threadServer, threadMessage;
    RcSocket::initRcSocket();

    //Creacion del thread servidor
    if (pthread_create(&threadServer, (pthread_attr_t*) NULL, serverFunction, (
        char*) NULL) != 0)
    {
    printf("Error en el servidor.\n");
    return (-1);
    }
    //Creacion del thread que analiza los mensajes
    if (pthread_create(&threadMessage, (pthread_attr_t*) NULL,
        messageAuctionFunction, (char*) NULL) != 0)
    {
    printf("Error en el servidor.\n");
    return (-1);
    }

    principal();

    return 0;
}

/*
Funcion: subscriptor al topico de la odometria donde se calcula las coordenadas y
     la orientacion del robot.
Metodos llamados: Ninguno.
Variables globales: current_position_x (double), current_position_y (double),
    current_angle(double).
*/
void chatterCallback(const nav_msgs::Odometry::ConstPtr& odom)
{
    current_position_x = odom->pose.pose.position.x;
    current_position_y = odom->pose.pose.position.y;
    double current_orientation_w = odom->pose.pose.orientation.w;
    double current_orientation_z = odom->pose.pose.orientation.z;
```

```
    current_angle = 2*atan2(odom->pose.pose.orientation.z, odom->pose.pose.
        orientation.w);
}

/*
Funcion: subscriptor al topico del laser donde se calculan los vectores de los
    obstaculos.
Metodos llamados: degrees2radians().
Variables globales: obstacle_vector_x (double), obstacle_vector_y (double).
*/
void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
    double x;
    double y;
    double module = 0;
    double angle = 0;
    double closestRange = 0;
    double obstacle_x = 0;
    double obstacle_y = 0;

    int minIndex = 0;
    int maxIndex = robotParams.AngleSensor;
    int sensor = 0;
    while (minIndex < robotParams.RangeLaser)
    {
        closestRange = scan->ranges[minIndex];
        for (int currIndex = minIndex + 1; currIndex < maxIndex; currIndex++)
        {
            if (scan->ranges[currIndex] < closestRange) {
            closestRange = scan->ranges[currIndex];
            sensor = currIndex;
            }
        }
        if (sensor >= minIndex && sensor <= maxIndex)
        {
            danger = false;
            if (closestRange < robotParams.Dmin && closestRange > 0)
            {
                double angle_sensor;

                angle_sensor = sensor - 90;
                angle_sensor = degrees2radians(angle_sensor);
                angle = current_angle + angle_sensor;

                module = ((robotParams.Dmin - closestRange)/robotParams.Dmin);
                x = cos(angle);
                y = sin(angle);
                x = (-1)*robotParams.wObstacle * x * module;
                y = (-1)*robotParams.wObstacle* y * module;

                obstacle_x += x;
                obstacle_y += y;
            }
        }

        minIndex += robotParams.AngleSensor;
        maxIndex += robotParams.AngleSensor;
```

```
    }
    obstacle_vector_x = obstacle_x;
    obstacle_vector_y = obstacle_y;
}

/*
Funcion: dirigirse hacia el objetivo.
Metodos llamados: distance_between(), need_rotate(), rotate_robot(),
    degrees2radians().
Variables globales: Ninguna
*/
bool move_goal(double goal_x, double goal_y)
{
    geometry_msgs::Twist cmd_vel;

    double distance = 0;

    distance = distance_between(goal_x, goal_y);

    ros::Rate rate(10);
    ros::spinOnce(); // Necesario llamarlo frecuentemente para que que ROS
        procese los mensajes recibidos de los subscriptores
    rate.sleep();
    while (distance > robotParams.GoalDmin)
    {
        if(need_rotate(goal_x, goal_y))
        {
            cmd_vel.linear.x = 0.0;
            cmd_vel_pub.publish(cmd_vel);

            rotate_robot(degrees2radians(robotParams.AngularSpeed), goal_x,
                goal_y);
        }
        else
        {
            cmd_vel.linear.x = robotParams.LinealSpeed;
            cmd_vel.angular.z = 0.0;
            cmd_vel_pub.publish(cmd_vel);    //Publicacion movimiento lineal
        }
        distance = distance_between(goal_x, goal_y);
        ros::spinOnce(); // Necesario llamarlo frecuentemente para que que ROS
            procese los mensajes recibidos de los subscriptores
        rate.sleep();
    }
    cmd_vel.linear.x = 0.0;
    cmd_vel_pub.publish(cmd_vel);
    cout<<endl<<"Objetivo alcanzado."<<endl;
    return true;
}

/*
Funcion: determinar si se debe rotar.
Metodos llamados: angle_to_rotate().
Variables globales: Ninguna.
*/
bool need_rotate(double goal_x, double goal_y)
{
```

```
    double relative_angle = 0;
    double angle_current = current_angle;
    relative_angle = angle_to_rotate(goal_x, goal_y);

    if(angle_current < 0)
    {
        angle_current = abs(current_angle) + PI;
    }
    if(relative_angle < 0)
    {
        relative_angle = abs(relative_angle) + PI;
    }
        if(abs(relative_angle - angle_current) < 0.1)
    {
        return false;
    }
    else
    {
        return true;
    }
}

/*
Funcion: rotar el robot.
Metodos llamados: dangle_to_rotate().
Variables globales: Ninguna
*/
void rotate_robot(double angular_speed, double goal_x, double goal_y)
{
    geometry_msgs::Twist vel_msg;

    //set a random linear velocity in the x-axis
    vel_msg.linear.x =0;
    vel_msg.linear.y =0;
    vel_msg.linear.z =0;

    //set a random angular velocity in the y-axis
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;

    bool clockwise = false;
    double relative_angle = 0;
    relative_angle = angle_to_rotate(goal_x, goal_y);

    clockwise = direction_of_rotation(relative_angle);  //Sentido de giro

    if (clockwise)
        vel_msg.angular.z =-abs(angular_speed);
    else
        vel_msg.angular.z =abs(angular_speed);

    cmd_vel_pub.publish(vel_msg);    //Publicacion movimiento angular
}

/*
Funcion: calcular angulo de rotacion.
Metodos llamados: calculate_final_vector().
```

```
Variables globales: Ninguna
*/
double angle_to_rotate(double goal_x, double goal_y)
{
    calculate_final_vector(goal_x, goal_y); //Vector de la ruta

    double theta;
    theta = atan((final_vector_y)/(final_vector_x));

    if(final_vector_x>0)
    {
        if(final_vector_y<0)
        {
            if(theta>0)
            {
                theta = (-1)*(PI - theta);
            }
        }
    }else
    {
        if(final_vector_y>0)
        {
            if(theta<0)
            {
                theta = PI + theta;

            }
        }
        else
        {
            if(theta>0)
            {
                theta = (-1)*(PI-theta);
            }
            if(final_vector_y==0 && final_vector_x<0)
            {
                theta = -3.14;
            }
        }
    }
    return theta;
}

/*
Funcion: calcular vector final (vector objetivo mas vector obstaculo)
Metodos llamados: calculate_goal_vector().
Variables globales: final_vector_x (double), final_vector_y (double).
*/
void calculate_final_vector(double goal_x, double goal_y)
{
    //Calcular vector objetivo
    calculate_goal_vector (goal_x, goal_y);

    //Calcular vector de la ruta
    final_vector_x = goal_vector_x + obstacle_vector_x;
    final_vector_y = goal_vector_y + obstacle_vector_y;
}
```

```
/*
Funcion: calcular el vector objetivo.
Metodos llamados: distance_between().
Variables globales: goal_vector_x (double), goal_vector_y (double).
*/
void calculate_goal_vector(double goal_x, double goal_y)
{
    double module = 0;

    module = distance_between(goal_x, goal_y);

    goal_vector_x = robotParams.wGoal * (goal_x - current_position_x)/module;
    goal_vector_y = robotParams.wGoal * (goal_y - current_position_y)/module;
}

/*
Funcion: calcular sentido rotacion
Metodos llamados: Ninguna.
Variables globales: Ninguna
*/
bool direction_of_rotation(double relative_angle)
{
    bool clockwise = false;

    if(current_angle>=0)
    {
        if(relative_angle>=0)
        {
            if((relative_angle - current_angle) > 0)
            {
                clockwise = false;
            }
            else
            {
                clockwise = true;
            }

        }
        else
        {
            if(current_angle + abs(relative_angle) > PI)
            {
                clockwise = false;
            }
            else
            {
                clockwise = true;
            }
        }
    }
    else
    {
        if(relative_angle>=0)
        {
            if((relative_angle + abs(current_angle)) > PI)
            {
```

```
                clockwise = true;
            }
            else
            {
                clockwise = false;
            }
        }
        else
        {
            if((abs(relative_angle) - abs(current_angle)) > 0)
            {
                clockwise = true;
            }
            else
            {
                clockwise = false;
            }
        }
    }
    return clockwise;
}

/*
Funcion: calcular distancia entre el robot y otro punto.
Metodos llamados: Ninguna.
Variables globales: Ninguna
*/
double distance_between(double coord_x, double coord_y)
{
    double distance;

    distance = pow((coord_x - current_position_x),2) + pow((coord_y -
        current_position_y),2);
    distance = sqrt(distance);

    return distance;
}

/*
Funcion: transformar grados en radianes.
Metodos llamados: Ninguna.
Variables globales: Ninguna.
*/
double degrees2radians(double angle_in_degrees)
{
    return angle_in_degrees *PI /180.0;
}

/*
Funcion: mision a realizar.
Metodos llamados: order_tasks(), leader_algorithm(), move_goal(),
    leader_auction_algorithm(), request_auction(), task_completed().
Variables globales: task_leader(int), request_auction_task_index (int),
    auctionFinish (bool), sort_x (double), sort_y (double),
auction_request (bool), petition_auction_task_index (int), petition_auction_task
    (matriz), auction_ok (bool)
*/
```

```
void principal()
{
    while(!monitorFinish)
    {
    usleep(200);
    }

    order_tasks();

    task_leader = leader_algorithm();

    if(task_leader < num_tasks)
    {
        ping_leader_message(task_coords[task_leader][0], task_coords[task_leader
            ][1]);
        cout<<endl<<"Lider de la tarea: "<<task_leader<<" X: "<<task_coords[
            task_leader][0]<<" Y: "<<task_coords[task_leader][1]<<endl;

        if(move_goal(task_coords[task_leader][0], task_coords[task_leader][1]))
        {
            if(leader_auction_algorithm(task_leader))
            {
                if(request_auction_task_index > 0)
                {
                    while(!auctionFinish)
                    {
                        usleep(200);
                    }
                }
            }
            if(move_goal(sort_x, sort_y))
            {
                cout<<endl<<"Robot "<<robot_id<<" ha finalizado la tarea "<<
                    task_leader<<"."<<endl;
            }
        }
    }
    else
    {
        cout<<endl<<"Robot "<<robot_id<<" en espera."<<endl;
        int task_auction = 0;

        bool task_end = false;

        double seconds;

        time_t timer;
        struct tm timer_start;
        struct tm timer_end;

            while(!task_end)
            {
                while(!auctionRequest)
                {
                usleep(200);
                }
```

```
                seconds = 0;
                time(&timer);
                timer_start = *localtime(&timer);
                while(seconds < 2)   //Recepcion mensajes de apoyo
                {
                    for(int i=0;i<petition_auction_task_index;i++)
                    {
                        if(petition_auction_task[i][3] < petition_auction_task[
                            task_auction][3])
                        {
                            task_auction = i;
                        }
                    }

                time(&timer);
                timer_end = *localtime(&timer);
                seconds = difftime(mktime(&timer_end), mktime(&timer_start));
            }

            request_auction(petition_auction_task[task_auction][1],
                petition_auction_task[task_auction][2], petition_auction_task[
                task_auction][0]);

            sleep(5);    //Espera aceptacion de participacion en la tarea

            if(auction_ok)
            {
                if(move_goal(petition_auction_task[task_auction][1],
                    petition_auction_task[task_auction][2]))
                {
                    task_completed(petition_auction_task[task_auction][1],
                        petition_auction_task[task_auction][2],
                        petition_auction_task[task_auction][0]);

                    if(move_goal(sort_x, sort_y))
                    {
                        task_end = true;
                        cout<<endl<<"Robot auxiliar "<<robot_id<<" ha finalizado
                            la tarea "<<task_leader<<"."<<endl;
                    }
                }
            }
            auctionRequest = false;
        }
    }
}

/*
Funcion: ordenar los objetivos de mas cercanos a menos.
Metodos llamados: distance_between().
Variables globales: task_coords (matriz).
*/
void order_tasks()
{
double coords[40][40];
    int i;
    int j;
```

```
    int index = num_tasks;

    for(i=0;i<num_tasks;i++)
    {
    coords[i][0] = task_coords[i][0];
    coords[i][1] = task_coords[i][1];
    }

    int new_index;
    int task_index;
    double distance_old;
    double distance_new;
    for(i=0;i<num_tasks;i++)
    {
        distance_old = 0;
        distance_new = 0;
        task_index = 0;
        for(j=0;j<index;j++)
        {
            distance_old = distance_new;
            distance_new = distance_between(coords[j][0], coords[j][1]);
            if(distance_new <= distance_old || index == 1)
            {
                task_index = j;
            }
        }
        task_coords[i][0] = coords[task_index][0];
        task_coords[i][1] = coords[task_index][1];

        new_index = 0;
        for(j=0;j<index;j++)
        {
            if(j != task_index)
            {
                coords[new_index][0] = coords[j][0];
                coords[new_index][1] = coords[j][1];
                new_index++;
            }
        }
        index--;
        cout<<endl;
    }
}

/*
Funcion: eleccion de los lideres de las tareas.
Metodos llamados: leader_message(), distance_between().
Variables globales: petition_leader_task_index (int), petition_leader_task (
    matriz), task_coords (matriz).
*/
int leader_algorithm()
{
    int task_index = 0;
    double distance = 0;
    double seconds = 0;
    time_t timer;
    struct tm timer_start;
```

```
    struct tm timer_end;

    leader_message(task_coords[task_index][0], task_coords[task_index][1]);

    time(&timer);
    timer_start = *localtime(&timer);
    while(seconds < 6 && task_index < num_tasks)
    {
        for(int i=0;i<petition_leader_task_index;i++)
        {
            if(task_index < num_tasks && petition_leader_task[i][1] ==
                task_coords[task_index][0] && petition_leader_task[i][2] ==
                task_coords[task_index][1])
            {
                distance = distance_between(task_coords[task_index][0],
                    task_coords[task_index][1]);
                if(petition_leader_task[i][3] < distance)
                {
                    task_index++;
                    leader_message(task_coords[task_index][0], task_coords[
                        task_index][1]);
                }
                else if(petition_leader_task[i][3] == distance)
                {
                    if(petition_leader_task[i][0] < robot_id)
                    {
                        task_index++;
                        leader_message(task_coords[task_index][0], task_coords[
                            task_index][1]);
                    }
                }
            }
        }

        time(&timer);
        timer_end = *localtime(&timer);
        seconds = difftime(mktime(&timer_end), mktime(&timer_start));
    }
    return task_index;
}

/*
Funcion: eleccion de los apoyos de los lideres en sus tareas.
Metodos llamados: auction_petition(), accept_request().
Variables globales: request_auction_task_index (int), petition_leader_task (
    matriz), task_coords (matriz).
*/
bool leader_auction_algorithm(int task_leader)
{
    double auction_message[40][40];
    int robot_auction[40];

    int index = 0;

    double seconds = 0;

    time_t timer;
```

```
struct tm timer_start;
struct tm timer_end;

auction_petition(task_coords[task_leader][0], task_coords[task_leader][1]);

sleep(5);    //Espera de solicitudes de apoyo

for(int i=0;i<request_auction_task_index;i++)
{
auction_message[i][0] = request_auction_task[i][0];
auction_message[i][1] = request_auction_task[i][1];
}

if(request_auction_task_index > 0)
{
    if(request_auction_task_index <= robotParams.MaxNumRobotsAuction)
    {
        for(int i=0;i<request_auction_task_index;i++)
        {
            cout<<endl<<"Robot "<<auction_message[i][0]<<" va a ayudar."<<
                endl;
            accept_request(task_coords[task_leader][0], task_coords[
                task_leader][1], auction_message[i][0]);

        }
    }
    else
    {
        int i = 0;
        int j = 0;
        int new_index = 0;
        int index = request_auction_task_index;
        int auction_index;
        double distance_old;
        double distance_new;
        for(i=0;i<request_auction_task_index;i++)    //Ordenar robots de apoyo
             segun cercania
        {
            distance_old = 0;
            distance_new = 0;
            auction_index = 0;
            for(j=0;j<index;j++)
            {
                distance_old = distance_new;
                distance_new = auction_message[j][1];
                if(distance_new <= distance_old || index == 1)
                {
                    auction_index = j;
                }
            }
            robot_auction[i] = auction_message[auction_index][0];

            new_index = 0;
            for(j=0;j<index;j++)
            {
                if(j != auction_index)
                {
```

```
                        auction_message[new_index][0] = auction_message[j][0];
                        auction_message[new_index][1] = auction_message[j][1];
                        new_index++;
                    }
                }
                index--;
            }
            for(int i=0;i<robotParams.MaxNumRobotsAuction;i++)
            {
                cout<<endl<<"Robot "<<robot_auction[i]<<" va a ayudar."<<endl;
                accept_request(task_coords[task_leader][0], task_coords[
                    task_leader][1], robot_auction[i]);
            }
        }

        return true;
    }
    else
    {
        cout<<endl<<"No hay robots disponibles para proporcionar ayuda."<<endl;
        return false;
    }
}

/*
Funcion: thread encargado de analizar los mensajes y extraer la informacion que
    contienen.
Metodos llamados: parse_message().
Variables globales: task_coords (matriz), sort_x (double), sort_x (double),
    petition_leader_task (matriz),
petition_leader_task_index (int), petition_auction_task (matriz),
    petition_auction_task_index (int),
auctionRequest (bool), request_auction_task (matriz), request_auction_task_index
    (int), auction_ok (bool),
monitorFinish (bool), auctionFinish (bool).
*/
void *messageAuctionFunction(void *arg)
{
    char messageTypeAux[5];
    char messageAux[256], messageBody[256];
    int num_auction_finish = 0;
    int auxI, messageType;
    struct sockaddr_in senderAddr;
    printf("Entrada a MessageFunction\n");

    while(1)
    {
        if(queueMessage != headMessage)
        {
        pthread_mutex_lock(&mutexState);

        parse_message();
        switch(task_id_mes)
        {
            case 1:
                task_coords[num_tasks][0] = coord_x_mes;
                task_coords[num_tasks][1] = coord_y_mes;
```

```
            num_tasks++;
            break;

        case 2:
            sort_x = coord_x_mes;
            sort_y = coord_y_mes;
            break;

        case 3:
            petition_leader_task[petition_leader_task_index][0] =
                robot_id_mes;
            petition_leader_task[petition_leader_task_index][1] = coord_x_mes
                ;
            petition_leader_task[petition_leader_task_index][2] = coord_y_mes
                ;
            petition_leader_task[petition_leader_task_index][3] =
                distance_mes;
            petition_leader_task_index++;
            break;

        case 4:
            petition_leader_task[petition_leader_task_index][0] =
                robot_id_mes;
            petition_leader_task[petition_leader_task_index][1] = coord_x_mes
                ;
            petition_leader_task[petition_leader_task_index][2] = coord_y_mes
                ;
            petition_leader_task[petition_leader_task_index][1] = 0;
            petition_leader_task_index++;
            break;

        case 5:
            double distance;
            distance = distance_between(coord_x_mes, coord_y_mes);

            petition_auction_task[petition_auction_task_index][0] =
                robot_id_mes;
            petition_auction_task[petition_auction_task_index][1] =
                coord_x_mes;
            petition_auction_task[petition_auction_task_index][2] =
                coord_y_mes;
            petition_auction_task[petition_auction_task_index][3] = distance;
            petition_auction_task_index++;
            auctionRequest = true;  //Algoritmo de subastas iniciado por la
                recepcion de una peticion de apoyo
            break;

        case 6:
            request_auction_task[request_auction_task_index][0] =
                robot_id_mes;
            request_auction_task[request_auction_task_index][1] =
                distance_mes;
            request_auction_task_index++;
            break;

        case 7:
            auction_ok = true;  //Aceptacion de participacion en la tarea
```

```
            break;

            case 9:
                if(!monitorFinish)
                {
                monitorFinish = true;
                }
                else
                {
                    if(request_auction_task_index < robotParams.
                        MaxNumRobotsAuction)
                    {
                        num_auction_finish++;
                        if(num_auction_finish == request_auction_task_index)
                        {
                            auctionFinish =  true;
                        }
                    }
                    else
                    {
                        num_auction_finish++;
                        if(num_auction_finish == robotParams.MaxNumRobotsAuction)
                        {
                            auctionFinish = true;
                        }
                    }
                }
                break;
            }

            pthread_mutex_unlock(&mutexState);
        }
    } // end while(1)
}

/*
Funcion: parsea los mensajes y separa cada dato.
Metodos llamados: Ninguna.
Variables globales: task_id_mes (int), coord_x_mes (double), coord_y_mes (double)
    , robot_id_mes (int), distance_mes (double).
*/
void parse_message()
{
    char messageAux[256];
    int pos_delimiter;

    bzero(messageAux, 256);

    memcpy(messageAux, buffMessages[queueMessage].message, 256);
    //printf("queueMessage=%i %s\n", queueMessage, messageAux);
    queueMessage = (queueMessage + 1) %RC_MAX_BUFFER_MESSAGE;

    char* chars_array = strtok(messageAux, ";$");
    pos_delimiter = 0;
    task_id_mes = 0;
    while(chars_array && task_id_mes != 9)
    {
```

```
        switch (pos_delimiter)
        {
            case 0:
            task_id_mes = atoi(chars_array);
            break;
            case 1:
            coord_x_mes = atof(chars_array);
            break;
            case 2:
            coord_y_mes = atof(chars_array);
            break;
            case 3:
            robot_id_mes = atoi(chars_array);
            break;
            case 4:
            distance_mes = atof(chars_array);
            break;
        }

        chars_array = strtok(NULL, ";$");
        pos_delimiter++;
    }
}

/*
Funcion: servidor encargado de recibir los mensajes y guardarlos.
Metodos llamados: Ninguna.
Variables globales: buffMessages (estructura de tipo mensaje), headMessage (int).
*/
void *serverFunction(void *arg)
{
    struct sockaddr_in fsin;
    char *service = robotParams.RemotePortRobot[robot_id]; //COMMUNICATION_PORT;
    int sock, alen;

    alen = sizeof(fsin);

    if (sock<0)
    {
    printf("Socket Error.\n");
    }
    sock = RcSocket::passiveSocket(service, "udp", 0);
    while (1) {
        bzero(buffMessages[headMessage].message, TAMANY_BUFFER);
        if (recvfrom(sock, buffMessages[headMessage].message, sizeof(buffMessages
            [headMessage].message), 0, (struct sockaddr *)&fsin, (socklen_t*)&
            alen) <= 0)
            printf("Error de recepcion \n");
        else { //Save
            printf("Recibido.\n");
            buffMessages[headMessage].senderAddr = fsin;
            pthread_mutex_lock(&mutexMessage);
            headMessage = (headMessage + 1) %RC_MAX_BUFFER_MESSAGE;
            pthread_mutex_unlock(&mutexMessage);
        }
    }
```

```
}

/*
Funcion: mensaje de tipo 3 de liderazo.
Metodos llamados: distance_between().
Variables globales: Ninguna.
*/
void leader_message(double coord_x, double coord_y)
{
    char messageAux[256];

    string message;
    string robot_id_s;
    string goal_x_s;
    string goal_y_s;
    string distance_s;
    stringstream convert;

    double distance;

    distance = distance_between(coord_x, coord_y);
    convert<<distance;
    distance_s = convert.str();
    convert.str(string());

    convert<<robot_id;
    robot_id_s = convert.str();
    convert.str(string());

    convert<<coord_x;
    goal_x_s = convert.str();
    convert.str(string());

    convert<<coord_y;
    goal_y_s = convert.str();
    convert.str(string());

    message = "$3;";
    message += goal_x_s;
    message += ";";
    message += goal_y_s;
    message += ";";
    message += robot_id_s;
    message += ";";
    message += distance_s;
    message += "$";

    strcpy(messageAux, message.c_str());

    for(int i=0;i<robotParams.NumRobots;i++)
    {
        if(i != robot_id)
        {
            RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[i],
                robotParams.RemotePortRobot[i]);
        }
    }
```

```
}

/*
Funcion: mensaje de tipo 4 de baliza de liderazgo.
Metodos llamados: Ninguna.
Variables globales: Ninguna.
*/
void ping_leader_message(double coord_x, double coord_y)
{
    char messageAux[256];

    string message;
    string robot_id_s;
    string goal_x_s;
    string goal_y_s;
    stringstream convert;

    convert<<robot_id;
    robot_id_s = convert.str();
    convert.str(string());

    convert<<coord_x;
    goal_x_s = convert.str();
    convert.str(string());

    convert<<coord_y;
    goal_y_s = convert.str();
    convert.str(string());

    message = "$4;";
    message += goal_x_s;
    message += ";";
    message += goal_y_s;
    message += ";";
    message += robot_id_s;
    message += "$";

    cout<<endl<<":    "<<message<<endl;

    strcpy(messageAux, message.c_str());

    for(int i=0;i<robotParams.NumRobots;i++)
    {
        if(i != robot_id)
        {
            RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[i],
                robotParams.RemotePortRobot[i]);
        }
    }
}

/*
Funcion: mensaje de tipo 5 de peticion de apoyo por parte del lider.
Metodos llamados: Ninguna.
Variables globales: Ninguna.
*/
void auction_petition(double coord_x, double coord_y)
```

```
{
    char messageAux[256];

    string message;
    string robot_id_s;
    string goal_x_s;
    string goal_y_s;
    stringstream convert;

    convert<<robot_id;
    robot_id_s = convert.str();
    convert.str(string());

    convert<<coord_x;
    goal_x_s = convert.str();
    convert.str(string());

    convert<<coord_y;
    goal_y_s = convert.str();
    convert.str(string());

    message = "$5;";
    message += goal_x_s;
    message += ";";
    message += goal_y_s;
    message += ";";
    message += robot_id_s;
    message += "$";

    cout<<endl<<":    "<<message<<endl;

    strcpy(messageAux, message.c_str());

    for(int i=0;i<robotParams.NumRobots;i++)
    {
        if(i != robot_id)
        {
            RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[i],
                robotParams.RemotePortRobot[i]);
        }
    }
}

/*
Funcion: mensaje de tipo 6 de respuesta al mensaje de tipo 5 por parte de los
    robots no lideres para aceptar la solicitud de apoyo.
Metodos llamados: distance_between().
Variables globales: Ninguna.
*/
void request_auction(double coord_x, double coord_y, int robot_id_auction)
{
    char messageAux[256];

    string message;
    string robot_id_s;
    string goal_x_s;
    string goal_y_s;
```

```
    string distance_s;
    stringstream convert;

    double distance;

    distance = distance_between(coord_x, coord_y);
    convert<<distance;
    distance_s = convert.str();
    convert.str(string());

    convert<<robot_id;
    robot_id_s = convert.str();
    convert.str(string());

    convert<<coord_x;
    goal_x_s = convert.str();
    convert.str(string());

    convert<<coord_y;
    goal_y_s = convert.str();
    convert.str(string());

    message = "$6;";
    message += goal_x_s;
    message += ";";
    message += goal_y_s;
    message += ";";
    message += robot_id_s;
    message += ";";
    message += distance_s;
    message += "$";

    cout<<endl<<":    "<<message<<endl;

    strcpy(messageAux, message.c_str());

    RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[
        robot_id_auction], robotParams.RemotePortRobot[robot_id_auction]);
}

/*
Funcion: mensaje de tipo 7 de aceptacion del apoyo por parte del lider.
Metodos llamados: Ninguna.
Variables globales: Ninguna.
*/
void accept_request(double coord_x, double coord_y, int robot_id_auction)
{
    char messageAux[256];

    string message;
    string robot_id_s;
    string goal_x_s;
    string goal_y_s;
    stringstream convert;

    convert<<robot_id;
    robot_id_s = convert.str();
```

```
    convert.str(string());

    convert<<coord_x;
    goal_x_s = convert.str();
    convert.str(string());

    convert<<coord_y;
    goal_y_s = convert.str();
    convert.str(string());

    message = "$7;";
    message += goal_x_s;
    message += ";";
    message += goal_y_s;
    message += ";";
    message += robot_id_s;
    message += "$";

    cout<<endl<<":    "<<message<<endl;

    strcpy(messageAux, message.c_str());

    RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[
        robot_id_auction], robotParams.RemotePortRobot[robot_id_auction]);
}

/*
Funcion: mensaje de tipo 9 indicando la finalizacion de una tarea.
Metodos llamados: distance_between().
Variables globales: Ninguna.
*/
void task_completed(double coord_x, double coord_y, int robot_id_auction)
{
    char messageAux[256];

    string message;
    string robot_id_s;
    string goal_x_s;
    string goal_y_s;
    stringstream convert;

    convert<<robot_id;
    robot_id_s = convert.str();
    convert.str(string());

    convert<<coord_x;
    goal_x_s = convert.str();
    convert.str(string());

    convert<<coord_y;
    goal_y_s = convert.str();
    convert.str(string());

    message = "$9;";
    message += goal_x_s;
    message += ";";
    message += goal_y_s;
```

```
    message += ";";
    message += robot_id_s;
    message += "$";

    cout<<endl<<":     "<<message<<endl;

    strcpy(messageAux, message.c_str());

    RcSocket::sendMessage(messageAux, robotParams.RemoteServerRobot[
        robot_id_auction], robotParams.RemotePortRobot[robot_id_auction]);
}
```

BIBLIOGRAFÍA

[1] G. O. C. y José Guerrero Sastre, "Multi-robot coalition formation in real-time scenarios," 2012. 1.1

[2] (20 de Agosto de 2017) Pioneer 3dx. [Online]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx 2.1, 2.1.1, 2.1, 2.1.2

[3] (22 de Agosto de 2017) Pioneer 3dx: Entorn de pràctiques. [Online]. Available: http://dmi.uib.es/~jguerrero/PresPioneer.pdf 2.1.3

[4] (22 de Agosto de 2017) Saphira/aria/aros. [Online]. Available: https://www.manualslib.com/manual/130418/Pioneer-2-Peoplebot.html?page=12 2.1.6, 2.2

[5] (24 de Agosto de 2017) Ros wiki. [Online]. Available: http://www.ros.org/history/ 3.1

[6] (24 de Agosto de 2017) Introducción de ros. [Online]. Available: http://erlerobotics.com/blog/ros-introduction-es/ 3.2

[7] (24 de Agosto de 2017) Stage wiki. [Online]. Available: http://playerstage.sourceforge.net/index.php?src=stage 3.3.1

[8] (24 de Agosto de 2017) Stage. [Online]. Available: https://rua.ua.es/dspace/bitstream/10045/3605/1/intro-ps.pdf 3.3.2

[9] (24 de Agosto de 2017) Nodo stageros. [Online]. Available: http://wiki.ros.org/stage_ros 3.3.2

[10] (22 de Agosto de 2017) Sense and act. [Online]. Available: http://www.cs.bham.ac.uk/internal/courses/int-robot/2014/lectures/14-ir-architectures.pdf 4.1

[11] (24 de Agosto de 2017) Cuaterniones y función atan2. [Online]. Available: http://www.wag.caltech.edu/home/ajaramil/libro_robotica/transformaciones_espaciales.pdf 5.1

[12] (24 de Agosto de 2017) Sockets. [Online]. Available: http://sopa.dis.ulpgc.es/ii-dso/leclinux/ipc/sockets/sockets.pdf 5.2

[13] (24 de Agosto de 2017) Rosaria. [Online]. Available: http://wiki.ros.org/ROSARIA 6.2.1

[14] (24 de Agosto de 2017) Crosstalk path. [Online]. Available: http://wiki.robotica.webs.upv.es/wiki-de-robotica/sensores/sensores-proximidad/sensor-de-ultrasonidos/ 6.2.1, 6.9