industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

PROGRAMACIÓN DE VEHÍCULOS AUTÓNOMOS CON INTEGRACIÓN DE SOFTWARE ROS EN PLATAFORMAS LABVIEW-CRIO

TRABAJO FIN DE GRADO

GRADO EN INGIENERIA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: **Fco Javier Rodríguez Armero**
Director: Francisco José Ortiz Zaragoza
Codirector: Leanne Rebecca Miller

Cartagena, 9 de Abril de 2019

Universidad Politécnica de Cartagena

RESUMEN

En los últimos años, las investigaciones relacionadas con la conducción autónoma han ido incrementando. El uso de esta tecnología ya tiene aplicaciones diversas en áreas, como: industria, militar, agrícola y otras. Pero la investigación apunta cada vez más a aplicaciones cotidianas, que requieren procesos más complejos y sofisticados. La conducción autónoma de vehículos de transporte pesado por grandes carreteras y la conducción de transporte público o privado por las ciudades, conlleva un factor humano: el comportamiento de otros conductores, a veces impredecible, o los peatones que transitan por las calles, son factores que hay que tener en cuenta al diseñar e implementar un sistema de conducción sin piloto. En la UPCT ya existe un equipo de investigación, que trabaja con el Cloud Incubator Car. Y se pretendía diseñar toda la inteligencia necesaria del robot únicamente utilizando LAbVIEW.

Este proyecto fin de grado presenta la posibilidad de combinar dos plataformas, LabVIEW como ya se estaba haciendo para controlar el robot a nivel bajo, es decir, control de velocidades y sensorización básica; y ROS o Robot Operating System, que está diseñado bajo Linux, y preparado para un gran procesamiento. Además, en esta plataforma ya existen diversas librerías que implementan algoritmos de cálculo de trayectorias, evitación de obstáculos, planificación local y global, sensorización con láser, etc. Y también cuenta con una gran comunidad de investigadores que comparten sus conocimientos en la red, que es de gran ayuda.

Es por eso que se ha realizado un proyecto, mezclando estas dos plataformas, LabVIEW con una sbRIO que controla el robot DANI, el Starter Kit 1.0 de National Instruments; y un pc de trabajo que utiliza como sistema operativo Ubuntu y tiene instaladas todas las funciones de ROS. Controlando el DANI con lenguaje G (LV), desde un PC-ROS que aporta la inteligencia necesaria para la navegación.

ABSTRACT

In recent years, research related to autonomous driving has been increasing. The use of this technology already has diverse applications in areas such as: industry, military, agricultural and others. But research is increasingly aimed at everyday applications, which require more complex and sophisticated processes. The autonomous driving of heavy transport vehicles on large roads and the driving of public or private transport through the cities, carries a human factor: the behavior of other drivers, sometimes unpredictable, or pedestrians passing through the streets, are factors that It must be taken into account when designing and implementing a driving system without a pilot. In the UPCT there is already a research team, which works with the Cloud Incubator Car. And it was intended to design all the necessary intelligence of the robot only using LAbVIEW.

This final project presents the possibility of combining two platforms, LabVIEW, as was already being done to control the robot at a low level, that is, speed control and basic sensorization; and ROS or Robot Operating System, which is designed under Linux, and prepared for great processing. Furthermore, in this platform there are already several libraries that implement trajectory calculation algorithms, obstacle avoidance, local and global planning, laser sensorization, etc. And it also has a large community of researchers who share their knowledge on the network, which is very helpful.

That's why a project has been done, mixing these two platforms, LabVIEW with a sbRIO that controls the DANI robot, the National Instrument Starter Kit 1.0; and a working pc that uses as Ubuntu operating system and has all the functions of ROS installed. Controlling the DANI with language G (LV), from a PC-ROS that provides the necessary intelligence for navigation.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

AGRADECIMIENTOS

Me gustaría comenzar nombrando a todas las personas que han hecho este trabajo posible. En primer lugar, a mi profesor, tutor de prácticas y director del proyecto, Francisco José Ortiz Zaragoza, por marcarme el camino y guiarme para que siempre fuese con buen rumbo y a mi codirectora Leanne Rebecca Miller por darme consejos esenciales para la buena marcha del proyecto.

En segundo lugar, a mi Madre, a mi Padre y a mi hermana, que me han aguantado mis más y mis menos, que me han apoyado siempre y en todo aquello que necesitaba y que han sabido darme fuerzas y ánimos en los momentos de bajón.

A los compañeros de carrera y a los compañeros de viajes de bus, tren y coche (nos ha faltado el barco y el avión), por hacer esto ameno y casi divertido, por poner las cosas fáciles, por ser buena gente y por todas esas anécdotas que hacen de este viaje una experiencia inolvidable.

A mi segunda familia, Iradier, que han creado en mí una persona responsable y comprometida, y en la que sé que me puedo apoyar sin temor a caerme, con la que tanto tiempo paso y con la que he conseguido crear una rutina llena de aventuras.

A todos ellos, GRACIAS.

CAPÍTULO 1: INTRODUCCIÓN

1.1 PROPUESTA

Con este proyecto se pretende poner en marcha un robot autónomo basado en DANI de National Instruments, el llamado “Starter Kit 1.0”. El robot llevará implementado en su controladora un sistema programado en LabVIEW, el cual solo aportará la programación para controlar el movimiento, a partir de una velocidad “lineal x” y una velocidad “angular z”, y de obtener los datos de la odometría dl vehículo a partir de los encoder que lleva el Starter Kit 1.0.

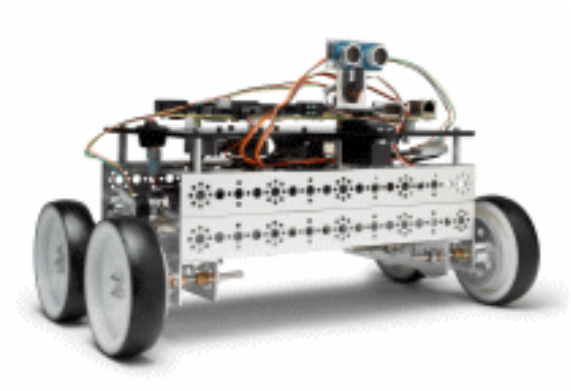

Figura 1. Robot DANI.

La programación o inteligencia que necesitará para desplazarse de un punto a otro y no colisionar con ningún obstáculo en su camino la proveerá ROS, un software dedicado a la navegación de robots, el cual se comunicará con el DANI a través de un modem instalado en el propio DANI. También se cuenta con la necesidad de integrar un software/librería adicional para conseguir una comunicación entre ROS y LabVIEW.

Se han elegido estas plataformas, tanto de hardware como de software, debido a que el vehículo autónomo real, al que se dedica esta investigación, está controlado por una placa controladora cRIO de National Instruments la cual es programada con LabView. Y ROS es escogido por su gran variedad de librerías y comandos, que nos proporcionan una gran versatilidad y capacidad para desarrollar aplicaciones de navegación inteligente orientada a robots y vehículos.

1.2 OBJETIVOS

Los objetivos de este trabajo fin de grado consistirá en desarrollar una aplicación en LabVIEW adecuada para el control a bajo nivel de un robot, movimiento lineal en x y angular en z, además de adquirir datos de los encoder y transmitir dicha información a la aplicación de alto nivel en forma de odometría. Montar el hardware e implementar el software necesario para interconectar las plataformas LabVIEW (bajo nivel) y ROS (alto nivel), para que puedan comunicarse y transmitirse los comandos de velocidad

necesarios y la odometría del robot, y por último, desarrollar la lista de comandos y funciones necesarias para poner en funcionamiento el robot Starter Kit 1.0, y que bajo el control de LabView y dirigido con el sistema ROS conseguirá que el robot navegue autónomamente de un punto a otro. Este proyecto demostrará el potencial que tiene ROS en el campo de la navegación frente a LabVIEW, ya que, aunque LabVIEW podría encargarse de la parte de navegación, no está optimizado y tampoco se pueden desarrollar aplicaciones sofisticadas y complejas, mientras que en ROS sí.

Figura 2. Logo LabVIEW y ROS.

Después de poner el robot en funcionamiento, se hará el estudio de los cambios y modificaciones que son necesarios para traspasar dicho sistema al vehículo real, incluyendo el estudio y simulación de librerías de vehículos no holonómicos.

Este documento, recogerá todas las tareas realizadas durante el avance del proyecto, incluyendo aquellas acciones que resultaron en error, explicando el porqué, y como solucionar todos los problemas que fueron surgiendo. Además, se incluye un capítulo entero dedicado a la comunicación entre ROS y LabVIEW, de opciones que había, pruebas y conclusión final. Las temáticas de cada capítulo son:

Capítulo 1: recoge el contexto del proyecto, los objetivos a cumplir y las fases.

Capítulo 2: contextualización del proyecto, breve explicación de los programas, librerías y dispositivos utilizados.

Capítulo 3: se explica el proceso de investigación de cómo comunicar las dos plataformas usadas, ROS y LabVIEW, así como el proceso escogido, y como implementarlo.

Capítulo 4: se explica detalladamente como si de un manual se tratase el cómo reproducir el proyecto.

Capítulo 5: varios problemas que han ido surgiendo durante el desarrollo son explicados aquí, como han sido o deberían ser resueltos.

Capítulo 6: estudio de traspaso de conocimientos recopilados en este proyecto al vehículo real INCUBATOR CAR.

Capítulo 7: las conclusiones y demás pensamientos relacionados con el proyecto y su futuro.

1.3 FASES DEL PROYECTO

Las fases del proyecto han estado muy marcadas por los objetivos a cumplir:

- Comenzó con la reproducción del TFM [1], que consistía en tele-operar el Turtlebot con un mando de ps4, conseguir mapear un espacio y hacer que el robot navegase de manera autónoma dentro de este. Con ello conseguimos entender la parte de mapeo y navegación que usaremos de manera idéntica en nuestro proyecto.
- Después se trató de conseguir comunicar una aplicación en LabView con un sistema Ubuntu-ROS, ya que dichas plataformas no eran compatibles en principio, en cuanto a comunicaciones se refiere.
- Luego se modificó el proyecto-ejemplo "Starter Kit 1.0" para que el DANI se moviese con las velocidades que recibía desde ROS y enviase su odometría hacia ROS; se hizo el programa de bajo nivel que controlaba lo más básico del robot.
- Posteriormente, se creó un script que contenía todos los comandos necesarios para hacer el control de alto nivel en ROS, se trató de hacer el mapeado del laboratorio y se consiguió que el robot navegase autónomamente.
- Por último, se investigó sobre como traspasar el proyecto del robot a un vehículo real, un vehículo no-holonómico; investigación sobre que librerías usar, modificaciones con respecto a este proyecto y otros cambios aplicables.

CAPÍTULO 2: CONTEXTUALIZACIÓN DEL PROYECTO

2.1 TIPOLOGÍA DEL ROBOT

La tipología de robot utilizado en este proyecto es la más común en el campo de la robótica, un robot terrestre, con dirección diferencial. Tanto el Turtlebot 2 como el DANI tienen esta configuración. Esto significa entre muchas, dos cosas: la más básica, que el robot se mueve en dos dimensiones y solo nos interesan la posición en x e y, además de la orientación, un ángulo θ; y la otra sería la capacidad de controlar las ruedas de cada lado de manera independiente, lo que le da la capacidad al vehículo de rotar sobre sí mismo, es decir, modificar la orientación sin cambiar de posición.

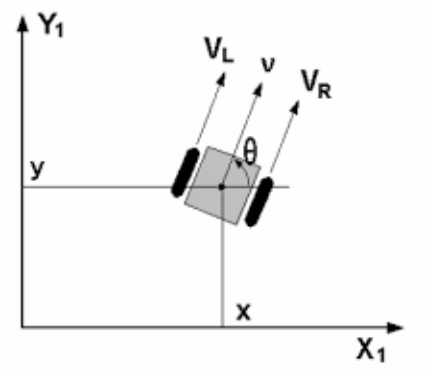

Figura 3. Topología diferencial.

Más adelante, al entrar a investigar sobre el coche real, necesitaremos incorporar modificaciones al proyecto para contemplar la nueva tipología del vehículo, esta vez será también un vehículo de tipo terrestre, pero no tendrá la capacidad de modificar solo la orientación, debido a que tiene las ruedas traseras configuradas para propulsar y las delanteras para dirigir, está configuración es de tipo no-holonómico.

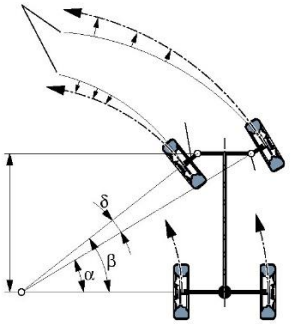

Figura 4. Topología no-holonómica.

Los robots que se muestran a continuación son los que se han utilizado; el primero es el Turtlebot, que se utilizó para reproducir el TFM [1], y el segundo es el DANI, que fue sobre el que se desarrolló este proyecto. El Turtlebot viene ya equipado con una base que tiene un controlador central, que tiene integrado el conocer la odometría, controlar velocidades, leer sensores de choque, etc, y que se comunica mediante un puerto serie

con el NUC. Mientras que el DANI tiene todo lo necesario para realizar lo mismo, pero sin un control central, es el programa que se carga en la sbRIO el que se encarga de leer sensores, calcular odometría, escribir velocidades y demás.

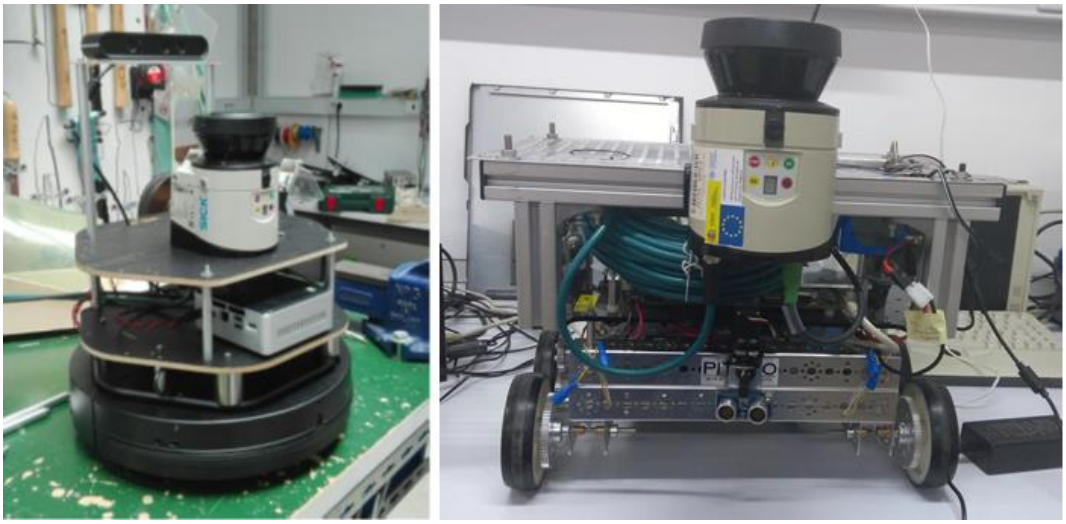

Figura 5. Turtlebot (Izquierda), Robot DANI con LIDAR (Derecha).

El Turtlebot está equipado con diversos sensores para reconocer su entorno y conocer y controlar su posición, como: un giroscopio de un eje, odometría con 52 pulsos/rev del encoder que se traduce a 11,7 pulsos/mm de avance del robot, detección de bordes a los lados y al frente, sensor de contacto con el suelo en las dos ruedas y el Orbec Astra que venía con el Turtlebot de fábrica, que es una cámara 3D con 60° de visión horizontal por 50° en vertical y reconocimiento hasta 8 metros de distancia. Además, fue equipado con un Laser, el SICK LMS111, un potente laser 2D de 270° de visión con reconocimiento de hasta 20 metros, con una resolución de 0.5° y una frecuencia de exploración de 50Hz.

El robot DANI también posee sensores para su reconocimiento propio y el del exterior, este cuenta con: un sensor de ultrasonidos, con un rango de 3 metros y montado sobre un servo que le da una visión de 180°, aunque con una frecuencia de exploración un poco baja y encoders en cada motor de 400 pulsos/rev. Además, al tener un sbRIO como controlador, existe la posibilidad de ampliar las capacidades del robot a través de los pines I/O de la tarjeta, pudiendo incorporar sensores como una IMU o GPS. Adicionalmente, al igual que al Turtlebot, se le incorporó un Laser SICK LMS111, cuyas características fueron descritas anteriormente.

2.2 ARQUITECTURA HARDWARE

La arquitectura del hardware del DANI para este proyecto se basa en la comunicación, que la controla o aporta el módem. A este módem están conectados todos los sistemas informáticos, la tarjeta sbRIO, el NUC y el PC de trabajo, adicionalmente también está conectado el láser LIDAR, para enviar los datos recibidos por la red.

El DANI lleva incorporado el módem, y así se conecta por cable ethernet: el LIDAR; el NUC, que también va sobre el DANI; y la propia tarjeta controladora sbRIO.

De manera inalámbrica se conecta: el PC de trabajo a la red vía WiFi, para poder comunicarse con la sbRIO (para cargar y ver el HMI que controla el DANI), y con el NUC (para controlarlo vía SSH de manera remota); y el NUC con el mando de PlayStation vía Bluetooth para manejar el robot.

Con esta arquitectura podemos mover el robot con total libertad, mientras nosotros manipulamos el mando de PS observando la pantalla del PC de trabajo, sin necesidad de movernos y seguir al robot.

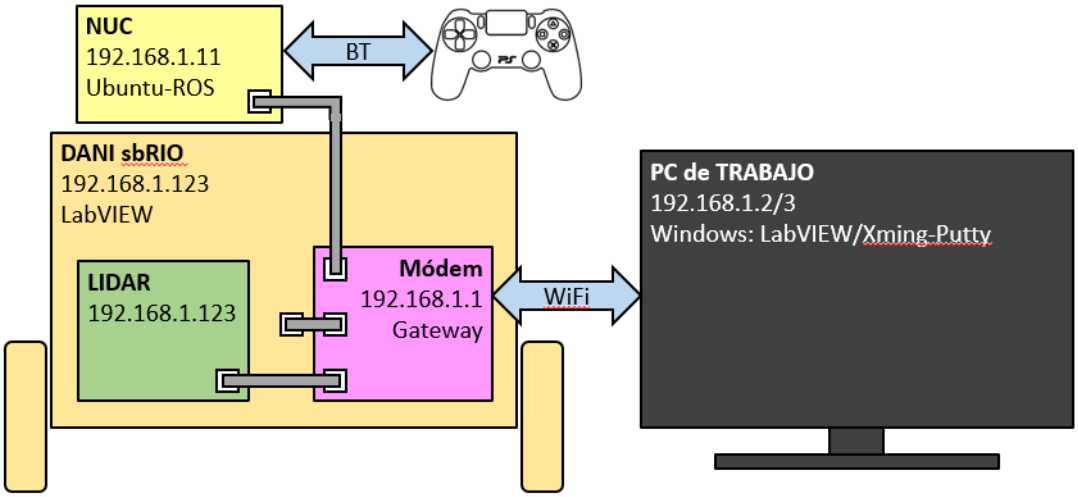

Figura 6. Diagrama de la arquitectura hardware.

2.2 INTRODUCCIÓN AL SOFTWARE UTILIZADO

2.2.1 ROS (ROBOT OPERATING SYSTEM)

Robot Operating System es uno de los softwares orientados a la robótica más avanzados de la actualidad. Es un entorno de programación de código abierto basado en UNIX, que permite controlar robot reales y simulados. Entre las tareas que más se utilizan y son más populares se encuentran: la navegación, reconocimiento 3D, SLAM, localización, etc.

Las funcionalidades más descriptivas de ROS son:

- Una de sus características más importantes es la de ser de **código abierto**, ya que distintos programadores, universidades, desarrolladores y demás personas pueden compartir sus implementaciones en la red y que les sirvan a otros, haciendo que un proyecto no tenga que partir de cero y acortando el periodo de aprendizaje e investigación de cualquiera.
- Tiene implementado la comunicación **P2P** o peer to peer, que significa que todos los programas que se ejecutan en ROS intercambian mensajes sin necesidad de un servicio, de manera que con una filosofía de publicación-suscripción se

comunican los distintos programas que lo necesiten. Este también hace que un sistema se pueda escalar de forma más sencilla.

- **Basado en herramientas**, simples, complejas, genéricas o específicas, que entre todas forman un sistema mayor, cada una dedicada a una tarea, así un proceso en particular puede ser mejorado sin necesidad de modificar ningún otro.
- Es un entorno que permite **multitud de lenguajes** como C++, Python, Java o MathLab entre otros, así cada cual elije de qué manera escribir los programas.
- ROS anima a los desarrolladores a escribir programas en forma de funciones, es decir, **programas sencillos y ligeros** que se llamen o ejecuten de manera independiente y que se comuniquen entre ellos si fuese necesario, para así poder reutilizar programas simples en otras aplicaciones, o escalar/mejorar cada una por separado más fácilmente.

Todo esto se consigue gracias un ROS master, que es el que proporciona la coordinación de todas las comunicaciones, publicaciones y suscripciones, registro de nodos, nombres y topics, etc. Para que no haya sobre-escrituras ni colisiones en la transmisión de datos.

La característica más destacable de ROS, al menos para este proyecto es que tiene implementado la navegación y todo lo relacionado con esta. Por tanto, se resumirá en que se basa y la arquitectura que sigue la navegación.

NAVEGACIÓN

En esta sección se pretende dar a conocer las características básicas del paquete de navegación de ROS. Primero definimos que dentro del esquema de control de un robot móvil hay cuatro componentes o subsistemas:

- Percepción: consiste en los componentes del robot que son responsables de conocer su entorno, su "mundo", que generalmente son sensores.
- Localización: es el componente en el que se calcula la posición absoluta del robot en el mundo, y también, la posición relativa de los objetos en movimiento, como una persona u otro robot. El mapeo es parte de esta componente.
- Planificación: es el componente responsable de calcular la ruta que hará que el robot alcance su objetivo.
- Control de movimiento: es el componente responsable de traducir la ruta dada por el componente de planificación en entradas de motor, de manera que el robot logre lo que se desea.

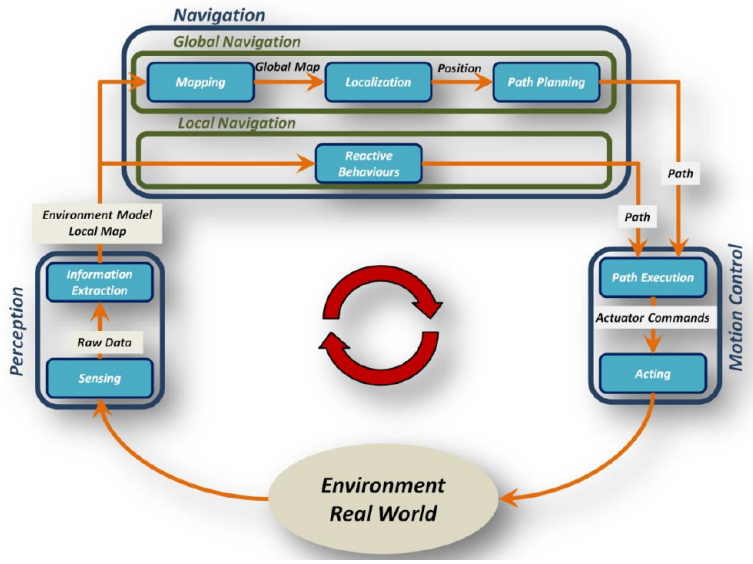

Figura 7. Sistema y subsistemas de navegación en ROS.

Cada subsistema está dedicado a una tarea y todos son utilizados para lograr la navegación, que se define como la capacidad del robot para actuar en función del conocimiento acerca de su entorno y su posición objetivo, y con los valores de los sensores, alcanzar sus posiciones objetivo de la manera más eficiente y confiable posible.

Un robot móvil autónomo debe poder calcular el mejor camino que le permita alcanzar su objetivo, pero también debe tener una actitud reactiva en la que debe poder reaccionar en tiempo real para evitar colisiones. Con esto, significa que el componente de navegación de un AMR (Robot móvil autónomo) incluye dos competencias:

- Dado un mapa y una ubicación meta, la planificación implica identificar una trayectoria que hará que el robot alcance la ubicación de la meta cuando se ejecute.
- Dadas las lecturas de los sensores en tiempo real, la evitación de obstáculos significa modular la trayectoria del robot para evitar colisiones, pero aun así alcanzar la meta.

Es por ello, que la navegación se divide en dos bloques:

1. Navegación local: se basa en que un robot actúa según comportamientos reactivos, cuya principal prioridad es evitar colisiones con los obstáculos en el entorno que rodea al robot, pero también para alcanzar un objetivo. Para poder hacerlo, el robot necesita lecturas de sensores exteroceptivos en tiempo real. En otras palabras, el robot decide cómo comportarse en función de un punto de vista local en tiempo real de lo que lo rodea. Hay muchos algoritmos que implementan comportamientos reactivos. Los dos más conocidos son el Histograma de campo vectorial (Vector Field Histogram) y el Enfoque de ventana

dinámica (Dynamic Window Approach, DWA). Esta última es la utilización del paquete de navegación de ROS.

2. Navegación global: es el tipo de navegación que un AMR necesita hacer cuando quiere planificar cómo ir de una posición a otra o alcanzar una meta. Para hacerlo, un AMR necesita ubicarse en el entorno utilizando un mapa del mismo y luego planificar la trayectoria que se necesita seguir para llegar a la ubicación de la meta. Con esto, queda claro que este tipo de navegación se divide en tres componentes:
    - La localización, es el componente responsable de la localización del robot en el entorno. En otras palabras, calcula la posición del robot, lo que se puede hacer haciendo coincidir las lecturas de los sensores con un mapa.
    - El mapeo, es el componente responsable de construir el mapa del entorno alrededor del robot. Para ello, hace uso de las lecturas de los sensores exteroceptivos.
    - La planificación de la ruta, es el componente responsable de encontrar la ruta para ir desde la posición actual del robot hasta la meta. Lo hace mediante el uso de algoritmos de búsqueda en un mapa del entorno.

Ambos tipos de navegación están orientados a objetivos, la principal diferencia entre ellos es que la navegación local se basa solo en el entorno real del robot, mientras que la global se basa en todo el entorno del robot.

Una vez que tenemos la trayectoria que el robot tiene que hacer, pasamos al siguiente subsistema, el control de movimiento. Es la parte más próxima al hardware de un sistema de navegación, que traduce comandos de velocidad tipo */cmd_vel o /mobile_base/commands/velocity a* comandos de actuación, por ejemplo, velocidad de giro del motor de cada rueda, para que el robot se mueva a la velocidad lineal y angular deseada.

NODO MOVE_BASE

Para hacer la navegación en ROS tenemos el paquete/nodo move_base. Este conjunto de algoritmos y funciones proporciona un sistema o implementación que, dado un objetivo en el mundo, intentará alcanzar controlando una base móvil. El nodo move_base recibe:

- información de un mapa creado previamente o un servicio que lo esté creando en tiempo real,
- los valores de los sensores exteroceptivos en tiempo real,
- la odometría de la propia base, y
- las transformaciones entre todos los links (es decir, entre el map, base_footprint, laser_link, base_link, etc.)

internamente tiene implementado un planificador local y un planificador global que van ligados a un mapa global y a un mapa local respectivamente, que a su vez van generando un comportamiento de recuperación, por sí el robot quedase atrapado o no pudiese seguir, es lo que realiza la navegación en sí como ya se vio anteriormente. Así se genera el control de la base, en forma de velocidad lineal y angular que debe llevar el robot en cada instante.

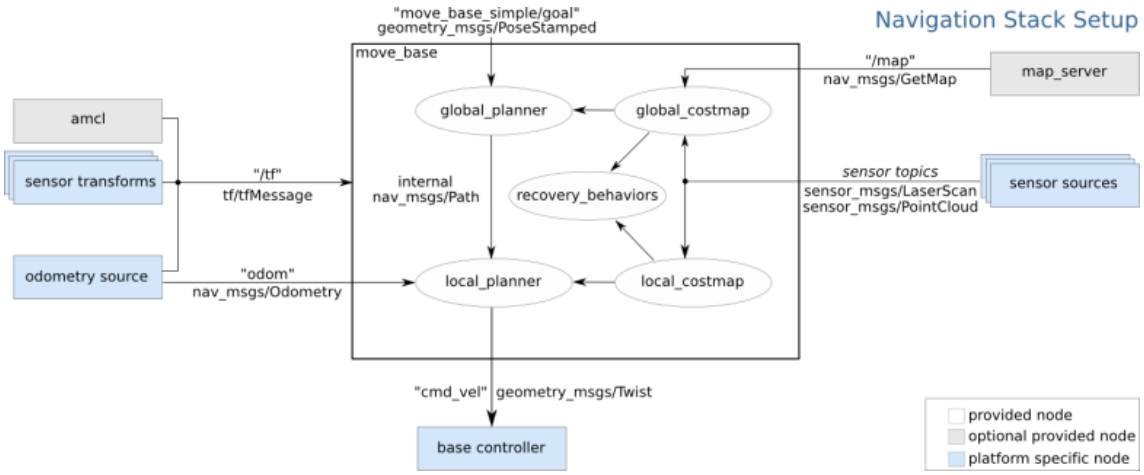

Figura 8. Paquete de navegación de ROS.

Cabe destacar que el robot al que se le puede aplicar este paquete tiene que tener características especiales, como que el robot tiene que tener rodaje diferencial u holonómico. La forma del robot debería ser cuadrada, rectangular o circular. Sin embargo, también puede hacer ciertas cosas con robots bípedos tales como localización, siempre y cuando el robot no se desplace en diagonal. Se requiere que el robot envíe información acerca de la unión de este con sus sensores, es decir, los transforms o TFs. El robot debe recibir y enviar información de velocidades lineales y angulares. Un láser plano debe estar ubicado sobre el robot para crear el mapa y realizar la localización. También se pueden generar alternativas similares con la mayoría de sonares y láseres, o proyectar sus valores medidos sobre el suelo si estos están montados en otro lugar del robot.

2.2.2 LabVIEW

LabVIEW, de National Instruments, es un lenguaje de programación G o programación gráfica, que se basa en bloques de control y cables que simulan el flujo de datos entre bloques. Esta principalmente dedicado o diseñado para la ingeniería de sistemas, control, medidas e instrumentación. Sus programas se escriben en ficheros llamados Vis (Virtual Instrument) que constan de dos partes principales: una interfaz gráfica (GUI) y un diagrama de bloques.

En la interfaz gráfica se muestran los controles e indicadores en forma de leds, gráficos, barras y demás elementos que permiten al usuario introducir o leer información, cada

elemento tiene su equivalente en el diagrama de bloques para poder controlar la información que entra o sale de cada elemento.

En el diagrama de bloques está la programación propiamente dicha, en ella aparecen los bloques que controlan la interfaz gráfica, y a su vez se le añaden bloques de control, para hacer operaciones y procedimientos sobre los datos que fluyen por los cables, existen bloques para infinidad de cosas, entre los más básicos: bucles de control, operaciones matemáticas, operaciones booleanas, arrays y cluster de datos, operaciones de adquisición de datos, control de errores...

Algunas funciones muy útiles de LabVIEW es que:

- Cuando se utilizan placas de National Instruments, el tratamiento de señales para adquirir variables es muy sencilla, además de la programación se hace tan simple como arrastrar un bloque al diagrama de bloques.
- Se puede realizar un examen de la programación para buscar errores directamente con un modo que ofrece el mismo software para ello, el modo Debugging.
- Permite tener una interfaz gráfica muy sencilla de programar (arrastrando bloques de librería), esto hace que el HMI de un VI, sea sencillo y fácil de construir.
- Una cualidad también es la facilidad de creación de SubVIs, que son VIs propiamente, pero con la característica que tienen asociado un icono y un esquema de entradas y salidas del bloque, así como si de programas o funciones se tratasen, luego se pueden utilizar en otros programas más complejos y aumentar la limpieza en los diagramas de bloques, facilitando la lectura y comprensión por otras personas distintas al programador, además de facilitar la mejora o escalabilidad de cada subproceso o SubVI.

LabVIEW es un software muy conocido y experimentado, es por eso que existen infinidad de tutoriales y explicaciones de cómo hacer desde los Vis más sencillos a crear complejos sistemas. [2] [3]

CAPÍTULO 3: COMUNICACIÓN ENTRE PLATAFORMAS DISTINTAS, LABVIEW Y ROS

2.1 LIBRERIAS DE ROS PARA LABVIEW

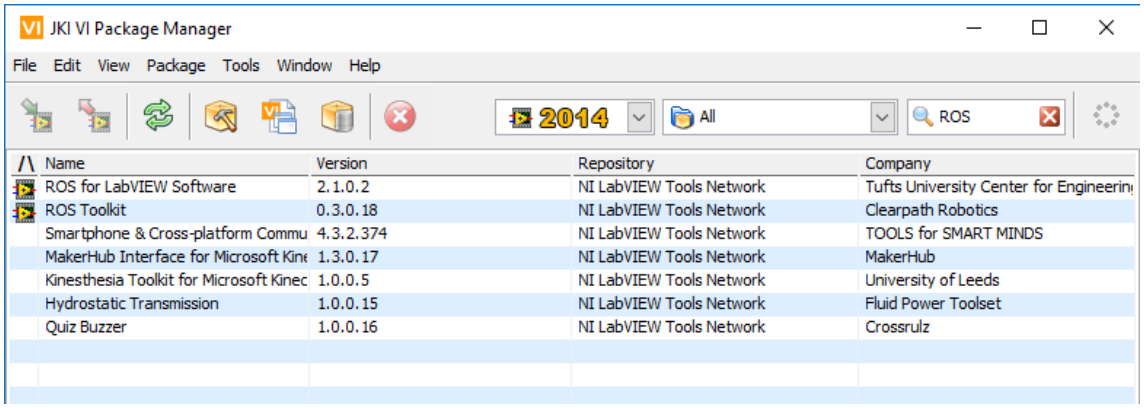

Figura 9. VI Package Manager, librerias relacionadas con ROS.

Intentamos buscar desde VI Package Manager, un complemento de LabVIEW para administrar paquetes y librerias adicionales, paquetes relacionados con ROS, se encuentran únicamente dos librerías que es posible que sean de utilidad, además de ser las dos únicas con referencias en internet si se busca acerca de ellas: la primera, de la Universidad de Tufts, fue la que primero se probó, ya que parecía ser la más testeada al tener una versión estable, y la segunda de Clearpath Robotics que también tiene muchos ejemplos en internet, aunque sea versión 0.

3.1.1 ROS FOR LABVIEW SOFTWARE (TUFTS)

Esta librería ofrece dos opciones principales, una que enlaza directamente con ROS, que da opciones básicas, como crear un topic, escribir o leer de este, construir un mensaje o desmenuzarlo, etc.

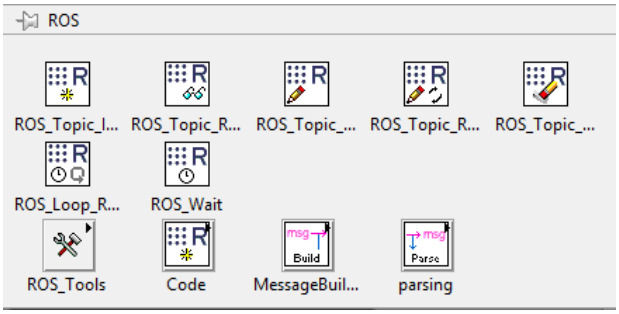

Figura 10. Librería Ros for LabVIEW Software (I)

La segunda opción va más relacionada con plataformas robóticas específicas, tiene para Baxter, ROSRIO, NAO y Turtlebot, la que interesa para este proyecto es ROSRIO, puesto que es la plataforma que se utiliza. En esta librería hay bloques para publicar o suscribirse a un topic y alojar un servicio o hacer una petición a uno como se muestra en la figura siguiente.

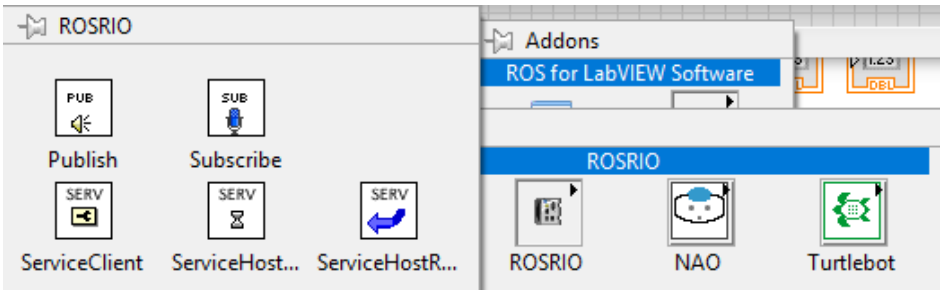

Figura 11. Librería Ros for LabVIEW Software (II)

Para probar si es útil y ver si cumple con las necesidades que se requieren, se hace un VI muy básico: crear el tag de un topic, suscribirse a él e intentar leer un topic que se publica desde ROS cada 100ms llamado cmd_vel, del tipo geometry_smgs, y mostrarlo en la pantalla. Con esto pretendemos que al ejecutar dicho VI, aparezca algún valor de texto o similar en el bloque 'Reply' cuando desde ROS publiquemos dicho topic.

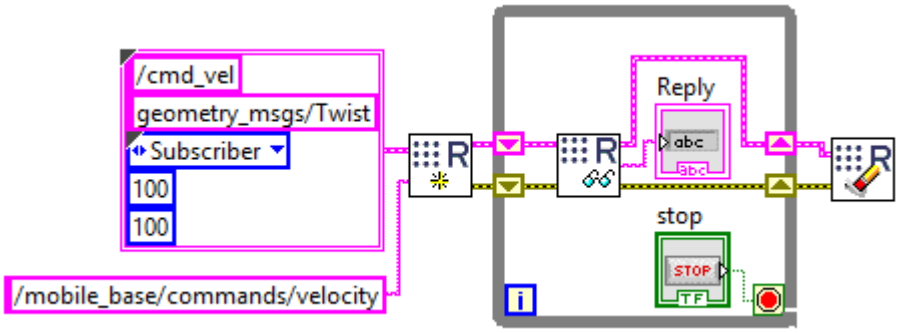

Figura 12. Diagrama de bloques para subscribirse al topic /cmd_vel.

Primero lo probamos corriendo el VI en el PC, lo ejecutamos, nos pide la dirección IP del Host de Roscore, en el NUC ejecutamos el comando para publicar un topic con el tipo geometry_msgs/Twist y como se esperaba, funciona. Luego, para ejecutar el VI en el DANI, se utiliza un ejemplo de LabVIEW Robotics, como se nombró al principio, el "Starter Kit 1.0". Sobre este ejemplo, que ya incorpora todas las librerías y Vis necesarios para que el robot funcione y se pueda cargar el programa en él, se introdujo dicho VI y se ejecutó. Automáticamente se carga en la placa y pregunta la IP del host, el dispositivo que aloja el roscore, se introduce la IP del NUC y comienza a ejecutarse. Ahora dentro del NUC en una terminal se ejecuta la base Kobuki (es como si se ejecutase un publicador del topic /cmd_vel simple, pero en vez de eso se hace desde un sistema más complejo, que creará varios nodos y topics que usa para su funcionamiento, entre los que está el que a nosotros nos interesa /cmd_vel):

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

Al hacer esto, en el VI debería aparecer en el cuadro de texto algo como lo que aparecia cuando se hacía el experimento usando el PC en vez de la sbRIO:

```
"linear:
```

```
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

Pero en cambio no se recibe ni se lee nada. Se probó para diferentes topics y nodos, y se obtiene el mismo resultado, ninguno.

Se intente hacer el proceso a la inversa, a ver si se puede escribir o crear un topic nuevo con el código de la figura, similar al anterior.

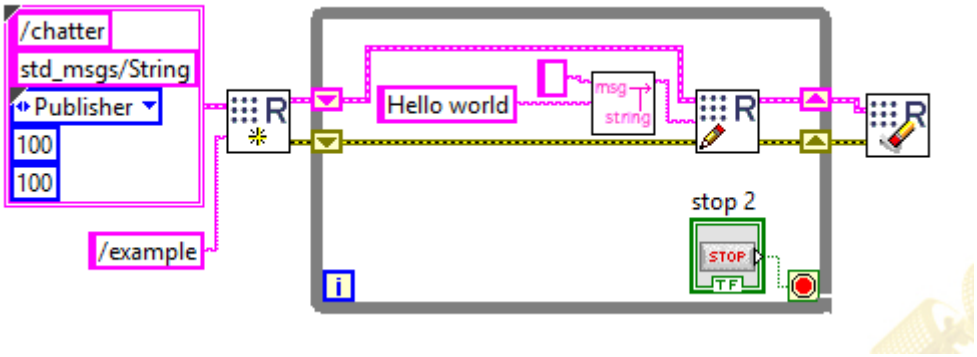

Figura 13. Diagrama de bloques para publicar el topic /chatter.

Al escribir en una terminal, con "roscore" ejecutado en otra:

```
>> rostopic echo /chatter
```

En el PC sí que se recibe la cadena de texto, pero usando la sbRIO no es solo que no se reciba la cadena "Hello world", sino que un warning dice que el topic no ha sido creado, lo que puede significar dos cosas:

- Una que no se estén comunicando, a pesar de que entre las dos máquinas se hacen ping mutuamente.
- O que, si se estén comunicando, pero no se están entendiendo entre las plataformas, que puede ser culpa de la sbRIO.

Es importante mencionar que usando el ordenador se podían hacer las dos operaciones en el mismo VI, es decir, publicar y subscribir, aunque debían ser topics diferentes, pero que utilizando la sbRIO por alguna razón no funciona nada de lo que se probó.

También se probó a seguir una guía creada por Clearpath Robotics [4], que aun no siendo el creador de esta librería, aporta más información que la Universidad de Tufts. Siguiendo dicha guía sucede exactamente lo mismo, si se prueban los VIs sobre el PC funcionan tal y como se va diciendo en la guía, pero si se ejecutan los mismos VIs en la sbRIO es como si no hubiese comunicación. (Se insiste en que la comunicación si era posible, ya que entre ellos ping se hacen y hay respuesta de ambos)

En cualquiera de los casos, se descarta la opción de usar esta librería de inmediato, ya que este proyecto necesita del funcionamiento de estos VIs en la sbRIO y por tanto no es de utilidad.

3.1.2 ROS TOOLKIT (CLEARPATH ROBOTICS)

En esta segunda librería que aún está en fase beta, se encuentran prácticamente los mismos bloques que en la otra librería: abrir, cerrar, escribir y leer de ROS.

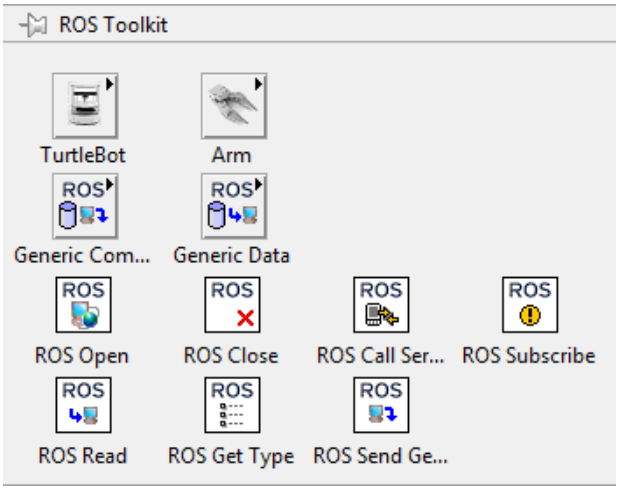

Figura 14. Librería ROS Toolkit.

Para probar su funcionamiento, se comenzó por intentar seguir una guía que ellos mismos tienen en su web [5], se intenta ejecutar uno de los ejemplos que tiene la librería, y se puede observar de primeras que no deja siquiera ejecutarlo en el ordenador debido a que no todos los bloques tienen o encuentran su SubVI, es decir, hay partes del código que no están dentro de la librería y que deberían estarlo. El ejemplo es el siguiente:

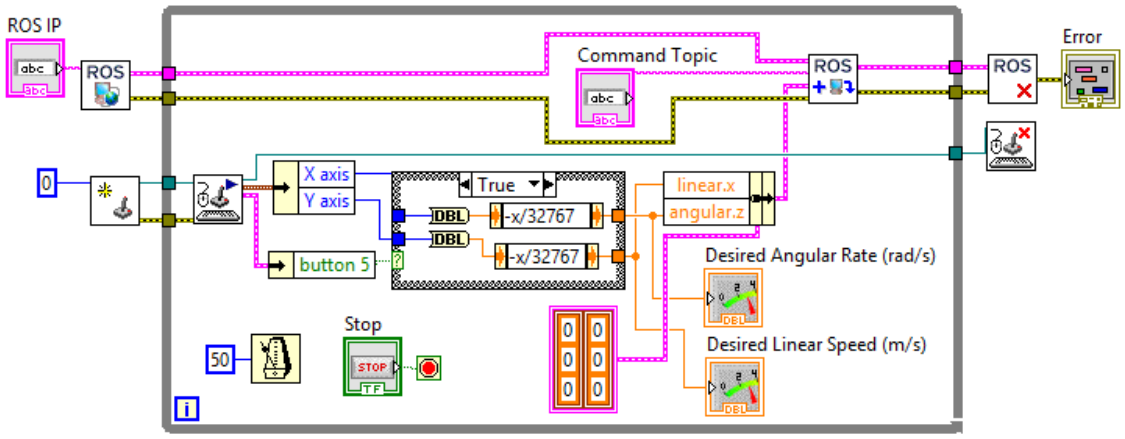

Figura 15. Diagrama de bloques del ejemplo de Clearpath Robotics.

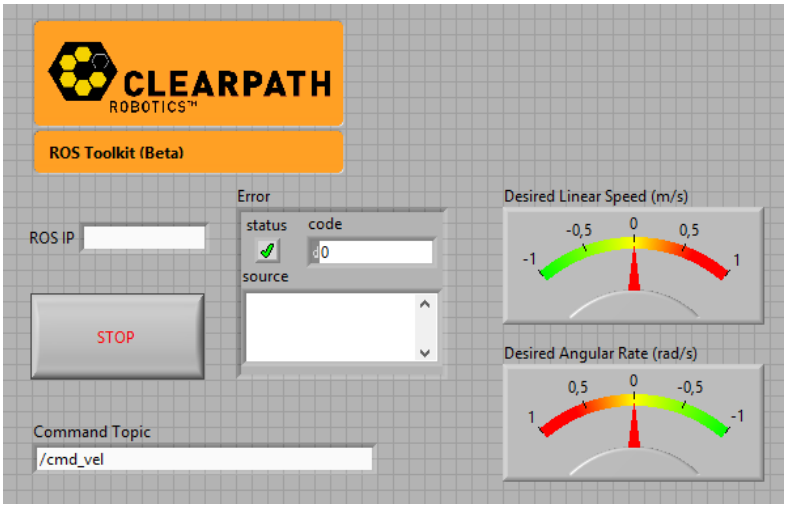

Figura 16. HMI del ejemplo de Clearpath Robotics.

Uno de los bloques que LabVIEW dice que no encuentra el SubVI es precisamente el principal y más básico, ROS Open, sin él no podemos crear una ID de comunicaciones y pasársela al resto de bloques. Debido a que no se dispone de dicho módulo, no se puede intentar simplificar el ejemplo quitando algunos módulos que pudiesen ser problemáticos. De haber sido de otra manera, se podría haber creado un VI sencillo con los bloques básicos, abrir la comunicación ROS, publicar/suscribirse a un topic y cerrar la comunicación, y haberlo probado. Remarcar que aún no funcionando en el PC por la falta de SubVIs, se probó a cargar en la sbRIO pero no fue posible por el mismo error, bloques que no se encuentran. Por esto, y añadiendo que esta librería solo deja utilizarla durante 30 días gratuitamente, esta opción también se descarta

3.2 ROS-Bridge

Dado que no se dispone de una librería específica de ROS para LabVIEW que satisfaga las necesidades del proyecto, se investiga acerca de cómo conseguir comunicar estas

dos plataformas de trabajo. Se da navegando por un foro con una librería para ROS que podría ser de ayuda, ROS-Bridge [6]. Esta librería permite ejecutar un programa, básicamente un servidor, que por medio de un puerto de la red acepta un par de cadenas de texto (con formato json), es capaz de analizarla y crear/publicar un topic o de leerlo/subscribirse.

3.2.1 CONEXIÓN TCP/IP

Para probar esta opción se hace un programa en LabVIEW que genere String, que para ser analizado por rosbridge tiene que tener la estructura de un json. Se comienza por ejecutar un ejemplo que viene con la librería y se lee un topic para ver que estructura tienen los json. El ejemplo es el siguiente, y vale para cualquier topic.

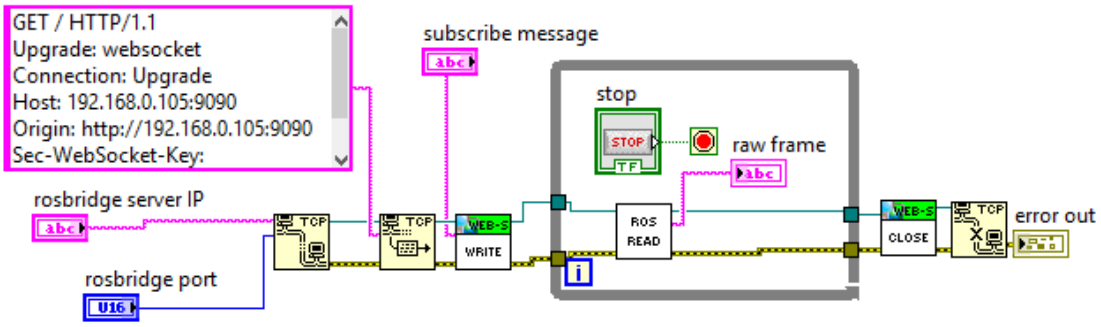

Figura 17. Diagrama de bloques para leer el formato json de un topic.

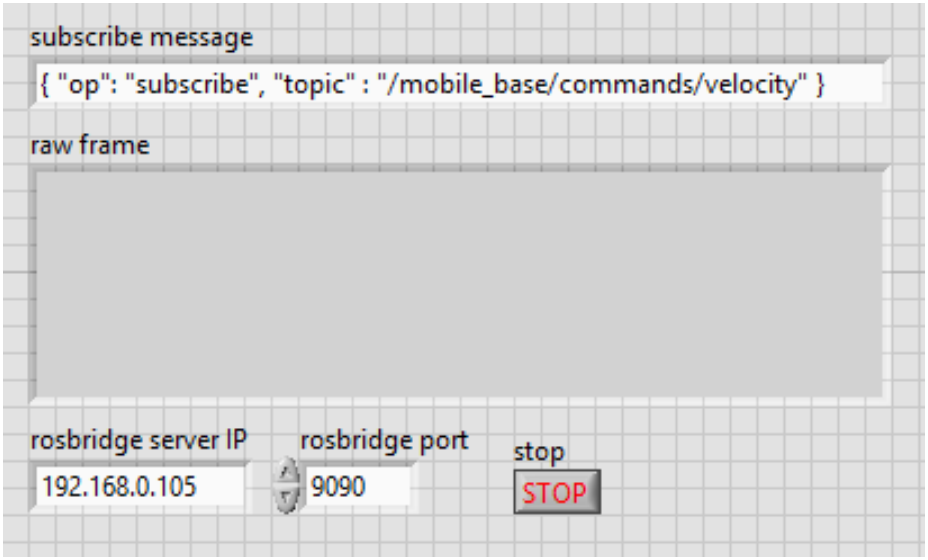

Figura 18.Interfaz del diagrama de bloques anterior.

Para saber el formato que tiene algún topic, hay que poner el nombre del topic que se quiera analizar su estructura de json, en el 'subscribe message' de la interfaz (donde pone /mobile_base/commands/velocity poner este topic u otro cualquiera), se configura la IP del host y el puerto que se abrirá y se ejecutará, en el NUC ejecutar 'Roscore' y publicamos el topic con el tipo de dato que se quiera analizar. Entonces en 'raw frame' nos saldrá la cadena tal cual tiene que se puesta en el mensaje de escritura.

Entonces en el VI se prueba a generar el topic /odom, que será el que se envíe desde LabVIEW a ROS, con el formato correcto, y así es como queda el código y la interfaz:

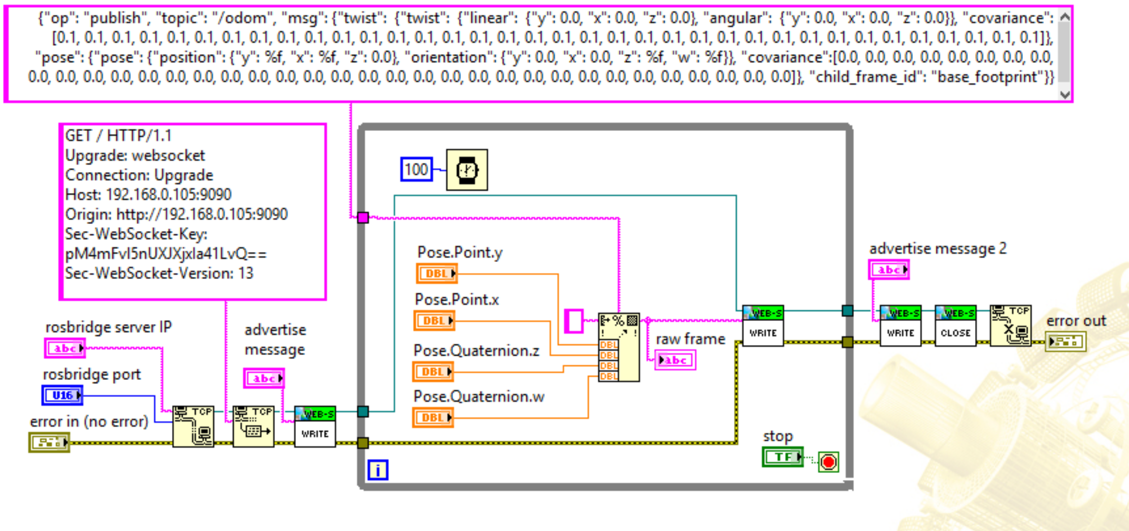

Figura 19. Diagrama de bloques para publicar un topic vía TCP.

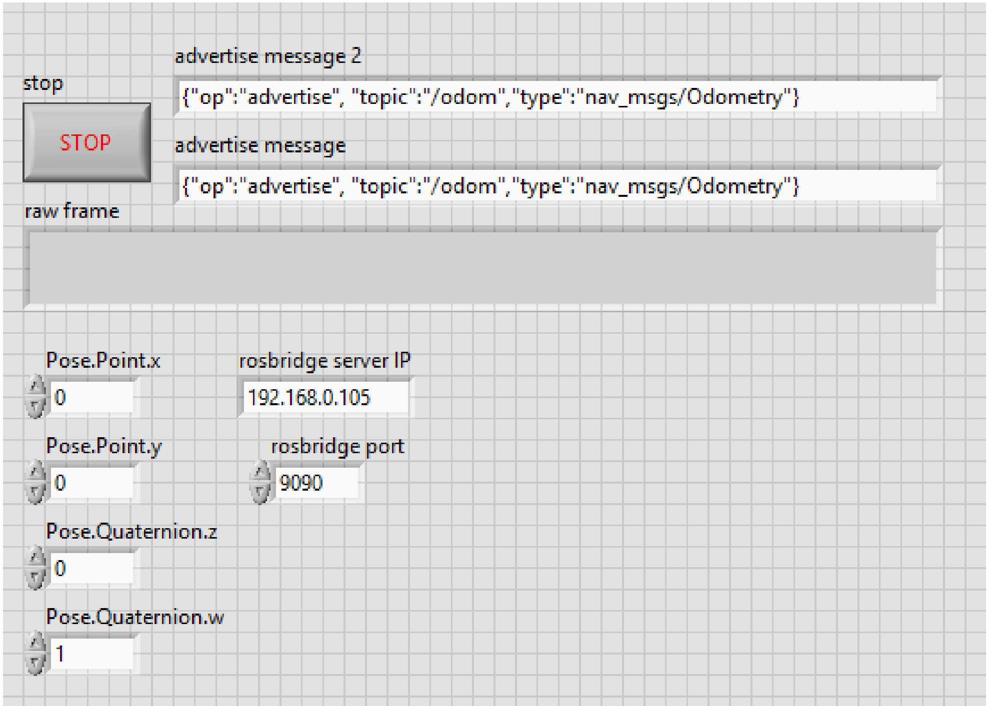

Figura 20, HMI del diagrama de bloques de la figura anterior.

El programa lo que hace es abrir el puerto 9090, en la IP 192.168.0.105, que son los datos del NUC (cuando se hizo el experimento), y le pasa un String que representa el json que interpretará la librería de ROS-Bridge, con el protocolo TCP/IP.

En realidad, se pasan dos cadenas de texto, la primera que es el 'advertise message', le dice al servidor de Rosbridge que la próxima instrucción será publicar un topic de nombre /odom y del tipo nav_msgs/Odometry. Y la segunda envía la operación, el

nombre del topic y el topic en sí, que tiene el formato tal cual lo obtuvimos de analizar el tipo de dato 'nav_msgs/Odometry':

```
{"op":"advertise", "topic":"/odom", "type":"nav_msgs/Odometry"}
```

```
{"op": "publish", "topic": "/odom", "msg": {"twist": {"twist": {"linear": {"y": 0.0, "x": 0.0, "z": 0.0}, "angular": {"y": 0.0, "x": 0.0, "z": 0.0}}, "covariance": [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]}, "pose": {"pose": {"position": {"y": %f, "x": %f, "z": 0.0}, "orientation": {"y": 0.0, "x": 0.0, "z": %f, "w": 1.0}}, "covariance":[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]}, "child_frame_id": "base_footprint"}}
```

En rojo, los '%f' que se ponen, es para que el bloque 'Format into String' introduzca los valores numéricos (doubles) que se quieran y son los tres únicos valores que cambiaran durante la ejecución del VI.

Nombrar que el topic /odom también lleva un 'Header' que es un identificador, este contiene un ID que da el ROS Master (Roscore), y un 'stamp' que es el tiempo de ejecución de ROS en segundos y nanosegundos que utiliza para la sincronización con otros topics. Pues como se puede observar en la cadena de texto anterior, este 'Header' no se está enviando, y por lo que se pudo probar, ROS tampoco le asigna uno, es decir que el topic no se está sincronizando con el resto de topics y programas en ejecución. Más adelante se dará una solución a esta situación.

Para probarlo, simplemente se carga en el DANI este programa/VI, y en el NUC se ejecuta en terminales distintas: Roscore, el comando que lee el topic /odom y el servidor TCP de Rosbridge configurado en el puerto 9090.

```
>> roscore
```

```
>> rostopic echo /odom
```

```
>> roslaunch rosbridge_server rosbridge_tcp.launch
```

Este experimento funciona correctamente, pero no envía o recibe el topic en tiempo real, es decir, desde que se comienza a publicar desde sbRIO hasta que se tiene respuesta en la pantalla del pc, transcurre un tiempo considerable.

El VI definitivo que llevará el DANI para su control a nivel bajo se desarrolla bajo este tipo de conexión, TCP/IP, se logra probar y mover el DANI con el mando de la PS4, pero el problema de la latencia en la comunicación persiste. Desde que se da una orden con el mando, hasta que el DANI reacciona, el tiempo transcurrido era de varios segundos.

Este retardo en las comunicaciones puede ser debido a muchos motivos, investigando sobre el tipo de conexión TCP y otros protocolos, llegamos a una conclusión. Por un lado, el protocolo TCP es más lento cuando el tráfico en la red es medio o bajo, transmite los datos en bytes y además se espera a que haya sincronización en la comunicación entre la fuente y el destino. Por otro lado, la comunicación por UDP (que desde LabVIEW también se puede utilizar), no necesita sincronización entre el emisor y receptor del mensaje, trabaja con paquetes enteros, y no es fiable ya que ante la pérdida de un paquete el emisor no lo sabe y no puede poner remedio.

Así que se plantea modificar el Vi que funciona con TCP y transformarlo para que utilice UDP, ya que la única pega que se tenía de la pérdida de paquetes y de información no era un problema. No era un problema al no repercutir en el sistema, si se envían paquetes desde LabVIEW cada 20-30 milisegundos, en ROS solo se necesita un refresco de los topics de entre 90-120 milisegundos, es decir, de cada cinco paquetes que se enviaban, se podía permitir el sistema perder cuatro, aunque obviamente no se perdía casi ninguno (el tráfico en la red era mínimo como para que se perdiesen).

3.2.2 CONEXIÓN UDP

Se prueba a realizar una lectura de un topic por UDP con el código de la figura, el programa es igual que el de TCP que se probó al principio, el que solo leía: abrir un puerto en una IP, suscribirse a un topic y leer lo que se recibe por el puerto; pero se cambian los bloques de TCP por los de UDP, y adicionalmente se añade un Timed Loop para ver si el proceso cumple con las necesidades temporales. Así se consigue que nos lea la velocidad del DANI en tiempo real. Para ponerlo a prueba, mientras el DANI ejecutaba el VI de la figura siguiente, en el NUC se tenía ejecutando en diferentes terminales: el Roscore, el Turtlebot_Bringup (aunque no se utiliza el Turtlebot, es solo para generar el topic /mobile_base/commands/velocity), y el joystick de la ps4, para poder variar el topic que se quiere leer. Adicionalmente, se ejecuta el programa de del servidor UDP configurado para el puerto 9090.

```
>> rsocore

>> roslaunch turtlebot_bringup minimal.launch

>> roslaunch Turtlebot_teleop ps4_teleop.launch

>> roslaunch rosbridge_server rosbridge_udp.launch
```

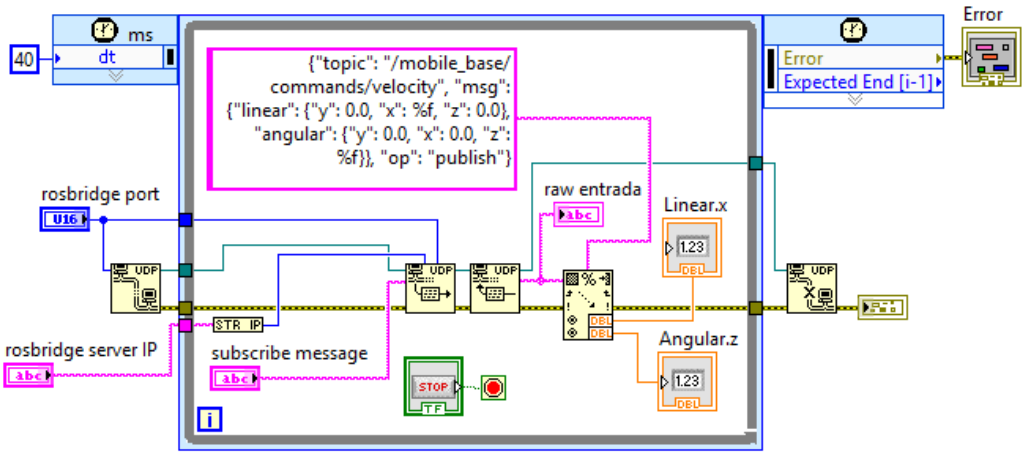

Figura 21. Diagrama de bloques para subscribirse a un topic vía UDP.

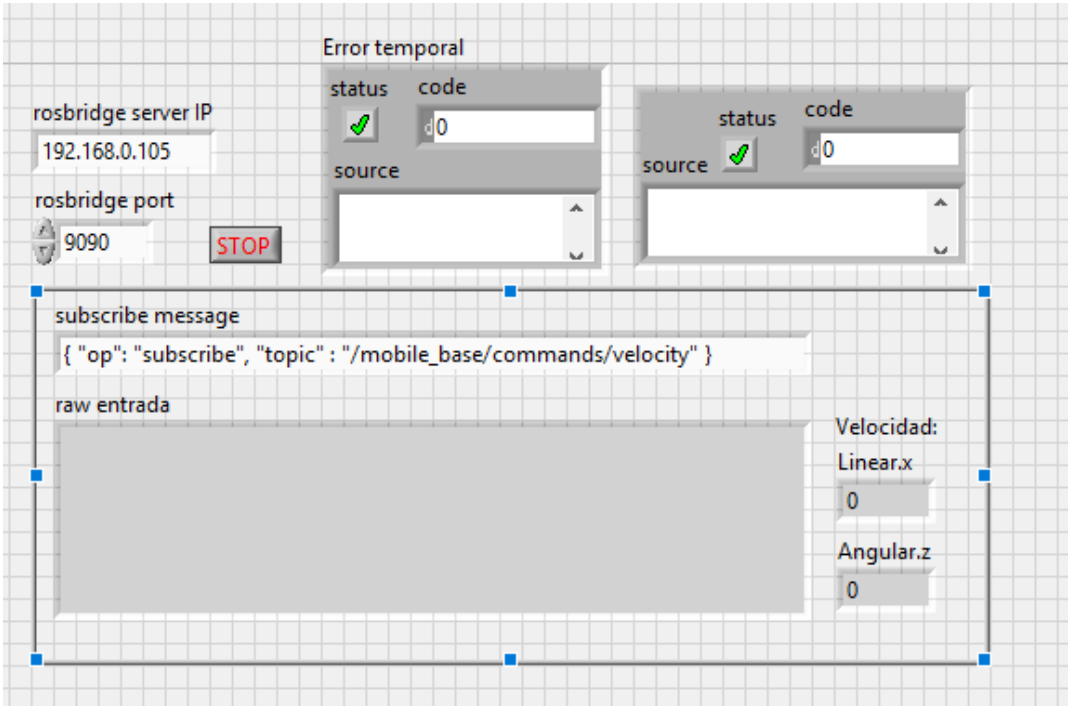

Figura 22. HMI del diagrama de bloques de la figura anterior.

Este VI funciona correctamente, no hay retrasos apreciables desde que se mueven los joysticks y aparecen los valores en la pantalla del ordenador (el programa cargado en la sbRIO) y reacciona el DANI, se puede mover con gran precisión en cuanto a tiempo se refiere, y no se aprecia retraso en las comunicaciones. Teniendo en cuenta además que: los movimientos sobre el mando deben pasar por Bluetooth hasta el NUC, ser procesados, publicar el topic /joy, internamente se traduce a velocidades en el topic /mobile_base/commands/velocity, se publica por UDP a través del servicio que se ha creado, el DANI recibe la cadena de texto en forma de json, se interpreta la cadena y se escriben dichas velocidades en los motores; la latencia con este VI es considerablemente menor.

No se conoce la causa que hace que no haya retraso, ya que puede ser por el cambio de TCP a UDP, o también por el cambio de bucle loop por un Timed loop, o incluso por los subVIs de WebSocket que se utilizaban con TCP, que posiblemente era lo que ralentizaba el proceso. Aun no conociendo la causa real, la latencia es insignificante y se puede usar esta configuración de comunicación.

Se procede a crear un programa que haga las dos cosas, subscribirse a un topic y crear y publicar otro distinto. Surge el problema de que este programa vuelve a refrescarse muy lento. Esto se debe a que hay que enviar cada vez un código para decirle a Rosbridge la operación que se desea hacer, es decir: cada vez que se vaya a publicar debemos hacer un advertise al servidor UDP y luego enviar la cadena json como ya se explicó en un apartado anterior, y para subscribirse primero se tiene que enviar una orden de suscripción al servidor (parecida a la de advertise):

{"op": "subscribe", "topic": "/mobile_base/commands/velocity"}

Y posteriormente leer el json. El tener que hacer cuatro operaciones en el mismo puerto, en el mismo bucle, hace que la frecuencia del bucle y por tanto la frecuencia de actualización de los topics fuera de 130ms. Para solucionarlo, se optó por abrir dos puertos diferentes en nuestro host de ROS, el 9090 y el 9091, cada uno dedicado a una operación, uno a subscribirse y otro a publicar respectivamente; y en LabVIEW se crean dos bucles independientes para que una operación no retrase a la otra, además de conseguir tener que hacer el 'advertise message' una sola vez en cada puerto, y luego solo leer o escribir según el objetivo de cada bucle del VI..

Por todo lo expuesto, esta opción es en la que se basa este proyecto para lograr comunicar ROS y LabVIEW de manera eficaz y en tiempo real, con posible pérdida de datos pero que no afectaría de manera significativa al proyecto.

3.3 ARQUITECTURA DE COMUNICACIONES ADOPTADA

En cuanto a las comunicaciones de topic e información se refiere, casi todo se centra en el NUC. Es el NUC el que se encarga de recibir datos del mando de PS4 y lo convierte en el topic /joy, recoge la información que el LIDAR envía por la red y lo convierte en el topic /scan, a la vez ejecuta dos servidores UDP: uno para enviar el topic /mobile_base/commands/velocity, y otro para recibir la odometría del robot en /odom, envía los datos de operación para que se pueda controlar desde el PC de trabajo y a su vez ejecuta todas las funciones de la navegación, como: roscore, slam_gmapping, el transformador de /odom a /tf (se explica más adelante) y el simulador rviz para ver el robot.

Por otro lado, en el PC de trabajo solo se ejecuta por un lado un cliente de SSH para controlar el NUC, y por otro LabVIEW para cargar el programa del sbRIO y ver su HMI.

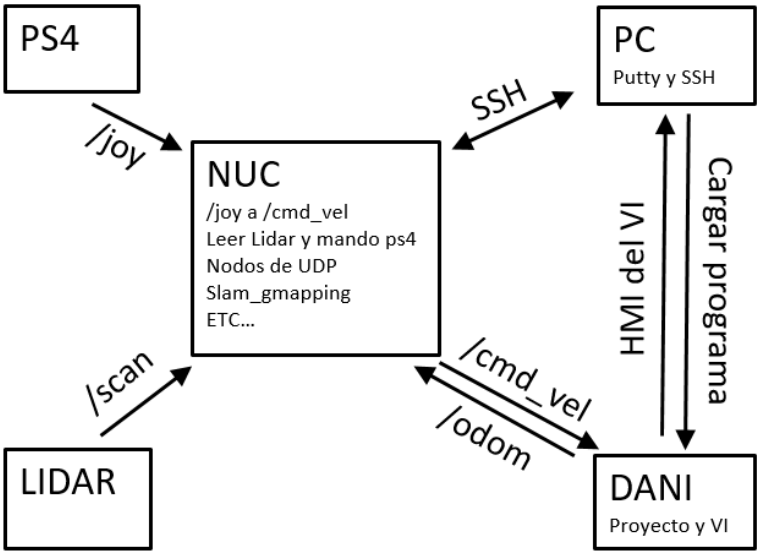

Figura 23. Diagrama de la arquitectura de comunicaciones.

CAPÍTULO 4: PUESTA EN MARCHA DEL ROBOT Y NAVEGACIÓN CON ROS

4.1 INSTALACIÓN DE SOFTWARE NECESARIO

El primer paso será verificar e instalar si fuese necesario los programas y librerías en el NUC, que será la estación principal de computación. Aunque se pueden utilizar versiones similares y posteriores, se recomienda usar como sistema operativo Ubuntu 16.04 LTS, que es con el que se ha hecho el proyecto y se está seguro de que funciona.

Sobre este SO, se instala ROS siguiendo los pasos que vienen en la propia página del software y asegurándose de usar la versión Kinetic. Para esta versión se puede instalar cualquiera de las opciones que da, pero recomendamos la instalación full/completa.

Después de crear la carpeta de proyectos catkin_ws, se introducen todos los ficheros de la carpeta “tfm”, que se reutilizan de un proyecto de TFM anterior a este [1], en el que se explica de manera detallada el funcionamiento de los códigos utilizados, y el cómo se generaron. A parte de esta carpeta, deben estar las carpetas de “Turtlebot” y “LMS1xx-master” dentro de la carpeta” src” del Workspace y también la llamada “odom2tf”, que contiene un programa en C++ propio que se explicará más adelante.

La carpeta ‘tfm’ contiene programas modificados como por ejemplo el slam_gmapping vinculado al LMS111 en vez del al Orbec Astra. La carpeta ‘Turtlebot’ contiene todo lo relacionado con el robot Turtlebot. La carpeta ‘LMS1xx-master’ es una librería en realidad, que sirve para vincular el láser y crear el topic /scan a partir de los datos que se reciban del sensor. Y por último, la carpeta ‘odom2tf’ solo contiene un programa en C++ que se subscribe al topic /odom y crea/publica el topic /tf.

Por último, para tener listo el NUC, se puede escribir un script para ejecutar todos los comandos necesarios más fácil. Se salva con la extensión .sh en el directorio Home.

El script ejecuta uno por uno los comandos necesarios para la puesta en marcha del DANI, primero hace el catkin_make, para compilar todos los programas y crear sus ejecutables, después ejecuta en un terminal nuevo cada comando y se espera x segundos a que ese comando se haya iniciado. Estos comandos por orden de ejecución son, primer el ‘roscore’, abre un puente UDP, abre el segundo puente UDP, se abre la conversión de topics del mando de PS4, se ejecuta el odom2tf para publicarel topic /tf, se inicia el mapeado con SLAM (vinculado al láser), y por último se ejecuta el simulador rviz para ir viendo el resultado del mapeado.

*script.sh (~/) - gedit

Open Save

```
#!/bin/bash
#-*- ENCODING: UTF-8 -*-
echo "<Ejecutando:"
cd catkin_ws
source ./devel/setup.bash
catkin_make
sleep 1
(xterm -e "roscore; bash" &); (sleep .5);
(xterm -e "roslaunch rosbridge_server rosbridge_udp.launch; bash" &); (sleep 1);
(xterm -e "roslaunch rosbridge_server rosbridge_udp2.launch; bash" &); (sleep 1);
(xterm -e "roslaunch turtlebot_teleop ps4_teleop.launch; bash" &); (sleep .5);
(xterm -e "rosrun odom2tf odom2tf; bash" &); (sleep .5);
(xterm -e "roslaunch tfm gmapping.launch; bash" &); (sleep .5);
(xterm -e "roslaunch turtlebot_rviz_launchers view_navigation.launch; bash" &) &
echo "Programa finalinado."
exit
```

sh Tab Width: 8 Ln 8, Col 43 INS

Figura 24. Script utilizado.

Luego, en el ordenador de trabajo, recomendable que sea con SO Windows, se instala LabVIEW Robotics, preferiblemente la versión de 2014 ya que es la que está probada, y se instalan los módulos posteriormente de “Real-Time” y de “FPGA” correspondientes a la versión base, necesarios para poder programar la placa sbRIO integrada en el DANI. Para crear el proyecto buscamos con la ayuda de NI Example Finder (que se puede encontrar en Help>Find Examples…), el proyecto de “Starter Kit 1.0.lvproj” que está dentro de Robotics>Starter Kit como se muestra en la figura siguiente. Una vez abierto el ejemplo, en el explorador de proyecto aparece algo como en las figuras siguientes. Se utiliza este proyecto como base ya que está diseñado para el robot que se usa en este proyecto específicamente, es por eso que la tarjeta sbRIO y demás parámetros ya vienen configurados para nuestra base y solo habría que modificar el VI.

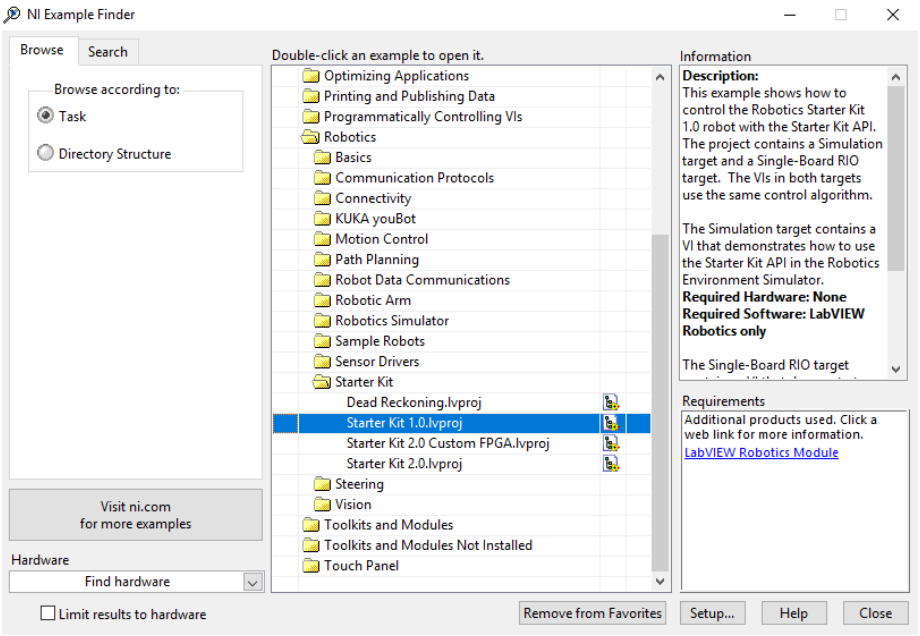

Figura 25. NI Example Finder.

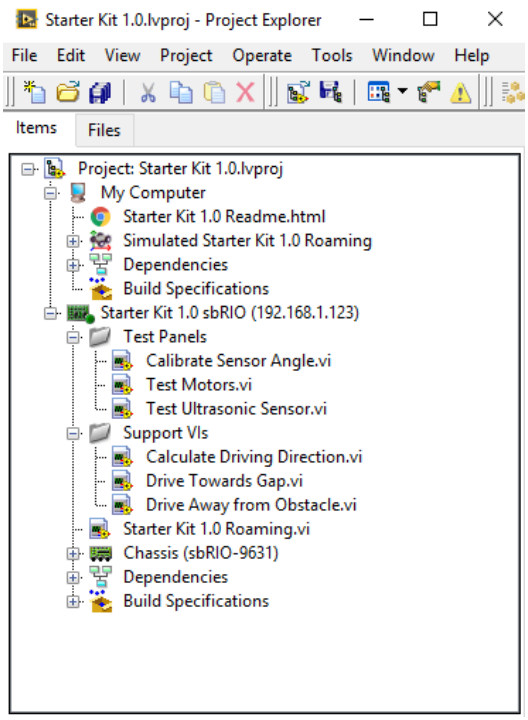

Figura 26. Arquitectura del proyecto Starter Kit 1.0.

Por tanto, se pasa a modificar el VI principal, "Starter Kit 1.0 Roaming.vi" se cambia por el archivo "Starter Kit 1.0 Final.vi" (Anexo B), por último, introducir la IP que tenga la sbRIO del DANI, para poder cargar el programa a través de la red. Para ello haciendo clic derecho sobre la tarjeta, en 'Starter Kit 1.0 sbRIO (X.X.X.X)' donde las X pueden ser cualquier número, entramos en Properties y se modifica el campo de IP Address, poniendo la dirección de la sbRIO que corresponda con la que se va a utilizar. Con esto estará lista la configuración para programar el DANI.

Adicionalmente, para ejecutar el script que se hizo anteriormente dentro del NUC desde nuestro PC con Windows se debe instalar un par de programas llamados: Xming y Putty, ambos gratuitos y fáciles de encontrar. Estos programas sirven para que por la pantalla del PC salgan las ventanas que en realidad se están ejecutando en el NUC.

Si es el caso es que se quiere ejecutar dicho script desde un sistema Ubuntu, hay que asegurarse de que el NUC posee una librería de SSH y esté activado. En el PC con SO Ubuntu no se necesita instalar nada.

4.2 PUESTA EN MARCHA

Una vez instalado y configurado todo lo que se necesita, se pasa a realizar los pasos que harán que el DANI, tenga las mismas capacidades y funcionalidades que tiene el Turtlebot.

Primero se coloca el NUC sobre el DANI, se conecta la alimentación y el cable de red Ethernet y se enciende el NUC y la tarjeta sbRIO.

En el PC de trabajo, ya conectado a la **red del DANI** (para este caso la red se llama 'Dani_Robot' y la contraseña es 'danirobot'), se abre LabVIEW Robotics con el proyecto (ya modificado) y se abre el VI principal, que debería ser el Starter Kit 1.0 Final.vi.

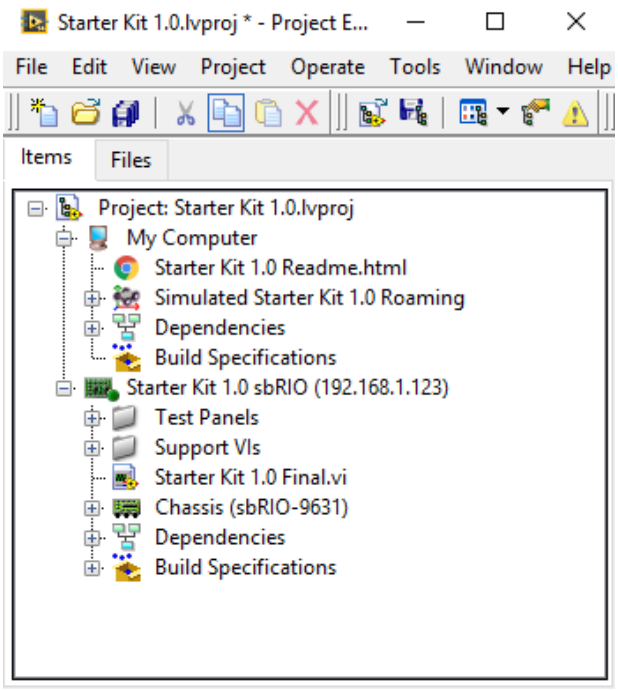

Figura 27. Arquitectura del proyecto Starter Kit 1.0 Modificado.

Si se trabaja con Ubuntu para conectarse al NUC, no hay más que ejecutar:

```
SSH –X <nombre_de_usuario_del_NUC>@<ip_del_NUC>
```

En el caso de este proyecto:

```
SSH –X turtlebot@192.1681.11
```

Se pide la contraseña de este usuario, que es "ros", y ya se tiene un terminal alojado en el NUC, lo que significa que lo que se haga en él, realmente se estará ejecutando en el NUC y no en el PC.

Si por el contrario se trabaja desde el mismo PC que para LabVIEW, que es Windows, se inicia Xming, se busca xLaunch en el navegador (Cortana también sirve), y se siguen los pasos dándole a Siguiente, hay que tener en cuenta el número que se ponga en 'Display Number', se marca también la casilla de 'No Access Control', y Finalizar.

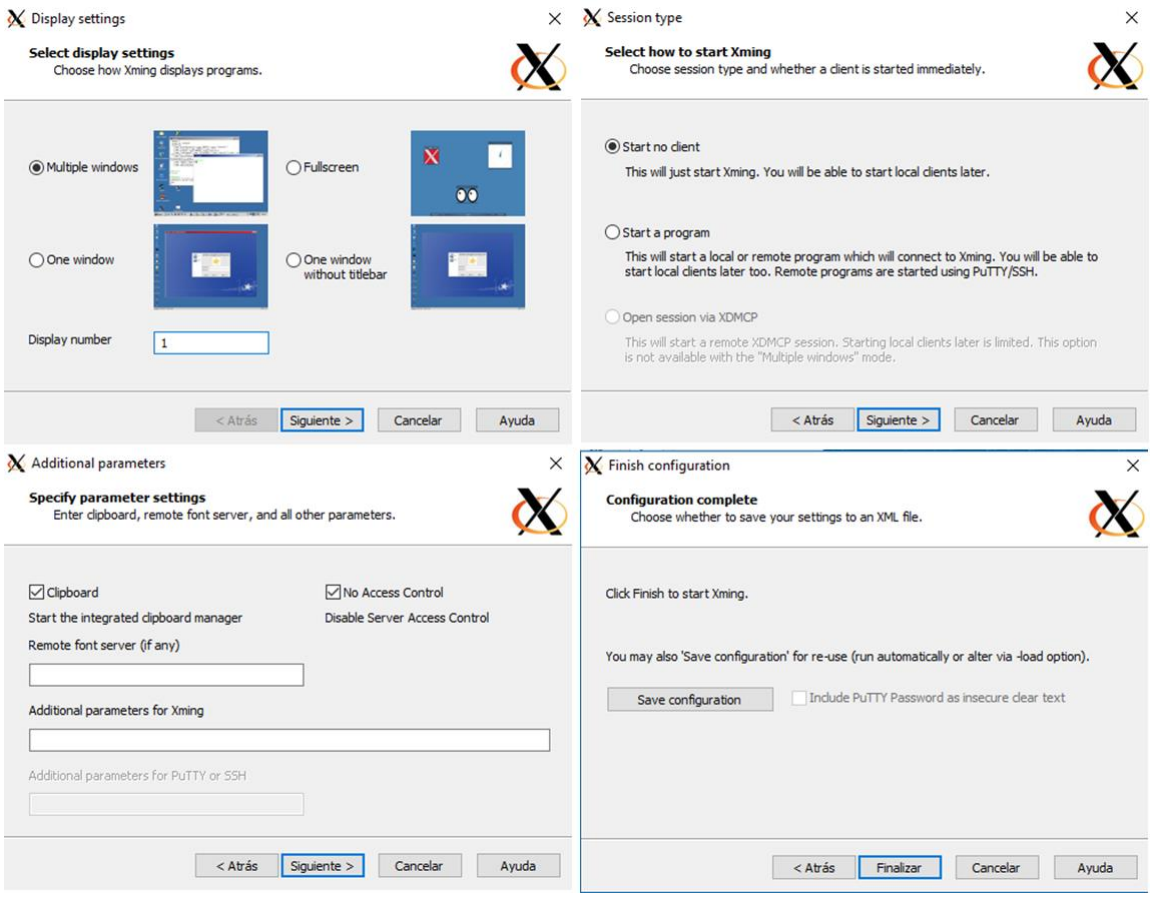

Figura 28. Wizard para crear un DISPLAY en Xming.

Cuando se haya creado el DISPLAY, con Putty se conecta el PC al NUC, con la configuración de la figura siguiente.

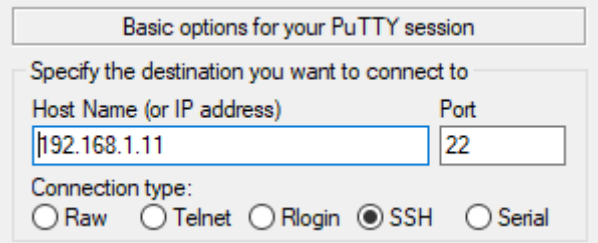

Figura 29. Parámetros para abrir SSH con el NUC.

Se escribe en el terminal que aparece el usuario, la contraseña y unos comandos para que Xming funcione:

- Usuario: Turtlebot
- Contraseña: ros

```
>> export DISPLAY=<ip_pc_trabajo>:<numero_del display>
```

Para saber la IP del PC de trabajo, se busca en Cortana o similar: “cmd”, para abrir el “Símbolo del sistema”, y una vez abierto escribimos:

```
>> ipconfig
```

Y saldrá mucha información, hay que buscar aquella línea que ponga “Adaptador de LAN inalámbrica WI-Fi”, y en la segunda línea se observa la Dirección IPv4, esta es la IP que se debe poner en el comando.

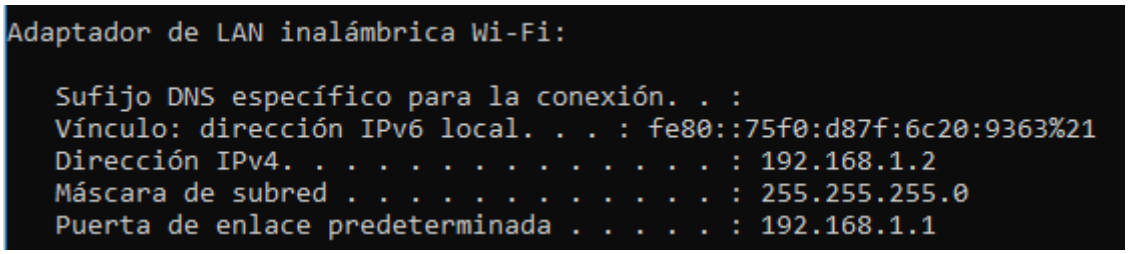

Figura 30. Información de la IP del PC.

Así el comando que habría ejecutar, sabiendo que el número de display se puso como 1, quedaría como:

```
>> export DISPLAY=192.168.1.2:1
```

Con este comando se logra que el NUC envíe la información necesaria que se muestra por pantalla a esa IP, al display configurado anteriormente como 1.

Y ahora ya se puede ejecutar el script en el NUC, escribiendo:

```
>> ./script.sh
```

Comenzaran a aparecer muchas ventanas pequeñas en el escritorio del PC de trabajo, llamadas xterm, que son cada comando del script, y adicionalmente una ventana más grande que es Rviz. Destacar que lo que se muestra en pantalla tiene mucha latencia, va muy lento, pero los programas que realmente están corriendo en el NUC van a su velocidad normal, es decir, lo que se ve por la pantalla del PC va con mucho retraso, el NUC no.

Posteriormente se puede dar a ejecutar al VI en el PC para que se vaya cargando.

Importante: *hay que asegurarse de que el VI comience su ejecución después de que termine el script, si no, no se podrá ni publicar el topic /odom, ni leer el topic /mobile_base/commands/velocity, debido a que la primera parte del VI, que hace que se subscriba a un topic y que advierta al servidor de UDP que se va a publicar otro, no se ejecutará correctamente y la comunicación no se establecerá*.

Adicionalmente, si se va a mover el DANI desde el mando de PS4, se tendrá que encenderlo antes de correr el script, esto último no debería dar problemas si se enciende a posteriori, pero si recomendable.

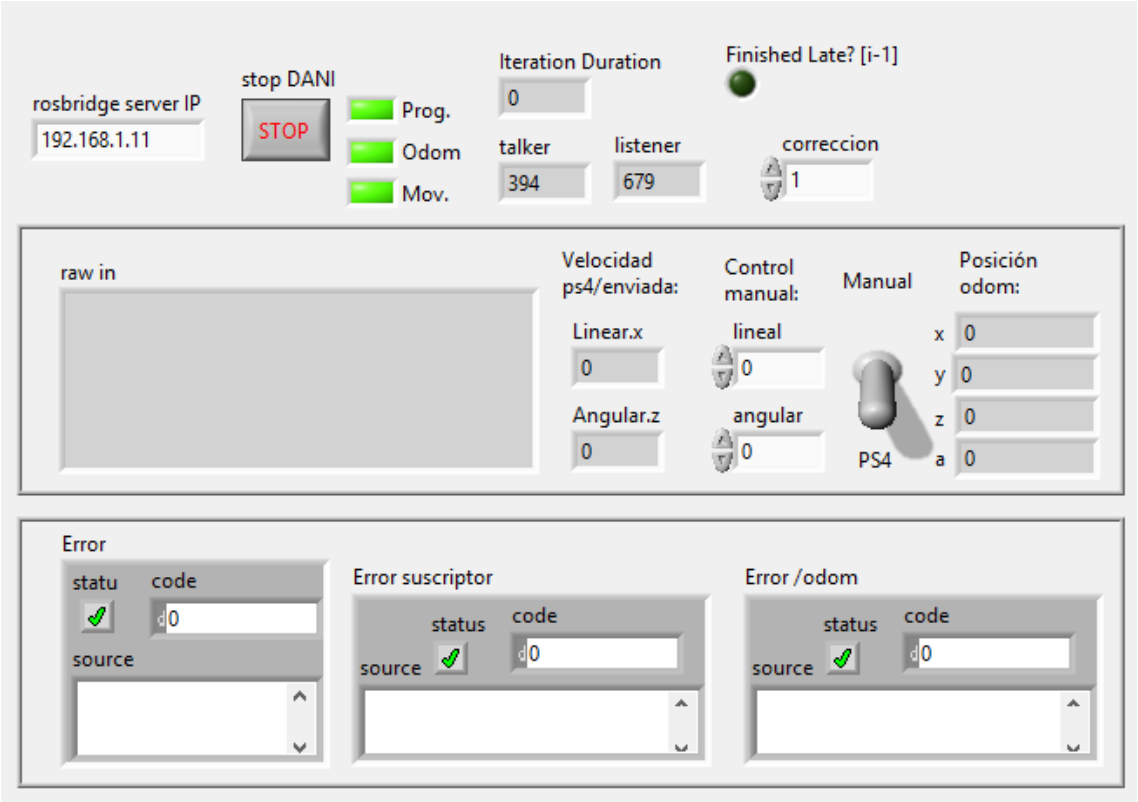

Figura 31. HMI del "Starter Kit 1.0 Final.vi".

Una vez que este todo siendo ejecutado, lo único que hay que hacer es mover el DANI por la habitación que se desea mapear. Hacer movimientos lentos con los joysticks o con las velocidades del VI por varias razones:

- Si la aceleración es demasiado rápida, los encoder pierden pasos y la odometría se vuelve errónea más rápido.
- Si el DANI se mueve rápido, a la persona que está viendo el PC que muestra las pantallas, el mapa en rviz y que no se está actualizando en tiempo real, no podrá ver ciertos fallos en el mapeo que pueden surgir.

Se irá viendo algo parecido a la figura siguiente, y cuando se esté satisfecho con el resultado, el robot se pude detener, en cuanto a movimiento se refiere, no el VI, ni los procesos del NUC, ya que se puede perder todo el progreso. Para guardar el mapa ejecutamos:

```
>> rosrun map_server map_saver -f <directorio_de_la_carpeta>
```

En este caso, se guardará en el mismo lugar que lo hacía Antonio en su TFM [1], del que se recuerda extrajimos los pasos de mapeo y navegación. Por tanto, sería:

```
>> rosrun map_server map_saver -f
→/home/turtlebot/catkin_ws/src/tfm/maps/mi_mapa
```

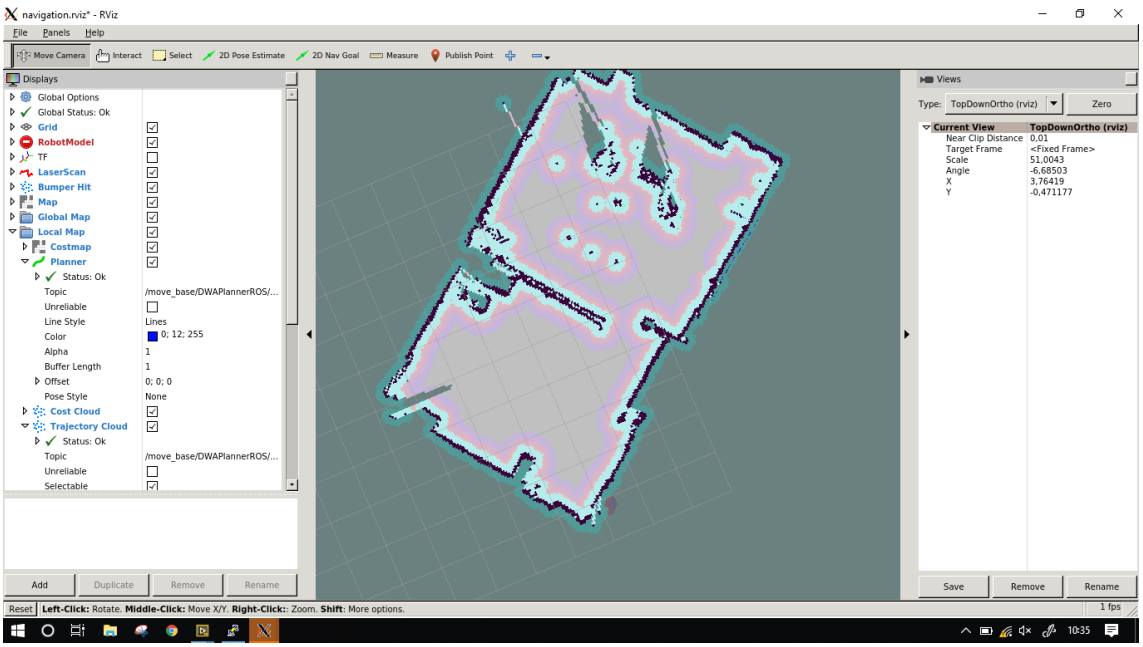

Figura 32. Mapa del laboratorio DSIE, hecho con DANI.

Con esto ya se tiene el mapa de la estancia, y que se puede utilizar para el stack de navegación y hacer que el robot se mueva autónomamente.

Para la navegación, ya no sirve el script que se hizo al principio, y aunque se podría hacer uno distinto que sí valiese, la modificación es muy sencilla y se explicará sin más.

Se necesita ejecutar prácticamente los mismos comandos:

```
>> roscore
```

```
>> roslaunch rosbridge_server rosbridge_udp.launch
```

```
>> roslaunch rosbridge_server rosbridge_udp2.launch
```

```
>> rosrun odom2tf odom2tf
```

Adicionalmente para la navegación se ejecutará:

```
>>roslaunch Turtlebot_navigaion amcl_demo.launch
→map_file:=/home/Turtlebot/catkin_ws/src/tfm/maps/mi_mapa.yaml
```

Como bien se explica en el TFM [1], se debe ejecutar este comando cuando el DANI se encuentre en la misma posición que cuando se comenzó a hacer el mapeo, puesto que ese es el punto que tanto el mapa, como la odometría del DANI cogieron como origen de coordenadas.

Después de esto, y antes de comenzar a darle ninguna orden de 'goal' al robot, recordar que se debe ejecutar el VI en el DANI que es el que controla a bajo nivel el robot.

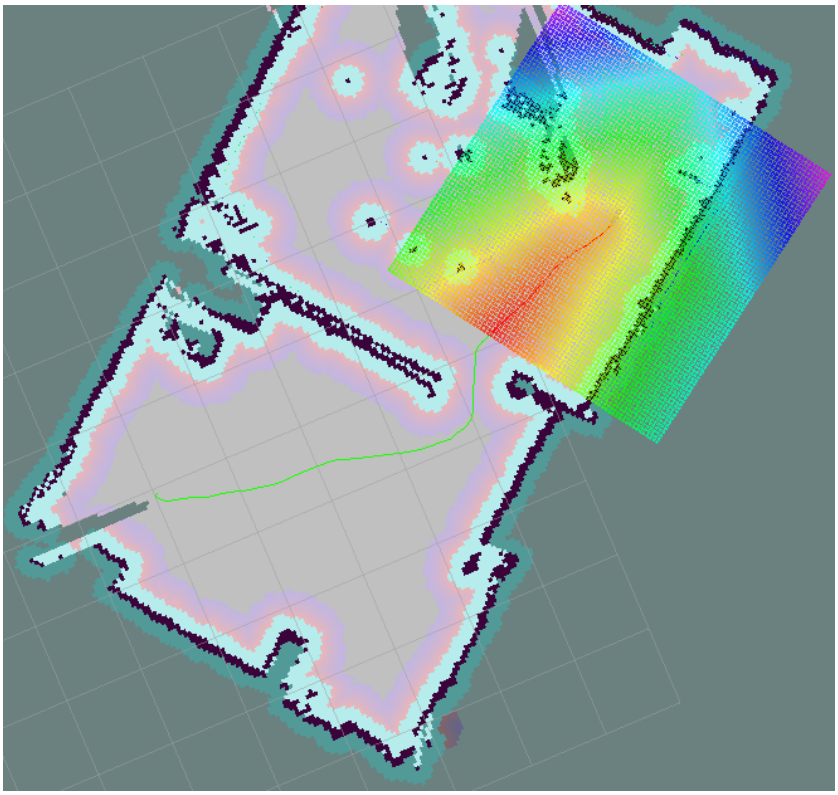

Figura 33. Ejemplo de navegación.

4.3 EXPLICACIÓN DEL SOFTWARE CREADO

Aquí se explica todo el software desarrollado en este TFE, con la intención de que próximos investigadores puedan si fuese necesario hacer modificaciones con conciencia, entendiendo que es lo que hace el programa actual y saber qué cambios hacer según sus necesidades.

4.3.1. ODOM2TF.CPP

Este programa simplemente crea el topic /tf a partir del topic /odom, recuerda que el /odom está siendo publicado por el servicio UDP, y desde LabVIEW no se le asigna el tiempo de ROS, que es muy importante, por tanto, internamente es como si no se publicase la odometría, ya que tiene una variable de tiempo siempre igual a 0. Por eso esta transformación, que de normal es automática, se hará desde este programa.

Se comienza por cargar las librerías necesarias, luego se crea la clase SubscribeAndPublish y se declaran las variables para publicar y subscribirse, además del mensaje que se enviará con el tipo adecuado. La función iniciar, crea una transformación y la mete dentro de nuestra variable de salida output, esto se hace ya que no se puede declarar directamente una transformación dentro del vector transforms[] de nuestra salida (véase Documentación tf2_msgs/TFMessage en docs.ros.org/jade/api/tf2_msgs/html/msg/TFMessage.html).

La función callback será la función que se llame cada vez que reciba un dato por el topic al que se suscribe, en este caso /odom, por tanto, cada vez que se publique un valor de /odom esta función se ejecutará. Lo que hace es, primero llama a la función iniciar, por si aún no se ha creado el vector, y luego cambia los valores de la transformación según las dimensiones del robot y la odometría (fíjese que la transformación es entre los links de la odometría y de la base del robot, se añade además la transformación del láser respecto a la base) además añade el tiempo de ejecución de ROS en el header del topic para que este sincronizado y luego publica el topic actualizado.

En 'private' se crea el nodo y las variables de llamada a publicar y subscribirse, y por último en el 'main' que será el que se ejecute continuamente, se inicia la comunicación con 'roscore', se crea el objeto para publicar conforme se recibe un dato por el topic subscrito y se vuelve a empezar el 'main', en el segundo ciclo como ya está iniciada la comunicación y el objeto, es simplemente como si estuviera en un bucle infinito en el que nada ocurre dentro, sino en el objeto.

Código en el anexo A.

4.3.2 STARTER KIT 1.0 FINAL.VI

Empezando por el HMI que se puede ver en el anexo, se puede ver que se divide en tres partes, la primera más arriba es la configuración e información del VI, se puede configurar la IP del Host de ROS, el botón de parada, leds que indican si se están ejecutando los 3 bucles, 'talker' que es el contador de ciclos del bucle que escribe la velocidad en los motores, 'listener' el contador de ciclos del bucle que envía la odometría, 'iteration duration' indica al parar el VI cuánto tarda el ciclo critico (de lectura del topic y escritura de los motores) y el led se enciende si este tiempo es mayor que 50ms, por último, la corrección es un factor que se le aplica a la lectura de los encoders, por si dan problemas y hay que multiplicar la distancia. Este último factor debería ser 1, una prueba fácil es dejarlo a uno, mover el robot un metro y ver si la lectura de la odometría es un metro también, si no lo es modificar el valor de corrección.

La segunda parte es la relacionada con el robot, en "raw in", se puede ver la cadena de texto que llega desde el servidor UDP tal cual, en su uso normal no hay que fijarse en qué pone, pero si veras si están llegando datos con forme se mueve el mando (si se mueve y no se recibe nada, es decir, se queda en blanco, quiere decir que no están llegando datos). Luego las velocidades de PS4 que llegan por UDP ya traducidas en números (m/s y rad/s respectivamente). Después los controles manuales que también corresponden con las velocidades lineal y angular, igual que las que se verían desde los indicadores del mando. Para que se usen las velocidades del mando o las de control manual (desde el HMI) hay elegir la opción desde el interruptor que hay a continuación, y por último están los valores de la odometría que es calculada y será enviada al servidor UDP, que son las posiciones X, Y, Z y el ángulo de orientación. Z siempre será 0.

En la tercera parte se dedica el espacio a ver los errores que hayan sido sufridos durante la ejecución del VI, solo se muestran al detener el VI.

En el diagrama de bloques también se pueden ver tres bucles, pero no se corresponden con los tres del HMI. El primero establece la comunicación con uno de los puertos y se dedica a escribir la odometría que coge de los indicadores del HMI como si de variables locales se tratase.

El segundo, el más crítico e importante, se encarga de establecer la comunicación con el otro puerto para leer las velocidades, configura el modo de operación entre manual (HMI) o Mando de la PS4, se encarga de escribir las velocidades en los motores, lee los encoders y calcula la odometría mostrando todo por el HMI. Al ser el que hace las cosas más importantes en cuanto a control del robot se refiere, se opta por monitorizar al terminar el tiempo de los bucles y si tardaba más de 50ms, ya que debe responder en el mínimo tiempo posible, y de no ser así saberlo.

El tercer bucle se encarga de parar el VI en estado seguro, es decir, que cuando se pulse STOP, no detener el VI sin más, puesto que el robot podría seguir moviéndose y causar accidentes, si no escribir en los motores velocidades nulas, asegurarse de que es así y solo entonces detener el VI.

CAPÍTULO 5: AYUDA A LA PUESTA EN MARCHA

5.1 COMUNICACIÓN INALÁMBRICA CON LIDAR

(Resuelto) Uno de los fallos más sencillos de solucionar y más problemas dio para llegar a la solución fue este. Resulta que el LIDAR está conectado al módem del DANI directamente por Ethernet, y se estaba conectando el NUC a este módem por Wi-Fi; de esta manera se puede leer sin problemas los datos que el láser publica en la red, pero no en este caso, al parecer dentro de la universidad, al haber tantas redes, es posible que interfirieran de alguna manera, así al ejecutar el comando para iniciar la lectura del láser, se mostraba un mensaje diciendo que el paquete recibido estaba corrupto, o que no era del tamaño que se esperaba. La solución: cambiar la comunicación inalámbrica, por una cableada, así el único problema era el alojar el NUC encima del DANI, que más que un problema era un simple cambio, aunque ya no se podía tener una pantalla conectada durante el movimiento del robot y había que usar SSH o Putty en este caso.

5.2 EL MAPA NO SE CREA

(Resuelto) Uno de los problemas más complejos de solucionar fue que al ejecutar el slam_gmapping el mapa no se creaba al mover el DANI. Al principio enviaba una odometría con valores nulos en todos los campos, al pensar que era de esto, se solucionó modificando el VI, para que la odometría fuese la adecuada al movimiento. Posteriormente, pensando que sería fallo del /tf, ya que la odometría no la cogía directamente el slam_gmapping, se creó un pequeño programa en C++ que se subscribía al topic /odom y publicaba en /tf. El siguiente problema era que no se actualizaba el tiempo de este topic (porque van ligados a una variable de tiempo, para saber en qué momento se produce la transformación, odometría, etc. y este todo sincronizado), por tanto, se solucionó este problema añadiendo un par de líneas a dicho programa y el /tf lo recibía correctamente sin errores ni warnings.

5.3 PROBLEMA CON EL MAPEO, ERRÓNEO POR GIRO

Es posible que mientras se esté mapeando, se quiera desde una misma posición hacer un giro de 360° para mejorar el mapa, pero lo cierto es que, si en el topic /odom no cambia la posición X o la Y, no se actualiza el /tf. Si solo se gira, el mapa va a girar y el mapeo también, por tanto, no tendremos un mapa real.

La solución a este problema se arregla dándole siempre que se quiera girar, aunque sea un giro de 90° porque se esté esquivando un obstáculo, un poco de velocidad lineal. Es decir, no aplicarle nunca una velocidad angular distinta de 0 mientras la velocidad lineal sea 0 también.

5.4 PROBLEMA AL EJECUTAR EL SCRIPT

Un error que puede aparecer al ejecutar el script, es que en uno de los terminales que se abran, más concretamente en uno que se ejecuta un servidor UDP, salga un error y se cierre el programa.

Puede probar dos opciones, la primera y menos engorrosa es, en el terminal de roscore, terminar el proceso con Ctrl+C y después cerrar todas las ventanas de xterm, y volver a ejecutar el script. Si esto no soluciona el problema y el error persiste, reinicie el NUC y vuelva a empezar.

5.5 LIDAR MAL CONFIGURADO

Uno de los problemas puede ser la comunicación con el LIDAR, si esta no responde cuando ejecutamos el script, y no recibimos el topic /scan es porque no se tiene bien configurada la conexión. Hay 2 formas de arreglarlo, cambiar la configuración del Lidar o el launch que se ejecuta.

Para cambiar la configuración hay que hacerlo desde el software SOPAS: se conecta directamente el LIDAR al ordenador vía Ethernet, se entra en la aplicación y se le da a buscar dispositivos. Si solo saliese un dispositivo en vez de dos como en la figura siguiente, se debe hacer un paso intermedio.

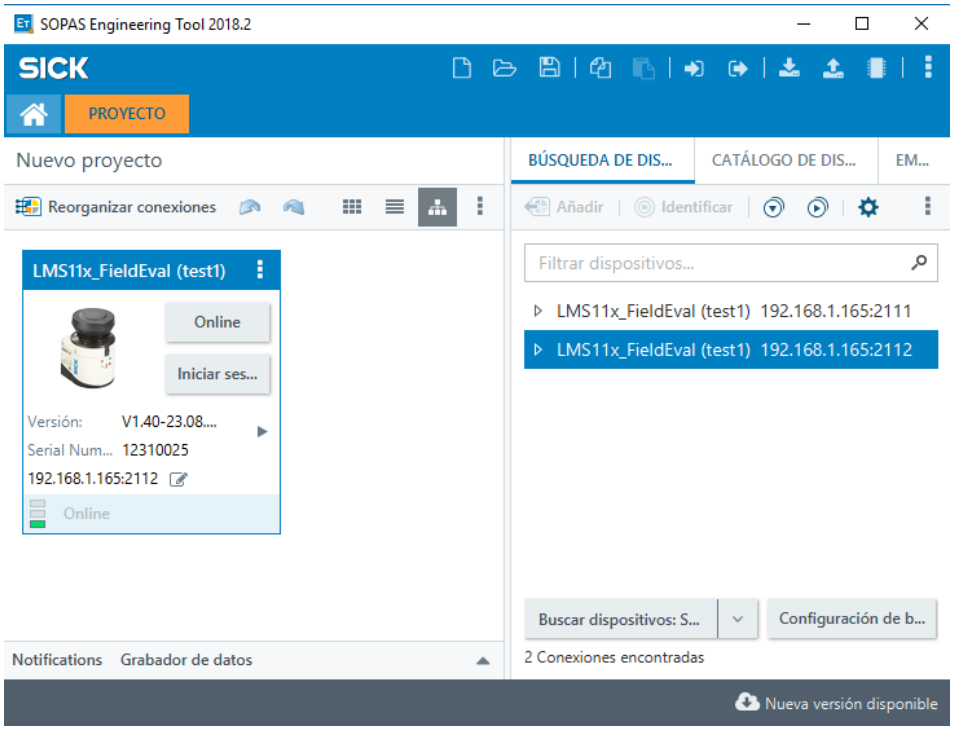

Figura 34. Menú principal SOPAS.

Si fuera el caso de que solo detecta un dispositivo, se debe ir a la configuración del adaptador de red (Inicio>Configuración>Red e Internet>Cambiar opciones del adaptador) en la opción Ethernet con el clic derecho y se selecciona Propiedades, se

vuelve a seleccionar Protocolo de Internet versión 4 (TCP/IPv4) y se clica en Propiedades, Allí se deben poner los valores tal cal están en la figura que hay a continuación.

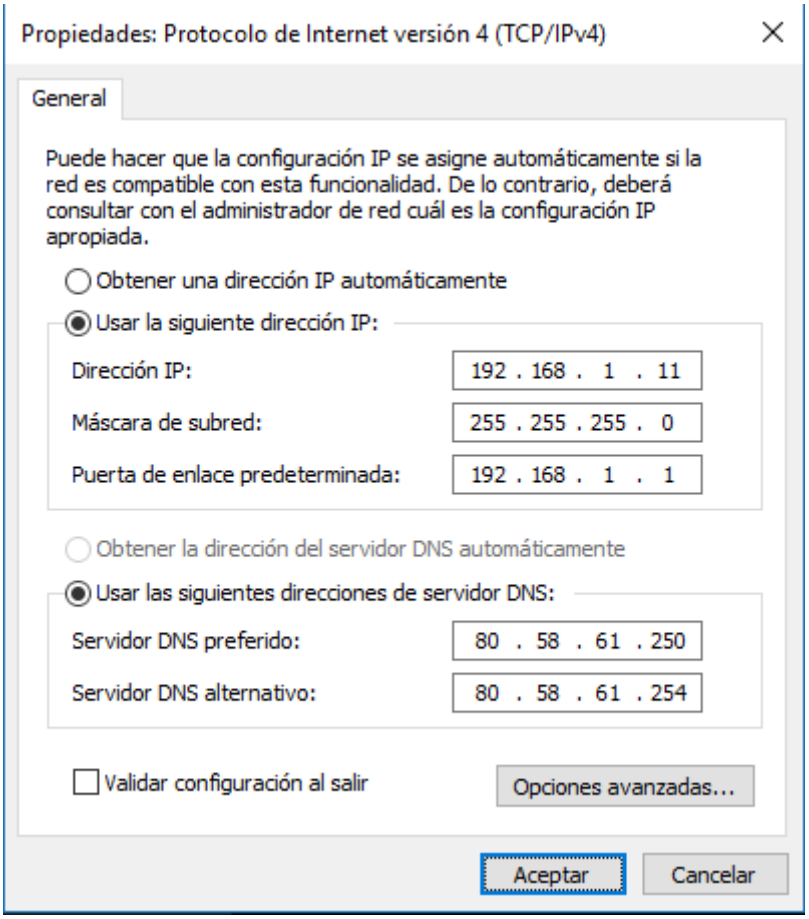

Figura 35. Configuración de red.

Una vez hecho esto, al buscar dispositivos otra vez deberían aparecer dos dispositivos iguales, con un numero de diferencia entre los puertos. Entonces se hace doble clic en cualquiera de ellos, se agregará en la ventana de la derecha un dispositivo, y se le da a Iniciar sesión, se identifica como admin, con contraseña admin (solo para este láser), entonces se le da clic derecho y se pueden abrir las opciones del dispositivo, desde allí se puede cambiar todo acerca de la configuración, hay que asegurarse que la configuración está bien revisándolo todo, además de fijarse en que la IP sea 192.168.1.165, o al menos que los tres primeros dígitos (192.168.1) sean iguales a los del módem para que pueda haber comunicación. Y además se recordará el ultimo dígito para más adelante.

Si no se quiere instalar SOPAS, hay instalar un Network Scanner en un móvil, conectarlo a la red del LIDAR (en este caso Dani_Robot, que su contraseña es “danirobot”) y escanear, saldrán todos los dispositivos que estén conectados al módem y donde se vea que el dispositivo es de la compañía SICK, ese debería ser el LIDAR. Se observa y recuerda la IP de ese dispositivo. En la figura siguiente se puede observar la interfaz de dicha aplicación.

Figura 36. Resultado de escanear la red Dani_Robot.

Una vez se sepa la IP del LIDAR por cualquiera de los dos métodos, se va al directorio de la carpeta LMS1xx-master, que debe estar en catkin_ws/src, en la carpeta launch, el único fichero que hay, LMS1xx.launch se modifica con gedit o cualquier otro editor.

Hay que modificar la segunda línea, cambiando la dirección IP por la que tenga el LIDAR, con esto se consigue que cuando una aplicación vaya a llamar al LIDAR, busque en esa IP y no en otra distinta.

5.6. DUPLICAR EL SERVIDOR UDP

(Instrucciones) Si se está rehaciendo el proyecto en otro NUC, cuando se instale el paquete de rosbridge, solo vendrá un launch para servidor de UDP, pero como para este proyecto se necesitan dos, hay que duplicarlo y hacer algunos cambios.

Una vez instalado el paquete de rosbridge se va al directorio /opt/ros/kinetic/share/rosbridge_server/launch, se duplica el fichero que corresponde con el de UDP, se le pone el mismo nombre pero acabado en 2 (u otro que recordemos), es decir “rosbridge_UDP2.launch” y se modifica en las líneas 2, 16 y 31: el número del puerto que se abrirá, en vez de 9090 se pone el 9091; el nombre del proceso, en vez de rosbridge_udp se pone rosbridge_udp2; y el nombre del nodo, en vez de rosapi se pone rosapi2.

CAPÍTULO 6: ADAPTACIÓN A UN VEHÍCULO NO HOLONÓMICO

Llegados a este punto hay que preguntarse qué modificaciones habría que hacer en este proyecto para conseguir portar el sistema al vehículo real, el INCUBATOR CAR. Surge la principal problemática del cambio de tipología del robot, puesto que se ha trabajado con un vehículo holonómico, y el vehículo es de tipo no-holonómico. El único cambio que esto refleja es en la información de velocidad, y las librerías que se deben usar en ROS, que se hablarán de ellas más adelante. En cuanto a la velocidad que se debe enviar, ya no se basa simplemente en una velocidad lineal de avance y una velocidad angular de giro; en el caso del vehículo no-holonómico se debe transmitir: ángulo de giro de las ruedas delanteras, velocidad y aceleración de dichas ruedas para el cambio de ángulo, y velocidad y aceleración de avance del vehículo.

6.1 CLOUD INCUBATOR CAR

El Cloud Incubator Car es un vehículo comercial modificado, se cogió un Renault Twizy para hacer cambios en el y poder convertirlo en autónomo, estos cambios se dividen en tres partes:

- El sistema de acción: tanto en el freno como en el acelerador fueron añadidos actuadores de tal forma que podían seguir siendo utilizados de forma manual, para la caja se cambios se diseñó un sistema electrónico que permitía elegir entre Estacionamiento, Neutral, Marcha Atrás y Conductor, para que también pudiese seguir siendo operado de forma manual, y para la dirección del vehículo se incorporó un motor dentro del vehículo junto al eje, de esta forma no hay ningún impedimento en usar el vehículo por una persona.
- El sistema control: está basado en una compactRIO 9082, que tiene un procesador Intel i7, y una FPGA Xilinx. De esta forma se consigue tener un control en tiempo real, con flexibilidad para ejecutar algoritmos sofisticados y complejos.
- El sistema de percepción: se divide en dos subsistemas, de corto (SRS) y de largo alcance (LRS).
    - El SRS permite detectar objetos hasta 10 m desde la parte delantera y trasera del vehículo (2D LIDAR) y a unos 3 m del lado derecho e izquierdo del vehículo (cámaras ToF). Los objetos que entran en el anillo de corto alcance supone un riesgo y un peligro inherentes para las acciones del CICar. Por esta razón, los sensores involucrados en el SRS son comandados por la unidad de procesamiento RT.
    - Por otro lado, el LRS la percepción se basa en un LIDAR 3D de alta definición (HDL64SE de Velodyne). Sus 64 rayos láser que giran a 800 rpm y puede detectar objetos a una distancia de hasta 100 m con una precisión de 2 cm. El LIDAR 3D permite establecer trayectorias (cambio de carril, evitación de obstáculos, reducción de velocidad dependiendo de las condiciones de tráfico, etc.), el seguimiento de objetos, la

clasificación de objetos y la predicción de comportamiento de otros conductores.

La información de los dos subsistemas se fusiona y procesa para detectar obstáculos como: Coches, bicicletas, peatones, semáforos, etc. Además, está equipado con una cámara a color en el techo del vehículo, CCD CAM, que se usa para detectar la carretera, los carriles, las vías y las señales de tráfico. Y también lleva incorporado una Unidad de Movimiento Inercial (IMU) y GPS para saber en todo momento cómo y dónde se encuentra.

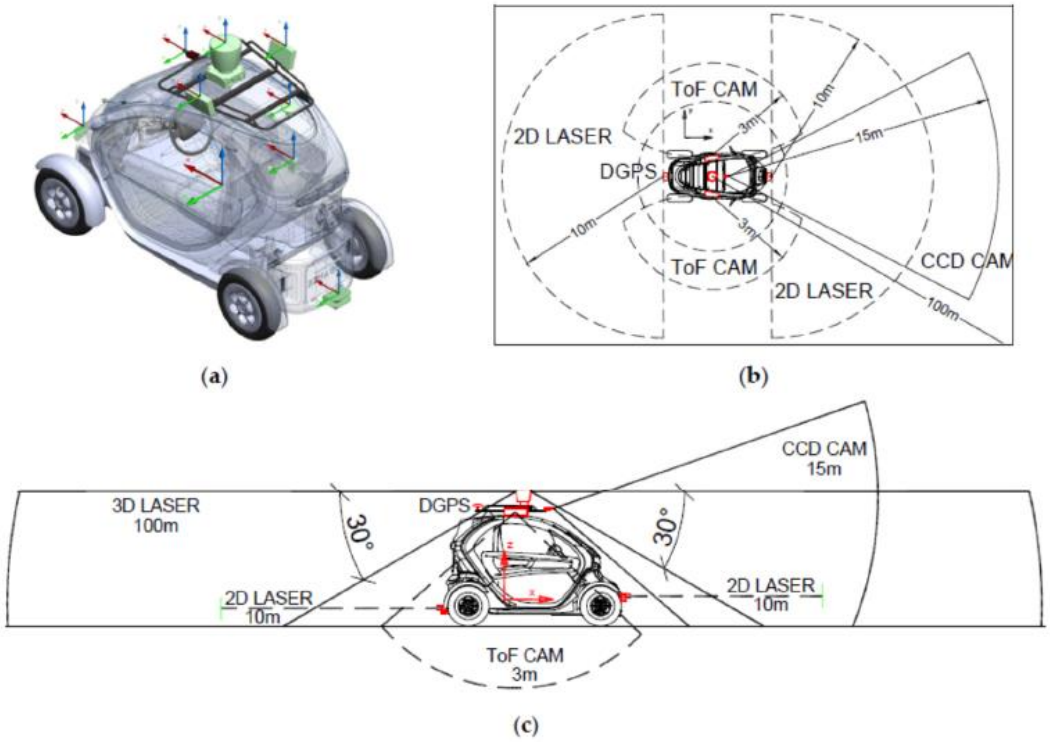

Figura 37. Sensores de percepción del entorno instalados en el CICar.

Figura 38. Cloud Incubator Car (I)

Es por todo lo expuesto que los cambios que habría que hacer son varios, aparte del cambio de tipología diferencial a Ackermann. Por ejemplo, ya no habría que controlar dos velocidades de dos motores, sino posición de acelerador, posición del volante o dirección, marcha del vehículo y posición del freno. Otro ejemplo es que ya no tenemos un láser 2D, sino que tenemos muchos más sensores que habría que configurar, y hacer las transformaciones de posición necesarias para cada uno y tener un mapa conjunto de todo el entorno, no cada sensor por separado.

Para las simulaciones y pruebas, se tomará como punto de partida una situación básica, para posteriormente ir haciendo modificaciones progresivas. Dicha situación se basa en, un vehículo con configuración no-holonómica tipo Ackermann, que se puede controlar el acelerador/velocidad y la dirección, y en la que aún no se poseen sensores perceptivos.

6.2 LIBRERIAS

Las librerías de las que se disponen, aunque de libre uso, son de Robotnik, una empresa Valenciana dedicada a la investigación, fabricación y venta de robots, vehículos y accesorios relacionados con la robótica móvil, la detección, sensorización, etc. Para aplicaciones de conducción autónoma o controlada, ya sea de vehículos, robots paletizadoras, grúas, brazos robóticos, etc.

La librería que se ha utilizado para las simulaciones se llama “rbcar_sim” y es la que contiene todo lo relacionado con la simulación de un vehículo que ellos tienen, el rbcar, pero que es sumamente parecido al INCUBATOR CAR, tiene una configuración Ackermann que es la que interesa a este estudio.

También hay dos librerías adicionales; rbcar_common y robotnik_purepursuit_planner, que están relacionadas con la navegación y el cálculo de trayectorias. No se utilizan aún pues se pretende controlar el simulador de forma manual.

Adicionalmente se utilizan librerías auxiliares como: robotnik_msgs o robotnik_sensors, que contienen las clases y funciones de los tipos de datos y variables que se usan en las otras librerías que hay que estar seguros que están instaladas en el PC donde se harán las simulaciones.

6.3 SIMULACIONES

Para simular el rbcar de robotnik, lo único que hay que hacer es ejecutar:

```
>> roslaunch rbcar_sim_bringup rbcar_complete.launch
```

Ni siquiera hay que ejecutar primero el roscore, porque este launch ya viene preparado para hacerlo automáticamente. Este comando ejecuta Gazebo con el modelo del RBCAR, ejecuta el control de la simulación y ejecuta también el controlador de las velocidades con un joystick (pero que no se usa es este experimento).

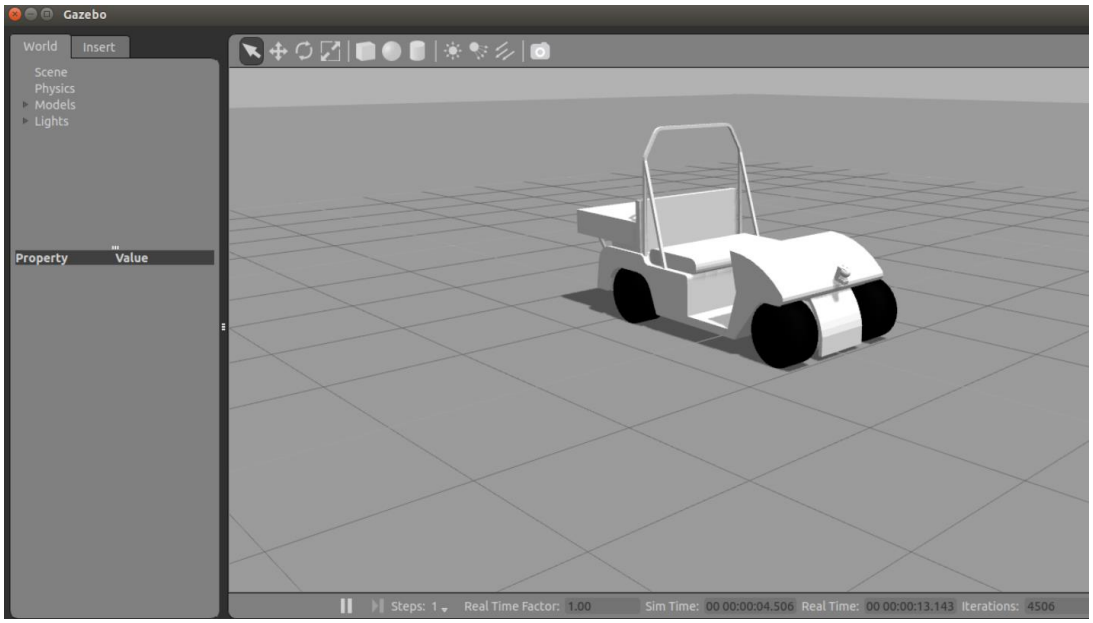

Figura 39. RBCAR simulado en Gazebo.

Cuando se abra el simulador ya estarán todos los topics relacionados con Gazebo y el robot publicados o abiertos, para verlos todos hacer:

```
>> rostopic list
```

Para controlarlo simplemente se hacen dos cosas:

- Primero se ejecuta en un terminal distinto a donde ejecutamos el bringup:

```
>> roslaunch rosbridge_server rosbridge_UDP.launch
```

- Segundo, se averigua que topic es/son los que mueven o controlan el movimiento del rbcar. Y en el VI Gazebo RBCAR.vi, en el diagrama de bloques, se modifica: el String que representa el tipo de dato del topic para que coincidan, añadiendo '%f' en los campos que sean variables, como ya se hacía con el DANI; se modifica también la línea donde se hace el advertise, poniendo el topic correcto y el tipo de dato; se añaden los valores del volante y de la velocidad haciendo el tratamiento adecuado para que el topic lo reconozca y se ejecuta, con esto ya se podría mover y controlar la simulación de Gazebo en ROS desde LabVIEW.

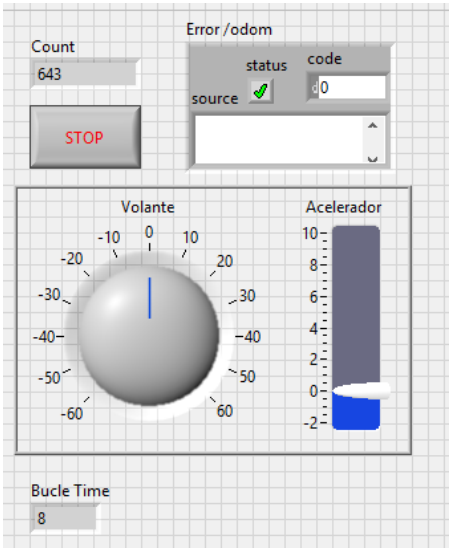

Figura 40. HMI de "Gazebo RBCAR.vi".

6.4 SIMULACIÓN DEL VEHÍCULO CICAR

No se dispone del modelo real del INCUBATOR-CAR para simularlo, pero lo único que necesitaríamos para simular correctamente el movimiento seria las posiciones de las ruedas, con respecto a ellas mismas, las velocidades y aceleraciones máximas que pueden tener y el ángulo de giro máximo de las ruedas de dirección.

Figura 41. Cloud Incubator Car (II)

Con todo esto se podría modificar el fichero del modelo (.yaml) y aunque visualmente no cambiaría, la manera de comportarse en cuanto a movimiento, velocidades y giros sería igual que el vehículo de la UPCT.

En cuanto al coche real, con la investigación que ha conllevado este proyecto se llega a la conclusión de que:

- El vehículo real controlado por una cRIO puede llevar un programa básico de nivel bajo, que controle la escritura de actuadores (posición angular del volante,

posición del acelerador, posición del freno y marcha); que lea sensores dedicados a la odometría, como pueden ser encoders, una IMU, GPS, etc; y que con la misma configuración que en el proyecto, se suscriba a un topic y publique otro (si se mejorase el vi que publica se podría añadirle el tiempo de sincronización, pudiendo sincronizar el tiempo de LabVIEW y el de ROS gracias al topic que se lee). No sería necesario un HMI como el que ya tiene el CICar (figura siguiente), pero si se quieren ver variables de Labview se puede crear un topic propio que publique un String de datos con lo que se quiera ver, y en un terminal del ordenador principal ejecutar:

```
>> rostopic echo <topic_propio>
```

- Y que a nivel alto en un ordenador con un sistema operativo Ubuntu, que haga de ordenador central/principal se desarrolle gracias a ROS todas las aplicaciones que estén relacionadas con navegación, mapeo, cálculo de trayectorias, etc. Este sería el que recibiría la información de todos los sensores de percepción y haría el tratamiento de los datos para crear trayectorias y navegar. Adicionalmente, los sensores del anillo de corto alcance (SRS) se podrían pasar también a la cRIO, (tratados por el ordenador central para que la cRIO haga una lectura más sencilla, o sin tratar) con la intención de tener una medida de seguridad reactiva, y que se detuviese el vehículo si fuese el caso que se detecta un objeto a menos de una distancia de seguridad.

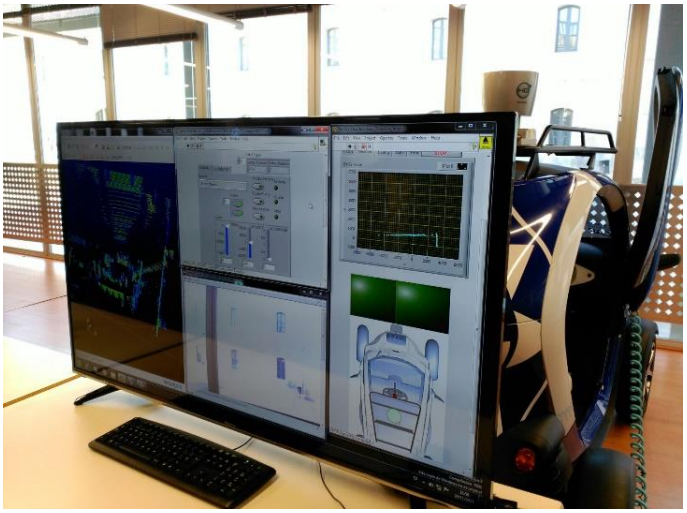

Figura 42. HMI LV Cloud Incubator Car.

CAPÍTULO 7: CONCLUSIONES Y TRABAJOS FUTUROS

El objetivo de este proyecto era conseguir comunicar dos plataformas distintas para poder desarrollar las aplicaciones necesarias que desde ROS controlasen un robot, DANI, que a nivel bajo se controla desde LabVIEW. Posteriormente a la implementación total del DANI igual al TFM [1] con el Turtlebot, hacer las simulaciones e investigaciones necesarias para portar el sistema a un vehículo real, el CICar.

7.1. CONCLUSIONES

En conclusión, para este proyecto y de manera general, se puede afirmar que todos los objetivos marcados al comienzo del mismo han sido cumplidos, haciendo ciertas menciones.

Se reprodujo en una primera instancia el TFM [1] tal cual él lo explico en su proyecto sin ningún problema. Posteriormente se consiguió, después de muchas pruebas, crear un enlace en la comunicación entre dos softwares tan distintos como ROS y LabVIEW. Se trató de controlar el movimiento del DANI con algoritmos de ROS, y tras la resolución de varios problemas en cuanto a la actualización de topics y demás se consiguió mapear un espacio y navegar en él.

Remarcar que el objetivo de investigación de traspaso del sistema a un vehículo real y de cómo hacer las simulaciones fue cumplido, pero el hacer las simulaciones dio problemas. El NUC que es donde se hacían las simulaciones, al intentar reproducir un TFG de la universidad de Valencia [7], se sobrecalentaba y se apagaba. Esto se cree que es debido a que el NUC no tiene tarjeta gráfica dedicada, y como las simulaciones llevan mucha carga en esta parte, se llevaba toda la ejecución al procesador y se sobrecalentaba, llegando a dar un Thermal Warning cada vez que se volvía a encender. Todos los pasos y procedimientos para hacerlo están descritos y mencionados en el proyecto, pero si alguien intentara reproducir este proyecto debería buscar un PC con sistema operativo Ubuntu y en la medida de lo posible sin partición de disco duro, es decir un pc únicamente con Ubuntu.

A mitad de proyecto se barajó la posibilidad de hacer una comparación en el mapeo entre el Turtlebot y el DANI, haciendo el mismo mapa de un espacio con ambos y comparando resultados. Con esto se pretendía comparar en realidad las bases, que es lo único que cambia, ya que el láser y las aplicaciones ejecutadas en el NUC eran las mismas. Se podrían haber comparado los topics /odom que envía cada base y ver la relación con la realidad de cada una, pudiendo calcular el error que comete cada una y ver cuál es mejor.

7.2 EXPERIENCIA PERSONAL

Para desarrollar este proyecto se partía de un conocimiento muy básico y pobre acerca de las plataformas software usadas que provenía de haber cursado ciertas asignaturas en el grado, además del desconocimiento total de cómo utilizar el Turtlebot o el DANI. Con una extensa investigación acerca de cómo realizar cada procedimiento se consiguió

adquirir los conocimientos al menos necesarios para desarrollar el proyecto. Desde cómo cargar un programa en el DANI, hasta como crear mi propio programa en C++ para ROS, pasando por aprender nociones de HTML/XML o conocer los diferentes tipos de datos que se usan en ROS más comúnmente.

Desde mi punto de vista en los grados relacionados con la robótica debería enseñarse de manera más detenida la utilización de ROS que es una plataforma muy versátil y que tanto se utiliza en la industria para desarrollar todo tipo de aplicaciones, y debería enseñarse de manera conjunta a la realización de un proyecto sencillo real, para la mejor adquisición del conocimiento.

No ha sido difícil llegar a conocer todo lo relacionado con ROS de este proyecto, pero sí se ha alargado en el tiempo. Gracias a la comunidad tan amplia que hay detrás de ROS, y que tantos proyectos comparten, a la vez que los mismos fallos que ya les han surgido a otros y se han solucionado; el único problema es el orden, al ser foros de preguntas o el mismo GitHub que cada cosa está en un sitio, la búsqueda de información acerca de una misma cosa o problema se hace eterna, hay decenas de hilos abiertos y cada uno con distintos matices e instrucciones, esto hace que para la búsqueda de información sobre algo el tiempo dedicado sea mucho, aunque eso sí, información no falta.

Durante el tiempo que he estado haciendo este proyecto también me he dado cuenta del potencial que ofrece ROS, y cuan fácil puede llegar a hacer cosas que de primeras eran muy complejas y enrevesadas. A pesar de todo lo que me he tenido que 'pelear' con el programa me llevo un buen recuerdo de el y unos conocimientos muy sólidos.

El cuanto a LabVIEW, ya era un programa que me gustaba y apreciaba, pero después del TFG más aún, hay librerías y paquetes para casi cualquier cosa, y aunque algo no se pueda hacer como quieres seguro que LabVIEW tiene otras mil de como sí hacerlo.

En definitiva, me ha gustado mucho haber cogido este proyecto, ya que ha llevado parte de investigación, parte de desarrollo de aplicaciones software, conocimiento de robots, caso real de manipulación de robots (dos en este caso), estudio de comunicaciones y sobretodo esfuerzo.

7.3 LINEAS FUTURAS DE INVESTIGACIÓN

Teniendo como base este trabajo, habría otros proyectos que podrían desarrollarse, entre los que se pueden comprender:

El más inmediato y de mayor envergadura o al menos el más duradero sería, terminar de simular el Cloud Incubator Car, creando el modelo real del vehículo, conseguir moverlo desde LabVIEW en la simulación, y posteriormente entrar en la programación del vehículo real, junto al equipo ya existente.

También podría basarse un proyecto en la mejora de la base, es decir, crear a partir del DANI, una base como la del Turtlebot. Añadir una IMU para tener una mejor odometría, hacer con el sensor de ultrasonidos un programa de seguridad para evitar colisiones e

implementar un programa que envié toda la información de control y sensorización a través de un cable serial, para que en el NUC: no hubiese que ejecutar la comunicación UDP, que fuese más rápida la comunicación y no tendría que haber ningún módem en medio. El NUC sería el centro de todo, conectándose al laser por Ethernet y la 'base DANI' por USB.

Una aplicación interesante podría ser la de hacer del DANI un robot que pudiese mapear autónomamente, haciendo que recorriese todo un espacio sin la intervención humana y que fuese evitando obstáculos. En definitiva, hacer del DANI un explorador.

Un proyecto, basándose en el anterior, podría ser el crear una red de robots, al menos dos, que se comunicasen entre ellos, pudiesen mapear un espacio en menos tiempo. Sería hacer un 'equipo' de robot que trabajen conjuntamente y no de forma individual, disminuyendo así el tiempo de mapeado, que para grandes espacios puede ser un problema, ya que si un robot te falla mapeando en solitario debería empezar desde cero. En cambio, si tienes una red y uno falla, los demás siguen mapeando.

Otro proyecto podría ser añadir un GPS a la base, y hacer que, al crear un mapa, este esté vinculado a unas coordenadas en el espacio real, y no vinculadas con respecto a un origen ficticio; así el robot podría comenzar a trabajar desde cualquier punto, sin hacer que empiece a ejecutar la navegación desde el origen del mapa que creamos. El robot sabría posicionarse en un entorno autónomamente sin intervención humana.

BIBLIOGRAFÍA

[1] A. J. Pérez Rodríguez, «Desarrollo de sistema de navegación para vehículos autónomos terrestres utilizando ROS,» 2017. [En línea]. Available: http://repositorio.upct.es/handle/10317/6081.

[2] National Instruments, «Introducción a LabVIEW,» [En línea]. Available: http://www.ni.com/getting-started/labview-basics/esa/.

[3] National Instruments, «Aprenda LabVIEW,» [En línea]. Available: https://www.ni.com/academic/students/learn-labview/esa/.

[4] Clearpath Robotics, «ROS for LabVIEW,» [En línea]. Available: http://www.clearpathrobotics.com/assets/guides/ros/ROSforLabVIEW.html.

[5] Clearpath Robotics, «ROS Toolkit Example,» [En línea]. Available: http://files.clearpathrobotics.com/ROS%20Toolkit%20Example.pdf.

[6] ROS, «Rosbridge_suite,» [En línea]. Available: http://wiki.ros.org/rosbridge_suite.

[7] C. A. González Moreno, «Desarrollo de una aplicación para el guiado automático de un vehículo eléctrico,» 2016. [En línea]. Available: https://riunet.upv.es/handle/10251/69123.

[8] Kobuki Team, «Kobuki - User Guide,» [En línea]. Available: https://docs.google.com/document/d/15k7UBnYY_GPmKzQCjzRGCW-4dIP7zl_R_7tWPLM0zKI/edit#heading=h.bkaog13qh9kp.

[9] National Instruments, «NI LabVIEW Robotics Starter Kit - Data Sheet,» [En línea]. Available: http://www.ni.com/datasheet/pdf/en/ds-217.

[10] ROS, «Indice de msgs,» [En línea]. Available: http://docs.ros.org/jade/api/.

[11] Slashdot Media, «Source Forge,» [En línea]. Available: https://sourceforge.net/projects/xming/.

[12] PuTTY.org, «Download PuTTY,» [En línea]. Available: https://www.putty.org/.

[13] SICK, «SICK Sensor Intelligence,» [En línea]. Available: https://www.sick.com/es/es/sopas-engineering-tool-2018/p/p367244.

[14] J. J. Sanz, «Métodos para redireccionar X11 de un servidor remoto UNIX/Linux,» [En línea]. Available: https://www.elarraydejota.com/metodos-para-redireccionar-x11-de-un-servidor-remoto-unixlinux/.

[15] ROS, «Creating a Package,» [En línea]. Available: http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

[16] ROS, «Writing a Simple Publisher and Subscriber (C++),» [En línea]. Available: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29.

[17] ROS, «Tutoriales de Navigaion,» [En línea]. Available: http://wiki.ros.org/navigation/Tutorials.

[18] ROS, «Turoriales de tf,» [En línea]. Available: http://wiki.ros.org/tf/Tutorials.

[19] Gazebo, «URDF in Gazebo,» [En línea]. Available: http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[20] Robotnik, «rbcar_sim,» [En línea]. Available: https://github.com/RobotnikAutomation/rbcar_sim.

[21] Robotnik, «robotnik_msgs,» [En línea]. Available: https://github.com/RobotnikAutomation/robotnik_msgs.

[22] Robotnik, «rbcar_common,» [En línea]. Available: https://github.com/RobotnikAutomation/rbcar_common.

[23] Robotnik, «robotnik_purepursuit_planner,» [En línea]. Available: https://github.com/RobotnikAutomation/robotnik_purepursuit_planner.

[24] Robotnik, «robotnik_sensors,» [En línea]. Available: https://github.com/RobotnikAutomation/robotnik_sensors.

[25] ROS, «ROS Answers,» [En línea]. Available: answers.ros.org\questions\.

[26] ROS, «move_base,» [En línea]. Available: http://wiki.ros.org/move_base.

[27] ROS, «Navigation,» [En línea]. Available: http://wiki.ros.org/navigation?distro=kinetic.

ANEXOS

ANEXO A: odom2tf.cpp

```
#include <ros/ros.h>

#include "geometry_msgs/TransformStamped.h"

#include "nav_msgs/Odometry.h"

#include "tf2_msgs/TFMessage.h"


class SubscribeAndPublish

{

  public:

    tf2_msgs::TFMessage output;

    SubscribeAndPublish()

    {

      pub_ =
→nuevo_nodo_tf_.advertise<tf2_msgs::TFMessage>("/tf", 100);

      sub_ = nuevo_nodo_tf_.subscribe("/odom", 100,
→&SubscribeAndPublish::callback, this);

    }


  void iniciar()

  {

    geometry_msgs::TransformStamped tr;

    if(output.transforms.size()<=0){

      output.transforms.push_back(tr);

    }

  }
```

```
  void callback(const nav_msgs::Odometry& input)

  {

    SubscribeAndPublish::iniciar();

    output.transforms[0].header.seq = input.header.seq;

    output.transforms[0].header.frame_id = "odom";

    output.transforms[0].header.stamp.sec =
→ros::Time::now().toSec();

    output.transforms[0].header.stamp.nsec =
→ros::Time::now().toNSec();

    output.transforms[0].child_frame_id = "base_footprint";

    output.transforms[0].transform.translation.x =
→input.pose.pose.position.y;

    output.transforms[0].transform.translation.y =
→input.pose.pose.position.x-0.036;

    output.transforms[0].transform.translation.z =
→input.pose.pose.position.z-0.02;

output.transforms[0].transform.rotation.z =
→input.pose.pose.orientation.z;

    output.transforms[0].transform.rotation.w = 1;


    pub_.publish(output);

  }

private:

  ros::NodeHandle nuevo_nodo_tf_;

  ros::Publisher pub_;

  ros::Subscriber sub_;


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

```
{
  ros::init(argc, argv, "subscribe_and_publish");
  SubscribeAndPublish SAPObject;
  ros::spin();
  return 0;
}
```

ANEXO B: Starter Kit 1.0 Final.vi

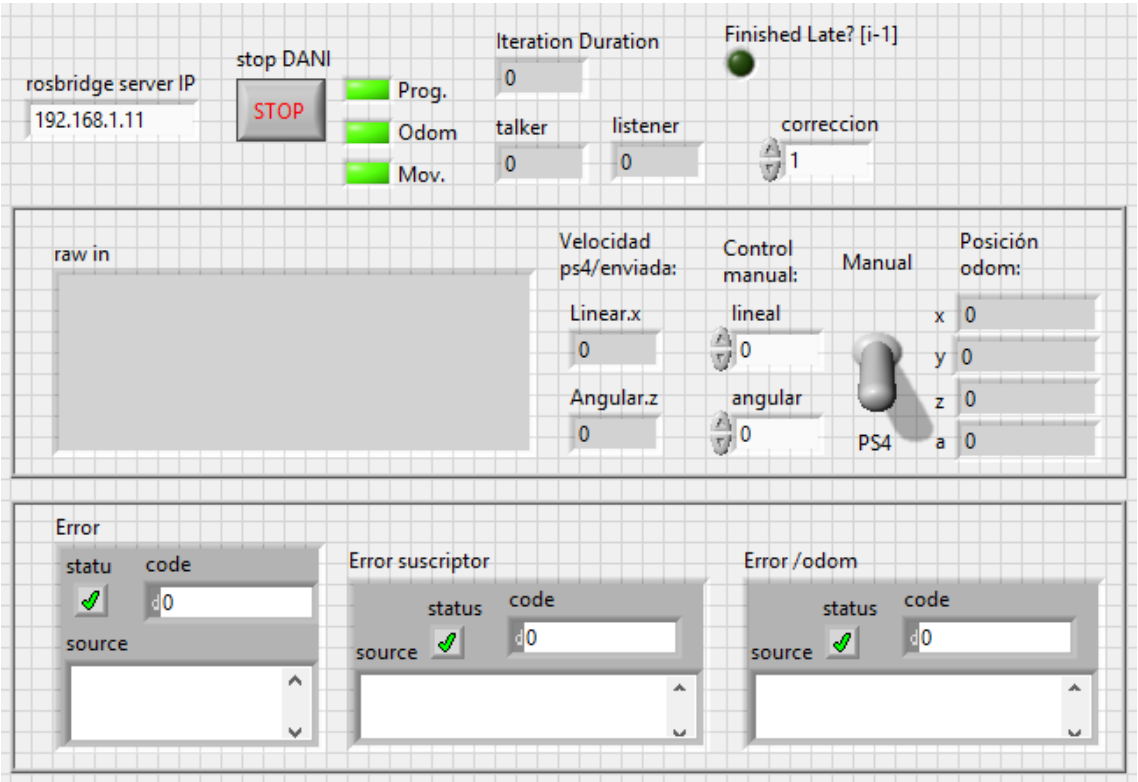

Figura 43.HMI Starter Kit 1.0 Final.

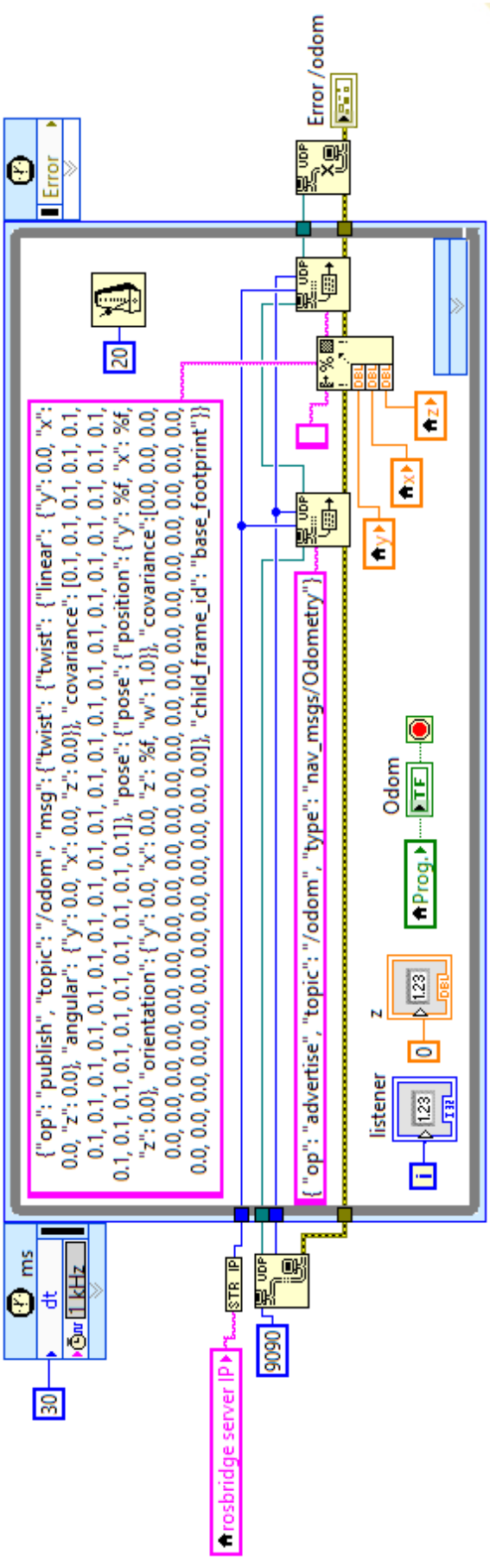

Figura 44. Diagrama de bloques (I) Starter Kit 1.0 Final.

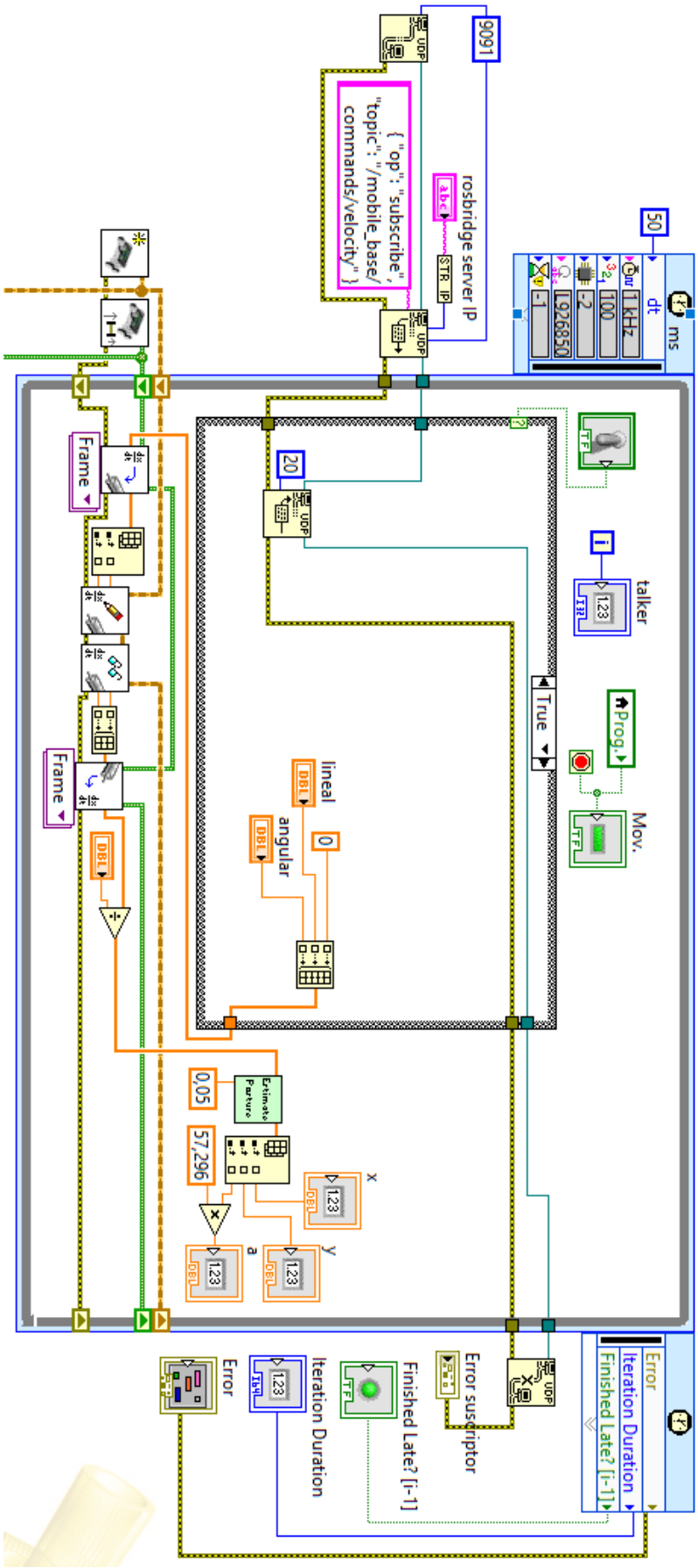

Figura 45. Diagrama de bloques (II) Starter Kit 1.0 FInal.

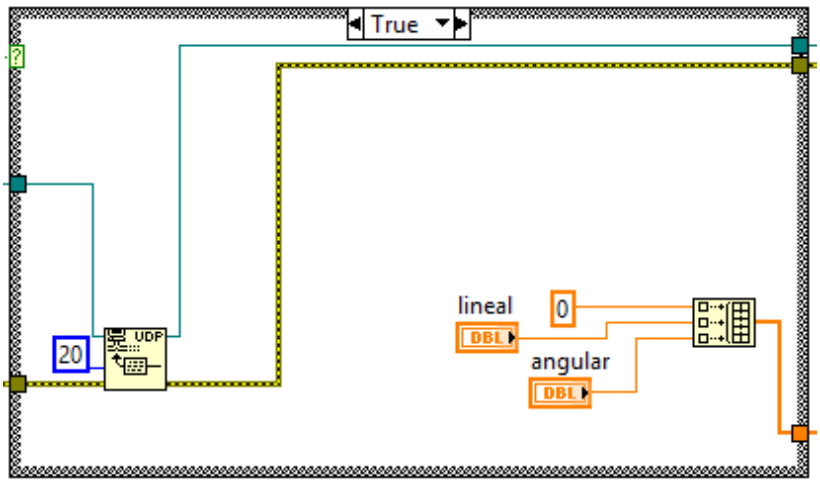

Figura 46. Caso True, DB(II) Starter Kit 1.0 Final.

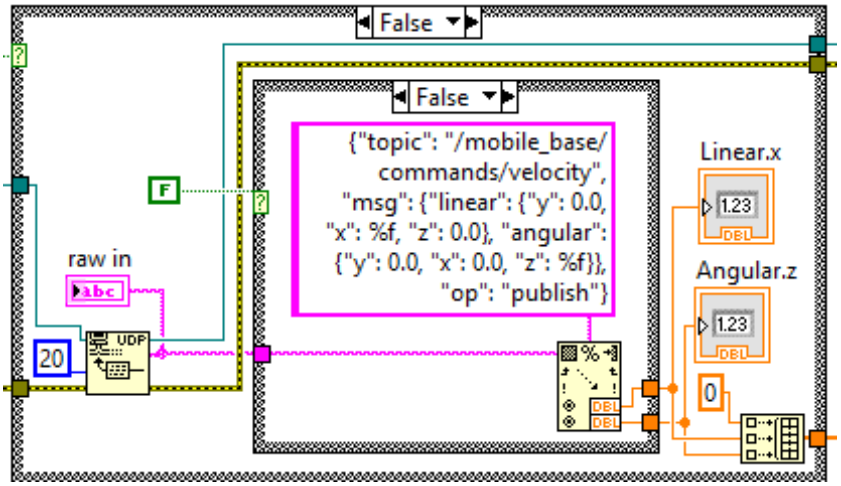

Figura 47. Caso False y False, DB(II) Starter Kit 1.0 Final.

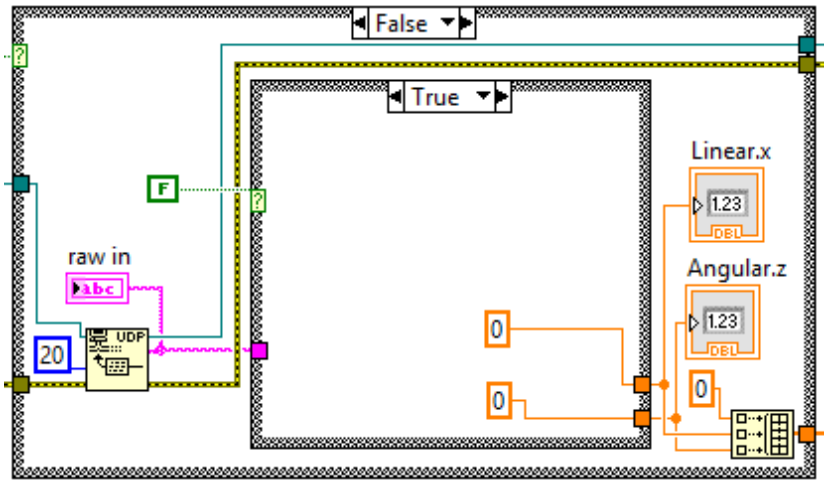

Figura 48. Caso False y True, DB(II) Starter Kit 1.0 Final.

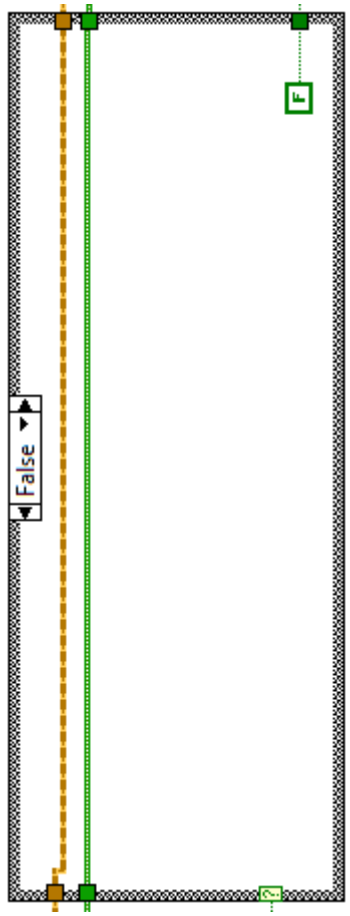

Figura 49. Caso False, DB(III) Starter Kit 1.0 Final.

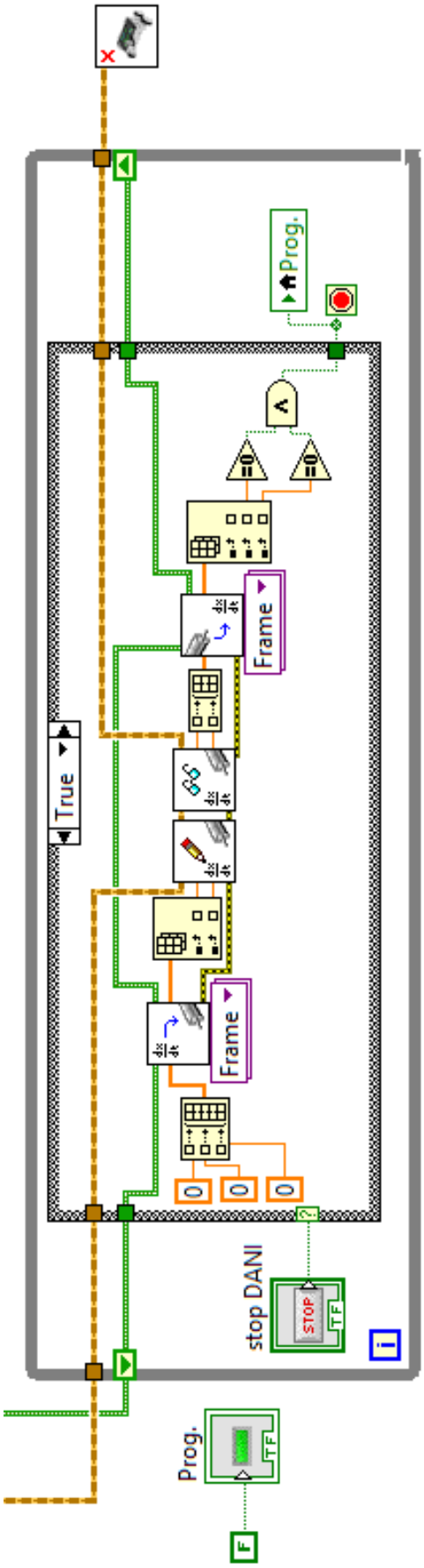

Figura 50. Diagrama de bloques (III) Starter Kit 1.0 Final.

ANEXO C: Gazebo RBCAR.vi

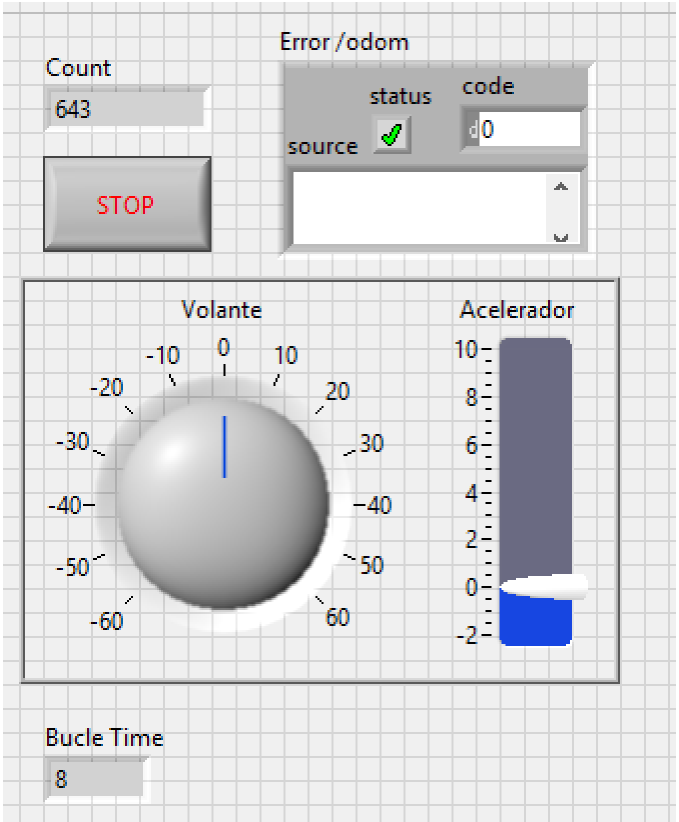

Figura 51. HMI Gazebo RBCAR.

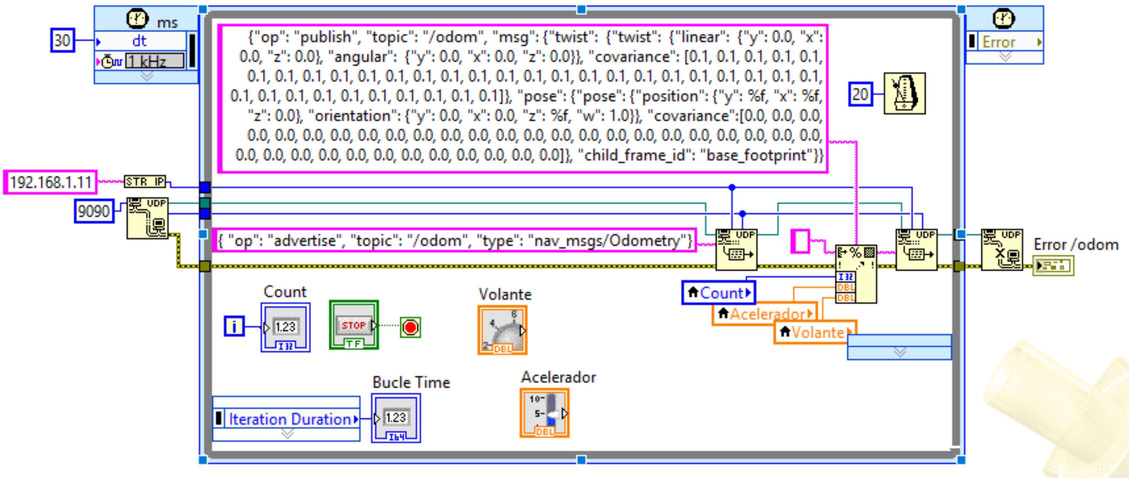

Figura 52. Diagrama de bloques Gazebo RBCAR