Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA EN TECNOLOGÍA INDUSTRIAL

TRABAJO FIN DE GRADO

CONTROL Y SENSORIZACIÓN DEL BRAZO MANIPULADOR WIDOWX SOBRE UNA PLATAFORMA TURTLEBOT 2

Alumno: Cuadra, Gomez, Julen

Director (1): Casquero, Oyarzabal, Oscar

Director (2): Orive, Revillas, Darío

Curso: 2018-2019

Fecha: Bilbao, 21, 06, 2019

Resumen:

En este trabajo fin de grado se desarrolla el control de un brazo robótico manipulador WidowX y su sensorización, todo ello montado sobre una plataforma TurtleBot 2. Este trabajo pretende integrar las funciones propias del brazo con las de una base móvil para ofrecer servicios de transporte y manipulación en un sistema de fabricación flexible. Para entender este conjunto se describen todos los equipos que lo componen. El control de todos estos dispositivos se realiza desde ROS (Robot Operating System) y se desarrolla la interacción de los equipos en este entorno.

Palabras clave: manipulador, WidowX, TurtleBot 2, fabricación flexible, ROS

Abstract:

Throughout this document, the control of a WidowX robotic arm and its sensorization are developed, mounted on top of a TurtleBot 2 structure. The aim of this project is to add the capabilities this robotic arm provides to a mobile base part of a flexible manufacturing system. Every piece of hardware is described in order to grasp how everything interacts. The control of these devices is conducted using ROS (Robot Operating System). ROS' working structure and how it manages the devices' control are explained.

Keywords: robotic arm, WidowX, TurtleBot 2, flexible manufacturing, ROS

Laburpena:

Gradu amaierako lan honetan WidowX beso robotiko manipulatzailearen kontrola eta honen gainean zenbait sentsoreen kokapena garatzen dira, TurtleBot 2 robot mugikorraren gainean antolatua. Lan honek besoak ahalbidetzen dituen ekintzak ematen dizkio fabrikazio sistema malguko robot bati. Multzo hau osatzen duten gailu guztiak azaltzen dira. Gailu hauek kontrolatzeko ROS (Robot Operating System) erabiltzen da eta ingurune honetan nola komunikatzen diren azaltzen da.

Gako-hitzak: beso robotikoa, WidowX, TurtleBot 2, fabrikazio malgua, ROS

Índice de contenido

Índice de Figuras

Índice de tablas

1 - Introducción

La sustitución de trabajadores humanos por máquinas automatizadas se viene dando desde hace varias décadas. Trabajos que destacan bien por ser tareas repetitivas o por ser tareas que requieren una alta precisión. La automatización de este tipo de procesos busca, entre otras cosas, abaratar los costes de producción, conseguir que los tiempos de producción se reduzcan o que la producción tenga una alta repetibilidad. Es en estos aspectos donde los robots altamente automatizados cobran fuerza. Las líneas de fabricación están repletas de robots que buscan cumplir estos requisitos, ya sean robots que manipulan material, robots que realizan soldaduras, máquinas de CNC y otras muchas más.

Además de estas ventajas físicas y temporales, la incorporación de robots en las líneas de producción ha posibilitado el control de toda la línea desde un puesto centralizado. Para ello, es necesario colocar los sensores adecuados en los robots y gestionar de manera eficiente todos estos datos y órdenes de mando que se les envían.

Todos estos requerimientos y posibilidades han derivado en lo que se conoce como sistemas de fabricación flexible. Un sistema de fabricación flexible es un grupo de estaciones interconectadas que permiten realizar series de piezas diferentes según los requerimientos de cada cliente. Cada una de estas estaciones realiza operaciones diferentes y todas ellas están controladas por un ordenador o grupo de ordenadores desde donde se puede controlar toda la línea de forma centralizada. Tanto los sistemas de alimentación, como los sistemas de transporte están también automatizados. Estos sistemas dotan a la empresa que los utilice de tantas posibilidades de producción como necesidades tengan los clientes.

Figura 1.1 Ejemplo de Sistema de Fabricación

2 - Contexto

Como se observa en la Figura 1.1, el nexo de unión de la línea de fabricación es la cinta transportadora. Aunque se puedan realizar diferentes configuraciones, la cinta en mayor o menor medida, restringe la posición de las distintas estaciones dentro de la línea. Las estaciones que se suceden tienen que estar físicamente colocadas en una posición posterior en la cinta.

Es aquí donde los robots móviles cobran una especial relevancia, es decir, bases que disponen de movimientos en todas direcciones y están equipadas con brazos robóticos, plataformas para realizar transportes u otras herramientas. De esta forma, pueden moverse con libertad por un espacio industrial. Además, gracias a las diferentes herramientas con las que pueden estar equipados, pueden realizar una gran variedad de operaciones en el lugar en el que estén en ese momento.

Estos robots permiten eliminar la cinta transportadora que une todas las estaciones (Figura 2.1). De esta forma, podemos redistribuir las estaciones eliminando la restricción de secuencialidad que impone la cinta, enfocándonos en otros aspectos como los tiempos de producción o realizar operaciones en secuencias diferentes.

Al igual que el resto de los elementos de la línea de fabricación, estos robots móviles están controlados desde los puntos de control centralizados. En este caso, las comunicaciones se realizan de forma inalámbrica eliminando restricciones impuestas por los cables. Además, se pueden tener numerosas unidades móviles actuando a la vez, cada una realizando una operación diferente y comunicándose entre ellas para realizar estas operaciones según cual minimice parámetros como la distancia o el tiempo.

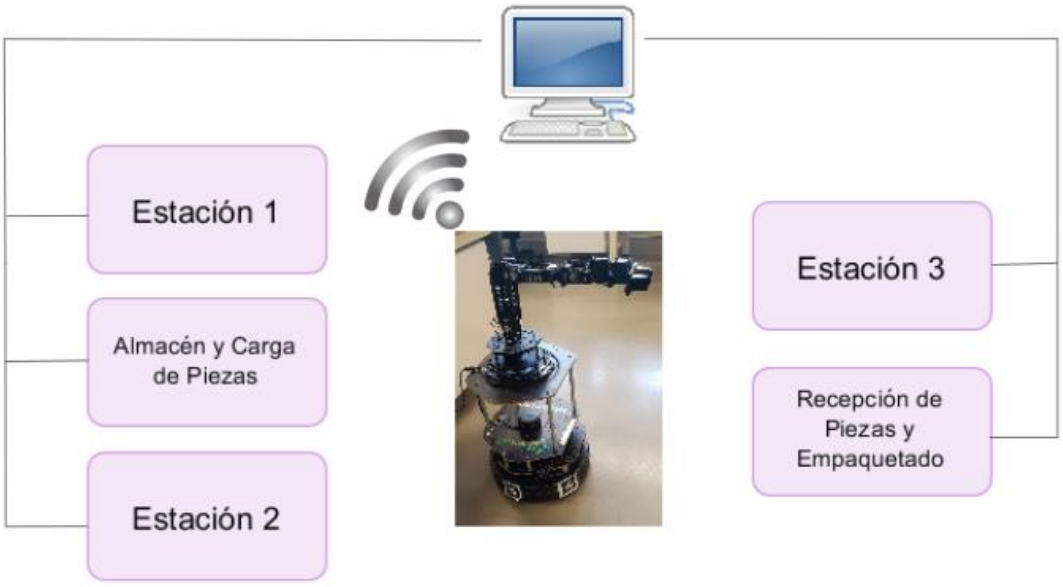


Figura 2.1 Sistema de Fabricación Flexible con Cinta sustituida por Manipulador Móvil

3 - Alcance y Objetivos

Este TFG forma parte de otro proyecto, realizado con anterioridad [2], que tenía como objetivo la creación de servicios robóticos de alto nivel para atender las peticiones de abastecimiento de materia prima y transporte de material entre máquinas en el marco de un sistema de fabricación flexible coordinado mediante un sistema multi-agente.

Concretamente, este trabajo tiene como objetivo dotar de funcionalidad a dichos servicios robóticos mediante el diseño y desarrollo de nodos funcionales en ROS para sensorizar y controlar el brazo manipulador WidowX sobre una plataforma TurtleBot 2.

Para la consecución de este objetivo principal hay que cumplir los siguientes objetivos secundarios:

1. Controlar cada una de las articulaciones del brazo individualmente.

2. Controlar la velocidad de movimiento del brazo.

3. Conseguir que el brazo realice operaciones diferentes en función de la información proporcionada por un sensor acoplado a la pinza del brazo.

4 - Beneficios que aporta el trabajo

El desarrollo de un manipulador móvil como este aporta beneficios en diferentes áreas, entre otros, aporta beneficios sociales, beneficios logísticos y beneficios económicos.

- Como beneficios sociales podemos destacar que, si bien en un primer momento la automatización de los procesos causa la desaparición de puestos de trabajo ocupados por trabajadores humanos, los puestos de trabajo que genera derivados de, entre otros, el mantenimiento, configuración y control permanente de los equipos, son también numerosos [7]. Además, la incorporación de los sistemas de fabricación flexible, permite al usuario una capacidad de personalización del producto solicitado sin precedentes, debido, en parte, a los manipuladores móviles que permiten minimizar los tiempos de traslado y manipulación de los materiales y productos durante la fabricación.

- En cuanto a los beneficios logísticos, estos quedan justificados con la eliminación de la cinta transportadora en la línea de producción. Se evitan tiempos de traslado innecesarios entre estaciones de trabajo. El tener todos estos manipuladores controlados desde un puesto centralizado, permite, además, un control exhaustivo de las trayectorias que éstos realizan, pudiendo así, optimizar tanto los tiempos de los traslados como los tiempos muertos entre operaciones.

- Los manipuladores móviles como este aportan beneficios económicos al eliminar los costes fijos de la cinta transportadora y toda la estructura que lleva consigo. Si bien los manipuladores añaden un coste relativamente alto, la posibilidad que otorgan a la empresa de reducir las piezas en un lote o personalizar la oferta, reduce el impacto económico de su compra. Por otro lado, una avería en la cinta transportadora supone el paro completo de la producción ya que todo el flujo de material se basa en ella. Sin embargo, la existencia de múltiples manipuladores móviles dentro de la flota de una misma fábrica hace posible que, ante la avería de uno de ellos, la sustitución sea inmediata ya que otro manipulador pasaría a ocupar su lugar. Además, el hecho de que el control esté descentralizado, permite que ante una avería, el personal sea notificado de inmediato y puedan comenzar a repararla.

5 - Descripción de requerimientos

Para poder realizar este trabajo de fin de grado, se han impuesto una serie de requerimientos. Si bien ciertas funcionalidades y elementos se han decidido una vez comenzada la ejecución del proyecto debido a restricciones que se ha ido encontrando el equipo de trabajo, otros aspectos se encontraban especificados en el planteamiento inicial del trabajo.

- El planteamiento de este trabajo y del proyecto en sí se basa en ROS. ROS es un software libre que busca aglutinar el control de diferentes robots de una forma sencilla y accesible. ROS elimina las barreras impuestas por las interacciones entre softwares de diferentes fabricantes y permite el control de dispositivos de empresas diferentes en una unidad centralizada como se verá más adelante.

- El brazo robótico que se va a utilizar es el WidowX Mark II, explicado más en detalle en la sección de diseño, en el apartado hardware. Este brazo está pensado como una herramienta para trabajos de investigación y pruebas de concepto. Su sencilla implementación en el entorno de ROS permite una puesta a punto fácil para facilitar la realización de prototipos.

- La base de este manipulador móvil será un robot Kobuki. Al conjunto del Kobuki y la estructura que va sobre él se la conoce o comercializa como TurtleBot 2. Es una estructura de hardware libre que está pensada para la implementación de sensores de todo tipo (o en nuestro caso manipuladores como el WidowX). Al igual que el brazo WidowX, esta base está ideada pensando en su implementación en ROS y su uso para investigación, por lo que facilita su implementación en el proyecto.

- Al implementar estas funcionalidades en un sistema de fabricación flexible coordinado por un sistema multi-agente se busca separar dichas funcionalidades en módulos independientes. De esta forma, se requiere la implementación de estas funciones en nodos de ROS para instanciar o invocar estos nodos como servicios de forma individual.

- Como se va a realizar un proyecto con una base móvil (Kobuki) y un brazo robótico (WidowX), es importante que las funcionalidades de ambos interactúen entre sí. Los equipos tienen que poder comunicarse e intercambiar información entre ambos a través de ROS.

- Las acciones recogidas en los requerimientos funcionales son: la aproximación del brazo a una pieza, la toma de decisiones en función de la información del sensor y la correspondiente actuación según la información que se haya procesado.

6 - Análisis de alternativas

Las primeras alternativas que se presentan a la hora de plantear este proyecto son las diferentes versiones de ROS que podemos encontrar en la web para instalar. Las tres versiones de ROS que se van a analizar son, ROS Kinetic (soporte hasta abril 2021), ROS Lunar (soporte hasta mayo 2019) y ROS Melodic (soporte hasta mayo 2023), ya que son las únicas versiones que tienen soporte a fecha de inicio de este trabajo. Las puntuaciones del análisis de alternativas están basadas en una escala ordinal 1(Muy malo)-5(Muy bueno). Los criterios que se van a utilizar para saber cuál es la versión más conveniente para la realización de este trabajo son:

- **Soporte:** El tiempo que queda desde el planteamiento del proyecto hasta que deje de darse soporte a la versión correspondiente.

- **Robustez Código:** Al ser un software libre, el código que podemos encontrar online ha sido, en muchos casos, desarrollado por otros usuarios. Si bien muchos de ellos son desarrolladores experimentados o investigadores expertos en el área, es posible que el código que encontremos tenga errores, no sea compatible con nuestro equipo o no haya sido revisado por otros usuarios. Por tanto, es más conveniente que la versión de ROS lleve mayor tiempo disponible.

- **Paquetes disponibles:** En ROS se publican versiones anualmente; las publicadas en años pares tienen un soporte de 5 años; las publicadas en impares tan solo 2. Como es obvio, la mayoría de usuarios desarrollan aplicaciones para las versiones más longevas ya que van a ser soportadas durante más tiempo.

Tabla 6.1 Análisis de alternativas de las versiones de ROS

	Soporte 50%	Robustez Código 20 %	Paquetes Disponibles 30%	Resultado
	5	2	2	3,5
	1	4	2	1,9
	3	4	5	3,8

El siguiente análisis que se va a realizar es relativo al lenguaje de programación a utilizar para programar los distintos nodos de ROS e interactuar con ROS en general. Si bien ROS tiene librerías de Lisp, Java, C++ y Python, los lenguajes mejor soportados y más ampliamente usados para desarrollar aplicaciones en ROS son C++ y Python por lo que el análisis se centrará en ellos. Las puntuaciones del análisis de alternativas están basadas en la siguiente escala ordinal 1(Muy malo)-5(Muy bueno). Los criterios para determinar que lenguaje es más conveniente son los siguientes:

- **Prototipado:** A la hora de realizar un trabajo como éste, es importante la capacidad de pasar de manera rápida de una idea a código. Hay que realizar numerosos prototipos para probar diferentes funcionalidades y posteriormente integrarlas.

- **Robustez:** La capacidad de un lenguaje de no ser ambiguo, de realizar siempre la misma función y ser altamente repetitivo. Si un lenguaje no es robusto pueden darse comportamientos inesperados, sin embargo, es más fácil de entender y utilizar.

- **Sintaxis:** La sintaxis del lenguaje afecta directamente a la facilidad con la que este se interpreta por el usuario. En este caso interesa que la sintaxis sea sencilla ya que el objetivo del trabajo no es realizar un código impecable sino llevar a cabo una prueba de concepto.

Tabla 6.2 Análisis de alternativas del lenguaje de programación

	Prototipado 60%	Robustez 10%	Sintaxis 30%	Resultado
Python	4	2	4	3,8
C++	3	5	3	3,2

7 - Descripción de la solución propuesta

Teniendo en cuenta los objetivos planteados al inicio del trabajo fin de grado y los requerimientos dados, la solución propuesta para abordar esta situación es la siguiente:

Desde el punto de vista de hardware, se va a describir cuales son los equipos que interactúan y mediante que conexiones físicas lo van a hacer. El núcleo de esta estructura es el ordenador de placa reducida (SCB, Simple Computer Board) Odroid-XU4, la cual se va a encargar de gestionar todos los elementos. Por un lado, se conectará a la base móvil Kobuki mediante una conexión USB. Lo mismo para el brazo robótico WidowX, el controlador ArbotiX-M que controla el brazo se conectará a la placa Odroid mediante un cable FTDI-USB. En un soporte impreso en una impresora 3D se colocará un sensor de temperatura para analizar la pieza. Este sensor será gestionado por una placa Arduino Nano y esta, a su vez, envía los datos a un módulo XBee mediante un circuito impreso. El módulo XBee envía inalámbricamente los datos y estos son recibidos por un segundo módulo XBee que, a su vez, envía los datos a su correspondiente placa Arduino Nano. Esta segunda Arduino estará conectada a la computadora Odroid mediante un cable USB. En este caso, la interfaz HMI estará constituida por un ordenador de sobremesa con el que nos conectaremos a la placa Odroid utilizando el protocolo SSH. Toda esta estructura queda resumida gráficamente en la Figura 7.1.

Atendiendo a la solución desde la perspectiva del software, la placa Odroid va a ser aquí también el núcleo de toda la computación. En ella se va a correr la distribución de Linux Ubuntu 16.04 LTS. Esto permite utilizar la versión de ROS Kinetic, la más conveniente según el análisis de alternativas realizado en un apartado anterior. En el brazo robótico WidowX, más concretamente en la controladora ArbotiX-M que lo gestiona, se va a instalar un driver que permite incorporar esta gestión del brazo en ROS. Lo mismo se va a realizar en la base móvil Kobuki; un driver va a hacer posible incorporar toda la gestión de los sensores y actuadores de la base en ROS. Para la utilización del sensor de temperatura, habrá que escribir los programas correspondientes mediante el Arduino IDE para la gestión del sensor, la transmisión de todos los datos a través de los módulos XBee y para incorporarlos en la placa Odroid. Toda esta estructura software queda reflejada en la Figura 7.1.

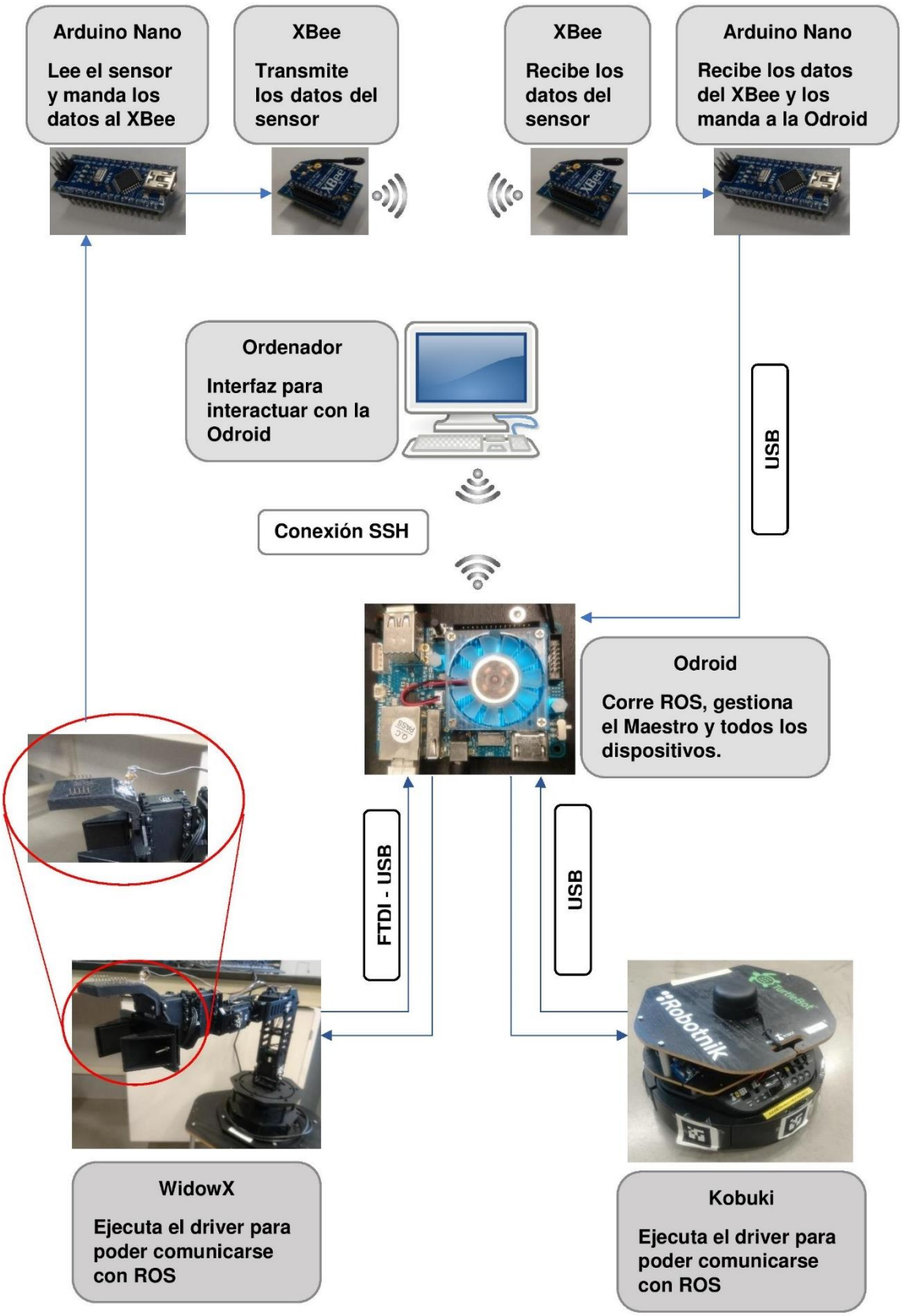


Figura 7.1 Estructura de comunicación de Hardware y Software

8 - Diseño

Para analizar cuál es el diseño final de la solución, primero se va a exponer el diseño desde el punto de vista de hardware, describiendo la interacción entre todos los elementos. Después se detallan cada uno de los elementos que se utilizan. A continuación, se detalla el software utilizado. Finalmente, se describe la interacción de los diferentes elementos de mayor a menor nivel de abstracción.

8.1- Diseño Hardware

Para el desarrollo de este trabajo fin de grado se han utilizado dos equipos principales: el brazo robótico WidowX Mark II y la plataforma móvil Kobuki. Además, se ha utilizado un sensor de temperatura MLX90614. La implementación de este sensor en ROS se ha realizado con la ayuda de dos placas Arduino Nano y dos módulos XBee para transmitir los datos de forma wifi.

- **Brazo robótico WidowX Mark II:** Este brazo, comercializado por la empresa Trossen Robotics, tiene tres partes bien diferenciadas: la estructura, compuesta por partes de aluminio y partes de metacrilato, lo que le otorga ligereza y rigidez; 6 servomotores Dynamixel de la empresa Robotis, servomotores punteros en el sector que permiten un extraordinario control; un controlador ArbotiX-M que se encarga de traducir las ordenes transmitidas desde ROS en consignas para los diferentes servomotores, además de otra serie de interacciones entre el brazo y ROS. A continuación, se explica más en detalle cada parte:

    - **Estructura:** La estructura se compone de dos partes diferenciadas: la base y la estructura propia del brazo (Figura 8.1). La base le otorga al brazo la estabilidad necesaria para moverse con libertad sin perder el equilibrio. Además, tiene el espacio necesario para alojar, tanto el controlador ArbotiX-M como el servomotor que se encarga de la rotación en el eje z. Para facilitar esta rotación, la base tiene un rodamiento de bolas de 14 cm de diámetro. Algunas características dadas por el fabricante se recogen en la Tabla 8.1.

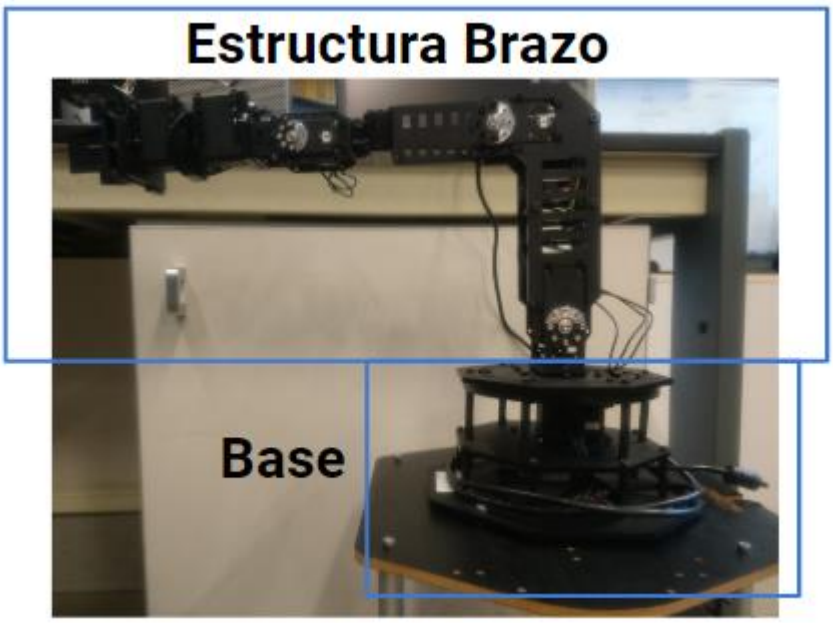
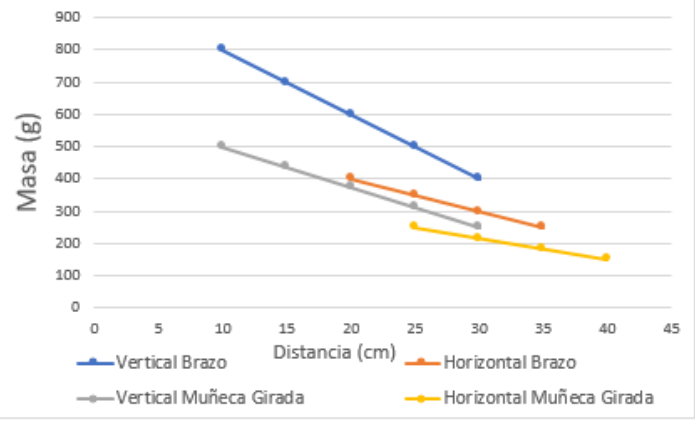


Figura 8.1 Partes de la Estructura del WidowX

Tabla 8.1 Características físicas y límites del brazo

	Características WidowX
Peso	1400 g
Alcance Horizontal	41 cm
Alcance Vertical	55 cm
Capacidad de Pinza	500 g
Capacidad del Brazo	

- **Servomotores:** Son 6 los servomotores incorporados en el brazo robótico, otorgándole 5 grados de libertad al brazo WidowX (Figura 8.2). Estos servomotores de la marca Dynamixel son servomotores punteros para robots. Están alimentados a 12 V de corriente continua. Cada uno de estos motores tiene incorporado un controlador PID para controlar la posición del eje de salida cuyos parámetros son ajustables. Están equipados con un controlador para realizar estas funciones. Casi todos pueden rotar 360° aunque por las barreras físicas que impone el brazo, no es posible rotar en todo este rango.

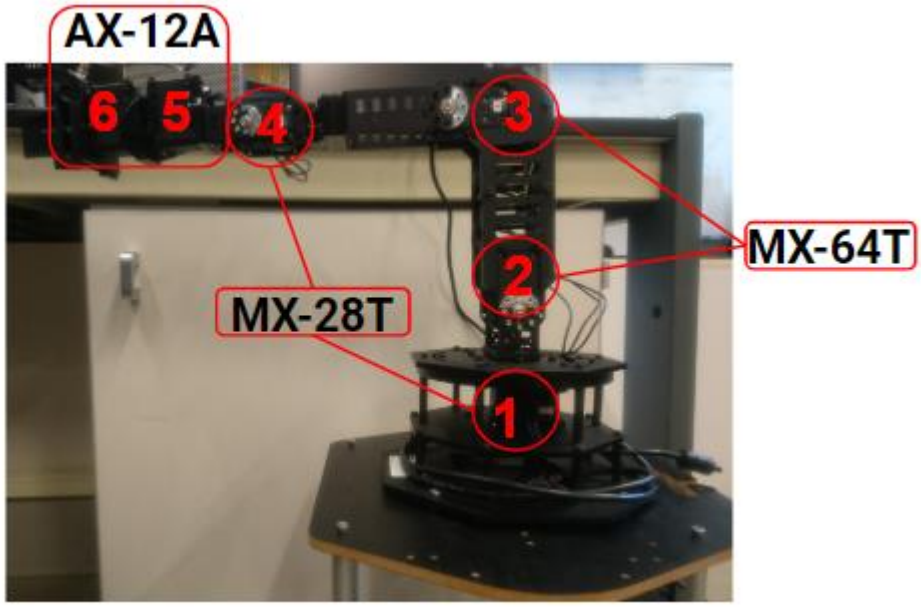


Figura 8.2 Colocación e Identificadores de los Servomotores

Esta manera de enumerar los servomotores es muy importante ya que son los identificadores que va a utilizar posteriormente el controlador ArbotiX para mandar las señales y que sean utilizadas por el servomotor correspondiente. La interconexión entre los servomotores se realiza mediante un esquema cableado en cadena a través de los conectores TTL de 3 pines presentes a cada lado de los servomotores. Estos conectores utilizan dos pines para alimentación (12V, 0V) y un tercer pin para transmitir toda la información. En función del identificador con el que se mande la información, será el servomotor correspondiente el que ejecute la acción. Hay que tener en cuenta que este canal de comunicación es bidireccional, así como el controlador ArbotiX manda información a los servomotores, estos mandan información al controlador indicando su temperatura, su posición o la corriente que están consumiendo. Este protocolo de comunicación permite que los servomotores se interconecten en lo que se conoce como esquema de cableado en cadena, es decir el controlador se conecta por un solo cable TTL al servomotor 1, este al 2, este al 3 y así hasta llegar al último servomotor, en este caso el 6. Las características más importantes de cada uno de ellos se recogen en la tabla 8.2.

Tabla 8.2 Características Servomotores Dynamixel

	MX-64T	MX-28T	AX-12A
Dimensiones	40,2x61,1x41 mm	35,6x50,6x35,5 mm	32x50x40 mm
Peso	126 g	72 g	54,6 g
Resolución	0,088°	0,088°	0,29°
Ángulo de Operación	360°	360°	300°
Maxima Corriente	4,1 A	1,4 A	0,9 A
Par Máximo	6 Nm	2,5 Nm	1,5 Nm
Protocolo	TTL	TTL	TTL
Velocidad de Comunicación	3 mbps	3 mbps	1 mbps
CPU	Cortex M3	Cortex M3	ATMega 8

Los rangos de los servomotores son los indicados en la tabla 8.2, pero estos límites no son físicamente alcanzables debido a las barreras físicas que impone la propia estructura del brazo. Los límites reales de los servos son los mostrados en la tabla 8.3.

Tabla 8.3 Límites Experimentales de los Servomotores

	Límite Inferior (Radianes)	Límite Superior (Radianes)
Servo 1	-π	π
Servo 2	-π/2	π/2
Servo 3	-π/2	π/2
Servo 4	-π/2	π/2
Servo 5	- π/2	π/2
Servo 6	0	2,5

- **Controlador:** El controlador incorporado en el brazo es un ArbotiX-M (Figura 8.3), este controlador basado en Arduino está pensado especialmente para el control de robots y servomotores Dynamixel. Existen librerías proporcionadas por Trossen Robotics para incorporar en el controlador utilizando el Arduino IDE, desde pruebas del brazo hasta controladores para poder funcionar con ROS. Las características más importantes del controlador son las siguientes:

  - Microcontrolador AV ATMEGA644p 16MHz.
  - 3 puertos TTL para los servos Dynamixel
  - Conectores para cable FTDI para programarlo
  - Alimentación 12V

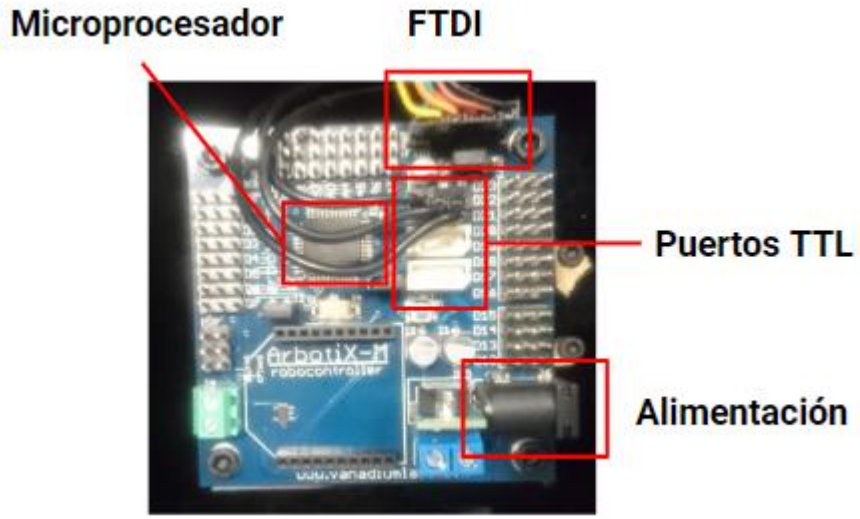


Figura 8.3 Partes más importantes ArbotiX-M

- **Base móvil Kobuki:** En este proyecto se ha utilizado la base móvil Kobuki (Figura 8.4). Es parte de la plataforma TurtleBot 2 que incluye esta base, la estructura que se le monta encima en la que se sitúa el brazo WidowX, además de otro hardware que no se utiliza en este proyecto. Las especificaciones más importantes de esta base móvil son:

    - Vmax = 70 cm/s
    - Carga máxima = 5 kg
    - Bumper izquierda, centro y derecha
    - Sensor de detección de pendientes
    - Sensor de caída de rueda
    - 4 entradas analógicas
    - 4 entradas digitales
    - 4 salidas digitales
    - Firmware actualizable vía USB
    - 3 botones táctiles programables

  En las páginas web oficiales de Kobuki o TurtleBot no se hace referencia a ningún microcontrolador o microprocesador que esté incorporado a la base móvil. No obstante, se entiende que sí lleva un MCU incorporado ya que existe la opción de actualizar el firmware de la base. Además, la base se conecta vía USB a un controlador externo (SBC) que se explicará a continuación. El micro de la base controla y gestiona los diferentes sensores presentes en la misma, mientras que el controlador externo se encarga de administrar toda esta información en ROS y comunicar la base con el resto de dispositivos. Esta misma estructura se encuentra presente en el TurtleBot 3 [8].

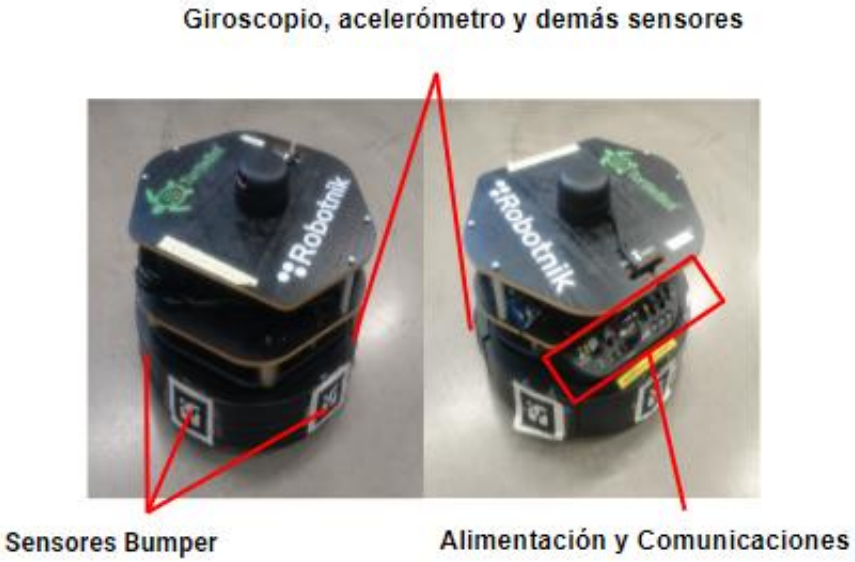


Figura 8.4 Partes más importantes del Kobuki

- **Odroid-XU4:** La palabra Odroid proviene de juntar las palabras open y Android, son unos ordenadores monoprocesador capaces de correr tanto versiones de Android, como algunas distribuciones de Linux, que es lo que nos interesa en este caso. Son computadoras con una gran capacidad de procesamiento y buenas características en general teniendo en cuenta tanto el tamaño como el precio de las mismas. Las características más importantes de la misma son (Figura 8.5):

    - Procesador Samsung Exynos de 8 núcleos a 2.0 GHz
    - 2 GB de RAM LPDDR3
    - 1 puerto USB 2.0
    - 2 puertos USB 3.0
    - 1 salida HDMI

  Esta placa no tiene conectividad Wifi, necesaria para acceder a ella desde un ordenador de sobremesa, por tanto, se le añade un módulo Wifi de TPLink.

- **Arduino Nano**: Se ha elegido este controlador debido a su reducido tamaño y la facilidad con la que se puede incorporar la gestión del sensor de temperatura en unas pocas líneas de código. Cómo características más destacables de esta placa tenemos:
    - Capacidad de alimentar dispositivos a 5 V y 3,3 V
    - Numerosas entradas y salidas tanto analógicas como digitales
    - Conector mini USB

- **Módulos XBee:** Estos módulos son dispositivos capaces de transmitir información de manera inalámbrica y sencilla. Para la realización de este proyecto se han utilizado dos módulos, uno que envía datos y otro que los recibe. En cuanto a los pines y características de estos módulos tenemos que:

    - Se alimentan a 5 V
    - Tienen dos pines de comunicación RX y TX
    - Poseen numerosas entradas y salidas digitales

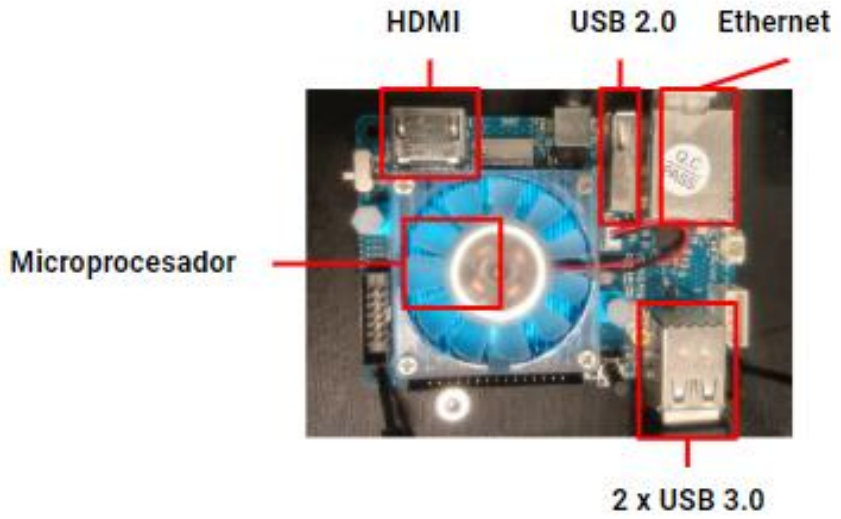


Figura 8.5 Características Odroid-XU4

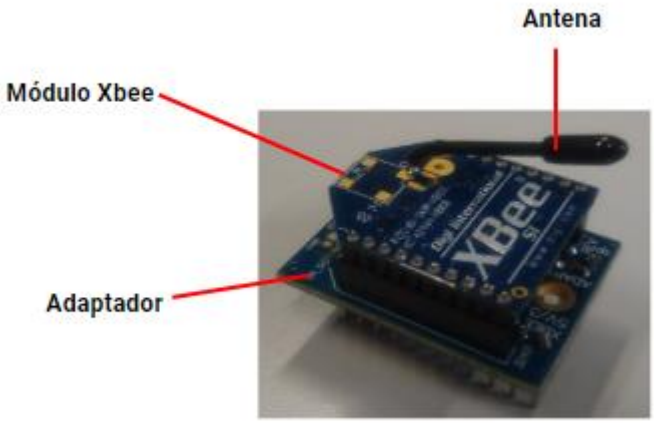


Figura 8.7 Módulo XBee y Adaptador

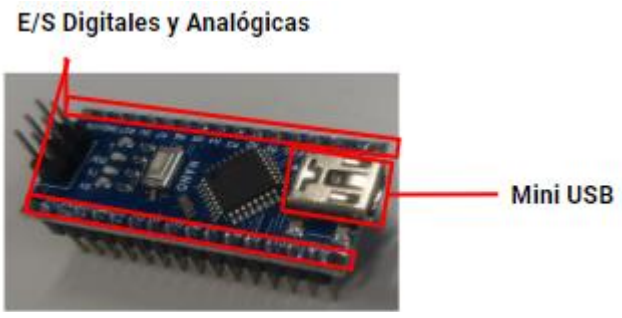


Figura 8.6 Partes más importantes Arduino Nano

- **Sensor de temperatura MLX90614:** Este sensor de temperatura es un pequeño sensor de infrarrojos para la medida a distancia de temperaturas. Tiene 4 pines para realizar sus conexiones.

    - Vdd 3 V
    - Vss 0 V
    - SDA y SCL

  Estos dos últimos pines se utilizan para comunicarse vía protocolo I2C con la placa Arduino. I2C es un bus serie de datos. Este pequeño sensor, además de detectar la temperatura de la pieza, nos dice también cuál es la temperatura del entorno. Para su utilización en este proyecto, se ha diseñado un soporte en CAD y se ha impreso utilizando CURA y una impresora 3D. El conjunto montado se puede ver en la Figura 8.9. Los planos de este soporte pueden encontrarse en: https://github.com/JulenCuadra/Planos_Soporte_Sensores_WidowX

Figura 8.8 Sensor MLX90614 by SparkFun Electronics Licencia CC BY 2.0

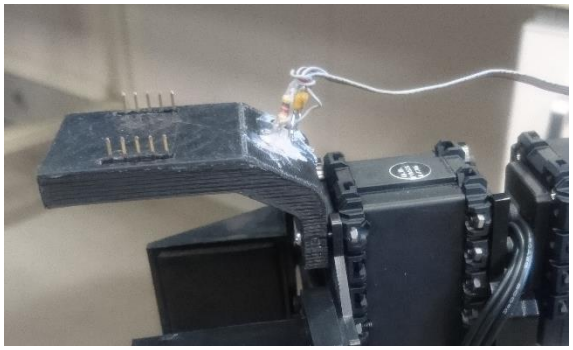


Figura 8.9 Colocación de sensor de temperatura (centro) en el soporte

8.2- Diseño Software

8.2.1- ROS

En cuanto al punto de vista de software, se va a explicar primero cómo funciona y qué partes componen el núcleo de este trabajo fin de grado que está constituido por ROS (Robot Operating System, Sistema Operativo Robot). ROS es un meta-sistema operativo para robots. Provee todos los servicios que ofrece un sistema operativo convencional como pueden ser: abstracción de hardware; control de bajo nivel de dispositivos, ya sea empleando C++, Python, Lisp o Java; comunicación entre diferentes procesos; además, implementa diferentes funcionalidades como la administración de paquetes de software. Es una herramienta de software libre, por lo que su acceso no tiene ninguna restricción y se puede integrar código desarrollado por terceros. El único requisito es la utilización de Linux para instalarlo.

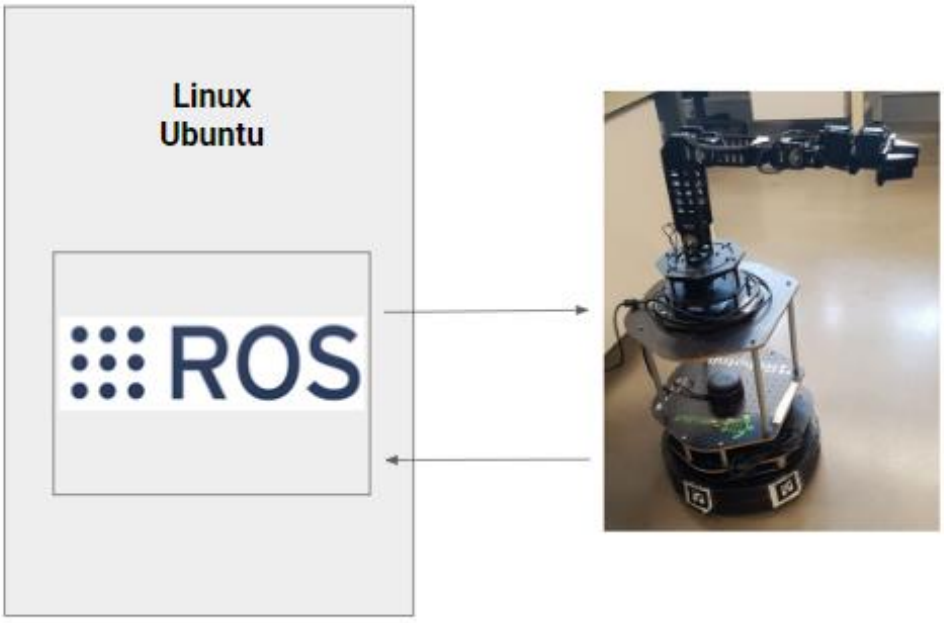


Figura 8.10 Comunicación de ROS y el equipo

ROS se basa en una red de procesos que pueden funcionar todos sobre una misma máquina o pueden estar repartidos entre diferentes máquinas, todos ellos centralizados en un mismo maestro. Esta red se basa en nodos y en la estructura de comunicación que emplea ROS.

Este meta-sistema operativo se organiza en tres estructuras bien diferenciadas:

- **Estructura de archivos:**

  - **Paquetes:** Al igual que en Linux, el software en ROS se distribuye mediante paquetes, estos son la unidad más básica de organización de software. Contienen el código de los nodos o los archivos de configuración entre otros. A su vez, los paquetes pueden estar compuestos de paquetes, con lo que pasarían a considerarse meta-paquetes, esto es útil para proyectos de mayor envergadura.

- **Mensajes:** Son archivos que contienen o definen diferentes tipos de estructuras de datos. Los mensajes pueden ser de tipos simples como, enteros, reales o strings. También pueden ser de tipos complejos como arrays de diferentes tipos simples o estructuras de datos complejas compuestas de tipos heterogéneos de datos. Estos últimos son los más utilizados en aplicaciones de movimiento o mapeado que requieren estructuras más complejas de información. Los tipos de mensajes están definidos en archivos tipo *.msg*, almacenados dentro del subdirectorio *msg/* del paquete correspondiente, por ejemplo, *mi_paquete/msg/mensaje.msg*. Como ejemplo de archivos de mensaje:

Mensaje Simple

```
std_msgs/String {
                    data String
                   }
```

Mensaje Complejo

```
geometry_msgs/Twist {
                        linear {
                                x Float64
                                y Float64
                                z Float64
                                }
                        angular{
                                x Float64
                                y Float64
                                z Float64
                                }
                    }
```

Figura 8.11 Ejemplos de Estructuras de Mensajes

- **Servicios:** Son archivos que habilitan el intercambio de datos en forma de petición/respuesta. Los servicios se definen mediante archivos de tipo .srv. Estos archivos están compuestos de dos mensajes diferentes, uno de petición y otro de respuesta, separados por '---'. Estos mensajes pueden ser también complejos y referirse a mensajes definidos por el usuario, en el mismo paquete o en otro.

mi_paquete/srv/servicio.srv

```
//Mensaje petición
float32 num_1
---
//Mensaje respuesta
float32 num_2
```

Figura 8.12 Ejemplo de Estructura de Servicio

- **Estructura de comunicación:** Se basa en un grafo computacional compuesto principalmente por Nodos, Tópicos y mensajes, todos ellos supervisados por un Maestro.
  - **Mensajes:** Son la forma principal que tienen los nodos de intercambiar información. Los nodos son los que utilizan estos mensajes para enviar datos de forma estructurada y ordenada. Los nodos publican los mensajes en los tópicos y se suscriben a otros tópicos para obtener los mensajes que otros nodos han publicado en ellos. El tipo de mensaje que se publica a un tópico, y el tipo del mensaje con el que se suscribe deben ser coincidentes.
  - **Nodos:** Los nodos son los procesos o programas escritos en C++, Python, Lisp o Java. Se basan en una comunicación asíncrona, no hay un intercambio de datos continuo o periódico. Esta estructura permite un gran modularidad en las aplicaciones desarrolladas en ROS. Los nodos tienen tres funciones: Se encargan de ejecutar la computación, *"publican"* los mensajes en los tópicos correspondientes y se suscriben a otros tópicos para obtener la información que contienen los mensajes ahí publicados y realizar diferentes operaciones con ellos.
  - **Tópicos:** Son la ruta por la que se transmiten los diferentes mensajes; en ete sentido, se asemejan a un bus de datos. Esta transmisión se realiza de manera asíncrona mediante la publicación de mensajes en ellos o la suscripción a los mismos. Los nodos pueden publicar y estar suscritos a varios tópicos a la vez. Esta forma de comunicarse permite que la producción de información sea desacoplada, los nodos no saben de la existencia los unos de los otros.
    Como ejemplo de esta estructura de tópicos y nodos podemos tener la siguiente situación:

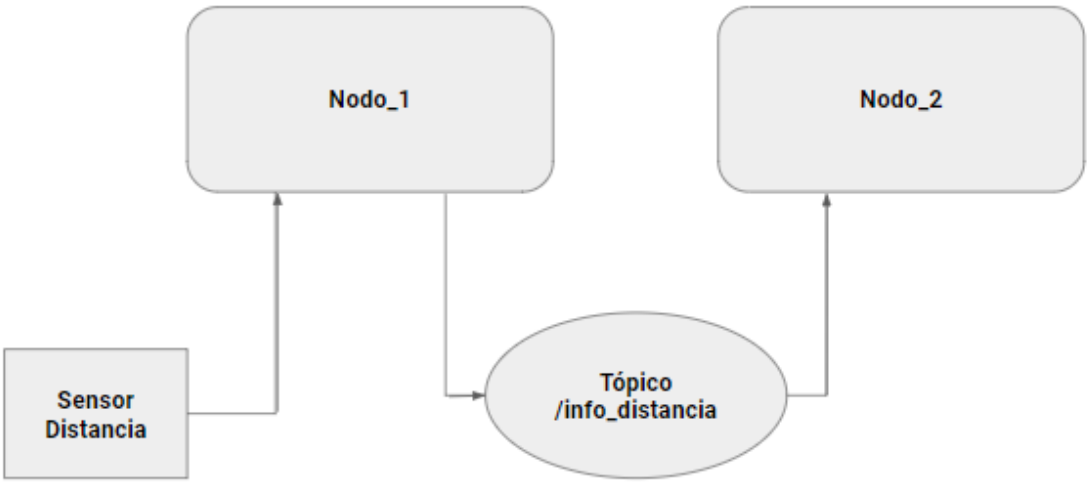


Figura 8.13 Ejemplo de Interacción de Nodos y Tópicos

En este ejemplo, el nodo_1 obtiene la información que proporciona el sensor de distancia mediante las conexiones físicas y protocolos pertinentes. Este nodo procesa esta información y publica un mensaje en el tópico

/info_distancia. El nodo_2 se suscribe a este tópico, por tanto, obtiene el mensaje que el nodo_1 ha publicado en el tópico y realiza otra operación con esa información.

- **Servicios:** Son también una forma de intercambio de datos, pero en este caso la interacción es de tipo cliente-servidor. Se basa en un protocolo RPC síncrono (llamada de procedimiento remoto). Un nodo servidor ofrece un servicio, otro nodo cliente, llama a ese servicio enviando un mensaje de petición y espera la respuesta.

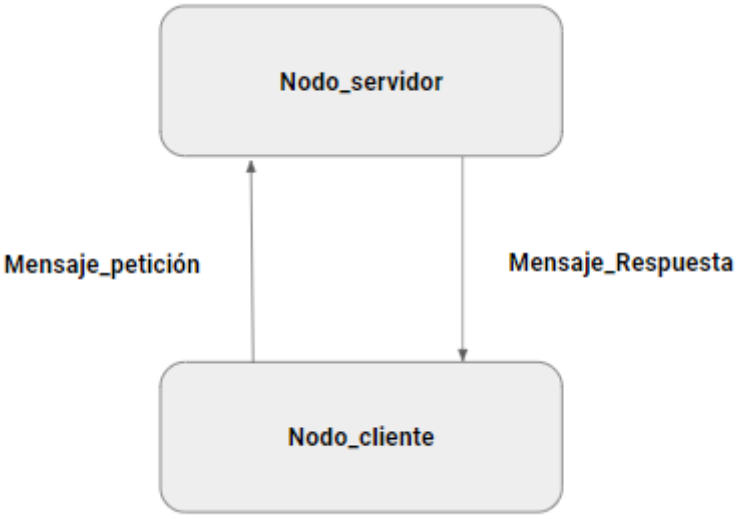


Figura 8.14 Ejemplo de interacción de Nodos en un Servicio

- **Maestro:** El maestro es la figura que supervisa el resto del grafo en ROS. Se encarga del registro y nombrado de los diferentes nodos que se lancen. Además, es el responsable de organizar las publicaciones y suscripciones a los diferentes tópicos. La secuencia de funcionamiento del maestro es la siguiente:

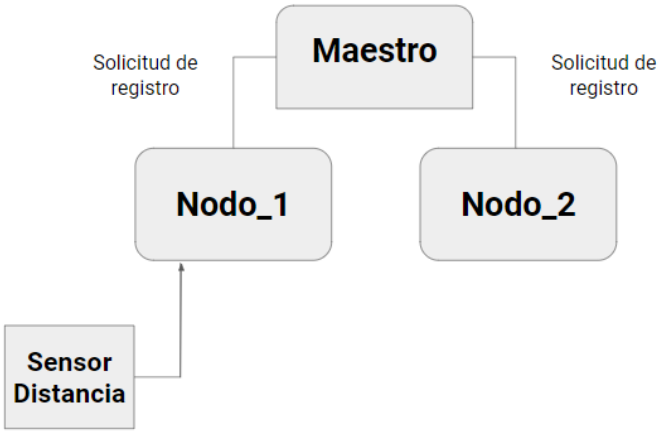


Figura 8.15 Registro de Nodos a través del Maestro

El maestro se ha encargado de registrar ambos nodos cuando han sido ejecutados.

En una situación como esta en la que tenemos dos nodos, nodo_1 y nodo_2, el nodo que quiere publicar, nodo_1, se lo comunica al maestro y este se encarga de registrar el tópico. De esta forma, el nodo_1 puede publicar la información, pero nadie la está utilizando.

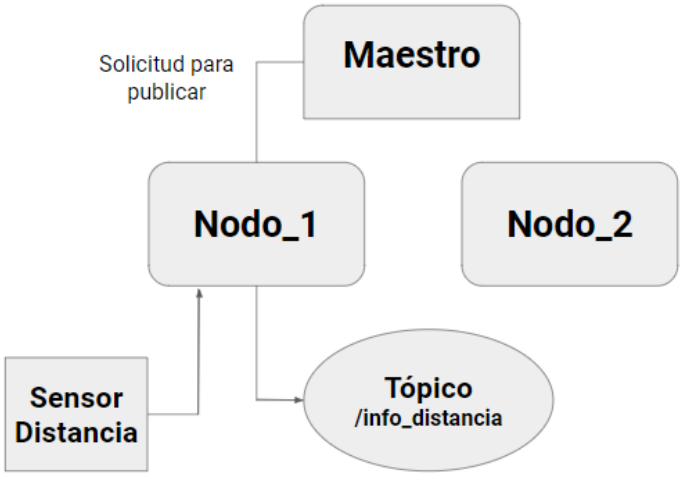


Figura 8.16 Solicitud al Maestro de publicación en Tópico

Lo mismo para el nodo_2, realiza una solicitud al maestro para suscribirse al tópico */info_distancia* y el maestro lo hace posible, completando así la comunicación y advirtiéndole a cada nodo de la existencia del otro.

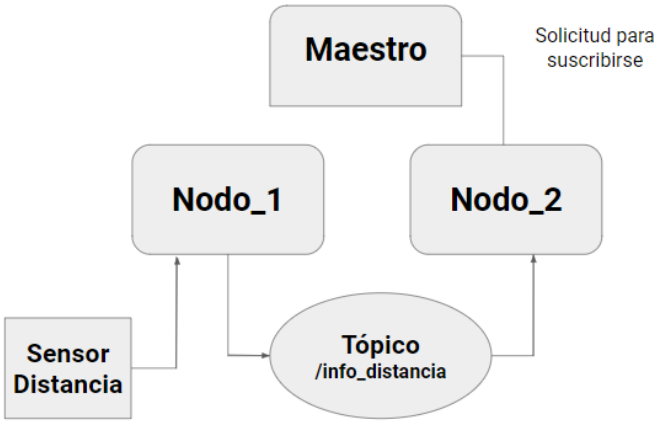


Figura 8.17 Solicitud al Maestro de suscripción al Tópico

En la situación final, una vez registrados los nodos y organizados los tópicos en los que publican y a los que se suscriben los nodos, la transmisión de información se realiza sin la intervención del maestro. La situación resultante es la de la Figura 8.13.

El maestro permite, además, que no sólo se registren nodos de una sola máquina, sino que se puede ejecutar el maestro en una máquina y los nodos en otra máquina. Estos nodos deberán registrarse contra el para una correcta comunicación entre ellos. Esto permite tener diversos equipos funcionando todos dentro de la misma red de ROS. Como inconveniente, hay que tener en

cuenta que esta estructura de un único maestro tiene el riesgo de saturarse en proyectos con múltiples equipos registrándose contra el mismo maestro.

- **Estructura de comunidad:** Al ser un software libre, la distribución del código se realiza a través de una comunidad compuesta por miles de desarrolladores, usuarios y empresas. Las partes más importantes de esta estructura se explican brevemente a continuación.
  - **Repositorios:** Son "almacenes" online de código. En ellos se pueden encontrar los diferentes paquetes que los usuarios han desarrollado y permiten un intercambio y distribución rápido y eficaz del código.
  - **Páginas web:** Son numerosas las direcciones web en las que podemos encontrar información y ayuda sobre ROS. Las más importantes son ROS wiki y ROS answers. En ellas hay información del funcionamiento de ROS y los diferentes paquetes, además de disponer de foros en los que obtener información y ayuda de otros usuarios.

8.2.2- Comandos más utilizados en ROS

La forma de comunicarse con ROS es a través de la consola de Linux Ubuntu, por tanto, hay que tener en cuenta cuáles son los diferentes comandos que podemos utilizar para acceder a las diferentes funcionalidades de ROS. Los comandos de mayor utilidad se recogen a continuación:

- **roscore:** Es el comando más fundamental en ROS, aunque con la utilización de archivos de tipo roslaunch, esta funcionalidad se realiza automáticamente sin tener que lanzar roscore. Con este comando se activa el maestro de ROS, que, como hemos visto, es la pieza fundamental para el funcionamiento de ROS. Además de esto se activan otras funcionalidades secundarias.

- **rosrun:** Con este comando puedes ejecutar un solo nodo en ROS. De esta manera, podemos lanzar los diferentes nodos que escribamos de forma individual y secuencial. Se usa de la siguiente manera:

rosrun <nombre_paquete> <nombre_nodo>

- **roslaunch:** Este comando permite dos cosas, si detecta que no hay un maestro activo, lanza uno y, además, permite lanzar varios nodos a la vez. Se usa con archivos .launch.

roslaunch <nombre_paquete> <nombre_archivo.launch>

- **rosnode:** Este comando tiene varias opciones, pero principalmente, se utiliza para obtener la lista de nodos activos o información de alguno de ellos como puede ser, a que tópico se suscribe o en que tópico publica. Algunas de estas opciones son:

rosnode list
rosnode info

- **rostopic:** Al igual que el comando anterior, se puede utilizar para varias cosas. La más importante es ver la lista de tópicos que están siendo utilizados en ese momento. También te permite ver la información de un tópico y ver qué tipo de mensaje se utiliza. Además, te permite publicar información a un tópico sin necesidad de que sea un nodo el que lo haga, es decir, directamente mandar una orden desde la ventana de comandos.

rostopic list
rostopic info
rostopic pub <nombre_tópico> <tipo_mensaje> <contenido_mensaje>

- **rosmsg:** Te permite ver qué mensajes están disponibles en todo ROS. Así mismo, permite obtener la información de un mensaje, devolviendo algo similar a las estructuras explicadas en la estructura de ficheros de ROS.

rosmsg list
rosmsg info

- **rospkg:** Se usa principalmente para ver qué paquetes están instalados en el entorno de ROS.

rospkg list

- **catkin_create_pkg:** Este comando se utiliza para crear un paquete propio en el espacio de trabajo configurado en ROS. El comando se encarga de crear un paquete con la estructura y los ficheros necesarios para ser interpretado correctamente en ROS.

catkin_create_pkg <nombre_paquete> lista de dependencias

- **catkin_make:** Lanzando el comando *catkin_make* en nuestro espacio de trabajo, conseguimos compilar y construir de forma automática todos los paquetes que tengamos añadidos en ROS.

8.2.3- Interacción Software

Ahora que ya se han visto los conceptos más importantes en lo que a ROS se refiere, se va a exponer el diseño de la solución final desde el punto de vista de software.

Lo primero es describir como interactúa el brazo WidowX en el entorno de ROS. En la placa Odroid, se va a ejecutar ROS, en este caso la versión ROS Kinetic sobre Linux Ubuntu 16.04. Es decir, la placa Odroid va a ser la encargada de ejecutar el controlador que permite interactuar con el brazo WidowX, pero desde el entorno de ROS. Además, gestiona toda la comunicación sobre nodos y tópicos de ROS. Mediante el uso de Arduino IDE, se carga en la placa ArbotiX-M un driver que permite controlar el brazo mediante ordenes de ROS. Este driver hace de intérprete entre las ordenes de alto nivel de ROS, las cuales convierte a paquetes de bytes interpretables por los servomotores. Esta comunicación se explica en el apartado 8.2.6- Driver ArbotiX-M. El sensor de temperatura MLX90614 obtiene la temperatura de la pieza colocada en la pinza del brazo y mediante un conjunto de XBees y Arduinos (explicadas en el apartado 7- Descripción de la solución propuesta) se obtiene esta información en ROS. En función de esta información el brazo realizará una operación u otra. Toda esta interacción se puede observar en la Figura 8.18.

En cuanto a la función de las placas Arduino en toda esta interacción, se va a exponer cuál es el papel de cada una de ellas. La placa Arduino que está conectada al sensor de temperatura tiene dos funciones, la primera de ellas es la lectura y tratamiento de datos que se obtienen de este sensor. Para ello se utiliza la librería de Arduino *SparkFun_MLX90614_Arduino_Library.* La segunda función de esta placa es el envío de datos vía serie al primero de los módulos XBee para que este los transmita de forma inalámbrica. Para ello se emula un puerto serie SW en un par de pines de la Arduino. La segunda placa Arduino se encarga de leer los datos que recibe el segundo XBee y traslada esta información a la placa Odroid. Para ello, lee los datos de un puerto serie SW conectado al XBee y los transmite a través del puerto serie HW a la placa Odroid. Toda esta funcionalidad está sustituida por un módulo XBee-Arduino en la Figura 8.18 pero se puede ver desarrollada en la Figura 8.20.

Como ya se ha indicado en otros apartados anteriores, el brazo no es más que un dispositivo que aporta una funcionalidad a una de las bases móviles que forman un proyecto de fabricación flexible. Dentro de esta funcionalidad, el brazo debe interactuar con la base para tomar decisiones y realizar operaciones. Teniendo esto en cuenta, la forma en la que el brazo actúa en función de la información del sensor y la base es la siguiente. Al inicializar el proceso, el brazo se incorpora a una situación de reposo. Después, el brazo se aproxima a la pieza. En este momento, el brazo utiliza la información que se obtiene del sensor de tal forma que, si la pieza está más fría que el entorno, el brazo vuelve a la posición inicial indicando al operario que la pieza estaba fría. En caso de que la temperatura de la pieza sea mayor que la temperatura

del entorno, el brazo la recoge y vuelve a la posición de reposo con la pieza cogida. Es en este punto en el que el manipulador móvil en conjunto se mueve y se dispone a llevar la pieza a otra parte del proceso. Una vez el manipulador ha llegado a la zona deseada, el brazo coloca la pieza en el lugar correspondiente. Por último, el brazo retorna a una posición de reposo. Ahora el manipulador móvil vuelve a la posición de partida con lo que se puede repetir de nuevo el proceso de evaluación y transporte. Toda esta estructura se puede ver esquemáticamente en la Figura 8.19.

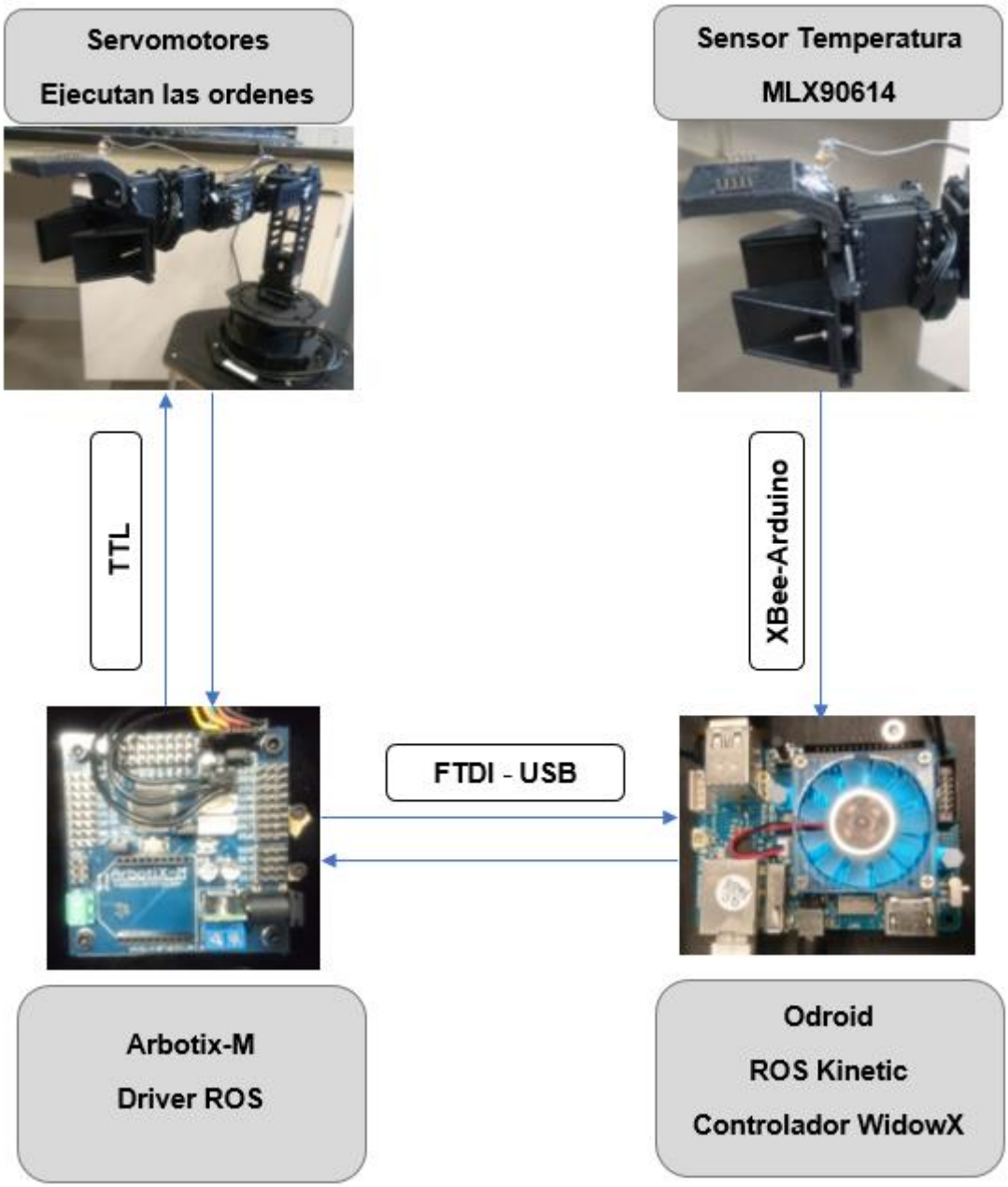


Figura 8.18 Distribución Software para el control del Brazo

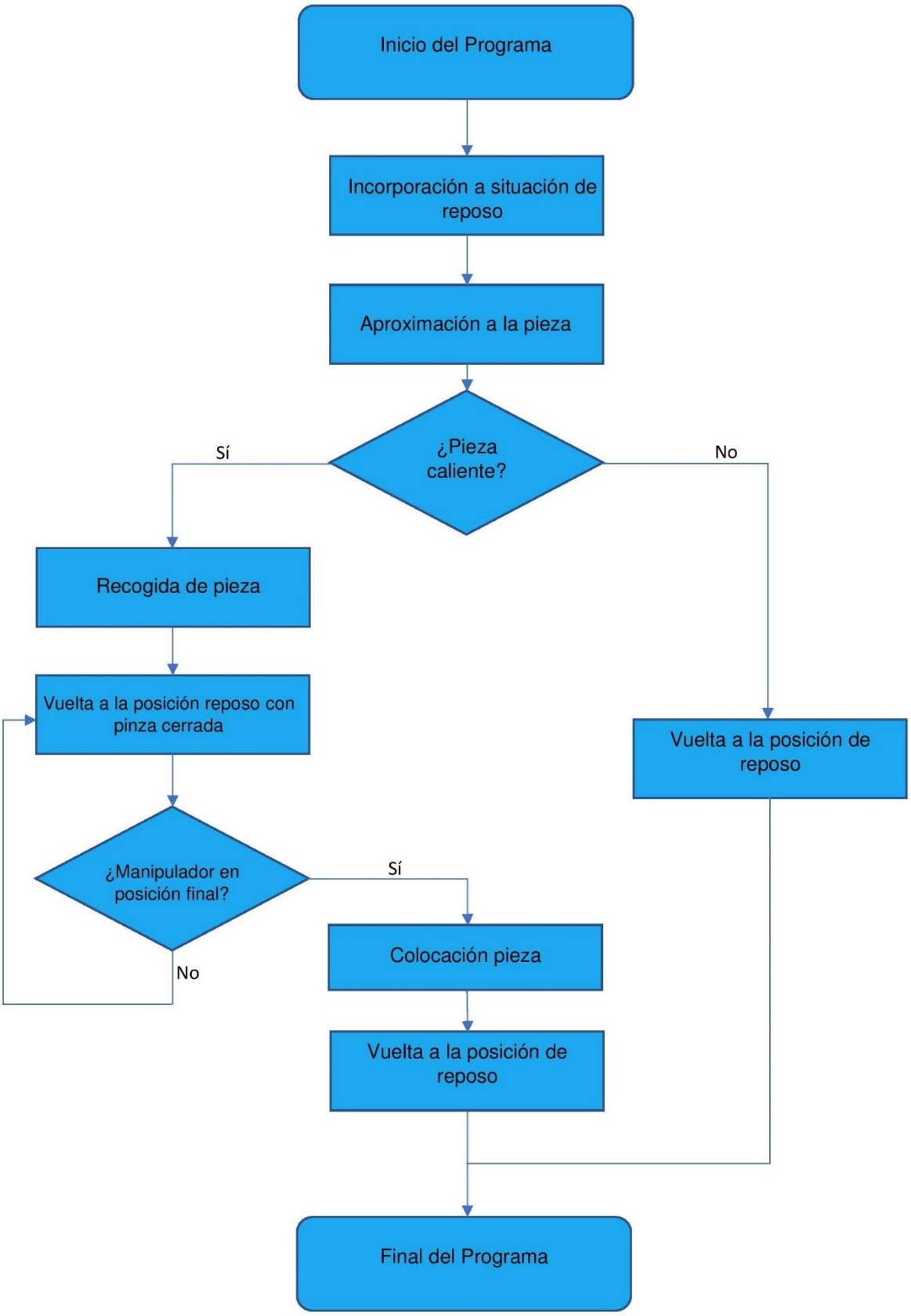


Figura 8.19 Diagrama de flujo de los procesos del Brazo

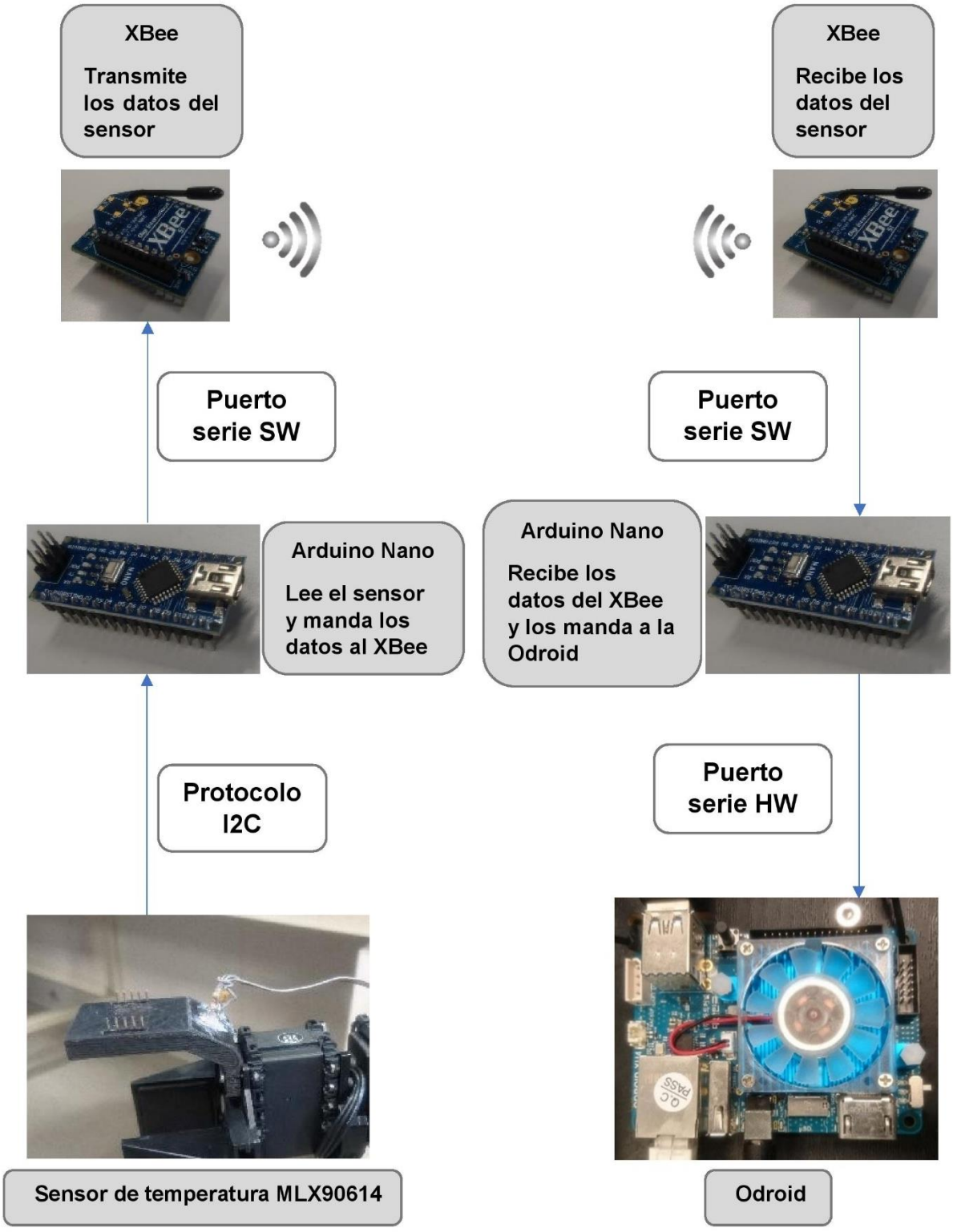


Figura 8.20 Comunicaciones de las Arduinos con el resto del conjunto

8.2.4- Controlador WidowX en ROS

Ya se ha explicado la interacción de los diferentes elementos, a continuación, se va a detallar cómo funciona el controlador que se ejecuta sobre la placa Odroid para poder controlar el brazo con la placa ArbotiX-M como intermediaria.

Este controlador es parte de un paquete de software mayor que podemos encontrar en:

https://github.com/Interbotix/widowx_arm

Este paquete incluye todos los archivos necesarios para implementar el brazo en la interfaz gráfica de MoveIt, además de todo lo necesario para el controlador. A parte de este paquete, se necesita también el paquete relativo a la comunicación con la placa ArbotiX-M el cual podemos encontrar en:

https://github.com/Interbotix/arbotix_ros/tree/turtlebot2i

Estos dos paquetes posibilitan el control del brazo robótico utilizando el hardware ya mencionado. Para ejecutar el controlador no tenemos más que hacer:

roslaunch widowx_arm_controller widowx_arm_controller.launch

Como ya se ha explicado al describir los comandos más importantes que se utilizan en ROS, el comando anterior ejecuta el archivo *widowx_arm_controller.launch* que se encuentra dentro del paquete *widowx_arm_controller.*

Este archivo *launch* se encarga de, lanzar un maestro en caso de que no haya uno activo, lanza un nodo */gripper_controller* que se encarga del control de la pinza del brazo y lanza un nodo */arbotix.* Este último nodo se encarga de la comunicación con la placa ArbotiX-M, el control de las articulaciones y de otras ordenes que se utilizarán más adelante. La estructura de nodos y tópicos más importantes que se lanzan con este controlador se muestra en la Figura 8.21. Este archivo *launch*, indica, además, a través de un archivo de configuración, en qué puerto serie se va a realizar la conexión con la ArbotiX-M. El tópico */joint_states*, es en el que tanto el nodo */arbotix* como el nodo */gripper_controller* publican la posición y la velocidad de los servomotores.

Estos nodos ofrecen dos formas de interactuar con el brazo de forma externa (con el usuario). La primera de ellas es a través de los tópicos */joint_x/command* (para x de 1 a 5) y */gripper_revolute_joint/*command (para controlar el giro del servomotor 6, la pinza). La segunda mediante el tópico /arm_controller/command, que permite el control completo del brazo, realizar trayectorias y controlar la velocidad de los movimientos.

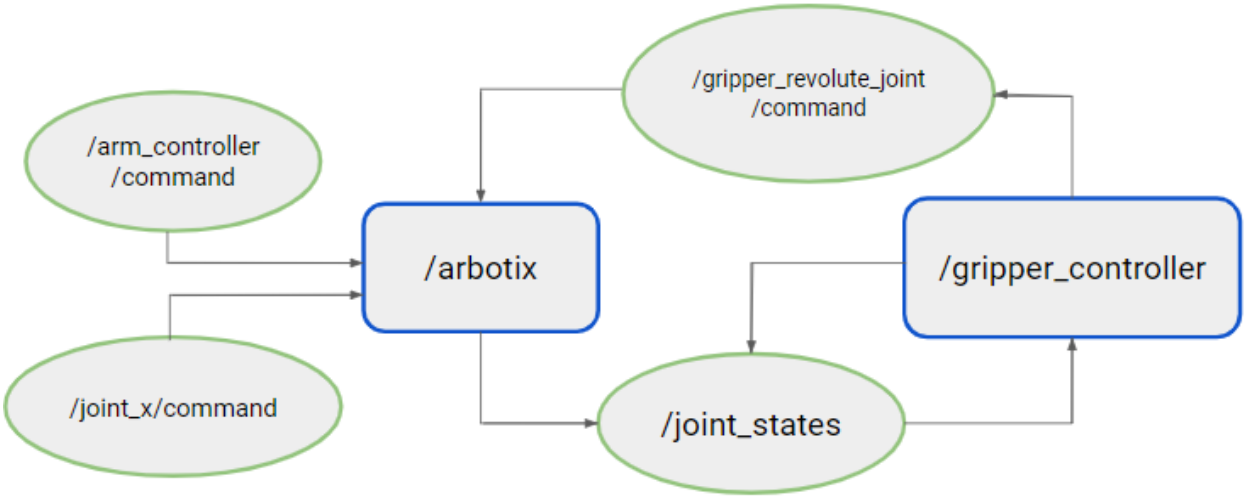


Figura 8.21 Nodos y Tópicos lanzados con widowx_arm_controller.launch

8.2.4.1- Control individual de las articulaciones

Primero nos vamos a centrar en los tópicos */joint_x/command* (para x de 1 a 5) y */gripper_revolute_joint/*command (para controlar el giro del servomotor 6, la pinza). Podemos publicar en estos tópicos la posición a la que queremos mandar cada uno de los servomotores individualmente. Esta orden se manda en forma de número real dentro del rango mostrado en la tabla 8.3. La forma de publicar la información en estos tópicos es mediante la estructura pertinente en un nodo de ROS o mediante un comando del tipo:

rostopic pub <tópico> <tipo_de_mensaje> <info_del_mensaje>

Por ejemplo, para colocar la articulación 1 en el ángulo 0.0 hacemos:

rostopic pub /joint_1/command std_msgs/Float64 "data: 0.0"

Esta forma de mandar órdenes es muy cómoda y rápida, pero como contrapunto tenemos que no es posible controlar la velocidad de los servomotores, lo que conlleva un claro riesgo para la persona que lo opera o las piezas que se manejan. Además, si se requiere controlar todas las articulaciones en numerosas posiciones diferentes (como es el objetivo de este trabajo), esta forma de comunicarse se vuelve muy incómoda y poco eficiente teniendo que recurrir a muchas líneas de código convenientemente secuenciadas para poder lograr una trayectoria compleja.

8.2.4.2- Control de trayectorias

La otra forma de comunicación que habilitan los nodos lanzados con el controlador, es la forma más cómoda de trabajar y por la que se ha optado en este trabajo. Consiste en utilizar el tópico */arm_controller/command*. La estructura final de comunicación en ROS de tópicos y nodos es la mostrada en la Figura 8.22.

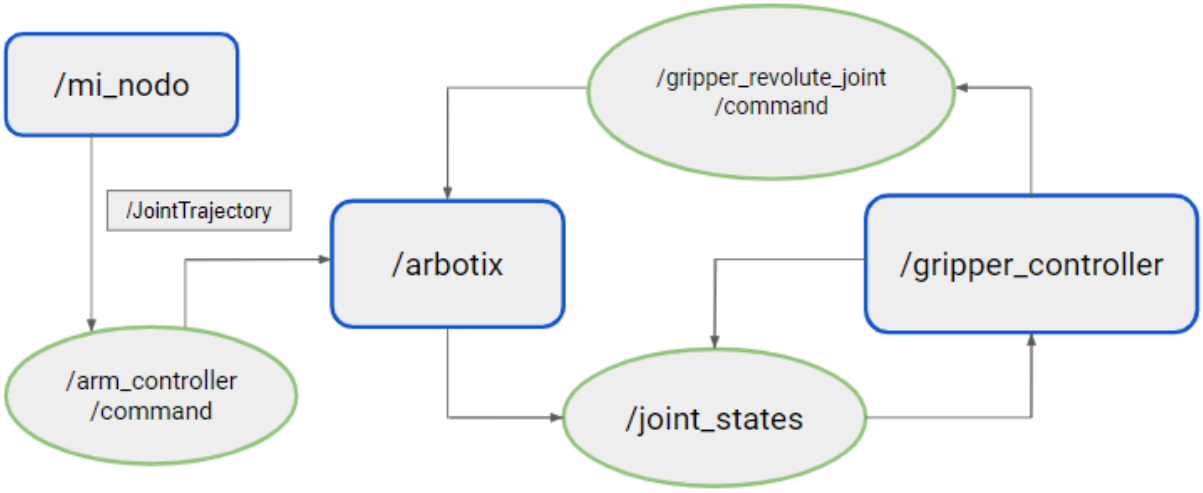


Figura 8.22 Publicación de Trayectorias en tópico /arm_controller/command

8.2.4.2.1- Mensaje JointTrajectory

El tópico */arm_controller/command* maneja mensajes del tipo */JointTrajectory*, del paquete *trajectory_msgs.* Se puede deducir fácilmente del nombre del mensaje como este está directamente relacionado con la generación de trayectorias para todas las articulaciones. La estructura de este mensaje es bastante compleja, en la Figura 8.23 se puede ver la estructura general del mismo. Está compuesto por tres campos:

- Una cabecera (*header*). En ella se indica el tiempo de inicio y la unión de referencia del brazo. El contenido de este mensaje se muestra en la Figura 8.24.
- Un array de strings (*joint_names*) donde se indican los nombres de las diferentes articulaciones del brazo. Estos nombres son los que se indican en el archivo de configuración que utiliza el archivo launch como ya se ha indicado con anterioridad.
- Un array de mensajes del tipo *JointTrajectoryPoints*. La estructura de este mensaje puede verse en la Figura 8.25.

Para entender del todo el mensaje *JointTrajectory*, es necesario seguir desglosando cada uno de sus campos, lo que nos lleva a la estructura que podemos ver en la Figura 8.26.

```
trajectory_msgs/JointTrajectory.msg {
                                    std_msgs/Header header
                                    string[] joint_names
                                    trajectory_msgs/JointTrajectoryPoint[] points
                                    }
```

Figura 8.23 Estructura general del mensaje JointTrajectory

```
std_msgs/Header  {
                 uint32 seq
                 time stamp
                 string frame_id
                 }
```

Figura 8.24 Estructura mensaje Header

```
trajectory_msgs/JointTrajectoryPoint.msg {
                                         float64[] positions
                                         float64[] velocities
                                         float64[] accelerations
                                         float64[] effort
                                         duration time_from_start
                                         }
```

Figura 8.25 Estructura mensaje JointTrajectoryPoint

```
trajectory_msgs/JointTrajectory.msg {
                                    header {
                                            uint32 seq
                                            time stamp
                                            string frame_id
                                            }
                                    string[ ] joint_names
                                    points {
                                            points[1] {
                                                    float64[] positions
                                                    float64[] velocities
                                                    float64[] accelerations
                                                    float64[] effort
                                                    duration time_from_start
                                                    }
                                            points[2] {
                                                    float64[] positions
                                                    float64[] velocities
                                                    float64[] accelerations
                                                    float64[] effort
                                                    duration time_from_start
                                                    }
                                            .
                                            .
                                            .
                                            points[n] {
                                                    float64[] positions
                                                    float64[] velocities
                                                    float64[] accelerations
                                                    float64[] effort
                                                    duration time_from_start
                                                    }

                                            }
                                    }
```

Figura 8.26 Desarrollo mensaje JointTrajectory

Lo más interesante de este desglose es analizar la composición del array de mensajes *JointTrajectoryPoints*. El campo *points* del mensaje *JointTrajectory* está compuesto de diferentes puntos dentro de una trayectoria. Cada uno de estos puntos tiene a su vez numerosos campos como son, un array de posiciones, de velocidades, de aceleraciones y de esfuerzos. Cada uno de estos arrays tiene la dimensión del campo *joint_names* del mensaje *JointTrajectory*, es decir, en cada uno de estos arrays se indican la posición, velocidad, aceleración y esfuerzo de cada uno de los servomotores, en el mismo orden de articulaciones que se ha indicado previamente en *joint_names*. Además, el apartado *time_from_start*, indica cuándo se va a llegar al punto i con respecto al inicio de la trayectoria, es decir, desde el tiempo indicado en el campo *stamp* del campo *header* del mensaje *JointTrajectory*. De esta forma, se indica lo que tarda en realizarse la trayectoria desde el inicio sin más que atender al campo *time_from_start* del último de los puntos. Esta forma de funcionar permite dejar sin rellenar los campos de velocidad, aceleración y esfuerzo, indicando únicamente que posición es la deseada para cada servomotor en cada posición. Notar también que los campos *time_from_start* de cada uno de los puntos tienen que evolucionar de forma creciente, es decir, no se va a llegar al punto 2 sin haber llegado al punto 1 por lo que el parámetro *time_from_start* del punto 2 deberá ser mayor que el del punto 1.

Para entender mejor este mensaje y su utilización, se plantea a continuación un caso de uso sencillo. En este caso, se va mandar al brazo a una posición con todos sus servomotores centrados y después, se va a cerrar la pinza. El mensaje completo se ve en la Figura 8.27. El brazo tardará 3 segundos en completar cada movimiento. El campo *header.seq* lo auto-rellena ROS, mientras que los campos *velocities*, *accelerations* y *effort* no es necesario rellenarlos para las funcionalidades de este trabajo fin de grado.

```
trajectory_msgs/JointTrajectory.msg {
                                    header {
                                            stamp "tiempo_actual"
                                            frame_id "base_footprint"
                                            }
                                    joint_names ["joint_1", "joint_2", "joint_3", "joint_4",
"joint_5", "gripper_revolute_joint"]
                                    points {
                                            points[1] {
                                                    positions [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
                                                    time_from_start 3 s
                                                    }
                                            points[2] {
                                                    positions [0.0, 0.0, 0.0, 0.0, 0.0, 2.0]
                                                    time_from_start 6 s
                                                    }

                                            }
                                    }
```

Figura 8.27 Ejemplo de uso del mensaje JointTrajectory

Con la utilización de este mensaje en el tópico */arm_controller/command* se ha conseguido controlar la posición individual de cada uno de los servomotores y más importante, se ha conseguido manipular la velocidad con la que el brazo realiza cada uno de los movimientos según sean movimientos de aproximación o movimientos más delicados.

Hay que tener en cuenta que, con la utilización de este mensaje es importante controlar en todo momento donde se encuentra el brazo robótico espacialmente. Es decir, el brazo no detecta obstáculos en su entorno por lo que, si en una trayectoria, para ir del punto A al punto B el brazo colisiona con un obstáculo, el brazo no va a reaccionar, como mucho provocaría un bloqueo del mismo que suele derivar en un bloqueo del controlador ArbotiX-M.

8.2.4.3- Sensor de temperatura en ROS

Una vez definida la manera de trabajar con el tópico */arm_controller/command*, se introduce el sensor de temperatura en ROS. Al añadir el sensor de temperatura, creamos un nuevo nodo */lector_temperatura* que gestiona la conexión serie con una de las Arduinos y publica en el tópico */info_temperatura* la información que se ha recibido del sensor de temperatura. Después, el nodo */mi_nodo*, que gestiona la trayectoria completa del manipulador móvil, publica información a */arm_controller/command* en función de la información que se reciba del sensor. Esta interacción con el sensor de temperatura se muestra en la Figura 8.28.

Es decir, cada vez que el brazo llegue a una etapa del diagrama de flujo (Figura 8.19) que conlleve movimiento, se va a publicar una trayectoria en el tópico */arm_controller/command.*

El paquete conteniendo los nodos utilizados en este apartado se puede encontrar en:

https://github.com/JulenCuadra/TFG_Kobuki_WidowX

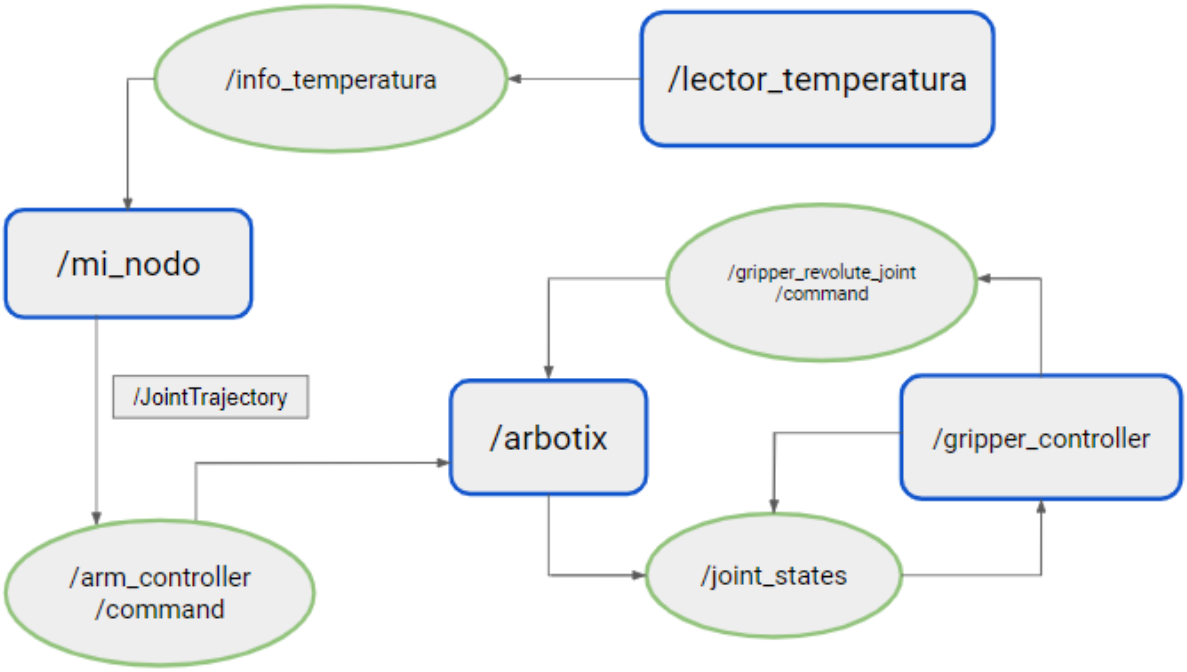


Figura 8.28 Funcionamiento de nodos de gestión de temperatura y gestión de trayectorias

8.2.5- Identificación Servomotores

Para que toda la comunicación anteriormente explicada sea posible, es necesario identificar los servomotores. Esto es lo primero que hay que hacer para posibilitar la comunicación con la ArbotiX.

Esta identificación consiste en darle un identificador a cada uno de los servomotores, siguiendo la numeración indicada en la Figura 8.2. De esta forma, cuando el controlador ArbotiX-M envíe un paquete de datos a través de la comunicación TTL, el servo con la identificación adecuada responda a esa orden. De esta forma, podemos tener los servomotores en una estructura en cadena en vez de conectar cada uno a un puerto diferente en la placa ArbotiX.

Para realizar esta identificación, es necesario seguir el tutorial disponible en la página:

https://learn.trossenrobotics.com/index.php/getting-started-with-the-arbotix/1-using-the-tr-dynamixel-servo-tool#&panel1-1

En este apartado, se explica cómo instalar un software para posibilitar la identificación de estos servomotores Dynamixel de forma individual. Además, da la opción de centrar los servomotores, algo crucial para un correcto montaje.

Esta identificación es crucial para entender la comunicación con el Driver de la placa ArbotiX-M que se explica en el siguiente punto.

8.2.6- Driver ArbotiX-M

La base de la comunicación entre ROS y los servomotores es este driver instalado utilizando el software Arduino IDE. Como ya se ha explicado en apartados anteriores, este driver hace de "interprete" entre ROS y los servomotores, los servos utilizan una comunicación basada en unos paquetes de bytes que se explicarán en este apartado. Sin embargo, las ordenes de ROS no siguen este sistema de comunicación. Por tanto, las ordenes que recibe el nodo */arbotix*, son transmitidas vía serie a la placa ArbotiX, la cual los transforma a ordenes interpretables por los servomotores.

Esta estructura de bytes se basa principalmente en paquetes de datos, cada movimiento es un conjunto de 17 bytes con una estructura muy concreta. El controlador recibe bytes de diferente significado según se quiera controlar en un modo cartesiano, cilíndrico o cada articulación individualmente. Atendiendo a este último la estructura de estos bytes es la de la Figura 8.29.

Los bytes relativos a los servos indican la posición que se requiere para cada uno de los servos. El byte delta indica la duración del movimiento, multiplicando delta (en decimal) por 16 obtenemos los ms que tardará en moverse. El byte de botón controla ciertas salidas digitales. Con el byte de instrucción extendida podemos cambiar entre los diferentes modos de control del brazo o hacer que este se mueva a la posición indicada dándole a este byte un valor de 0.

El byte checksum se utiliza para comprobar que la comunicación serie entre ambos extremos ha sido correcta y no se ha perdido información en el proceso.

En caso de querer controlar el brazo en modo cilíndrico o cartesiano la estructura de los bytes de cada servo se modifica por coordenadas espaciales. En todo caso la comunicación es similar, paquetes de 17 bytes.

Es importante saber que la placa ArbotiX ejecutará un paquete antes de pasar al siguiente, es decir, irá leyendo todos los bytes de un paquete y no pasará al siguiente hasta que acabe.

Por ejemplo, si queremos centrar todos los servos en el modo de control individual de estos, habría que mandar:

```
0xff 0x8 0x0 0x8 0x0 0x8 0x0 0x8 0x0 0x2 0x0 0x2 0x0 0x80 0x0 0x0 0x5b
```

Nº Byte	Significado	
1	Cabecera (0xFF/255)	
2	Byte Alto Rotación Servo Base	Servo #ID 1
3	Byte Bajo Rotación Servo Base	
4	Byte Alto Rotación Servo Hombro	Servo #ID 2
5	Byte Bajo Rotación Servo Hombro	
6	Byte Alto Rotación Servo Codo	Servo #ID 3
7	Byte Bajo Rotación Servo Codo	
8	Byte Alto Ángulo Muñeca	Servo #ID 4
9	Byte Bajo Ángulo Muñeca	
10	Byte Alto Rotación Muñeca	Servo #ID 5
11	Byte Bajo Rotación Muñeca	
12	Byte Alto Pinza	Servo #ID 6
13	Byte Bajo Pinza	
14	Byte Delta	
15	Byte de Botón	
16	Byte Instrucción Extendida	
17	Checksum	

Figura 8.29 Estructura de un paquete de bytes para control individual de servos

9 - Descripción de los resultados

Ahora que ya se ha explicado la totalidad del proyecto tanto desde el punto de vista del hardware como del software, se van a comentar los resultados conseguidos. Para esto hay que tener en cuenta que estos resultados se van a contrastar con los objetivos planteados en el inicio de este proyecto y teniendo en cuenta los requerimientos dados.

Primero, hay que recordar el objetivo principal de este trabajo fin de grado:

Diseñar y desarrollar nodos funcionales en ROS para sensorizar y controlar el brazo manipulador WidowX sobre una plataforma TurtleBot 2.

Como se ha visto en el apartado 8.2- Diseño Software, se ha conseguido realizar una trayectoria para el brazo WidowX. Además, se ha conseguido incorporar un sensor, en este caso, un sensor de temperatura MLX90614 para poder tomar decisiones con respecto a las operaciones que debe realizar este manipulador. La realización de esta trayectoria ha sido posible mediante la utilización de mensajes de ROS diseñados para esta funcionalidad, por lo que se ha conseguido implementar esta trayectoria desde bajo nivel en ROS.

Atendiendo a los objetivos secundarios, que son:

1. Controlar cada una de las articulaciones del brazo individualmente.
2. Controlar la velocidad de movimiento del brazo.
3. Conseguir que el brazo realice operaciones diferentes en función de la información proporcionada por un sensor acoplado a la pinza del brazo.

A través de la utilización del mensaje */JointTrajectory* se ha podido controlar de forma individual cada uno de los 6 servomotores que tiene el brazo WidowX. También se ha podido controlar la velocidad de los movimientos del brazo mediante la correcta utilización del parámetro *time_from_start*. Además, como ya se ha comentado, se ha conseguido implementar un sensor en la estructura de ROS para la utilización de esa información por parte de los nodos que publican las trayectorias del brazo.

En cuanto a los requerimientos dados al inicio del planteamiento de este trabajo fin de grado, se han realizado las trayectorias utilizando el brazo manipulador WidowX sobre una base TurtleBot 2.

Se ha conseguido también implementar la interacción del brazo WidowX y el Kobuki desde el entorno de ROS utilizando la placa Odroid. Las funciones se han separado en módulos aprovechando la funcionalidad de los nodos de ROS.

10 - Plan de trabajo

Este trabajo fin de grado es parte de un proyecto mayor que ya se ha explicado en apartados anteriores. El equipo de trabajo que ha realizado este proyecto está compuesto por:

- **Director de proyecto:** Oscar Casquero Oyarzabal
- **Ingenieros junior:** Asier Alonso Tejeda y Julen Cuadra Gomez
- **Gestora:** Maialen González Jaio

A continuación, se describen las diferentes fases que han compuesto el proyecto y las distintas tareas que componen cada una de estas fases.

- **Fase 1:** Gestión
  Descripción: Esta fase permite la correcta ejecución del resto de tareas a lo largo del proyecto y su interrelación.
  Responsable: Maialen González Jaio
  Participantes: Asier Alonso Tejeda y Julen Cuadra Gomez
  Duración: 109 días

- **Fase 2:** Puesta a punto de los equipos
  Descripción: En esta fase se preparan, configuran y montan todos los equipos necesarios para llevar adelante el proyecto.
  Responsables: Asier Alonso Tejeda y Julen Cuadra Gomez
  Participantes: Maialen González Jaio
  Duración: 18 días

  - **Tarea 2.1:** Montaje WidowX
    Descripción: Esta tarea consiste en realizar tanto el montaje como las configuraciones del hardware necesario para poder implementar el control del manipulador WidowX en ROS.
    Responsable: Julen Cuadra Gomez
    Participantes: Asier Alonso Tejeda y Maialen González Jaio
    Duración: 12 días

  - **Tarea 2.2:** Configuración TurtleBot 2
    Descripción: En esta tarea se prepara y configura la base móvil Kobuki, se monta la estructura TurtleBot 2 sobre este se acopla el brazo WidowX
    Responsable: Asier Alonso Tejeda
    Participantes: Maialen González Jaio y Julen Cuadra Gomez
    Duración: 13 días

- **Tarea 2.3:** Instalación Software
  Descripción: El objetivo de esta tarea es preparar todos los equipos pero desde el punto de vista de software, es decir, la instalación del sistema operativo Linux Ubuntu, la instalación de ROS y demás software.
  Responsables: Asier Alonso Tejeda y Julen Cuadra Gomez
  Participantes: Todos
  Duración: 6 días

- **Fase 3:** Formación
  Descripción: Esta fase consiste en un periodo para que los participantes del proyecto adquieran los conocimientos y la facilidad necesaria para usar el software necesario para desarrollar las aplicaciones explicadas en el apartado de requerimientos.
  Responsables: Oskar Casquero Oyarzabal y Maialen González Jaio
  Participantes: Todos
  Duración: 32

  - **Tarea 3.1:** Familiarización con Linux
    Descripción: Tarea para poder adquirir un conocimiento básico de la forma de operar en Linux y de sus comandos fundamentales.
    Responsable: Asier Alonso Tejeda
    Participantes: Maialen González Jaio y Julen Cuadra Gomez
    Duración: 7

  - **Tarea 3.2:** Familiarización con ROS
    Descripción: Tarea para aprender sobre ROS, la forma en la que este se integra en Linux, como se interacciona con ROS desde la ventana de comandos de Linux y la interacción básica con el resto de equipos.
    Responsable: Julen Cuadra Gomez y Maialen González Jaio
    Participantes: Todos
    Duración: 10

  - **Tarea 3.3:** Aprendizaje Python
    Descripción: En esta tarea se obtendrán los conocimientos necesarios del lenguaje de programación de alto nivel Python, pero enfocado al uso del mismo para programar nodos de ROS, es decir, la utilización de Python y más concretamente la librería rospy.
    Responsable: Oskar Casquero Oyarzabal
    Participantes: Todos
    Duración: 15

- **Fase 4:** Implementación de equipos en ROS
  Descripción: Fase para instalar en ROS todos los paquetes necesarios para el control de los diferentes equipos que van a utilizarse en el desarrollo del proyecto.
  Responsable: Asier Alonso Tejeda y Julen Cuadra Gomez
  Participantes: Maialen González Jaio
  Duración: 57 días

  - **Tarea 4.1:** Instalación paquetes de WidowX
    Descripción: Identificación en los repositorios de los paquetes más adecuados para el control del WidowX, su posterior instalación y configuración
    Responsable: Julen Cuadra Gomez
    Participantes: Asier Alonso Tejeda y Maialen González Jaio
    Duración: 5 días

  - **Tarea 4.2:** Instalación paquetes Kobuki
    Descripción: Buscar los paquetes más adecuados para el control del Kobuki, su instalación y configuración.
    Responsable: Asier Alonso Tejeda
    Participantes: Maialen González Jaio y Julen Cuadra Gomez
    Duración: 3 días

  - **Tarea 4.3:** Desarrollo de programas WidowX
    Descripción: Tarea para desarrollar los programas más básicos de control del brazo WidowX, análisis de los mensajes, tópicos y nodos utilizados y su uso.
    Responsable: Julen Cuadra Gomez
    Participantes: Asier Alonso Tejeda y Maialen González Jaio
    Duración: 15 días

  - **Tarea 4.4:** Desarrollo de programas Kobuki
    Descripción: En esta tarea se analizarán los mensajes, nodos y tópicos que se utilizan para el control de la base móvil Kobuki y su implementación para el desarrollo de programas de control.
    Responsable: Asier Alonso Tejeda
    Participantes: Maialen González Jaio y Julen Cuadra Gomez
    Duración: 15 días

- **Tarea 4.5:** Implementación sensor de temperatura
  Descripción: El objetivo de esta fase es buscar un sensor de temperatura adecuado al proyecto, implementarlo en el brazo robótico WidowX y realizar todas las configuraciones necesarias.
  Responsable: Julen Cuadra Gomez
  Participantes: Asier Alonso Tejeda y Maialen González Jaio
  Duración: 12 días

- **Tarea 4.6:** Implementación XBees y Arduinos
  Descripción: Esta tarea consiste en implementar un sistema de configuración inalámbrica utilizando una pareja de XBees y dos Arduinos, configurar toda la cadena e implementarla en ROS.
  Responsable: Asier Alonso Tejeda
  Participantes: Maialen González Jaio y Julen Cuadra Gomez
  Duración: 10 días

- **Tarea 4.7:** Combinación de todos los equipos
  Descripción: Tarea para, una vez se han configurado todos los equipos, juntar las funcionalidades y hacer que estas interaccionen entre ellas en el entorno de ROS.
  Responsables: Asier Alonso Tejeda y Julen Cuadra Gomez
  Participantes: Maialen González Jaio
  Duración: 15 días

- **Fase 5:** Validación de resultados
  Descripción: Fase para comprobar que la implementación de todas las funcionalidades desarrolladas es la deseada, es decir, comprobación del cumplimiento de los objetivos.
  Responsable: Oskar Casquero Oyarzabal
  Participantes: Todos
  Duración: 4 días

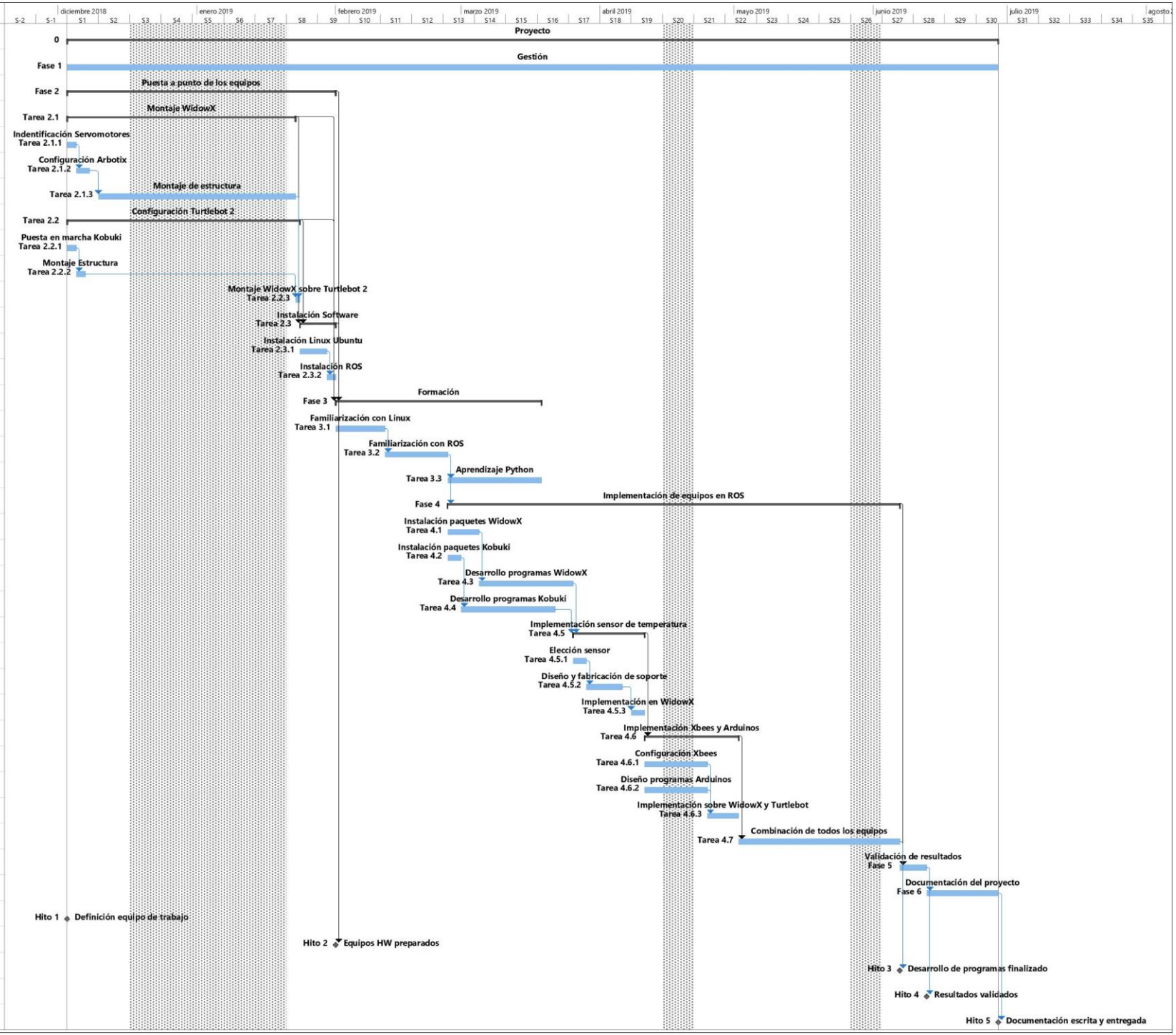
- **Fase 6:** Documentación del proyecto
  Descripción: Fase para documentar todo el desarrollo del proyecto, realizar el informe final y preparar su presentación.
  Responsable: Maialen González Jaio
  Participantes: Asier Alonso Tejeda y Julen Cuadra Gomez
  Duración: 12 días

Además de las fases y tareas descritas anteriormente se han definido varios hitos para evaluar el progreso del proyecto. Los hitos propuestos son los siguientes:

1. **Definición equipo de trabajo**
2. **Equipos HW preparados**
3. **Desarrollo de programas finalizado**
4. **Resultados validados**
5. **Documentación escrita y entregada**

A continuación, se muestra el diagrama de Gantt en el que se puede visualizar la planificación detallada en el apartado anterior. En este diagrama se pueden ver las diferentes fases de las que se compone el proyecto, las diferentes tareas en las que se dividen estas y su duración. Además, las franjas verticales que figuran en el diagrama representan periodos no laborales que han interferido con el desarrollo del proyecto. En la parte inferior del diagrama se pueden ver los diferentes hitos que se han planteado.

Id	EDT	Nombre de tarea	Duración
1	**0**	**Proyecto**	**109 días**
2	Fase 1	Gestión	109 días
3	**Fase 2**	**Puesta a punto de los equipos**	**19 días**
4	**Tarea 2.1**	**Montaje WidowX**	**12 días**
5	Tarea 2.1.1	Indentificación Servomotores	2 días
6	Tarea 2.1.2	Configuración Arbotix	3 días
7	Tarea 2.1.3	Montaje de estructura	7 días
8	**Tarea 2.2**	**Configuración Turtlebot 2**	**13 días**
9	Tarea 2.2.1	Puesta en marcha Kobuki	2 días
10	Tarea 2.2.2	Montaje Estructura	2 días
11	Tarea 2.2.3	Montaje WidowX sobre Turtlebot 2	1 día
12	**Tarea 2.3**	**Instalación Software**	**6 días**
13	Tarea 2.3.1	Instalación Linux Ubuntu	4 días
14	Tarea 2.3.2	Instalación ROS	2 días
15	**Fase 3**	**Formación**	**32 días**
16	Tarea 3.1	Familiarización con Linux	7 días
17	Tarea 3.2	Familiarización con ROS	10 días
18	Tarea 3.3	Aprendizaje Python	15 días
19	**Fase 4**	**Implementación de equipos en ROS**	**57 días**
20	Tarea 4.1	Instalación paquetes WidowX	5 días
21	Tarea 4.2	Instalación paquetes Kobuki	3 días
22	Tarea 4.3	Desarrollo programas WidowX	15 días
23	Tarea 4.4	Desarrollo programas Kobuki	15 días
24	**Tarea 4.5**	**Implementación sensor de temperatura**	**12 días**
25	Tarea 4.5.1	Elección sensor	3 días
26	Tarea 4.5.2	Diseño y fabricación de soporte	6 días
27	Tarea 4.5.3	Implementación en WidowX	3 días
28	**Tarea 4.6**	**Implementación Xbees y Arduinos**	**10 días**
29	Tarea 4.6.1	Configuración Xbees	5 días
30	Tarea 4.6.2	Diseño programas Arduinos	5 días
31	Tarea 4.6.3	Implementación sobre WidowX y Turtlebot	5 días
32	Tarea 4.7	Combinación de todos los equipos	15 días
33	Fase 5	Validación de resultados	4 días
34	Fase 6	Documentación del proyecto	12 días
35	Hito 1	Definición equipo de trabajo	0 días
36	Hito 2	Equipos HW preparados	0 días
37	Hito 3	Desarrollo de programas finalizado	0 días
38	Hito 4	Resultados validados	0 días
39	Hito 5	Documentación escrita y entregada	0 días



11 - Presupuesto

Para el desarrollo de este proyecto se diferencian dos presupuestos diferentes, el primero de ellos muestra el coste de desarrollar el proyecto de forma conceptual, es decir, el coste que tienen las horas que se requieren para detallar el procedimiento a llevar a cabo y el funcionamiento global del proyecto. Este presupuesto queda desglosado en la tabla 11.1.

El segundo presupuesto desglosa los costes de puesta en marcha del proyecto. Para esto hay que tener en cuenta, los costes de los diferentes equipos que se van a utilizar, el coste de los ingenieros que van a llevar a cabo el montaje y el desarrollo del proyecto y las diferentes amortizaciones de las máquinas que se utilizan a lo largo del proyecto. Este presupuesto queda desglosado en la tabla 11.2.

Tabla 11.1 Presupuesto Desarrollo Conceptual

CONCEPTO	UNIDADES	Nº DE UNIDADES	COSTE UNITARIO	COSTE	TOTAL
Horas internas					**9.520,00 €**
Director de Proyecto	h	22	35,00 €	770,00 €	
Ingenieros Junior	h	250	25,00 €	6.250,00 €	
Gestora	h	125	20,00 €	2.500,00 €	
Amortizaciones					**79,40 €**
Ordenador	h	397	0,20 €	79,40 €	
Costes Directos					9.599,40 €
Costes Indirectos	10%				959,94 €
Subtotal 1					10.559,34 €
Imprevistos	10%				1.055,93 €
TOTAL					**11.615,27 €**

Tabla 11.2 Presupuesto de Implantación

CONCEPTO	UNIDADES	Nº DE UNIDADES	COSTE UNITARIO	COSTE	TOTAL
Horas internas					**2.225,00 €**
Director de Proyecto	h	5	35,00 €	175,00 €	
Ingenieros Junior	h	50	25,00 €	1.250,00 €	
Gestora	h	25	20,00 €	500,00 €	
Técnico de Laboratorio	h	10	30,00 €	300,00 €	
Amortizaciones					**18,60 €**
Ordenador	h	60	0,20 €	12,00 €	
Máquina Para PCBs	h	4	0,15 €	0,60 €	
Impresora 3D	h	20	0,30 €	6,00 €	
Gastos					**3.650,91 €**
WidowX		2	1.512,75 €	3.025,50 €	
Kobuki		1	499,00 €	499,00 €	
Sensor MLX90614		1	17,75 €	17,75 €	
Xbee		2	50,00 €	100,00 €	
Arduino Nano		2	4,33 €	8,66 €	
Costes Directos					5.894,51 €
Costes Indirectos	10%				589,45 €
Subtotal 1					6.483,96 €
Imprevistos	10%				648,40 €
TOTAL					**7.132,36 €**

12 - Conclusiones

En este apartado se exponen las conclusiones que se extraen de la realización de este trabajo fin de grado.

La implementación de todas estas funcionalidades desde un entorno de software libre permite un prototipado rápido y cómodo. La posibilidad de integrar el control de dispositivos de diferentes fabricantes desde un punto de control centralizado es una ventaja clara para desarrollar proyectos de investigación. Como contrapunto, el uso de software desarrollado por otros usuarios y obtenido a través de repositorios online no asegura un correcto funcionamiento. A lo largo del desarrollo de este proyecto se han encontrado incompatibilidades con el código obtenido de esta forma y ha ralentizado en ciertos momentos el desarrollo del proyecto. Es necesario contar con personal que domine, en mayor o menor medida, la interacción con el sistema operativo Linux ya que es la base para poder interactuar con ROS. Por otro lado, la posibilidad de acceder y escribir código propio otorga una gran libertad al personal que desarrolle el proyecto. Es posible basarse en el código ya desarrollado por otros usuarios (en su mayoría investigadores) para añadir aplicaciones concretas como se ha mostrado en este proyecto.

Si atentemos al hardware utilizado en este trabajo fin de grado, el uso de dispositivos enfocados a la investigación y a la formación es muy adecuado para este tipo de proyectos. Estos dispositivos permiten una cómoda implementación en el entorno de ROS sin barreras introducidas por la falta de compatibilidad del software desarrollado por los distintos fabricantes de robots. El precio reducido de estos equipos facilita también su adquisición para realizar prototipos. Como inconveniente de estos equipos, para ciertas tareas carecen de la potencia o robustez necesaria, pero esto es un precio a pagar para obtener equipos de tamaño reducido con estas capacidades.

Además del trabajo expuesto en este documento y todo lo desarrollado en este proyecto, se pretende seguir ampliando estos equipos e ir añadiéndoles funcionalidades. Entre estas podemos destacar la incorporación un sensor de color para aumentar las posibilidades de la toma de decisiones del brazo WidowX y la implementación de unas placas de circuito impreso para reducir el impacto visual y el espacio que ocupan los módulos XBee y los controladores Arduino Nano.

BIBLIOGRAFÍA

[1] O. Robotics, «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/es. [Último acceso: 15 Febrero 2019].

[2] G. Alfonso Aberasturi y A. Armentia, *Puesta en marcha del control de robots de transporte.*

[3] A. Cruz Martín, «ResearchGate,» Enero 2016. [En línea]. Available: https://www.researchgate.net/publication/330216268_Turtlebot_2_con_brazo_WidowX_primeros_pasos. [Último acceso: 20 Marzo 2019].

[4] M. Marcos, B. Fortes, O. Casquero y J. Martin, «ResearchGate,» 2018. [En línea]. Available: https://www.researchgate.net/publication/329907421_A_Generic_Multi-Layer_Architecture_Based_on_ROS-JADE_Integration_for_Autonomous_Transport_Vehicles. [Último acceso: 15 Junio 2019].

[5] T. Robotics, «Trossen Robotics,» [En línea]. Available: https://www.trossenrobotics.com/widowxrobotarm. [Último acceso: 20 Febrero 2019].

[6] O. Robotics, «ROS Answers,» [En línea]. Available: https://answers.ros.org/questions/.

[7] C. F. Mora, *Robotización y transformación del empleo,* Barcelona, 2018.

[8] Robotis, 2019. [En línea]. Available: http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/. [Último acceso: 25 junio 2019].

ANEXO I: Manual de Usuario

En este manual se va a explicar cómo poner en marcha el manipulador móvil para poder probar todo lo explicado en este trabajo fin de grado.

Este manual da por hecho que se dispone de una instalación de Linux Ubuntu sobre la que se corre ROS Kinetic. Además, los paquetes correspondientes al controlador del brazo, los paquetes para controlar la base Kobuki y el paquete desarrollado por el grupo de trabajo que puede encontrarse en:

https://github.com/JulenCuadra/TFG_Kobuki_WidowX.

Para lanzar todos los comandos necesarios, se va a utilizar el programa Terminator ya que permite trabajar de forma cómoda con más de una terminal en Linux.

En cada terminal que se abra en la que se quiera conectar con el Odroid hay que hacer lo siguiente:

- Primero conectarnos vía protocolo SSH a la placa Odroid para ello introducimos el comando:

  ssh root@192.168.1.108

  Como contraseña introducimos: odroid
  Una vez introducido esto deberíamos obtener una salida en la terminal como la siguiente:

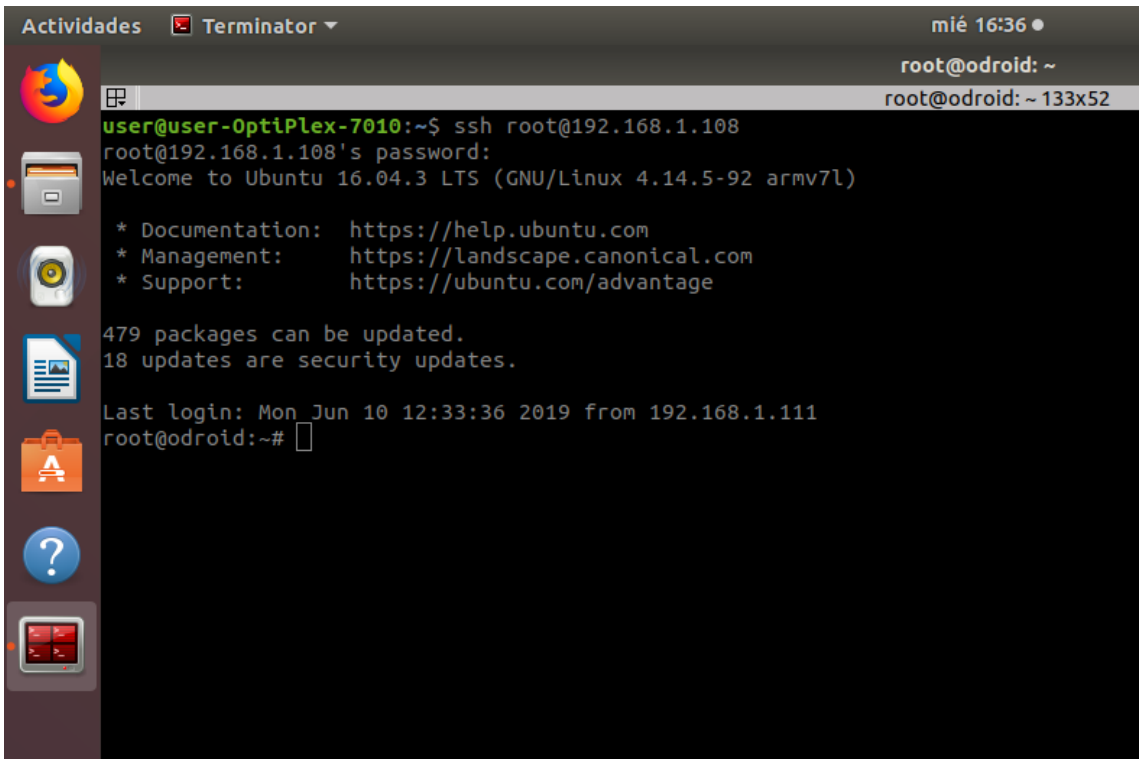


Figura 1 Conexión vía SSH a Odroid

- Una vez conectados a la placa Odroid tenemos que añadir nuestro espacio de trabajo a ROS, para ello nos colocamos en la carpeta de este espacio de trabajo:

cd /home/odroid/catkin_ws/

En esta carpeta ejecutamos:

source ./devel/setup.bash

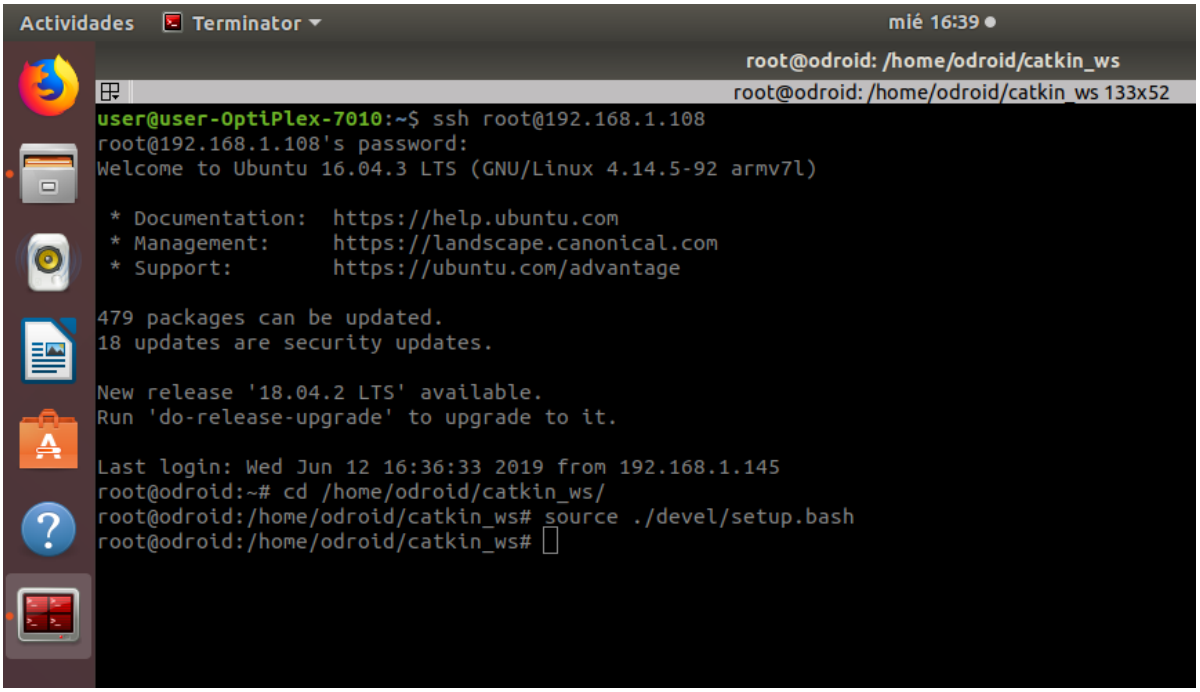


Figura 2 Configuración entorno de ROS

Recordar que en cada ventana nueva que se abra, cada nueva terminal, debe conectarse a la Odroid y ejecutar estos comandos, de lo contrario, no se podrán ejecutar correctamente los siguientes pasos.
Ahora, sin movernos de esta carpeta, ejecutamos el comando:

catkin_make

Con este comando nos aseguramos de que los paquetes están bien instalados y no existen problemas de incompatibilidades o paquetes mal instalados.

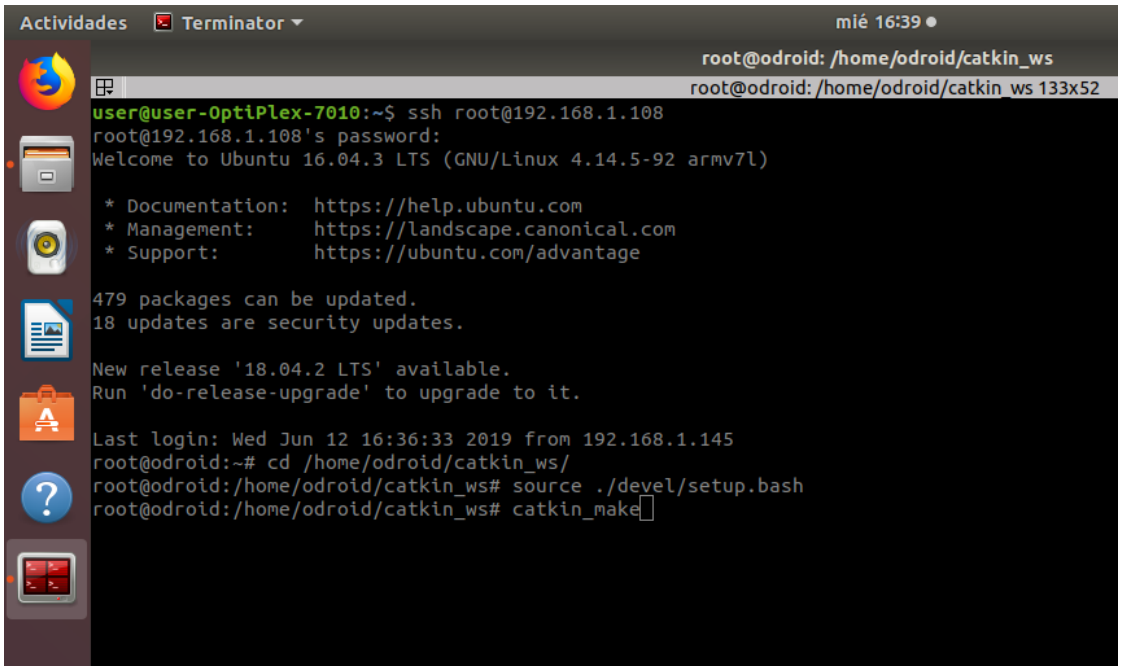


Figura 3 Construcción y Compilación de Paquetes

El siguiente paso es lanzar los dos controladores que se van a utilizar, el de la base y el del brazo. Primero vamos a lanzar el controlador de la base Kobuki, para ello ejecutamos el comando:

roslaunch kobuki_node minimal.launch

El resultado de la consola lo podemos ver en la Figura 4.

Para poder habilitar el control del brazo ejecutamos el siguiente comando:

roslaunch widowx_arm_controller widowx_arm_controller.launch

El resultado de la consola lo podemos ver en la Figura 5.

Una vez lanzados los dos controladores, podemos lanzar el nodo que gestiona el sensor de temperatura, para ello:

rosrun paquete_kobuki lector_sensor.py

Lo que provocará que se imprima en la pantalla la información que publica el sensor (Figura 6).

Por último, ejecutamos en una nueva ventana nuestro programa (Figura 7):

rosrun paquete_kobuki sensores_kobuki.py

Con todo esto ejecutándose la situación debería de ser la de la Figura 8.

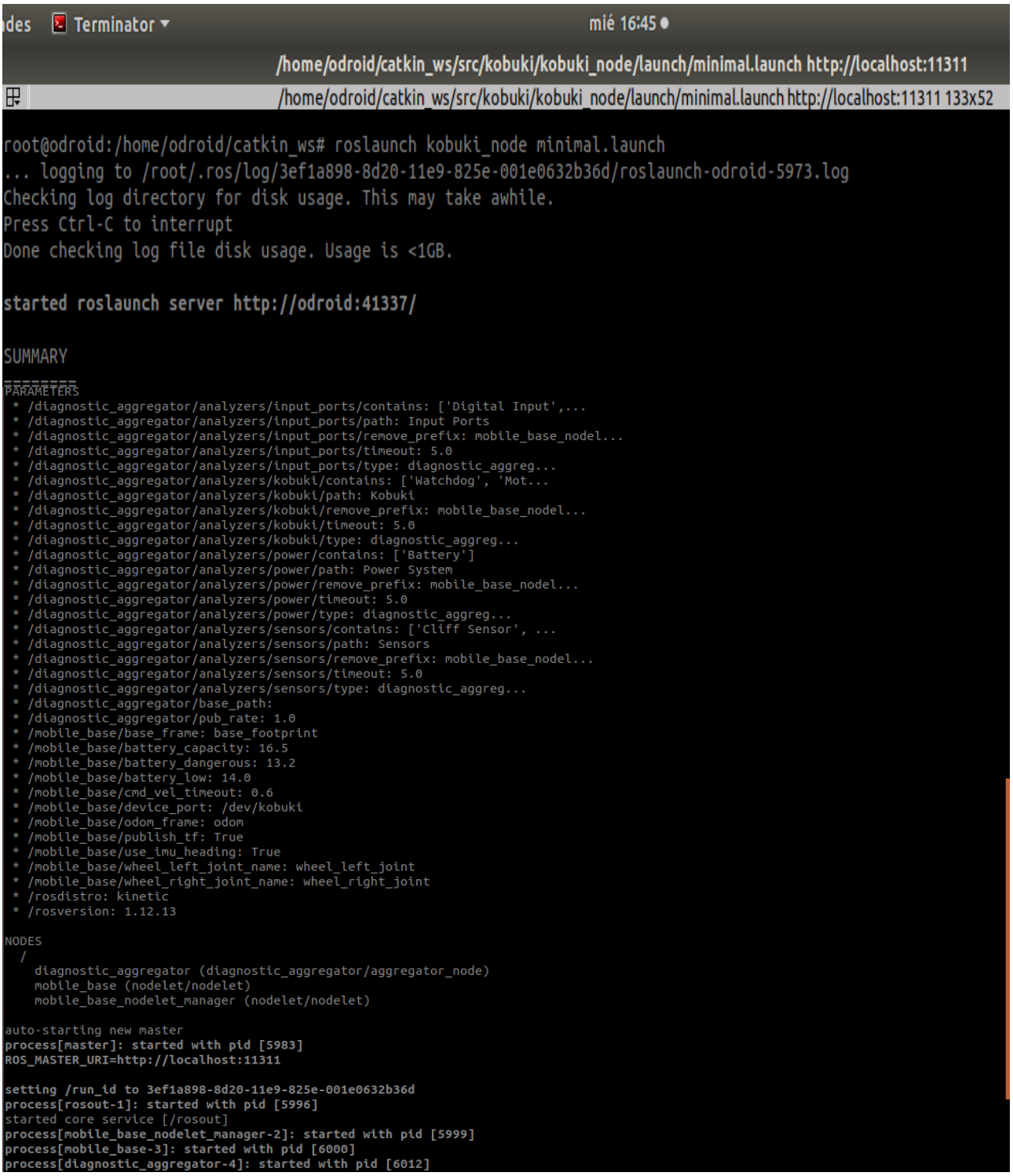


Figura 4 Lanzamiento Controlador Kobuki

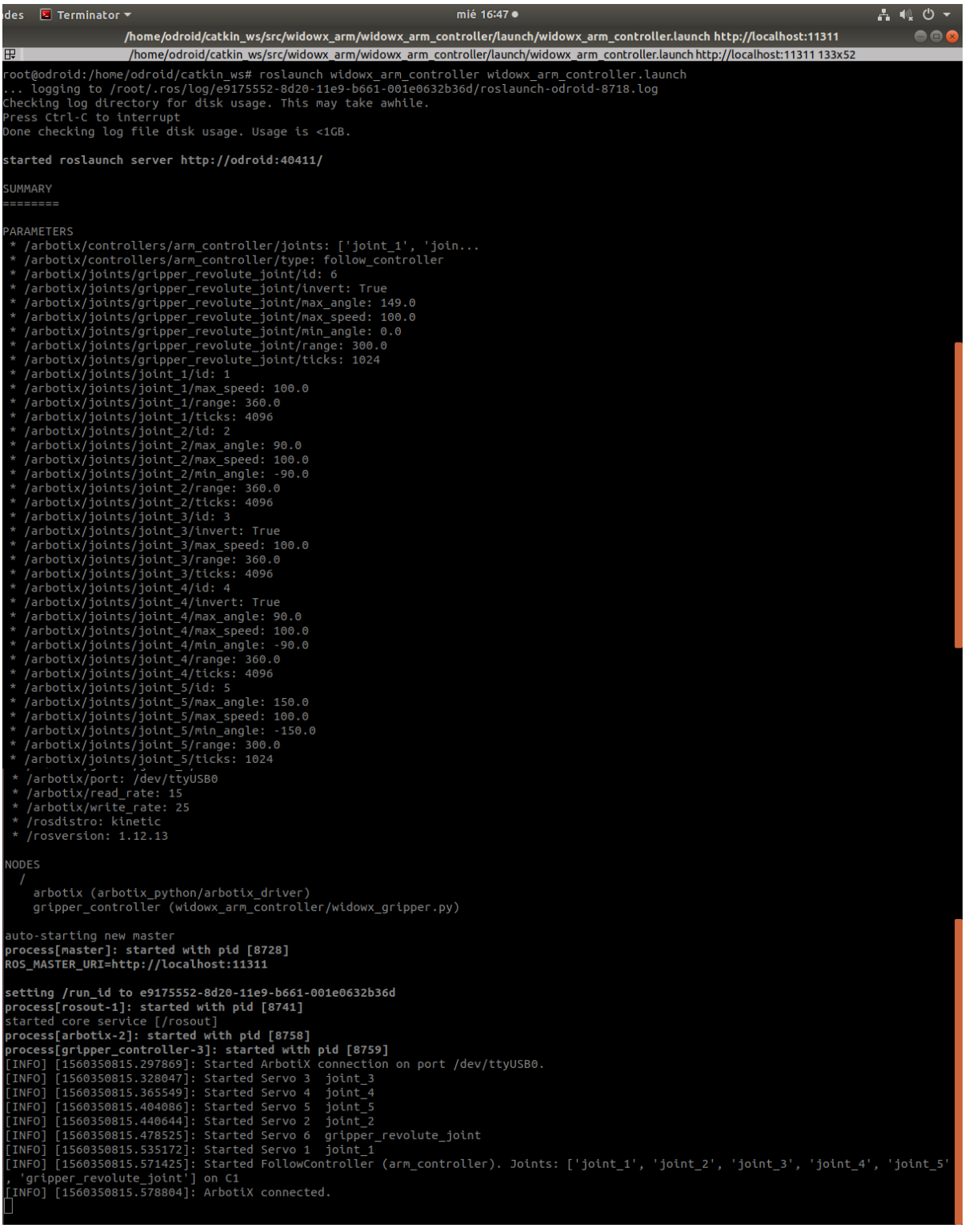


Figura 5 Lanzamiento Controlador WidowX

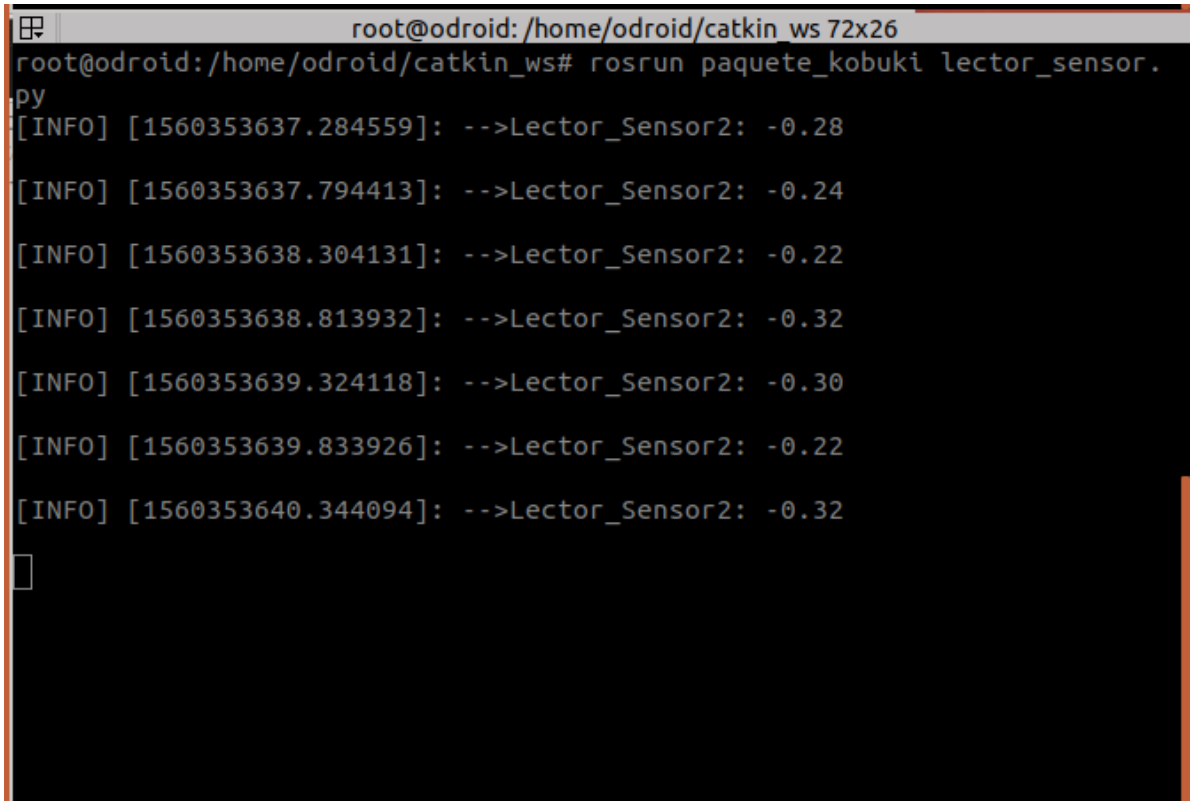


Figura 6 Lanzamiento del Lector del Sensor

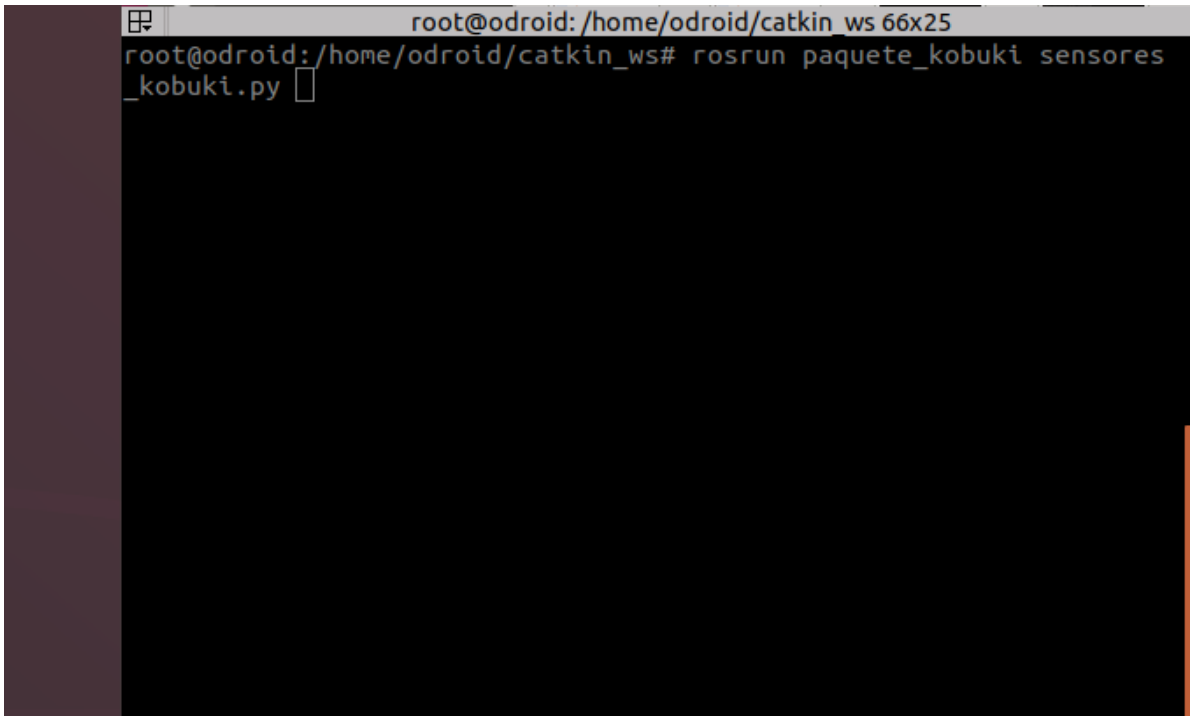


Figura 7 Lanzamiento programa final

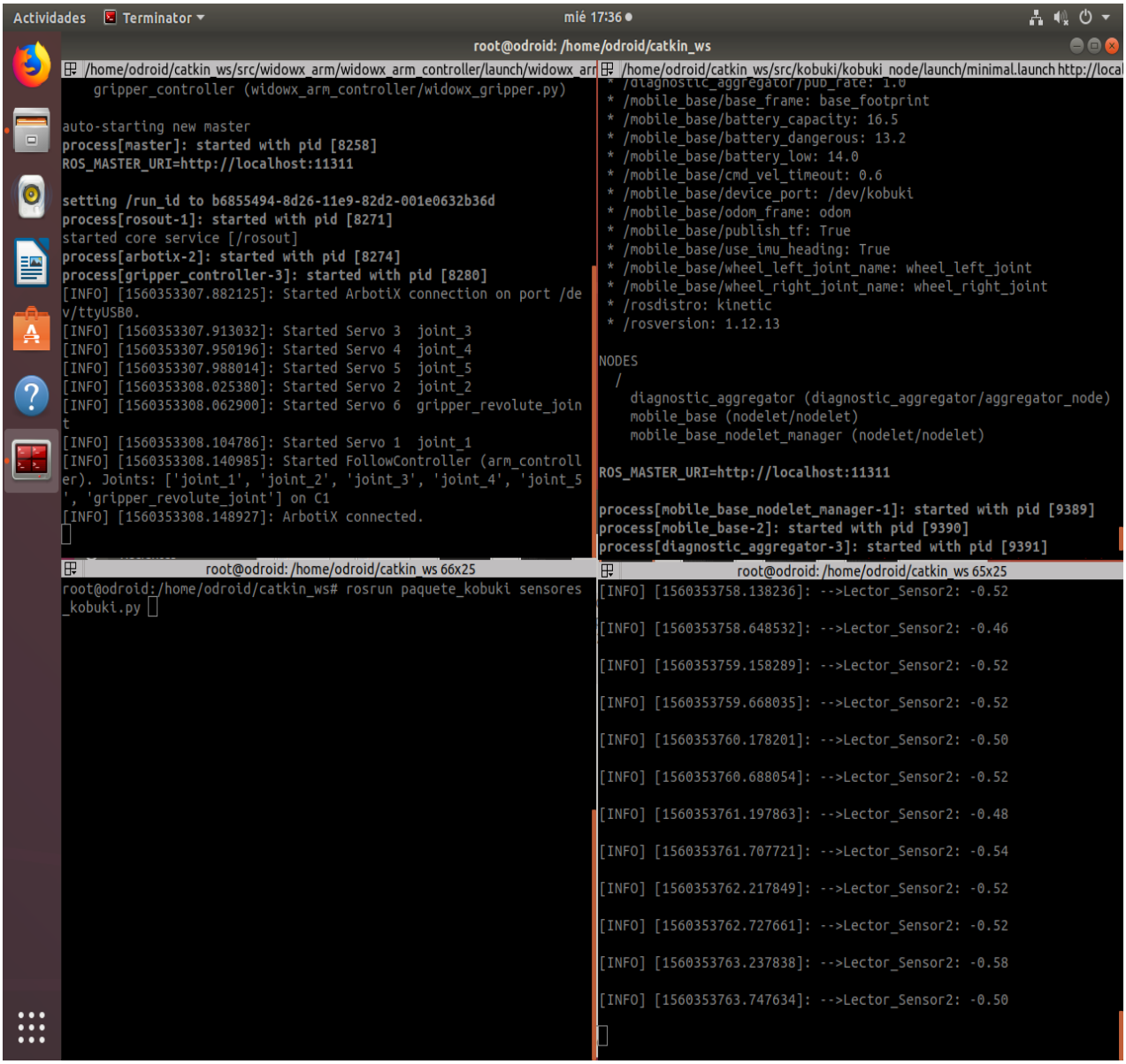


Figura 8 Situación final con todos los controladores, el lector del sensor y el programa propio