Robot Mensajero

Universidad de Alicante - Grado en Ingeniería Robótica

José Ángel Martín González y Daniel Frau Alfaro

***Resumen*— La robótica móvil consiste en la creación y programación de robots que sean capaces de moverse por el entorno con cierto grado de libertad, a la vez que obtienen la capacidad de conocer el espacio gracias a mapas o recursos similares. Esta versatilidad les dota de una utilidad para resolver ciertas tareas que otro tipo de autómatas no serían capaces.**

Estas características junto con el uso de herramientas como la Inteligencia Artificial (IA) y de estructuras de decisión como las máquinas de estado resultan en comportamientos que parecen ser inteligentes (entender palabras, decidir lugares a los que ir ...).

***Palabras clave*—Robot Mensajero, SLAM, mapeado, localización, reconocimiento de texto**

I. Introducción

En el presente informe se va a describir el desarrollo de la tercera práctica de la asignatura de Robots Móviles. Durante la duración de la misma, se creó un robot mensajero en la plataforma ROS de Ubuntu, haciendo uso de diferentes tecnologías como puede ser IA o SLAM, implementando los algoritmos mediante el lenguaje de programación Python. Además, el robot que se utilizó en la práctica fue el Turtlebot3 en simulación, pues se utiliza la versión de Ubuntu 20.04, mientras que en el laboratorio se hace uso del Turtlebot2 junto con un brazo robótico de la marca Interbotix.

El desarrollo de la práctica se dividió en distintas secciones, de manera que se puedan desarrollar en paralelo y aplicarlas juntas una vez se tuvieran terminadas. Esto es posible gracias al funcionamiento de ROS mediante *topics* y nodos. Así, se establecen los medios de comunicación a través de los *topics*, sin tener en cuenta como funciona cada nodo internamente.

La sección central del proyecto es un máquina de estados, que define el comportamiento del robot a gran escala. Desde aquí se recoge información de cada uno de los nodos para garantizar el funcionamiento de la aplicación; moverse a una posición, recoger información de los sensores, ... Para crear la máquina de estados se usará SMACH, un *software* preparado para crear este tipo de estructuras, definiendo transiciones, mensajes y otros elementos necesarios.

Otra parte del proyecto sería la creación del mapa y la posterior localización del robot en el entorno. El mapa sirve para poder definir posiciones que se corresponden con las localizaciones de los destinos donde llevar las cartas. La creación del mapa se hace con el algoritmo de creación de mapas que ofrece ROS a través de SLAM, mientras que la localización se realiza con el algoritmo AMCL, que también viene en la plataforma de ROS.

Por otro lado, se necesita de una manera para que el robot sea capaz de interaccionar con su entorno a la hora de coger la carta del usuario. Así, se utiliza el brazo robótica disponible en los Turtlebot del laboratorio.

Finalmente, se necesita de una manera para que el robot sepa la dirección a la que debe dirigirse. Con el objetivo de que el comportamiento del Turtlebot se asemeje al de un mensajero real se hace uso de la cámara del robot, de manera que al mostrarle la carta a enviar sea capaz de extraer la información necesaria para concer el destino al que debe dirigirse.

II. Objetivos

En este proyecto, se valora la capacidad de un robot móvil para la entrega de cartas en un entorno conocido. De esta manera, el objetivo de la práctica consiste en la programación de un robot capaz de moverse por un espacio conocido hasta una posición determinada a través de la información ofrecida por la cámara. Además, el robot debe ser capaz de coger la carta y ofrecerla al llegar al destino.

Por otro lado, el desarrollo debe incorporar partes en simulación y pruebas con el robot real simultáneamente.

III. Organización y seguimiento

La organización y seguimiento del proyecto se llevó a cabo en el repositorio *online* de GitHub [1]. En el proyecto presenta dos ramas, la que se corresponde con el robot real y la que se ejecuta con la simulación en Gazebo.

En cuanto al repositorio, se tienen tres paquetes:

- **Turtlebot**: paquetes para la simulación y operación del Turtlebot2.
- ***simulation***: paquetes para la simulación de Turtlebot3.
- **robot_cartero**: paquete para ejecutar los nodos del proyecto del robot mensajero.

Además de estos hay otras carpetas como las de los mapas o las de los modelos de Gazebo, que se explicarán a lo largo del informe.

IV. Desarrollo

A continuación se expondrá el desarrollo del proyecto, explicando cada una de las partes con detalle.

A. *Máquina de estados*

La máquina de estados es la parte central del proyecto, pues es la que decide que hacer en cada uno de los estados, llamando a acciones y permitiendo la ejecución de otros nodos. Además es la estructura que define el comportamiento del robot, orientándolo a que sea sencillo de utilizar para los humanos, ya que un robot mensajero es una aplicación enfocada a los servicios, por lo que es muy importante que las vías de comunicación sean cómodas e intuitivas.

La máquina de estados ha sido implementada con SMACH [2], una arquitectura para la creación de comportamientos complejos para robots en ROS [3] en lenguaje Python. Se trata de una aproximación a nivel de tarea, esto es, de alto nivel, por lo que los estados deben ser generales, primando la abstracción y claridad frente al rendimiento. Los estados se representan mediante clases de Python en las que se ejecutan una serie de comandos en un método de la clase con nombre predefinido (*execute*), a la vez que se envía información entre los estados a través de entradas y salidas preestablecidas. De esta manera, un estado tendrá las mismas salidas que entradas tenga el estado siguiente. Además, para cambiar el estado actual se tienen transiciones; al finalizar la ejecución de la función de estado se devuelve un mensaje indicando cual es el resultado u *outcome* que se debe seguir. Así, las transiciones, mensajes e incluso los estados se definen en el *main* del programa, especificando que tipos de mensajes se van a enviar así a que estado corresponde cada transición, creando así la máquina de estados completa.

En cuanto al caso del robot mensajero, se deben tener en cuenta varios aspectos relacionados con su ejecución; debe ser capaz de recoger y devolver la carta, debe poder moverse a una ubicación y ha de ser capaz de recibir comandos de destino. Cabe destacar que las entradas que tiene el humano con el robot son los tres botones del Turtlebot del laboratorio, que se adaptarán en la simulación. Así, se tienen los siguientes estados:

- **Reposo**: estado en el que no se realiza ninguna acción. Se trata de un bucle infinito con tres posibles acciones:
  - Empezar la detección de la imagen, es decir, se cambia de estado con el botón 0.
  - Se lleva el robot a la dirección de *home* presionando el botón 1.
  - El robot se dirige a la posición anterior a la que se encuentra actualmente con el botón 2. Esta funcionalidad es útil si se quiere devolver la carta al anterior remitente sin tener que pasar por la fase de detección.

  En el caso de las dos últimas acciones, cabe destacar que se efectúa una transición al tercer estado, saltándose el de detectar la imagen. Cada una de las tres transiciones se activa pulsando un botón.

- **Detectar**: en este estado se habilita la detección de texto por parte del nodo de la cámara. En esta clase es donde se definen todos los posibles destinos del robot en el mapa. Al inicio de la ejecución, se envía un mensaje al nodo de la cámara para que inicie la ejecución y se espera en un bucle infinito hasta tener respuesta con la dirección especificada. Este destino se envía a los subsiguientes estados. Además, se puede volver al anterior estado pulsando el botón 1, cancelando la ejecución. La transición de este estado se produce una vez se recibe un destino desde el nodo de reconocimiento de texto por carta.

- **Imagen leída**: este estado indica que hay un destino al que ir, por lo que recibe la posición del estado Detectar. Luego, esa misma posición se envía al siguiente. La transición de este estado se produce de manera automática.

- **Recoger carta**: este estado recibe la posición de destino y la pasa al siguiente. Además, se mandan comandos al nodo que ejecuta el brazo con el fin de recoger la carta, esperando unos segundos hasta tenerla agarrada en la pinza. La transición del estado se produce después de que pase el tiempo para coger la carta y de manera automática.

- **Ir al destino**: en esta clase se llama al nodo para ir a un destino específico, determinado por el estado Detectar. Para evitar llamar al servicio desde la máquina de estados (ya que SMACH da problemas a la hora de enviar *goals* a acciones) se creó un nodo a parte que realizara esta acción.
  De esta manera, en la máquina de estados se envía el mensaje *goal* al nodo y se espera en un bucle. Luego, en el nodo que llama a la acción se sale del modo en reposo y se ejecuta el *callback* del *topic* por el que llega el mensaje de la máquina de estados con la posición. Finalmente, se ejecuta la acción, guardando la posición inicial del movimiento, pues esta será la dirección del remitente del nuevo destino. Al llegar al destino, se envía esta dirección a la maquina de estados, que vuelve a su ejecución normal, solo que ahora el mensaje que se transmite entre estados no es la posición objetivo, sino la del remitente. La transición del estado se produce después de llegar al destino.
  Cabe destacar que en el robot real la especificación de la dirección de destino se hace de manera distinta a la de la simulación. En esta última se crea un cliente para la acción creada por el algoritmo de localización, de manera que el mensaje se publica desde ahí. En el Turtlebot real no se podía publicar de esta manera; aparecía la publicación pero el algoritmo no se ejecutaba. Así, se decidió publicar manualmente en el *topic goal*. Al hacerlo, se generan mensajes por los *topics* de *feedback* y *result*, a los que se suscribe para obtener la posición inicial y detectar cuando finaliza la ejecución.

- **Llega al destino**: este estado existe para indicar que se ha alcanzado un estado satisfactorio, pasando la dirección del remitente del anterior estado al siguiente. También se indica al usuario que se ha alcanzado la posición emitiendo sonidos por el altavoz. La transición de este estado se produce de manera automática.

- **Entregar carta**: por último se tiene el estado encargado de recoger la carta. Aquí, se espera a que el destinatario pulse el botón 0. Luego, se ejecuta la acción de soltar la carta tras un indicador al humano, para después volver al estado de Reposo.

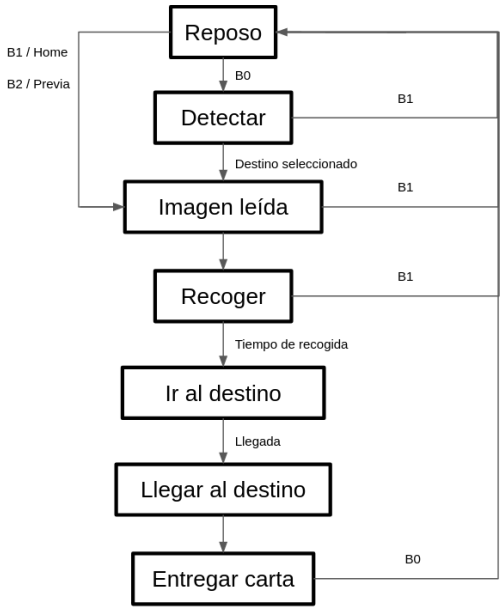

Fig. 1. Esquema de la máquina de estados del robot mensajero

En la figura anterior se puede ver con más claridad la estructura de la máquina de estados.

Hasta aquí se tiene el funcionamiento de la máquina de estados a nivel general, pero en la práctica el desarrollo se dividió en dos partes, la simulación en un escenario virtual y el robot real, siendo ambos similares pero no por ello iguales por los problemas que conlleva cada uno de ellos. Esto es debido a que en cada uno de los casos se hace uso de un robot diferente. En la simulación se usa el Turtlebot3 mientras que los robots del laboratorio corresponden al Turtlebot2.

1) ***Robot real****:* Como se mencionó anteriormente, el modelo de robot usado es el Turtlebot2. Así, las transiciones con los botones que se usan en la máquina de estados se implementan con los botones con los que viene equipado. Además, se implementan una serie de indicadores luminosos (gracias a los leds de la plataforma móvil) y sonidos por el altavoz del Trutlebot, que marcan los estados en los que se encuentra el robot.

En cuanto al brazo, se pudo actuar para la recogida y entrega de cartas.

2) ***Simulación****:* A diferencia del desarrollo con el robot real, en la simulación no se pudo accionar los botones del Turtlebot o activar leds y sonidos del mismo. El brazo tampoco pudo ser actuado por que el modelo URDF del Turtlebot3 en Gazebo no tiene un brazo robótico incorporado.

Aun así, las pulsaciones de los botones son una parte clave del proyecto, pues permiten al usuario elegir que tipo de comportamiento realizar. Así, se usó la librería Pynput [4], que permite registrar las pulsaciones de las teclas para generar *callbacks*. Para hacer más versátil la máquina de estados y que difiera tanto de la del robot real se creó un nodo a parte que simule los *topics* de los botones del Turtlebot2. Además, el *callback* para los botones de la máquina de estados es diferente, ya que no se usan los botones. Los botones *B0, B1, B2* se mapean a las teclas numéricas *0, 1, 2* del teclado.

B. Mapeado y localización

1) ***Mapeado****:* El mapeado consiste en la creación de un mapa por el que el robot trabajará y se desplazará, de manera que sea capaz de conocer su ubicación en todo momento a la vez que es capaz de moverse por el entorno. Para construirlo se utilizó el paquete de *slam_gmapping* [5] [6] [7] implementado en ROS, un conjunto de paquetes que implementan el algoritmo de SLAM en ROS. A grandes rasgos, SLAM es un algoritmo que trata de localizar al robot a la vez que se construye el mapa [8]. Estas dos actividades dependen cada una de la otra; el robot no puede localizarse sin tener un mapa, a la vez que no puede construir un mapa sin conocer su ubicación en todo momento. SLAM es el método por el que se soluciona esta disyuntiva. Concretamente, en ROS se usa el algoritmo *Gmapping*, que trabaja con mapas de rejilla y filtros de partículas Rao - Blackwellizados, donde las posiciones del robot son independientes entre si, dependiendo solamente de las lecturas de un sensor láser (no se usan lecturas de odometría). Al usar un filtro de partículas se esparcen muestras alrededor de la posición inicial del robot, que se tomará como el inicio del mapa, de manera que cada una de ellas mantiene una estimación del mapa que se está construyendo. Después de cada medida, el peso de la partícula se actualiza acorde con lo que se detecta y se toman nuevas muestras a partir de las anteriores siguiendo la odometría del robot o la orden de movimiento, aunque en ROS se usa la odometría para realizar la proyección de los movimientos de las partículas. La detección en un instante de tiempo se añade al mapa general de cada una de ellas, generando así el mapa global del entorno.

El mapa se construyó moviendo el robot por el entorno con el nodo de teleoperación para el Turtlebot [9] [10]. Durante el desarrollo de la práctica, se siguieron dos líneas de trabajo que coinciden con las mencionadas anteriormente; un mapa en simulación que se asemeje al del laboratorio de la universidad y un mapa real que se corresponda con la clase real del laboratorio. De esta manera, en Gazebo se creó un modelo a partir del *building_editor_model* buscando parecerse a la forma del aula. Después, haciendo uso del *model_editor_models* se crearon lo que serían las mesas y muebles presentes en laboratorio. Finalmente, el escenario creado quedaría:

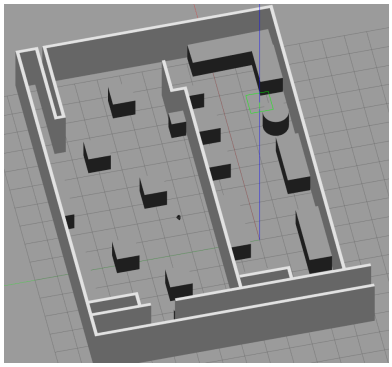

Fig. 2. Escenario del laboratorio en simulación de Gazebo

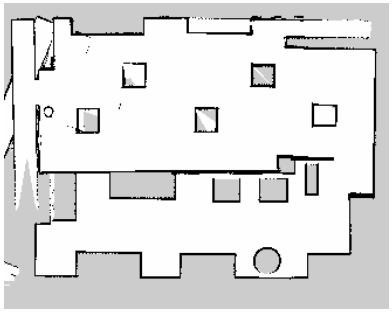

Fig. 3. Mapa del laboratorio en simulación

Por otro lado, el mapa de la clase se construyó haciendo uso de los Turtlebot2 del laboratorio y ejecutando los algoritmos de SLAM en la consola del propio robot en los ordenadores del aula (para así tener acceso a la red local por cable). Finalmente, se obtuvo el mapa del laboratorio real:

Fig. 4. Mapa del laboratorio real

En ambos casos se siguió el mismo modo de construcción; mover el robot por el mapa de manera que cada vez que se desplace a una distancia alejada de la posición inicial se vuelve a esa misma posición o cerca de ella, por lo que para aumentarla se debe volver a lugares conocidos o con bajo error en la localización. La razón por la que salen un poco diferentes los mapas es porque en el escenario real está la cristalera, que no es detectable por el sensor láser, mientras que en la simulación se usa una pared rígida, como se puede ver en la Figura 2.

En cuanto a las posiciones para llevar el robot, se tienen 4 por cada mesa excepto para las que están cerca de las paredes (que se consideran inaccesibles), otra para la puerta, el pasillo, la mesa del profesor, la mesa de la pecera y la pecera en conjunto tal y como se muestra en la siguiente imagen:

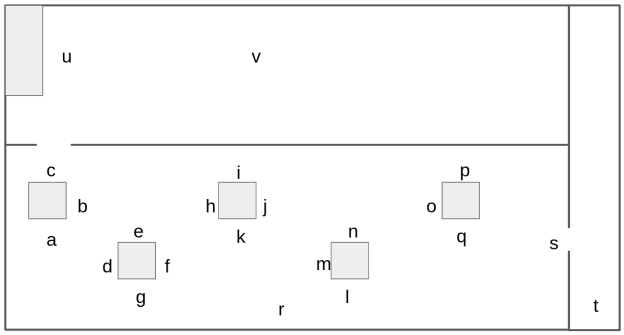

Fig. 5. Mapa del laboratorio real

*2) **Localización y Navegación**:* Una vez se tiene el mapa por el que el robot se va a desplazar, se necesita que este sea capaz de conocer su posición en el mapa construido a la vez que se le mandan posiciones a las que se debe dirigir. Así, se vuelve a definir el problema de la localización del robot, a la vez que aparece el problema de la navegación por el espacio. [11]

Para conseguir localizar al robot se va a usar el paquete de ROS para la localización siguiendo el enfoque de AMCL [12] [13]. Este algoritmo sigue el enfoque de Monte Carlo [14]. Este tipo de algoritmos consiste en una simulación multiprobabilística que sirven para estimar el resultado de cierto evento a partir de casos iniciales aleatorios o pseudo - aleatorios. En localización de robots, este método se usa para establecer la posición del robot en un instante de tiempo, ya que esta no se conoce on exactitud debido al ruido de los sensores, imprecisiones en el mapeado y odometría, etc. Por lo tanto, tener una distribución de posibles posiciones del robot que converjan alrededor de la real es una buena aproximación. Así, los pasos a seguir del algoritmo son similares al funcionamiento de SLAM (ambos usan un filtro de partículas), aunque en este caso cada partícula no mantiene una representación del mapa:

- **Muestreo**: la primera parte que se ejecuta en este tipo es el muestreo de los posibles casos o partículas, que en este caso serían las posibles posiciones del robot en el espacio. Estas partículas se reparten por el mapa y representan tanto la orientación como la posición del robot en el mapa. Durante las iteraciones, las partículas se muestrean a partir de las que se seleccionaron en la iteración anterior, teniendo en cuenta que las que presentaban una mejor aproximación a la posición real (mayor peso o mayor probabilidad) se tienen más en cuenta.

- **Distribución de pesos**: una vez se han repartido las partículas por el espacio, se toman las medidas de los sensores. Luego, para cada una de las partículas se computa el modelo de movimiento y del sensor. Esto es, se proyecta la partícula siguiendo el camino detectado por la odometría (modelo de movimiento) y luego se superpone la medida del sensor con el mapa (modelo de movimiento), comparando lo que está detectando el robot con lo que debería estar detectando. Así, las partículas que presenten mayor coincidencia o probabilidad de ubicarse en la posición real serán asignadas con un mayor peso o importancia de cara al muestreo.

- **Re-muestreo**: los pasos anteriores funcionan correctamente si se tiene un error bajo, es decir, se conoce con antelación la posición del robot en el mundo. En caso de que la incertidumbre del robot aumente tanto que de una ubicación incorrecta se necesita corregir la localización de alguna manera. El re-muestreo permite colocar nuevas partículas en diferentes lugares de los que estima el algoritmo para que en caso de pérdida, el robot sea capaz de acercarse a la posición real. Esta figura del re-muestreo se puede usar cuando se pierda el robot o se a la hora de muestrear se pueden colocar algunas partículas de manera aleatoria, sin tener en cuenta los pesos anteriores.

Con el objetivo de disminuir la complejidad temporal a la hora de muestrear partículas y mantenerlas en el tiempo, en vez de usar un vector (con lo que la complejidad sería cuadrática) se usa un árbol binario cuyos nodos son las posiciones del robot, de manera que si se quiere añadir una partícula que no estaba solo se tiene que cambiar una de las ramas, dejando el resto del árbol igual que en la iteración anterior.

Hasta aquí se tiene el robot localizado en el espacio dentro del mapa construido. Para que sea capaz de llegar a posiciones objetivo se requiere de un planificador de trayectorias. Para ello, se va a usar el paquete *move_base* [15] que implementa ROS. Este paquete utiliza lo que se conoce como *Navigation Stack*, una estructura compuesta por un planificador global [16] y otro local [17]. Ambos obtienen información de un *costmap* global o local, una representación del mapa de zonas por las que es más o menos seguro pasar, tanto a nivel global como local. Estos mapas se componen de información del mapa del entorno creado en el apartado anterior y de la información de los sensores.

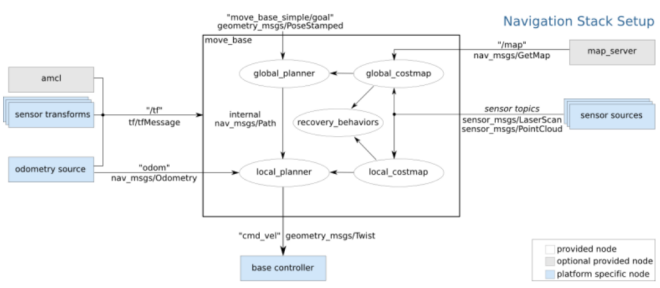

Fig. 6. Esquema del *Navigation Stack*

- **Planificador global**: este elemento se encarga de producir una trayectoria desde el punto de inicio al punto objetivo. Para hacerlo, se usa la información del *global costmap*, una versión del mapa de rejilla donde se ponderan las celdas en función de como de lejos estén de los obstáculos, de manera que a mayor distancia menor será su peso. Así, se considera el mapa ponderado como un grafo ponderado. El problema se reduce a encontrar el camino de menor coste entre los nodos de inicio y destino. El algoritmo que usa ROS para realizar este cálculo es el algoritmo de Dijkstra [18] [19], que calcula el camino más corto entre el nodo inicio con los demás nodos del grafo, creando así el árbol de caminos más cortos entre ellos.

- **Planificador local**: el planificador global calcula la trayectoria a seguir a nivel general, por lo que se trata de un algoritmo pesado. Aun así, se puede dar el caso de que el robot se encuentre en un entorno poco estructurado, por lo que se necesite de un controlador reactivo que evite los obstáculos. Por eso, en ROS se implementa el algoritmo de la Ventana Dinámica [20] [21]. Este algoritmo también usa un *costmap* de rejilla como el planificador global pero a nivel local, por lo que se tiene un espacio discretizado. Luego, se hace un muestreo de las velocidades posibles del robot en base a la actual, cada una de estas se proyecta hacia delante. Así, se genera una simulación sobre cada posible posición del robot en el siguiente instante de tiempo a partir de un comando de control. Finalmente se evalúa cada trayectoria local y se elige la que maximiza la velocidad y la evasión de obstáculos en base al mapa global y local.

La información de ambos planificadores se combina para ir generando las velocidades de la plataforma móvil, de manera que en el planificador local tiene mayor peso la información reactiva (sensores LIDAR, Láser, etc) y en el global es más importante la información del mapa del entorno.

Otra característica de la navegación es la rutina de recuperación, esto es, como se comporta el robot ante diferentes escenarios de error. Generalmente, se está navegando por el espacio pero se puede dar el caso de que se pierda la localización objetivo o que la trayectoria planificada no resulte correcta, por lo que no se puede alcanzar la posición de destino. Estos sucesos suelen ocurrir por desajuste de las constantes y ganancias presentes en los algoritmos. Para corregirlos, ROS implementa una serie de comportamientos para recuperar la posición del robot y replanificar trayectorias. Por orden de ejecución en caso de pérdida se tiene; reinicio conservativo, donde se eliminan obstáculos fuera de un espacio determinado para que dejen de influir en la planificación. Luego, se hace un giro de reconocimiento para disminuir incertidumbre. Si no se ha conseguido solucionar el problema, se eliminan obstáculos fuera del área ocupada por el robot, seguido de un nuevo giro. Finalmente, si después de realizar estas acciones se sigue sin poder llegar al objetivo, se lanza una excepción y el sistema de navegación se detiene.

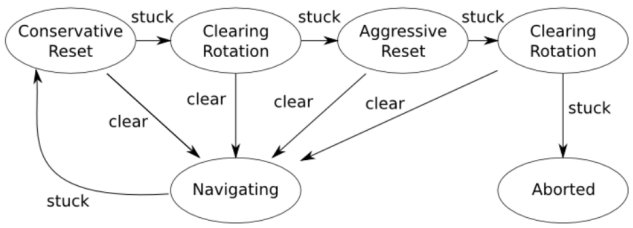

Fig. 7. Esquema de las rutinas de recuperación del paquete *move_base* en ROS

C. *Brazo robótico*

Una parte fundamental a planificar para la realización de un robot mensajero es como gestionar físicamente el traslado del paquete/carta. En este caso, esta función es llevada a cabo por el brazo robótico open source de 4 GDL, *PhantomX Reactor Robot Arm* [22], que incorporan los turtlebots del laboratorio.

Fig. 8. PhantomX Reactor Robot Arm - Interbotix

Para controlar este brazo, se ha diseñado un nodo el cual se encarga de gestionar los diferentes movimientos según el estado en el que se encuentre la aplicación. Para ello, el primer paso realizado fue identificar como publicar el movimiento de cada articulación y el rango de movimiento que tienen con el fin de obtener el comportamiento buscado. Para ello, es necesario lanzar el launch del controlador del brazo que se encuentra en el sistema del turtlebot [23].

```
roslaunch phantomx_reactor_arm_controller..
..arbotix_phantomx_reactor_arm_wrist.launch
```

Una vez lanzado, es posible observar los diferentes topics para el movimiento del brazo. El movimiento que se quiere conseguir en esta tarea se trata de una extensión y retracción del brazo, además de abrir y cerrar la pinza, por lo que no se hace uso de los giros en los ángulos de *roll* y *yaw*. Así pues, comprobando manuelmente el rango de la rotación en *pitch* de hombro, codo y muñeca, se obtuvieron los siguientes datos:

	hombro	**codo**	**muñeca**	**pinza**
rango (rad)	$-\pi/2$ a $\pi/2$	0 a π	$-\pi/2$ a $\pi/2$	0 a $\pi/2$
extendido / pinza abierta (rad)	-1	0.4	-1	0
retraído / pinza cerrada(rad)	-1.5	0	0	1.2

Cabe destacar que durante este proceso se observaron diversos problemas con los brazos de los turtlebots del laboratorio. En primer lugar, no todos los brazos funcionaban correctamente, hay modelos que no respondían ante comandos de movimiento en ciertas articulaciones y otros no respondían ante ninguna, sin embargo, el turtlebot 5 funcionaba correctamente. Pese a ello, los motores del brazo se saturaban con frecuencia pese a moverlos sin carga en el brazo, por lo que se tuvieron que hacer ajustes como eliminar el movimiento de la articulación del codo y reducir el rango de movimiento en las otras articulaciones al extender el brazo para disminuir el par necesario en los motores. Otro problema a tener en cuenta es que la pinza, pese a tener un rango de 0 a $\pi/2$, no cerraba completamente y se saturaba a menudo cuando el valor para cerrar era cercano a $\pi/2$, por lo que como se observa en la tabla, se tiene que para cerrarla, el valor usado es 1.2 rad. Además, debido a que el brazo no cuenta con nigún tipo de sensor de fuerza en la pinza para detectar el tamaño del objeto que se está cogiendo en función a la resistencia a la fuerza de agarre, los objetos manipulados deben ser de un grosor de aproximadamente 0.5 cm, ya que este es el tamaño de la abertura de la pinza con este valor para el cierre, lo que se considera un tamaño apropiado para simular el grosor de un sobre de mensajería lleno.

Así pues, finalmente el funcionamiento del nodo se basa en esperar constantemente a que la máquina de estados envíe el movimiento oportuno a través del topic correspondiente. Este mensaje puede ser:

- **Recoger:** Se abre la pinza, se extende el brazo de manera coordinada y espera un tiempo para que el remitente acerque el mensaje a la pinza. Tras ello, se cierra la pinza, espera un breve tiempo para que el usuario aleje la mano y finalmente se retrae el brazo con el paquete sujetado.
- **Soltar:** Se extiende el brazo y espera un tiempo para que el destinatario acerque la mano al paquete. Una vez pasado el tiempo, se abre la pinza y espera de nuevo un breve tiempo para que la persona retire el paquete con seguridad. Finalmente, se vuelve a retraer el brazo.

D. *Reconocimiento de texto*

Otro aspecto importante en el diseño general del robot mensajero es definir como es realizada la interacción humano-robot en las diferentes etapas del proceso de la aplicación. Así pues, para la etapa en la que el remitente ha de especificar la dirección de destino dentro del mapa del entorno, se consideró el reconocimiento de texto como la mejor opción. Esta técnica es comúnmente conocida como reconocimiento óptico de caracteres *(OCR)* y existen multitud de modelos de inteligencia artifical, librerías y algortimos disponibles para implementarla, por lo que se realizaron diferentes pruebas para elegir de que manera realizar este reconocimiento. Así pues, se optó por implementar un nodo ROS en Python debido a que este lenguaje es quizá el que cuenta con más soporte para realizar técnicas de este tipo. A su vez, se debe tener en cuenta que este proceso se debe realizar de manera offline debido a que el nodo debe funcionar dentro de la red local del laboratorio, por lo que se descartó el uso de APIs de reconocimiento,

aunque probablemente de todas maneras el reconocimiento offline hubiese sido la mejor opción para eliminar retardos y conseguir un reconocimiento más fluido.

Teniendo esto en cuenta, en busca de una buena eficacia con una fluidez aceptable se probaron tres métodos distintos de implementación del OCR, *pytesseract* [24], *easy OCR* [25] y *keras OCR.* [26].

- **Pytesseract:** Pytesseract es una biblioteca de Python que permite reconocer el texto en imágenes utilizando el motor de OCR Tesseract [27], de código abierto desarrollado por Google. Una vez instalado, se puede utilizar el método *image_to_string()* para reconocer el texto en una imagen dada. También se pueden especificar varios parámetros opcionales, como el idioma de reconocimiento de texto, la configuración de procesamiento de imagen y la capacidad de detectar varias líneas de texto.
- **Easy OCR:** EasyOCR es una biblioteca de software de reconocimiento óptico de caracteres (OCR) de código abierto escrita en Python. EasyOCR utiliza una combinación de técnicas de aprendizaje automático y procesamiento de imágenes para lograr una alta precisión en la extracción de texto. Es compatible con varios idiomas y puede ser utilizado en aplicaciones como la digitación automática de documentos, la indexación de imágenes y la automatización de formularios.
- **Keras OCR:** Keras OCR es un proyecto de código abierto que utiliza la biblioteca Keras para reconocimiento de caracteres. Utiliza técnicas de aprendizaje automático para identificar y reconocer caracteres en imágenes, como números y letras. Para funcionar, Keras OCR utiliza una red neuronal conocida como red neuronal recurrente (RNN) para analizar las imágenes de entrada y reconocer los caracteres contenidos en ellas. Esta red neuronal se entrena con un conjunto de datos de imágenes etiquetadas previamente, de manera que pueda aprender a reconocer los caracteres de forma precisa. Una vez entrenada, la red neuronal puede utilizarse para reconocer caracteres en imágenes desconocidas.

Para calificar cada método se tuvo en cuenta el framerate de la aplicación haciendo uso de la clase FPS de la librería de *imutils*. Así pues, el método más rápido para realizar OCR, resultó ser mediante pytesseract, ya que aunque el framrate con éste era bajo, los otros métodos basados deep learning mostraban ser computacionalmente aún más costosos, especialmente tratándose de OCR en un stream de vídeo, para lo cual se intentó buscar optimizaciones sin éxito. Pese a la baja fluidez, la eficacia reconocimiento es satisfactoria y en la aplicación final este vídeo no está pensado para ser mostrado al usuario, por lo que el remitente solo debe acercar el paquete con la dirección en formato ”DESTINO (+letra)” a la cámara, y cuando se escuche el sonido de confirmación de lectura, ya sabrá que ha sido detectado. Este formato de dirección es debido a que, en nuestro caso, durante las pruebas se comprobó que la detección de letras era algo más consistente que la de números.

Finalmente, una vez realizadas todas las pruebas para seleccionar el método de reconocimiento, se implementó el nodo de ROS. Éste está suscrito al topic de la cámara que incorpora el turtlebot, cuyo launch (astra.launch) se debe ejecutar previamente. Él nodo de la cámara publica constantemente cada frame en el topic *'/camera/rgb/image_raw'*, por lo que en el callback de este topic, se debe procesar cada frame de la siguiente manera:

1) Transformar el mensaje de sensor_msgs/Image a imagen de opencv mediante CvBridge [28] .
2) Reconocer el texto del frame mediante pytesseract, guardado en formato en string.
3) Si este texto tiene el formato *”DESTINO (+ letra)”*, se reconoce como destino valido.
4) Si el destino reconocido es el mismo que el de una serie de frames anteriores, se envía a la máquina de estados la letra reconocida como destino del paquete y se emite un sonido de confirmación.

Este último paso es para eliminar los reconocimientos erróneos debido a posible ruido en la imagen en cierto instante, que puede ser causado por ejemplo por iluminación inconsistente.

V. RESULTADOS

En este apartado se van a presentar los resultados del proyecto desarrollado, diferenciando entre simulación y robot real.

A. *Simulación*

En el caso del robot en simulación, se tuvo que obviar la parte del brazo robótica, ya que el modelo URDF del Turtlebot3 (desarrollo en Noetic) no incluye un brazo robótico que accionar. Por otro lado, la detección del texto se hace a través de la cámara del ordenador.

En el vídeo se puede ver como el robot se dirige a la ubicación H desde la psición de *Home* (el inicio del mapa). Luego se dirige a la posición K para luego volver a la anterior ubicación, de manera que se simula como el robot entrega una carta (en H) y devuelve una contestación(en K). (ver vídeo)

B. *Robot Real*

A nivel de simulación se puede ver como se ha conseguido crear una máquina de estados capaz de transcicionar y obtener resultados de manera solvente. El robot indica claramente los estados por los que va pasando. Además, la detección de texto se realiza correctamente, por lo que es posible enviar multitud de comandos. Así, el robot presenta un comportamiento propio y similar al de un mensajero a través de la superposición de sistemas.

A la hora de probar el algoritmo en los robots reales se tuvieron en cuenta dos aproximaciones; una en la que la cámara que se usa la *webcam* del ordenador para la detección del destino o usar la cámara del Turtlebot para ese fin. (ver vídeo cámara ordenador) (ver vídeo cámara Turtlebot)

VI. INSTRUCCIONES

En este apartado se expondrán los pasos necesarios para obtener todas las librerías y paquetes necesarios para poder compilar sin errores el presente proyecto. Asimismo, se explicarán las maneras de ejecutar el proyecto tanto en los robots reales del laboratorio como para hacerlo en simulación. Estas instrucciones se han realizado en ROS Noetic.

A. *Instalación*

Las librerías y paquetes necesarios para la ejecución del proyecto y sus comandos de instalación son los siguientes:

- **Paquete del algoritmo de localización**: localización basada en filtros de Monte Carlo Adaptativos (AMCL):
  ```
  sudo apt install ros-noetic-amcl
  ```
- **Paquete de navegación**: es el que lanza el algoritmo de planificación de trayectorias.
  ```
  sudo apt install ros-noetic-move-base
  ```
- **Paquete *Map Server***: es el encargado de almacenar el mapa del entorno.
  ```
  sudo apt install ros-noetic-map-server
  ```
- **Paquete con los controladores de Kobuki**: para disponer de los mensajes tipo *kobuki_msgs* para enviar y recibir comandos en el Turtlebot2 real.
  ```
  sudo apt install ros-noetic-kobuki- ...
  ... controller
  ```
- **Paquete para la planificación local**: implementación de la ventana dinámica en ROS:
  ```
  sudo apt install ros-noetic- ...
  ... dwa-local-planner
  ```
- **Reconocimiento de texto**: librerías para el reconocimiento de texto mediante *pytesseract*.
  ```
  pip3 install pytesseract
  pip3 install tesseract
  pip3 install tesseract-ocr
  sudo apt-get install libleptonica-dev
  sudo apt-get install tesseract-ocr
  sudo apt-get install tesseract-ocr-dev
  sudo apt-get install libtesseract-dev
  sudo apt-get install python3-pil
  sudo apt-get install tesseract-ocr-eng
  sudo apt-get install tesseract-ocr- ...
  ... script-latn
  ```
- **Registro del teclado**: librería para registrar las pulsaciones del teclado.
  ```
  pip3 install pynput
  ```

Aun así, es posible que algunos comandos varíen en función de la distribución de ROS que se tenga. Además, si algún paquete no está instalado, durante la compilación se mostrará un error con el nombre del mismo.

B. *Instrucciones de ejecución*

Una vez se han instalado las librerías y paquetes necesarios para la ejecución del proyecto se puede iniciar la compilación. Primero se debe crear un directorio en el equipo:

```
- mkdir catkin_ws && cd catkin_ws
- mkdir src && cd src
- git clone https://github.com/
joseamg24/ProyectoRobotsMoviles
- cd ..
- catkin_make
- source devel/setup.bash
```

Si no se obtienen errores, el proyecto estará compilado. Cabe destacar que se tienen varias ramas; una dedicada al robot real y otra dedicada a la simulación. Entre ellas solo difiere la ejecución de los ficheros del Turtlebot.

*1) **Simulación**:* En este caso, se tienen que lanzar varios paquetes para ejecutar la simulación completa de la aplicación. Primero se debe lanzar la simulación de Gazebo con el Turtlbeot3 en el mundo.

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_gazebo ...
... turtlebot3_world.launch
```

Es necesario especificar el modelo que se va a la lanzar, que puede ser *burger*, *waffle*, etc. Esto se debe a que hay varios modelos de Turtlebot3. Luego, se debe importar el escenario de la clase para la simulación. En la carpeta *building_editor_models* está el modelo de la clase. Así, se elimina el que se lanza por defecto y se añade la clase. Para hacerlo, hacer *click* en el menú *Insert/AddPath*. Luego seleccionar la carpeta mencionada y arrastrar el atributo Clase al mundo. Con esto solo se importan las paredes, pero se necesitan los muebles. Para importarlos se sigue el mismo procedimiento pero añadiendo la carpeta *model_editor_models* y arrastrar la carpeta Clase3 cuadrándola con el escenario.

Una vez se tiene la simulación funcionando se lanza el algoritmo de localización y navegación en otro terminal.

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_navigation ...
... turtlebot3_navigation.launch ...
... map_file:=RUTA_DEL_MAPA
```

En este caso también se debe definir el modelo de robot, ya que hay muchos modelos distintos de robots y el cálculo de la odometría y el control es diferente para cada uno de ellos. En la carpeta *mensajero_sim* se encuentra el mapa construido para la simulación, por lo que la ruta de ese directorio sería la que habría que especificar.

Finalmente, en otro terminal se lanza un fichero *.launch* la máquina de estados junto con los nodos del reconocimiento de texto y las teclas.

```
roslaunch robot_cartero ...
... robot_cartero.launch
```

2) ***Robot Real****:* Para ejecutar la aplicación del robot mensajero primero se lanzan los siguientes ficheros *.launch* desde la consola del Turtlebot2 en distintos terminales. Para ello, previamente se debe estar conectado al ordenador interno del robot y a la red local, de manera que el Turtlebot2 tenga ejecutando el nodo maestro de ROS. El robot debe estar conectado a una LAN. Cabe destacar que el robot debe ser el Turtlebot2 modelo 5, ya que es el único donde funciona la pinza. Los comandos se lanzan dentro del ordenador del Turtlebot, por lo que se debe entrar ejecutando *ssh tb2@192.168.1.9*:

```
roslaunch turtlebot_bringup ...
... minimal.launch

    # En otro terminal
export TURTLEBOT_3D_SENSOR=astra
roslaunch turtlebot_navigation ...
... amcl_demo.launch  ...
... map_file:=/home/tb2/mapa.yaml

    # En otro terminal
roslaunch phantomx_reactor_arm_controller..
..arbotix_phantomx_reactor_arm_wrist.launch

    # En otro terminal
roslaunch astra_launch astra.launch
```

Hasta aquí se tiene el robot con los *topics* de velocidad, botones y leds disponibles gracias al primer comando. En el segundo se ejecuta el algoritmo de navegación y localización del Turtlebot, mientras que el tercero lanza los controladores del brazo robótico. Finalmente se lanza el nodo de la cámara del Turtlebot.

Estos son todos los comandos que se deben lanzar en el robot. Para lanzar el proyecto del robot mensajero con sus algoritmos se ejecuta el siguiente comando desde un terminal definiendo las variables de entorno *ROS_IP=192.168.1.9* y *ROS_MASTER_URI=http://ip_del_equipo:11311*, con el equipo debe estar conectado a una red local junto con el robot:

```
roslaunch robot_cartero ...
... robot_cartero.launch
```

En el fichero *robot_cartero.launch* se lanza tanto la máquina de estados como el nodo de la cámara y el control del brazo. Una vez realizados todos los pasos se tendría al robot capaz de seguir el comportamiento del espacio de estados, controlar el brazo y recibir información de la cámara en el estado correspondiente.

VII. Conclusión

Para terminar, el proyecto realizado ha cumplido los objetivos propuestos al inicio de la práctica; la programación de un robot capaz de moverse por el espacio y entregar objetos con un brazo robótico. Además, se experimentó con el Turtlebot en simulación y con el Turtlebot del laboratorio a la vez, por lo que ese objetivo también se considera cumplido.

El tener un proyecto a largo plazo como ha sido el caso y el usar una plataforma como Github ha resultado de gran ayuda para tener un historial de los cambios que se van haciendo así como tener un seguimiento más preciso del proyecto. También se pudieron establecer dos ramas; una para la simulación y otra para el robot real, ya que la ejecución del algoritmo de Navegación no es la misma en ambos caso.

Así, se evalúa la práctica positivamente pues se acerca a una labor más ingenieril, donde se pasaron por diferentes procesos como la elección del proyecto, desarrollo, pruebas y experimentación, propias de cualquier trabajo de ingeniero.

VIII. Referencias

[1] GitHub del proyecto: https://github.com/joseamg24/ProyectoRobotsMoviles

[2] Máquina de estados SMACH: http://wiki.ros.org/smach

[3] ROS: http://wiki.ros.org/

[4] Pynput: https://pypi.org/project/pynput/

[5] ROS SLAM: http://docs.ros.org/en/hydro/api/gmapping/html/index.html

[6] B. L. E. A. Balasuriya et al., "Outdoor robot navigation using Gmapping based SLAM algorithm," 2016 Moratuwa Engineering Research Conference (MERCon), 2016, pp. 403-408, doi: 10.1109/MERCon.2016.7480175.: https://ieeexplore.ieee.org/abstract/document/7480175?casa_token=YdtOAfpupcwAAAAA:8da_SPA8rtDy229MX5oqt7Hhg6gosVdmxN2rkFVq1j_SEwbvda7rzxNhDbPsmIZhdpZF7mI1kQ

[7] SLAM vídeo: https://www.youtube.com/watch?v=ueSmN_VxNPg

[8] Transparencias de SlAM: http://oramosp.epizy.com/teaching/211/rob-autonoma/clases/11b_FastSLAM.pdf?i=1

[9] Teleoperación del Turtlebot: http://wiki.ros.org/teleop_twist_keyboard

[10] Tutorial de mapeo: https://emanual.robotis.com/docs/en/platform/turtlebot3/slam_simula

[11] Tutorial navegación: https://emanual.robotis.com/docs/en/platform/turtlebot3/nav_simula

[12] AMCL ROS: https://navigation.ros.org/configuration/packages/configuring-amcl.html

[13] AMCL: https://roboticsknowledgebase.com/wiki/state-estimation/adaptive-monte-carlo-localization/

[14] Algoritmo de Monte Carlo: https://www.ibm.com/topics/monte-carlo-simulation

[15] Algoritmo de navegación ROS: http://wiki.ros.org/move_base

[16] *Global Planner* ROS: http://wiki.ros.org/global_planner

[17] *Local Planner* ROS: http://wiki.ros.org/base_local_planner

[18] Algoritmo de Dijsktra: https://www.codingame.com/playgrounds/7656/los-caminos-mas-cortos-con-el-algoritmo-de-dijkstra/el-algoritmo-de-dijkstra

[19] A. Alyasin, E. I. Abbas and S. D. Hasan, "An Efficient Optimal Path Finding for Mobile Robot Based on Dijkstra Method," 2019 4th Scientific International Conference Najaf (SICN), 2019, pp. 11-14, doi: 10.1109/SICN47020.2019.9019345. https://ieeexplore.ieee.org/abstract/document/9019345?casa_token=GLnw52rp7WIAAAAA:hfL6YIcFaeUAeGYYZWIazDmbGZUAxQHZB0aAifSqCn-aa8sgRZM20pPgSdgA3_1tQ8SeZBioYA

[20] Fox, Dieter, Burgard, Wolfram, Thrun, Sebastian. (1997). The Dynamic Window Approach to Collision Avoidance. Robotics & Automation Magazine, IEEE. 4. 23 - 33. 10.1109/100.580977. https://www.researchgate.net/publication/3344494_The_Dynamic_Window_Approach_to_Collision_Avoidance

[21] *Dyanamic Window* ROS: http://wiki.ros.org/dwa_local_planner

[22] Interbotix PhantomX Reactor Robot Arm: https://www.interbotix.com/Robotic-Arms

[23] PhantomX Reactor Arm Controller: https://github.com/RobotnikAutomation/phantomx$_reactor_arm$

[24] Python Pytesseract OCR: https://pypi.org/project/pytesseract/

[25] Python EasyOCR: https://github.com/JaidedAI/EasyOCR

[26] Python Keras-OCR: https://keras-ocr.readthedocs.io/en/latest/

[27] Google Tesseract OCR: https://es.wikipedia.org/wiki/Tesseract$_OCR$

[28] Ros CvBridge: http://wiki.ros.org/cv$_bridge$