Universidad Zaragoza

1542

Trabajo Fin de Máster

Diseño e implementación de un algoritmo que evite colisiones en un sistema multi-robot utilizando el Modified Banker's Algorithm.

Design and implementation of algorithm which avoids deadlocks in multi-robot system using Modified Banker's Algorithm.

Autor/es

Jose Benigno García Barreto

Director/es

Cristian Mahulea

Joaquín Ezpeleta

Titulación del autor

Máster en Ingeniería Industrial

Escuela de Ingeniería y Arquitectura

2020

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. Jose Benigno Garcia Barreto ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster) Máster (Título del Trabajo)

Diseño e implementación de un algoritmo que evite colisiones en un sistema multi-robot utilizando el Modified Banker's Algorithm.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 23 de Septiembre de 2020

Fdo:

Diseño e implementación de un algoritmo que evite colisiones en un sistema multi-robot utilizando el Modified Banker's Algorithm

RESUMEN

La planificación y el control de robots móviles es un área de investigación muy extendida, cuyo objetivo de estudio es el movimiento de los robots. En el funcionamiento de un sistema multi-robot hay que diferenciar entre el proceso de planificación de trayectorias (la aplicación con la que se trabaja dispone de una serie de estrategias implementadas como Cell Descomposition, Voronoi, etc.) y el control de ejecución de estas.

El presente Trabajo Fin de Máster pretende utilizar herramientas de Sistemas de eventos discretos (SED), para controlar el movimiento de los robots a lo largo de las trayectorias, obtenidas siguiendo una determinada estrategia de planificación. En este proyecto no se ahonda en el proceso de obtención de las trayectorias, sino en la implementación de un método que evite los posibles problemas que puedan aparecer en un sistema multi-robot: colisiones entre robots y bloqueos del sistema, ya sean parciales o totales.

Partiendo de unas trayectorias predefinidas, se modela el sistema mediante un grafo formado por las regiones del espacio de trabajo recogidas en las trayectorias para evitar colisiones. Dichas regiones presentan capacidad unitaria, es decir, no pueden ser ocupadas por más de un robot al mismo tiempo. El modelo utilizado considera que una trayectoria es una secuencia de recursos que el robot debe ir utilizando secuencialmente, por lo tanto, se trabaja con un sistema que dispone de recursos compartidos. Se entiende por recurso del sistema al arco formado por dos puntos consecutivos de una trayectoria.

La utilización de recursos compartidos genera modos de espera, debido a la competencia por los recursos, que pueden derivar en situaciones de bloqueo. En este proyecto se utiliza un algoritmo de evitación de bloqueos, el algoritmo del banquero, para controlar dichas situaciones. El algoritmo del banquero se ejecuta previamente a un cambio de estado del sistema con el fin de determinar si ese posible futuro estado no deriva en una situación bloqueante.

Los robots con los que se trabajan son de bajo coste, con lo cual inducen un error en su movimiento, por tanto, es posible que haya colisiones con obstáculos y con otros robots, aunque idealmente estas colisiones no deberían darse ya que tanto la planificación de trayectorias como los algoritmos de prevención o de evitación de bloqueos garantizan que estas situaciones estén controladas.

Además de la implementación del algoritmo dentro del control de ejecución de trayectorias, se han añadido funcionalidades y modificaciones en el algoritmo que modela el funcionamiento de cada uno de los robots para conseguir que el movimiento de estos esté monitorizado y sea más preciso. Entre los cambios realizados destaca un rediseño del control interno de los robots, la implementación de una comunicación entre PC y robots, así como la detección de posibles colisiones con obstáculos; el añadir estas funcionalidades ha implicado un rediseño integral de la aplicación, la cual se ha desarrollado siguiendo una programación orientada a objetos mediante el lenguaje C++.

Este Trabajo Fin de Máster puede considerarse una iniciación a la investigación y se confía en que lo conseguido pueda servir de base para futuros trabajos que serán realizados bajo la supervisión del departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza.

Tabla de contenido

Capítulo 1 INTRODUCCION 1
- **1.1. Objetivo del trabajo** 1
- **1.2. Alcance del trabajo** 2
- **1.3. Metodología (Enfoque y herramientas)** 2
- **1.4. Fases de desarrollo** 2

Capítulo 2 ESTADO DEL ARTE 5
- **2.1. Planificación de trayectorias** 7
- **2.2. Control de movimiento** 9
- **2.3. Colisiones en un sistema multi-robot** 9

Capítulo 3 APLICACIÓN MULTI-ROBOT 11
- **3.1. Elementos** 11
- **3.2. Comunicación PC-Robots** 14

Capítulo 4 ALGORITMO DEL BANQUERO 17
- **4.1. Modelo de representación de la plataforma** 18
- **4.2. Aplicación del algoritmo del banquero** 19
- **4.3. Control de ejecución utilizando el algoritmo del banquero** 22
  - **4.3.1. Proceso de control** 23

Capítulo 5 MODIFICACIONES EN LA APLICACIÓN 25
- **5.1. Comunicación bidireccional** 25
- **5.2. Cambios en control interno del robot** 27

Capítulo 6 RESULTADOS Y CONCLUSIONES 29

BIBLIOGRAFIA 33

ANEXOS 35

ANEXO I 37
- **Anexo I.A. Arduino Leonardo** 38
- **Anexo I.B. Antenas XBee** 38
- **Anexo I.C. Puerto serie** 43
- **Anexo I.D. Paquetes de mensajería** 45
  - **Paquetes que se envían desde PC** 45
  - **Paquetes que envían los robots** 47

ANEXO II 49
- **Anexo II.A. Descripción de librería "*BankerAlgorithm*"** 50
  - **Clase *"obstacles"*** 50
  - **Subclases usadas en clase "Syst"** 51
  - **Clase "Syst"** 53

Struct “VertexData” 55
Typedef “Graph_t” 55
Clase “Banker_Graph” 56
Clase “nextState” 56
Anexo II.B. Funciones usadas en control de ejecución 57
getNextStateByBanker(Syst& , std::vector<nextState*>&) 57
control_RobotStateBA(Syst& , std::vector<nextState*>, std::vector<std::vector<double>>) 58
checkStatus(Syst& , std::vector<nextState*>&) 60
sendSubPaths(Syst& , std::vector<nextState*>&) 60
sendPosition(std::vector<std::vector<double>>,Syst&) 61
Funciones utilizadas en el algoritmo de la aplicación 61
Anexo III 63
Anexo III.A. Modelo matemático de los robots 64
Anexo III.B. Control P 65
Anexo III.C. Control interno de los robots 68

Capítulo 1
INTRODUCCION

En este capítulo se aborda una explicación del objetivo y alcance del proyecto, así como la metodología seguida a lo largo del mismo.

1.1. Objetivo del trabajo

El objetivo de este proyecto es asegurar la ejecución concurrente de unas trayectorias, las cuales son asignadas a cada uno de los robots que conforman una aplicación multi-robot. Para ello es necesario modelar e implementar un algoritmo que evite situaciones de bloqueo y colisiones dentro de dicha aplicación; se ha elegido el algoritmo del banquero ya que es una técnica que da buenos resultados en sistemas operativos y que es muy utilizada en el ámbito de la producción.

Se trabaja sobre una aplicación que sirve para estudiar el comportamiento tanto individual como colectivo de un conjunto de robots móviles, a través de un análisis del proceso de navegación de cada robot y un control que garantiza la ejecución concurrente y sincronizada de varios procesos.

Los valores de entrada a la aplicación son los puntos de destino que tienen que alcanzar cada uno de los robots, así como una serie de especificaciones a alto nivel. A partir de dichas entradas y mediante una estrategia de planificación de trayectorias se obtiene una ruta óptima a alto nivel, entendida como una secuencia de puntos dentro del espacio de trabajo de la aplicación, por los cuales cada robot debe circular.

Las rutas obtenidas son discretas y cumplen las especificaciones dadas por el usuario, de forma que utilizando un formalismo matemático discreto se ejecuta un algoritmo de evitación de bloqueos previamente a un posible cambio de estado, de manera que si el estado alcanzado es seguro se puede asegurar que todos los robots completarán la ruta planificada que se le ha sido asignada, mientras que si el estado es no seguro se procede a encontrar un estado alternativo que sí lo sea.

1.2. Alcance del trabajo

Controlar un sistema multi-robot engloba la planificación de trayectorias y la planificación de ejecución de estas. Este proyecto se centra en la planificación y el control de la ejecución de unas trayectorias predefinidas considerando un control con aplicación on-line.

Se considera un control on-line cuando se pueden asignar nuevas tareas durante la ejecución de las trayectorias a pesar de que la planificación ya haya sido hecha; esto se puede entender como que las trayectorias que deben seguir los robots pueden sufrir modificaciones durante la ejecución del control, mientras que en una aplicación off-line las trayectorias deben permanecer invariantes.

Para conseguir acoplar el algoritmo del banquero a la aplicación, ha sido necesario añadirle ciertas funcionalidades como la implementación de una comunicación bidireccional entre robots y PC, y llevar a cabo un rediseño de está utilizando una programación orientada a objetos. Asimismo, se ha hecho una modificación completa del algoritmo de control intrínseco a cada robot.

1.3. Metodología (Enfoque y herramientas)

A la hora de implementar la estrategia de evitación de bloqueos se dispone de una entrada discreta a alto nivel, las trayectorias a ejecutar, por lo que es necesario trabajar con formalismos matemáticos discretos (autómatas de estados finitos o redes de Petri).

Para la implementación del algoritmo se ha seleccionado C++ como lenguaje de programación y se ha usado una librería que permita trabajar con grafos “*Boost Graph Library*”.

Este proyecto sirve como futura herramienta para otros alumnos que trabajarán con temas relacionados a la planificación de sistemas multi-robot dentro del departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza. Es importante dejar claro que existe una línea de trabajo futuro muy interesante ya que a esta aplicación se le pueden añadir estrategias que aborden un mismo problema desde distintas perspectivas con el fin de hacer una comparación y determinar cuál de ellas da mejores resultados. Asimismo, hay que tener en cuenta que los resultados dependen del sistema de localización y del control supervisor de cada robot.

1.4. Fases de desarrollo

Como se puede ver en la propuesta de este proyecto, se han establecido una serie de pasos necesarios para alcanzar resultados tangibles que tengan validez.

- Formación en programación orientada a objetos y en lenguaje de programación C++.
- Estudio de la lógica implementada en la aplicación.
- Estudio del algoritmo del banquero.

- Estudio del algoritmo del banquero a través de RdP.
- Estudio de la comunicación bidireccional mediante antenas XBee.
- Modificaciones implementadas en el controlador interno de los robots.
- Integración dentro de la plataforma existente.
- Comparación y evaluación.

La presente memoria está dividida en seis capítulos en los cuales se pretende plasmar el trabajo realizado para implementar el algoritmo del banquero en un sistema multi-robot.

- En este primer capítulo, se han presentado los objetivos del proyecto.
- En el capítulo 2 se presentan los distintos temas que se abordan dentro del funcionamiento de un sistema multi-robot con el fin de que el lector tenga una idea general del tema abordado.
- Posteriormente, en el capítulo 3 se aborda una explicación de la aplicación que sirve de base para el proyecto.
- En el capítulo 4 se profundiza en el algoritmo del banquero, de forma sencilla y didáctica.
- A continuación, en el capítulo 5 se describen las modificaciones implementadas en la plataforma, las cuales son independientes al algoritmo del banquero.
- Por último, en el capítulo 6 se describen los resultados obtenidos y se recogen las conclusiones y futuras aplicaciones.

Capítulo 2
ESTADO DEL ARTE

Hoy en día, el uso de los robots está muy extendido, encontrando robots en actividades de limpieza, movilización de mercancías dentro de una fábrica, manipulación de utensilios en un quirófano, exploración de terrenos, entre otros.

Al hablar de robots, hay que diferenciar entre un robot-manipulador, que dispone de un rango limitado de movimiento debido a su posición fija, y un robot móvil, el cual puede desplazarse libremente por un entorno. El valor añadido del que dispone un robot móvil debe compaginarse con una adecuada estrategia de navegación, dentro de la cual se pueden distinguir dos enfoques:

- Navegación reactiva.

Un robot reacciona a estímulos que se generan en su entorno de trabajo en tiempo real, llegando a moverse de forma rápida y sencilla si los parámetros detectados en el entorno se conectan directamente a la acción que le da movimiento; dichos parámetros, que determinan el movimiento del robot, se obtienen del entorno de forma que no es necesaria una representación de este.

Este enfoque presenta la desventaja que el movimiento del robot se limita a un espacio del entorno predefinido con anterioridad, aunque es muy utilizada por robots AGVs, los cuales se encargan de realizar rutinas de recogida y entrega en procesos de fabricación o distribución [8,9].

- Navegación planificada.

Estrategia según la cual un robot presenta un razonamiento autónomo que le permite definir una sucesión de movimientos mediante los cuales alcanza un destino. Para ello es necesario un reconocimiento del entorno por donde el robot se puede mover sin ninguna limitación.

Esta técnica se empezó a utilizar en 1970 con la aparición del robot "*Shakey*" [10], el cual era capaz de percibir y crear un mapa de su entorno para después planificar su trayectoria de forma razonada; otro ejemplo es el robot *"Terregator"* [11] que aplicó esta técnica a entornos que estaban al aire libre.

El proceso de navegación típico en robótica se reduce a que un robot alcance un punto de destino desde su posición de partida. Sin embargo, al disponer de especificaciones a alto nivel, se está generalizando dicho proceso de navegación con el fin de que un robot se adapte a un entorno de trabajo en el que coopere con personas.

De esta forma el proceso de navegación de un robot consiste en alcanzar un punto de destino siguiendo unas especificaciones a alto nivel que se entienden como un conjunto de tareas que el robot debe cumplir antes de llegar a su destino; alguna de estas especificaciones puede ser, por ejemplo, que el robot evite los obstáculos a lo largo de su navegación o que pase por unas determinadas regiones del entorno.

La navegación planificada de un robot, como se puede ver en Figura 1, engloba cuatro módulos: una planificación de trayectorias a partir de la cual se define la secuencia de movimientos que debe ejecutar un robot para alcanzar un destino, el control de movimiento según el cual un robot genera unas acciones de control que le permitan seguir de la forma más precisa posible la trayectoria obtenida en el módulo anterior, los controladores intrínsecos al robot que facilitan la transmisión de las acciones de control a los accionadores del motor y por último, un conjunto de sensores que permiten definir la posición actualizada del robot. La precisión de la detección de la posición actual del robot influye considerablemente en el movimiento del robot.

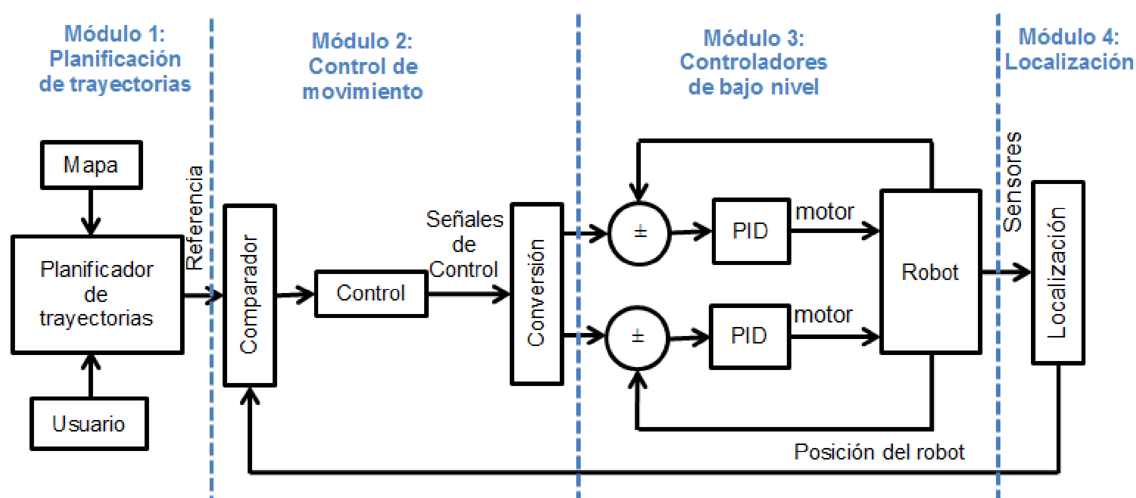

Figura 1: Arquitectura de navegación de un robot [7].

Hay que tener en cuenta que el tiempo de muestreo en cada una de las fases de la navegación de un robot es diferente; por ejemplo, el tiempo para obtener las trayectorias es del orden de minutos o segundos, mientras que el control del movimiento puede tardar segundos, pero definitivamente la fase más rápida es el control interno del robot ya que presenta un tiempo de ejecución del orden de milisegundos.

2.1. Planificación de trayectorias

Módulo dentro del proceso de navegación de un robot que representa el nivel cognitivo del mismo y le permite obtener rutas seguras para alcanzar una posición objetivo; dichas rutas son una secuencia de posiciones dentro del entorno de trabajo que cumplen unas especificaciones de alto nivel.

Como se ha mencionado antes, el robot ”*Shakey*” fue pionero en lo que respecta a la planificación de su movimiento [12] pero se limitaba al cálculo de trayectorias que aseguraban que el robot no colisione con los obstáculos dentro de un entorno plano; las siguientes investigaciones en este ámbito fueron dirigidas a planificar las trayectorias teniendo en cuenta ciertas limitaciones cinemáticas, por ejemplo, considerar al robot un sistema no holonómico lo que significa que un robot tipo coche no puede girar sobre sí mismo sin que haya un cambio de su posición [13].

Los libros [14,15] introdujeron los conceptos de planificación de trayectorias que ahora se utilizan, los cuales se pueden clasificar, como se puede ver en la Figura 2 , en dos grandes grupos: algoritmos combinacionales y algoritmos basados en un muestreo aleatorio [7].

Por un lado, las soluciones basadas en un muestreo aleatorio buscan una rápida respuesta computacional mediante un muestreo pequeño del entorno, el cual se utiliza para llevar a cabo la búsqueda de trayectorias; este tipo de algoritmos son propios de una navegación reactiva, en la que no es necesaria una representación del entorno. Los más utilizados son el algoritmo PRM (“*Probabilistic Roadmap*”) y el algoritmo RRT (“*Rapidily-exploring Random Tree*”), el primero presenta la ventaja de que necesita pocos puntos para determinar si el camino entre un punto inicial y final está libre de obstáculos pero el asumir que un robot tiene capacidad omnidireccional supone un inconveniente, mientras que el algoritmo RRT (“*Rapidily-exploring Random Tree*”) tiene en cuenta el modelo matemático del robot pero no obtiene una trayectoria optimizada.

En cuanto a la otra vertiente, los algoritmos de planificación de trayectorias tradicionales (algoritmos exactos o combinacionales), necesitan de una representación discreta del entorno para buscar una solución; son los utilizados al aplicar una estrategia de navegación planificada.

- “*Road Map*”. En este enfoque se construye un grafo del entorno de trabajo libre de obstáculos, cuyos arcos se utilizan para conectar un punto inicial y un punto final. Los métodos más utilizados son el grafo de visibilidad [16] que se basa en la construcción de un grafo no dirigido, el cual busca trayectorias de longitud mínima situando sus arcos lo más cerca posible de los obstáculos, y el diagrama de Voronoi [17] que en contraposición al anterior intenta fijar sus arcos lo más alejados posible de los obstáculos.
- Descomposición de celdas. Se basa en la partición del espacio libre de obstáculos en un conjunto finito de regiones que pueden ser recorridas de forma segura por un robot [18]. Es la técnica más utilizada dentro de un enfoque de planificación con especificaciones a alto nivel, según las cuales el robot puede visitar algunas regiones basándose en requisitos lógicos o temporales.

Aplicar técnicas de planificación de trayectorias partiendo de una serie de especificaciones a alto nivel es difícil por lo que tras discretizar el espacio de estados (entorno de trabajo), la obtención de una ruta óptima que una un punto de partida y de destino se lleva a cabo mediante algoritmos basados en grafos.

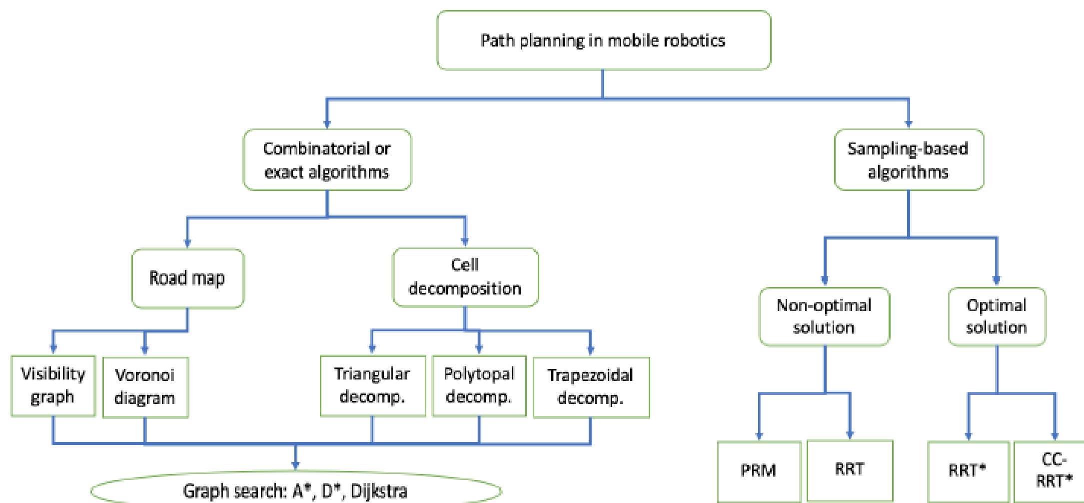

Figura 2: Clasificación de algoritmos de planificación de trayectorias [7].

Los algoritmos más utilizados son A*, Dijkstra y D*; los dos primeros se basan en el uso de un grafo invariante en el que cada uno de sus arcos se caracteriza por un peso definido y se diferencian en que el algoritmo A* usa una función de coste heurística, teniendo en cuenta tiempo y distancia recorrida, para encontrar la secuencia de puntos con el menor coste posible e ir de un punto a otro [19], mientras que el algoritmo Dijkstra busca la trayectoria más corta o de menor coste, teniendo en cuenta distancia y el peso de los arcos del grafo, partiendo desde el punto de destino [20]; por último, el algoritmo D* es una modificación del algoritmo A* que presenta la ventaja de poder replanificar la ruta inicial mientras esta se ejecuta si el robot percibe que el entorno ha cambiado [21].

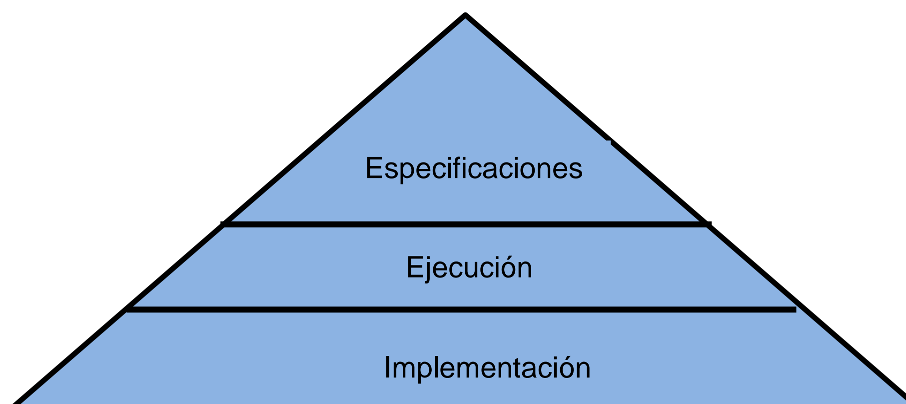

Figura 3: Niveles dentro de un proceso de navegación estándar [7].

El proceso de navegación se puede entender como un proceso jerárquico que consta de tres niveles (Figura 3), en el que los dos niveles superiores se recogen dentro de la estrategia de planificación mientras que el tercero corresponde al control del movimiento. El nivel de especificación corresponde a la discretización del entorno, el nivel de ejecución se encarga de generar una trayectoria óptima que debe ser seguida por el robot teniendo en cuenta la proximidad de los obstáculos y las posibles colisiones con ellos, y, por último, el nivel de implementación asegura que el robot siga la trayectoria de referencia atravesando la secuencia de regiones dentro del entorno que vienen recogidas en la ruta obtenida.

2.2. Control de movimiento

El objetivo de un robot dentro de su estrategia de navegación es seguir la ruta generada en la etapa de planificación de trayectorias; por lo tanto, controlar un robot consiste en determinar las acciones (fuerzas o velocidades) que deben ser desarrolladas por sus actuadores con el fin de que alcance una posición deseada, siguiendo una trayectoria predefinida que cumple ciertas especificaciones de alto nivel.

Tanto la posición 2D del robot como la orientación, en el caso de que haya que tenerla en cuenta, cambian a cada instante, por lo que el controlador debe asegurarse de que el límite del error ($e(t)$) entre la posición actual del robot y su posición de destino tienda a cero.

$$e(t) \quad \cong 0$$

Según [22], el problema del control de movimiento de un robot se puede definir como una realimentación en la que el controlador reciba desde un sensor la posición del robot y la compare con el punto de destino, calculando la acción necesaria que se necesita aplicar al actuador del robot en cada instante.

Es importante resaltar que la controlabilidad de un robot se puede ver afectada tanto por limitaciones (limitaciones físicas de los actuadores, limitaciones introducidas por las dimensiones del entorno, etc.) como por incertidumbres (dinámica del robot no modelada, simplificación de los modelos, ruidos en las medidas, entre otros).

La primera referencia encontrada es el controlador “Pure Pursuit” que se implementó a inicios de los años 80 en el robot Terragator y consistía en fijar un punto de destino delante del robot y dirigirse hacia él [23]; un algoritmo similar al anterior es el controlador *“Carrot heading*” que determina el siguiente punto al que se dirige el robot calculando la intersección entre un círculo con centro en el origen del robot y la trayectoria deseada [24].

Ambos controladores siguen una trayectoria considerando únicamente la posición del robot, por lo que una vez el robot haya definido el siguiente punto a donde debe moverse es necesario aplicar un control sobre la posición del robot, a partir del cual se determina la velocidad de mismo

Cuando se rastrea una trayectoria, es decir la información de la trayectoria incluye posición y orientación, el robot debe seguir a un robot virtual con cierta frecuencia teniendo en cuenta ambas variables; entre las estrategias destacadas se encuentra un control que usa un modelo predictivo (MPC), el cual utiliza unas leyes de control variables para optimizar el rastreo de la trayectoria.

Por último, hay que considerar que la respuesta obtenida de la planificación de trayectorias es una ruta a alto nivel que incluye una determinada secuencia de puntos, por lo que dichas especificaciones son una limitación ya que no se pueden aplicar directamente al robot; esto hace necesario utilizar un enfoque simbólico a través de distintos formalismos matemáticos que pueden ser especificaciones algebraicas, redes de Petri, lógica temporal lineal, lógica temporal ramificada, entre otros.

2.3. Colisiones en un sistema multi-robot

Al considerar un sistema multi-robot como por ejemplo un conjunto de robots omnidireccionales que trabajan sobre un mismo espacio continuo, donde se aprecian obstáculos, es necesario que los robots estén sincronizados y cooperen con el fin de que cada uno de ellos alcance su destino.

Dentro de un sistema multi-robot, la evitación de colisiones entre los robots es un problema a tener en cuenta y la estrategia utilizada para controlar estas posibles colisiones es independiente a la estrategia de navegación propia de cada robot, la cual tiene en cuenta la colisión con los obstáculos.

En el caso de este proyecto, las trayectorias asignadas a cada uno de los robots se obtienen usando modelos discretos formales, de forma que el espacio de trabajo se discretiza, es decir, se ha divido en regiones a las cuales se le asigna un estado discreto, por lo tanto, para garantizar un funcionamiento libre de colisiones se utiliza un modelo matemático discreto, autómata finito o red de Petri, que permita controlar la sincronización de los robots.

- Sistema de transición/autómata finito.

Formalismo que modela el sistema como un grafo en el que cada nodo es un estado del sistema y hace referencia a un conjunto de celdas que pueden ser ocupadas en un momento dado por el grupo de robots; además, cada nodo dispone de una salida que dice si es una región de interés o es un espacio vacío. La complejidad del modelo (número de nodos necesarios para describir el sistema) incrementa de forma exponencial con el número de robots

- Redes de Petri.

Formalismo que modela los posibles movimientos de un robot, pero el estado del sistema es numérico y distribuido, lo que permite la definición de una ecuación de estado que puede ser usada para análisis y diseño.

Otro factor que influye sobre la complejidad de modelo es la precisión con la que se haya discretizado el entorno, la cual se ve reflejada en el coste computacional requerido para ejecutar el control de sincronización.

Los cambios de estado se generan por el movimiento síncrono de uno o varios robots. Una técnica para evitar colisiones entre robots es introducir una capacidad limitada a las regiones del entorno de forma que estás solo puedan ser ocupadas por un solo robot; la capacidad de cada región se entiende como un recurso de forma que el sistema se modela como un RAS. Se encuentran más detalles de este tipo de sistemas en el capítulo 4.

Dentro de este tipo de sistemas, existe una competencia por los recursos que induce tiempos de espera, los cuales pueden generar estados de bloqueo en el sistema, por lo que la estrategia que controla los bloqueos puede ser:

- Evitación de bloqueos

Técnicas que consisten en un análisis dinámico del sistema cada vez que se vaya a asignar un recurso a un proceso con el fin de asegurar de que dicha asignación no derive en un bloqueo. En [7] se plantea una evitación de bloqueos estableciendo un retraso inicial a cada uno de los procesos, mientras que otra técnica comúnmente utilizada es el algoritmo del banquero [1, 2,3], la cual se implementa en este proyecto.

- Prevención de bloqueos.

Técnicas que modelan el comportamiento del sistema para eliminar toda posibilidad de bloqueo entre las que destaca la utilización de un algoritmo de búsqueda de unos elementos estructurales llamados sifones malos en una red de Petri [5]. Según la definición de una red de Petri perteneciente a la clase S4PR cualquier estado de bloqueo está asociado con al menos un sifón mínimo vacío (sifón malo) por lo que es necesario encontrar dichos sifones [5] y modificar el modelo añadiendo lugares monitor que garanticen que estos no se vacíen.

Capítulo 3
APLICACIÓN MULTI-ROBOT

3.1. Elementos

La aplicación es un sistema multi-robot que consta de tres elementos: espacio de trabajo, robots y sistema de visión.

- Espacio de trabajo

  Rectángulo de 3.32 x 2.34 m, por el que se desplazan los robots y en el que se sitúan los obstáculos. Como se puede ver en la Figura 4, sobre dicho espacio se fijan ocho marcadores que forman parte del sistema de visión; cuatro de ellos, los marcadores esquina, están situados en las esquinas a ras de suelo y los otros cuatro, marcadores elevados, se sitúan a la altura de los robots posicionados a 32 cm de los marcadores esquina.

  Además, dispone de unos rectángulos de color verde, rojo y azul, que actúan como filtro en la detección de obstáculos, por lo que cualquier forma geométrica con esos colores se considera un obstáculo, siempre que la misma esté dentro del espacio de trabajo. La posición de los obstáculos no es fija ya que el usuario puede modificar dichas posiciones en cada ejecución de la aplicación.

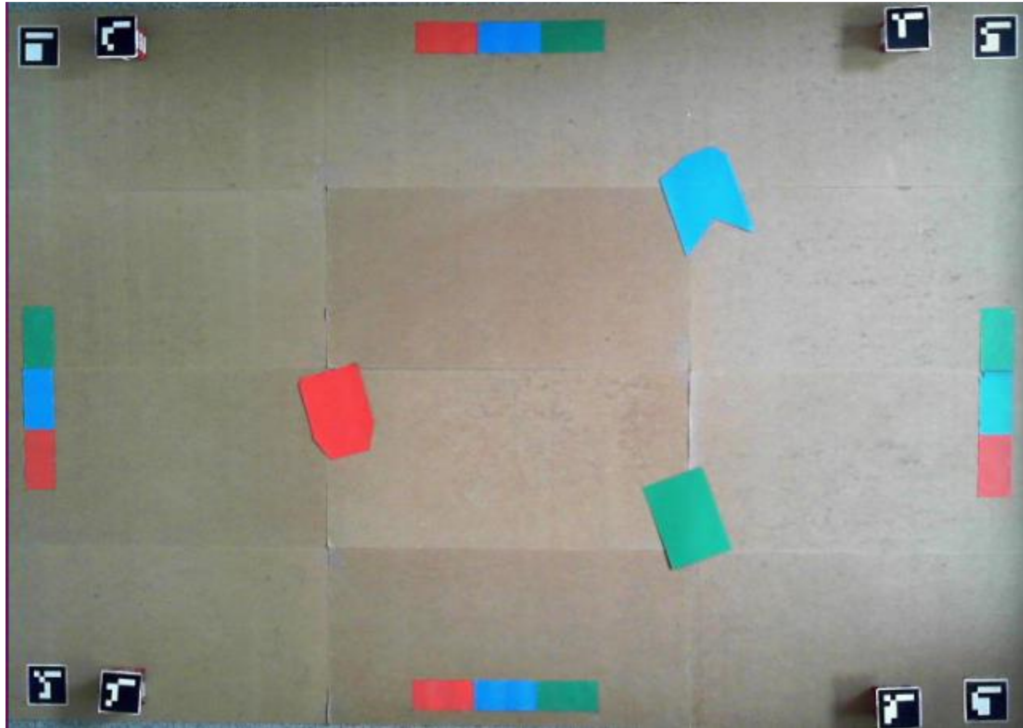

Figura 4: Espacio de trabajo de la aplicación

- Robots

La plataforma móvil utilizada es la “Turtle 2WD Mobile Plataform” caracterizada por ser pequeña, de bajo coste y compatible con el uso de un microcontrolador Arduino.

Presenta dos cubiertas de una aleación de aluminio con alta resistencia sobre las que se distribuyen: dos motores de accionamiento, dos ruedas de goma flexible, una bola de aleación de aluminio de alta resistencia que favorece el giro de la plataforma, un soporte de batería de 5 AA, un conector de alimentación, un interruptor de encendido/ apagado. La cubierta superior dispone de un marco para el microcontrolador, soportes para tres sensores y una toma de un servomotor que modele el giro del robot.

Para conseguir que este robot funcione se utiliza la placa “DFRobot RoMeo A11 In One COntroller V2.2”, que presenta un controlador de un motor DC de dos vías y un conector XBee, que es un módulo de baja frecuencia que transmite y recibe información. La placa de robot Arduino RoMeo V2 se comporta como Arduino Leonardo (Anexo I.A) basado en el chip ATmega32u4, siendo este el único microcontrolador que modelará el comportamiento del robot.

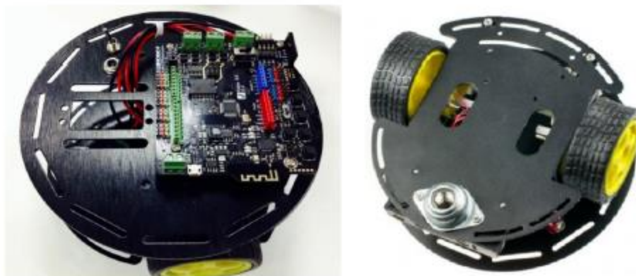

Figura 5: "Turtle 2WD Mobile Platform"

- Sistema de visión

El sistema de visión tiene como objetivo localizar los obstáculos en el espacio de trabajo y detectar la posición de los robots que participan en cada ejecución de la aplicación. Esta información, considerada la salida del sistema de visión, se almacena en un conjunto de variables que sirven de entrada para los distintos algoritmos que modelan la navegación de los robots.

Se compone de una cámara y de marcadores [6]. La cámara se sitúa en posición cenital a una determinada altura en la que su campo de visión abarca todo el espacio de trabajo (marcadores, obstáculos y robots).

Figura 6: Microsoft LifeCam Studio

La cámara utilizada es Microsoft LifeCam Studio 1080p HD que presenta tres resoluciones 640x480 píxeles, 1280x720 píxeles y 1920x1980 píxeles.

Tabla 1: Posición de robots con las distintas resoluciones de la cámara

	640x480 píxeles	**1280x720 píxeles**	**1920x1080 píxeles**
Rob.1	263,168,-2.028(243.82º)	263,168,-2.004(245.195º)	263,168,-2.009(244.91º)
Rob.2	256,53,2.229(127.69º)	256,53,2.245(128.63º)	256,53,2.223(127.37º)
Rob.3	166,101,-1.758(259.26º)	166,101,-1.75(259.72º)	166,101,-1.754(259.49º)

Como se puede ver en la Tabla 1, un aumento en la resolución de la cámara implica una mayor precisión en la orientación del robot, sin apreciarse unas diferencias significativas entre las distintas resoluciones de las que dispone; es por ello por lo que se ha decidido utilizar la máxima resolución posible.

Para conseguir trabajar con imágenes se han usado las librerías para visión por computador OpenCV y ArUco; OpenCV permite trabajar con los fotogramas obtenidos desde la cámara y ArUco permite estimar la posición de los robots usando los marcadores que se sitúan en el espacio de trabajo. La aplicación dispone de ocho marcadores fijos, marcadores esquina y marcadores elevado, y marcadores móviles, tantos como robots estén trabajando en cada ejecución de la aplicación.(Figura 7).

El sistema de visión está encapsulado en la clase “*Eng*” que contiene dos funciones: “*Set_Points*” que inicializa la posición de las cuatro áreas rectangulares localizadas en la parte central de cada lado del entorno de trabajo y “*Eng_Process*” que modela el sistema de visión solicitando varias entradas tales como las dimensiones del espacio de trabajo, el número de robots con que se ejecuta la plataforma, la imagen obtenida por la cámara y el vector obtenido de “*Set_Points*”.

Por último, se ha considerado necesario fijar los marcadores móviles a la cubierta superior de los robots ya que en la fase de testeo se observó que el movimiento del robot podía no ser solidario al movimiento del marcador y derivar en un movimiento del marcador que falsee datos de la posición del robot.

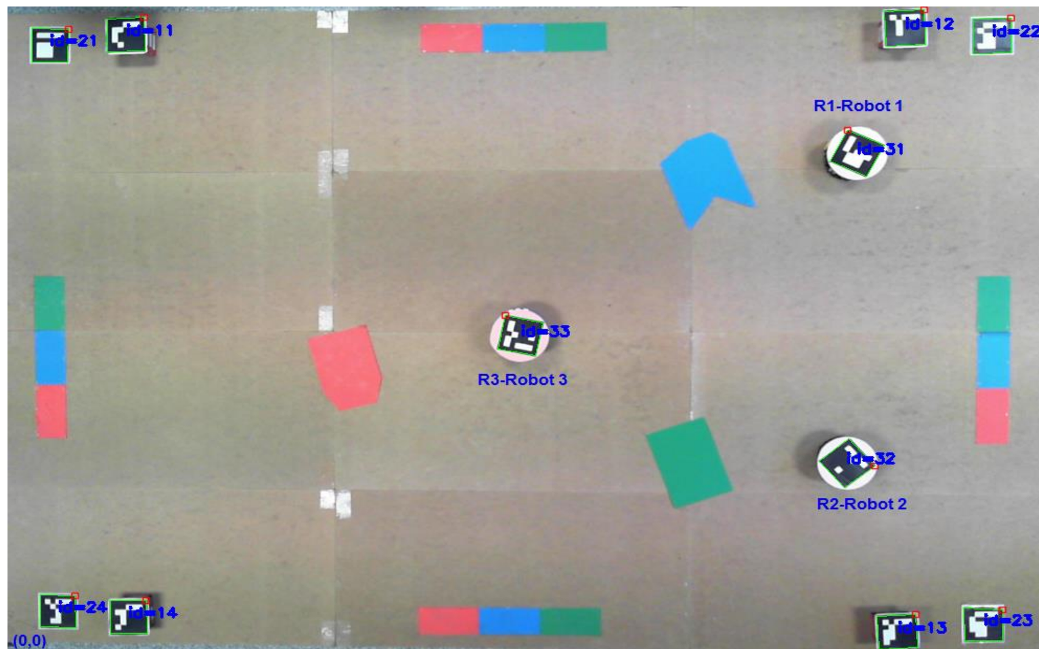

Figura 7: Imagen que muestra los elementos detectados por el sistema de visión

3.2. Comunicación PC-Robots

Para conseguir un funcionamiento colaborativo de los robots dentro de la plataforma se define una metodología IoT que permita una comunicación M2M (interacción de máquina a máquina) entre los dispositivos inteligentes (PC y microcontrolador de cada robot). Dicha comunicación se consigue mediante dispositivos XBee, que son pequeños módulos RF (radio frecuencia) que transmiten y reciben datos sobre el aire usando señales de radio.

En concreto, se utilizan antenas XBee serie1 para modelar una comunicación inalámbrica punto a multipunto, bidireccional, ya que tanto PC como robots envían y reciben información, y “*broadcast*”, lo que significa que la información emitida por cualquiera de los elementos de la red es pública y la reciben el resto de los componentes de esta.

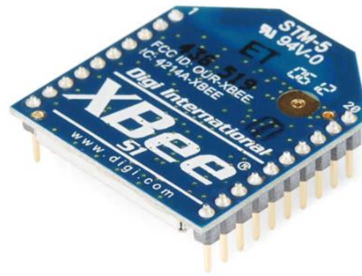

Figura 8: Antena XBee Serie 1.

Dentro del proceso de comunicación XBee, además de una comunicación inalámbrica, es necesaria una comunicación serial que en este caso es en modo transparente, lo que significa que cada antena XBee transmite la información al dispositivo inteligente tal cual la recibe al puerto serie sin ningún tipo de tratamiento de la información. Una ampliación sobre los conceptos asociados a los dispositivos XBee se puede encontrar en el Anexo I.B.

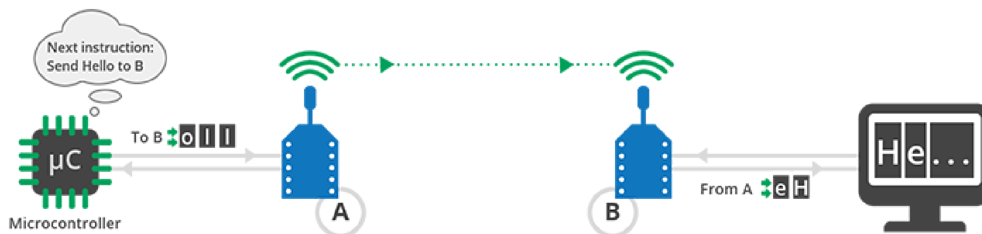

Figura 9: Esquema comunicación XBee

Capítulo 4
ALGORITMO DEL BANQUERO

La herramienta con la que se trabaja es un sistema multi-robot en el que existe una ejecución concurrente de procesos, entendidos como las trayectorias asignadas a cada uno de los robots que participan en la ejecución de la aplicación.

A cada robot se le asigna una ruta a alto nivel que incluye una secuencia de puntos por los que dicho robot debe pasar para llegar a un destino; dada la naturaleza discreta de las trayectorias es necesario modelar el sistema multi-robot utilizando herramientas SED (sistemas de eventos discretos) con el fin de conseguir una ejecución concurrente y simultánea de las trayectorias.

Una solución para garantizar la finalización de las trayectorias sin que las mismas se interrumpan por colisiones o bloqueos es dotar a cada región discretizada del espacio de trabajo de una capacidad unitaria, considerándolas recursos compartidos del sistema. Este enfoque permite considerar a la herramienta un RAS (Resource Allocation System) que se define como un sistema compuesto por un conjunto de procesos (secuenciales o no) que comparten un conjunto de recursos conservativos, es decir, ni se crean ni se destruye [2]. Los RAS se aplican a programas concurrentes (los procesos son los programas en ejecución y los recursos, los dispositivos que deben compartir: memoria, disco, impresoras, etc.), sistemas de producción (los procesos son las piezas involucradas, mientras que los recursos son las estaciones de trabajo, almacenes, robots, etc.), etc.

Al considerar que la aplicación es un RAS, la utilización de recursos compartidos genera modos de espera ya que en el caso de que una región (recurso) sea requerida por dos robots al mismo tiempo, hay una competencia para adquirirlo de forma que el robot que no consiga acceder a dicho recurso debe esperar a que el robot que ha accedido al recurso lo libere; cada recurso está disponible o asignado a un único proceso, el cual puede liberar un recurso únicamente si previamente lo ha adquirido. El inconveniente de este tipo de sistemas en el que se trabaja con recursos compartidos es que la competencia por los mismos puede derivar en situaciones de bloqueos que deben ser controladas.

La ejecución concurrente de procesos usando recursos compartidos es la razón que exige la implementación de un algoritmo de evitación o prevención de bloqueos que garantice el funcionamiento cooperativo y sincronizado de los robots que forman la herramienta. Considerando que se utiliza una representación discreta del sistema (redes de Petri, grafo, álgebra de procesos, etc.) se ha decidido utilizar el algoritmo del banquero, el cual presenta una baja complejidad computacional y tiene como objetivo determinar si un estado del sistema es seguro, lo cual permite asegurar que, partiendo desde dicho estado todos los robots completarán sus procesos de forma secuencial.

Existe una literatura variada que utiliza este algoritmo como estrategia de evitación de bloqueos y se diferencian en el modelo matemático sobre el que se aplica. En [1] se propone aplicar el algoritmo del banquero considerando la distribución de una planta por la que circulan múltiples AGVs como un grafo cuyas intersecciones y fin de trayectorias son los nodos, mientras que las trayectorias en sí mismas son los arcos; por otro lado en [2,3] se utiliza una red de Petri para controlar la ejecución concurrente de procesos dentro de un sistema RAS con la diferencia de que en [3] la naturaleza de los procesos es no secuencial (NS-RAS). Los modelos matemáticos difieren entre ellos por las limitaciones impuestas en el uso de recursos en cada proceso y por las limitaciones que introduce la estructura del conjunto de secuencias de procesos.

4.1. Modelo de representación de la plataforma

Siguiendo [1], se ha optado por utilizar un grafo, H = (N, A), que represente al conjunto de trayectorias asignadas a los robots que trabajan sobre el espacio de trabajo discreto, cuyos nodos (N) representan el conjunto de puntos recogidos en las trayectorias, mientras que los arcos (A) representan los distintos tramos que conforman cada una de las trayectorias. Algunos supuestos aplicados en la planificación de robots son:

- Se permite a un único vehículo ocupar un arco, con lo cual se está considerando que los recursos del sistema son los arcos. Dado que el modelo discreto utilizado es un grafo que considera las trayectorias como una secuencia de tramos(arcos) formados por dos regiones de la discretización utilizada en la planificación de trayectorias, es necesario un cambio de formato en las mismas.
- Los nodos representan el punto central de una celda o región dentro del espacio de trabajo.
- Los vehículos no pueden dar marcha atrás en su trayectoria, es decir, cada robot se mueve en un único sentido para alcanzar su punto de destino.

Dentro del modelo utilizado, se dice que una misión (m_i= (σ_i, i),$i = 1, \dots, n_{Robots}$) es el conjunto formado por una trayectoria y el robot al que ha sido asignada. Una trayectoria (σ_i) se define como la secuencia de arcos que conectan los nodos a visitar por cada $robot_i$ y el índice de arco (k_i), indica la posición del $robot_i$ dentro de su trayectoria (k_i = j significa que el $robot_i$ ocupa el arco j de la trayectoria σ_i).

El estado del sistema viene definido por el conjunto de trayectorias ($\Sigma = \{\sigma_1, \sigma_2, \sigma_3 \dots, \sigma_{nRobots}\}$) y el conjunto de arcos indexados ($K = \{k_1, k_2, k_3 \dots, k_{nRobots}\}$). En lo que respecta al sistema se deben conocer los siguientes conceptos:

1. Estado inicial. Todos los robots a los que se le ha asignado una trayectoria están en el primer arco de esta.

$$S_0 = (\Sigma, K) \forall\, k_{robot_i} \in K : k_{robot_i} = 1$$

2. Estado final. Todos los robots han alcanzado su último arco.$|\sigma_i|$ representa el número de arcos que conforman la trayectoria del $robot_i$.

$$S_f = (\Sigma, K) \forall k_i \in K : k_i = |\sigma_i|$$

3. Estado Permitido. Un estado es permitido si todos los robots que están activos se sitúan en un arco diferente.

$$\forall p, q \in R, p \neq q : \sigma_p(k_p) \neq \sigma_q(k_q)$$

4. Cambio de estado por el movimiento de un robot.

$$1 \leq r \leq |R|, \quad k_i' = \begin{cases} k_i & , i \neq r \\ k_i + 1, & i = r \end{cases}$$

Habrá un cambio de estado si un robot cambia el arco en el que se sitúa.

$$S = (\Sigma, K) \quad robot\, i \rightarrow \quad S' = (\Sigma', K')$$

5. Estado de bloqueo. Se puede dar un bloqueo total, situación que surge si ningún robot puede cambiar de estado y conseguir un estado permitido, o un bloqueo parcial, que se da cuando un número determinado de robots no puede avanzar ni, por tanto, terminar sus misiones. Utilizando el algoritmo del banquero se evitan ambas situaciones.

4.2. Aplicación del algoritmo del banquero

Un cambio de estado, según lo descrito en el apartado anterior, se da cuando al menos uno de los robots cambia el arco en el que se sitúa, por lo tanto, se asocia a operaciones de asignación y liberación de recursos.

El enfoque aportado por el banquero a la hora de evitar bloqueos se basa en un proceso de decisión que permite asignar recursos a los distintos procesos activos dentro del sistema, usando información acerca de todos los recursos necesarios que pueden ser solicitados por un proceso para asegurar su culminación.

La aplicación formal del algoritmo se explica mediante un ejemplo, en el que los robots deben seguir las siguientes trayectorias.

.

Trayectorias

$$\left\{ \begin{array}{c} Q_3\,,\,Q_6\,,\,Q_{14}\,,\,Q_{13}\,,\,Q_{15} \rightarrow \text{arco5, arco7, arco8, arco9} \\ Q_{11}\,,\,Q_8\,,\,Q_7\,,\,Q_6\,,\,Q_3\,,\,Q_4 \rightarrow \text{arco12, arco3, arco4, arco5, arco6} \\ Q_1\,,\,Q_2\,,\,Q_8\,,\,Q_{11},\,Q_{12}\,,\,Q_{16} \rightarrow \text{arco1, arco2, arco11, arco10} \end{array} \right.$$

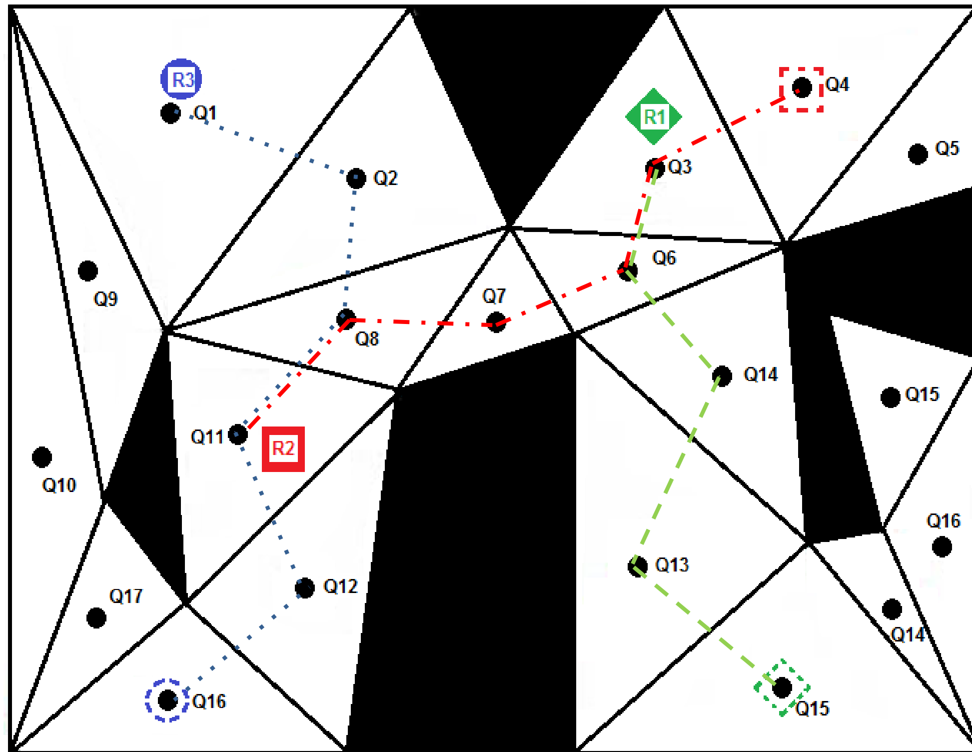

Figura 10: Trayectorias sobre el espacio de trabajo discreto.

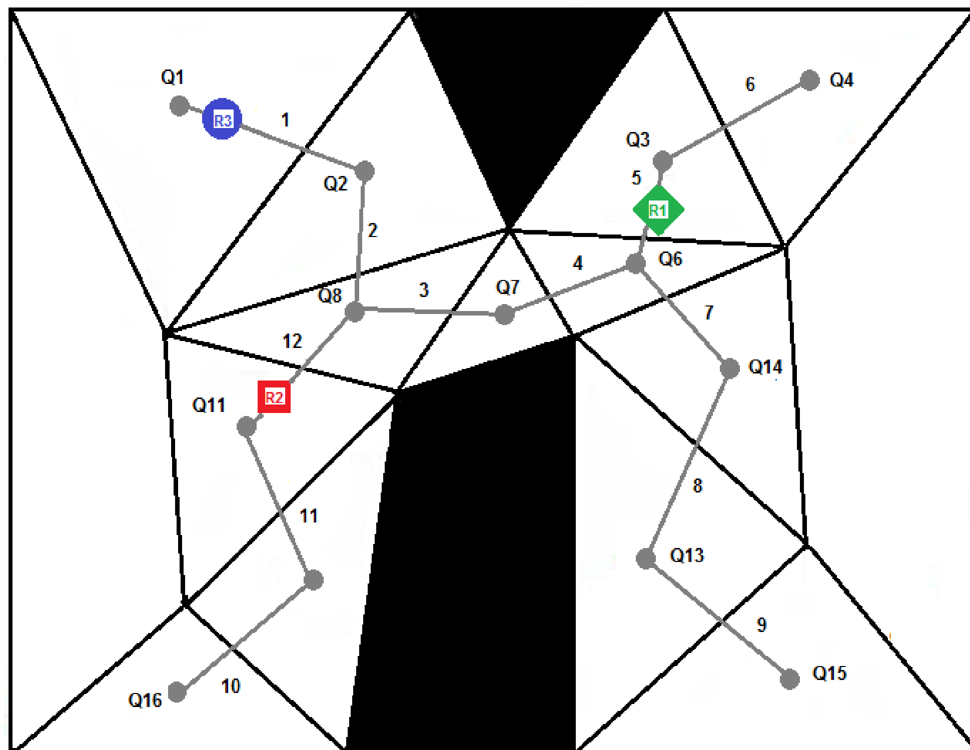

Figura 11: Grafo que modela la aplicación multi-robot

La Figura 10 muestra un entorno discretizado y las trayectorias, entendidas como secuencias de puntos, que los robots deben seguir, mientras que la Figura 11 muestra el modelo del sistema en el que solo se tiene en cuenta las regiones que conforman las trayectorias entendidas como secuencia de arcos. En el ejemplo se procede a aplicar el algoritmo del banquero para el estado inicial del sistema; el primer paso es definir todos los recursos que están siendo utilizados en dicho estado, los cuales vienen recogidos en el vector de arcos indexados.

$$arcos_indexados \ = \ [1, 1, 1]$$

A partir de las trayectorias y el vector de arcos indexados se construye un grafo dirigido "*wait-for*" ($G = (N_g, A_g)$) que presenta tantos nodos (N_g) como procesos se están ejecutando; existe un arco entre dos nodos ($robot_i$,$robot_j$) si y sólo si el vehículo representado por el $nodo_j$ ocupa un arco que está en la trayectoria necesaria para que el robot representado por el $nodo_i$ termine su misión, por ejemplo, el robot 1 del ejemplo ocupa el arco 5, el cual forma parte de la trayectoria del robot 2, por lo que el grafo presenta un arco que va desde nodo correspondiente al robot 2 hasta el nodo del robot 1.

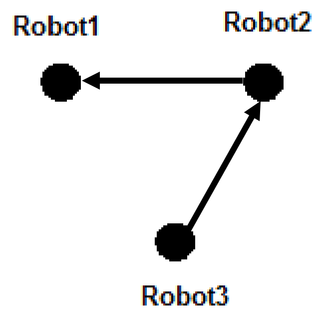

Figura 12: "Wait-for" Graph del estado inicial del sistema asociado al ejemplo.

Partiendo del grafo dirigido "*wait-for*" de la Figura 12, se lleva a cabo un proceso iterativo en el que se comprueba si existen nodos que no tengan arcos de salida. Si un nodo no presenta arcos de salida, significa que los recursos que conforman la trayectoria de dicho robot no están siendo utilizados por ningún otro robot, de forma que dicho robot puede avanzar hasta su posición de destino. Una vez se haya comprobado que el grafo es acíclico, es decir, dispone de al menos un nodo con arcos de salida, se elimina dicho nodo para seguidamente repetir el proceso, el cual termina cuando se han eliminado todos los nodos o cuando el grafo es cíclico, lo que indica que el estado no es seguro.

La salida del algoritmo indica si el estado analizado es seguro o no; en caso de que el estado sea seguro el algoritmo devuelve una secuencia de salida que describe el orden de ejecución de todas las trayectorias activas, la cual no tiene que ser única y no tiene que ser seguida obligatoriamente por los robots.

Algorithm 1: BA-SAFETY CHECK $(G = (N_g, A_g))$
Output; state is BA-safe
while $(\exists n_i \in N_g : outdegree(n_i) = 0)$

$$N_g = \frac{N_g}{n_i}$$

end
if $(N_g = 0)$ output := true
else output: = false
end

Figura 13: Algoritmo del Banquero

La aplicación de este algoritmo está supeditada a que el sistema se encuentre en un estado seguro y su objetivo es determinar si un estado subsiguiente lo es; si dicho estado es seguro se garantiza la existencia de al menos una secuencia de ejecución en la que todos los procesos activos se lleven a cabo de forma secuencial. Por otro lado, las trayectorias son consideradas como el conjunto de recursos que un robot debe utilizar para alcanzar un punto de destino, de forma que la información correspondiente a los recursos que un robot necesita para finalizar su misión es dinámica ya que cambia con el estado del sistema.

4.3. Control de ejecución utilizando el algoritmo del banquero

Se considera relevante mencionar que el algoritmo del banquero establece que el estado inicial y final del sistema no utilizan recursos, por lo tanto, son seguros. [1] define unos nodos denominados “nodos reposo” como puntos de inicio y fin de una trayectoria modelando la ausencia de recursos en dichos estados. En la aplicación no se consideran dichos nodos, por lo tanto, esta debe corroborar que el estado inicial sea seguro y garantizar que el estado final del sistema también lo sea.

Si el estado inicial de un sistema es seguro, se puede garantizar que todos los robots lleguen a su posición de destino sin bloqueos, pero esta solución presenta una baja utilización de los robots e induce elevados tiempos de espera ya que cada robot ejecuta toda su trayectoria mientras el resto permanecen en su estado inicial. Para conseguir una mayor utilización de los robots y garantizar una ejecución de las trayectorias previniendo posibles situaciones de bloqueo, ya sean totales o parciales, se define un controlador supervisor que vaya aplicando el algoritmo del banquero cada vez que haya un posible cambio de estado en el sistema.

En este proyecto, se ha considerado que un cambio de estado únicamente puede ser consecuencia del movimiento de un robot a su siguiente arco dentro de su trayectoria, el cual se elige de forma aleatoria. Si no se introduce ninguna limitación, partiendo de un estado seguro con “n” robots se tendrán “n!+1” posibles estados siguientes; con esta propuesta, el número de estados posibles se reduce a “n” posibilidades de movimiento.

La aplicación del algoritmo del banquero a la plataforma ha necesitado de la implementación de la librería “*BankerAlgorithm”* (Anexo II.A), que sigue una programación orientada a objetos en C++.

4.3.1. Proceso de control

La función "*ConcurrentNavigationProcess()"* contiene el proceso de navegación de cada uno de los robots y el control de ejecución de dichos procesos; esta función es uno de los "threads" de ejecución implementados para conseguir la comunicación bidireccional que viene explicada con mayor detalle en el apartado 5.1.

La función inicia con una definición tanto de las características de la cámara como de las características de la aplicación (ancho, largo del espacio de trabajo, contador que modela el bucle de control, constructor de la clase "*Syst*", la cual se detallada en el Anexo II.A, entre otros). Una vez parametrizada la plataforma, esta función lleva a cabo un bucle que se ejecuta de forma ininterrumpida hasta que la cámara deje de proporcionar imágenes y el usuario salga de la aplicación.

Dentro de dicho bucle se ejecuta el sistema de visión, que devuelve la posición de los robots e información que permite calcular la orientación de estos, además, el sistema de visión está modelado por las variables booleanas "*aver*" y "*go*" para devolver, en su primera ejecución, la posición de los obstáculos. Tras la primera ejecución del sistema de visión, la variable "*aver"* se activa de forma que en las siguientes llamadas al sistema de visión únicamente devuelve información de la posición de los robots.

Esta función diferencia entre una primera ejecución y las subsiguientes ejecuciones del bucle. En la primera ejecución, se aplica un filtro que comprueba que los obstáculos obtenidos desde el sistema de visión dispongan de un área mínima para después crear y definir el objeto de la clase "*obstacles*" (Anexo II.A) cuyas propiedades "*centroids*" y "*threshold_values*" son entrada de la función "*send_ObstaclesPacket()*", que se encarga de enviar a los robots, el centroide y el área de influencia de cada uno de los robots con el fin de modelar en su interior una funcionalidad que evite la colisión con los obstáculos.

Además, se aplica la técnica de planificación de trayectorias previamente seleccionada por el usuario, "*PlanificationPathProcess()*", obteniendo las trayectorias que serán entrada del control de ejecución de trayectorias. Una vez se dispone de las trayectorias a cumplimentar se obliga al usuario a seleccionar el algoritmo de control, siendo este paso bloqueante.

En el caso del algoritmo del banquero, se inicializan las propiedades del objeto de la clase "*Syst*" entre las que se incluyen tareas como el cambio de formato de trayectorias, la definición del vector de arcos indexados, la inicialización de las variables de control, entre otros (Anexo II.A).

Cada vez que se ejecute la aplicación es necesario corroborar que el estado inicial sea seguro, lo que implica la ejecución del algoritmo del banquero para dicho estado; la aplicación esta modelada de forma que en caso dicho estado no sea seguro acabe la ejecución, mientras que en caso contrario se proceda a la definición de las variables que modelan el funcionamiento del controlador durante las sucesivas ejecuciones ("c*ontrol_access*", "*modo*", "*status*"). El Anexo II.A contiene una explicación de dichas variables, que al inicializarse para cada robot estarían en "true", "wait" y "stop" respectivamente.

Se ha considerado que en el caso de que la aplicación sólo disponga de un robot no es necesario ejecutar el algoritmo del banquero cada vez que haya un cambio de estado por lo que se ha implementado una lógica que crea un objeto *"nextState"* y procede a completar las características propias de dicha clase como son el robot que debe moverse y el vector de arcos indexados, el cual contaría con una única componente que sería el número de segmentos del que dispone la trayectoria asignada a ese robot. Con el robot y el vector de arcos indexados se procede a determinar la trayectoria a enviar mediante la función "*setPathToSend(Syst&).* " descrita en el Anexo II.A. Además, las variables de control cambian de forma que "*control_access*" pasa a ser "false", "modo" cambia a "active" y estatus no sufre cambio con lo cual permanece siendo "stop".

En el caso de que la aplicación se ejecute con más de un robot, se propone un funcionamiento colaborativo de todos los robots ya que deben trabajar de forma simultánea para alcanzar el estado final del sistema, a pesar de la limitación de que sólo se permite el movimiento de un único robot en cada cambio de estado.

A lo anterior hay que añadir que el control interno de los robots dispone de un controlador proporcional sobre su orientación, que se ejecuta cada vez que el robot se reorienta con el fin de disminuir el error de orientación y proceder a moverse de la forma lo más recta posible. La reorientación inducía tiempos de espera a la hora de la ejecución de trayectorias por lo que se decidió separar el giro inicial de un robot al recibir una trayectoria definiendo el estatus "*turn*", del movimiento hacia delante de cada robot, que viene representado por el estatus "*walk*". Con esto se consigue que mientras un robot este moviéndose hacia a su destino, el resto puede ir ejecutando el control P respecto a su orientación, de forma que la ejecución de las trayectorias es más dinámica. Además, si un robot está moviéndose hacia su destino o realizando el giro inicial no puede recibir una trayectoria, correspondiente a un siguiente estado a cumplimentar, hasta que el primer estado haya sido alcanzado.

Esta lógica se ha implementado definiendo la variable "*list_FutureStates*", que es un vector que contendrá objetos "*nextState*"; cuya dimensión máxima es igual al número de robots que participan en la ejecución.

Los robots informan de que giran, en caso sea necesario, y de que están en disposición para iniciar el movimiento únicamente al recibir la trayectoria, por tanto, el PC no tiene constancia de que un robot se está reorientando para alcanzar los puntos que son diferentes al punto inicial de la trayectoria.

El control de ejecución de trayectorias se realiza tras la primera ejecución con una frecuencia de 0.5 segundos; este es uno de los puntos de mejora ya que en trabajos anteriores la frecuencia con la que se ejecutaba el controlador eran 2 segundos. Dicho controlador incluye 5 funciones(Anexo II.B) ejecutadas en el siguiente orden:

- Ejecución del algoritmo del banquero con su subsiguiente definición de los estados a alcanzar →"*getNextStateByBanker()*".
- Control del estado de los robots en el que se modela el tratamiento de la información recibida por el PC desde los robots → "*control_RobotStateBA()*"
- Gestión del inicio de movimiento de un robot hacia su destino utilizando la variable "*status*" → "*checkStatus()*"
- Envío de subtrayectoria →*"sendSubPaths()"*.
- Envío de la posición de los robots y control de colisiones con otros robots →"*sendPosition()"*

La ejecución de dichas funciones está supeditada al estado en el que se encuentren las variables de control.

Por último, esta propuesta se adecua a una ejecución online del algoritmo del banquero ya que en el caso de que exista una modificación de las trayectorias, estas son tenidas en cuenta.

Capítulo 5 MODIFICACIONES EN LA APLICACIÓN

Con el fin de mejorar el funcionamiento de la herramienta se han implementado funcionalidades que aportan valor a la misma.

5.1. Comunicación bidireccional

El implementar una comunicación bidireccional entre robot y PC permite tener un mayor control del sistema y monitorear, en tiempo real, el funcionamiento de este, lo que hace disponer de un sistema con mayor robustez.

Para implementar esta funcionalidad se ha modificado la configuración de la comunicación serial entre la antena XBee y el dispositivo inteligente (Anexo I.C), y ha sido necesaria la utilización de hilos de ejecución que han implicado una modificación de la estructura de la aplicación.

La plataforma dispone de funciones que envían información desde el PC a los robots.

- send_path_WiFi
- send_ObstaclesPacket
- send_packet_WiFi
- sendStopPacket
- SendForwardPacket

Todas estas funciones, tratan la información que reciben y la almacenan en un "*string*", según el formato de los distintos paquetes de datos definidos (Anexo I.D); dicha ristra de caracteres se envía a través de la antena XBee byte a byte mediante la función "*write()*", previa configuración del puerto "*setUSB()*". El hecho de que las funciones, anteriormente mencionadas, transmitan los paquetes de información byte a byte implica una mayor probabilidad de que haya un fallo en la recepción de dicha información por interferencia, es por ello, que el control interno de los robots realiza verificaciones para garantizar que la información recibida es correcta.

En cuanto al envío de información por parte de los robots al PC, se busca una comunicación más eficiente que presente un menor coste computacional y que disminuya el riesgo de perder información, por lo que la información enviada por el robot se encapsula en un único byte (Anexo I.D).

El implementar una comunicación bidireccional supone definir una función lectura,"*listen_response_WiFi()*", que induce tiempos de espera debido a que sigue un modo de procesamiento no canónico, el cual requiere la configuración de las propiedades "*TIME*" y "*MIN*" (Anexo I.C). Para implementar dicho modelo de trabajo se ha cambiado la configuración preexistente, la cual abría el puerto serie en modo no bloqueo, a la configuración por defecto que abre el puerto en modo bloqueo ya que si se especifica el modo no bloqueo ("*O_NONBLOCK*"), este tendrá prioridad respecto a las propiedades de configuración TIME y MIN, lo que deriva en que la función lectura devuelva error en el caso de que no haya datos disponibles.

Los valores de TIME y MIN definidos son 0 y 1 respectivamente, de forma que la ejecución de la función se bloquea hasta que se reciba un único carácter. La operación de lectura conlleva un bloqueo en la ejecución de la aplicación de forma que para conseguir que esta pueda recibir información mientras lleva acabo el funcionamiento del sistema multi-robot se implementa una ejecución concurrente de dos hilos.

```
int main(int argc,char **argv){
   // Targets contain the target point for each robot.
   //Number of robots is equal to target points

   std::vector<std::vector<double>> Targets;
   std::vector<double> target_rob;
   int nrobots = 0;

   std::cout << "Enter the number of Robots: ";
   std::cin >> nrobots;

   for(unsigned int i = 0; i < nrobots; i++){
        int x_coordinate,y_coordinate;

        std::cout << "Enter Target Point (in cm) for Robot "<<i+1<<std::endl;
        std::cout << "Enter x-coordinate"<<std::endl;
        std::cin  >>    x_coordinate;
        target_rob.push_back(x_coordinate);

        std::cout << "Enter y-coordinate"<<std::endl;
        std::cin >>     y_coordinate;
        target_rob.push_back(y_coordinate);

        Targets.push_back(target_rob);
        target_rob.clear();
   }

   std::thread first(ConcurrentNavigationProcess,nrobots,Targets);
   std::thread second(listen_response_WiFi);

   first.join();                // pauses until first finishes
   second.join();               // pauses until second finishes

} //END main
```

Figura 14: Main de la aplicación.

Como se puede ver en Figura 14, que muestra la estructura del programa principal, se han definido dos hilos de ejecución, el primero es la función "*ConcurrentNavigationProcess()*", que lleva a cabo tanto el proceso de navegación de los robots como el control de ejecución de las trayectorias asignadas a estos, y el otro es la función lectura "*listen_response_WiFi()*".

La implementación de más de un hilo de ejecución lleva consigo el problema de la "condición de carrera" que se genera cuando dos hilos intentan acceder de forma simultánea a la misma posición de memoria y alguno de dichos accesos es de escritura.

C++ dispone del tipo atómico que soluciona dicho conflicto y está libre de carreras de datos, lo que significa que si se da el caso de que un hilo escribe en un objeto atómico mientras otro hilo lee de ese mismo objeto, el comportamiento del proceso está definido.

Ambos hilos están relacionados por variables globales del programa, una variable atómica booleana "*StopReading*" que modela el final de la función lectura, y el vector "*ReceivedData"* que almacena los caracteres recibidos desde los robots.

El objetivo de esta funcionalidad es disminuir el coste computacional del programa ya que anteriormente, con la frecuencia definida en los parámetros de la ejecución (2 segundos), se hacía un cálculo de la distancia existente entre la posición obtenida del robot y la posición de destino; si al momento de hacer el cálculo, el robot se localizaba en un radio de 20 cm tomando como referencia el punto de destino se consideraba que el robot había alcanzado el destino de la trayectoria que se le había enviado.

La función "*listen_response_WiFi()*" consiste en un bucle controlado por la variable "*StopReading*", de forma que, al llegar información procedente de algún robot, esta se almacena en un vector denominado "*ReceivedData*". El vector "*ReceivedData"* contiene los caracteres recibidos y se trata en la función "control_RobotStateBA *()*".

5.2. Cambios en control interno del robot

Hay que considerar que los robots son de bajo coste, lo que significa que variables como la superficie, la presión es sus ruedas, lo motores que accionan el robot influyen sobre estos. El primer paso es calibrar el movimiento rectilíneo del robot que consiste en determinar si para moverse de forma recta hay que aplicar las mismas acciones sobre las ruedas o hay que aplicar un factor corrector sobre alguna de ellas.

Se ha definido un sketch para cada robot, "*cal_robot#*", en el que se alimentan los motores de este durante un tiempo predefinido; el usuario debe modificar las acciones hasta conseguir que el robot se mueva de forma recta el mayor tiempo posible.

Además, es necesaria una calibración del giro con el fin de obtener las acciones necesarias para realizar un giro cercano a 180º, que es lo máximo que va a girar un robot en un sentido. La función *"Turning()*" dispone de la posibilidad de desactivar la variable que modela la ejecución del controlador interno, "*controller_on()*", tras la primera estimación del giro. Ambas calibraciones se deben realizar tras estar seguros de que las pilas están completamente cargadas. Asimismo, hay que tener en cuenta que las ruedas presentan una inercia que se debe vencer para que el robot empiece a moverse.

Según lo especificado en el proyecto de partida [4], el control interno de los robots no necesita una actualización de la posición ya que utilizando un modelo matemático (Anexo III.A) se puede estimar la posición del robot en cada instante; considerando que el estimador no es exacto y que el movimiento del robot no es ideal, incluso aplicando un factor de calibración tiende a desviarse después de un determinado tiempo, se han añadido diversas mejoras:

- Durante el movimiento del robot a un punto de destino, este calcula la distancia a los obstáculos presentes en la plataforma, de forma que, si está localizado en el radio de influencia de alguno de dichos obstáculos, el robot procede a parar y a comprobar su posición; en esta situación se puede reorientar, haber llegado al destino o quedarse parado, momento en el que el usuario lo debe alejar del obstáculo para continuar el proceso.

- Tras recibir una trayectoria el robot envía un mensaje confirmando que la trayectoria recibida es correcta o un mensaje de que la trayectoria es errónea. Una trayectoria se considera errónea si el número de puntos leídos no coincide con el número de puntos indicado en el paquete o si alguno de los puntos leídos está fuera de la plataforma .Tras esto, se activa el controlador y por tanto, el robot puede trabajar en uno de los cuatro modos disponibles: "girar", modo en el que el robot lleva a cabo el controlador proporcional respecto a la orientación para alinearse con el primer punto de la trayectoria, "esperar", el robot espera la confirmación de que puede iniciar el movimiento hacia su destino, "avanzar", movimiento hacia un destino y "check", modo en el que se comprueba si ha alcanzado un punto de destino, que puede ser intermedio o final.
- El giro del robot no siempre alcanza los resultados esperados tras la primera estimación de giro ("*estimationmodel_rot()*"), de forma que se ha implementado un proceso de corrección en el que se estipula un valor límite de 10º. Esto significa que para que el robot pueda iniciar el movimiento hacia su destino, la diferencia de orientación entre el punto de inicio y dicho punto de destino debe ser inferior a 10.

  El robot solo envía información de que está realizando giros al inicio de la trayectoria, es decir, si una trayectoria presenta 5 puntos, el robot sólo informa al PC que está girando cuando se dispone a alcanzar el primer punto, por lo tanto, el PC no tiene constancia de los giros realizados para alcanzar el resto de los puntos de la trayectoria. Con esto se consigue que mientras un robot esté movilizándose a su destino el resto puedan reorientarse, consiguiendo que la ejecución de las trayectorias sea más eficiente.

  Esta funcionalidad modela un controlador proporcional sobre el giro del robot.
- En el proyecto de partida [4], el robot no presenta una actualización de la posición, con lo cual, en caso de que la distancia a recorrer por el robot sea grande no hay una corrección de su posición real. La unidad de control actualiza la posición de los robots cada 0.5 segundos, de forma que el robot podrá actualizar su posición con dicha frecuencia y de esta forma realizar un movimiento que tenga mayor precisión.
- El movimiento rectilíneo del robot utiliza un controlador P sobre la posición del robot, Anexo III.B, que determina la velocidad del robot en cada instante a partir de la posición; en el proyecto de partida [4] se consideraba que la orientación del punto de destino era fija, pero eso es incorrecto ya que la orientación de destino varía en cada instante al depender de la posición del robot, la cual cambia, según el modelo o la información proveniente desde el sistema de visión; esto hace necesaria la aplicación de un controlador proporcional respecto a la orientación que devuelva el ángulo de accionamiento del robot. Tanto la velocidad como el ángulo de accionamiento son variables que permiten obtener la velocidad de cada rueda y, por ende, la acción que se aplica a cada una de ellas.
- Cuando la posición estimada del robot supera el valor umbral en el que se considera que el robot ha alcanzado un punto de destino, ya sea intermedio o final, se ha implementado una comprobación en la que se espera la posición real actualizada del robot para determinar si dicha posición está dentro del umbral; en caso afirmativo, se considera que el robot ha llegado a su destino mientras que en caso contrario se procede a iniciar otro movimiento para alcanzar la posición de destino.

El Anexo III.C contiene una explicación con más detalle del controlador interno implementado y de todas las funciones de las que dispone.

Capítulo 6
RESULTADOS Y CONCLUSIONES

En cuanto al controlador interno de los robots, no se han replicado las condiciones de las pruebas experimentales realizadas en los proyectos anteriores, pero las mejoras son evidentes ya que se ha conseguido un 100 % de éxito a la hora de alcanzar un punto de destino, sin importar la distancia a recorrer; con la incorporación de la actualización de la posición se consigue que el robot pueda recorrer mayor distancia y que se vaya reorientando según convenga.

El que la posición del robot se vaya actualizando permite al robot determinar si tiene que reorientarse, lo cual puede derivar en el que el robot retroceda; hay que tener en cuenta que el algoritmo del banquero no considera la “marcha atrás”, y que, si retrocede, por motivos de reorientación, una distancia considerable el sistema no estaría controlado y podrían generarse colisiones.

Se ha conseguido una mayor rapidez en el movimiento de los robots, lo cual se refleja en la constante proporcional asociada al controlador de posición, anteriormente se consideraba una K_p de 1, mientras que ahora, el controlador proporcional trabaja con una constante de 1.2, lo cual supone una respuesta del sistema más rápida y una disminución en el error sin que el sistema sea inestable.

Asociado al movimiento del robot, se ha dejado de considerar una orientación de destino fija, lo que deriva en la implementación de un controlador proporcional sobre la orientación del robot; dicho control proporcional mejora el funcionamiento del robot ya que la estimación de las acciones a aplicar es más real. Estas modificaciones han

favorecido que el radio para considerar que un robot se encuentra en su destino haya disminuido de 20 a 15 cm.

Hasta el momento se consideraba que el giro del robot era ideal, pero la realidad dista mucho de ello, ya que factores como en nivel de batería pueden afectar. Lo anterior, ha llevado a incluir una funcionalidad que garantiza que los giros alcancen una desviación máxima de 10º, lo cual, favorece disminuir el error y tener mayor precisión a la hora de alcanzar el destino.

Tras este proyecto, el robot será capaz de detectar si corre peligro de colisionar con algún obstáculo y pararse para analizar la situación en la que se encuentra, pudiendo haber alcanzado su destino, reorientarse o dejar de moverse porque está muy cerca del obstáculo y necesita que el usuario lo aleje de este.

El añadir funcionalidades implica que la carga sobre el Arduino sea mayor, lo que ha obligado a modificar el código y optimizarlo para que el Arduino no se sobrecargue; por esta razón se han utilizado funciones propias de Arduino como "*parseInt()" y "parseFloat()"*. Asimismo, se ha disminuido el número de variables globales a utilizar dentro del sketch.

La implementación de una comunicación bidireccional disminuye el coste computacional del algoritmo de la aplicación y la hace más robusta. El añadir esta funcionalidad ha derivado en un rediseño de la aplicación, ya que ha sido necesario la utilización de threads de ejecución, lo cual ha supuesto la definición de clases que favorecen la programación orientada a objetos en el controlador de ejecución de trayectorias.

A la hora de comparar el algoritmo del banquero con el algoritmo de sifones, que es la estrategia de control implementada hasta el inicio de este proyecto, hay que tener en cuenta diversos condicionantes. Si únicamente se compara el tiempo de ejecución de los algoritmos, el algoritmo del banquero da mejores resultados ya que sólo tiene en cuenta un único estado, mientras que el algoritmo de sifones hace un análisis de los procesos en general. Si se compara el algoritmo del control de ejecución de trayectorias unido a la estrategia de prevención o evitación de bloqueos, se podría decir que el algoritmo de sifones da mejor resultado ya que se ejecuta una sola vez, mientras que el algoritmo del banquero tendrá que ejecutarse tantas veces como posibles estados existan.

Los tiempos medios de ejecución de las funciones que conforman el algoritmo del banquero, considerando 30 ejecuciones de este, son:

- La función que obtiene el "wait-for" Graph tarda de media 114.71 microsegundos.
- El proceso iterativo que determina si el estado es seguro tarda 31,2 microsegundos.
- Si se considera que la aplicación del algoritmo incluye los dos procesos el tiempo medio de ejecución es 145.89 microsegundos.

El rediseño del controlador de ejecuciones favorece un uso más eficiente de memoria, lo mismo que la disminución de variables globales; únicamente se han definido como variables globales las variables atómicas que relacionan los threads de ejecución.

Este proyecto aporta una solución al control de ejecución de trayectorias múltiples dentro de un sistema multi-robot, garantizando una ejecución sin colisiones ni ningún tipo de bloqueos; esto se consigue gracias a las funcionalidades añadidas tanto al algoritmo de la aplicación, como al algoritmo interno de cada uno de los robots que conforman la aplicación. El funcionamiento de la aplicación se puede visualizar en diversos videos [25,26,27,28].

Se quiere resaltar la dificultad de replicar una ejecución, cuando esta falla; para ello se ha utilizado el monitor serie de Arduino, lo cual obliga a tener siempre conectado el

robot a un ordenador, sin dejar de comentar que dicha conexión puede influir en el movimiento de los robots ya que estos son muy sensibles.

La herramienta dispone de varios puntos sobre los que seguir trabajando ya que a pesar de haber realizado un trabajo integral sobre ella, garantizando un funcionamiento adecuado, no se ha conseguido comparar de manera objetiva los algoritmos de evitación y prevención disponibles, ya que el haber rediseñado no sólo la aplicación, sino también el control interno de los robots imposibilita hacer una comparación entre los mismos, puesto que el algoritmo de búsqueda de sifones no se ha integrado en el nuevo diseño.

Además de lo anterior, se podría analizar y comparar los resultados aplicando un controlador PI al movimiento de los robots, teniendo en cuenta que la variable que acumule la integral debe tratarse de forma adecuada cuando se reciba una nueva posición desde el sistema de visión, para reflejar la realidad.

Otro punto de mejora que se ha observado es que no siempre coincide la posición en la que debe parar un robot con la que para realmente. Una explicación a ello, sobre la que se debe profundizar en caso se considere, es que el sistema de visión se ejecuta fuera del controlador de ejecución de trayectorias, por tanto, desde que se obtiene la posición hasta que se envía pasa un tiempo, que puede ser mayor o menor según las funciones que se lleven a cabo en cada iteración del controlador; durante dicho tiempo el robot no deja de moverse, con lo cual la posición que se envía no coincide con la posición real del robot. Son milésimas de segundo que en ciertas ocasiones pueden derivar en un error a tener en cuenta.

Para terminar, otro punto a analizar sería valorar si es necesario utilizar sensores y como influiría la utilización de estos en la aplicación.

BIBLIOGRAFIA

[1] L. Kalinovcic, T. Petrovic, S. Bogdan y V. Bobanac, «Modified Banker's algorithm for scheduling in multi-AGV systems.,» de *IEE International Conference on Automation Science and Engineering.*, Trieste,Italy, 2011.

[2] J. Ezpeleta, F. Tricas, F. García-Vallés y J. M. Colom, «A Banker's Solution for Deadlock Avoidance in FMS With Flexible Routing and Multiresource States.,» *IEEE Transactions on Robotics and Automation.,* vol. 18, nº 4, pp. 621-625, 2002.

[3] J. Ezpeleta y L. Recalde, «A deadlock avoidance approach for nonsequential resource allocation systems.,» *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans.,* vol. 34, nº 1, pp. 93-101, 2004.

[4] E. Vitolo, C. Mahulea y F. Basile, *Multi-Robot Platform for Path Planning and Control Using High-Level Specifications: Implementation and Experiments,,* Salermo,Italy, 2017.

[5] J. Oroz y C. Mahulea, *Diseño e implementación de un método para evitar colisiones en un sistema multi-robot.,* Zaragoza, Spain, 2016.

[6] J. Barrio-Arbex y J. M. Martinez-Montiel, *Localización de múltiples robots móviles mediante una cámara cenital.,* Zaragoza,Spain, 2016.

[7] C. Mahulea, M. Kloetzer y R. González, Path planning of cooperative mobile robots using discrete event models,2020.

[8] J. Luo, h. Ni y M. Zhou, «Control program design for automated guided vehicle systems via petri nets.,» *IEEE Transactions on Systems, Man, and Cybernetics: Systems.,* vol. 45, nº 1, pp. 44-55, 2015.

[9] R. Arkin y R. Murphy, «Autonomous navigation in a manifacturing environment.,» *IEEE Transactions on Robotics and Automation.,* vol. 6, nº 4, pp. 445-454, 1990.

[10] N. Nilsson, «A mobile automaton: an application of artificial intelligence techniques.,» de *Proc.Int. Joint Conf. on Artificial Intelligence (IJCAI).*, Whasington DC, 1969.

[11] L. Champeny-Bares, S. Coppersmith y K. Dowling, «The terregator mobile robot.,» de *Techical Report CMU-RI-TR-93-03, The Robotics Institute. Carnegie Mellon University*, Pittsburgh, 1991.

[12] J. C. Latombe, Robot Motion Planning., Kluger Academic Pub., 1991.

[13] J. Laumond, «Robot Motion Planning and Control.,» *Lecture Notes in Control and Information Sciences,* vol. 229, 1998.

[14] S. LaValle, Planning Algorithms., Cambridge, 2006.

[15] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki y S. Thrun, Principles of Robot Motion:Theory, Algorithms and Implementations., Boston: MIT Press, 2005.

[16] T. Lozano-Perez y M. Wesley, «An algorithm for planning collision-free paths among polyhedral obstacles.,» *Communications of the ACM.,* vol. 22, nº 10, pp. 560-570, 1979.

[17] R. Gonzalez, F. Rodriguez, J. Sanchez-Hermosilla y J. Donaire, « Navigation Techniques for Mobile Robots in Greenhouses.,» *Applied Engineering in Agriculture.,* vol. 25, nº 2, pp. 153-165, 2009.

[18] M. Kloetzer, C. Mahulea y R. Gonzalez, «Optimizing cell decomposition path planning for mobile robots using different metrics.,» de *ICSTCC'2015: 19th*

International Conference on System Theory, Control and Computing., Cheile Gradistei, Romania, 2015.

[19] M. Lin, K. Yuan, C. Shi y Y. Wang, «Path planning of mobile robot based on improved A* algorithm.,» de *Chinese Control And Decision Conference.*, Chongqing, China, 2017.

[20] E. Dijkstra, «A note on two problems in connexion with graphs.,» *Numerische Mathematik.,* vol. 1, pp. 269-271, 1959.

[21] A. Stentz, «The Focussed D* Algorithm for Real-Time Replanning.,» de *Proc. of the Int. Joint Conference on Artificial Intelligence.*, 1995.

[22] R. Gonzalez, F. Rodriguez y J. L. Guzman, *Autonomous Tracked Robots in Planar Off- Road Conditions. Modelling, Localization and Motion Control. Series: Studies in Systems, Decision and Control.,* Springer, 2014.

[23] R. Coulter, *Implementation of the pure pursuit path tracking algorithm. Technical Report CMU–RI–TR–92–01.,* Carnegie Mellon University, USA, 1992.

[24] D. Helmick, S. Roumeliotis, Y. Cheng, D. Clouse, M. Bajracharya y L. Matthies, «Slipcompensated Path Following for Planetary Exploration Rovers.,» *Advanced Robotics,* vol. 20, nº 11, pp. 1257-1280, 2006.

[25] https://youtu.be/pXIAg8VqREI

[26] https://youtu.be/kREuRvDsT6Q

[27] https://youtu.be/nS_1xDdvCOE

[28] https://youtu.be/DNn3U_MI0zc

ANEXOS

ANEXO I

Anexo I.A. Arduino Leonardo

Arduino es una plataforma abierta utilizada para fines electrónicos que combina una fácil utilización de software y hardware; es capaz de leer entradas (luz de un sensor, el accionamiento de un botón, un mensaje codificado) y convertirlas en una salida (activar un motor, encender un LED, publicar algo online). Entre las ventajas que ofrece Arduino se encuentran:

- Barato
- Compatibilidad con diversos sistemas: Windows, Macintosh OSX y Linux.
- Entorno de programación claro y simple.
- El lenguaje de programación que utiliza es C/C++, siendo posible utilizar todas las clases de C++ disponibles.
- El compilador usado es GCC por lo que se puede escribir y compilar un programa rápidamente con un simple editor de texto.
- Software abierto y extensible. Las herramientas con las que trabaja Arduino están publicadas y son de acceso libre.

Lo primero para trabajar con Arduino es instalar el software (IDE), que es de acceso abierto y permite escribir código, así como cargarlo a la placa Arduino; en el caso que atañe a este proyecto cada robot presenta acoplada una de dichas placas.

Todo código que se implementa en una placa Arduino consta de dos funciones imprescindibles: “*setup()*”, que es la función de inicialización y “*loop()*”, que es un bucle que se repita de forma indefinida.

Arduino permite descargar y utilizar una gran variedad librerías que el usuario elige según sus necesidades, así como importar una librería creada por el propio usuario; en lo referente a este TFM se ha trabajado con la librería *“StandarCplusplus”* que es compatible con la versión 1.6.0.

En cuanto al Arduino Leonardo es un microcontrolador que presenta 20 pines digitales de entrada/salida (7 de estos pines se puede utilizar como salida PWM y 12 como entradas análogas), un oscilador de cristal de 16 MHz, una conexión microUSB emulada que deja el puerto serial libre para la programación evitando conflictos de programación mientras se tengan periféricos conectados a la placa (los periféricos pueden ser de E/S para la comunicación del PC con medio externo o periféricos de almacenamiento), un conector de alimentación, un encabezado ICSP y un botón de reinicio.

Anexo I.B. Antenas XBee

Dentro de la plataforma se ha definido una metodología IoT que permite una comunicación M2M (interacción de maquina a máquina) entre los dispositivos inteligentes.

Se define una metodología IoT (Internet of things) como la agrupación e interconexión de dispositivos y objetos a través de una red (red privada o red pública como internet) donde todos ellos pueden ser visibles e interaccionar.

Cuando se habla de un dispositivo inteligente, se hace referencia a un dispositivo electrónico, generalmente conectado a otros dispositivos o redes a través de diferentes protocolos inalámbricos como Bluetooth, Zigbee, NFC, Wi-Fi, etc.

Los dispositivos inteligentes usados son el PC, que ejecuta el algoritmo que lleva a cabo el funcionamiento de la plataforma, y los microcontroladores que se adhieren a cada robot. La metodología usada en la plataforma consiste en que un PC emita una serie de mensajes enviándolos a todos los elementos que conforman la red de trabajo de modo que cada receptor filtrará el mensaje que le interesa; asimismo, los robots envían diversos mensajes de respuesta. Para conseguir la comunicación entre los elementos de la plataforma se utilizan antenas XBee, dispositivos de baja frecuencia que reciben y transmiten datos de forma inalámbrica usando una frecuencia.

Para contextualizar las antenas XBee es necesario comentar su procedencia. El instituto de ingenieros eléctricos y electrónicos (IEEE) dispone de un proyecto llamado IEEE802, que se creó con el fin de crear estándares para que diferentes tipos de tecnologías puedan integrarse y trabajar juntas; dentro de este proyecto se pueden encontrar algunos estándares largamente utilizados en la actualidad como: Ethernet (IEEE 802.3), Wi-Fi (IEEE 802.11), Bluetooth (IEEE 802.15). El grupo IEEE 802.15 se especializa en redes inalámbricas de área personal y la tecnología ZigBee está dentro del grupo de trabajo 4 que se centra en WPAN (redes personales inalámbricas) de baja velocidad.

El protocolo ZigBee engloba un conjunto de protocolos a alto nivel de comunicación inalámbrica basadas en el estándar IEEE 802.15.4 y se centra en comunicaciones seguras con baja tasa de envío de datos y maximización de la vida útil de sus baterías. La diferencia entre el protocolo ZigBee y el protocolo Bluetooth reside en que ZigBee realiza las comunicaciones a través de un único canal (normalmente dispone de 16 canales).

Los módulos XBee de Digi International, son pequeños módulos RF (radio frecuencia) que transmiten y reciben datos sobre el aire usando señales de radio. En concreto, se utiliza el modelo Digi XBee 802.15.4 (antena XBee serie1) que es el modelo más barato y fácil de implementar. Presenta las siguientes características:

- Comunicaciones a baja frecuencias simples e integrales sin configuración.
- Topología de red punto a multipunto.
- 2.4 GHz para implementaciones mundiales.
- Pines XBee comunes para una variedad de módulos de baja frecuencia.
- Corriente de reposo líder en la industria de sub 1uA.
- Actualizaciones de firmware a través de UART, SPI o por aire.
- Capaz de migrar a los protocolos DigiMesh y ZigBee PRO y viceversa.

Una antena XBee se comunica con otra a través del aire, enviando y recibiendo mensajes inalámbricos, pero no puede gestionar la información que recibe o envía por lo que se necesita conectar a un dispositivo inteligente (PC o microcontrolador) a través de una interfaz serial.

Para hacer posible la comunicación, los dispositivos inteligentes deben controlar y gestionar los mensajes que se envían/reciben a través de las antenas XBee. Dentro de un proceso de comunicación XBee existen dos tipos de transmisión de datos de forma inalámbrica:

- Comunicación inalámbrica. Tiene lugar entre módulos XBee, que deben formar parte de la misma red y usar la misma radio frecuencia.
- Comunicación serial. Comunicación entre la XBee y un dispositivo inteligente a través de un puerto serial asíncrono de alto nivel; cada antena XBee dispone de una UART para comunicarse con el microcontrolador o PC.

Figura 15: Esquema de comunicación con antenas XBee

En cuanto a la comunicación inalámbrica, es necesario disponer de un emisor, un mensaje y al menos un receptor. Se pueden distinguir dos modelos de comunicación.

- Comunicación punto a punto.

Forma de comunicación más sencilla que consiste en una unión inalámbrica entre dos puntos a través de dos antenas XBee. Este tipo de comunicación se puede ejemplificar con una llamada de teléfono para la que se necesita una línea de comunicación y un número de teléfono al que se pretende llamar.

En el caso de los módulos XBee la línea de comunicación es la red a la que pertenecen las dos antenas, las cuales deben estar bajo el mismo canal y tener el mismo ID, que hace referencia a la red de comunicación, mientras que el número de teléfono es la dirección de la antena de destino con la que se pretende establecer una comunicación.

Figura 16: Comunicación inalámbrica punto a punto

- Comunicación punto a multipunto.

En este modelo una antena puede comunicarse con otro módulo o múltiples módulos que estén en la misma red por lo que es necesario definir un nodo central coordinador y uno o varios nodos remotos (“*end devices*”) conectados a dicho nodo emisor. Presenta una tipología en forma de estrella lo que signifca que los datos se envían a todos los nodos posibles de la red.

En el caso del protocolo 802.15.4, los módulos XBee pueden cumplir dos roles:

  - Coordinador. Nodo central de una red que es el elemento principal de la misma y que permite a otras antenas unirse a dicha red, además, selecciona el canal de frecuencia sobre el que transmiti, Para que una antena XBee funcione como coordinador se debe cambiar el parámetro *“Coordinator Enable (**CE**)”* a *“coordinator”* dentro de la configuración.
  - Dispositivo final. Se trata de un nodo remoto de la red que se puede comunicar con esta pero también con otros dispositivos finales. Si el parámetro *“Coordinator Enable (**CE**)”* se define como “*end device*” dicha antena funciona como dispositivo final.

El coordinador puede enviar datos a un destinatario en concreto o a todos.

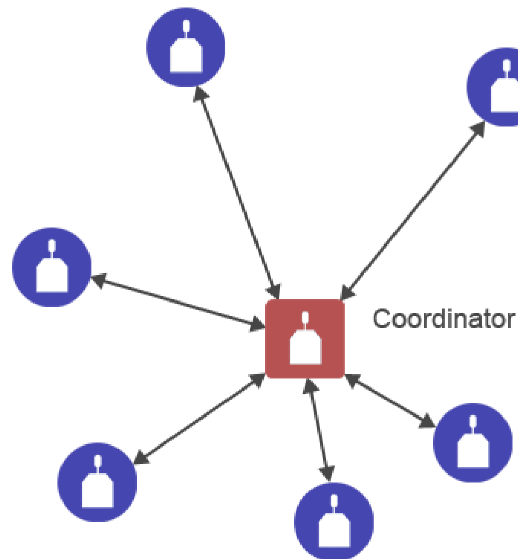

Figura 17: Comunicación inalámbrica punto a multipunto.

Tras definir los modelos de comunicación, hay que distinguir las formas de transmitir la información:

- Unicast. Consiste en enviar mensajes a un nodo sencillo de la red que se identifica por una dirección única. Para realizar una transmisión *“unicast”*, la antena XBee necesita saber la dirección de destino de la información.
- Broadcast. Los datos se envían a todos los nodos posibles de la red. Para utilizar este tipo de transmisión la dirección de destino debería ser 000000000000FFFF.

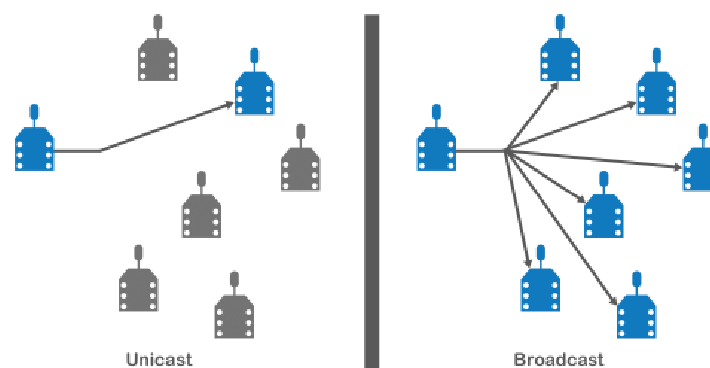

Figura 18: Transmisión Unicast vs. Broadcast

La plataforma presenta un modelo de comunicación punto a multipunto con transmisión de datos “*broadcast*”. La configuración de los ajustes de cualquier tipo de comunicación incluye:

- Canal. Controla la banda de frecuencia en la que se realiza la comunicación inalámbrica.
- PAN ID (Personal Area Netork ID): Identificador de la red que conformada por las antenas que se pretenden
- Dirección. Dentro de una red cada antena presenta una dirección asignada de 16 bits y presenta una dirección objetivo dentro de la red a la que puede enviar información.

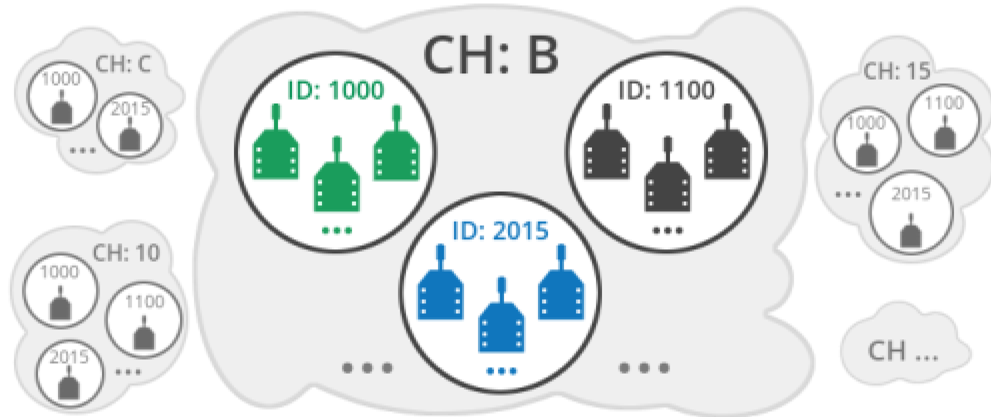

Figura 19: Elementos a tener en cuenta en la configuración de una red de antenas XBee

Una antena XBEE:

- Solo puede recibir datos de otras antenas que pertenezcan a la misma red (mismo ID) y usan el mismo canal (mismo valor de CH).
- Solo es capaz transmitir datos a otras antenas que estén en la misma red y usen el mismo canal.

En un proceso de comunicación XBee no siempre hay una comunicación serial ya que un módulo XBee puede funcionar como un módulo independiente de forma que envía datos al nodo central de la red a la que pertenece. En el caso de que el módulo XBee se conecte a un dispositivo inteligente es necesario establecer una comunicación serial a través de la UART en la que se pude configurar la velocidad de transmisión de datos y el formato de los datos transmitir.

Cualquier dispositivo externo conectado a un módulo XBee mediante puerto serie puede tener varios modos de operación en función de cómo se comunican por el puerto serie. Los módulos XBee soportan dos modos de operación:

- Modo Transparente (Aplicación transparente). La radio pasa la información tal cual la recibe por el puerto serie. Es el modo utilizado por defecto en el que todos los datos recibidos como entrada serial por un módulo XBee se transmiten de forma inalámbrica al módulo de destino, el cual los recibe y envía sin tratamiento alguno a través de la interface serial al dispositivo inteligente de destino. Es este modelo el que se ha utilizado en este proyecto.

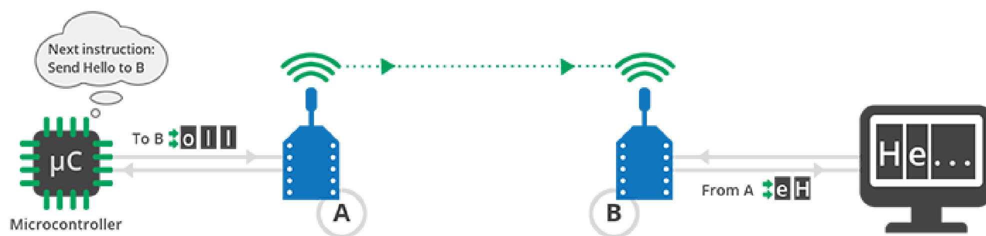

Figura 20: Comunicación XBee con puerto serie en modo transparente

- Modo API (Aplicación de programación). En este caso un protocolo determina la forma en que los datos son intercambiados. Este modo permite hacer una red de comunicaciones más grande. No es un envío secuencial de información, los datos se

envían en paquetes con un determinado orden (API frames) por lo que es necesario un tratamiento de la información a enviar para formar dichos paquetes.

.

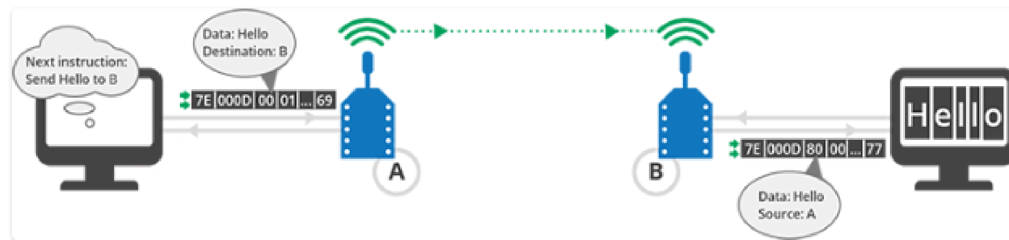

Figura 21: Comunicación XBee con puerto serie en modo API

Anexo I.C. Puerto serie

Un puerto serial no controla un dispositivo, simplemente sirve de comunicación entre el PC o microcontrolador y el hardware, en otras palabras, sólo se puede enviar bytes de información a través del puerto serial, o bien, leer esos bytes del propio puerto e interpretarlos.

La forma de acceder al puerto serie es la misma que para trabajar con cualquier otro fichero, utilizando las llamadas "*read()*" y "*write()*"; a diferencia de los archivos normales, antes de usar el puerto serie es necesario configurarlo con el fin de establecer la velocidad de trabajo y su modo de funcionamiento.

En este proyecto se trabaja con el sistema operativo Linux de modo que la lectura de datos desde el puerto serial presenta dos tipos de entrada:

- Entrada canónica. El terminal procesa las entradas como líneas por lo que lee toda la información que hay en la UART hasta que aparezca el comando de nueva línea o fin de fichero, es decir, la función "*read()*" no devuelve un resultado hasta que el usuario haya ingresado una línea completa y por tanto, no podrá leer ninguna otra entrada; la función de lectura devuelve como máximo una sola línea de entrada, sin importar cuántos bytes se soliciten. La ventaja que aporta este modo es que da al usuario la posibilidad de editar una entrada línea por línea.
- Entrada no canónica. En este modo de procesamiento, los caracteres de entrada no se agrupan en líneas, de modo que el nivel de detalle con el que los bytes son leídos se controla por las propiedades de configuración MIN (número mínimo de caracteres en una lectura no canónica) y TIME (tiempo de espera en decisegundos en una lectura no canónica). Este modelo de trabajo permite que el dispositivo reciba al menos un carácter, sobre el cual no se pueden aplicar las propiedades de edición. El comportamiento del puerto serial varía según la combinación de las propiedades antes mencionadas.
    - MIN=0 & TIME = 0.

    Se trata de una lectura no bloqueante, en la que se obtiene una respuesta inmediata desde la cola de entrada al driver. Si hay datos disponibles, estos se transfieren a un buffer, mientras que, si no hay dato a leer, la respuesta "sin datos" es inmediata. Si se ejecuta una sentencia de lectura repetidamente puede consumir una enorme cantidad de tiempo de procesado y es ineficiente por lo que es recomendable no utilizarla.

    - MIN=0 & TIME > 0.

Consiste en un tiempo de espera en el que se pueden dar diversas casuísticas: una de ellas es que haya datos en la cola de entrada por lo que dichos datos son transferidos a un buffer hasta completar su número máximo de bytes, momento en el que la ejecución de la función termina; otra situación es que no hay datos en la cola de entrada, lo que produce un bloqueo del driver hasta que llega un dato o hasta que haya finalizado el tiempo de espera; en el caso de que el driver esté en modo espera, un solo byte es suficiente para satisfacer esta llamada de lectura, por lo que los caracteres que le siguen no se leerían. Se trata de un temporizador general no un temporizador de caracteres.

- MIN>0 & TIME > 0.

La sentencia "*read()*" deja de ejecutarse en dos situaciones: cuando un determinado número de caracteres (valor "*MIN*") se transfieren al buffer o cuando un tiempo definido, entre la aparición de caracteres, expira (valor "*TIME*"). Se trata del modo de operación más común ya que considera al parámetro "*TIME"* como un temporizador de caracteres, es decir, se pone en marcha una vez se ha leído el primer carácter. Si se trabaja con esta configuración la sentencia "*read()*" nunca debería devolver cero bytes.

- MIN>0 & TIME = 0.

Se trata de una lectura enumerada que se satisface cuando al menos un número determinado de caracteres ("*MIN*") han sido transferidos al buffer que almacena la información recibida. Esta sentencia puede cumplimentarse desde la cola de entrada del driver, por lo que la respuesta será inmediata, o espera que llegue un nuevo dato, momento en el que se bloquea hasta completar el número de caracteres definido.

La función que realiza la configuración se llama "*SetUSB()*" y se ejecuta cuando se accede al puerto serial mediante la función "*open*()", tras esto se inicializa a cero la estructura "t*ermios*", interfaz de un terminal, mediante la cual se pude controlar los puertos de comunicaciones asíncronas. Dicha estructura almacena la configuración actual del puerto y establece la configuración con la que se pretende que trabaje el mismo.

Como se puede ver en el código de la función, el puerto se abre en modo lectura/escritura (O_RDWR) y no controla el proceso que se sigue en el terminal (O_NOCTTY). La configuración de un terminal contiene 4 modos:

- Modo entrada ("*termios.c_iflag*"). Se desactiva el flujo de control en salida y entrada (IXON y IXOFF), así como la opción de que cualquier carácter pueda reinicializar la salida (IXANY).

- Modo salida ("*termios.c_oflag*"). No hay ningún procesamiento en la salida.

- Modo control ("*termio.c_cflag*"). Los modos activados son:

  - CS8, estableciendo el tamaño del byte en 8 bits, sin paridad y con un bit de parada.
  - CREAD, activando la recepción de caracteres.
  - CLOCAL, definiendo una conexión local sin control del terminal.

  El activar los modos anteriores, obliga a desactivar algunas opciones:

  - PARENB, se desactiva la generación de paridad a la salida y la opción de revisar si la entrada presenta paridad.
  - CSIZE, significa que no se establece un tamaño de máscara para un carácter.

- CSTOPB, desactiva la opción de establecer dos bits de parada, con lo cual se trabaja con un bit de parada debido a CS8.
- CRTSCTS, por lo que no hay un control de flujo de salida por hardware; este modo está asociado a la configuración con la que se accede al puerto en la que no se controla el proceso (O_NOCTTY).

- Modo local (“*termios.l_cflag*”). Se procede a desactivar el modo canónico (ICANON) al igual que todas las funcionalidades del eco, y no envía señales al programa que llama al puerto.

- Caracteres especiales (“*c_cc*”). Modelan los parámetros del funcionamiento no canónico: VMIN representa el número mínimo de caracteres en una lectura canónica y VTIME es el tiempo de espera en decisegundos durante la misma lectura.

Además, se establece la velocidad de transmisión de datos en 9600 bytes por segundo.

Anexo I.D. Paquetes de mensajería

Paquetes que se envían desde PC

1. Paquete de trayectoria(“*Path-Packet*”). Contiene la trayectoria al completo asignada a cada uno de los robots. La estructura del paquete presenta la siguiente estructura de bytes:

   - Letra “P” que define el tipo de paquete
   - Número del robot.
   - Número de puntos que conforman la trayectoria.
   - Secuencia de bytes que corresponden a los puntos que forman la trayectoria.
     - “x”
     - Valor numérico de X
     - “y”
     - Valor numérico de Y

     Esta secuencia se repetirá tantas veces como puntos disponga la trayectoria.
   - Letra “F” que cierra el paquete

P	#Rob	#Puntos	x	Val.X	y	Val.Y	T	Val.Orient	...	x	Val.X	y	Val.Y	T	Val.Orient	F

Figura 22: "Path-Packet"

2. Paquete de obstáculos(“*Obstacles-Packet*”).

   - Letra “O” que define el tipo de paquete
   - Número de obstáculos
   - Secuencia de bytes asociada a cada obstáculo:
     - “x”
     - Componente X del centroide de cada obstáculo.
     - “y”
     - Componente Y del centroide de cada obstáculo.

- "T"
- Valor que hace referencia al radio máximo de influencia de cada obstáculo

Esta secuencia se repetirá tantas veces como obstáculos disponga la plataforma.

- Letra "F" que cierra el paquete

O	#Obst.	x	Val.X	y	Val.Y	T	Val.Rad Infl	...	x	Val.X	y	Val.Y	T	Val.Rad Infl	F

Figura 23: "Obstacles-Packet"

3. Paquete posición ("*Frame-Packet*"). Proporciona la posición actual y la orientación de un robot en concreto. Presenta la siguiente secuencia de bytes:

   - Letra "V" que precede al número del coche al que va dirigida su posición.
   - Número del robot
   - Letra "X"
   - Componente X de la posición de robot en cuestión en cm,
   - Letra "Y"
   - Componente Y de la posición del robot en cm.
   - Letra "T"
   - Valor en radianes de la orientación del robot
   - Letra "E"

V	# Robot	x	Valor X	y	Valor Y	T	Valor Orientación	E

Figura 24: "Frame-Packet"

La orientación del robot se obtiene a partir del lado derecho de un marcador.; dado un marcador ArUco cuyas esquinas se enumeran empezando por la esquina superior izquierda y siguiendo dirección según las agujas del reloj, su orientación es el ángulo que forma el lado derecho del marcador.

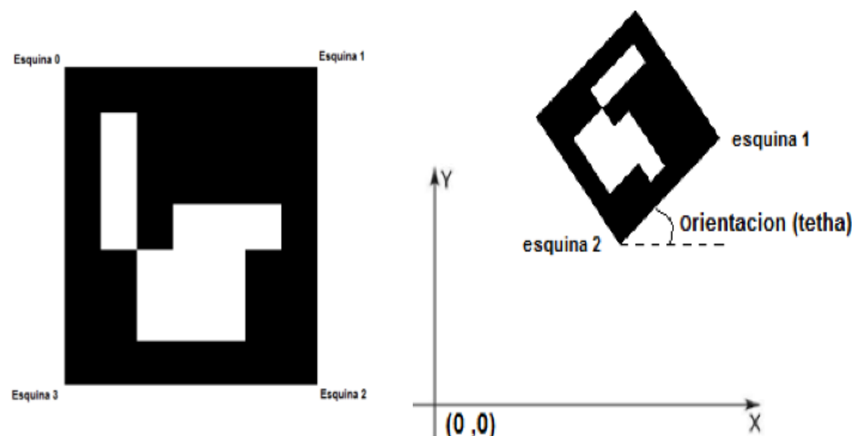

Figura 25: Orientación de un robot a partir de marcador

4. Paquete de parada (“*Stop-Packet*”). Letra “S”, seguida del número del robot que debe parar por peligro de colisión con otro robot.

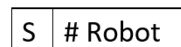

Figura 26: "Stop-Packet"

5. Paquete de inicio movimiento(“*Move-Packet*”). Letra “M”, seguida del número del robot que debe empezar a moverse a su próximo destino.

Figura 27: "Move-Packet"

Paquetes que envían los robots

Los mensajes que envían los robots se encapsulan en un único byte; los 5 primeros bits recogen el tipo de mensaje, mientras que los tres bits restantes hacen referencia al robot al que va dirigido el mensaje. En el proceso de decodificación de los cinco primeros caracteres se diferencia un primer número, conformado por los tres primeros bits desde la derecha, al que se le resta el número que se recoge en los dos primeros caracteres desde la izquierda.

- Informa de que el robot en cuestión no ha recibido la trayectoria.

Tabla 2: Caracteres asociados al tipo de mensaje 3

Componente 1	Componente 2	Robot	Numero de mensaje	Carácter
01	100	001	3	97(a)
01	100	010	3	98(b)
01	100	011	3	99(c)

- Informa que el robot ha recibido la trayectoria y esta es correcta.

Tabla 3: Caracteres asociados al tipo de mensaje 4

Componente 1	Componente 2	Robot	Numero de mensaje	Carácter
00	100	001	4	33(!)
00	100	010	4	34(“)
00	100	011	4	35(#)

- El robot vuelve a girar al estar demasiado cerca de un objeto.

Tabla 4: Caracteres asociados al tipo de mensaje 5

Componente 1	Componente 2	Robot	Numero de mensaje	Carácter
00	101	001	5	41())
00	101	010	5	42(*)
00	101	011	5	43(+)

- El robot que recibe el mensaje ha completado la ejecución del controlador proporcional respecto a la orientación.

Tabla 5: Caracteres asociados al tipo de mensaje 6

Componente 1	Componente 2	Robot	Numero de mensaje	Carácter
00	110	001	6	49(1)
00	110	010	6	50(2)
00	110	011	6	51(3)

- El robot en cuestión ha alcanzado la posición final de la trayectoria que ha recibido.

Tabla 6: Caracteres asociados al tipo de mensaje 7

Componente 1	Componente 2	Robot	Numero de mensaje	Carácter
00	111	001	7	57(9)
00	111	010	7	58(:)
00	111	011	7	59(;)

ANEXO II

Anexo II.A. Descripción de librería "*BankerAlgorithm*"

La implementación del algoritmo del banquero se engloba dentro de la librería *"BankerAlgorithm*" que incluye cuatro clases:

- obstacles. Almacena los obstáculos presentes en la plataforma; la información contenida en este objeto se envía a los robots con el fin de implementar en el controlador interno de cada robot una funcionalidad que evite la colisión de dicho robot con los obstáculos.
- Syst. Representa el sistema multi-robot.
- nextState. Permite definir un nuevo estado a alcanzar dentro del sistema. Tiene asociado el movimiento de un robot, al que se le envía una determinada trayectoria.
- Banker_Graph. Modela la aplicación del banquero.

Clase *"obstacles"*

Presenta 3 propiedades privadas, y una serie de funciones públicas que permiten trabajar con las distintas características del objeto.

- **"*list_obstacles*",** contiene la relación de obstáculos presentes en la plataforma; se trata de un tipo formado por un vector de obstáculos, entendiendo como obstáculo a la secuencia de puntos que conforman el perímetro de este; dichos puntos se definen como un vector del tipo "double"; el tipo definido de esta característica es std::vector< std::vector< std::vector<double>>> y corresponde a la salida del sistema de visión.
- **"*centroids*",** vector que contiene los centroides de los obstáculos; al igual que en la propiedad anterior, cada punto se define como un vector del tipo "double" (std::vector< std::vector<double>>).
- **"*threshold_values*",** vector del tipo "double" (std::vector<double>) que almacena el radio de influencia de cada uno de los obstáculos.

Entre las funciones de las que dispone un objeto de esta clase encontramos:

- **obstacles()**. Constructor de la clase.
- **~obstacles()**.Destructor de la clase.
- **getlistObstacles()**. Permite obtener los obstáculos que aparecen en la plataforma. El formato en el que devuelve los obstáculos es el mismo con el que trabaja el sistema de visión (std::vector< std::vector< std::vector<double>>>).
- **InitializeObstacles(std::vector<std::vector<std::vector<double>>>,ofstream &) .** Función que inicializa las propiedades de un objeto de esta clase, partiendo de los obstáculos obtenidos desde el sistema de visión; la otra variab2le de entrada es una referencia a un documento "txt" de salida, que es una propiedad pública de la clase "*Syst*", detallada en el siguiente apartado.

  Los centroides se obtienen usando la función "*obstacles_centroid()"*, mientras que los radios de influencia de cada objeto se obtienen a través de la función pública "*getInfluenceRadiumObstacles()*".

  - "*obstacles_centroid()"* es una función que se ha añadido a la librería "*obstacles_filter.cpp*"(definida previamente a este proyecto). El primer paso en esta función, es definir cada obstáculo como una secuencia de puntos cuya primera y última componente es el mismo punto; dicha secuencia de

puntos es la entrada a la función “*boost::geometry::centroid()”*, que devuelve el centroide de dicho obstáculo. La salida de esta función es un objeto del tipo std::vector< std::vector<double>>.

- “*getInfluenceRadiumObstacles()*”. El radio máximo de influencia de un obstáculo se entiende como el radio de un robot, 9 cm, sumado a la mayor distancia que existe entre el centroide de cada obstáculo y los puntos que conforman su perímetro. Para obtener dicha distancia se procede a utilizar la función, también pública, “*getMaxDistance()”*, que partiendo del centroide de un obstáculo y sus puntos perimetrales, devuelve el punto que se encuentra a una mayor distancia del centroide del obstáculo.

- **getcentroids(**). Devuelve la propiedad “centroids” del objeto.
- **getThreshold_values().** Devuelve el radio de influencia de cada uno de los obstáculos.

Subclases usadas en clase “Syst”

Paths

Consta de dos elementos privados: “*PathsNumber*”, que es el número de trayectorias que se asignan en la plataforma, y “*list*” que es un vector de punteros que hacen referencia a objetos de la clase *“Path*

(std::vector<Path*>). Las funciones disponibles son:

- **Paths().** Constructor.
- **Paths(const Paths&).** Constructor copia.
- **~Paths().** Destructor.
- **setPaths(std::vector<std::vector<std::vector<double>>>)**. Función que realiza el cambio de formato en las trayectorias. Para completar “*list*” se recorre cada una de las trayectorias de entrada y se va creando un objeto nuevo de la clase *“Path*
- , inicializado a través de la función “*Path(std::vector<std::vector<double>>)*”. A la característica “*PathsNumber*” se le asigna el número de elementos que conforman “*list*”.
- **getPaths()**. Devuelve las trayectorias como un vector del objeto *“Path*
- .
- **setPathsNumber(int)**. Asigna al objeto el número de trayectorias con las que se trabaja utilizando el entero de entrada.

  setlistPaths(std::vector<Path*>). Asigna a la característica “*list”* un objeto **“***Paths***”** que se recibe como entrada a la función.
- **getPathsNumber()**. Devuelve el número de trayectorias con las que cuenta este objeto.
- **printlistPaths(std::ofstream&)**. Imprime en el archivo “txt” propio de un objeto de la clase “*Syst*” las trayectorias.
- **deletePath(int).** Función que libera un bloque de memoria en el que se sitúa el puntero contenido en el índice de entrada dentro de la lista de trayectorias.
- **addPath(int , Path*).** Función que asigna un puntero a un objeto *“Path*

- dentro la propiedad "*list*" tomando como referencia el índice de entrada.

Path

Considera una trayectoria como una secuencia puntos que se recogen en el objeto "*list",* que es un vector de punteros al tipo "*Point2D*" (std::vector<Point2D*>), siendo "*Point2D*" una clase de la librería "*Point.h*"(ya existente en la aplicación). Además de "*list",* la clase dispone de la variable "*PointsNumber"* que contiene el número de puntos que conforman la trayectoria. Las funciones disponibles en esta clase son:

- **Path().** Constructor.
- **~Path().** Destructor
- **Path(std::vector<std::vector<double>>).** Constructor de un objeto a partir de una trayectoria de entrada en su formato original. Implica la creación de nuevos objetos de la clase "*Point2D*" que son almacenados en el vector "*list*"; "*PointsNumber*" se asigna obteniendo el número de componentes que forman propiedad "*list*".
- **getPoints()**. Función que devuelve la secuencia de punteros a los puntos que conforman una trayectoria (std::vector<Point2D*>).
- **getPointsNumber()**. Devuelve el número de puntos que conforman la trayectoria.
- **setlistPoints(std::vector<Point2D*>)**. Asigna a la característica "*list*" del objeto la secuencia de punteros a los puntos que conforman la trayectoria.
- **setPointsNumber(int).** Asigna al objeto el número de puntos que conforman una trayectoria.
- **printlistPoints(std::ofstream&)**. Imprime los puntos que se incluyen dentro de la trayectoria. Necesita de un archivo "txt" donde se imprime la información.
- **addPoint(Point2D*)**. Introduce un puntero a un objeto Point2D utilizando la función "*push_back*" que es propia de la librería "*vector*"; además, suma en uno el valor de puntos que conforman la trayectoria.
- **deletePoints()**. Elimina los punteros que contiene "*list*" y define que el número de puntos en un objeto *"Path*
- sea 0.

PathsSegment

Esta clase consta de dos propiedades: "*list"* (std::vector<PathSegment*>) y "*numPaths*" que es el número de trayectorias que se deben ejecutar en la aplicación. Las funciones con las que se puede trabajar son:

- **PathsSegment()**. Constructor.

  setPathsSegment(int , Paths &, std::ofstream &). Función que partiendo del objeto ***"Paths"*** recorre cada una de las trayectorias y comprueba que dicha trayectoria dispone al menos de dos puntos para considerarla una trayectoria válida; en el caso de que los puntos sean más de dos se crea un nuevo objeto "*PathSegment*
  usando un objeto *""Path*
  .
- **printlistSegPaths(std::ofstream&)**. Imprime en el archivo "txt" de entrada las trayectorias en el formato segmentos; para ello utiliza la función "*printlistSegments(std::ofstream&)*" ejecutada para cada objeto "*PathSegment*

- que contiene esta clase.
- **getlistSegPaths().** Función que devuelve un vector que contiene punteros a los objetos de la clase "*PathSegment*
- .
- **getnumMaxSegments().** Esta función devuelve un vector que contiene el número de segmentos que conforman las trayectorias a ejecutar por la aplicación.

PathSegment

Una trayectoria en formato segmento consta de: "*list"* (std::vector<Segment>), entendida como un vector de punteros referenciados a la clase "*Segment*", "*numSegements*", que contiene el número de segmentos que conforman una trayectoria, y "*numPoints*", que es el número de puntos que contiene la misma trayectoria.

- **PathSegment(std::vector<Point2D*>&).** Constructor que tiene como entrada una referencia a un vector de punteros a objetos de la clase "*Point2D",* los cuales se usan para crear objetos "*Segment*", definidos previamente en la librería "*Segment.h*"
- **~PathSegment()**. Destructor de la clase.
- **printlistSegments(std::ofstream&).** Imprime en el archivo "txt" de entrada los segmentos que forman una trayectoria. Esta función se utiliza dentro de la función "*printlistSegPaths()*", propia de la clase "*PathSegment*
- .
- **getSegments()**. Devuelve los segmentos que conforman la trayectoria.
- **getNumSegments().** Devuelve el número de los segmentos que conforman la trayectoria.
- **getNumPoints()**. Devuelve el número de puntos que conforman la trayectoria.

Clase "Syst"

Clase que dispone de propiedades privadas y públicas. Entre las propiedades privadas se encuentra:

- "***nRobots***". Número de robots que participan en la ejecución de la plataforma.
- "***RobotsOnDest***". Se inicia en 0 y se utiliza para tener constancia del número de robots que han alcanzado el punto final de la trayectoria que se le ha sido asignada.
- "***paths***". Objeto de la subclase ***"Paths"***, que contiene las trayectorias predefinidas. Se ha decidido implementar subclases para favorecer la programación orientada objetos, las cuales se detallan en el apartado anterior.
- "***indexed_arcs***". Vector de enteros que modela los arcos indexados del modelo discreto utilizado; dicho vector contiene el número del arco en el que se encuentra cada uno de los robots dentro de la ejecución de la trayectoria.
- "***obstaculos***". Objeto de la clase "*obstacles"*, que contiene los obstáculos de la plataforma, los centroides de los mismos y sus respectivos radios de influencia.
- "***control_access***". Vector de booleanos (std::vector<bool>) que se tiene en cuenta para ejecutar el algoritmo del banquero y analizar un posible nuevo estado.

- “***modo***”. Vector de string (std::vector<std::string>) que presenta tres posibles valores: “*active*”, “*wait*”, “*No active*”.
- “***status***”. Vector de string(std::vector<std::string>) que presenta tres posibles valores: “stop”, “turn”, “walk” , OnDestination”.

La propiedad pública es un archivo “.txt”(fout) del tipo std::ofstream en el que se lleva un registro de toda la actividad de la plataforma. Asociadas a esta propiedad, existen las funciones “*openfile()*” y “*closefile()*”, que se encargan de la apertura y el cierre del archivo. El resto de las funciones permite trabajar con las propiedades mencionadas anteriormente:

- **Syst()**. Constructor de la clase.
- **Syst(const Syst&)**. Constructor copia de la clase que recibe como entrada una referencia a un objeto de la misma clase.
- **~Syst()**. Destructor de la clase.
- **getnumRobots()**. Devuelve un entero con el número de robots que participan en la ejecución de la aplicación.
- **getnumRobotsOnDestination()**. Función que permite conocer el número de robots que han alcanzado su destino final.
- **updateRobOnDestination()**. Aumenta en uno el número de robots que han alcanzado su destino.
- **PrintnumRobotsOnDestination()**. Función que informa, en el documento “txt” de salida asociado al objeto de esta clase, el número de robots que están en su destino final.
- **setPaths(int ,Paths &).**Función que permite asignar valor a las propiedades ”*nRobots*” y “*paths*”; los valores que asigna son entradas de la función.
- **getPaths()**. Devuelve un objetivo del tipo *“Path*
- que contiene las trayectorias que se deben cumplimentar por los robots.
- **printPaths()**. Imprime en “*fout*” las trayectorias, utilizando la función “*printlistPaths()*” de un objeto
- **setIndexedArcs()**. Inicializa el vector de arcos indexados en el estado inicial, lo que significa que las componentes del vector inician en 1.
- **updateIndexedArcs(std::vector<unsigned int>)**. Asigna un nuevo valor al vector de arcos indexados.
- **getIndexedArcs()**. Retorna el valor de arcos indexados.
- **printIndexedArcs(std::ofstream &)**. Función que recibe como entrada un archivo “txt”, dónde se imprime el vector de arcos indexados. El motivo por el que se aporta la referencia al archivo de texto es porque a la hora de aplicar el algoritmo del banquero se trabaja sobre una copia del objeto “*Syst*” y se quiere tener constancia de dicho proceso.
- **set_PathsSegmentFormat()**. Función que cambia el formato de las trayectorias; se pasa de unas trayectorias entendidas como una secuencia de puntos a unas trayectorias que son una secuencia de arcos. Viene definido por la función “*setPathsSegments*” asociado a un objeto de la subclase ***“PathsSegment”***.
- **printSegmentPaths(std::ofstream &)**. Imprime las trayectorias en formato segmentos utilizando la función “*printlistSegPaths()*” de la subclase ***“PathsSegment”.***
- **getSegPaths()**. Función a partir de la cual se obtiene un objeto de la subclase ***“PathsSegment”***, que contiene las trayectorias en formato segmentos.

- **setObstacles(std::vector<std::vector<std::vector<double>>>).** Función que crea un objeto de la clase *“obstacles*” y lo inicializa con la función “*InitializeObstacles()*” utilizando la variable de entrada.
- **getObstacles()**. Devuelve un objeto de la clase *“obstacles*” que contiene los obstáculos existentes en la plataforma.
- **setControlAccess()**. Inicializa la variable “*control_access*” con todos sus componentes activos(true); el vector contendrá tantas posiciones como el número de robots(“*nRobots*”).
- **getControlAccess()**. Obtiene el vector “*control_access”*, que es una propiedad privada de la clase.
- **printControlAccess().** Imprime en fout los valores de “*control_access*”; cada posición se asocia a cada uno de los robots que participan en la plataforma.
- **updateControlAccess(std::vector<bool>)**. Actualiza el vector “*control_access*” asociado al objeto de la clase “*Syst*” con la entrada a la función.
- **setModo().** Se ha definido que todos los robots, que participan en una ejecución de la plataforma, empiezan en modo “wait”.
- **updateModo(std::vector<std::string>).** Función que se utiliza para actualizar el modo en el que se encuentra cada uno de los robots que participan en la ejecución de la plataforma.
- **printModo().** Imprime en fout el modo en el que se encuentran los robots que participan del sistema multi-robot.
- **getModo().** Obtiene el vector que contiene el modo en el que se encuentra cada uno de los robots que participan de la ejecución.
- **setStatus()**. Inicializa el vector que contiene el estatus del que parten los robots; por defecto, dichos robots parten en estatus “stop”.
- **updateStatus(std::vector<std::string>)**. Actualiza el estatus en el que se encuentran los robots.
- **printStatus()**. Imprime en fout el estatus en el que se encuentran los robots en un determinado momento.
- **getStatus().** Obtiene el vector estatus.

Struct “VertexData”

Esta estructura de datos contiene “*RobotNumber*”, que es un entero, y “*name*”, que es un “*string*”.

Typedef “Graph_t”

La clase “*adjacency_list*”, propia de la librería “*Boost*”, permite definir una estructura de grafo determinada con nombre “Graph_t”. Dicha estructura presenta dos dimensiones, la primera dimensión representa un vértice, y cada vértice contiene una estructura dimensional que es la lista de arcos asociados al mismo. Los parámetros a tener en cuenta a la hora de definir el tipo de grafo son:

- EdgeList →Almacena la lista de arcos del grafo (boost::listS).
- OutEdgeList → Contenedor de lista de vértices del grafo (boost::vecS).

- Directed → Determina si el grafo es dirigido, no dirigido, o dirigido con acceso bidireccional de los arcos. En el caso que atañe a este proyecto se ha definido un grafo dirigido (boost::directedS).
- VertexProperties → Define las propiedades que se tienen en cuenta a la hora de almacenar los vértices; se tiene en cuenta la estructura "*VertexData*".

Clase "Banker_Graph"

Esta clase se caracteriza por un grafo del tipo *"Graph_t"*, en el que se dispone de las siguientes funciones:

- **Banker_Graph()**. Constructor.
- **~Banker_Graph()**. Destructor.
- **setGraph(Syst& , std::ofstream &)**. Función que recoge la definición de un grafo "Wait for-Graph" de un estado partiendo de las trayectorias y el vector de arcos indexados, recogidos en el objeto "*Syst*". El proceso de construcción de dicho grafo aparece descrito en el apartado 4.2., y utiliza la función "*check_arc_IntOtherPath()*"; la otra entrada es el archivo "txt" donde se puede imprime el grafo y las actividades realizadas a partir del mismo.

  La función "*check_arc_IntOtherPath()*" necesita como entrada un segmento, una trayectoria y el arco indexado asociado al robot para comprobar si el segmento en cuestión forma parte de la trayectoria de algún otro robot.
- **getGraph()**. Devuelve el grafo del tipo *"Graph_t"* que es el modelo discreto utilizado para aplicar el algoritmo del banquero.
- **printEdges(std::ofstream &)**. Función meramente informativa que imprime sobre un archivo "*txt*", que es entrada de esta, los arcos que forman el grafo que es el modelo discreto.
- **printVertex(std::ofstream &)**. Presenta la misma utilidad que la anterior con la diferencia de que se imprimen los vértices.
- **BA_SafetyCheck(std::ofstream & , std::vector<unsigned int>)**. Función que modela la aplicación del algoritmo del banquero sobre un grafo no dirigido; el proceso consiste en ejecutar un bucle en el que se determina si el grafo es acíclico. El estado será seguro si el número de vértices borrados coincide con el número de vértices de los que disponía el grafo inicialmente.

Clase "nextState"

Clase que se ha definido para modelar un futuro estado que debe alcanzar el sistema y que garantiza el envío de la subtrayectoria que se le tiene que enviar a un determinado robot para alcanzar dicho estado. Entre las características de dicha clase se encuentra el estado, que viene marcado por el vector de arcos indexados, una trayectoria a enviar, una variable booleana, que determina si la trayectoria ha sido recibida por el robot, y el robot que se tiene que mover para alcanzar dicho estado. Hay que tener en cuenta que la implementación de esta clase se ha realizado considerando la particularidad de que la generación de un nuevo estado sólo puede ser motivada por el movimiento de un único robot.

- **nextState()**. Constructor.
- **~nextState()**. Destructor.

- **updateIndexedArcs(std::vector<unsigned int>)**. Actualiza el vector de arcos indexados que va asociado a un objeto de esta clase.
- **updateRobot(int**). Actualiza el robot que se mueve para alcanzar el siguiente estado.
- **setPathToSend(Syst&)**. Función que se encarga de determinar la subtrayectoria que debe enviarse a un robot para que el sistema alcance un nuevos estad, previa ejecución de las funciones; "*updateIndexedArcs()*" y "*updateRobot()*".

  Una vez conocido el robot que se debe movilizar y el segmento dentro de la trayectoria en el que se encuentra, se pueden seguir diversas lógicas:

  - La ejecución de la aplicación dispone de un solo robot, de forma que la trayectoria que se envía es una copia del objeto ***"Paths"*** que es característica del objeto de entrada "*Syst*".
  - En el caso de más de un robot, se utilizan las trayectorias en formato segmentos propias del objeto "*Syst*" que es entrada de la función y según el segmento en el que se encuentre el robot se procederá de distinta forma:
    - Si el robot se sitúa en el segmento 1, significa que la trayectoria del robot solo presenta un único arco por tanto se envían ambos puntos de dicho segmento.
    - En el caso de que el robot deba moverse al segmento 2, significa que hay que enviar los puntos correspondientes al segmento 1, y el punto final del segmento 2; el motivo de esto es que el punto final del segmento 1 coincide con el punto inicial del segmento 2.
    - En el caso de que el robot se sitúe en cualquier otro segmento únicamente hay que enviar el punto final del segmento en cuestión.

- **getSendedPath()**. Permite obtener la variable booleana que informa si la trayectoria necesaria para alcanzar el estado al que hace referencia este objeto ha sido enviada o no.
- **getRobToMove()**. Devuelve un entero que hace referencia al robot que debe moverse para alcanzar el futuro estado.
- **getPathToSend()**. Esta función proporciona la subtrayectoria asociada al nuevo estado.
- **getIndexedArcs()**. Permite obtener el vector de arcos indexados asociado a un futuro estado.
- **printInfoState(std::ofstream &fout)**. Imprime en un archivo "txt" la información asociado a un futuro estado indicando el robot que se debe mover, la subtrayectoria y si esta se ha enviado o debe enviarse.
- **updatebool(bool)**. Actualiza el estado de la variable booleana que indica si la trayectoria enviada ha sido recibida correctamente por el robot en cuestión.

Anexo II.B. Funciones usadas en control de ejecución

getNextStateByBanker(Syst& , std::vector<nextState*>&)

La aplicación del algoritmo del banquero se limita a que la referencia a la lista de futuros estados (std::vector<nextState*>) no tenga más componentes que el número de robots que participan en la aplicación y que todos los robots se puedan controlar. Para determinar si los robots se pueden controlar se ha seguido una lógica que supone

recorrer el vector "*control_access*" y comprobar que cada uno de los robots dispongan de esa variable activa("true").

Para proceder a ejecutar el algoritmo del banquero, se hace una copia del objeto "*Syst*" , en el que se asignará el vector de arcos indexados asociado a un futuro estado, y se ejecuta "*getRandomlyRobot()*" que devolverá un robot seleccionado aleatoriamente. En este punto, hay que determinar el estado de partida:

- En el caso de que lista de futuros estados este vacía, se considera como estado de partida el estado propio del sistema.
- En el caso de que haya futuros estados y no se haya completado la capacidad máxima de la lista se coge el ultimo objeto *"nextState"* de esta y se considera como estado de partida al vector de arcos indexados asociado a dicho objeto.

Tras la asignación del estado de partida, hay que comprobar que el estado resultante al mover el robot seleccionado a su siguiente arco está permitido, es decir, el robot debe estar en un segmento distinto al último segmento recogido en su trayectoria; si se da el caso de que el robot se encuentra en su último segmento hay distintas casuísticas a tener en cuenta:

- El robot seleccionado aleatoriamente se encuentra en el último segmento (la trayectoria puede tener un único arco) y su estatus es "*walk*", lo que significa que dicho robot ya está avanzando a su punto final; si, además es el único robot al que le queda alcanzar su destino, se procede a desactivar su componente en el vector "*control_access*" y salir de la función de obtención de un posible nuevo estado.
- El robot seleccionado se encuentre en el último segmento de su trayectoria, por lo tanto, se procede a seleccionar otro robot, con la condición de que el nuevo robot no esté en el último segmento de su trayectoria asignada y que no esté en estatus "OnDestination".

En este punto se actualiza el vector de arcos indexados en la copia del sistema, sumando en uno el valor correspondiente al robot seleccionado, de forma que dicho robot alcanzará el arco siguiente en su trayectoria.

Se aplica el algoritmo del banquero al futuro estado ejecutando, en primer lugar, la función "*setGraph()*" y luego, "*BA_SafetyCheck()*". Al aplicar el algoritmo del banquero se pueden dar dos situaciones:

- El estado analizado es seguro, por lo que se define un nuevo objeto *"nextState"* que contiene la información del estado analizado; además, se modifican las variables del control asociadas al robot seleccionado: el robot pasa a estar en modo "active" y no se puede acceder a controlarlo de forma que la componente del vector "*control_access*" asociada al robot en cuestión se desactiva(false).
- Si el estado es no seguro se procede a deshacer el estado analizado en el objeto temporal "*Syst*" y se selecciona otro robot para proceder de la misma manera descrita anteriormente.

control_RobotStateBA(Syst& , std::vector<nextState*>, std::vector<std::vector<double>>)

Función que trata la información recibida desde los robots y actualiza el estado de las variables de control, por lo tanto, depende de que existan datos procedentes de los robots, los cuales se recogen en la variable global "*ReceivedData*".

La información recibida desde los robots esta encriptada en único carácter por lo que hay que ejecutar la función "*decTobin()*", que devuelve el carácter recibido en binario,

obteniendo el robot al que va dirigido el mensaje y el tipo de mensaje. Se trabaja con los siguientes mensajes:

- El robot no ha recibido la trayectoria correctamente, de forma que se procede a salir de la función.
- El robot recibe correctamente la trayectoria, lo que implica un cambio en el estatus a "*turn*". Además, que un robot haya recibido correctamente la trayectoria debe reflejarse en el objeto *"nextState"* correspondiente.

  Hay una lista de futuros estados, por lo tanto, hay que asegurarse de que el robot que ha recibido el mensaje coincide con alguno de los robots asociados a los distintos estados que aparecen en la lista de futuros estados, para reflejar que la trayectoria ha sido recibida.
- El robot recibe información de que un determinado robot ha iniciado el control proporcional respecto a su orientación que se aplica al iniciar una trayectoria. El recibir este tipo de información implica que el robot pase a estar en modo "*active*", en estatus "*turn*" y una restricción de acceder al controlador ya que "*control_access*" se desactiva.
- El robot que recibe el mensaje se encuentra demasiado cerca de un objeto y se reorienta de forma que las variables de control sufren variaciones: el modo se actualiza en "*active*", estatus en "*turn*" y la componente asociada al robot de "*control_access*" se desactiva.
- El robot se ha reorientado y se pueden dar dos situaciones:
  - El robot en cuestión está en modo "*active*" y estatus "*walk*", por lo tanto, recibe este mensaje proveniente de una reorientación interna del robot en su camino al primer punto de la trayectoria y no afecta a las variables de control.
  - El robot se encuentra en modo "*active*" y estatus "*turn*", lo que implica un cambio en el modo a "*wait*", estatus permanece en "*walk*" y además, se activa posibilidad de que se acceda al control de dicho robot ("*control_access*" del robot en cuestión activada).
- En el caso de que se reciba un mensaje que indique que determinado robot ha alcanzado el punto final de la trayectoria que ha recibido, hay que trabajar con el primer objeto *"nextState"* de la lista de futuros estados y actualizar el objeto "*Syst*", borrar dicho estado de la lista de futuros estados y actualizar las variables de control (modo "*wait*", estatus "*stop*" y control_access[robot] activado).

  Esta actualización de las variables de control es general por lo que hay que tener en cuenta que el robot puede estar en su destino final, de forma que se aplica la función "*checkIsFinalState()*" para determinar si el robot ha culminado su trayectoria; en ese caso la actualización de las variables de control es otra y el robot pasa a estar en modo "*No active*", status "*OnDestination*" y la variable control_access asociada al robot desactivada.

  Tras esto, se puede tener un estado en cola, por lo que el modo del robot asociado al estado en cola se actualiza y pasa a "*active*" ya que como el estado anterior se ha completado, el robot asociado al siguiente estado puede alcanzar dicho estado. Para terminar, se comprueba usando las variables "*numRobots*" y "*numRobotsOnDesination*" si se ha alcanzado el estado final que lleva a la culminación de la ejecución.

Una vez tratada la información recibida desde los robots se procede a borrarla.

checkStatus(Syst& , std::vector<nextState*>&)

Esta función asegura de que mientras un robot está moviéndose hacia su destino ningún otro robot puede iniciar un movimiento hacia su respectivo destino. Las variables de control a tener en cuenta en este proceso son: el vector "*modo*" y el vector "*status*".

Para garantizar de que se siga el orden reflejado en la lista de futuros estados, hay que hacer un análisis de las variables de control de cada uno de los robots; en el caso de que un robot se encuentre en modo "*wait*" y estatus "*walk*", por lo tanto, ha completado su giro inicial y se dispone a iniciar el movimiento hacia su destino, se puede dar el caso de que hayan otros robots activos y esto se produce por las siguientes razones:

- Existe otro robot activo moviéndose hacia su destino, por lo tanto, el robot que ha acabado su giro debe esperar a que se cumpla el estado anterior para que pueda moverse a su destino.
- Existe otro robot activo y girando, lo que significa que un determinado robot ha hecho el giro, pero no necesariamente debe moverse ya que no se estaría siguiendo el orden de la lista de futuros estados.

Una vez hechas las comprobaciones anteriores se procede a enviar el paquete que indica al robot que puede iniciar el movimiento hacia su destino, partiendo de la información disponible en el primer estado de la lista de futuros estados.

sendSubPaths(Syst& , std::vector<nextState*>&)

Las variables de control utilizadas son el vector "*modo*" y el vector "*control_access*" que se obtienen desde la referencia al objeto de la clase "*Syst*", que es entrada de la función; la otra variable de entrada es una referencia a un vector que contiene punteros a direcciones dónde se localizan objetos de la clase *"nextState"*, que son los futuros estados a alcanzar por el sistema.

El primer punto por comprobar es que el vector que contenga los punteros a futuros estados no esté vacío, de forma que si existen futuros estados se procede a recorrer cada uno de ellos obteniendo la variable booleana que indica si la trayectoria para alcanzar dicho estado se ha enviado y el robot que se debe mover.

Para que una trayectoria se envíe, la variable booleana debe estar desactivada("*false*"), el robot que debe moverse debe estar en modo "*active*" y status "*stop*", además, en caso de que el vector de futuros estados disponga de más de un componente hay que evitar que se envíen trayectorias diferentes al mismo robot, por lo que al inicio de la función se define la variable entera "*beforeRobot*" que contendrá el ultimo robot al que se le ha enviado una trayectoria.

En el caso de que se cumplan todas las condiciones se procede a enviar las trayectorias mediante la función "*send_path_WiFi()*", la cual exige un cambio de formato realizado por "*changeFormatPathToSend()*". El enviar una trayectoria no implica una actualización de las variables de control ya que, gracias a la comunicación bidireccional, se espera que el robot envíe una confirmación de que ha recibido la trayectoria de forma correcta. Tras haber enviado la información hay que actualizar la variable "*beforeRobot*".

sendPosition(std::vector<std::vector<double>>,Syst&)

A la hora de actualizar la posición de los robots se tienen en cuenta las tres variables de control ("*control_access*", "*modo*", "*status*"), obtenidas desde la referencia a la clase "*Syst*", y las posiciones obtenidas desde el sistema de visión en cm (std::vector<std::vector<double>>).

Para actualizar la posición de un robot, este debe estar en modo "active". En un primer paso, se recorre el vector "modo" para determinar el valor de la variable "*nActiveRobots*" que hace referencia al número de robots que están activos en el momento de la ejecución de la función. Al analizar la situación de cada robot y en caso este se encuentre en modo "*active*" y estatus "*walk*", es decir moviéndose a su próximo destino, se lleva a cabo un proceso que evita la colisión de un robot activo con el resto de los robots.

La funcionalidad que evita la colisión de un robot con otros exige la comprobación de que las posiciones de los robots son válidas. Tras comprobar que estas son válidas se ejecuta la función *"distance()"* teniendo en cuenta el robot que se está moviendo en dirección a un destino, el límite de distancia entre robots que se ha definido como dos veces el diámetro de un robot y la posición de cada uno de los robots restantes; en caso de que dicha distancia sea inferior, se procede a enviar un mensaje al robot en cuestión que le hará parar y reorientarse.

Tras determinar el número de robots activos y comprobar que no hay riesgo de colisión se procede a enviar la posición de los robots, siempre y cuando el robot está en modo "*active*" y con un estatus que puede ser "*walk*" o "*turn*". La función mediante la que se actualiza la posición es "send_packet_WiFi()".

Funciones utilizadas en el algoritmo de la aplicación

- **changeFormatPathToSend(Path, std::ofstream&).** Función que cambia el formato de las trayectorias partiendo de un objeto "*Path*";obtiene el tipo std::vector< std::vector<double>>.

- **distance(std::vector<double>, std::vector<double>, float, std::ofstream&).** Función que comprueba si la distancia entre dos puntos (std::vector<double>) es menor que un valor limite que es entrada a la función.

- **decTobin(int).** Función que, partiendo de un valor entero, hace la conversión a binario del mismo utilizando el método de la escalera. La salida de la función es un array de ocho enteros.

- **checkIsFinalState(Syst&, int).** Función que recibe un entero, que hace referencia a un robot determinado, y el objeto sistema para comprobar si el robot en cuestión se encuentra en el último segmento de su trayectoria.

- **RandomlyRobot().** Obtiene un robot de forma aleatoria. No selecciona un robot que haya completado su trayectoria.

- **print(Syst& ,int ,std::vector<std::vector<double>>,std::vector< std::vector< std::vector<double>>>).** Función que imprime información en la primera ejecución; toda la información se imprime en el archivo "txt" asociado al objeto "*Syst*", que es entrada a la función. Se muestra en el archivo la salida del sistema de visión (std::vector<std::vector<double>>), los obstáculos, que son una propiedad del objeto sistema, y la trayectoria obtenida desde el proceso de planificación de trayectorias

(std::vector< std::vector< std::vector<double>>>). La variable entera que es entrada de la función indica que estrategia se ha utilizado en la planificación de trayectorias.

- **PlanificationPathProcess(int,double,double,std::vector<std::vector<std::vector< >double>>,std::vector<std::vector<double>>,std::vector<std::vector<double>>, std::ofstream&)**. Función que recoge el proceso de planificación de trayectorias con todas las estrategias implementadas hasta el momento. Las variables de entrada son: estrategia, ancho plataforma, largo plataforma, obstáculos, posición de los robots desde el sistema de visión, posiciones de destino y archivo “txt” donde mostrar la información.

Anexo III

Anexo III.A. Modelo matemático de los robots

A la hora de aplicar un controlador es necesario un modelo matemático que refleje el comportamiento de un robot; la configuración que siguen los robots es "*drive mechanism*" que consiste en el accionamiento de sus ruedas de forma independiente.

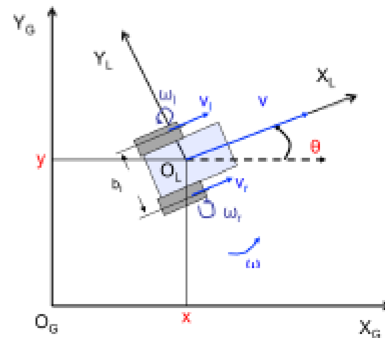

Figura 28: Robot "Differential-drive"

El robot dispone de dos ruedas que son accionadas de forma independiente con una tensión entre 0 y 3,3 V, siendo la velocidad máxima aquella que se alcanza cuando la alimentación es máxima en cuanto a tensión y que se obtiene en el proceso de calibración definiendo una distancia rectilínea que el robot recorre en un determinado tiempo.

Figura 29: Mecanismo "Differential-drive"

Combinando las velocidades aplicadas a cada una de las ruedas ($v_{derecha}$ y $v_{izquierda}$) se puede modificar la dirección del movimiento del robot:

- Movimiento rectilíneo, se consigue cuando las dos ruedas se alimentan a la misma tensión; hay que tener en cuenta la calibración en el caso de que sea necesaria.
- Giro horario. Se alimenta la rueda izquierda, mientras la rueda derecha permanece inmóvil.
- Giro antihorario. Se alimenta la rueda derecha, mientras la rueda izquierda permanece inmóvil lo que genera un giro antihorario sobre la rueda derecha.

En el caso de que no haya deslizamiento se cumple que la velocidad lineal de una rueda es el radio de esta por su velocidad angular y que la velocidad lineal del robot es el valor medio de las velocidades propias de cada rueda

$$v_{rueda} = radio * \omega_{rueda}$$

$$v_{robot} = \frac{v_{derecha} + v_{izquierda}}{2}.$$

Utilizando las hipótesis anteriores se puede definir el modelo de robot con las siguientes ecuaciones.

$$\dot{x}(t) = v_{robot} * \cos\theta(t)$$

$$\dot{y}(t) = v_{robot} * \sin\theta(t)$$

$$\dot{\theta}(t) = \frac{v_{izquierda}(t) - v_{derecha}(t)}{longitud_{robot}}$$

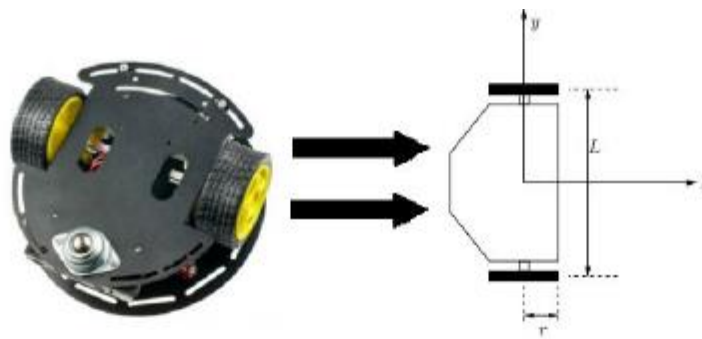

Figura 30: Modelo "Differential-drive"

Anexo III.B. Control P

En este proyecto es necesaria la implementación de un controlador proporcional sobre la posición del robot, el cual aplica un factor (K_p) a la diferencia entre la posición actual y la posición de destino, con el fin de minimizar el error. El valor de la constante proporcional influye en el comportamiento del robot de forma que si esta aumenta:

- Aumenta la velocidad de respuesta del sistema, lo cual se refleja en que el robot tarda menos en alcanzar el destino.
- Disminuye el error del sistema en régimen permanente.
- Aumenta la inestabilidad.

De esta forma se tiene que buscar un punto de equilibrio en el que se consiga una suficiente rapidez de respuesta del sistema y reducción del error, sin que el sistema sea demasiado inestable.

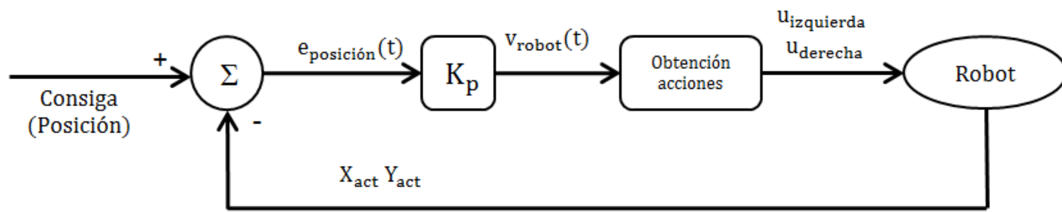

Figura 31: Esquema control P

Se ha tomado como referencia el libro [7], que describe un controlador PI, para modelar el control sobre la posición del robot, en el que se considera que la distancia euclídea entre la posición actual del robot y el punto de referencia buscado es el error en cada instante.

$$\mathrm{error}(t) = \sqrt{(x_{ref} - x(t))^2 + \left((y_{ref} - y(t))\right)^2}$$

El objetivo del controlador es determinar la acción que se debe aplicar en cada instante a las ruedas por lo que el primer paso es determinar la velocidad del robot partiendo de la posición y orientación obtenida del modelo en cada instante.

$$v(t) = K_p * \mathrm{error}(t) + K_i * \int \mathrm{error}_d(t) * dt$$

En este proyecto se ha implementado un controlador P, con lo cual la segunda parte de la formula no se tiene en cuenta. Por otro lado, la posición en cada instante del robot cambia lo que modifica la orientación de destino que pretende alcanzar el robot, con lo cual es necesario definir un control proporcional sobre la orientación.

Se calcula la diferencia entre la orientación de destino actualizada y la orientación del robot en cada instante y se le aplica un factor proporcional, manteniendo el valor de la dirección del robot entre [a,2π).

$$\begin{cases} \theta_{destino}(t) = \tan^{-1}\left(\frac{y_{destino} - y(t)}{x_{destino} - x(t)}\right) \\ \\ \alpha(t) = K_h * (\theta_{destino}(t) \ominus \theta(t)) \end{cases}$$

α es la entrada de control que modela el ángulo de accionamiento del robot. Para obtener la velocidad de cada una de las ruedas utilizamos las dos entradas de control del robot (velocidad y ángulo) y aplicamos las fórmulas del modelo "*Car-like robot*"

$$\dot{\theta(t)} = \omega(t) = \frac{v(t)}{radio}$$

$$radio = \frac{longitud_{robot}}{\tan\alpha(t)}$$

$$\frac{v_{izquierda}(t) - v_{derecha}(t)}{longitud_{robot}} = \frac{v(t)}{radio} = \frac{v(t) * tan\alpha(t)}{longitud_{robot}}$$

El sistema de ecuaciones utilizado para obtener la velocidad de las ruedas sería:

$$v_{izquierda} - v_{derecha} = v_{robot} * tan\alpha(t)$$

$$v_{robot} = \frac{v_{derecha} + v_{izquierda}}{2}$$

Despejando:

$$v_{derecha} = v_{robot} - \frac{v_{robot} * tan\alpha(t)}{2}$$

$$v_{izquierda} = v_{robot} + \frac{v_{robot} * tan\alpha(t)}{2}$$

El rendimiento del controlador depende en gran medida de las constantes k_p, k_i, k_h. La salida del controlador, previo paso a las unidades correspondientes se alimenta a las ruedas y sirven, junto con la ubicación para estimar la posición de la siguiente iteración.

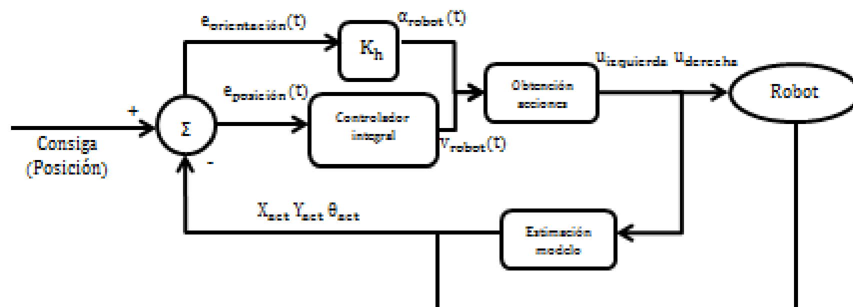

Figura 32 : Control P considerando variación de orientación destino

Si al controlador proporcional se le añade un integrador que es una acumulación del error a lo largo del proceso, dicha acumulación aumenta a medida que pase el tiempo de forma que se conseguirá reducir el error en régimen permanente; en contraposición se añade una inercia que puede hacer más inestable el sistema.

A dicha acumulación del error se le aplica un factor (K_i) que si se aumenta produce los siguientes efectos:

- Disminuye el error del sistema en régimen permanente.
- Aumenta la inestabilidad del sistema.
- Aumenta un poco la velocidad del sistema.

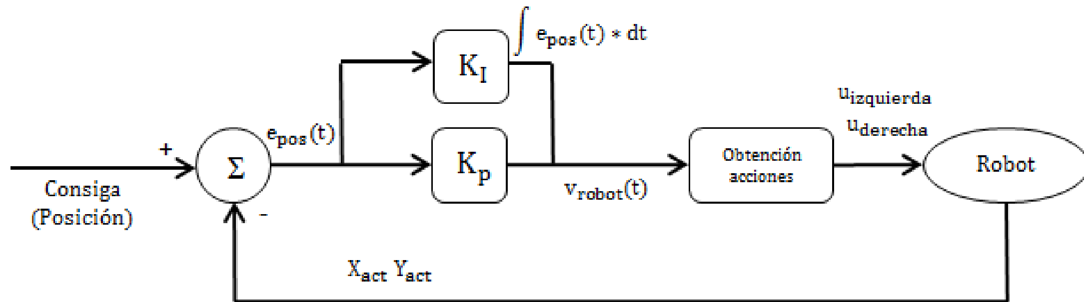

Figura 33: Esquema control PI

Anexo III.C. Control interno de los robots

El control interno de los robots utiliza la librería "*StandardCplusplus*" y es idéntico para los tres robots, lo que implica que existan ciertas variables que deben modificarse según el robot:

- I_AM, identifica al robot en cuestión.
- v_max, es la velocidad máxima alcanzada por el robot en un movimiento recto teniendo en cuenta la calibración propia del robot ("*cal_value*").
- "cal_value", es la diferencia que existe entre las acciones que se aplican a las ruedas para que el robot se movilice de forma recta.
- "veld_turning" y "vels_turning", son las acciones de giro utilizadas en la estimación del giro "e*stimationmode_rot()*".
- Los valores de los mensajes de respuesta ("*messageWrongPath*", "*messageRightPath*", "*messageTurnAfterWarning*", "*messageEndTurning*", "*messageOnPos*").

Arduino exige la existencia de la función "*setup()*", que define la velocidad de transmisión de los puertos seriales utilizados por el robot, y "*loop()*" que se ejecuta ininterrumpidamente. Dentro de "*loop()*", se ejecuta la función "*controller_final()*" previa ejecución de "*ReadObstaclePacket()*" y "*ReadPathPacket()*", que se encargan de recibir los obstáculos que conforman la plataforma y la trayectoria a cumplimentar por el robot respectivamente.

La función "*controller_final()*" modela el control interno del robot que puede trabajar en cuatro modos: "girar", "esperar", "avanzar", "check"; estos modos son modelados por las siguientes funciones:

- Turning().

El robot está parado mientras espera la llegada de un paquete que contenga una actualización de su posición ("*Frame-Packet*") mediante la ejecución de la función "*ReadVisionPacket*()", la cual devuelve un booleano que refleja si la posición leída, que incluye orientación, es adecuada para este robot. Con la confirmación de que la posición es válida se procede a guardarla y a definir el punto al que el robot se debe mover.

Tras determinar el punto de destino, se calcula la diferencia de orientación existente entre dicho punto y la posición actual, la cual no debe superar un valor límite de 10º;

el robot debe ejecutar una estimación de giro("*estimationmodel_rot()*") hasta cumplir dicha condición. Este proceso se puede considerar como un controlador porporcional sobre la orientación.

Cuando se disponga de una diferencia de orientación que no supere el valor límite, el robot pasa a estar en modo "esperar", lo que significa que espera la señal desde PC para iniciar su movimiento.

- Waiting().

El robot espera parado la llegada del paquete que da la señal de inicio del movimiento("*Move-Packet*"); una vez recibido dicho paquete, se procede a calcular el valor límite para considerar que un robot está en el destino, el cual depende de la distancia a recorrer. Tras definir el valor limite el robot cambia a modo "avanzar".

- movement().

Función que se ejecuta cuando el robot se encuentra en modo "avanzar". Durante el movimiento del robot, la posición de este va cambiando según un modelo que estima dicha posición usando "*estimationmodel*()". Se ha añadido la capacidad de actualizar la posición del robot con información procedente del sistema de visión, de forma que cuando llegue un paquete "*Frame-Packet"* y este sea válido, se procede a actualizar la posición del robot consiguiendo una ejecución más cercana a la realidad.

Cada vez que se actualice la posición del robot, desde el modelo o desde el sistema de visión, hay que almacenar dicha posición y calcular la diferencia de orientación entre la posición actual y la posición de destino.

Otra capacidad añadida es la recepción del paquete "*Stop-Packet*" que indica que el robot corre peligro de colisión con otro robot, lo cual supone que el robot cambia a modo "check", es decir, el robot debe parar y comprobar su posición.

Con la posición del robot actualizada, se procede a asegurar de que este no colisionara con algún obstáculo; en caso el robot se encuentre en el radio de influencia, este parará y cambiara a modo "check".

Una diferencia de orientación superior a 28º implica una desviación del movimiento del robot, por lo tanto, en caso de que la diferencia de orientación no cumpla dicha condición, el robot cambia a modo "check".

Tras todas las comprobaciones se procede a ejecutar el controlador proporcional respecto a la posición, "*controlP()*", que determina el "*error*", entendido como la distancia hasta el destino, y las acciones que reciben las ruedas de los robots.

Después de la aplicación del controlador proporcional se puede dar el caso de que el error sea menor al límite fijado, lo que lleva al robot a cambiar al modo "check", mientras que en el caso de que no se haya alcanzado el valor limite se pueden dar tres situaciones:

- Que la variable "*firstMov_done*" este desactivada, reflejando que el robot no ha iniciado el movimiento; esta situación se da cuando el estado anterior del robot ha sido "esperar" o cuando se recibe información desde el sistema de visión, momento en el que las variables "*error*", "*error_old*", que almacena el error del estado anterior y "*firstMov_done*" se inicializan. Tras activar "*firstMov_done*", se procede a aplicar la función "*move_forward*()" que aplica las acciones a los motores del robot.
- El robot retrocede, situación que aparece cuando el error en el estado actual "*error*" es 0,5 veces mayor que el error en estado anterior "*error_old"*. En este caso el robot para y pasa a estado "check".

  - La variable "*error*", que refleja error en el estado actual, es inferior a "*error_old*", error del estado anterior, por lo tanto, se procede a ejecutar "*move_forward()*", previa aplicación de la calibración propia de cada robot.

- checkingProcess().

  El robot se encuentra parado, a la espera de la llegada de un "*Frame-Packet*"; si la posición recibida es válida y corresponde al robot en cuestión, se procede a calcular la distancia existente entre la posición recibida y la posición de destino. Se ha fijado como radio de destino 15 cm, de forma que, si la distancia es mayor a ese límite, el robot pasa a estado "girar", mientras que, si la distancia es menor al límite, hay que determinar si el robot se encuentra en:

  - Su destino final, con lo cual finaliza la ejecución de "*controller_on()*", se inicializan las variables "*path*",que contiene trayectoria, "*mem_index*", que es índice de puntos dentro de las trayectorias y se desactiva el estado "avanzar".
  - Destino intermedio, con lo cual se suma uno a "*mem_index*" y el robot cambia a estado "girar", por lo tanto, el robot inicia el proceso para alcanzar el siguiente punto en la trayectoria.

Las funciones, anteriormente descritas, utilizan una serie de funciones que se detallan a continuación:

- stop(void). Función que alimenta de forma nula los motores del robot.
- stopandcheck(void). Función que alimenta de forma nula los motores del robot; se diferencia con la función anterior en que activa la variable booleana "*ckeckPos*", por lo tanto, el robot cambia a modo "*check*"; además inicializa a 0, tanto las acciones de las ruedas, como las variables que almacenan el error del estado actual y del estado anterior. Se asegura de que las variables booleanas, que hacen referencia a otros modos de trabajo del robot estén inicializadas de forma que el robot únicamente trabaje en modo "*check*".
- move_forward(int, int). Aplica unas determinadas acciones, que son entrada de la función, a los motores del robot con el fin de conseguir un movimiento rectilíneo hacia adelante.
- turn_right(int,int). Función que utiliza las acciones de entrada para llevar a cabo un giro a derechas del robot.
- turn_left(int,int). Función idéntica a la anterior con la diferencia que modela un giro a izquierdas.
- ReadObstaclesPacket(). Función que recibe los obstáculos de la plataforma, ejecutándose tras recibir el carácter "O". Dentro de esta función se utilizan las funciones propias de Arduino: "*parseInt()*", que recoge un entero desde el puerto serial, y "*parseFloat()*", que recoge un float desde el puerto serie; una vez leídos los obstáculos, se hace una comprobación de que los centroides estén dentro de la plataforma.
- ReadPathPacket(). Se ejecuta cuando al puerto serie llega un carácter "P" de forma que se inicia el proceso para recibir un "*Path-Packet*". El proceso consiste en usar "*ParseInt()*" para detectar a que robot va dirigida la trayectoria contenida en el paquete. Si la trayectoria corresponde al robot en cuestión se procede a leer el número de puntos que conforman la trayectoria usando "*ParseInt()*" para después, almacenar todos los puntos que conforman la trayectoria usando "*ParseFloat()*". Una vez se dispone de todos los puntos hay que comprobar que el número de puntos recibidos es el adecuado y que dichos puntos se encuentran dentro de la plataforma para poder enviar el mensaje que indica que la trayectoria

recibida es adecuada; en caso contrario, se envía el mensaje indicando que la trayectoria es incorrecta. Si la trayectoria es correcta se activa la variable booleana que ejecuta "*controller_on()*" y el robot pasa a estar en modo "girar".

- ReadVisionPacket(). Se ejecuta al recibir el carácter "V" y consiste en detectar el robot al que va dirigido usando "*ParseInt()*"; la función devolverá un booleano que indica si la posición recibida es válida o no. La posición no es válida cuando el paquete corresponde a otro robot o cuando la posición recibida es incorrecta, lo que significa que las componentes del punto recibido son menores que cero o la orientación del robot es 0. Tras leer la posición y la orientación, usando "*parseFloat()*" se procede a almacenar la posición.
- storePosition(). Consiste en asignar la posición (punto medio del robot, "*xact*" y "*yact*", y orientación "*thetaact*") a las variables que almacenan la posición en el estado anterior ("*xact_old*", "*yact_old*" y "*thetaact_old*").
- cleanProcess(). Se utiliza en la estimación del giro y se utiliza para borrar toda la información recibida por el puerto serial durante el giro.
- calculate_diffdegree(bool&, bool&). Función que devuelve la diferencia de orientación, en grados, existente entre el punto actual en el que se encuentra el robot y el punto de destino; además, actualiza las variables booleanas "*diff_Positive*" y "*counterClockWise*", que modelan si el giro del robot es a derecha o a izquierdas.
- Estimationmodel_rot(int rot_degree, bool&, bool&). Esta función realiza una estimación del giro hasta alcanzar el valor entero de entrada. Consiste en un bucle en el que se va estimando la orientación con un determinado periodo y considerando la acción mínima (50 pasos). En cada ejecución del bucle se alimentan los motores del robot con una acción de giro estipulada para cada robot ("veld_turning" y "vels_turning").

  Tiene en cuenta si la diferencia entre la orientación actual y la orientación de destino es positiva o negativa y a partir de lo anterior, si el giro es horario o antihorario. Una vez finalizado el giro procede a ejecutar "*cleanProcess()*" que limpia el puerto serie de la toda la información recibida durante el giro.
- calculate_diffdegreeMovement(). Calcula la diferencia de orientación entre la posición actual y la posición de destino; se diferencia de "*calculate_diffdegree()*" en que no aporta información sobre el sentido del giro, lo que disminuye el coste computacional del proceso.
- estimationmodel(). Aplica las ecuaciones del modelo "*drive-mechanism*" (Anexo III.A) para estimar la posición del robot partiendo de las acciones del instante anterior.
- ControlP(). Modela el controlador de posición que determina la acción que se debe aplicar a los motores del robot (Anexo III.B). Para determinar la acción que se aplica a cada una de las ruedas hay que conocer la velocidad del robot, que se obtiene con un control proporcional respecto a la posición, y el ángulo de accionamiento del robot, que se determina con un controlador proporcional respecto a la orientación.

  - Control Proporcional de la posición. Determina la velocidad del robot de forma proporcional al error, que se entiende como la distancia entre la posición actual del robot y la posición de destino. El valor de la constante proporcional (k_p) es 1.2, de forma que el punto de destino se alcanza con mayor velocidad.

- Control Proporcional de la orientación. La posición actual del robot cambia en cada ejecución, por lo tanto, la orientación de destino va cambiando con ella; esto obliga a aplicar un controlador proporcional a la diferencia de orientación entre la posición actual y la posición de destino con el fin de obtener el ángulo de accionamiento del movimiento del robot. Dado que k_h es 1, la diferencia de orientación coincide con el ángulo de accionamiento, el cual se redondea; esto implica que una diferencia de orientación mayor que 28º supone un giro del robot.

Con la velocidad del robot y el ángulo de accionamiento se procede a obtener la velocidad de cada rueda, a partir de la cual se obtiene la acción a aplicar a cada una de ellas; hay que tener en cuenta que estas acciones deben superar 50 ya que sino el robot no se movería.