Teleoperación de un brazo robot mediante el sensor Kinect

Autor: Pablo Melero Cazorla

Director: José Carlos Moreno Úbeda

14 de septiembre de 2014

Dedicado a todos mis amigos,
y a toda la gente que se interesó
y me ayudó con el proyecto.

Contenido

Índice de Figuras

Índice de Tablas

Capítulo I.

I. Interés y objetivos

1. Interés

La teleoperación surgió del interés para aumentar la capacidad de manipulación del hombre, uno de los objetivos iniciales era la de aumentar el alcance de manipulación mediante el uso de herramientas para alcanzar objetos lejanos. En la actualidad, este término se ha usado para la manipulación de objetos en entornos de trabajo de difícil acceso para el hombre, o entornos de trabajo en los que no se desea interferir en el entorno del objeto a manejar debido al alto riesgo, para ello se han desarrollado equipos sensorizados que permiten realizar estas operaciones de manera remota.

El término telerrobótica surge como la aplicación de la teleoperación a robots, se pueden identificar: robots manipuladores, robots móviles o incluso vehículos aéreos no tripulados (UAVs). Sus aplicaciones son diversas y abarcan todo tipo de campos, desde aplicaciones militares para la desactivación de bombas o detección de minas, pasando por el tratamiento de sustancias radioactivas, hasta modernos equipos para realizar operaciones quirúrgicas con gran precisión.

La teleoperación maestro-esclavo es el concepto que se va a tratar durante este proyecto. Hace referencia a la réplica del movimiento de un esclavo mediante el movimiento del manipulador, considerado como maestro.

El entorno de programación ROS es una gran herramienta usada para el control de toda clase de robots. Se trata de una plataforma de programación capaz de proveer de servicios estándar de un sistema operativo, tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes.

2. Objetivos

En el presente proyecto se pretende el control maestro-esclavo de un brazo robótico, considerado esclavo, a través del brazo real de un usuario, denominado maestro. Para ello se hará uso del sensor Kinect, que monitorizara los movimientos del brazo del usuario que serán procesados y enviados a través de ROS al brazo robot. Se realizarán ensayos en simulación de cada uno de los programas creados y se diseñará una interfaz gráfica para la puesta a punto del sistema, posteriormente se realizarán los ensayos sobre el robot real.

Por lo tanto, los objetivos de este proyecto se pueden resumir en los siguientes puntos:

- Puesta a punto del entorno de trabajo. Incluye la puesta en marcha del brazo manipulador, de la unidad de procesado junto a ROS y del sensor Kinect.
- Obtención y procesados de las articulaciones del usuario mediante el sensor infrarrojo de Kinect.
- Desarrollo de un paquete en ROS que permita el control directo del brazo robot a partir de las coordenadas obtenidas por Kinect. Para ello, se realizara un programa en Python en el entorno ROS capaz de comunicarse con Kinect y realizar un procesado de los datos recibidos para su posterior envío al robot real o simulado.
- Simulación y control del brazo robótico mediante la herramienta de simulación Gazebo para la comprobación del programa.
- Implementación del programa en el robot real.
- Desarrollo de una interfaz gráfica básica para la operación del sistema.

Capítulo II.

II. Revisión bibliográfica

3. Sistemas teleoperados

La teleoperación hace referencia a la operación de una máquina a distancia. También se puede entender como "control remoto", debido a su similar significado, aunque el término teleoperación se aplica con mayor frecuencia en el ámbito de la investigación, en entornos académicos y técnicos. Por ello, se asocia con mayor frecuencia a la robótica y a los robots móviles, aunque es posible su aplicación en una amplia gama circunstancias en las que un dispositivo o máquina es operada por una persona desde una distancia [12].

La Figura II-1 muestra el flujo de información común en un sistema teleoperado.

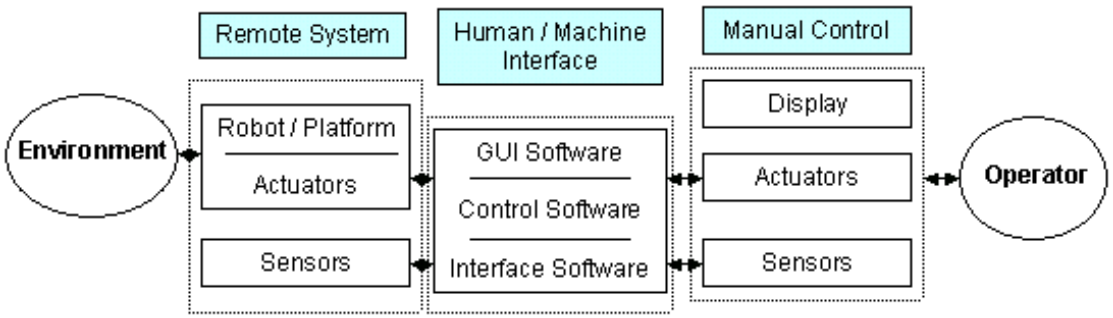


Figura II-1. Flujo de información en un sistema teleoperado [38]

La función principal de un sistema teleoperado es la de asistir al operador a realizar y llevar a cabo tareas complejas en entornos peligrosos o de difícil acceso, como el espacio, plantas nucleares, entornos militares, vigilancia, operaciones quirúrgicas y la realización de tareas en las profundidades marinas [51].

3.1. Definiciones

Para poder entender la teleoperación de manera clara, es necesaria la definición y aclaración de algunos de los conceptos básicos referentes a la teleoperación.

- Teleoperacion, conjunto de tecnologías que comprenden la operación o gobierno a distancia de un dispositivo por un ser humano. Por tanto, teleoperar es la acción que realiza un ser humano de operar o gobernar a distancia un dispositivo; mientras que un sistema de teleoperación será aquel que permita teleoperar un dispositivo, que se denominará dispositivo teleoperado [34].

- Teleoperación maestro-esclavo. Los sistema de teleoperación por lo general están formados por dos robot manipuladores que están conectados de tal manera que permite al humano operador controlar uno de los manipuladores, el cual es llamado brazo maestro, con el fin de generar acciones correlativas en el manipulador remoto, que es llamado brazo esclavo. Dentro de este campo destacan los esqueletos asistenciales, conocidos como HAL (*Hybrid Assistive Limb*), se tratan de estructuras robóticas, también llamados exoesqueletos, capaces de dotar al portador de mayor capacidad de movimiento. En la Figura II-2 se puede observar un robot teleoperado maestro-esclavo de tipo HAL de la compañía japonesa Cyberdyne Corp.

Figura II-2. Teleoperación de un robot maestro-esclavo HAL de Cyberdyne Corp

- Telerrobótica, se denomina al conjunto de tecnologías que comprenden la monitorización y reprogramación a distancia de un robot por un ser humano, se habla entonces de la teleoperación de un robot, que se denominará telerrobot o robot teleoperado.

- Telemanipulación es un sistema en el que un brazo de robot esclavo que está situado por lo general en un entorno remoto o ambiente peligroso, rastrea el movimiento de un manipulador maestro. En general, es telemanipulación dividido en dos procesos fuertemente acoplados: la interacción entre el operador y el dispositivo maestro y la interacción entre el control remoto del dispositivo esclavo y su entorno [1].

- Telepresencia. En 1987 Thomas B. Sheridan, profesor de Ingeniería Mecánica del Instituto Tecnológico de Massachusetts pionero en robótica y control remoto describió la telepresencia como "el sistema ideal de transmisión de información de manera natural de modo que el usuario sienta estar físicamente presente en el sitio remoto", o como una "ilusión convincente" y "una sensación subjetiva" [5].

En el campo de la robótica, la telepresencia generalmente se refiere al control remoto de un sistema que combina el uso la visión por ordenador, tratamiento de imagen y realidad virtual. En la Figura II-3 se puede ver un ejemplo de un robot de este tipo, de la compañía iRobot.

Figura II-3 iRobot AVA 500

3.2. Elementos

Un sistema teleoperado está constituido por una serie de elementos, como se muestra en la Figura II-4.

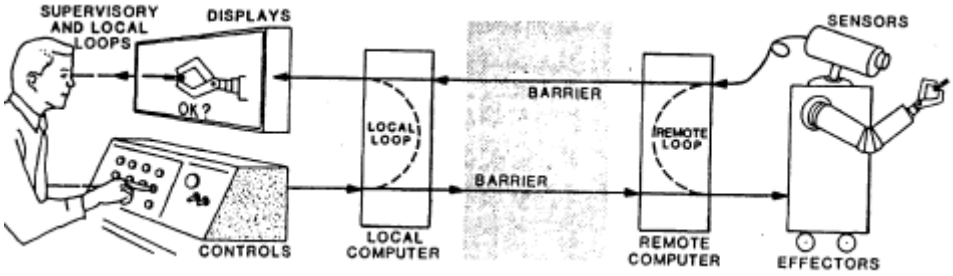


Figura II-4. Esquema de un sistema de teleoperación y sus elementos

Un sistema teleoperado se compone de los siguientes elementos:

- Operador, es el ser humano que realiza una serie de acciones en un entorno remoto, dicha acción puede ser el control de un joystick, o el envío de información a través de pulsadores en una estación de teleoperación.

- Teleoperador es una máquina que permite a un ser humano operador moverse, sentir y manipular objetos mecánicamente a distancia. Esta máquina realiza la acción de comunicar al robot con el humano a través de un medio de comunicación físico o inalámbrico.

- Interfaz, se define como el conjunto de dispositivos hardware que dan al usuario información del entorno donde se encuentra el robot a manipular y del propio robot, así como la capacidad de realizar acciones en el mismo. Se pueden incluir, pantallas, teclados, joysticks, etc.

- Telerobot, es el elemento final de la teleoperación. En la cual el robot acepta instrucciones desde la distancia, generalmente de un humano operador que realiza acciones en tiempo real en un entorno distante a través del uso de sensores u otros mecanismos de control. Por lo general tiene sensores y actuadores para la manipulación y proporcionar movilidad al robot teleoperado, además de un medio por el cual se realiza la comunicación entre ambos.

- Canal de comunicación. Es el medio por el cual circula la información que envía el operador y su retroalimentación, este medio puede ser físico, mediante cableado, o inalámbrico, por el aire. Los mensajes se codifican a través de un protocolo de comunicación que varía según el medio y el tipo de mensaje. En la Figura II-5 se muestra un esquema de teleoperación maestro-esclavo a través de internet.

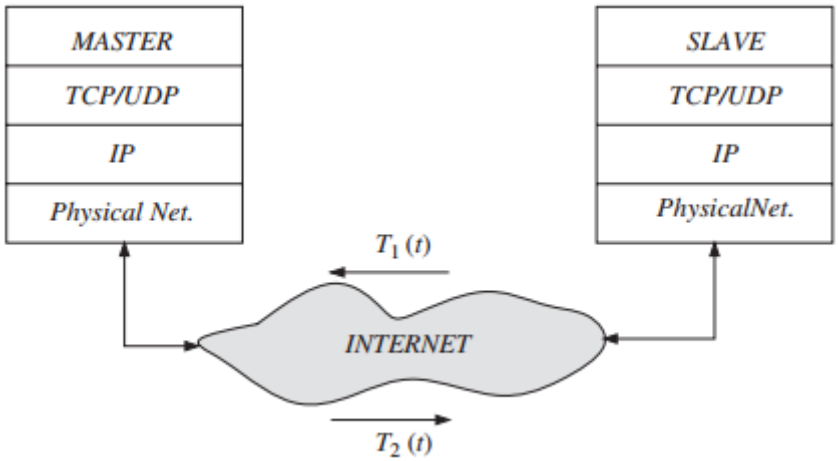


Figura II-5. Teleoperación a través de internet

- Sensores. Un sensor es un dispositivo capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y transformarlas en variables eléctricas. Las variables de instrumentación pueden ser por ejemplo: temperatura, intensidad lumínica, distancia, aceleración, inclinación, desplazamiento, presión, fuerza, torsión, humedad, movimiento, pH, etc. [54]. Estas variables proporcionan la información necesaria del entorno remoto donde se encuentra el robot, o son usadas por el usuario para mandar información a este.

3.3. Evolución histórica

El origen de la teleoperación puede datarse en 1803 con la invención del polígrafo por John Isaac Hawkinsen. Se trataba de un dispositivo mecánico capaz de producir una copia simultánea de un escrito a partir del original, usando una pluma y un folio. Fue el más famoso usado por Thomas Jefferson que adquirió el primer polígrafo en 1804 [41], en la Figura II-6 se muestra una réplica de este polígrafo.

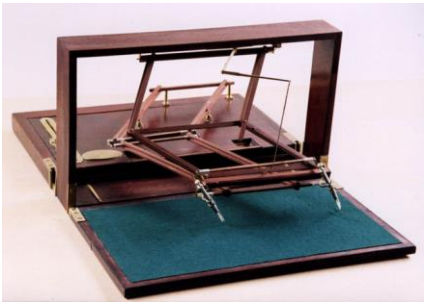


Figura II-6. Reproducción de un polígrafo de Jefferson del Instituto Smithsoniano

Por otra parte, la historia de la telerobótica se remonta a los principios de la radio comunicación, Nikola Tesla puede decirse que fue la primera persona en realizar el primer sistema teleoperado, a mediados de 1898 publicó la patente *US 613809 A*, que tenía como título original "Method of and Apparatus for Controlling Mechanism of Moving Vessels or Vehicles" [32]. Consiste en un bote controlado mediante radio frecuencia en una estación remota, en la Figura II-7 se puede ver un esquema del mismo.

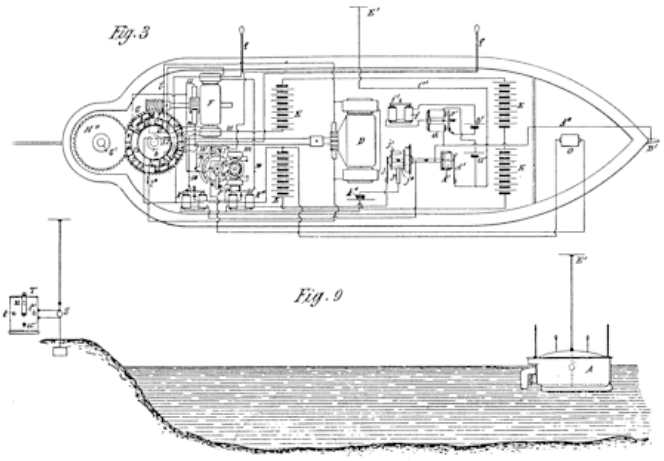


Figura II-7. Esquema del bote de la patente de N. Tesla "*Method of and Apparatus for Controlling Mechanism of Moving Vessels or Vehicles*"

En 1947 comenzaron las primeras investigaciones, lideradas por Raymond Goertz del Argonne National Laboratory en Estados Unidos, encaminadas al desarrollo de algún tipo de manipulador de fácil manejo a distancia mediante el uso por parte del operador de otro manipulador equivalente. El primer manipulador remoto maestro-esclavo se desarrolló en 1948, la empresa Central Research Laboratories fue contratada para fabricar un manipulador remoto para el Laboratorio Nacional

de Argone. La intención era sustituir los dispositivos que manipulan materiales altamente radiactivos desde una cámara sellada o celda caliente, con un mecanismo que operase a través de la pared lateral de la cámara, lo que permitía al investigador un fácil acceso al material a tratar. El resultado fue el manipulador maestro-esclavo Mk. 8 o MSM-8 (Figura II-8).

Figura II-8. Manipulador remoto MSM-8

A principios de los años 50 se comenzaron los desarrollos encaminados a motorizar ambos manipuladores, maestro y esclavo, de una forma adecuada. Fue en 1954 cuando Goertz presentó el primer manipulador maestro-esclavo con accionamiento eléctrico y servocontrol en ambos manipuladores llamado E1 (Figura II-9).

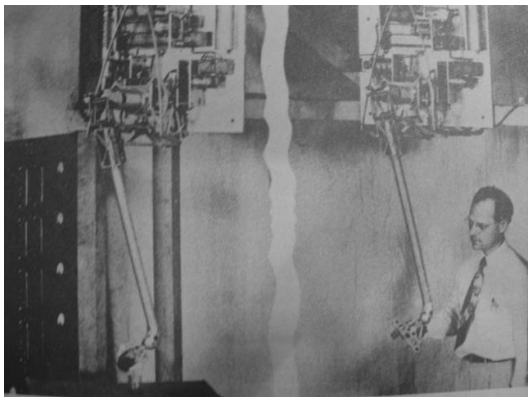


Figura II-9. Primer manipulador maestro-esclavo eléctrico

A mediados de los años sesenta, se realizaron estudios para la aplicación de sistemas teleoperados en tareas submarinas, como la exploración y extracción de petróleo, incluyendo sistemas más sofisticados, como la visión remota a partir de cámaras inalámbricas.

A finales de los años sesenta, aparecieron las primeras aplicaciones espaciales, en esta etapa el problema del retraso temporal en la comunicación jugó un papel muy importante. Algunos de los logros obtenidos durante esta época fueron los siguientes:

- 1967: Surveyor III aterrizó en la superficie de la Luna (unos segundos retraso en la transmisión a la tierra para comandos e información).
- 1976: Viking spacecraft aterrizó en Marte, estaba programado para llevar a cabo operaciones estrictamente automatizados.
- Shuttle Remote Manipulator System (SRMS): recupera satélites y los coloca en la bodega de carga de la nave; plataforma de trabajo móvil para los astronautas durante las salidas espaciales.

El desarrollo de nuevas tecnologías en la década de los noventa tuvo una respuesta directa en la telerrobótica, aumentando las capacidades de autonomía, una mejora en el tiempo de respuesta durante la comunicación y un aumento de las distancias entre los elementos gracias a las mejoras en los sistemas de comunicación: Internet, fibra óptica y radio.

3.4. Aplicaciones de la teleoperación

En las últimas dos décadas se han ido desarrollando sistemas teleoperados para la ejecución de tareas en entornos remotos o peligrosos, en una variedad de campos de aplicación desde el espacio hasta los fondos marinos, plantas nucleares, entornos militares, vigilancia, etc. [37, 51].

Con el desarrollo de Internet y otras tecnologías relacionadas, la aplicación de la teleoperación se convierte en un campo de estudio mucho más amplio y más indispensable que antes. A continuación se enumeran los campos de aplicación más representativos de un sistema teleoperado.

3.4.1. Aplicaciones en el espacio

La teleoperación ha sido usada en aplicaciones espaciales frecuentemente. La mayoría de las sondas espaciales han sido telerrobots, que incluían la posibilidad de ser controlados de manera sencilla, pero fiable, teniendo la capacidad de enviar imágenes y otros datos sensoriales, y la capacidad de ser reprogramadas en el espacio. Un ejemplo es el robot Curiosity (Figura II-10), un vehículo explorador usado por la NASA para misiones de investigación.

La aplicación de técnicas de teleoperación en el espacio para la manipulación remota o el envío de información de una superficie terrestre posee las siguientes ventajas:

- Seguridad. Al tratarse de operaciones de alto riesgo, que en algunos casos podrían provocar la muerte del astronauta, ya que se trabaja en entornos peligrosos.

- Costo. El coste del equipo necesario y los medios necesarios para mandar a un humano son mucho mayores a los de un sistema de teleoperación.

- Tiempo. Las misiones en muchos casos requieren años de investigación en el sitio remoto, para ello es necesario que la tripulación no sea humana. La autonomía para la movilidad del telerrobot es obtenida mediante paneles solares.

Figura II-10. Vehículo explorador Curiosity [13]

3.4.2. Aplicaciones en la industria nuclear

Su uso ha sido el más extendido desde los principios de la teleoperación, surgió de la necesidad de tratar y manipular sustancias radiactivas, así como de poder

moverse por entornos contaminados, sin hacer peligrar la salud de un operario. Algunos de sus principales usos son: la realización de tareas de manipulación con sustancias radioactivas, la limpieza de residuos tóxicos, operaciones de mantenimiento, descontaminación de instalaciones, etc. Uno de los ejemplos para la realización de tareas en entornos radiactivos es el del iRobot PackBot [26], robot teleoperado usado para sellar el tanque de contención del interior de uno de los reactores, tras el accidente nuclear en Fukushima (Figura II-11).

Figura II-11. iRobot PackBot. En la imagen de la derecha se encuentra el robot dentro del área 3 del interior de la planta

3.4.3. Aplicaciones submarinas

Para este tipo de telerrobots, se usa la expresión ROVs (*Marine remotely operated vehicles*), son usados para trabajar a grandes profundidades o entornos peligrosos, donde el ser humano podría correr algún tipo de riesgo. De entre sus funciones, se pueden destacar las siguientes: reparación y mantenimiento de plataformas petrolíferas en alta mar y estudios geológicos.

Para una de las exploraciones del Titanic de se usó el ROV Hercules [23] (Figura II-12). Construido para el Institute For Exploration (IFE), Hércules está equipado con características especiales que le permiten realizar tareas complejas a profundidades de 4.000 metros.

Aunque Hércules fue diseñado principalmente para estudiar y recuperar artefactos de naufragios antiguos, es también muy adecuado para estudiar la biología y la geología en las profundidades del mar.

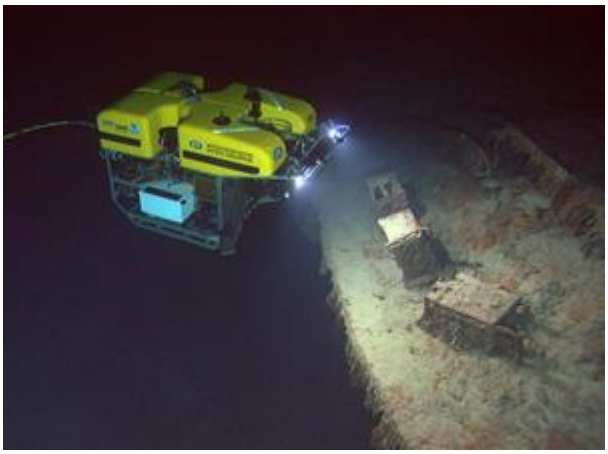


Figura II-12. ROV Hércules investigando la popa del Titanic durante una expedición en 2004

3.4.4. Aplicaciones militares

Esta área provee muchas posibilidades para los sistemas teleoperados, las tecnologías aquí usadas van desde sistemas de monitorización remota [16], hasta el uso de UAV (*Unmanned Air Vehicles*).

Los UAVs tienen un campo de aplicación muy grande, vigilancia, adquisición de objetivos militares, detección de enemigos, reconocimiento, etc. Hoy en día gracias a las nuevas tecnologías este tipo de vehículos se vuelven cada vez más inteligentes y autónomos, de modo que la intervención de un operador pasa cada a un segundo plano. Un ejemplo es el del US Air Force Predator (Figura II-13).

Figura II-13. US Air Force Predator [21]

Otra clase de vehículos de este tipo son los terrestres, llamados UGVs (*Unmanned Ground Vehicles*). Generalmente, el vehículo tendrá un conjunto de sensores para

observar el medio ambiente, tomar decisiones sobre su comportamiento de forma autónoma o pasar la información a un operador humano en un lugar remoto donde un operario controlaría el vehículo mediante teleoperación. Un ejemplo es el del Rush Demostrator (Figura II-14), usado en operaciones militares. Algunas de las tareas llevadas a cabo coinciden con la de los UAVs, para el mantenimiento de la paz, operaciones de vigilancia terrestre, presencia urbana de la calle. Además, los UGVs están siendo utilizados en misiones de rescate y recuperación.

Figura II-14. Rush Demonstrator UGV, Singapur [55]

3.4.5. Aplicaciones médicas

En la actualidad una importante aplicación de las tecnologías de teleoperación ha sido dentro del sector médico, desde el control y asistencia de pacientes mediante telepresencia, hasta los más novedosos y precisos sistemas teleoperados para la realización de complejas operaciones quirúrgicas, como en el caso del sistema quirúrgico Da Vinci (Figura II-15). Las principales ventajas de la teleoperación en el sector médico son las siguientes:

- La posibilidad del control remoto y supervisión de pacientes, haciendo que el médico no se tenga que desplazar hasta el hospital o incluso poder atender a pacientes alrededor de todo el mundo mediante telepresencia.

- En operaciones quirúrgicas, dotar al cirujano de mayor precisión y control, a partir de una cirugía asistida.

Figura II-15. Sistema Quirúrgico Da Vinci [14]

El esquema de control para un sistema quirúrgico se compone los elementos mostrados en la Figura II-16.

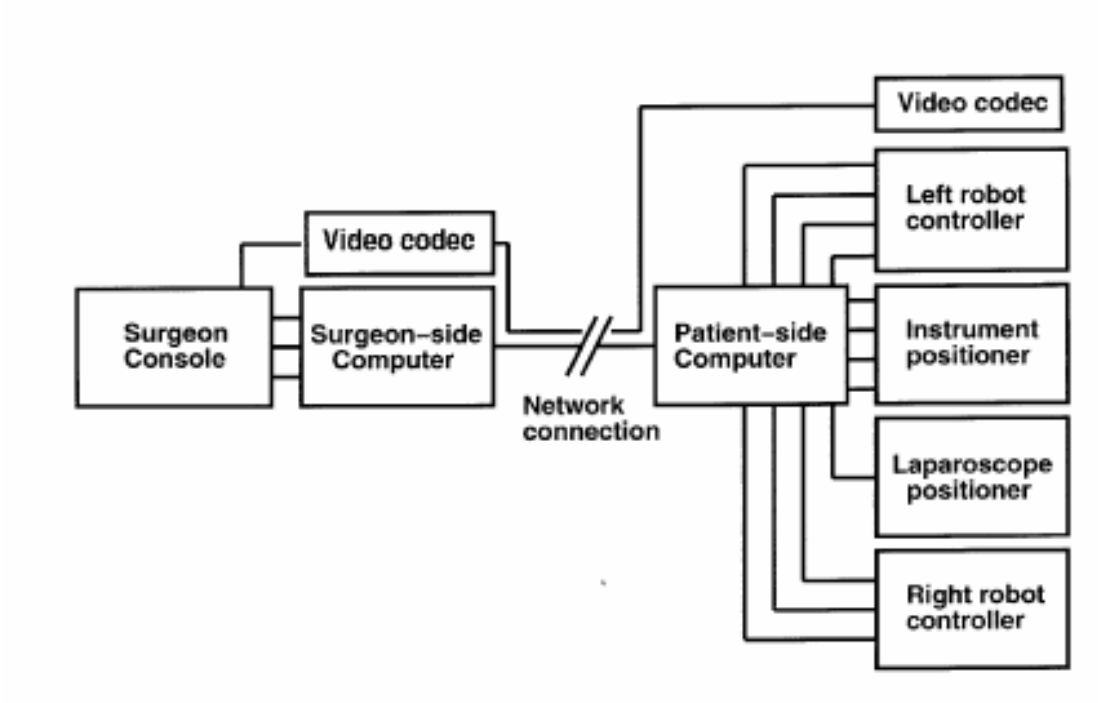


Figura II-16. Esquema de control de un sistema quirúrgico teleoperado

3.4.6. Futuras aplicaciones

En la actualidad se pueden encontrar otras aplicaciones en auge, como el desarrollo de prótesis robóticas en humanos, gracias al avance tecnológico es posible incorporar todo el sistema electrónico en un espacio cada vez más pequeño.

Con el desarrollo de nuevas tecnologías, los sistemas teleoperados podrán actuar en multitud de diferentes entornos, o en espacios cada vez más reducidos. Aunque el principal problema de la teleoperación es el retardo de las señales entre dos puntos lejanos, por ello se están investigando nuevos métodos de comunicación que reduzcan el retardo de la señal y hacer posible una comunicación fluida, necesaria en la mayoría de aplicaciones. En la Figura II-17 se muestra un ejemplo de retardo en la comunicación entre dos sistemas distanciados 4 Km, los retardos oscilaban entre los 0,1 segundos con máximos puntuales de 0,2 segundos, y eran causados por el direccionamiento variable de los paquetes a través de internet.

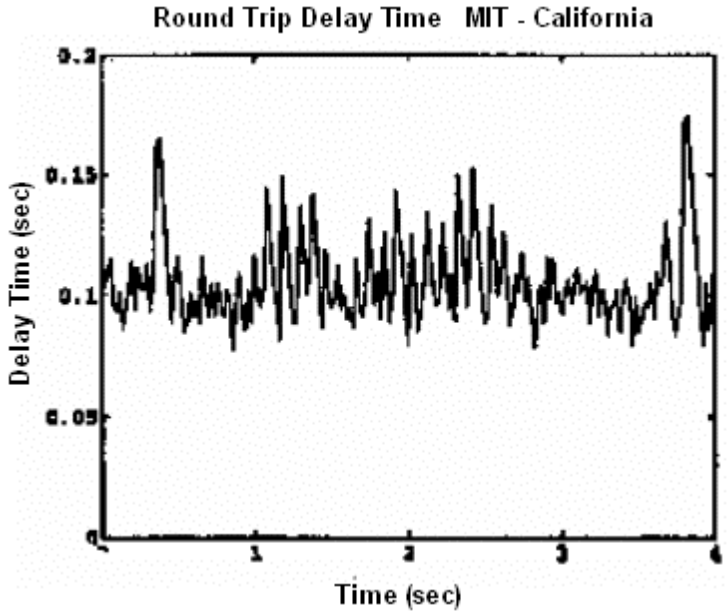


Figura II-17. Tiempo de retardo observado para una comunicación entre MIT y California (4 Km) cuando son transferidos 50 mensajes/segundo a través de internet. [19]

4. Interfaz natural de usuario (NUI)

La interfaz natural de usuario (en inglés *natural user interface* o NUI) es aquella en las que se interactúa con un sistema, aplicación, etc. haciendo uso de movimientos gestuales similares a los que los humanos emplean habitualmente en su interacción con objetos y personas en su entorno, el cuerpo se convierte así en el mando de control. Las NUIs se diseñan considerando estos dos principios fundamentales [10]:

- La NUI ha de ser imperceptible y su uso intuitivo. El objetivo de una *NUI* es crear una interacción sin fisuras entre el humano y la máquina, haciendo que la interfaz en sí misma parezca no existir. Un sensor capaz de capturar los gestos, un micrófono capaz de registrar la voz o una pantalla táctil capaz de seguir los movimientos de las manos, son imperceptibles al usuario. La interfaz no debe desviar nuestra atención de las funciones centrales del sistema.

- La NUI ha de fundamentarse en elementos y gestos naturales. El gesto de desplazar, tocar, el movimiento del cuerpo, las órdenes de voz, todas ellas son acciones naturales que no difieren de las realizadas habitualmente en la interacción con objetos y personas de nuestro entorno.

Algunos ejemplos y aplicaciones de interfaz natural de usuario [64]:

- Las pantallas táctiles permiten al usuario interaccionar con controles y aplicaciones de forma más intuitiva a como se hace con una interfaz basada en un cursor, al ser más directa. Por ejemplo, en lugar de mover el cursor para seleccionar un archivo y hacer clic para abrirlo, el usuario solo ha de dar un toque a una representación gráfica del archivo. El objeto digital en la pantalla responde así de forma parecida a como lo haría un objeto real, haciendo que parezca más natural que usar el ratón y el teclado. Smartphones y tablets permiten hacerlo así.

- Los sistemas de reconocimiento gestual registran el movimiento y lo convierten en instrucciones. Las consolas de juegos Nintendo Wii y PlayStation Move utilizan controles con acelerómetros y giroscopios para detectar inclinaciones, rotaciones y aceleraciones. Una NUI más intuitiva es la que ofrecería una cámara y un software en el dispositivo capaz de reconocer gestos específicos y traducirlos a acciones. Microsoft Kinect, por ejemplo, es un sensor de movimiento para la consola Xbox 360 que permite a los usuarios interaccionar mediante el movimiento de su cuerpo,

gestos y comandos de voz. El reconocimiento de gestos también puede ser utilizado para interaccionar con ordenadores.

- El reconocimiento de voz permite a los usuarios interaccionar con un sistema a través de comandos de voz. El sistema reconoce las palabras y frases que se le dicen y las convierte en un lenguaje que la máquina pueda entender. Entre las aplicaciones del reconocimiento de voz se encuentran programas de voz a texto, manos libres en teléfonos o computadores y también se usa en sistemas embebidos.

- Los sistemas de seguimiento de ojos permiten a los usuarios controlar diferentes dispositivos mediante el movimiento de sus ojos. En marzo de 2011, Lenovo anunció que habían producido el primer portátil controlado con la vista. El sistema de Lenovo combina una fuente de luz infrarroja con una cámara que detecta la luz reflejada sobre los ojos del usuario. El software determina el punto de la pantalla que está siendo mirado y utiliza esa información.

- Las interfaces cerebro-máquina registran la actividad neuronal y las traducen en acciones. Esta interfaz permite a personas con parálisis utilizar computadores, manejar sillas de ruedas o prótesis ortopédicas mediante la mente.

5. Robots manipuladores

Un robot manipulador es un dispositivo para la manipulación de materiales sin un contacto directo con el operador. En sus origines la principal aplicación fue la de tratar con materiales radioactivos o peligrosos, mediante el uso de brazos robóticos, o para ser usados en sitios inaccesibles para un operario humano.

La historia de la automatización industrial y el uso de este tipo de robot se caracteriza por la necesidad del aumento de la producción. El uso industrial del robot se identificó en la década de 1960, junto a los sistemas de diseño asistido por ordenador (CAD) y los sistemas de fabricación asistida por ordenador (CAM). En Norte América tuvo una importante repercusión, a principios de 1980 hubo una gran demanda de equipos robóticos, seguido de un breve retroceso a finales de 1980. Desde entonces el mercado tiende a ir creciendo (Figura II-18), aunque está sujeto a variaciones en los mercados.

Una de las principales razones para el crecimiento en el uso de robots industriales fue su declive en el coste (Figura II-19), los precios de los robots cayeron, mientras que aumentaban los costes laborales humanos Los robots no eran cada vez más baratos, sino más rentables, ya que realizaban tareas con mayor rapidez, con mayor precisión y flexibilidad, o tareas peligrosas o imposibles para los trabajadores humanos [27].

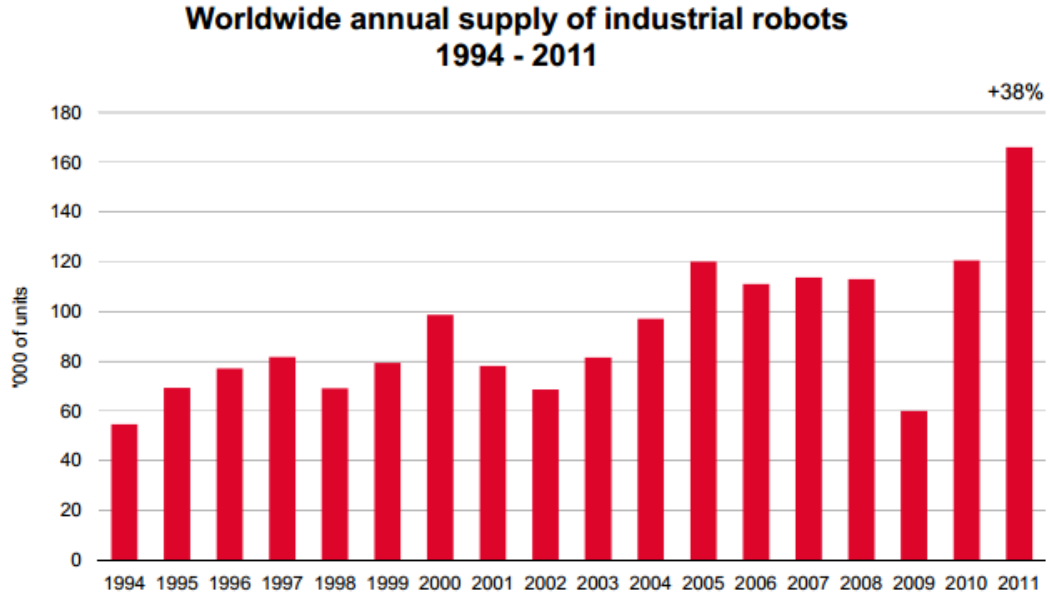


Figura II-18. Demanda de robots industriales entre 1994 y 2011

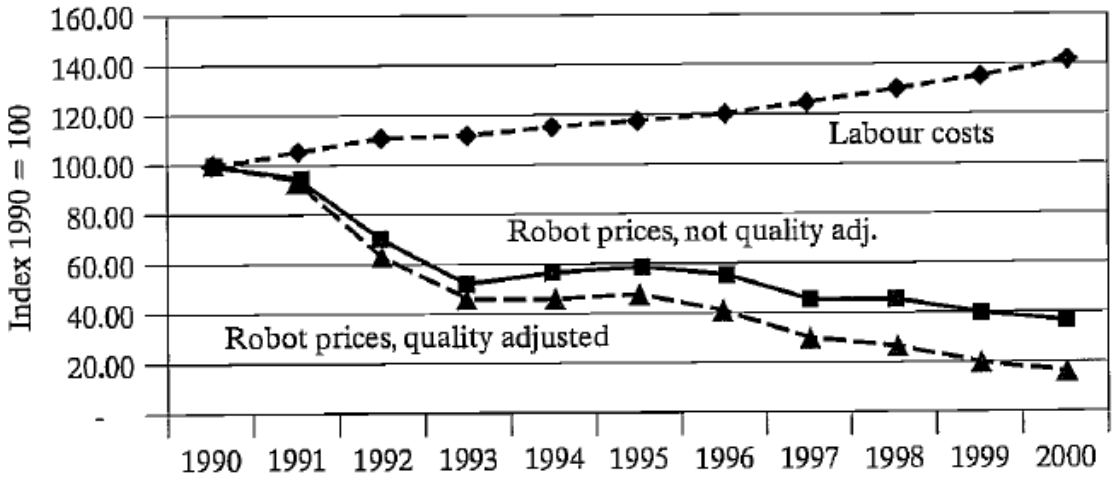


Figura II-19. Precio de los robots en comparación con el coste humano para la realización de labores en los 90 [60]

5.1. Configuraciones de los robots manipuladores

Los robots manipuladores son esencialmente, brazos articulados. De forma más precisa, un manipulador industrial convencional es una cadena cinemática abierta formada por un conjunto de eslabones o elementos de la cadena interrelacionados mediante articulaciones o pares cinemáticos. Las articulaciones permiten el movimiento relativo entre los sucesivos eslabones.

5.2. Articulaciones

Existen diferentes tipos de articulaciones. Las más utilizadas en robótica son las que se indican en la Figura II-20, donde se puede observar cada uno de los tipos de articulaciones más usados, y el número de grados de libertad que proporcionan.

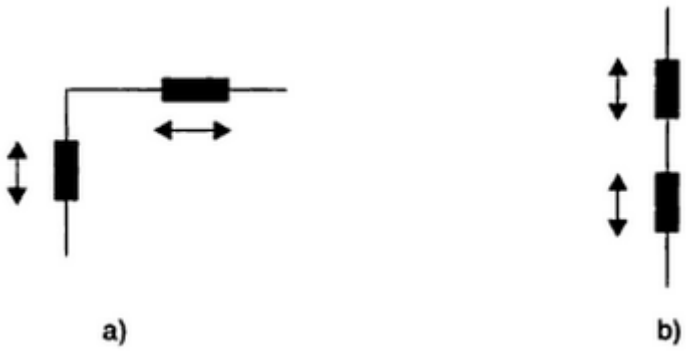


Figura II-20. Pérdida de grados de libertad en estructura de dos eslabones: a) dos grados de libertad y b) un grado de libertad

Los grados de libertad son el número de parámetros independientes que fijan la situación del órgano terminal. El número de grados de libertad suele coincidir con el número de eslabones de la cadena cinemática. Así, en la figura II-20a, se ilustra una estructura con dos eslabones, dos articulaciones prismáticas y dos grados de libertad.

Sin embargo, pueden existir casos degenerados, tal como el que se ilustra en la figura II-20b, en el cual se aprecia que, aunque existan dos eslabones y dos articulaciones prismáticas, tan sólo se tiene un grado de libertad. Por consiguiente,

en general, el número de grados de libertad es menor o igual que el número de eslabones de la cadena cinemática.

- Articulación de rotación. Suministra un grado de libertad que consiste en una rotación alrededor del eje de la articulación. Esta articulación es la más empleada.

- Articulación prismática. El grado de libertad consiste en una traslación a lo largo del eje de la articulación.

- Articulación cilíndrica. Existen dos grados de libertad: una rotación y una traslación, como se indica en la Figura II-21, existiendo, por tanto, dos grados de libertad.

- Articulación esférica. Combina tres giros entres direcciones perpendiculares en el espacio [2].

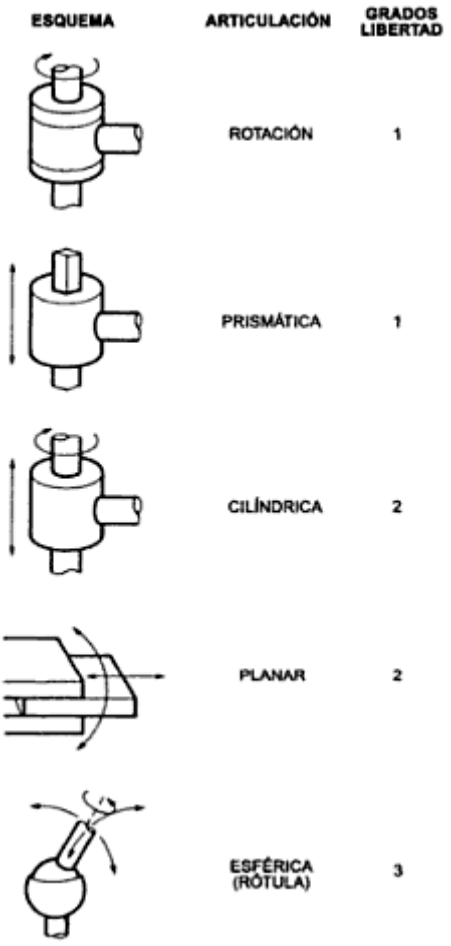


Figura II-21. Tipos de articulaciones

5.3. Estructuras

La estructura común para los robots manipuladores consiste en un brazo compuesto por elementos o eslabones unidos mediante articulaciones. Donde en el último enlace se coloca un elemento terminal, que da la posibilidad de manipular, suelen ser pinzas o elementos diseñados para tareas específicas.

- Configuración cartesiana. Posee tres articulaciones prismáticas (3D o estructura PPP).
- Configuración cilíndrica. Tiene dos articulaciones prismáticas y una de rotación (2D, 1G).
- Configuración polar o esférica. Estructura con dos articulaciones de rotación y una prismática (2G, 1D o estructura RRP)
- Configuración angular. Se caracteriza por tres articulaciones de rotación (3G o RRR).
- Configuración SCARA. Estructura especial, diseñada para tareas de montaje en un plano, constituida por dos articulaciones de rotación respecto a los dos ejes paralelos y una de desplazamiento perpendicular al plano [2].

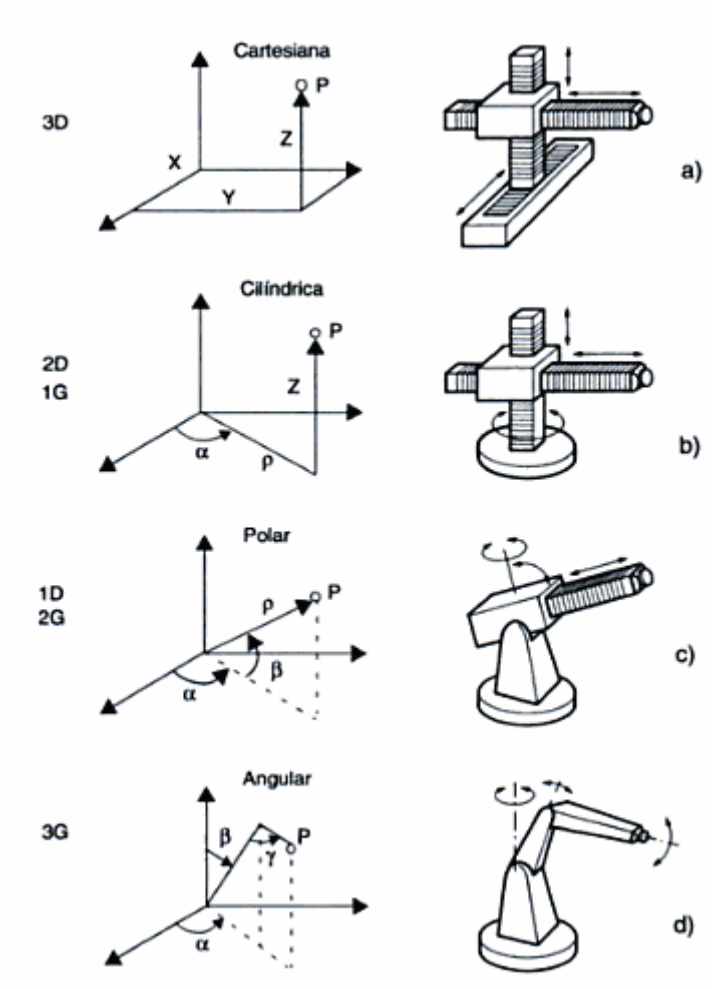


Figura II-22. Configuraciones básicas de robots manipuladores

6. ROS

Sistema Operativo Robótico (en inglés *Robot Operating System*, ROS) es un *framework* para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. ROS se desarrolló originalmente en 2007 bajo el nombre de switchyard por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR2). Desde 2008, el desarrollo continúa primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un modelo de desarrollo federado [56].

En los últimos años, ROS ha crecido formando una gran comunidad de usuarios en todo el mundo. Históricamente, la mayoría de los usuarios estaban en los

laboratorios de investigación, pero cada vez se está viendo adopción en el sector comercial, sobre todo en la robótica industrial y de servicios.

La comunidad de ROS es muy activa. Cuenta con más de 1.500 participantes en la lista de correos electrónicos de ROS, más de 3.300 colaboradores en el sitio web wiki de documentación [15], y alrededor de 5.700 usuarios en la comunidad de Respuestas y preguntas [44]. El sitio web wiki tiene más de 22.000 páginas y más de 30 ediciones de página por día. El apartado de preguntas y respuestas de la página web tiene 13.000 preguntas hechas hasta la fecha, con una tasa de respuesta del 70% por ciento [48].

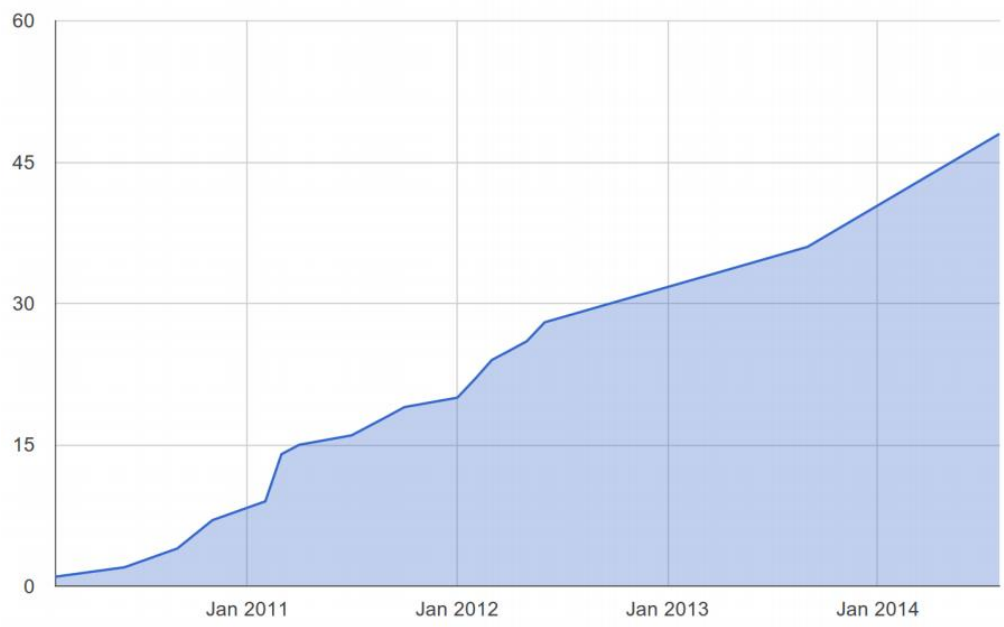


Figura II-23. Numero de diferentes tipos de robot soportados por ROS[50]

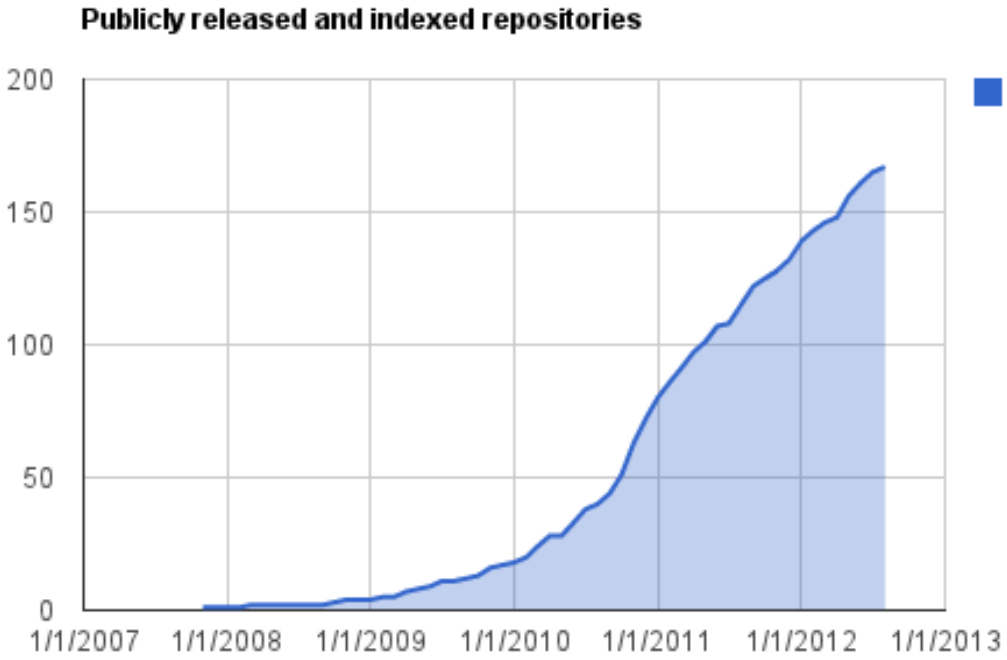


Figura II-24. Crecimiento del número de repositorios oficiales de código ROS [50]

Capítulo III.

III. Materiales y Métodos

7. Materiales

En este apartado se describirá brevemente los distintos materiales usados para la realización de este proyecto. Se indicará el manipulador usado, que se trata del modelo LWA 4P de Robotnik, un brazo robótico de 6 grados de libertad. Por otra parte, se describirá el sensor Kinect, junto al entorno software usado para la comunicación de ambos dispositivos, ROS.

7.1. Brazo Robótico LWA 4P

Es un robot manipulador fabricado por la empresa SCHUNK [42] y distribuido por Robotnik [46]. Este brazo robótico destaca por su ligereza y versatilidad, por ello, puede ser usado en una plataforma móvil o en un humanoide. Entre sus aplicaciones sobresale su uso como robot manipulador o en robótica de servicios.

El brazo robótico LWA 4P (Figura III-1) integra servoaccionamientos modulares de 2 Grados De Libertad, las denominadas ERB, que incorporan etapa de potencia y controlador, como se puede ver en la Figura III-2, por lo que no requiere de un armario de control externo y lo diferencian de un brazo robot industrial. De sus especificaciones, se puede destacar su alcance de 610mm, su repetitividad de 0,06mm y su alimentación a 24VDC, por lo que no requiere de un inversor de gran peso y dimensiones [31].

Figura III-1. Brazo robótico LWA 4P

Figura III-2. Servoaccionamientos modulares de 2 grados de libertad

El espacio de trabajo se define en la Figura III-3.

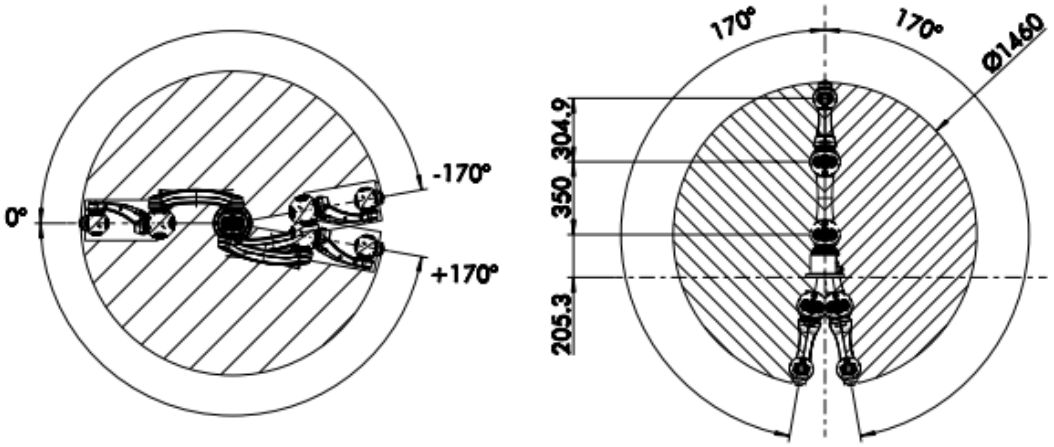


Figura III-3. Espacio de trabajo del brazo robótico LWA 4P

A continuación se muestran las características generales y especificas del robot LWA-4p, recogidas en la Tabla III-1 y la Tabla III-2.

Tabla III-1. Datos técnicos generales del brazo robot LWA 4P

Nombre	LWA 4P
ID	0306960
Tipo	6 GDL
Numero de ejes	6
Max. Carga de trabajo [Kg]	6
Precisión [mm]	±0.15
Sensor posicionamiento	Posicionamiento pseudo-absoluto
Motores	Sin escobillas con freno magnético
Eje de instalación	Cualquiera
Peso total [kg]	15

Tabla III-2. Datos específicos del brazo robot LWA 4P

Clase IP [IP]	40	
Fuente de alimentación	24 V DC / med. 3 A / max. 12 A	
Protocolo	CANopen (CiA DS402:IEC61800-7-201)	
Ejes	Velocidad con carga nominal	Rango
Eje 1	72°/s	±170°
Eje 2	72°/s	±170°
Eje 3	72°/s	±155.5°
Eje 4	72°/s	±170°
Eje 5	72°/s	±170°
Eje 6	72°/s	±170°
Pinzas	WSG 50, PG-plus 70, MEG, SDH 2, SVH	
Conexionado de la pinza	FWS 115 [18]	
Sistema de control robot	ROS node (ROS.org) o KEBA CP 242/A (KEBA.com)	

7.1.1. Fuente de alimentación

Una vez conocidos los requisitos de este robot, se ha buscado una fuente de alimentación acorde. Los datos fundamentales para la búsqueda de han sido obtenidos de la Tabla III-2:

Fuente de alimentación 24 V DC / med. 3 A / max. 12 A

Por tanto, se ha buscado una fuente que proporcione una potencia necesaria para soportar la corriente máxima a un voltaje de corriente continua de 24V.

$$P = \frac{dw}{dt} = \frac{dw}{dq} \cdot \frac{dq}{dt} = V \cdot I\ [W]$$

Tomando como voltaje los 24V y una intensidad instantánea de 12A. Obtenemos

$$P_{min} = V_{cc} \cdot I_{\max inst} = 24\ [V] \cdot 12\ [A] = 288[W]$$

La fuente de alimentación seleccionada fue la DRA480 [17], una fuente de alimentación de montaje superficial en panel y carril DIN [8]. Fue seleccionada debido a su buena relación calidad-precio, a su eficiencia y a su robustez.

Figura III-4. Fuente DRA480 e instalación en el carril DIN

Proporciona una salida de 20A, frente a los 12A requeridos, con una eficiencia del 89%. Proporcionando una potencia de 480W. Su precio es de 177,07 €.

7.2. Interfaz CAN para USB

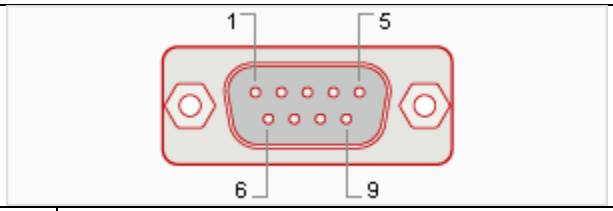
Se trata de un adaptador PCAN a USB de Peak-System [39] (Figura III-5) usado para permitir una fácil conexión a las redes CAN y la gestión de datos CAN. Este dispositivo incluye un microcontrolador, con el objetivo de realizar una conversión de datos garantizando la seguridad y consistencia de los mismos.

Sus especificaciones son las siguientes:

- Adaptador para el puerto USB (USB 1.1, compatible con USB 2.0)
- Fuente de alimentación a través de USB
- Velocidades de transferencia de hasta 1 Mbit/s
- Resolución *Timestamp* sobre 42 microsegundos
- Cumple con las especificaciones CAN 2.0A (ID de 11 bits) y 2.0B (29 bits ID)
- La conexión al bus CAN a través de D-sub de 9 pines (Tabla III-3)
- NXP SJA1000 controlador CAN con 16 MHz de frecuencia de reloj
- NXP transceptor CAN PCA82C251
- Extendido rango de temperatura operativa de -40 a 85 ° C

Figura III-5. Adaptador PCAN-USB de Peak-System

Tabla III-3. Asignación de pines conexión D-Sub



Pin	Ocupación
1	No conectado / opcional +5 V
2	CAN-L
3	GND
4	no asignado
5	no asignado
6	GND
7	CAN-H
8	no asignado
9	No conectado / opcional +5 V

7.3. Sensor Kinect

El sensor Kinect, es un controlador usado para juegos y entretenimiento, fue creado por Alex Kipman y desarrollado por Microsoft para la videoconsola Xbox 360. El objetivo inicial de este sensor fue el de usarlo como un accesorio para ampliar las opciones de jugabilidad y aumentar la experiencia de juego. Permite al usuario interactuar con un entorno mediante gestos y movimientos, sin necesidad

de un contacto físico con ningún elemento. Es el elemento principal que hace posible la comunicación mediante una interfaz natural de usuario NUI (*Natural User Interface*) capaz de reconocer comandos de voz, gestos o imágenes.

Figura III-6. Sensor Kinect de Microsoft

En el interior de la carcasa del sensor, como se muestra en la Figura III-7, contiene los siguientes componentes:

- Una cámara RGB con una resolución de 640x480 pixeles a una frecuencia de 30 Hz, también es capaz de capturar video a una resolución de 1280x1024 pero a una frecuencia de muestreo menor ~9 Hz.

- Un emisor de infrarrojos y sensor infrarrojo de profundidad. El emisor emite unos rayos de luz infrarroja y el sensor de profundidad lee los haces de luz IR reflejados. Los haces reflejados se convierten en información de profundidad midiendo la distancia entre un objeto y el sensor. Esto hace que la captura de una imagen de profundidad sea posible.

- Un micrófono multiarray, contiene cuatro micrófonos para la captura de sonido. Al tratarse de cuatro micrófonos, es posible determinar la posición de la fuente emisora así como la dirección de su onda.

- Inclinación motorizada, para proporcionar una inclinación de hasta 27º hacia arriba o hacia abajo. Por ello es necesario el uso de una fuente externa, aparte de la proporcionada por la conexión USB [30, 28].

Las características técnicas del sensor Kinect han sido recogidas en la Tabla III-4.

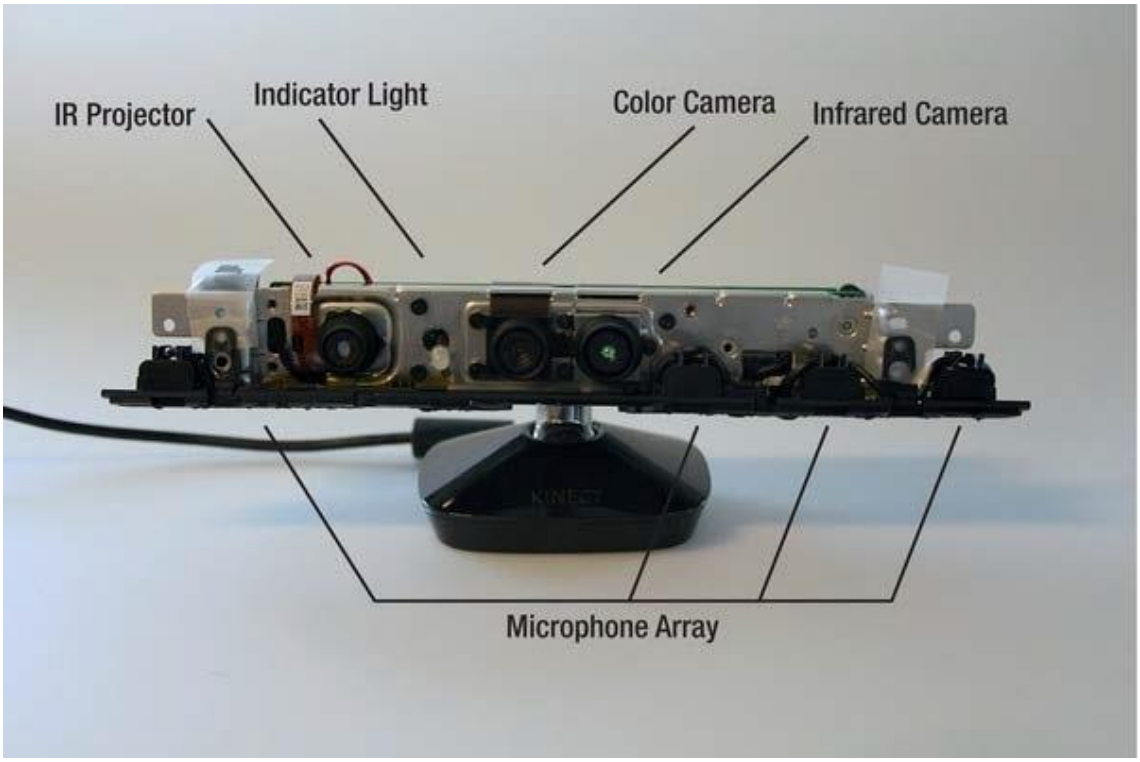


Figura III-7. Interior del sensor Kinect

Tabla III-4. Especificaciones Kinect [30, 28]

Características	Especificaciones
Ángulo de visión	43º vertical por 57º en horizontal (Figura III-8 y Figura III-9)
Ángulo de inclinación	±27º (Figura II-9)
Frecuencia fotogramas	30 imágenes por segundo
Formato de audio	16-kHz, 24-bit mono modulación por pulsos codificados (PCM)
Características entrada de audio	Array de cuatro micrófonos de 24-bits con conversor analógico-digital
Distancia de detección mínima	~0.8 m

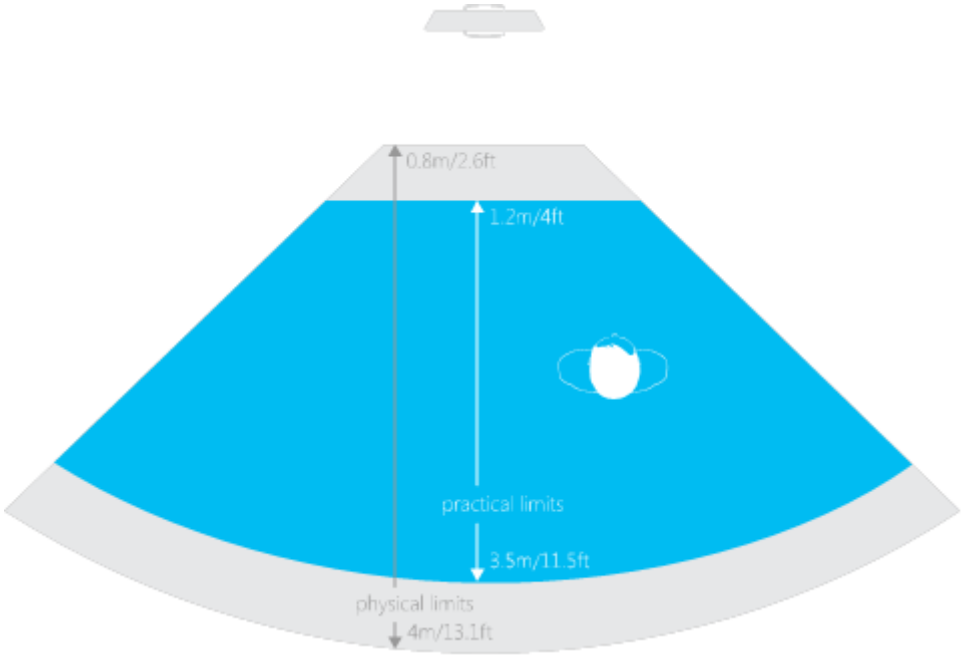


Figura III-8. Campo de visión horizontal del sensor Kinect [57]

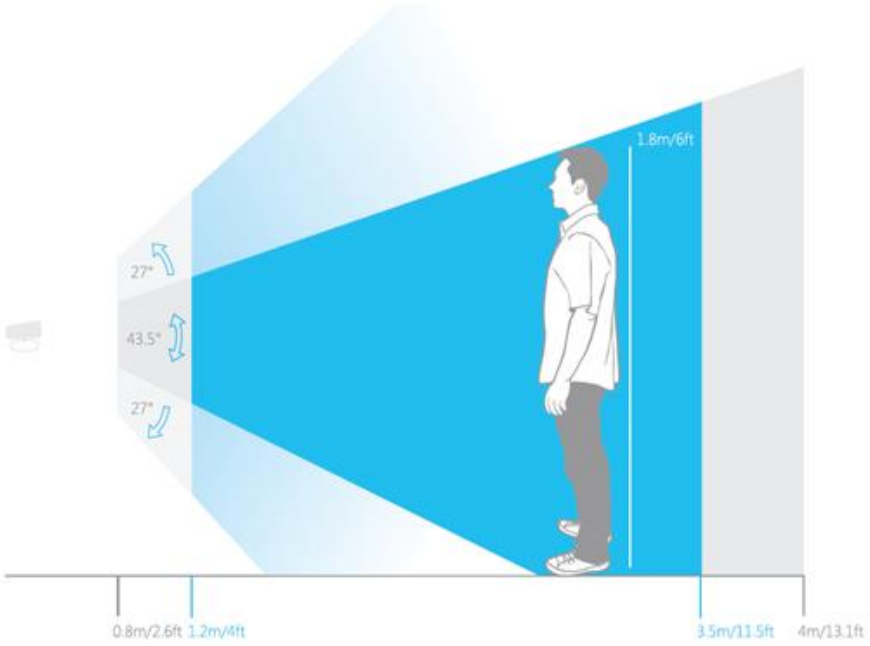


Figura III-9. Campo de visión vertical del sensor Kinect [57]

7.4. Entorno ROS

Sistema Operativo Robótico (en inglés Robot Operating System, ROS) es un *framework* para el desarrollo de software para robots. Proporciona las herramientas, librerías, y convenios que tienen como objetivo simplificar la tarea de crear un comportamiento complejo y robusto en una amplia variedad de plataformas robóticas [49].

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento es realizado en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros.

La librería está orientada para un sistema UNIX. Ubuntu (Linux) es el sistema soportado aunque también se está adaptando a otras distribuciones: Fedora, Arch, gentoo, OpenSUSE, Slackware y debían, y a otros sistemas operativos como Mac OS X y Microsoft Windows [45].

En la Figura III-10 se muestra un esquema de comunicación ROS para una aplicación desarrollada.

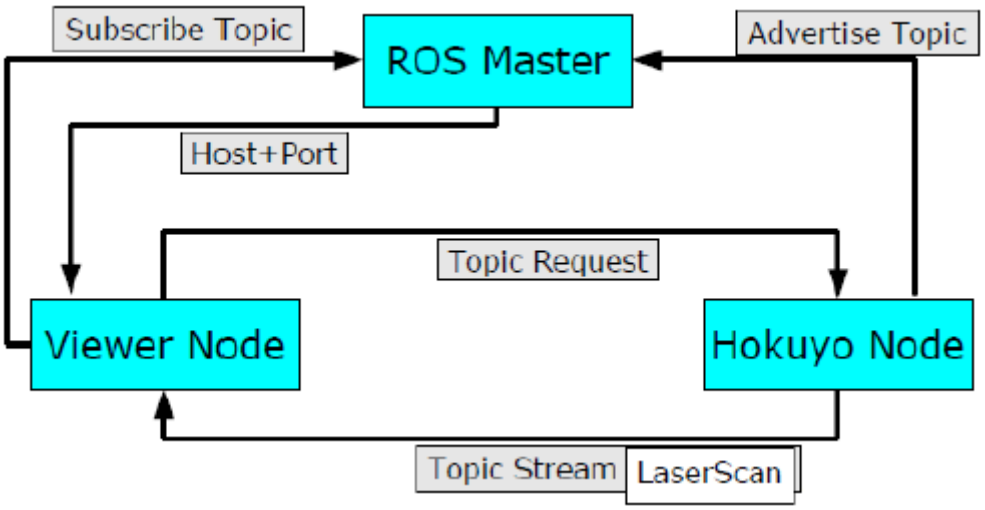


Figura III-10. Ejemplo de comunicación en ROS

7.4.1. Sistema de archivos en ROS

Para entender el funcionamiento de ROS, se han de definir los siguientes elementos que ponen su sistema de archivos. En la Figura III-11 se muestran los archivos de los que se compone el sistema de ROS.

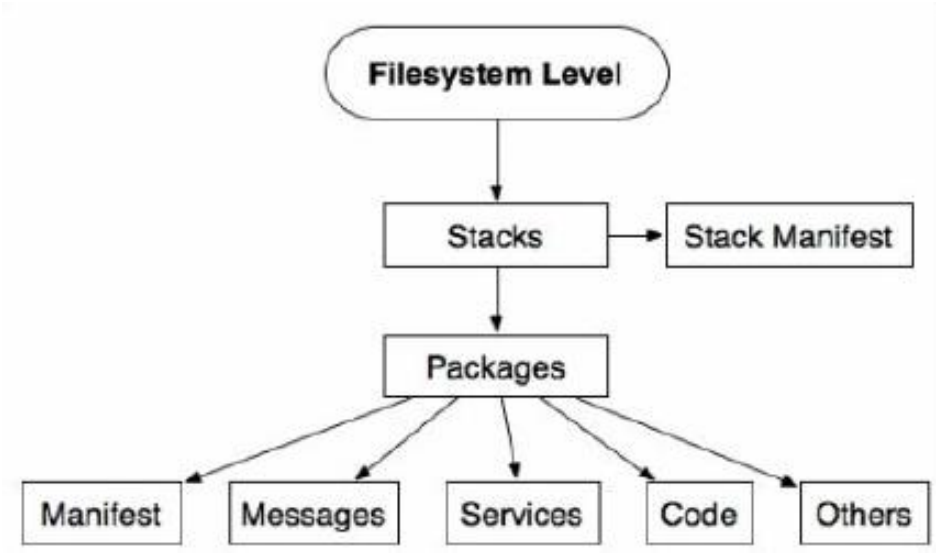


Figura III-11. Esquema del sistema de archivos en ROS

- Paquetes. Son la unidad principal de ROS, contiene nodos o procesos, librerías de desarrollo, sets de datos, archivos de configuración, etc.
- Manifest (manifest.xml). Proporcionan información sobre un paquete (meta-datos), información de la licencia, dependencias, información sobre el lenguaje de programación usado y parámetros de compilación,
- *Stacks*. Son colecciones de paquetes que proporcionan funcionalidades específicas, existen muchas stacks para distintos usos, por ejemplo, el stack de navegación.
- *Stack manifest* (stack.xml). Proporcionan datos sobre el stack, incluyendo su licencia y sus dependencias de otros stacks.
- Tipos de mensajes (msg). Un mensaje es la información que un proceso manda a otro. ROS posee muchos estándares de tipos de mensaje, que

definen la estructura de datos de los mansajes. Las estructuras de los mensaje son almacenados en:

```
my_package/msg/MyMessageType.msg
```

- Tipos de Sevicios (svr). Describen el mensaje de petición y respuesta de un servicio en ROS, se almacena en;

```
my_package/srv/MyServiceType.srv
```

7.4.2. Grafo de comunicación en ROS

En este apartado se definirá la estructura de comunicación de ROS, se trata de una red compuesta por procesos que se comunican entre ellos a través de mensajes, estos mensajes pueden ser punto a punto o uno a muchos mediante *broadcast*. La Figura III-12 muestra los elementos implicados una comunicación a través de ROS.

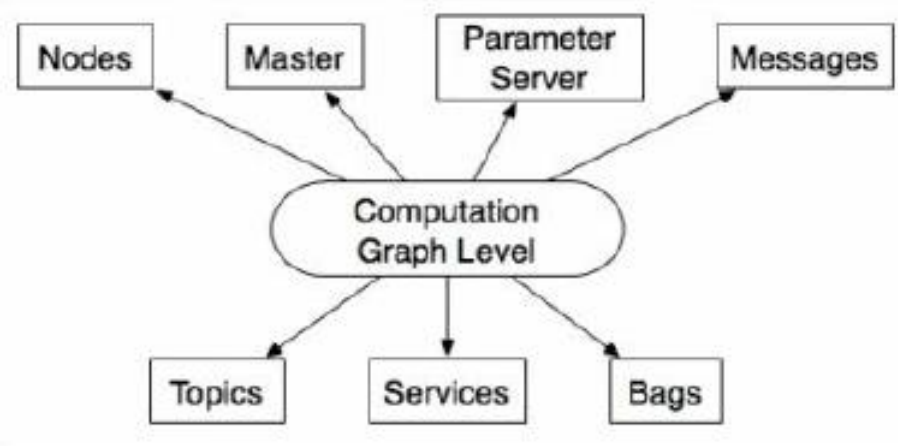


Figura III-12. Esquema del grafo de comunicación en ROS

- Nodos. Son procesos que realizan el cómputo. Usualmente, un sistema tendrá muchos nodos para controlar diferentes funciones, esto proporciona una mejor detección de errores en caso de fallo en alguna de las comunicaciones entre nodos, se trata de un sistema modular.

- Máster. Proporciona el registro de nombres y direcciones, Sin el máster, los nodos no podrían direccionarse correctamente, siendo imposible su comunicación o la llamada a servicios.

- Parámetros del servidor. Dan la posibilidad de tener datos almacenados en una localización central específica. Con estos parámetros, es posible configurar nodos mientras están en ejecución o cambiar el trabajo realizado por los nodos.

- Mensajes. Los nodos se comunican entre ellos a través de mensajes. Un mensaje contiene información que es enviada a otro nodo. ROS proporciona muchos tipos de mensajes, también es posible diseñar uno propio usando los estándares primitivos (entero, punto flotante, booleano, etc.)

- Tópicos. Cada nodo debe tener un nombre específico para ser direccionado por la red ROS. Cuando un nodo está enviando datos, se dice que está publicando un tópico. Un nodo puede suscribirse a otro nodo usando su tópico.

- Servicios. Un servicio ofrece la posibilidad de interactuar con otros nodos, cuando un nodo posee un servicio, todos los nodos pueden comunicarse con él. Es necesario el uso de servicios cuando se requiere una petición o respuesta de un nodo específico, y no un envío de datos mediante tópicos.

- *Bags*. Es el formato para guardar y reproducir mensajes de datos en ROS.

En Figura III-13, se puede observar una representación gráfica de un grafo de comunicación mediante ROS de un robot real. La gráfica ha sido obtenida mediante la ejecución del comando *rxgraph* proporcionado por ROS.

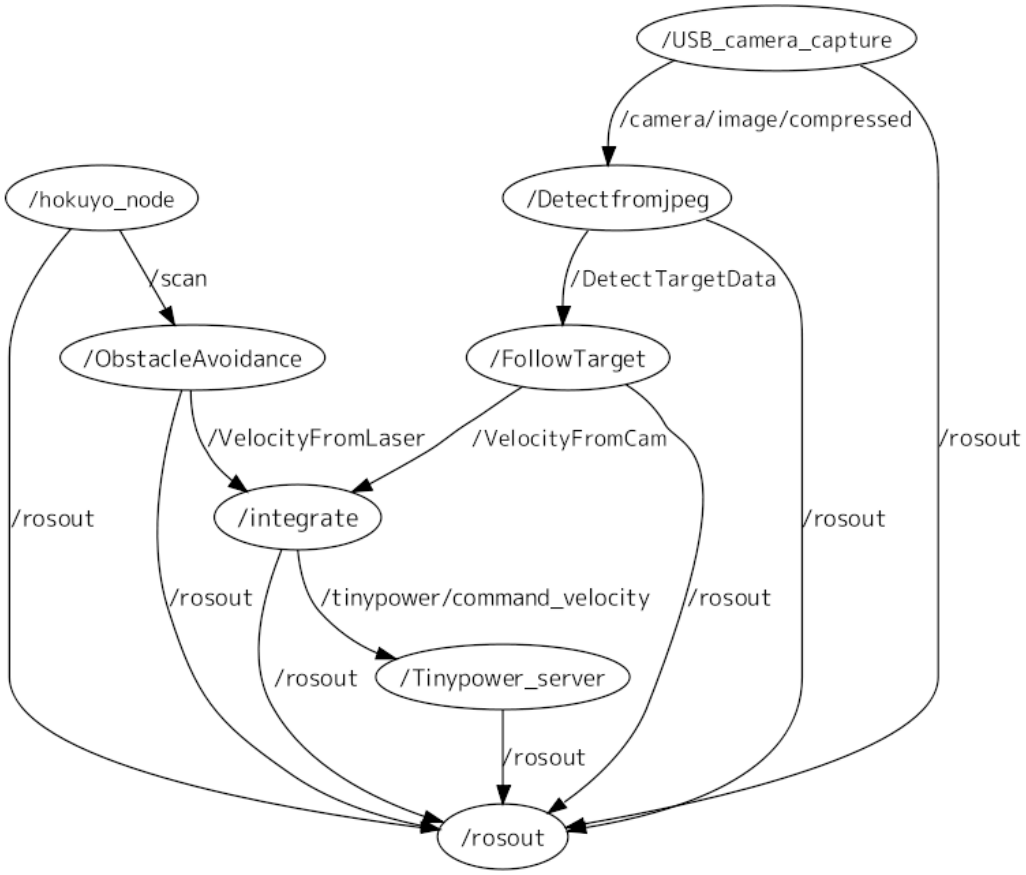


Figura III-13. Grafo de comunicación de un robot real mediante rxgraph

7.5. RViz

ROS dispone de una herramienta de visualización en 3D llamada RViz, esta herramienta puede ser usada para mostrar lecturas recogidas por sensores, datos devueltos por la visión estereoscópica (Cloud Point), para la localización, detección de obstáculos con la recreación de entornos en 3D, etc. En la Figura III-14 se muestra un ejemplo de su uso para una aplicación con Kinect.

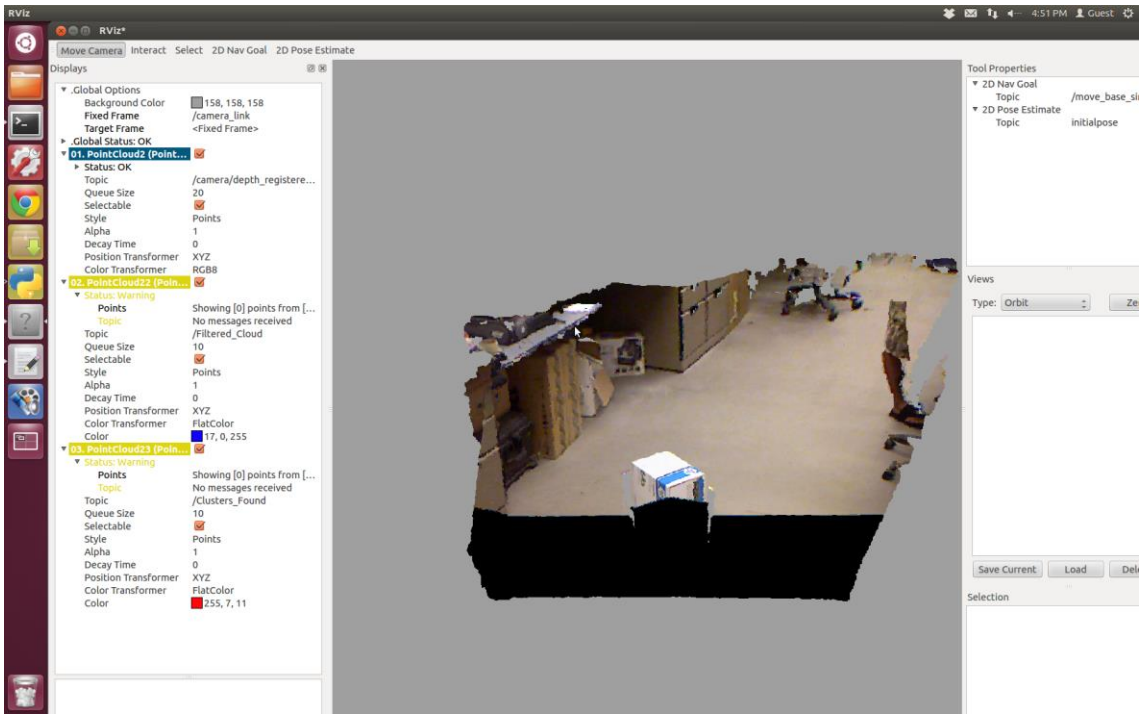


Figura III-14. Recreación de un entorno 3D mediante Kinect con RViz

7.6. Gazebo

El desarrollo de Gazebo [20] comenzó en el otoño de 2002 en la Universidad del Sur de California. Los creadores originales fueron el Dr. Andrew Howard y su estudiante Nate Koenig. El concepto de un simulador de alta fidelidad se deriva de la necesidad de simular robots en entornos exteriores bajo diversas condiciones.

Gazebo, es una herramienta de simulación robot de código abierto, usado también dentro del entorno ROS. Se trata de un *toolbox* de robótica, capaz de simular y probar rápidamente y de manera sencilla toda clase de algoritmos, diseños de robots, y simulación de toda clase de entornos 3D, con la posibilidad de realizar simulaciones de robots interactuando en ellos.

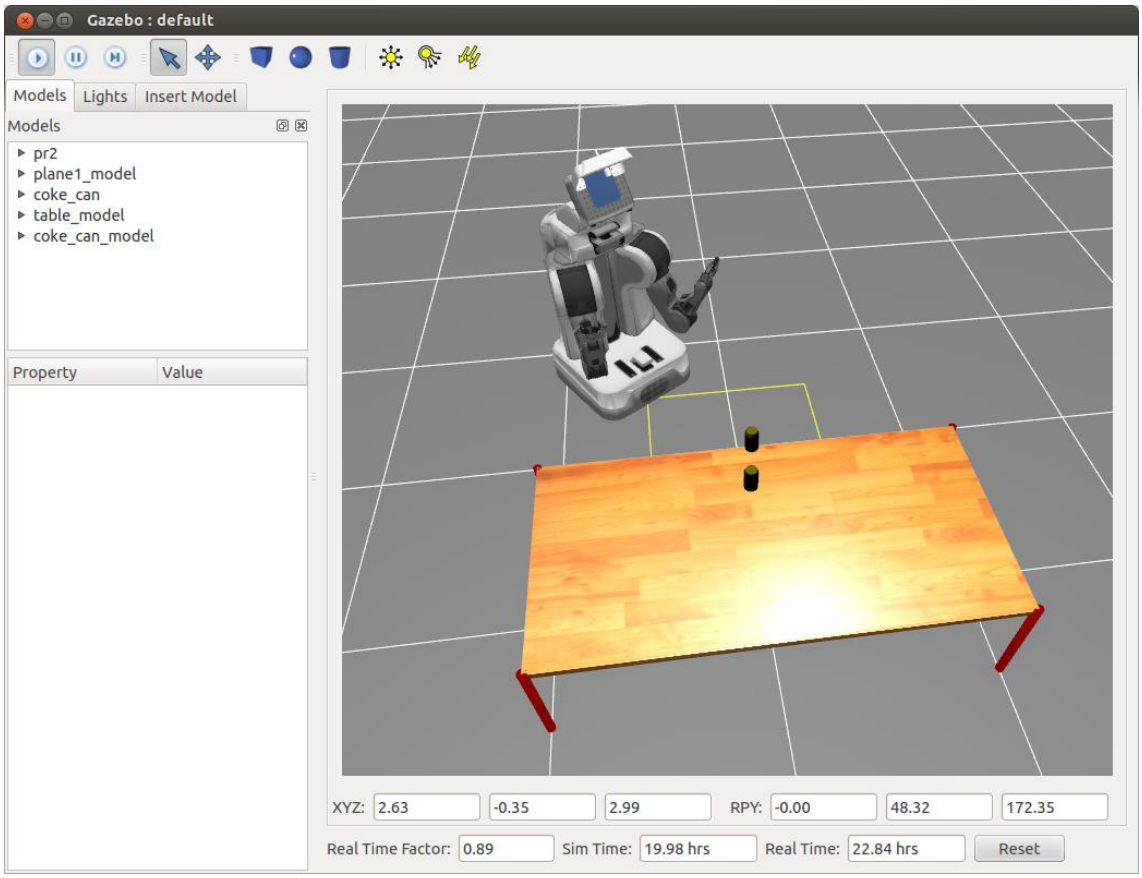


Figura III-15. Interfaz de la herramienta Gazebo

7.1. Qt Creator

Qt Creator es un entorno de desarrollo integrado creado por Trolltech en 2007 para el desarrollo de aplicaciones con las bibliotecas Qt, está compuesto por un conjunto de herramientas de programación; un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

8. Métodos

En este apartado se detallarán los procedimientos usados para lograr la implementación del paquete ROS encargado de realizar una comunicación entre Kinect y el brazo robot LWA 4p.

8.1. Comunicación CANopen

CANopen es un protocolo de comunicación usado mayormente en el área de automatización. En el modelo OSI, CANopen implementa la capa de red y las superiores. El estándar CANopen consiste en un esquema de direccionamiento, varios protocolos de comunicación pequeños y una capa de aplicación. Los protocolos de comunicación tienen soporte para la gestión de red, supervisión de dispositivos y la comunicación entre los nodos, incluyendo una capa de transporte simple para la segmentación/desegmentación del mensaje [7].

El bus de campo CAN sólo define la capa física y de enlace por lo que es necesario definir cómo se asignan y utilizan los identificadores y datos de los mensajes CAN. Para ello se definió el protocolo CANopen, que está basado en CAN, e implementa la capa de aplicación (Figura III-16). Actualmente está ampliamente extendido, y ha sido adoptado como un estándar internacional.

La construcción de sistemas basados en CAN que garanticen la interoperatividad entre dispositivos de diferentes fabricantes requiere una capa de aplicación y unos perfiles que estandaricen la comunicación en el sistema, la funcionalidad de los dispositivos y la administración del sistema:

- Capa de aplicación. Proporciona un conjunto de servicios y protocolos para los dispositivos de la red.
- Perfil de comunicación. Define cómo configurar los dispositivos y los datos, y la forma de intercambiarlos entre ellos.
- Perfiles de dispositivos. Añade funcionalidad específica a los dispositivos.

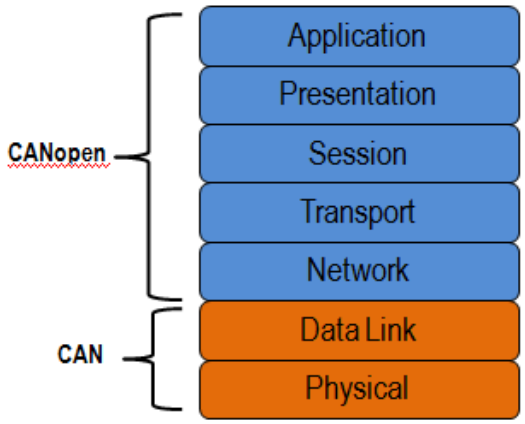


Figura III-16. Modelos OSI de CAN y CANopen

8.2. Seguimiento de personas (*Skeletal Tracking*)

Una de las principales funcionalidades del sensor Kinect es el seguimiento de personas (*Skeletal tracking*), se basa en un algoritmo que logra identificar diferentes partes del cuerpo y sus uniones, el sensor Kinect es capaz de reconocer hasta seis personas y realizar un seguimiento de dos (Figura III-17). Para ello es necesario estar a una distancia mínima de 0.8m, como se indicó en el apartado 7.3 de este capítulo.

Un año después de la salida oficial del sensor Kinect para la consola Xbox 360, Microsoft hizo pública una herramienta de desarrollo software, Kinect SDK para Windows 7 [29], que permitía a los desarrolladores programar aplicaciones para Kinect en C++/CLI, C# o Visual Basic .NET [28].

Por otro lado se desarrollaron librerías y drivers de código abierto que incluían algunas de las funcionalidades originales del sensor Kinect, con la posibilidad de trabajar con ellas tanto en Windows como en Linux.

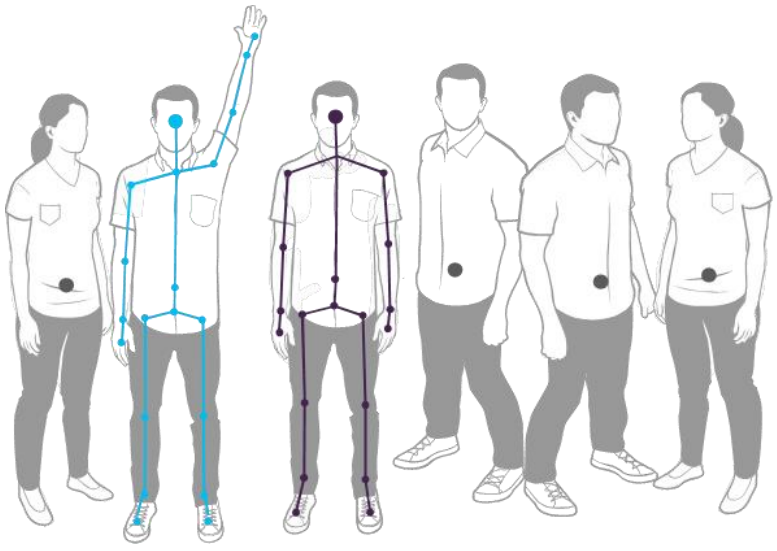


Figura III-17. Kinect puede reconocer hasta seis personas y realizar un seguimiento de dos [57]

OpenNI, (del inglés *Open Natural Interaction*) es una organización sin ánimo de lucro, centrada en promover la compatibilidad e interoperabilidad de dispositivos,

aplicaciones y middleware de interacción natural (*NUI*). El *framework* de OpenNI provee de una serie de APIs [3] de código abierto, ofreciendo soporte para reconocimiento de voz, gestos de mano o seguimiento de cuerpos. En la Figura III-18 se muestra una aplicación desarrollada con una de las librerías de OpenNI.

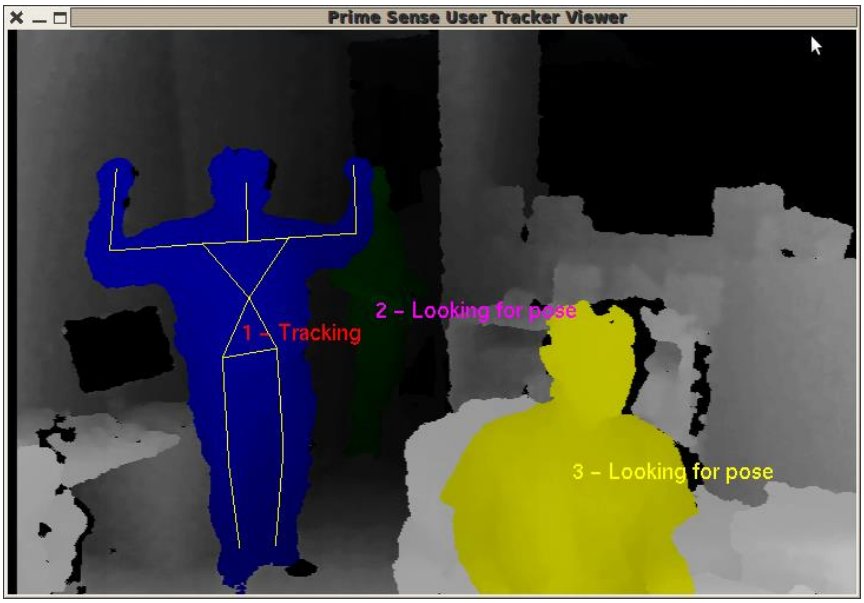


Figura III-18. Skeletal Tracking mediante OpenNI [36]

8.3. Paquetes ROS

En esta sección se indicarán los paquetes, *stacks*, o librerías ROS necesarias para la comunicación y control del robot mediante el sensor Kinect.

8.3.1. Paquete `TF`

Se trata de un paquete que permite al usuario hacer un seguimiento de múltiples vectores de coordenadas en el tiempo. Posee una estructura de árbol, cada una de las coordenadas o *frames* deben estar referenciadas respecto a otro punto, llamado origen. Este paquete permite al usuario transformar puntos, vectores, etc. entre cualquier punto del sistema en cualquier instante de tiempo. En la Figura III-19 se pude ver la representación de estos *frames* en un robot simulado.

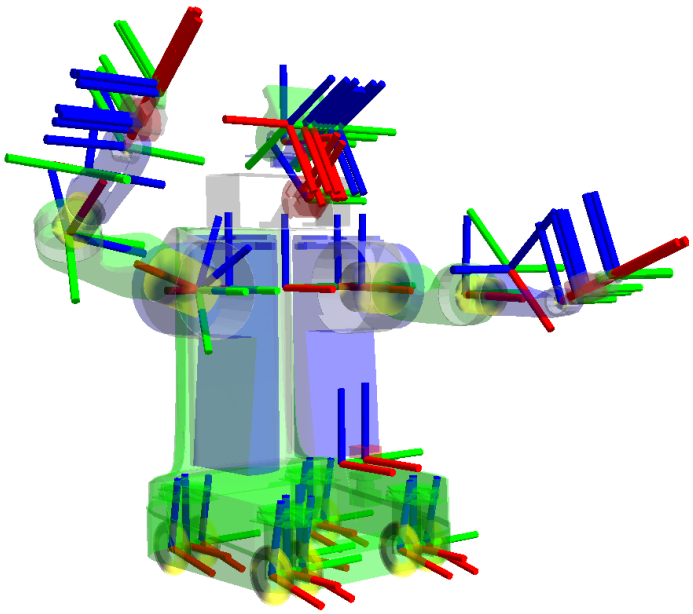


Figura III-19. Representación en RViz de un esquema de transformadas.

Uno de sus usos es el de trabajar con sistemas robóticos donde se encuentran multitud de puntos de coordenadas que cambian respecto al tiempo, como el entorno, la base, una pinza, etc. Este paquete permite el seguimiento de estos puntos de coordenadas y permite la determinación de la matriz de posición y orientación de un elemento del robot respecto de otro en un instante de tiempo determinado.

El paquete tf opera en un sistema distribuido, toda la información está disponible para cualquier componente o equipo comunicado con ROS [58].

La posición relativa entre dos sistemas de coordenadas está representada por una posición relativa de 6 GDL, una transformación seguida de una rotación. Si W y A son dos sistemas de coordenadas, la posición de A respecto de W esta dada por la transformación del origen de W al origen de A, y la rotación de las coordenadas de A respecto de W.

La traslación es un vector en las coordenadas de W, definido como W_{t_A}. Está representado mediante un mensaje del tipo *Vector3*, definido en el paquete de ROS `geometry_msg` [22]. Se puede ver la definición del mensaje, dentro del archivo `geometry_msgs/Vector3.msg`

```
# This represents a vector in free space.

float64 x
float64 y
float64 z
```

La rotación de A esta dada por una matriz, representada como ${}^{W}_{A}R$. Esta rotación se define como la rotación del sistema A en el sistema de coordenadas de W. Las columnas de R están formadas por tres vectores unitarios en los ejes de W: ${}^{W}X_{A}$, ${}^{W}Y_{A}$, ${}^{W}Z_{A}$.

No existe un tipo de mensaje para definir una matriz de rotación. Por ello el mensaje usado es del tipo *Quaternion* definido en el paquete `geometry_msgs`, al igual que el usado en el vector de translación.

```
# This represents an orientation in free space in quaternion for
m.

float64 x
float64 y
float64 z
float64 w
```

Existen métodos para crear cuaterniones [43] a partir de las matrices de rotación y viceversa.

Es conveniente describir la translación y rotación de manera conjunta, mediante el uso de coordenadas homogéneas, en una matriz simple 4x4. De tal modo, la posición relativa del sistema A con respecto a W quedaría definida como:

$$T = \begin{bmatrix} {}^{W}_{A}R & W_{t_A} \\ 0 & 1 \end{bmatrix}$$

En `tf`, las posiciones relativas son representadas mediante el mensaje *Pose*, equivalente al mensaje `Transform`. Este mensaje engloba los dos mensajes definidos anteriormente.

```
# This represents the transform between two coordinate frames in
 free space.

Vector3 translation
Quaternion rotation
```

Mediante la clase definida `btTransform` [6], es posible obtener la traslación, posición así como la obtención de sus vectores y matrices y realizar operaciones. [59].

8.3.2. Software de manipulación móvil PR2

Se trata de un sistema software creado por Willow Garage [47], provee de las herramientas software necesarias para trabajar con toda clase de plataformas robot. Todo este software está disponible para su uso en una interfaz ROS. Consta de los siguientes elementos:

- Configuración básica (Tabla III-5. Componentes de configuración PR2. Mensajes, servicios, modelos de robots y archivos de configuración de host remotos para establecer una comunicación con el robot.

Tabla III-5. Componentes de configuración PR2

Componente	Paquete/stack ROS
Mensajes	pr2_msgs
Servicios	pr2_srvs
Modelos robots (URDF)	pr2_description
Configuración de *hosts*	pr2_machine

- Controladores y simuladores (Tabla III-6). Conjunto de paquetes que implementan controladores para los sensores y actuadores, así como de herramientas para la simulación 3D

Tabla III-6. Componentes de controladores PR2

Componente	Paquete/stack ROS
Controladores para actuadores	pr2_controllers
Controladores para sensores	camera_drivers, imu_drivers, laser_drivers, sound_drivers, pr2_ethercat_drivers, pr2_power_drivers
Simulación (3D)	pr2_simulator

- Componentes de alto nivel (Tabla III-7). Establecen una comunicación con los robots soportados por el software PR2 mediante funciones de alto nivel.

Tabla III-7. Componentes de alto nivel PR2

Componente	Paquete/stack ROS
Teleoperación	pr2_teleop
Navegación	pr2_navigation
Cinemáticas	pr2_kinematics
Control de trayectorias	pr2_arm_navigation
Manipulación	pr2_object_manipulation
Percepción	vision_opencv, pcl, tabletop_object_perception
Ejecución de tareas	executive_smach, continuous_ops

8.3.3. *Stack* `schunk_robots`

Se trata de una recopilación de paquetes creados por Florian Weisshardt que proporcionan una configuración hardware así como archivos de ejecución (*Launch files*) para iniciar componentes SCHUNK [53]. Está compuesto de los siguientes paquetes:

- `schunk_bringup`. Después de realizar una configuración del dispositivo SCHUNK mediante *schunk_hardware_config*, se puede realizar un correcto inicio de los componentes mediante el archivo *Launch*.

- `schunk_bringup_sim`. Permite simular los dispositivos SCHUNK en el entorno de Gazebo.

- `schunk_controller_configuration_gazebo`. Contiene archivos de configuración YAML para simulación.

- `schunk_default_config`. Configuración por defecto de los diferentes componentes SCHUNK. Se trata de configuraciones predefinidas, como la posición inicial de las articulaciones.

- `schunk_hardware_config`. Ejemplos de configuración para cada componente SCHUNK.

8.3.4. *Stack* `arm_navigation`

El *stack* arm_navigation [4] contiene herramientas útiles para la generación de aplicaciones de navegación para brazos robots y herramientas interactivas. Está compuesto por los siguientes elementos:

- `motion_planning_common`. Contiene un conjunto común de componentes y paquetes que son útiles para la planificación de movimiento. También contiene paquetes para la representación de formas geométricas.

- `collision_environment`. Contiene herramientas para crear una representación del entorno para la detección de colisiones. También contiene un paquete para convertir los datos del sensor 3D a nubes de puntos para la creación de mapas de colisión.

- `motion_planning_environment`. Contiene herramientas que se pueden utilizar para la detección de colisiones de robot dado y evaluar las posibles trayectorias.

- `motion_planners`. Contiene 3 diferentes controladores de movimiento para la creación de planes de movimiento de un brazo robótico.

- `kinematics`. Contiene un conjunto de mensajes y servicios de planificación de movimientos que le permiten llevar a cabo el cálculo de la cinemática mediante el uso de un nodo específico.

- `trajectory_filters`. Contiene un conjunto de filtros que pueden ser utilizados para suavizar trayectorias. También contiene suavizadores que se pueden utilizar para crear trayectorias suaves libres de colisión para el robot PR2.

- `arm_navigation`. Contiene la implementación de un nodo que se comunica con los controladores de movimiento, el entorno del robot y los controladores para crear una implementación completa de planificación y control de movimiento. También implementa el monitoreo de las trayectorias realizadas con el fin de anticiparse a posibles colisiones con algún elemento del entorno.

- `motion_planning_visualization`. Contiene *plugins* para el visualizador RViz que le permiten visualizar trayectorias y mapas de colisión.

8.3.5. *Stack* `cob_command_tools`

El *stack* cob_command_tools [11] contiene los paquetes necesarios para establecer una comunicación a alto nivel con plataformas robots. Es de especial interés debido al paquete `cob_script_server`, que proporciona una serie de

comandos para el control de movimientos. En la Tabla III-8 se detallan todos los comandos disponibles.

Tabla III-8. Comandos Python API `cob_script_server`

Comando	Nombre del componente	Parámetro	Ejemplo	Descripción
`init()`	base|torso| tray|arm|sdh	--	`sss.init("base")`	Inicializa los componentes
`recover`	base|torso| tray|arm|sdh	--	`sss.recover("base")`	Recupera los componentes tras una parada de emergencia
`stop`	base|torso| tray|arm|sdh	--	`sss.stop("base")`	Parada actual de la trayectoria
`move`	torso|tray| arm|sdh	string of parameter	`sss.move("torso","front")`	Mueve los componentes a una posición predefinida
`move`	torso|tray| arm|sdh	list of points with joint values	`sss.move("torso", [[0, 0, 0, 0], [0.1, 0, 0.2, 0], [...], ...])`	Mueve los componentes del robot mediante una lista de puntos
`move`	base	[x,y,th] in map coordinates	`sss.move("base",[0, 0, 0])`	Mueve los componentes a una posición definida numéricamente
`sleep`	--	time in seconds	`sss.sleep(2.0)`	Pausa
`wait_for_input`	--	--	`ret = sss.wait_for_input()`	Espera a una entrada por teclado
`say`	--	string of parameter	`sss.say("sentence1")`	Reproduce una cadena predefinida
`say`	--	list of strings	`sss.say(["Hello."])`	Reproduce una cadena

`set_light`	--	string	`sss.set_light("red")`	Fija los LEDs a un color predefinido
`set_light`	--	[r,b,g]	`sss.set_light([0.5,0.5,0.5])`	Fija los LEDs a un color RGB dado

Tabla III-9. Comandos de espera Python API cob_script_server

Comando	Ejemplo	Descripción
`wait()`	handle.wait()	Espera hasta la finalización de un comando
`wait(timeout)`	handle.wait(2.0)	Espera hasta la finalización de un commando, con un tiempo de espera máximo
`get_state()`	handle.get_state()	Obtiene el estado del commando en ejecución

8.3.6. Stack `ipa_canopen`

El *stack* `ipa_canopen` [25] incluye paquetes que proporcionan acceso al hardware CANopen mediante mensajes, servicios y acciones de ROS (Figura III-20). Su autor es Thiago de Freitas. Está compuesto por los siguientes paquetes:

- `ipa_cannopen_core`. Proporciona un nodo configurable para operar con modulos CANopen.

- `ipa_canopen_ros`. Este nodo da uso de un control de trayectoria y recibe datos sobre este, para poder realizar una comunicación con el Hardware mediante el protocolo CANopen. Las cadenas o grupos de dispositivos (cada uno de los módulos de control del brazo robótico) pueden ser iniciados, parados o recuperados tras una parada de emergencia a través de servicios.

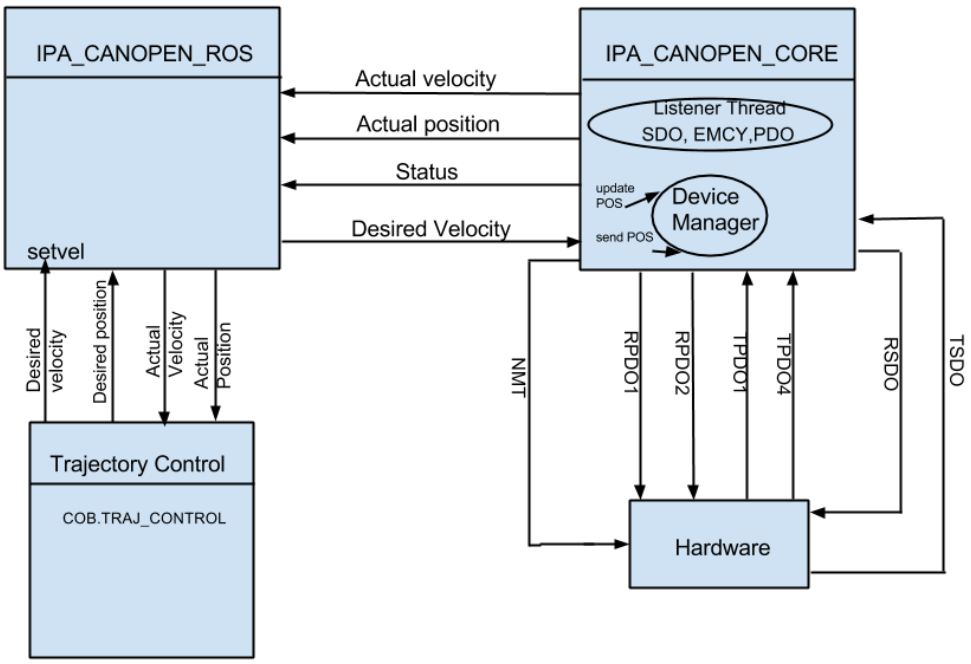


Figura III-20. Esquema de comunicación `ipa_canopen` [25]

8.3.7. Paquete *openni_tracker*

El seguidor OpenNI transmite mediante *broadcast* las coordenadas del usuario mediante el uso de `tf` [9]. Este paquete fue creo por Tim Field y es distribuido bajo la licencia Berkeley Software Distribution o BSD (en español, distribución de software berkeley).

La posición y orientación de las articulaciones del usuario son publicadas usando los siguientes nombres para cada *frame*: `/head`, `/neck`, `/torso`, `/left_shoulder`, `/left_elbow`, `/left_hand`, `/right_shoulder`, `/right_elbow`, `/right_hand`, `/left_hip`, `/left_knee`, `/left_foot`, `/right_hip`, `/right_knee` y `/right_foot`. Estos *frames* pueden ser capturados y procesados mediante RViz, como se puede apreciar en la Figura III-20.

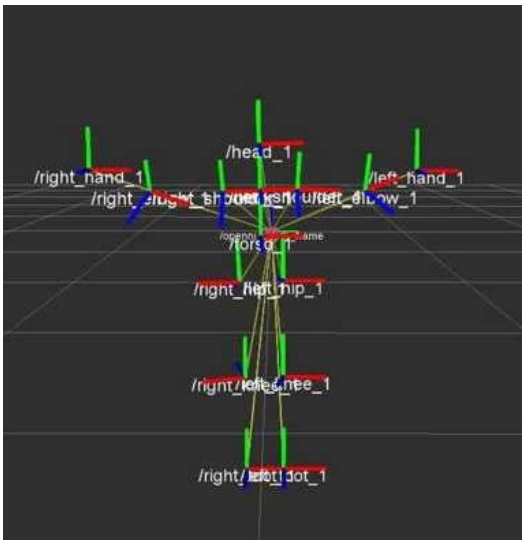


Figura III-21. Representación de las transformadas a través de RViz

Capítulo IV.

IV. Puesta a punto del sistema

En el siguiente capítulo se describirá el proceso realizado para la instalación del entorno de trabajo. Se trataran los temas de los elementos usados, el conexionado de los distintos dispositivos y la puesta a punto de brazo robótico y del ordenador que realiza el procesado, indicando las características necesarias y prestaciones.

En la Figura IV-1 se muestra el esquema que se pretende implementar en el entorno de trabajo, consta de los materiales nombrados en el capítulo anterior.

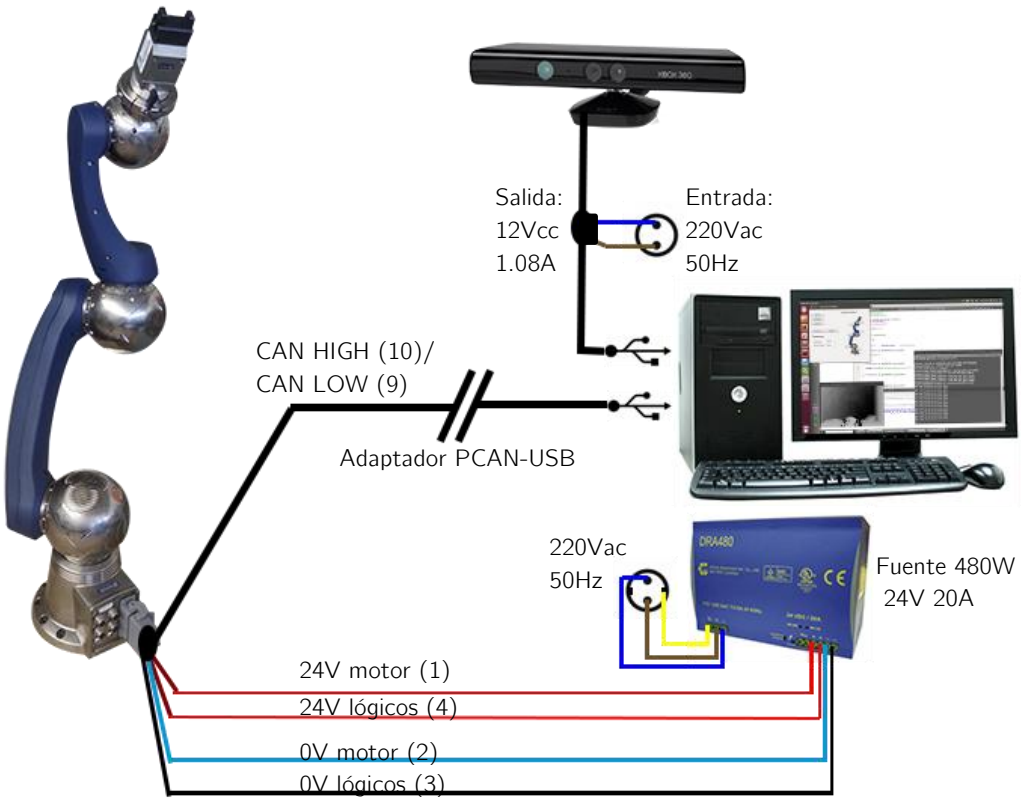


Figura IV-1. Esquema y conexionado de los elementos del entorno de trabajo.

Como se aprecia, el sensor Kinect para Xbox 360, requiere de una fuente de alimentación externa, con el fin de proporcionar corriente suficiente al dispositivo y poder usar el servo-motor para la inclinación de la cámara.

Las indicaciones de la Figura IV-1 para el cableado del robot hacen referencia a los pines de conexión HAN 10A-M pertenecientes a la base del robot, como se puede apreciar en la Figura IV-2.

Figura IV-2. Numeración de pines macho y hembra para las conexiones HAN 10A

9. LWA 4p

En este apartado se describirá todo el proceso de instalación y testeo del brazo robot. Se comenzará con el entorno de trabajo del robot, la actualización de su firmware y se finaliza con una comprobación de cada una de las articulaciones del brazo robot.

9.1. Espacio de trabajo

El manipulador se ha situado anclado a una mesa de madera con el tamaño suficiente para cubrir la totalidad de su área de trabajo (Figura III-3), como se puede apreciar en la Figura IV-3.

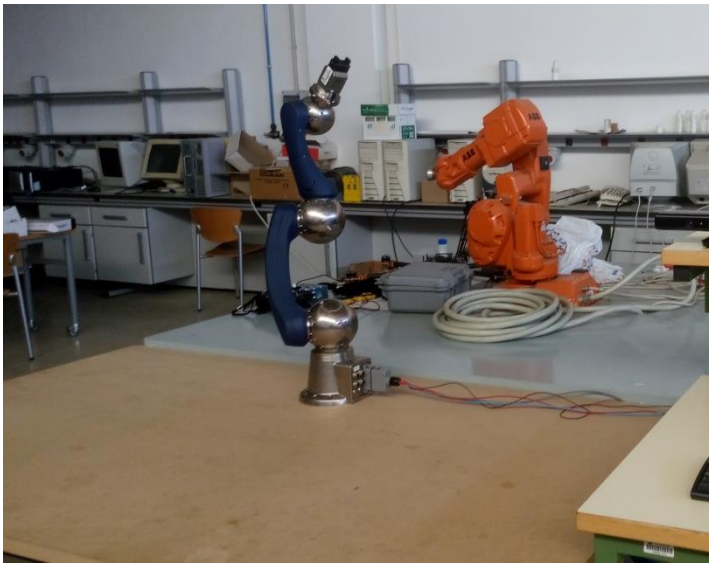


Figura IV-3. Área de trabajo del brazo robot

9.2. Comprobaciones iniciales

Se realizaron una serie de pruebas para comprobar el correcto funcionamiento del brazo robot. También tuvo que ser configurado para una conexión a través de CANopen.

9.2.1. Actualización firmware

Para poder usar el último software y drivers de CANopen en Ubuntu, fue necesaria una actualización internada de cada uno de los módulos que componen el brazo robot. Se usó el software Motion Tool Schunk [52], el cual permite operar y programar módulos SCHUNK, puede ser descargado desde su página oficial.

Para actualizar el software fue necesario un reinicio de la configuración de cada uno de los módulos, para posteriormente realizar una comunicación por puerto

serie. Se abrieron cada una de las cubiertas de las ERB, como se muestra en la Figura IV-4.

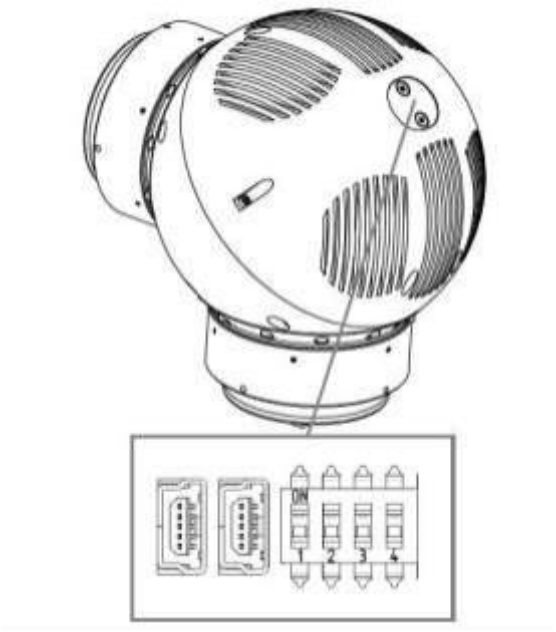


Figura IV-4. Esquema ERB y configuración Switch

Los pasos seguidos fueron los siguientes:

1. Apagar la alimentación del dispositivo.
2. Seleccionar la configuración "DEFAULT" del DIP-Switch (ver Tabla IV-1)
3. Activar la alimentación del robot durante unos segundos y volverla a apagar.
4. Volver a la configuración inicial del DIP-Switch (todos OFF).
5. Activar la alimentación del robot
6. Conectar módulo mediante conexión mini-USB al PC e instalar los controladores.
7. Iniciar la herramienta MTS (Motion Tool Schunk).

Tabla IV-1. Configuraciones DIP-Switch

DIP-Switch	Función
1	Terminación CAN Bus
2	Boot Input side
3	Boot Output side
4	Default

Una vez iniciado el software, se comprobó su versión actual, para ello se realizaron los siguientes pasos:

- Se establece una comunicación con uno de los módulos, para ello en Settings>Open Communication, marcando la opción "SCHUNK-USB" con un baudrate de 9600.

- Una vez establecida la comunicación con el dispositivo SCHUNK, se conecta cada uno de los módulo a actualizar (Module > Scan Bus),

Al cargar uno de los módulos se obtiene un resumen de las principales características de este modulo como su tipo, numero de serie, versión software o la versión del protocolo, como se puede ver en la Figura IV-5.

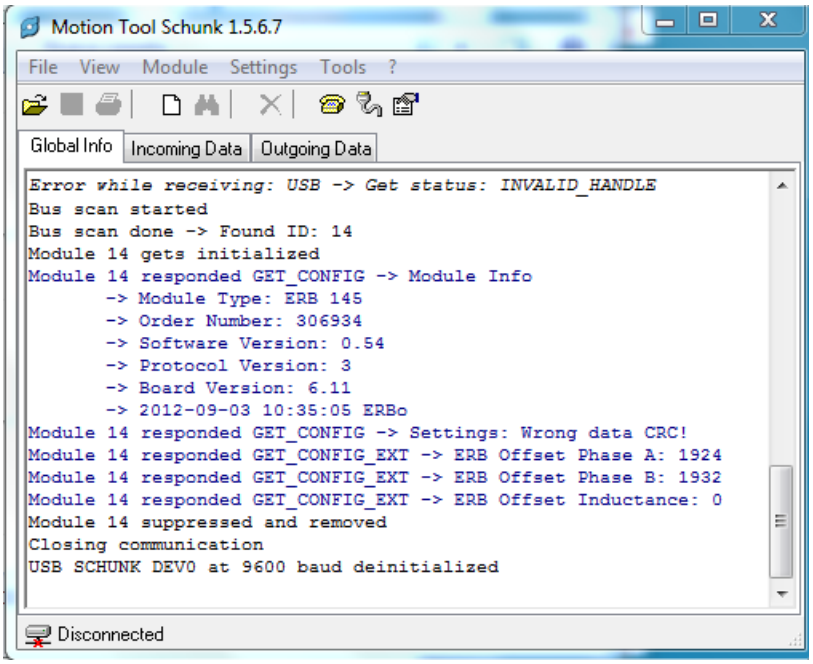


Figura IV-5. Obtención de la configuración inicial de los módulos mediante el software MTS.

En la Figura IV-6, se muestra la versión software 0.54, para uno de los módulos. Se actualizó mediante la herramienta "Update firmware" del software MTS a la versión 0.63 mediante un archivo .flash proporcionado por el distribuidor.

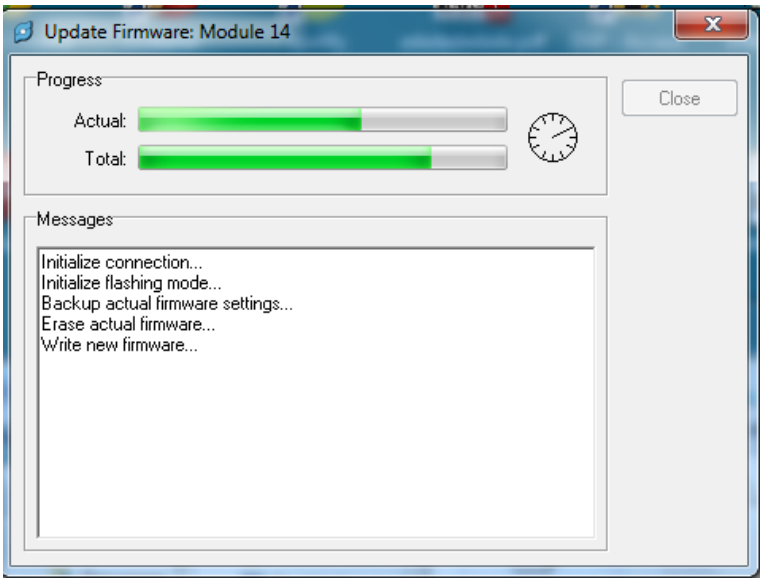


Figura IV-6. Proceso de actualización firmware para uno de los módulos mediante la herramienta MTS.

Al finalizar la actualización, se pudo comprobar que fue implementado correctamente, mediante una nueva conexión al módulo (Figura IV-7).

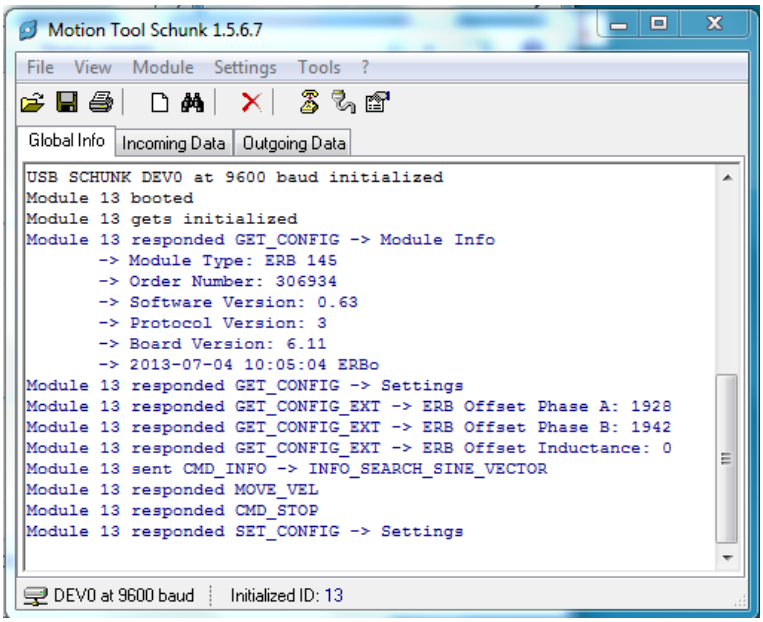


Figura IV-7. Configuración actualizada de los módulos mediante el software MTS.

Este proceso fue repetido, para cada uno de los módulos, un total de ocho módulos.

9.2.2. Comunicación por puerto serie

La comunicación inicial realizada fue mediante comunicación por puerto serial. Los nodos quedaron configurados a modo serial tras el reinicio de los módulos, según se muestra en la Figura IV-8.

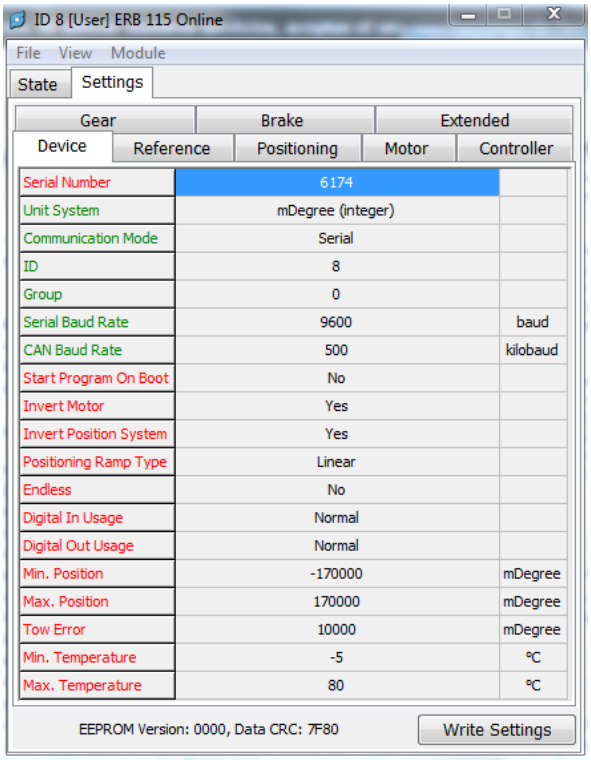


Figura IV-8. Configuración módulo en modo serial.

Los parámetros no son determinantes a la hora de realizar una comunicación por puerto serie módulo a módulo, por tanto, se pueden dejar por defecto tal cual se configuran tras el reinicio. Estos parámetros serán descritos en el apartado de CANopen.

La herramienta MTS, integra una interfaz para poder realizar pruebas (Figura IV-9), tanto de velocidad, como de posición en cada uno de los módulos del robot de manera individual.

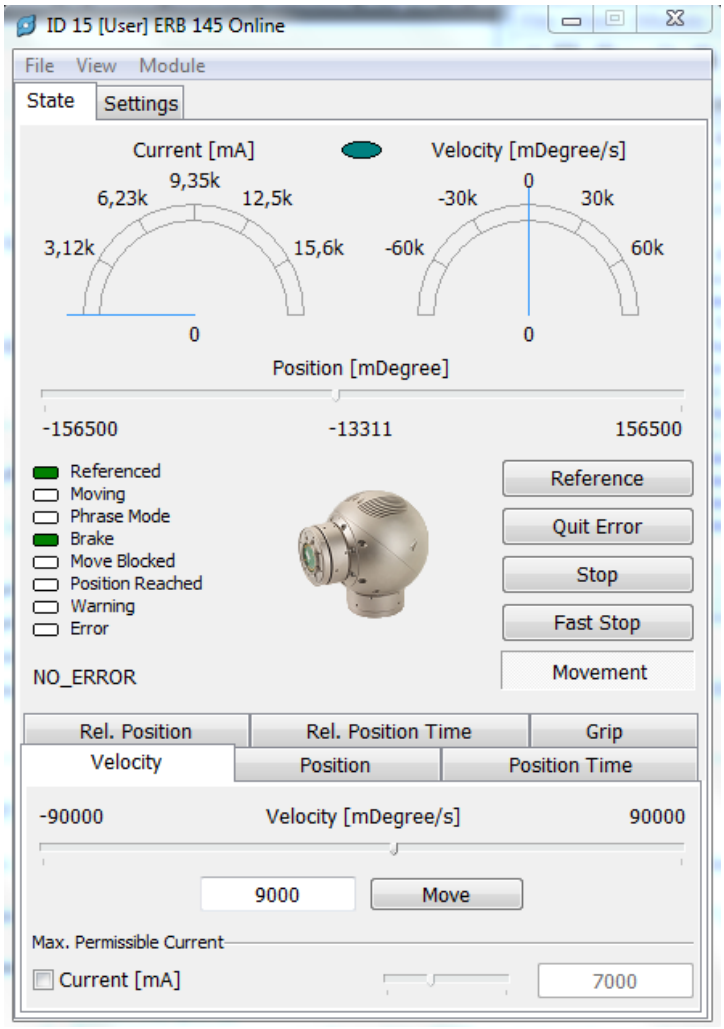


Figura IV-9. Control mediante puerto serie a través del software MTS.

Se realizaron ensayos en cada uno de los módulos, comprobando que funcionan correctamente.

9.2.3. Comunicación mediante CANopen

Para realizar una comunicación CANopen, se tuvieron que configurar cada uno de los módulos, según se muestra en la Figura IV-10.

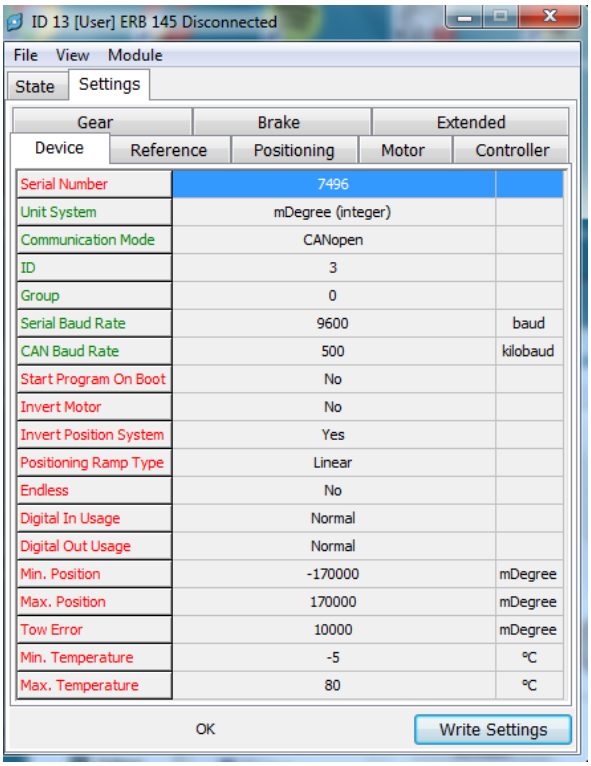


Figura IV-10. Configuración para una comunicación CANopen

Los parámetros principales a configurar fueron los siguientes:

- Modo de comunicación (*Communication Mode*). Se selecciona una comunicación del tipo CANopen, con el fin de controlar cada uno de los módulos de manera conjunta mediante una única salida CANopen.

- Número de identificación (*ID*). Al tratarse de una comunicación por un canal común, es necesario definir cada uno de los módulos según un número identificativo. La configuración establecida se puede ver en la Figura IV-11.

- Grupo (*Group*). Engloba todas las IDs en un mismo grupo. Se dejara en 0, ya que los números de identificación se definirán en un mismo grupo.

- Tasa de baudios por puerto serie (*Serial Baud Rate*). Es el número de unidades de señal por segundo, se establece en 9600 baud (valor por defecto).

- Tasa de baudios CAN (*CAN Baud Rate*). Número de unidades de señal por segundo por el puerto CAN, se fija en 500 kilobaud (valor por defecto).

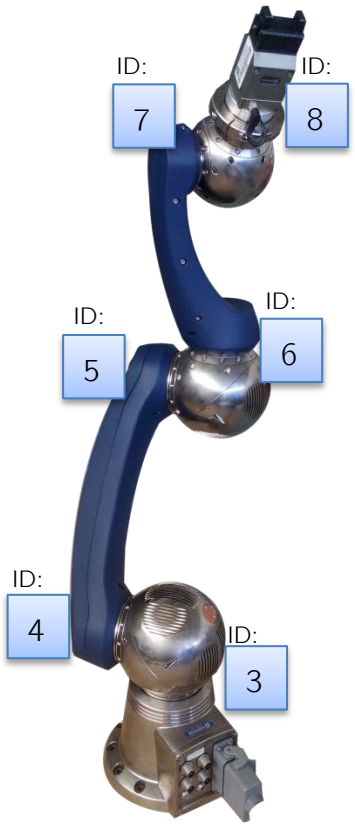


Figura IV-11. Configuración IDs para cada uno de los módulos.

Una vez configurados para cada uno de los módulos, se realizó una comunicación con el brazo robot a través del paquete de comunicación `ipa_canopen`, descrito en el apartado 8.3.6. Para ello es necesario haber instalado los drivers del adaptador CAN-USB, el proceso de instalación será descrito en el apartado 10.2.1.

Dentro de stack `ipa_canopen` se usará elejecutable move_device, del paquete `ipa_canopen_core`. Para localizar este archivo se ejecutan los siguientes comandos:

- Localizar ubicación del paquete `ipa_canopen_core`

```
$ roscd ipa_canopen_core
```

- Dentro de la carpeta `bin`, se encuentra el ejecutable move_device

```
$ cd bin
```

Los argumentos del ejecutable move_device son los siguientes:

```
./move device [device file] [CAN device ID] [Baud Rate] [sync
rate (msec)] [target velocity (rad/sec)] [acceleration
(rad/sec^2)]
```

Ejemplo de movimiento de la base:

```
alumno@ROS bin$ ./move_device /dev/pcan32 3 500k 10 0.2 0.05
Interrupt motion with Ctrl-C
Resetting devices
Waiting for Node: 3 to become available
Bootup received. Node-ID = 4
Not found
Ignoring
Bootup received. Node-ID = 5
Not found
Ignoring
Bootup received. Node-ID = 6
Not found
Ignoring
Bootup received. Node-ID = 3
Not found
Ignoring
Bootup received. Node-ID = 7
Not found
Ignoring
Bootup received. Node-ID = 8
Not found
Ignoring
Node: 3 is now available
Initializad the PDO mapping for Node:3
Concluded driver side init succesfully for Node3
Accelerating to target velocity
Target velocity reached!
```

En [61] se muestra un vídeo de comunicación a partir de `ipa_canopen_core`.

10. Configuración del controlador del robot

Una de las características de ROS es la de poder ejecutar y procesar datos en núcleos independientes en un procesador multinúcleo. Esta característica no es usada de manera automática por ROS, es definida en el código de cada nodo.

ROS puede ser ejecutado e instalado en cualquier sistema operativo basado en Linux, la distribución recomendada es la de Ubuntu.

Por otra parte, el simulador Gazebo requiere de una tarjeta gráfica dedicada con soporte para gráficos drivers OpenGL 3D.

10.1. Instalación ROS

En este apartado se describirá brevemente el proceso realizado en el equipo de trabajo para la instalación de ROS.

La versión de Ubuntu usada ha sido la 12.04 (Precise) en su versión de 32bits, debido a que es la versión más estable de entre las versiones soportadas, se trata de una versión LTS (*Long Term Support*), recibiendo soporte continuamente.

La versión de ROS usada es la versión Groovy, ya que es la última versión de ROS soportada por el *stack* `schunk_robots` (apartado 8.3.3).

A continuación se describe el proceso de instalación de ROS:

- En primer lugar se añade al gestor de paquetes de Ubuntu la dirección para obtener los paquetes ROS y sus actualizaciones, para ello es necesario editar el archivo */etc/apt/sources.list.d*:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
precise main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Se establecen las firmas, sirven para comprobar que el paquete descargado no ha sido modificado por terceros, son similares a las usadas en las comunicaciones SSH:

```
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

- Se actualizan los paquetes del gestor.

```
$ sudo apt-get update
```

- Una vez actualizados los repositorios de Ubuntu, se pasa a descargar la versión de ROS deseada, mediante el siguiente comando:

```
$ sudo apt-get install ros-groovy-desktop-full
```

Esta versión incluye los siguientes componentes: ROS, rqt, RViz, bibliotecas genéricas, simuladores 2D/3D, navegación y percepción 2D/3D

Una vez instalado ROS se debe inicializar rosdep, necesario para instalar fácilmente las dependencias para los archivos fuente a compilar, también es usado por componentes ROS.

```
$ sudo rosdep init
$ rosdep update
```

Por otra parte, es recomendado instalar rosinstall, un complemento para la descarga de árboles de código fuente de paquetes ROS a partir de un solo comando:

```
$ sudo apt-get install python-rosinstall
```

10.1.1. Configuración del entorno ROS

En esta sección se trata la configuración de las variables de entorno de ROS, en ellas se describe la localización de sus paquetes.

Las variables de entorno proporcionan al usuario el acceso a los comandos internos de ROS a través del terminal, basta con ejecutar el archivo siguiente cada vez que se quiera usar uno de estos comandos.

```
$ source /opt/ros/hydro/setup.bash
```

Este comando puede ser implementado de manera que no sea necesario que sea escrito en consola cada vez que se necesite usar un comando ROS, para ello se añade al archivo `/home/"usuario"/.bashrc` el comando nombrado anteriormente.

Para poder compilar paquetes ROS es necesario crear un entorno de trabajo mediante `catkin` [9], este paquete viene instalado por defecto con ROS. El entorno de trabajo lo se define dentro de la carpeta `/groovy_workspace` dentro de la carpeta `home` del usuario.

```
$ mkdir -p ~/groovy_workspace/catkin_ws/src
$ cd ~/ groovy_workspace/catkin_ws/src
$ catkin_init_workspace
```

Se construye el espacio de trabajo

```
$ cd  ~/groovy_workspace/catkin_ws
$ catkin_make
```

En la carpeta `src`, dentro de la carpeta `catkin` se incluyen los paquetes ROS a compilar.

Para poder indicar al sistema ROS que hay paquetes localizados en un entorno catkin, se ejecuta el siguiente comando:

```
$source devel/setup.bash
```

Con el fin de integrar el entorno de trabajo `catkin` en `/groovy_workspace` según se muestra en la Figura IV-12, se han añadido al archivo `.bashrc`, las siguientes líneas:

```
## ROS Groovy
source /opt/ros/groovy/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/groovy_workspace
export ROS_WORKSPACE=~/groovy_workspace
# Catkin workspace
source ~/groovy_workspace/catkin_ws/devel/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/groovy_workspace
```

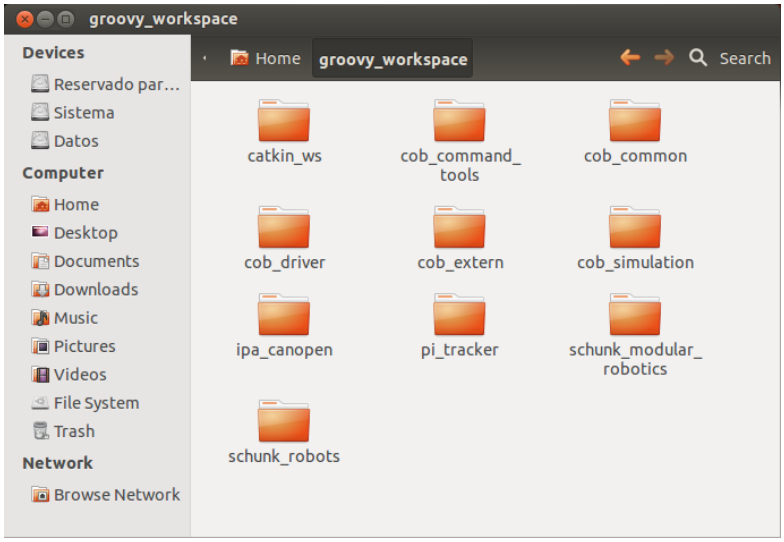


Figura IV-12. Entorno de trabajo ROS

10.2. Instalación de controladores

En esta sección se describirá el proceso de instalación de los controladores necesarios para establecer una comunicación con el ordenador.

10.2.1. Controladores CANopen

Estos controladores permiten establecer una conexión con el brazo robótico a través del adaptador CAN-USB descrito en el apartado 7.2.

La versión a instalada en el sistema ha sido la última versión estable, obtenida de la página oficial del proveedor del adaptador [40], versión 7.12 con fecha del 23 de Julio de 2014. Los drivers funcionan para cualquier versión de Linux con un Kernel desde su versión 2.4 hasta cualquier versión 3.

Una vez descargado el archivo anterior se descomprime en la carpeta `/opt`, es importante que para versiones del Kernel mayores a la 2.6.25, añadir el parámetro `NET=NO_NETDEV_SUPPORT`.

```
$ cd opt/peak-linux-driver-7.X
$ make NET=NO_NETDEV_SUPPORT
$ sudo make install
```

Tras reiniciar el equipo y conectar el adaptador se comprueba que se ha instalado correctamente. Para comprobar que se ha cargado el módulo correctamente.

```
$ lsmod | grep pcan
```

Se comprueba que ha sido compilado con `NO NETDEV SUPPORT`. Para ello se debe mostrar debajo de ndev un –NA- tras ejecutar el comando que se muestra a continuación

```
$ cat /proc/pcan
*------------- PEAK-System CAN interfaces (www.peak-system.com) -------------
*------------------------- Release_20120319_n (7.12.0) ----------------------
*---------------- [mod] [isa] [pci] [dng] [par] [usb] [pcc] -----------------
*--------------------- 1 interfaces @ major 250 found -----------------------
*n -type- ndev --base-- irq --btr- --read-- --write- --irqs-- -errors- status
32    usb -NA- ffffffff 255 0x0014 00000005 00000034 00000168 00000075 0x000c
```

Una vez realizado estos pasos, el equipo está listo para una comunicación CANopen con el brazo LWA-4p a través del adaptador.

10.2.2. Controladores Kinect

Para lograr una comunicación del sensor Kinect con ROS, es necesario instalar una serie de drivers, que serán descritos a continuación.

Una vez instalado el paquete `openni_tracker`, es necesario instalar los drivers de Kinect, proporcionados por OpenNI en su paquete NITE. Para ello se descarga el paquete `Nite-bin-linux` [24], la versión instalada fue la 1.5.2.21, en su versión de 32 bits.

Se descomprime el archivo en la carpeta `/opt` en la raíz del sistema, con el nombre "kinect", y se realiza la instalación:

```
$ sudo ./install.sh
Installing NITE
***************

Copying shared libraries... OK
Copying includes... OK
Installing java bindings... OK
Installing module 'Features_1_3_0'...
Registering module 'libXnVFeatures_1_3_0.so'... OK
Installing module 'Features_1_3_1'...
Registering module 'libXnVFeatures_1_3_1.so'... OK
Installing module 'Features_1_4_1'...
Registering module 'libXnVFeatures_1_4_1.so'... OK
Installing module 'Features_1_4_2'...
Registering module 'libXnVFeatures_1_4_2.so'... OK
Installing module 'Features_1_5_2'...
Registering module 'libXnVFeatures_1_5_2.so'... OK
Copying XnVSceneServer... OK
Installing module 'Features_1_5_2'
registering module 'libXnVHandGenerator_1_3_0.so'...OK
Installing module 'Features_1_5_2'
registering module 'libXnVHandGenerator_1_3_1.so'...OK
Installing module 'Features_1_5_2'
registering module 'libXnVHandGenerator_1_4_1.so'...OK
Installing module 'Features_1_5_2'
registering module 'libXnVHandGenerator_1_4_2.so'...OK
Installing module 'Features_1_5_2'
registering module 'libXnVHandGenerator_1_5_2.so'...OK
Adding license.. OK

*** DONE ***
```

Se conecta el sensor Kinect al equipo y se comprueba que funciona correctamente, para ello se lanza mediante ROS el nodo `openni_tracker`.

Capítulo V.

V. Resultados y discusión

11. Resultados y discusión

En este capítulo se describirá el proceso realizado para implementar el sistema de comunicación entre el Kinect y el brazo robot, se incluirán el paquete ROS desarrollado, así como las simulaciones y los resultados obtenidos.

11.1. Integración de los dispositivos en el entorno ROS

El brazo robótico LWA 4p junto al sensor Kinect se han de integrar en el sistema de ROS de manera individual, para posteriormente enlazar ambos dispositivos a través del diseño un paquete ROS. En la ¡Error! La autoreferencia al marcador no es válida. se muestra el esquema de comunicación para cada uno de los dispositivos, la comunicación del brazo robot con ROS es bidireccional mientras que Kinect se comunica unidireccionalmente.

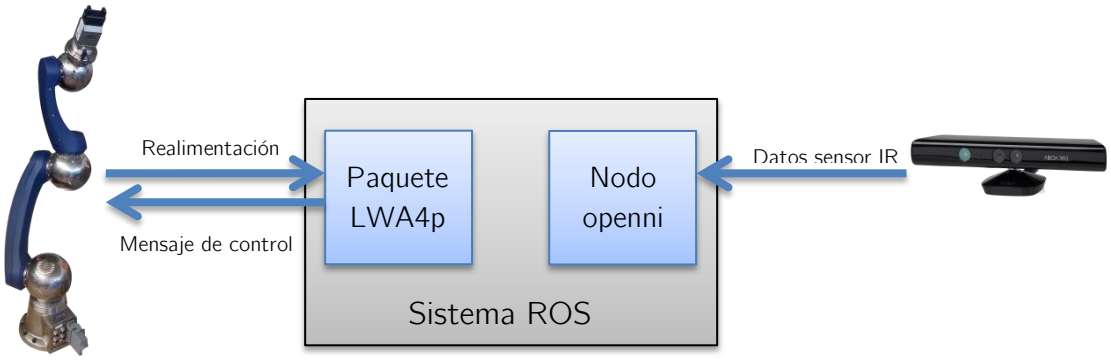


Figura V-1. Esquema simplificado de la comunicación entre los dispositivos y el sistema ROS de manera individual

11.1.1. Brazo robótico LWA 4p

Este elemento fue integrado en ROS con ayuda de algunos de los *stacks* y paquetes definidos, donde se definen los parámetros del robot y sus límites de trabajo, así como los nodos y archivos de ejecución (.launch) para su inicio.

11.1.1.1. Archivos de configuración

Los archivos de configuración del brazo robot LWA 4p se encuentran en la carpeta `config`, dentro del entorno de trabajo de ROS en `schunk_robots/schunk_lwa4p`.

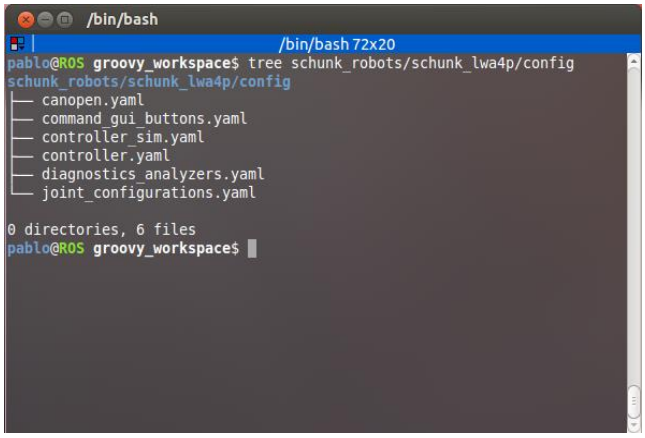


Figura V-2. Árbol de archivos de configuración del brazo robot LWA 4p

A continuación se describirán los archivos de configuración principales:

- Configuración CANopen (canopen.yaml). En este archivo se definen las variables necesarias para realizar una comunicación a través del adaptador CAN.

```
devices:
  - name: /dev/pcan32
    baudrate: 500K
    sync_interval: 10
# - name: /dev/pcan33
#   baudrate: 1M
#   sync_interval: 5
chains: ["arm_controller"]
```

- Configuración del controlador (controller.yaml). Se define los parámetros que definen el comportamiento del brazo robot para el control de trayectorias y movimientos.

```
# canopen parameter
can_module: PCAN
can_baudrate: 1000
max_accelerations: [0.8,0.8,0.8,0.8,0.8,0.8]
OperationMode: position

joint_names: ["arm_1_joint", "arm_2_joint", "arm_3_joint",
"arm_4_joint", "arm_5_joint", "arm_6_joint"]
module_ids: [3,4,5,6,7,8]
devices: ["/dev/pcan32", "/dev/pcan32",
"/dev/pcan32","/dev/pcan32","/dev/pcan32","/dev/pcan32"]

# trajectory controller parameters
ptp_vel: 0.4 # rad/sec
ptp_acc: 0.1 # rad/sec^2
max_error: 0.1 # rad
frequency: 100
```

- Configuración de las articulaciones (joint_configurations.yaml). Parámetros que definen los nombres dados a cada articulación y una serie de puntos y trayectorias predefinidas para que sean usadas de manera global tras el lanzamiento del robot.

```
joint_names: ["arm_1_joint","arm_2_joint","arm_3_joint",
"arm_4_joint","arm_5_joint","arm_6_joint"]

# back side positions
home: [[0,0,0,0,0,0]]
folded: [[-0.746, -0.569, -1.017, 0.036, 0.493, 0.0]]
waveleft: [[0.321, 0.499, -0.406, -0.237, 0, 0]]
waveright: [[0.474, -0.791, 0.00415, 0, 0, 0]]

# trajectories
wave: [waveleft,waveright,home,folded]
```

11.1.1.2. Modelo del robot (`lwa4p.urdf.xacro`)

Para generar cada modelo de robot en particular, se tiene que editar el archivo de formato URDF (*Unified Robot Description Format*) para especificar las dimensiones del robot, los movimientos de las articulaciones, parámetros físicos como masa e inercia, etc.

```
<!-- joint between arm_2_link and arm_3_link -->
            <joint name="${name}_3_joint" type="revolute">
                  <origin xyz="0 0.350 0" rpy="0 3.14159 0"/>
                  <parent link="${name}_2_link"/>
                  <child link="${name}_3_link"/>
                  <axis xyz="0 0 1"/>
                  <calibration rising="${arm_3_ref}"/>
                  <dynamics damping="5" />
                  <limit effort="176" velocity="2.0" lower="-
6.2831853" upper="6.2831853"/>
                  <safety_controller k_position="20"
k_velocity="25" soft_lower_limit="${-6.2831853 + 0.01}"
soft_upper_limit="${6.2831853 - 0.01}" />
            </joint>

            <link name="${name}_3_link">
                  <inertial>
                        <origin xyz="0 0 0" rpy="0 0 0"/>
                        <mass value="2.1"/>
                        <inertia ixx="0.03" ixy="0" ixz="0"
iyy="0.03" iyz="0" izz="0.03" />
                  </inertial>
```

En el archivo `lwa4p.urdf.xacro` se definen las limitaciones software definidas en la Tabla V-1.

Tabla V-1. Limite software y hardware para el robot LWA 4p.

	arm_1_joint	**arm_2_joint**	**arm_3_joint**	**arm_4_joint**	**arm_5_joint**	**arm_6_joint**
	lower/upper	**lower/upper**	**lower/upper**	**lower/upper**	**lower/upper**	**lower/upper**
urdf limit	-6,25/6,25	-1,97/1,97	-6,25/6,25	-1,97/1,97	-6,25/6,25	-1,97/1,97
hardware limit	-6,28/6,28	-2/2	-6,28/6,28	-2/2	-6,28/6,28	-2/2

11.1.1.3. Archivos de ejecución (.launch)

Los archivos *launch* ejecutan una serie de nodos junto a unos parámetros definidos (.yaml) de manera simultánea y concurrida.

- Ejecución del brazo robot real (`lwa4p.launch`). Ejecuta los nodos necesarios para establecer una comunicación con el brazo robot a través de ROS, así como los archivos de configuración comentados en apartado 11.1.1.1.

- Ejecución del brazo robot simulado en Gazebo (`lwa4p_sim.launch`). Ejecuta los nodos, archivos de configuración y un entorno en Gazebo con el modelo del brazo robot LWA 4p.

11.1.1.4. Grafos de comunicación

Es una representación de la comunicación realizada a través de ROS entre los distintos nodos.

- Comunicación con el robot real (Figura V-3).

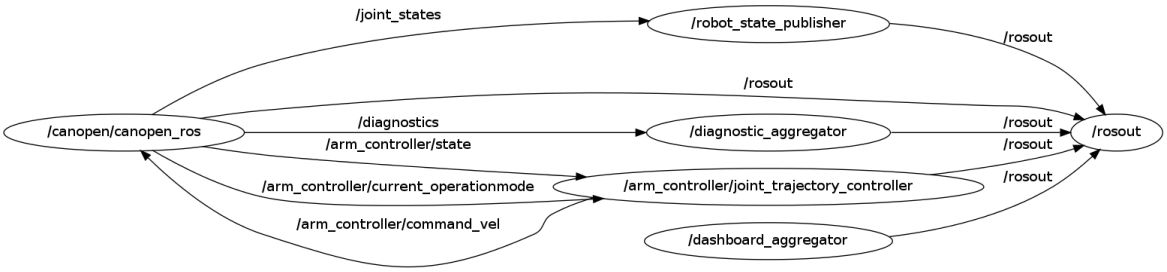


Figura V-3. Grafo de comunicación del robot real

- Comunicación en simulación (Figura V-4).

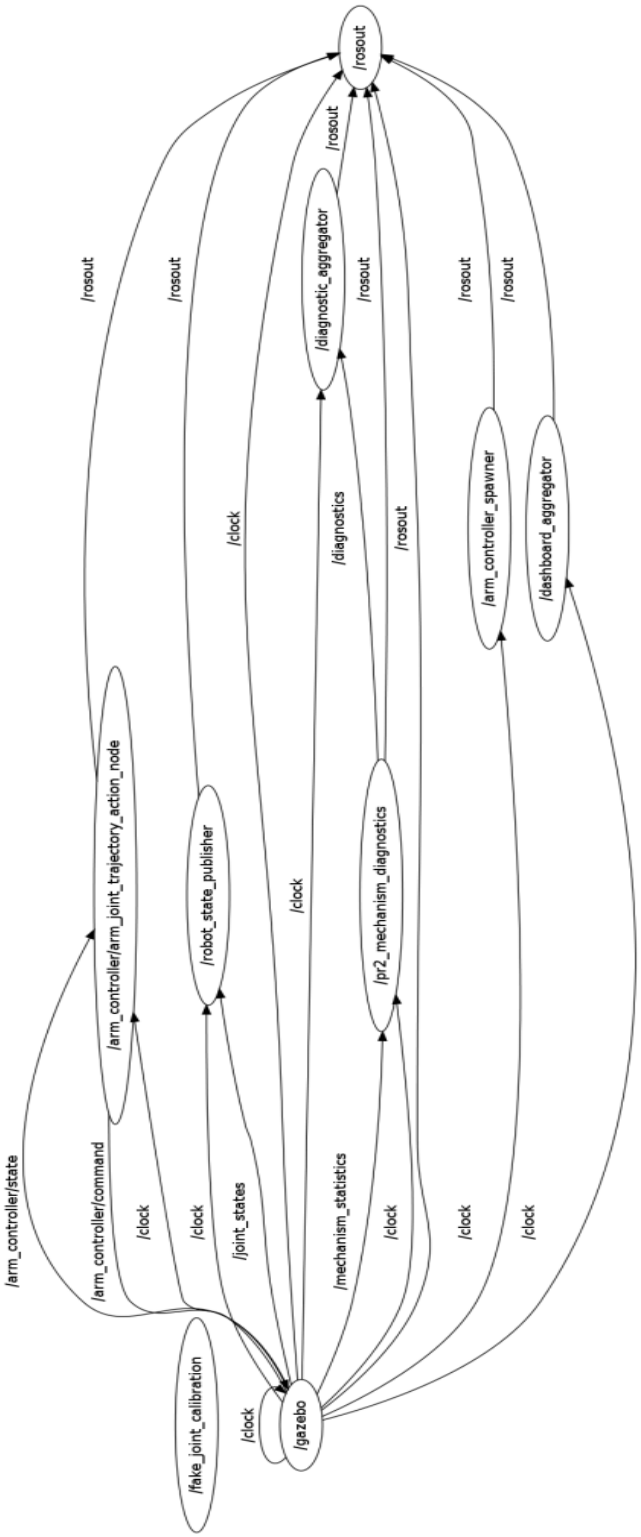


Figura V-4. Grafo de comunicación del brazo robot en simulación

11.1.2. Sensor Kinect

El sensor Kinect fue integrado dentro del entorno ROS a través del paquete `openni_tracker` [36]. Los elementos de los que se compone serán descritos a continuación, así como la ejecución del mismo y su grafo de comunicación.

11.1.2.1. Nodo `openni_tracker`

La adquisición de datos mediante ROS a partir del sensor Kinect es realizada por este único nodo. Su función es capturar toda la información y empaquetarla en un mensaje del tipo tf y publicada mediante *broadcast* a través de todo el sistema ROS.

11.1.2.2. Mensajes

El mensaje proporcionado es del tipo `tf/tfMessage`, este mensaje coincide con la estructura tratada en el apartado 8.3.1. Se pude realizar una captura del mensaje transmitido mediante el siguiente comando:

```
$ rostopic echo /tf
```

Obteniendo a la salida una serie de transformadas, una para cada una de las articulaciones detectadas, a continuación se muestra un ejemplo:

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1410539768
        nsecs: 266649920
      frame_id: /openni_depth_frame
    child_frame_id: /right_shoulder_1
    transform:
      translation:
        x: 0.873967100138
        y: -0.203011377143
        z: -0.0539883329387
      rotation:
        x: 0.974136188072
        y: -0.17991576506
        z: 0.119451796387
        w: 0.0664851330034
```

11.1.2.3. Ejecución

Para iniciar el nodo es necesario haber ejecutado roscore, Los pasos para ejecutar este nodo son los siguientes:

```
$ roscore
$ rosrun openni_tracker openni_tracker
```

El nodo `openni_tracker` muestra en pantalla los eventos actuales detectados según muestra la Figura V-5.

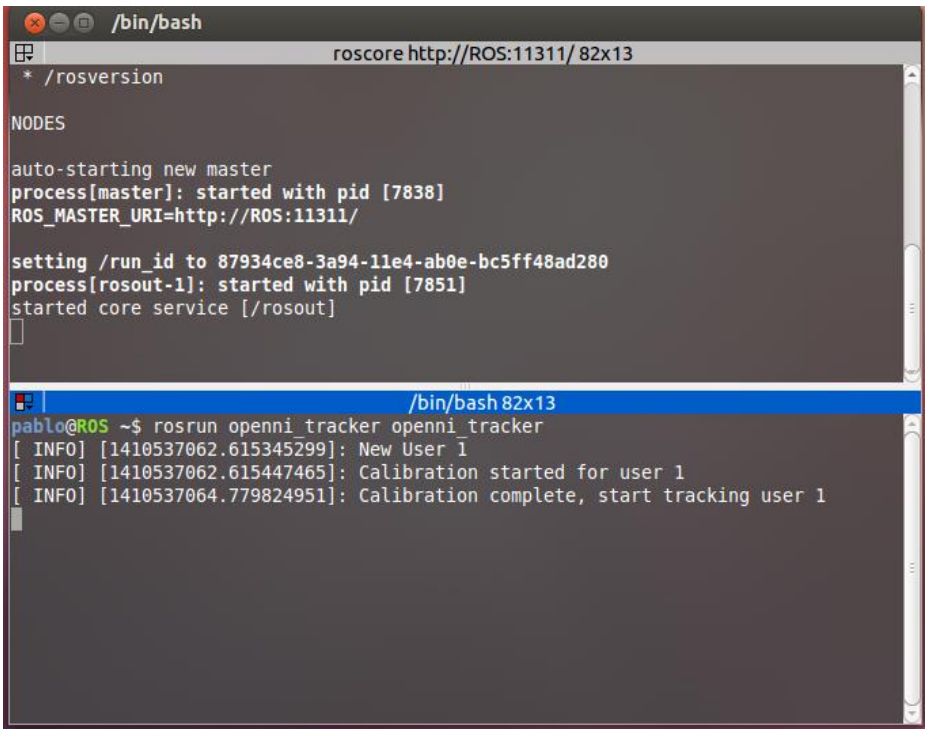


Figura V-5. Ejecución del nodo `openni_tracker`

11.1.2.4. Esquema de grafos

En esquema de grafos muestra a su vez la lista de tópicos, se representa en la Figura V-6.

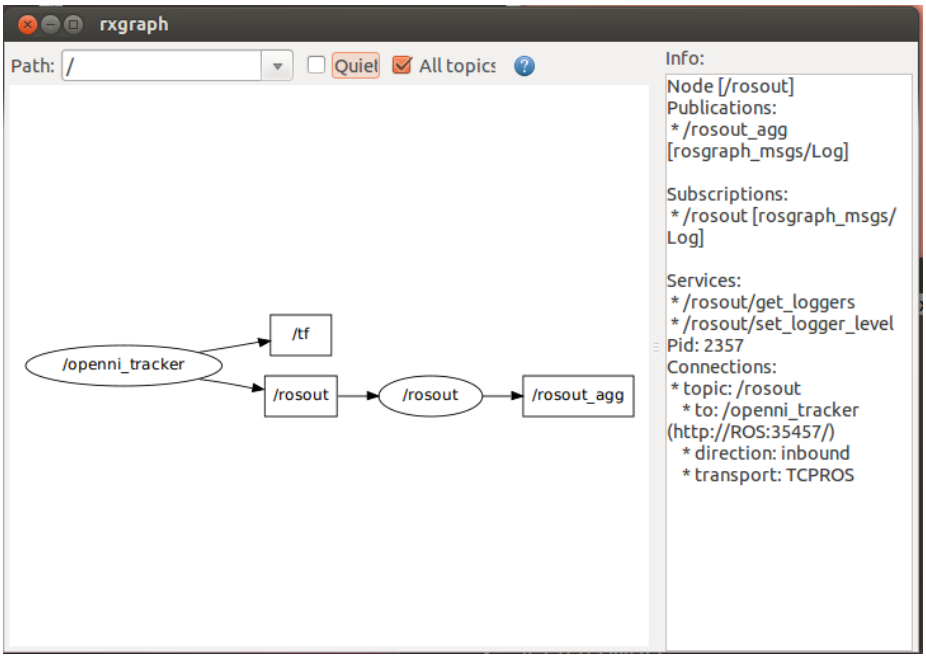


Figura V-6. Grafo de comunicación del nodo `openni_tracker`

Se trata de un nodo simple, donde el nodo `/openni_tracker` realiza una captura de las transformadas de las articulaciones del usuario y las transmite mediante *broadcast* a todo el sistema ROS.

11.2. Ensayos iniciales

Se realizaron una serie de pruebas antes de desarrollar el paquete final, el objetivo de estos ensayos era el de poder abordar con mayor facilidad la implementación de los nodos de comunicación.

11.2.1. Tratamiento de los mensajes *tf* mediante RViz

Para poder interpretar correctamente los datos adquiridos a través del sensor Kinect se usó la herramienta RViz descrita en el apartado 7.5. A continuación se describirán los pasos necesarios:

Se ejecuta uno de los nodos de seguimiento:

```
$ roslaunch ki2_arm skeletal.launch
```

o

```
$ roscore
$ rosrun openni_tracker openni_tracker
```

Tras ello ejecutamos RViz:

```
$ rosrun rviz rviz
```

Una vez ejecutado el programa, para lograr una correcta interpretación de las transformadas publicadas se selecciona `/openni_depth_frame` como *frame* fijo. Este frame corresponde a la posición de la camara, de tal modo que todas las articulaciones capturadas serán representadas respecto a este frame fijo.

Tras ello, se añade el elemento TF al entorno RViz, con el fin de obtener una lectura de los mensajes de cada articulación y procesarlas graficamente (Figura V-7 y Figura V-8).

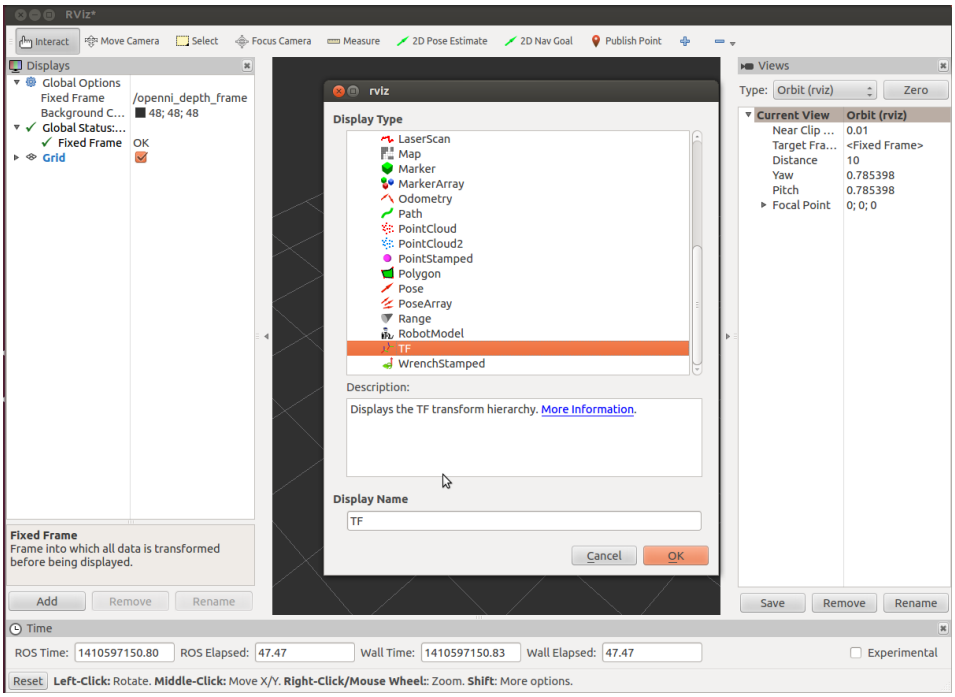


Figura V-7. Configuración inicial RViz para la representación de los mensajes tf

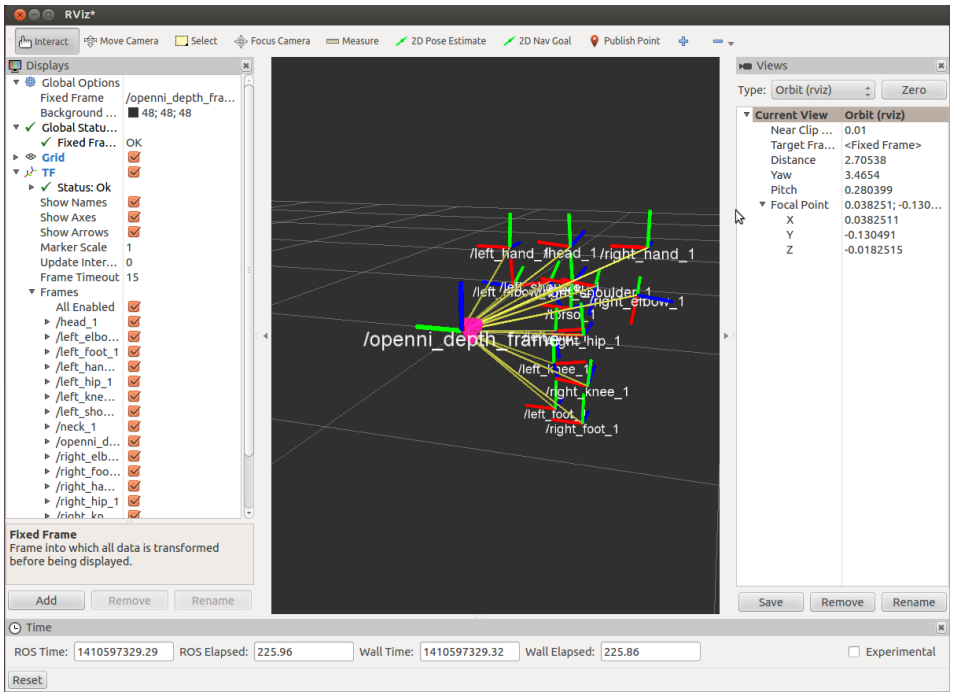


Figura V-8. Representación de las transformadas en RViz con el frame fijo /openni_depth_frame

Se seleccionan los *frames* que se desean representar, en este caso los *frames* del brazo derecho: `/right_shoulder`, `/right_elbow` y `/right_hand` (Figura V-9).

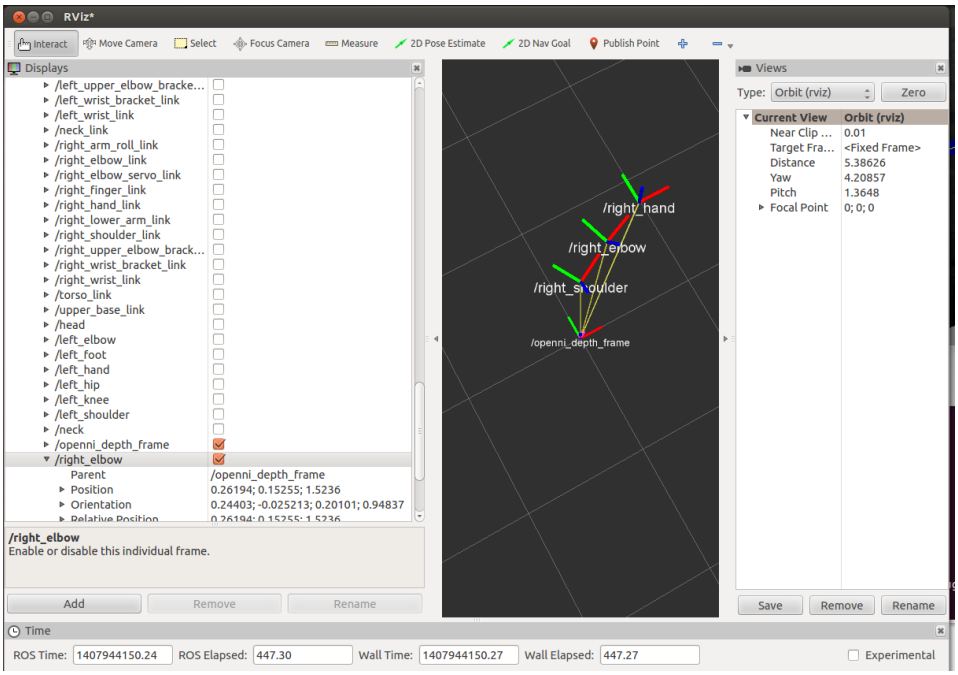


Figura V-9. Representación de los frames del brazo derecho del usuario en RViz

11.2.2. Determinación del tiempo de muestreo

A la hora de seleccionar un tiempo de muestreo para elegir la frecuencia de escucha, se realizó una prueba mediante el nodo `view_frame` del paquete `TF`.

```
$ roscore
$ rosrun tf view_frames
```

Este nodo da como salida un resumen de todos los datos recogidos de la escucha de los mensajes `tf` durante cinco segundos.

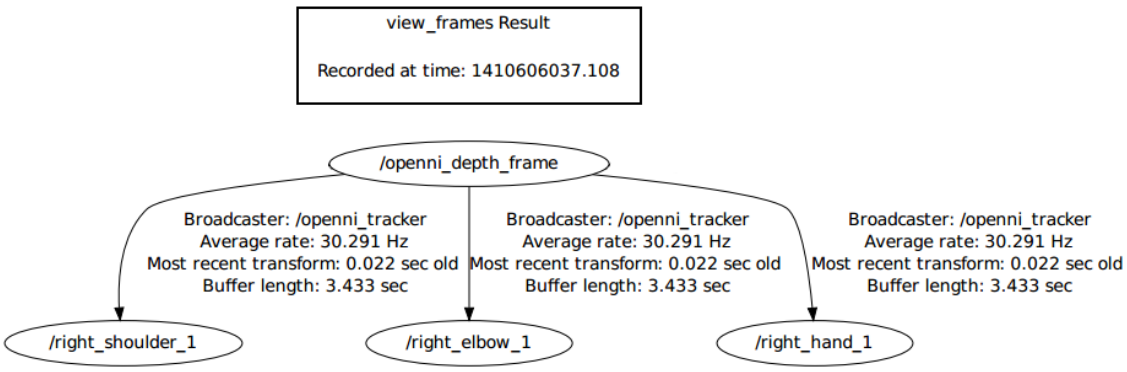


Figura V-10. Características principales de los *frames* publicados

Como se puede apreciar en la Figura V-10, con el fin de no obtener datos redundantes en el nodo de escucha el intervalo de muestreo debe ser menor de 30.0291Hz.

11.3. Desarrollo del paquete ROS `ki2_arm`

La función del paquete ki2_arm es la de comunicar los elementos integrados en el sistema ROS (apartado 11.1) y realizar una comunicación directa entre Kinect y el brazo robótico LWA 4p (Figura V-11). Para ello se dará uso de nodos, mensajes y archivos de ejecución de ROS.

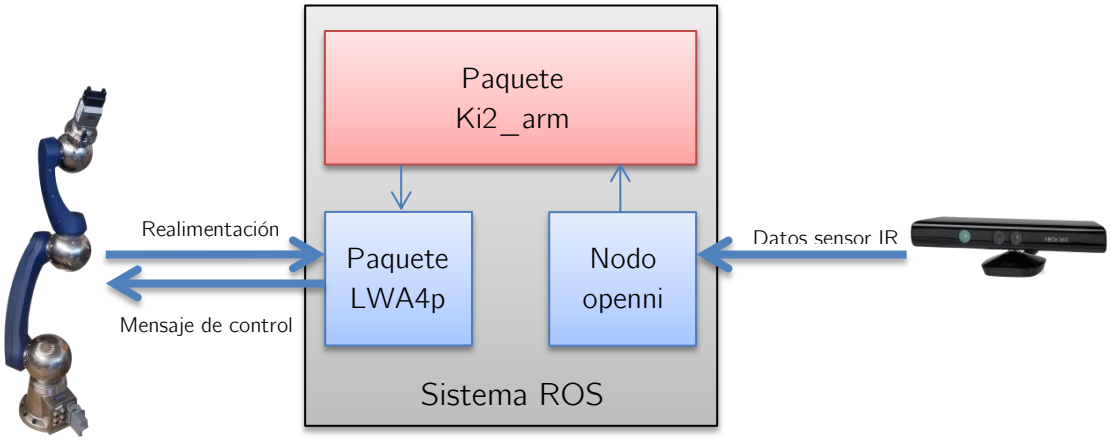


Figura V-11. Esquema simplificado de la comunicación entre Kinect y el brazo robot por medio del paquete ki2_arm

11.3.1. Nodos

El paquete se compone de dos nodos principales: uno para la escucha y tratamiento de la publicación *tf* de Kinect y otro para la detección de movimiento del usuario y su posterior publicación al brazo robot.

11.3.1.1. `Skeletal_tracking`

Se trata de una versión modificada del nodo original `openni_tracker`, incluye una representación de las coordenadas capturadas del usuario. Su fin es el de informar al usuario si se le está reconociendo correctamente. En la interfaz se representa el número de usuario reconocido junto a su estado actual: calibrando o siguiendo (Figura V-12).

A su vez, el grafo de comunicación incorpora un nuevo nodo para la representación gráfica de los datos obtenidos (Figura V-13).

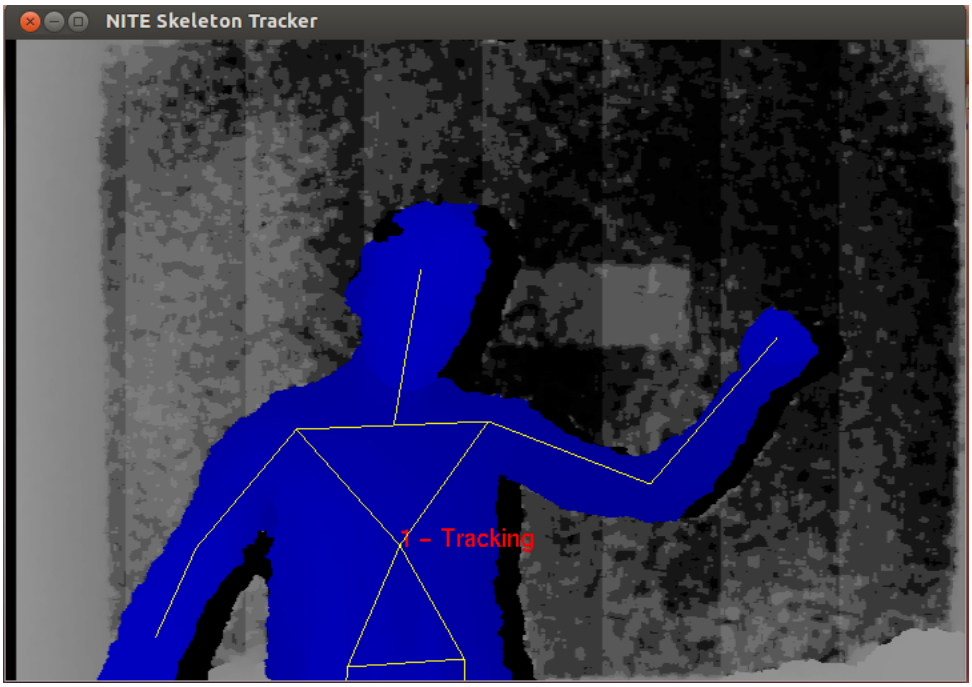


Figura V-12. Interfaz del nodo skeleton_tracking.

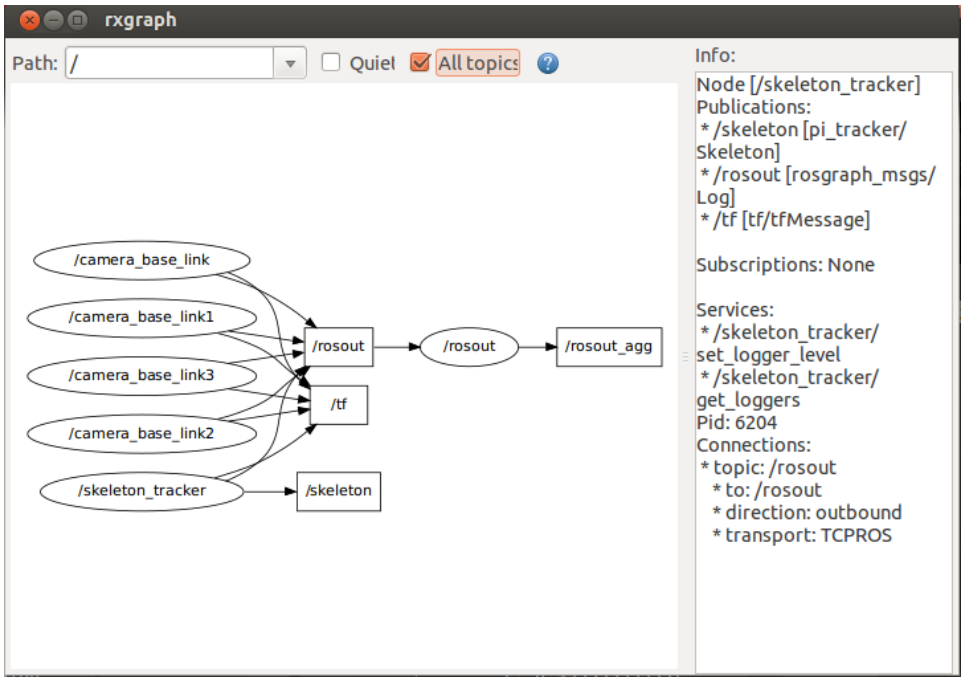


Figura V-13. Grafos nodo skeleton_tracker

11.3.1.2. *kinect*_tf_listener_pub

Realiza una escucha los mensajes `tf` realizados por Kinect, donde se incluye una estructura de datos definida en el apartado 4.1.2. Será necesario una transformación con el fin de establecer un sistema de coordenadas equivalente entre en brazo del usuario y el brazo robot.

A continuación se detalla el código del nodo a implementar

```
if __name__ == '__main__':

    pub = rospy.Publisher('arm_pos', ArmPos)
    rospy.init_node('tf_Kinect_listener_pub')
    listener = tf.TransformListener()
```

```
rate = rospy.Rate(10.0)
   while not rospy.is_shutdown():
      try:
        (trans0,rot0) =
             listener.lookupTransform('/openni_depth_frame',
             '/right_shoulder', rospy.Time())
        (trans1,rot1) =
              listener.lookupTransform('/openni_depth_frame',
              '/right_elbow', rospy.Time())
        (trans2,rot2) =
               listener.lookupTransform('/openni_depth_frame',
               '/right_hand', rospy.Time())
      except (tf.LookupException, tf.ConnectivityException):
         continue
```

Obtención de la posición y orientación de los frames: /right_shoulder, /right_elbow y /right_hand respecto al frame fijo /openni_depth_frame. Para ello se usa la function lookupTransform, que devuelve la posición y orientación de un frame respecto a otro.

Una vez obtenidos los puntos de cada articulación se calculan los vectores que forman entre ellos.

$$\vec{V}_1 = trans_{elbow}(x,y,z) - trans_{shoulder}(x,y,z)$$

$$\vec{V}_2 = trans_{hand}(x,y,z) - trans_{elbow}(x,y,z)$$

$$\vec{V}_3 = trans_{hand}(x,y,z) - trans_{shoulder}(x,y,z)$$

En las Figura V-14 y Figura V-15 se realiza una representación de los vectores formados en 3D y los puntos correspondientes a los `frames` fijos.

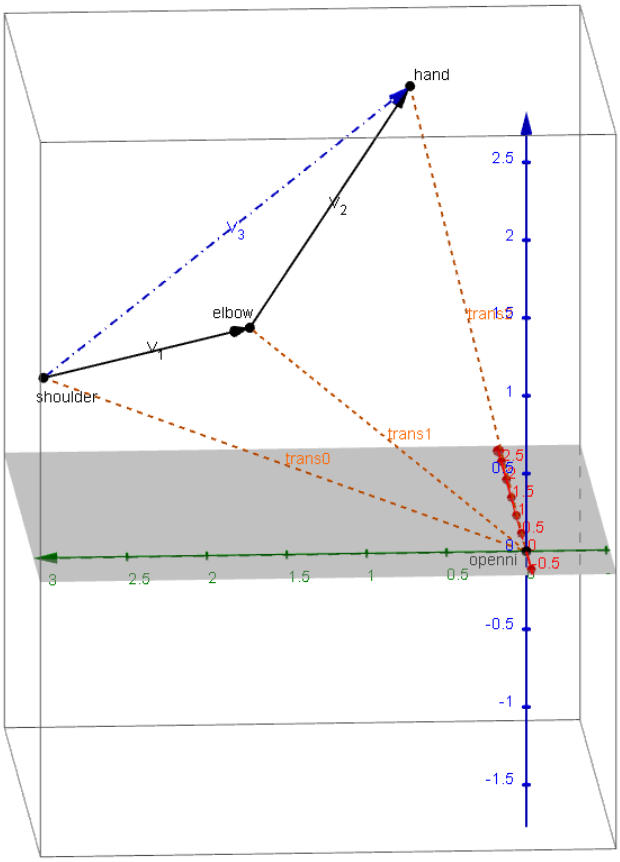


Figura V-14. Representación del sistema de coordenadas en GeoGebra

Su implementación en código es la siguiente:

```
    v1 = np.zeros(3)
    v2 = np.zeros(3)
    v3 = np.zeros(3)

    for k in range(len(v1)):
        v1[k] = trans1[k] - trans0[k]

    for m in range(len(v2)):
        v2[m] = trans2[m] - trans1[m]

    for n in range(len(v2)):
        v2[m] = trans2[m] - trans0[m]
```

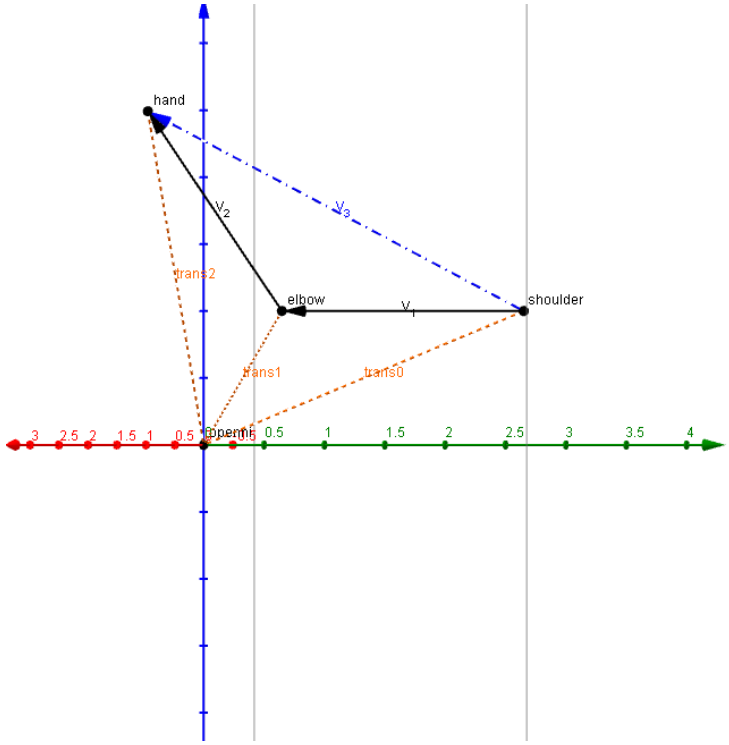


Figura V-15.Representación del sistema de coordenadas visto de perfil en GeoGebra

El cálculo de los ángulos equivalentes a las articulaciones del brazo robot (Figura V-17) fue realizado a partir de las siguientes operaciones vectoriales.

- Giro de la base (θ). Se realiza el coseno director del vector $\vec{V}_3$ respecto al eje y, se le resta 90º para establecer el punto intermedio de la base del robot con el punto intermedio en la rotación del hombro del usuario.

$$\theta = arc\cos\left(\frac{\vec{V}_3(y)}{|\vec{V}_3|}\right) - \frac{\pi}{2}$$

- Ángulo de elevación del primer eslabón (β). Se realiza a partir del coseno director del vector $\vec{V}_1$ sobre el eje x.

$$\beta = arc\cos\left(\frac{\vec{V}_1(x)}{|\vec{V}_1|}\right)$$

- Ángulo formado entre el eslabón uno y dos (α). Equivale al ángulo formado entre los vectores $\vec{V}_1$ y $\vec{V}_2$.

$$\alpha = arc\cos\left(\frac{\vec{V}_1 \cdot \vec{V}_2}{|\vec{V}_1| \cdot |\vec{V}_2|}\right)$$

En la Figura V-16 se puede ver una representación en 3D de los vectores calculados anteriormente, el plano XY equivale al plano del robot donde se sitúa su base.

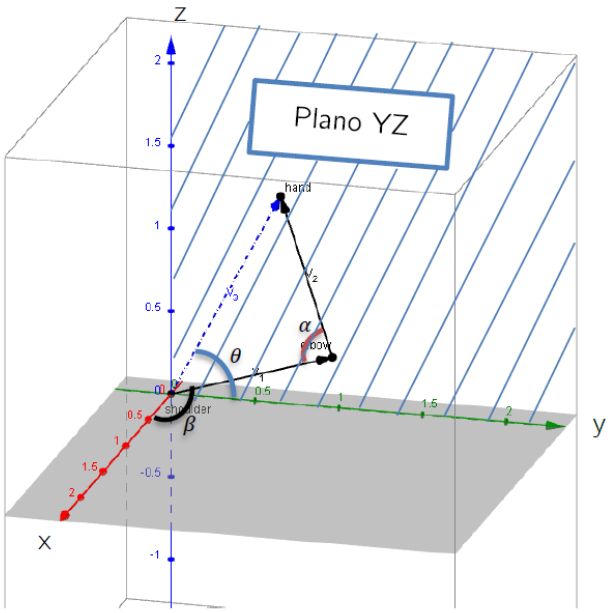


Figura V-16. Representación geométrica en 3D de los ángulos finales equivalentes

Estos cálculos son implementados de la siguiente manera:

```
# Cosenos directores
art1 = (math.acos(v3[1]/np.linalg.norm(v3)))-1.57
art2 = (math.acos(v1[0]/np.linalg.norm(v1)))

# Angulo formado entre v1 y v2
art3 = math.acos(np.dot(v1,v2)/(np.linalg.norm(v1)*np.linalg.norm(v2)))
```

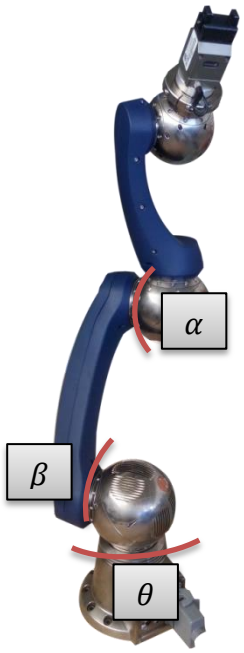


Figura V-17. Equivalencia de los ángulos en el robot LWA 4p

Tras realizar los cálculos, se envía un mensaje a través de ROS con el ángulo de cada una de las articulaciones y un número de secuencia, este mensaje será descrito en el siguiente apartado.

```
arm_angulos = [art1, art2, art3, art4, art5, art6]

 sec += 1
 pub.publish(sec, arm_angulos)
```

A su vez, se desarrolló un nodo específico para la impresión en pantalla de los mensajes `tf`, los vectores formados y el ángulo final correspondiente cada articulación del brazo robot, con el fin de corregir posibles errores a la hora de la implementación del nodo final. El nombre de este nodo es `Kinect_tf_listener_print.py`.

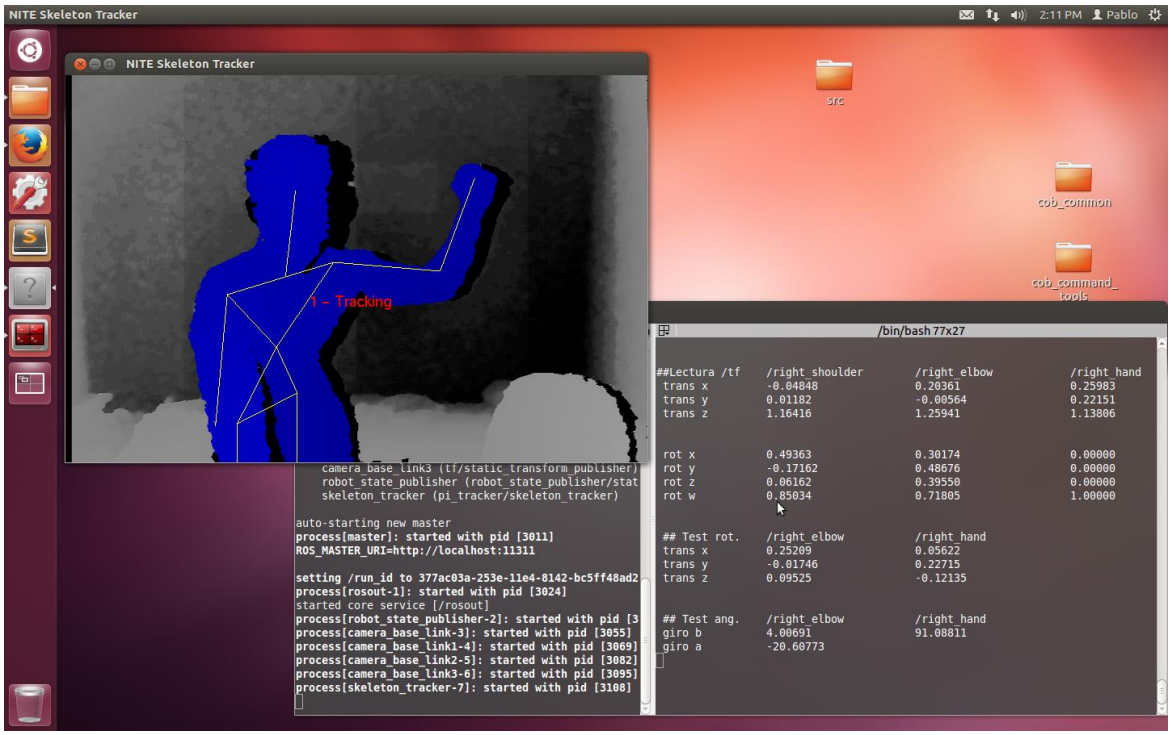


Figura V-18. Nodo `kinect_tf_listener_print` en ejecución

11.3.1.3. Kinect2_arm

Este nodo se encarga de realizar una lectura del mensaje enviado por el nodo `Kinect_tf_listener_pub`, de la detección de movimiento del usuario y de establecer una comunicación con el brazo robot LWA 4p a través de CANopen. A continuación de detalla el funcionamiento del nodo.

En este nodo se han definido las siguientes librerías:

```
import roslib
roslib.load_manifest('ki2_arm')
roslib.load_manifest('cob_script_server')
import rospy
import math
import tf
import os
import numpy as np
from ki2_arm.msg import ArmPos

from simple_script_server import *
sss = simple_script_server()
```

Se han incluido las funciones proporcionadas por `simple_script_server`, con el fin de poder realizar una comunicación con el robot a alto nivel.

```
#############################################################
##      MAIN

if __name__ == '__main__':

    while not rospy.is_shutdown():
        try:
            start()
        except rospy.ROSInterruptException:
            pass
```

Al iniciar el nodo se ejecuta la función `start()`

```
def start():

   rospy.Subscriber("arm_pos", ArmPos, callback, queue_size = 1)
   rospy.init_node('Kinect2_arm')
   rospy.spin()
```

Este nodo se suscribe al tópico `arm_pos` recibiendo el mensaje `ArmPos` publicado por el nodo `Kinect_tf_listener_pub`. Cada vez que recibe uno de estos mensajes se ejecuta la función `callback`. El parámetro `queue_size` define la cola máxima de mensajes almacenados en el buffer, se establece a 1 ya que no se desea que mientras el robot este realizando un movimiento se reciban nuevos datos.

```
def callback(msg):

    global mat_hist
    global move
    for l in reversed(range(n_articulaciones)):
        for j in reversed(range(m_datos_pasados-1)):

            mat_hist[l,j+1]=mat_hist[l,j]

        mat_hist[l,0]=msg.angulos[l]
```

La función `callback()` forma una matriz con el dato recibido y `n` datos anteriores. La matriz de ángulos recibidos A_{hist} aplicada a este caso tiene una dimensión nxm, siendo n el número de articulaciones y m el número de datos pasados:

$$A_{hist} = \begin{pmatrix} \vartheta_{11} & \cdots & \vartheta_{1m} \\ \vdots & \ddots & \vdots \\ \vartheta_{n1} & \cdots & \vartheta_{nm} \end{pmatrix}$$

Aplicado a un número fijo de articulaciones, correspondiente a los ángulos θ, β y α. Obtenemos:

$$A_{hist} = \begin{pmatrix} \theta_1 & \cdots & \theta_m \\ \beta_1 & \cdots & \beta_m \\ \alpha_1 & \cdots & \alpha_m \end{pmatrix}$$

```
#detectar mov brazo usuario
if (abs(mat_hist[0,0]-mat_hist[0,1])>ang_min or
    abs(mat_hist[1,0]-mat_hist[1,1])>ang_min or
    abs(mat_hist[2,0]-mat_hist[2,1])>ang_min) and
    mat_hist[0,1]!=0:

       #print 'brazo moviendose'
       move1=1

#mover brazo robot tras 2
if move>1 and move1==1:

    #print '\t \t\tMOVER BRAZO REAL'
    sss.move("arm", [[msg.angulos[0], msg.angulos[1],
             msg.angulos[2], msg.angulos[3], msg.angulos[4],
             msg.angulos[5]]])
    move1=0
    move=0

move +=1
```

A su vez, la función `callback()` detecta el movimiento del brazo del usuario y su parada, para ello resta el valor actual recibido con el valor anterior y lo comparar con el valor fijado ang_min, establecido por defecto en 5°, de tal modo que sí se produce una variación a este ángulo establece la variables `move` y `move1` en 0 y 1 respectivamente.

La variable `move` es usada para la detección de movimiento del usuario y `move1` para el control de movimiento del robot. De modo que `move1` pasa a valer uno

cuando se ha detectado un movimiento por parte del usuario y `move` indica el número de mensajes recopilados antes de iniciar el movimiento en el brazo robot.

Si se trabaja para seguimiento de puntos, el valor `move` debe ser de 1 al ejecutar el movimiento ya que no es necesario trabajar con los valores anteriores, mientras que para seguimiento de trayectorias si sería necesario.

A continuación se muestra un ejemplo de la trayectoria construida a partir de la posición actual y la de los 4 valores anteriores:

```
if move>4 and move1==1:

    sss.move("arm", [[mat_hist[0,4], mat_hist[1,4],
                      mat_hist[2,4] 0, 0, 0],
                      [mat_hist[0,3], mat_hist[1,3],
                      mat_hist[2,3] 0, 0, 0]]
                      [mat_hist[0,2], mat_hist[1,2],
                      mat_hist[2,2] 0, 0, 0]]
                      [mat_hist[0,1], mat_hist[1,1],
                      mat_hist[2,1] 0, 0, 0]]
                      [mat_hist[0,0], mat_hist[1,0],
                      mat_hist[2,0] 0, 0, 0]]])
```

Tras el diseño de este nodo se realizaron pruebas tanto en el robot real como en el simulado (Figura V-19), los resultados fueron recogidos en los siguientes vídeos [62, 63].

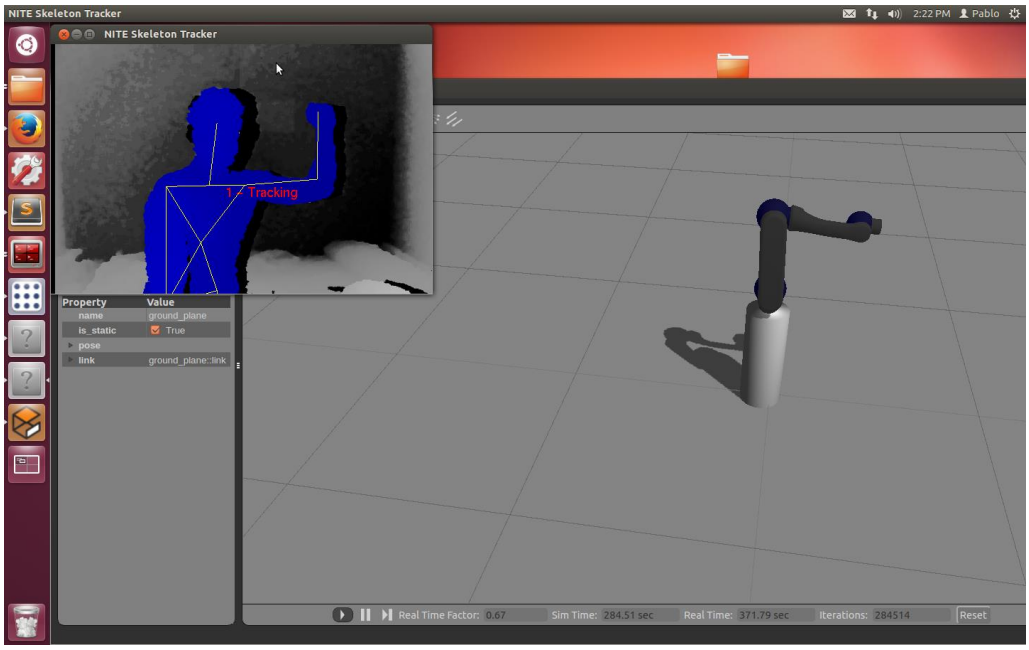


Figura V-19. Comunicación final con el brazo simulado

11.3.2. Mensajes

Se realizará una descripción de los mensajes de comunicación entre los distintos componentes del paquete `ki2_arm`, así como los mensajes de comunicación de los dispositivos finales con el entorno ROS.

- `ArmPos`. Define el mensaje de comunicación entre los dos nodos desarrollados, en él se incluyen los ángulos calculados por el nodo `kinect_tf_listener_pub` junto a un número de secuencia correspondiente al número del mensaje. A continuación se representa el archivo de mensaje (.msg) situado en la carpeta `/ki2_arm/msg`:

```
int32 secuencia
float64[] angulos
```

- Mensaje producido por Kinect. Se trata del mensaje `tf` descrito en el apartado 11.1.2.2.

- Mensajes finales para el control de movimiento. Como se ha visto en el apartado 11.3.1.3, se puede definir un movimiento hacia un punto, o una serie de puntos para formar una trayectoria, en cada caso se estructura un mensaje final distinto, como aparecen representados en la Figura V-20 y Figura V-21.

Al trabajar a alto nivel con el *stack* definido en 8.3.5 y su API `cob_script_server` no se tiene control directo sobre los mensajes de control de movimiento, de manera que son generados de manera automática tras la ejecución de uno de estos comandos.

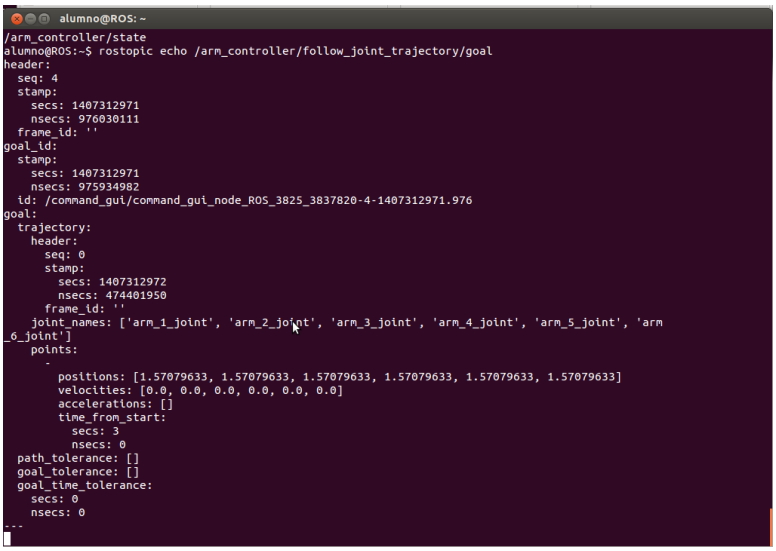


Figura V-20. Escucha del tópico /arm_controller/follow_joint_trayectory/goal para un mensaje de un único punto.

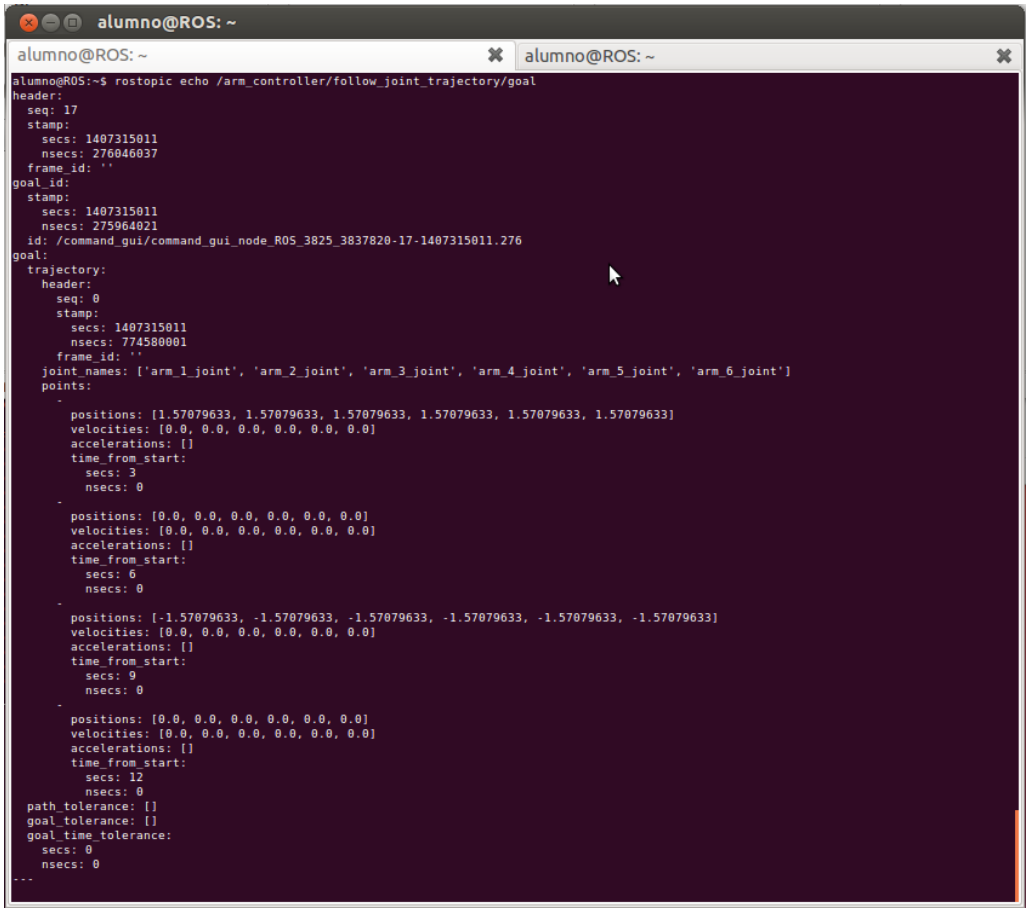


Figura V-21. Escucha del tópico /arm_controller/follow_joint_trayectory/goal para un mensaje de 4 puntos

11.3.3. Archivos *Launch*

- `Skeletal_tracking.launch`. Ejecuta el nodo de seguimiento del usuario, así como una interfaz donde se muestra una representación.

- `Ki2arm_follow.launch`. Ejecuta todos los nodos necesarios y archivos de configuración para comunicar el brazo robot real con Kinect. Incluye los nodos principales del paquete ki2_arm y el visualizador de seguimiento.

- `Ki2arm_follow_sim.launch`. Ejecuta los nodos necesarios para la creación de un entorno de simulación en Gazebo, también incluye los nodos principales del paquete ki2_arm y el visualizador de seguimiento.

11.3.3.1. Grafos de comunicación

En la Figura V-22 y Figura V-23 se detallan el grafo de comunicación entre cada uno de los componentes que hacen posible la comunicación con el dispositivo final a través de Kinect. Cada uno de los nodos, representados como círculos, son ejecutados y procesados de manera individual por parte de ROS, de modo que es necesario establecer una comunicación entre ellos, a través de tópicos, repesentados como `/nombre`, en el grafo de comunicación, indicando el sentido de comunicación, de tal modo que los nodos pueden actuar como subscriptores o publicadores.

En el caso de una comunicación con el robot real (Figura V-22), se puede apreciar como nodo final es el nodo de comunicación por CAN `/canopen/canopen_ros`. Mientras que en el robot simulado (Figura V-23), los mensajes de trayectorias son direccionados al nodo de Gazebo, `/gazebo`. En ambos casos el mensaje enviado es el mismo, de modo que se pueden realizar ensayos en el robot simulado sin necesidad de trabajar con nodos distintos.

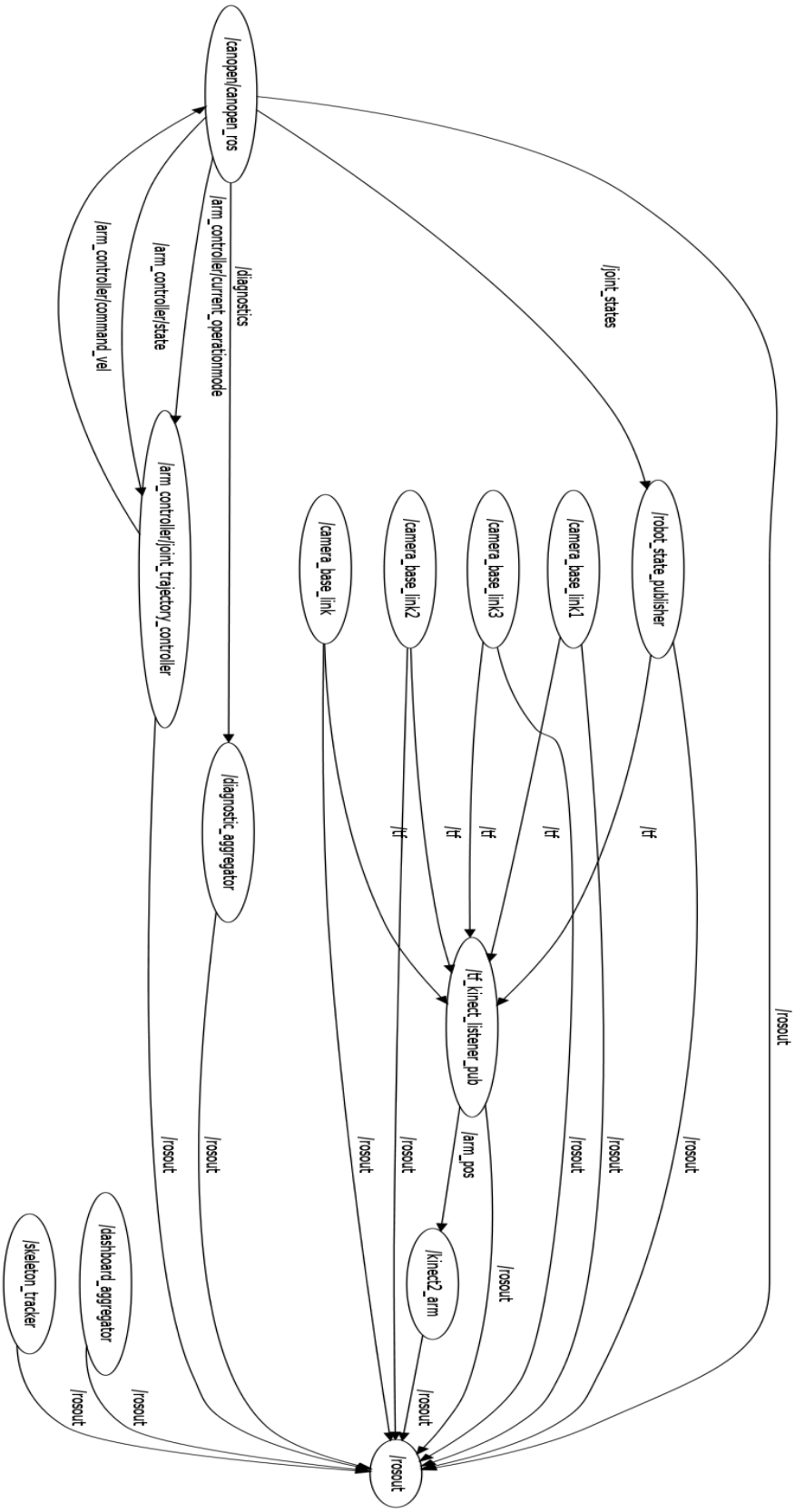


Figura V-22. Grafo de comunicación del paquete ki2_arm en el robot real

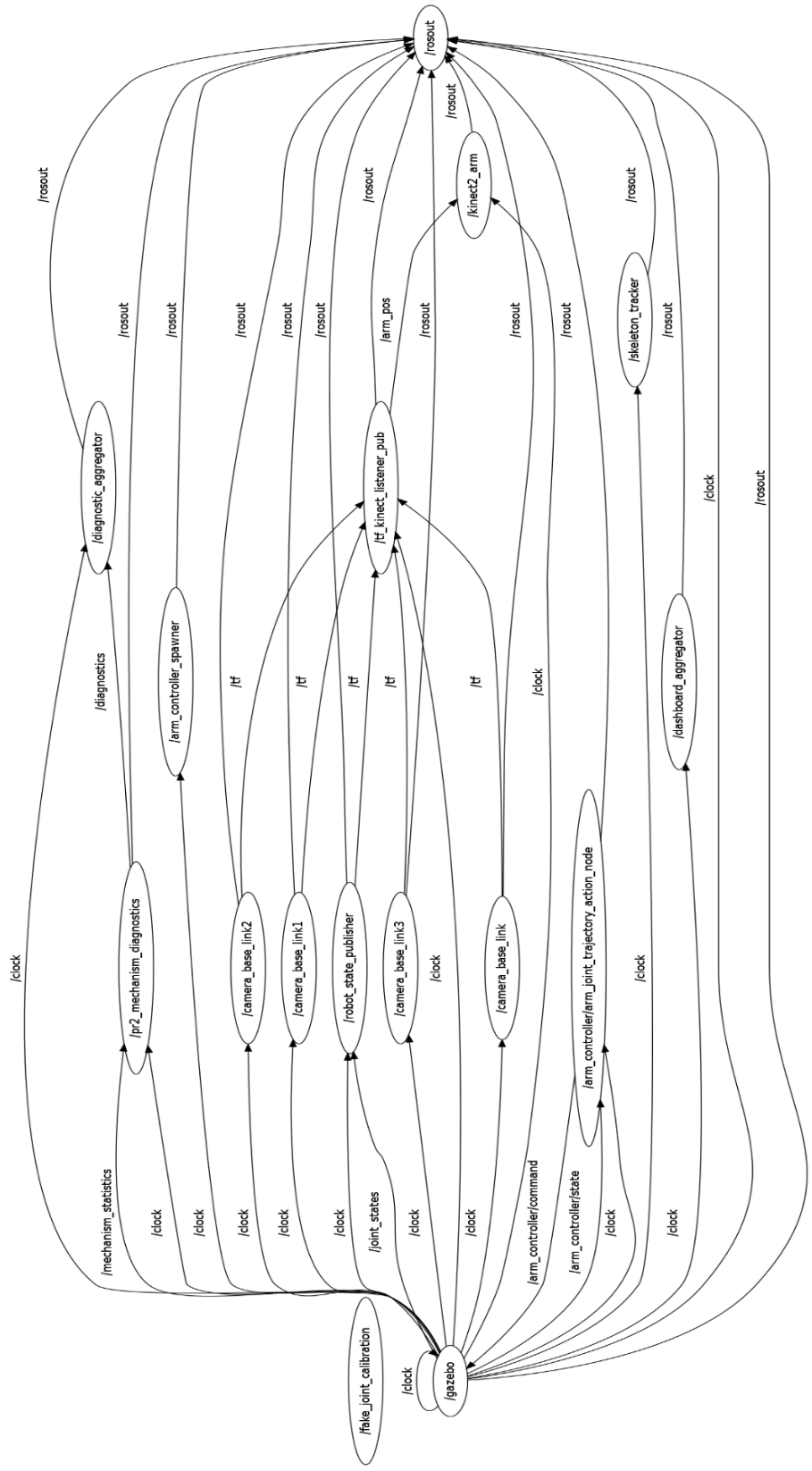


Figura V-23. Grafo de comunicación del paquete ki2_arm en el brazo robot simulado

11.4. Interfaz gráfico de usuario

La interfaz gráfica *LWA-4p controller* permite la ejecución de todo el sistema de comunicación a través de una interfaz sencilla, con el fin de que pueda ser usada por cualquier usuario que no tenga conocimientos de ROS y ser usada en el ámbito educativo. Para la ejecución del programa basta con ejecutar el archivo compilado a través de un terminal.

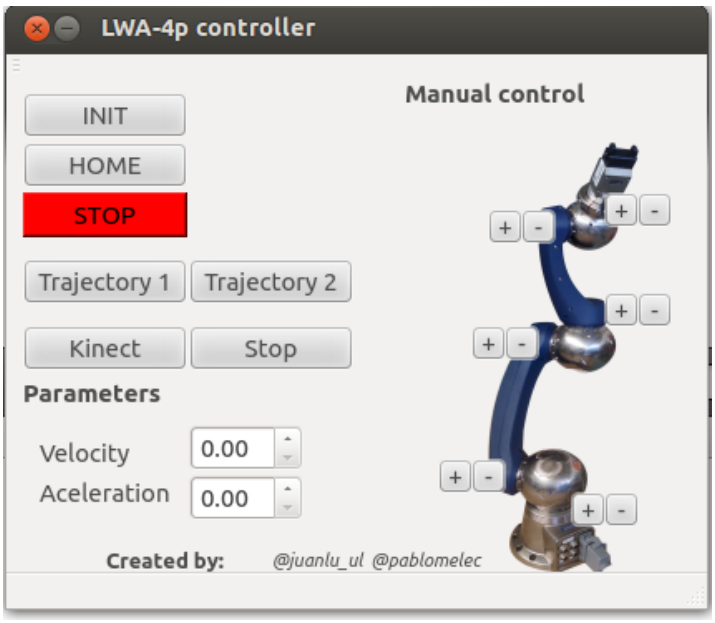


Figura V-24. Interfaz de usuario *LWA-4p controller*

Está compuesto por un conjunto de botones e imágenes que representan acciones disponibles de comunicación con el robot real. A continuación se describirá brevemente cada una de sus funciones:

- Botón *INIT*. Inicia cada uno de los módulos del brazo robot, este proceso tarda unos segundos.

- Botón *HOME*. Lleva al robot a la posición *home*, definida como (0, 0, 0, 0, 0, 0) en el archivo de configuración correspondiente.

- Botón *STOP*. Parada del brazo robot, en el caso de que el robot este ejecutando un movimiento éste es interrumpido.

- Botón *Trajectory* 1 y 2. Envía una trayectoria a seguir de manera predefinida, estas trayectorias serán configuradas dentro del archivo /script/trajectory1.py y trajectory2.py.

- Kinect. Ejecuta el modo seguimiento, carga una ventana de usuario donde se muestra el seguimiento realizado por ROS del usuario y carga todos los nodos necesarios para establecer una comunicación del Kinect con el brazo robot a partir del paquete "ki2_arm" desarrollado anteriormente. El botón de Stop asociado para todos los nodos ejecutados deteniendo el seguimiento y parando el robot.

Por otra parte, posee un apartado para el control manual del brazo robot, consta de los siguientes elementos:

- Sección *Parameters*. Definen la aceleración y velocidad para la ejecución del movimiento manual.

- Sección *Manual Control*. A partir de los parámetros definidos en la sección *Parameters* se ejecuta un el movimiento según la articulación y el sentido de giro seleccionado.

Esta interfaz puede ser editada con el fin de añadir nuevas funcionalidades, por ello se facilitan los archivos de necesarios para su edición en los Anexos de la memoria.

Capítulo VI.

VI. Conclusiones

12. Conclusiones

El sensor Kinect, es un sensor muy versátil capaz de realizar reconocimiento de movimientos, reconocimiento facial o realizar escaneos de entornos en 3D. Sus prestaciones y su bajo coste lo hacen de un dispositivo cada vez más usado en investigación y desarrollo de aplicaciones, por todo ello ha sido seleccionado a la hora de realizar este proyecto.

Durante la instalación del sistema e integración de los elementos en ROS hubo algunos problemas por la incompatibilidad del firmware del brazo robot con la versión del paquete `ipa_canopen` de ROS. También se encontraron problemas para la localización de las librerías OpenNI para Linux usadas por Kinect, ya que tras la adquisición de PrimeSense por Apple la página cerró [35] y se tuvo que recurrir a una página web tercera. Todo esto mostró la dificultad de integrar todos los dispositivos bajo un mismo sistema de manera concurrente.

Con el paquete ROS `ki2_arm` desarrollado se logró establecer una comunicación entre los datos recogidos de un usuario a través de Kinect y el brazo robot LWA 4p. Fue necesario establecer en uno de los nodos una detección de movimiento del brazo del usuario, ya que no se podían enviar los mensajes de posición o trayectoria en tiempo real, sino que estos mensajes tenían que ser enviados cada cierto tiempo, de tal modo que cuando el nodo detectara una parada en el movimiento del usuario ejecutara una acción determinada.

Finalmente fue realizada una interfaz gráfica que integra Kinect junto al sistema robotizado para lograr una comunicación sencilla y entendible. Esta aplicación es de especial interés para su uso en enseñanza.

13. Trabajos futuros

Respecto al control del brazo robot se podría incorporar un control a bajo nivel a través del protocolo CANopen, teniendo así más libertad y control a la hora de ejecutar órdenes sobre el robot. También se podría implementar un control del brazo robot mediante el software de manipulación móvil *MoveIt!* [33] que ofrece una serie de *plugins* dirigidos para la implementación de algoritmos de cinemática inversa o planificación de trayectorias.

En el campo de la visión artificial se podría implementar una planificación de trayectorias con percepción 3D, también incluido en *MoveIt!*, pudiendo así calcular trayectorias alternativas en función de obstáculos del entorno para evitar colisiones, o la manipulación de objetos del entorno detectados por el sensor Kinect mediante la pinza del brazo robot. La Figura VI-1 muestra estructura del nodo `move_group` de *MoveIt!* encargado de realizar todo el procesado.

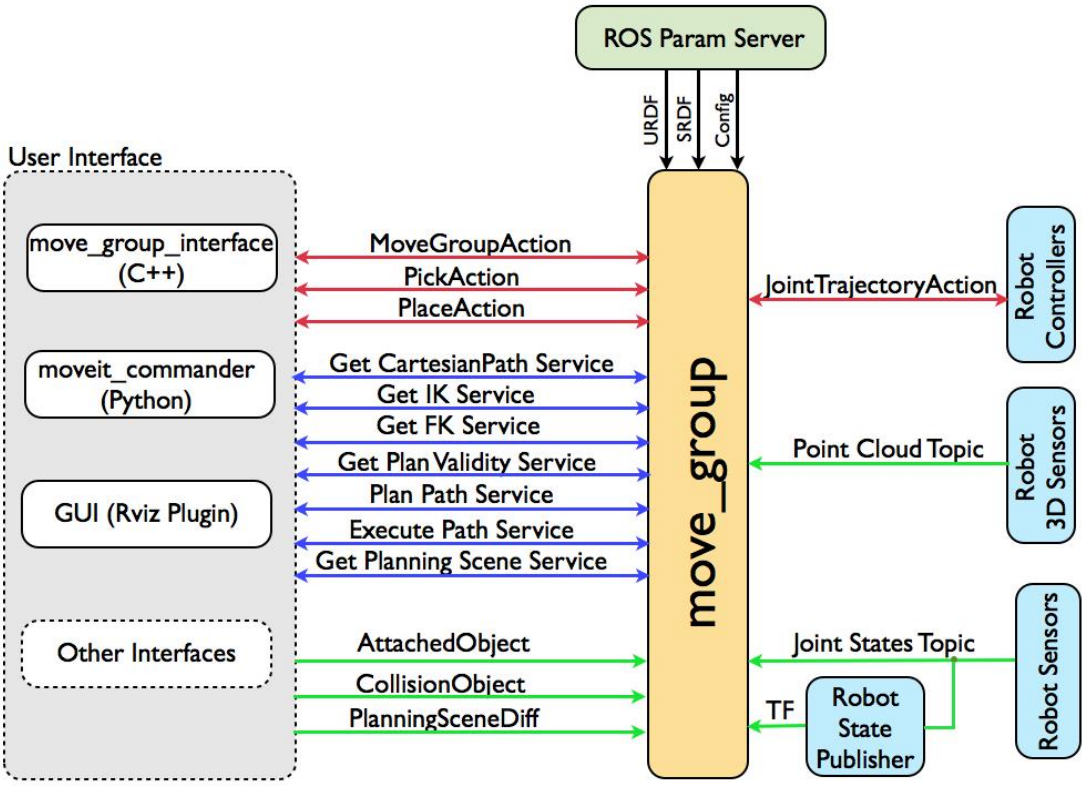


Figura VI-1. Estructura del nodo move_group de MoveIt!

Capítulo VII.

VII. Bibliografía

[1] A. Weber, H. Breitwieser, and J. Benner. A Flexible Architecture for Telemanipulator Control, Control Engineering Practive 10, pp. 1293-1299, 2002.

[2] Aníbal Ollero Baturone. Robótica: manipuladores y robots móviles, 2011.

[3] Application programming interface - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Application_programming_interface. Consultado el 7 de agosto de 2014.

[4] arm_navigation - ROS Wiki http://wiki.ros.org/arm_navigation. Consultado el 5 de agosto de 2014.

[5] Board on Human-Systems Integration, Committee on Human Factors, National Research Council.

[6] Bullet Collision Detection & Physics Library: btTransform Class Reference http://www.bulletphysics.com/Bullet/BulletFull/classbtTransform.html. Consultado el 12 de agosto de 2014.

[7] CANopen - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/CANopen.Consultado el 22 de junio de 2014.

[8] Carril Din - Wikipedia, la enciclopedia libre http://es.wikipedia.org/wiki/Carril_DIN. Consultado el 24 de junio de 2014.

[9] catkin - ROS Wiki http://wiki.ros.org/catkin. Consultado el 2 de agosto de 2014.

[10] Clemente Giorio, Massimo Fascinari. Kinect in Motion – Audio and Visual Tracking by Example, abril de 2013.

[11] cob_command_tools - ROS Wiki http://wiki.ros.org/cob_command_tools. Consultado el 5 de agosto de 2014.

[12] Corley, Anne-Marie. The Reality of Robot Surrogates. spectrum.ieee.com, 19 de marzo de 2013.

[13] Curiosity - Wikipedia, la enciclopedia libre http://es.wikipedia.org/wiki/Curiosity. Consultado el 12 de agosto de 2014.

[14] da Vinci Surgery - Minimally Invasive Robotic Surgery with the da Vinci Surgical System http://www.davincisurgery.com/da-vinci-surgery/da-vinci-surgical-system/. Consultado el 27 de junio 2014

[15] Documentation - ROS Wiki http://wiki.ros.org/. Consultado el 30 de junio de 2014.

[16] Fogle R. F., The use of Teleoperators In Hostile Environment Applications, Proceedings of the 1992, IEEE International Conference on Robotics and Automation, mayo de 1992.

[17] Fuente de alimentación de montaje en carril DIN, 480W, 1 salida, salida 24V dc 20A http://es.rs-online.com/web/p/fuentes-de-alimentacion-de-montaje-en-panel-y-carril-din/7894026/. Consultado el 24 de junio de 2014.

[18] FWS 115 Flat Change System: SCHUNK Mobile Greifsysteme http://mobile.schunk-microsite.com/en/produkte/produkte/fws-115-flat-change-system.html. Consultado el 10 de julio de 2014.

[19] G. Niemeyer, and J. E. Slotine. Using Wave Variables For System Analysis and Robot Control, IEEE International Conference Robotic & Automation, abril de 1997.

[20] Gazebo http://gazebosim.org/. Consultado el 26 de julio de 2014.

[21] General Atomics MQ-1 Predator - Wikipedia, the free enciclopedia http://en.wikipedia.org/wiki/General_Atomics_MQ-1_Predator. Consultado el 29 de junio de 2014.

[22] geometry_msg - ROS Wiki http://wiki.ros.org/geometry_msgs. Consultado el 10 de agosto de 2014.

[23] Hercules (ROV) http://oceanexplorer.noaa.gov/technology/subs/hercules/hercules.html. Consultado el 23 de junio de 2014.

[24] Index of /downloads/3rdparty/bin-linux http://www.mira-project.org/downloads/3rdparty/bin-linux/. Consultado el 27 de julio de 2014.

[25] ipa_canopen_ros - ROS Wiki http://wiki.ros.org/ipa_canopen_ros. Consultado el 20 de julio de 2014.

[26] iRobot 510 PackBot, the Multi-mission Robot http://www.irobot.com/us/learn/defense/packbot.aspx. Consultado el 23 de junio de 2014.

[27] John J. Craig, Introduction to Robotics: Mechanics and Control, Pearson Prentice Hall.

[28] Kinect - Wikipedia, the free enciclopedia http://en.wikipedia.org/wiki/Kinect. Consultado el 30 de junio de 2014.

[29] Kinect for Windows http://www.microsoft.com/en-us/kinectforwindows/. Consultado el 30 de junio de 2014.

[30] Kinect for Windows Sensor Components and Specifications http://msdn.microsoft.com/en-us/library/jj131033.aspx. Consultado el 30 de junio de 2014.

[31] LWA 4P - Robotnik http://www.robotnik.es/brazos-roboticos/lwa-4p/. Consultado el 4 de julio de 2014.

[32] Method of and apparatus for controlling mechanism of moving vessels or vehicles US 613809 A.

[33] MoveIt! http://moveit.ros.org/. Consultado el 25 de agosto de 2014.

[34] Nuño Ortega, Enmanuel y Basañez Villaluenga, Luis. Teleoperación: técnicas, aplicaciones, entorno sensorial y teleoperación inteligente IOC-DT-P-2004-05.

[35] OpenNI to close http://www.i-programmer.info/news/194-kinect/7004-openni-to-close-.html. Consultado el 30 de agosto de 2014.

[36] openni_tracker - ROS Wiki http://wiki.ros.org/openni_tracker. Consultado el 5 de agosto de 2014.

[37] P Milgram and J. Ballantyne. Real Word Teleperation via Virtual Environment Modelling, International Conference on Artificial Reality & Tele-existence ICAT' 97, Toky, 3-5 de diciembre de 1997.

[38] P. Batsomboon, S. Tosunoglu, and D. W. Repperger. Development of a Mechatronic System: a telesensation system for training and teleoperation, Chapter, Recent Advances in Mechatronics, Springer-Verlag, New York, pp. 304-321, 1999.

[39] PCAN-USB: PEAK-System http://www.peak-system.com/PCAN-USB.199.0.html. Consultado el 5 de julio de 2014.

[40] EAK-System LINUX Website http://www.peak-system.com/fileadmin/media/linux/index.htm. Consultado el 5 de julio de 2014.

[41] Polygraph (duplicating device) - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Polygraph_(duplicating_device) . Consultado el 28 de junio de 2014.

[42] Productos: sistemas de agarre móvil SCHUNK http://mobile.schunk-microsite.com/. Consultado el 12 de agosto de 2014.

[43] Quaternion - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Quaternion. Consultado el 7 de agosto de 2014.

[44] Questions - ROS Answers: Open Source Q&A Forum http://answers.ros.org/questions/. Consultado el 30 de junio de 2014.

[45] Robot Operating System - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Robot_Operating_System. Consultado el 29 de junio de 2014.

[46] Robotnik: Computer Sales, Service & Support http://www.robotnik.com. Consultado el 5 de julio de 2014.

[47] Robots/PR2 - ROS Wiki http://wiki.ros.org/Robots/PR2. Consultado el 2 de agosto de 2014.

[48] ROS.org | Is ROS For Me? http://www.ros.org/is-ros-for-me/. Consultado el 27 de junio de 2014.

[49] ROS.org | Powering the world's robots http://www.ros.org. Consultado el 27 de junio de 2014.

[50] ROS: Five Years - ROS robotics news http://www.ros.org/news/2012/12/ros-five-years.html. Consultado el 27 de junio de 2014.

[51] S. Hong, B. S. Km, and S. Kim. Artificial Force Reflection Control for Teleoperated Mobile Robots, Mechatronics, pp. 707-717, 1998.

[52] SCHUNK GmbH & Co. KG Spann- und Greiftechnik http://www.schunk.com/schunk/schunk_websites/service/downloads.html?submenu=215&submenu2=0&country=INT&lngCode=EN&lngCode2=EN. Consultado el 10 de julio de 2014.

[53] schunk_robots - ROS Wiki http://wiki.ros.org/schunk_robots. Consultado el 5 de agosto de 2014.

[54] Sensor - Wikipedia, la enciclopedia libre http://es.wikipedia.org/wiki/Sensor. Consultado el 22 de junio de 2014.

[55] Singapore Armed Forces Develops Unmanned Ground Vehicles for Military Operations http://www.azorobotics.com/news.aspx?newsID=1528. Consultado el 25 de junio de 2014.

[56] Sistema Operativo Robótico - Wikipedia, la enciclopedia libre http://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico. Consultado el 30 de junio de 2014.

[57] Skeletal Tracking http://msdn.microsoft.com/en-us/library/hh973074.aspx. Consultado el 4 de julio de 2014.

[58] tf - ROS Wiki http://wiki.ros.org/tf. Consultado el 5 de agosto de 2014.

[59] tf/Overview/Transformations - ROS Wiki http://wiki.ros.org/tf/Overview/Transformations. Consultado el 5 de agosto de 2014.

[60] The International Federation of Robotics, and the United Nations, "World Robotics 2001," Statistics, Market Analysis, Forecasts, Case Studies and Profitability of Robot Investment, United Nations Publication, Nueva York y Ginebra, 2001.

[61] Vídeo comunicación a partir del paquete ipa_canopen https://www.youtube.com/watch?v=4pij3307yf4

[62] Vídeo ki2_arm usado en el robot real https://www.youtube.com/watch?v=AFGLrJHC4wI

[63] Vídeo ki2_arm usado en el robot simulado https://www.youtube.com/watch?v=B-HF4i6LwIY

[64] What is natural user interface (NUI)? - Definition from WhatIs.com http://whatis.techtarget.com/definition/natural-user-interface-NUI. Consultado el 13 de julio de 2014.

Capítulo VIII.

VIII. Anexos

14. Archivos ROS

14.1. Código fuente

14.1.1. skeleton_tracker.cpp

```
1.  // http://www.ros.org
2.
3.  #include <ros/ros.h>
4.  #include <std_msgs/String.h>
5.  //#include <roslib/Header.h>
6.  #include <std_msgs/Header.h>
7.  #include <pi_tracker/Skeleton.h>
8.  #include <tf/transform_broadcaster.h>
9.  #include <kdl/frames.hpp>
10. #include <GL/glut.h>
11. #include <string>
12. #include "KinectController.h"
13. #include "KinectDisplay.h"
14.
15. using std::string;
16.
17. #ifndef PI
18. #define PI 3.14159265359
19. #endif
20. #ifndef HALFPI
21. #define HALFPI 1.57079632679
22. #endif
23. #ifndef QUARTPI
24. #define QUARTPI 0.785398163397
25. #endif
26.
27. namespace skeleton_tracker
28. {
29.     class SkeletonTracker
30.   {
31.     public:
32.       std::string fixed_frame;
33.
34.       SkeletonTracker()
35.       {
36.       }
```

```
37.
38.       void init()
39.       {
40.         ros::NodeHandle n;
41.         int rate;
42.         n.param("tracking_rate", rate, 1);
43.         n.param("fixed_frame", fixed_frame, std::string("openni_depth_frame"))
    ;
44.         skeleton_pub_ = n.advertise<pi_tracker::Skeleton>("/skeleton", rate);

45.       }
46.
47.       void publishTransform(KinectController &kinect_controller, XnUserID cons
    t &user, XnSkeletonJoint const &joint, string const &frame_id, string const &c
    hild_frame_id, pi_tracker::Skeleton &skeleton)
48.       {
49.
50.         xn::UserGenerator& UserGenerator = kinect_controller.getUserGenerator(
    );
51.         static tf::TransformBroadcaster br;
52.
53.         XnSkeletonJointPosition joint_position;
54.         UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(user, joint, j
    oint_position);
55.         double x = joint_position.position.X / 1000.0;
56.         double y = joint_position.position.Y / 1000.0;
57.         double z = joint_position.position.Z / 1000.0;
58.
59.         XnSkeletonJointOrientation joint_orientation;
60.         UserGenerator.GetSkeletonCap().GetSkeletonJointOrientation(user, joint
    , joint_orientation);
61.
62.         XnFloat* m = joint_orientation.orientation.elements;
63.         KDL::Rotation rotation(m[0], m[1], m[2],
64.                    m[3], m[4], m[5],
65.                    m[6], m[7], m[8]);
66.         double qx, qy, qz, qw;
67.         rotation.GetQuaternion(qx, qy, qz, qw);
68.
69.         geometry_msgs::Vector3 position;
70.         geometry_msgs::Quaternion orientation;
71.
72.         position.x = x;
73.         position.y = y;
74.         position.z = z;
75.
76.         orientation.x = qx;
77.         orientation.y = qy;
78.         orientation.z = qz;
79.         orientation.w = qw;
80.
81.         skeleton.name.push_back(child_frame_id);
82.         skeleton.position.push_back(position);
83.         skeleton.orientation.push_back(orientation);
84.         skeleton.confidence.push_back(joint_position.fConfidence);
85.
86.         tf::Transform transform;
87.         transform.setOrigin(tf::Vector3(x, y, z));
88.         transform.setRotation(tf::Quaternion(qx, qy, qz, qw));
89.         br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), fra
    me_id, child_frame_id));
90.       }
```

```
91.
92.     void processKinect(KinectController &kinect_controller)
93.       {
94.         XnUserID users[16];
95.         XnUInt16 users_count = 15;
96.         xn::UserGenerator& UserGenerator = kinect_controller.getUserGenerator(
    );
97.         UserGenerator.GetUsers(users, users_count);
98.         pi_tracker::Skeleton g_skel;
99.
100.               for (int i = 0; i < users_count; ++i)
101.               {
102.                 XnUserID user = users[i];
103.                 if (!UserGenerator.GetSkeletonCap().IsTracking(user))
104.                   continue;
105.
106.                 publishTransform(kinect_controller, user, XN_SKEL_HEAD,
         fixed_frame, "head", g_skel);
107.                 publishTransform(kinect_controller, user, XN_SKEL_NECK,
         fixed_frame, "neck", g_skel);
108.                 publishTransform(kinect_controller, user, XN_SKEL_TORSO,
         fixed_frame, "torso", g_skel);
109.
110.                 publishTransform(kinect_controller, user, XN_SKEL_LEFT_SHOULD
    ER,  fixed_frame, "left_shoulder", g_skel);
111.                 publishTransform(kinect_controller, user, XN_SKEL_LEFT_ELBOW,
         fixed_frame, "left_elbow", g_skel);
112.                 publishTransform(kinect_controller, user, XN_SKEL_LEFT_HAND,
         fixed_frame, "left_hand", g_skel);
113.
114.                 publishTransform(kinect_controller, user, XN_SKEL_RIGHT_SHOUL
    DER, fixed_frame, "right_shoulder", g_skel);
115.                 publishTransform(kinect_controller, user, XN_SKEL_RIGHT_ELBOW
    ,    fixed_frame, "right_elbow", g_skel);
116.                 publishTransform(kinect_controller, user, XN_SKEL_RIGHT_HAND,
         fixed_frame, "right_hand", g_skel);
117.
118.                 publishTransform(kinect_controller, user, XN_SKEL_LEFT_HIP,
         fixed_frame, "left_hip", g_skel);
119.                 publishTransform(kinect_controller, user, XN_SKEL_LEFT_KNEE,
         fixed_frame, "left_knee", g_skel);
120.                 publishTransform(kinect_controller, user, XN_SKEL_LEFT_FOOT,
         fixed_frame, "left_foot", g_skel);
121.
122.                 publishTransform(kinect_controller, user, XN_SKEL_RIGHT_HIP,
         fixed_frame, "right_hip", g_skel);
123.                 publishTransform(kinect_controller, user, XN_SKEL_RIGHT_KNEE,
         fixed_frame, "right_knee", g_skel);
124.                 publishTransform(kinect_controller, user, XN_SKEL_RIGHT_FOOT,
         fixed_frame, "right_foot", g_skel);
125.
126.                 g_skel.user_id = user;
127.                 g_skel.header.stamp = ros::Time::now();
128.                 g_skel.header.frame_id = fixed_frame;
129.                 skeleton_pub_.publish(g_skel);
130.                 break;    // only read first user
131.               }
132.             }
133.
134.         private:
135.           ros::Publisher  skeleton_pub_;
136.         };
```

```
137.
138.       }
139.
140.
141.       #define GL_WIN_SIZE_X 720
142.       #define GL_WIN_SIZE_Y 480
143.       KinectController g_kinect_controller;
144.       skeleton_tracker::SkeletonTracker g_skeleton_tracker;
145.
146.       void glutIdle (void)
147.       {
148.           glutPostRedisplay();
149.       }
150.
151.       void glutDisplay (void)
152.       {
153.           xn::SceneMetaData sceneMD;
154.           xn::DepthMetaData depthMD;
155.
156.         g_kinect_controller.getContext().WaitAndUpdateAll();
157.         g_skeleton_tracker.processKinect(g_kinect_controller);
158.         g_kinect_controller.getDepthGenerator().GetMetaData(depthMD);
159.         g_kinect_controller.getUserGenerator().GetUserPixels(0, sceneMD);
160.
161.           glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
162.
163.           // Setup the OpenGL viewpoint
164.           glMatrixMode(GL_PROJECTION);
165.           glPushMatrix();
166.           glLoadIdentity();
167.
168.           glOrtho(0, depthMD.XRes(), depthMD.YRes(), 0, -1.0, 1.0);
169.
170.         glDisable(GL_TEXTURE_2D);
171.
172.         kinect_display_drawDepthMapGL(depthMD, sceneMD);
173.           kinect_display_drawSkeletonGL(g_kinect_controller.getUserGenerator(
    ),
174.                                         g_kinect_controller.getDepthGenerator()
    );
175.
176.           glutSwapBuffers();
177.       }
178.
179.       void glutKeyboard (unsigned char key, int x, int y)
180.       {
181.           switch (key)
182.           {
183.           case 27:
184.           exit(1);
185.           break;
186.           }
187.       }
188.
189.       int main(int argc, char** argv)
190.       {
191.         ros::init(argc, argv, "skeleton_tracker");
192.         ros::NodeHandle np("~");
193.
194.         string filepath;
195.         bool is_a_recording;
196.         np.getParam("load_filepath", filepath);
```

```
197.         np.param<bool>("load_recording", is_a_recording, false);
198.
199.         g_skeleton_tracker.init();
200.         g_kinect_controller.init(filepath.c_str(), is_a_recording);
201.
202.         glutInit(&argc, argv);
203.         glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
204.         glutInitWindowSize(GL_WIN_SIZE_X, GL_WIN_SIZE_Y);
205.         glutCreateWindow ("NITE Skeleton Tracker");
206.
207.         glutKeyboardFunc(glutKeyboard);
208.         glutDisplayFunc(glutDisplay);
209.         glutIdleFunc(glutIdle);
210.
211.         glDisable(GL_DEPTH_TEST);
212.         glEnable(GL_TEXTURE_2D);
213.
214.         glEnableClientState(GL_VERTEX_ARRAY);
215.         glDisableClientState(GL_COLOR_ARRAY);
216.
217.         glutMainLoop();
218.
219.         g_kinect_controller.shutdown();
220.
221.         return 0;
222.       }
```

14.1.2. kinect_tf_listener_print.py

```
1.  #!/usr/bin/env python
2.  ##############################################################################
3.  #
4.  #   Copyright (c) 2014
5.  #   Pablo Melero Cazorla for Universidad de Almeria
6.  #
7.  ##############################################################################
8.
9.  import roslib
10. roslib.load_manifest('ki2_arm')
11. import rospy
12. import math
13. import tf
14. import os
15. ##
16. ## Usamos la funcion np.sqrt() debido a su rapida respuesta de calculo
17. ##
18. ## import timeit
19. ## timeit.timeit('np.linalg.norm(x)', setup='import numpy as np; x = np.arange
    (100)', number=1000)
20. ##      0.0450878
21. ## timeit.timeit('np.sqrt(x.dot(x))', setup='import numpy as np; x = np.arange
    (100)', number=1000)
22. ##      0.0181372
23. ##
24. import numpy as np
25.
26.
27. # Funcion 'clear()' para limpiar la pantalla del terminal
```

```
28. def clear():
29.     os.system('clear')
30. 
31. if __name__ == '__main__':
32. 
33.     rospy.init_node('tf_kinect_listener_print')
34.     listener = tf.TransformListener()
35.     fo = open("/home/alumno/groovy_workspace/catkin_ws/src/ki2_arm/output/outp
    ut.txt", "wb")
36. 
37.     rate = rospy.Rate(10.0)
38.     while not rospy.is_shutdown():
39.         try:
40.             (trans0,rot0) = listener.lookupTransform('/openni_depth_frame', '/
    right_shoulder', rospy.Time())
41.             (trans1,rot1) = listener.lookupTransform('/openni_depth_frame', '/
    right_elbow', rospy.Time())
42.             (trans2,rot2) = listener.lookupTransform('/openni_depth_frame', '/
    right_hand', rospy.Time())
43.         except (tf.LookupException, tf.ConnectivityException):
44.             continue
45. 
46. 
47.         ######################################################################
48.         ## Test de coordenadas '/openni_depth_frame', '/right_arm'
49.         ######################################################################
50. 
51.         trans = ['trans x', 'trans y', 'trans z']
52.         rot = ['rot x', 'rot y', 'rot z', 'rot w']
53. 
54.         print '\n\n##Lectura /tf \t /right_shoulder \t/right_elbow \t\t/right_
    hand'
55. 
56.         for i in range(len(trans0)):
57.            print ' %s \t %0.5f \t\t' '%0.5f \t\t' '%0.5f \t\t' % (trans[i], tr
    ans0[i], trans1[i], trans2[i])
58. 
59.         print '\n'
60. 
61.         for j in range(len(rot0)):
62.             print ' %s \t\t %0.5f \t\t' '%0.5f \t\t' '%0.5f \t\t' % (rot[j], r
    ot0[j], rot1[j], rot2[j])
63. 
64.         print '\n'
65. 
66. 
67.         ######################################################################
68.         ## Test de coordenadas '/right_shoulder', '/right_arm'
69.         ######################################################################
70. 
71.         # Vector0  /right_shoulder <-->  Origen
72.         # Vector1  /right_shoulder   ,   /right_elbow
73.         # Vector2  /right_shoulder   ,   /right_hand
74. 
75.         v1 = np.zeros(3)
76.         v2 = np.zeros(3)
77.         v3 = np.zeros(3)
78. 
79.         for k in range(len(v1)):
80.             v1[k] = trans1[k] - trans0[k]
81. 
82.         for m in range(len(v2)):
```

```
83.             v2[m] = trans2[m] - trans1[m]
84.
85.         for s in range(len(v3)):
86.             v3[s] = trans2[s] - trans0[s]
87.
88.         print ' ## Test ang. \t /right_elbow \t\t/right_hand'
89.         for l in range(len(trans)):
90.            print ' %s \t %0.5f \t\t' '%0.5f \t\t' % (trans[l], v1[l], v2[l])
91.
92.         print '\n'
93.
94.
95.         ######################################################################
96.         ## Test de angulos '/right_shoulder', '/right_arm'
97.         ######################################################################
98.         # Coordenadas cilindricas (proyeccion en plano XY y)
99.
100.                art1 = (math.acos(v3[1]/np.linalg.norm(v3)))-1.57
101.                art2 = (math.acos(v1[0]/np.linalg.norm(v1)))
102.
103.                # Angulo formado entre v1 y v2
104.                art3 = math.acos(np.dot(v1,v2)/(np.linalg.norm(v1)*np.linalg.no
    rm(v2)))
105.                print ' ## Test ang. \t /right_elbow \t\t/right_hand'
106.                print ' giro b \t %0.5f \t\t' '%0.5f \t\t' % (art1*(180/math.pi
    ), art3*(180/math.pi))
107.                print ' giro a \t %0.5f \t\t'              % (art2*(180/math.pi
    ))
108.
109.                rate.sleep()
110.                clear()
```

14.1.3. kinect_tf_listener_pub.py

```
1.  #!/usr/bin/env python
2.  ##############################################################################
3.  #
4.  #   Copyright (c) 2014
5.  #   Pablo Melero Cazorla for Universidad de Almeria
6.  #
7.  ##############################################################################
8.
9.  import roslib
10. roslib.load_manifest('ki2_arm')
11. import rospy
12. import math
13. import tf
14. import os
15. ##
16. ## Usamos la funcion np.sqrt() debido a su rapida respuesta de calculo
17. ##
18. ## import timeit
19. ## timeit.timeit('np.linalg.norm(x)', setup='import numpy as np; x = np.arange
    (100)', number=1000)
20. ##      0.0450878
21. ## timeit.timeit('np.sqrt(x.dot(x))', setup='import numpy as np; x = np.arange
    (100)', number=1000)
22. ##      0.0181372
```

```
23. ##
24. import numpy as np
25. from ki2_arm.msg import ArmPos
26. 
27. # Iniciamos el numero de secuencia para los mensajes a 0
28. sec=0
29. 
30. # Funcion 'clear()' para limpiar la pantalla del terminal
31. def clear():
32.     os.system('clear')
33. 
34. if __name__ == '__main__':
35. 
36.     pub = rospy.Publisher('arm_pos', ArmPos)
37. 
38.     rospy.init_node('tf_kinect_listener_pub')
39. 
40.     listener = tf.TransformListener()
41. 
42.     # Frecuencia de muestreo 20Hz
43.     rate = rospy.Rate(20.0)
44.     while not rospy.is_shutdown():
45.         try:
46.             (trans0,rot0) = listener.lookupTransform('/openni_depth_frame', '/
    right_shoulder', rospy.Time())
47.             (trans1,rot1) = listener.lookupTransform('/openni_depth_frame', '/
    right_elbow', rospy.Time())
48.             (trans2,rot2) = listener.lookupTransform('/openni_depth_frame', '/
    right_hand', rospy.Time())
49.         except (tf.LookupException, tf.ConnectivityException):
50.             continue
51. 
52.         ######################################################################
53.         ## Test de coordenadas '/right_shoulder', '/right_arm'
54.         ######################################################################
55. 
56.         # Vector0  /right_shoulder <-->  Origen
57.         # Vector1  /right_shoulder   ,   /right_elbow
58.         # Vector2  /right_shoulder   ,   /right_hand
59. 
60.         v1 = np.zeros(3)
61.         v2 = np.zeros(3)
62.         v3 = np.zeros(3)
63. 
64.         for k in range(len(v1)):
65.             v1[k] = trans1[k] - trans0[k]
66. 
67.         for m in range(len(v2)):
68.             v2[m] = trans2[m] - trans1[m]
69. 
70.         for s in range(len(v3)):
71.             v3[s] = trans2[s] - trans0[s]
72. 
73.         ######################################################################
74.         ## Test de angulos '/right_shoulder', '/right_arm'
75.         ######################################################################
76. 
77.         art1 = (math.acos(v3[1]/np.linalg.norm(v3)))-1.57
78.         art2 = (math.acos(v1[0]/np.linalg.norm(v1)))
79. 
80.         # Angulo formado entre v1 y v2
```

```
81.         art3 = math.acos(np.dot(v1,v2)/(np.linalg.norm(v1)*np.linalg.norm(v2))
    )
82.
83.         # Resto de angulos a 0, ya que no es posible un control de ellos con u
    n solo brazo
84.         # Posible control con el brazo izquierdo
85.         art4 = 0
86.         art5 = 0
87.         art6 = 0
88.
89.         arm_angulos = [art1, art2, art3, art4, art5, art6]
90.
91.         sec += 1
92.         pub.publish(sec, arm_angulos)
93.
94.         rate.sleep()
```

14.1.4. kinect2_arm.py

```
1.  #!/usr/bin/env python
2.  ###############################################################################
3.  #
4.  #   Copyright (c) 2014
5.  #   Pablo Melero Cazorla for Universidad de Almeria
6.  #
7.  ###############################################################################
8.
9.  import roslib
10. roslib.load_manifest('ki2_arm')
11. roslib.load_manifest('cob_script_server')
12. import rospy
13. import math
14. import tf
15. import os
16. import numpy as np
17. from ki2_arm.msg import ArmPos
18.
19. from simple_script_server import *
20. sss = simple_script_server()
21.
22. ###############################################################################
23. #       INICIO
24. #
25. # Introducir los datos pasados con los que se desea trabajar
26. # Y el numero de modulos o articulaciones
27.
28. n_articulaciones = 3
29. m_datos_pasados = 5
30.
31. ###############################################################################
32. #       MATRIZ HIST[nxm]
33.
34. mat_hist = np.zeros([n_articulaciones,m_datos_pasados])
35.
36. v_mov = np.zeros(3)
37.
38. # Grado de variacion inima
39. grado_min = 5
```

```
rot_min = grado_min*(np.pi/180)


# Inicializamos el numero de mensajes recibidos en 0
msj_recibidos = 0

move = 0
move1 = 0

#####################################################################
##      FUNCIONES


def callback(msg):

    global mat_hist
    global move
    global move1

    for l in reversed(range(n_articulaciones)):

        for j in reversed(range(m_datos_pasados-1)):
            mat_hist[l,j+1]=mat_hist[l,j]

        mat_hist[l,0]=msg.angulos[l]

    #print msg.secuencia

    #detectar mov brazo
    if (abs(mat_hist[0,0]-mat_hist[0,1])>rot_min or abs(mat_hist[1,0]-
mat_hist[1,1])>rot_min or abs(mat_hist[2,0]-
mat_hist[2,1])>rot_min) and mat_hist[0,1]!=0:
        #print 'brazo moviendose'
        move=0
        move1=1

    #mover brazo tras 2
    if move>1 and move1==1:
        #print '\t \t\tMOVER BRAZO REAL'
        sss.move("arm", [[msg.angulos[0], msg.angulos[1], msg.angulos[2], 0, 0
, 0]])
        move1=0

    move +=1




def start():

    rospy.init_node('kinect2_arm')
    rospy.loginfo("Initializing all components...")
    sss.init("arm")
    #sss.move("arm","home")
    rospy.Subscriber("arm_pos", ArmPos, callback, queue_size = 1)
    rospy.spin()


#####################################################################
##      MAIN

if __name__ == '__main__':

```

```
99.     while not rospy.is_shutdown():
100.                 try:
101.                     start()
102.                 except rospy.ROSInterruptException:
103.                     pass
```

14.2. Archivos *launch*

14.2.1. skeleton.launch

```
1.  <launch>
2.    <arg name="fixed_frame" value="camera_depth_frame" />
3.
4.    <arg name="debug" value="False" />
5.    <arg name="launch_prefix" value="xterm -e gdb --args" />
6.
7.
8.    <include file="$(find openni_launch)/launch/kinect_frames.launch" />
9.
10.   <group if="$(arg debug)">
11.     <node launch-
    prefix="$(arg launch_prefix)" name="skeleton_tracker" pkg="pi_tracker" type="s
    keleton_tracker" output="screen"  >
12.       <param name="fixed_frame" value="$(arg fixed_frame)" />
13.       <param name="load_filepath" value="$(find pi_tracker)/params/SamplesConf
    igNewOpenNI.xml" />
14.     </node>
15.   </group>
16.   <group unless="$(arg debug)">
17.     <node name="skeleton_tracker" pkg="pi_tracker" type="skeleton_tracker">
18.       <param name="fixed_frame" value="$(arg fixed_frame)" />
19.       <param name="load_filepath" value="$(find pi_tracker)/params/SamplesConf
    igNewOpenNI.xml" />
20.     </node>
21.   </group>
22. </launch>
```

14.2.2. lwa4p.launch

```
1.  <?xml version="1.0"?>
2.  <launch>
3.
4.      <!-- send lwa4p urdf to param server -->
5.      <param name="robot_description" command="$(find xacro)/xacro.py '$(find sc
    hunk_lwa4p)/urdf/lwa4p.urdf.xacro'" />
6.
7.      <!-- robot state publisher -->
8.      <node pkg="robot_state_publisher" type="state_publisher" name="robot_state
    _publisher"/>
9.
10.     <!-- parameters for canopenmaster node -->
```

```
11.     <rosparam command="load" ns="canopen" file="$(find schunk_lwa4p)/config/ca
    nopen.yaml" />
12.
13.     <!--
     parameter description of the CAN modules and for the corresponding trajectory
     controller -->
14.     <rosparam command="load" ns="arm_controller" file="$(find schunk_lwa4p)/co
    nfig/controller.yaml" />
15.
16.     <!-- this is the CANopen ROS wrapper node -->
17.     <node ns="canopen" name="canopen_ros" pkg="ipa_canopen_ros" type="canopen_
    ros" cwd="node" respawn="false" output="screen" />
18.
19.     <!--
     the trajectory controller listens for JointTrajectoryFollowAction and sends v
    elocity commands to the CANopen node -->
20.     <node ns="arm_controller" name="joint_trajectory_controller" pkg="cob_traj
    ectory_controller" type="cob_trajectory_controller" cwd="node" respawn="false"
     output="screen" />
21.
22.     <!-- start diagnostics -->
23.     <node pkg="diagnostic_aggregator" type="aggregator_node" name="diagnostic_
    aggregator" >
24.         <rosparam command="load" file="$(find schunk_lwa4p)/config/diagnostics
    _analyzers.yaml" />
25.     </node>
26.     <node pkg="pr2_dashboard_aggregator" type="dashboard_aggregator.py" name="
    dashboard_aggregator" />
27.
28.     <!-- upload script server parameters -->
29.     <rosparam command="load" ns="/script_server/arm" file="$(find schunk_lwa4p
    )/config/joint_configurations.yaml"/>
30.
31. </launch>
```

14.2.3. lwa4p_sim.launch

```
1.  <?xml version="1.0"?>
2.  <launch>
3.
4.      <!-- launch an empty world -->
5.      <include file="$(find gazebo_worlds)/launch/empty_world.launch" />
6.
7.       <!-- send lwa4p urdf to param server -->
8.      <param name="robot_description" command="$(find xacro)/xacro.py '$(find sc
    hunk_lwa4p)/urdf/lwa4p.urdf.xacro'" />
9.
10.     <!-- spawn robot in gazebo -->
11.     <node name="spawn_gazebo_model" pkg="gazebo" type="spawn_model" args="-
    urdf -param robot_description -model lwa4p -
    z 0.01 " respawn="false" output="screen" />
12.
13.     <!-- robot state publisher -->
14.     <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot
    _state_publisher">
15.         <param name="publish_frequency" type="double" value="50.0" />
16.         <param name="tf_prefix" type="string" value="" />
17.     </node>
```

```
18.
19.     <!-- Fake Calibration -->
20.     <node pkg="rostopic" type="rostopic" name="fake_joint_calibration" args="p
    ub /calibrated std_msgs/Bool true" />
21.
22.     <!-- load controller -->
23.     <rosparam file="$(find schunk_lwa4p)/config/controller_sim.yaml" command="
    load"/>
24.     <node name="arm_controller_spawner" pkg="pr2_controller_manager" type="spa
    wner" args="arm_controller" />
25.     <group ns="arm_controller">
26.         <node name="arm_joint_trajectory_action_node" pkg="joint_trajectory_ac
    tion" type="joint_trajectory_action" />
27.     </group>
28.
29.     <!-- start diagnostics -->
30.     <node pkg="pr2_mechanism_diagnostics" type="pr2_mechanism_diagnostics" nam
    e="pr2_mechanism_diagnostics" />
31.     <node pkg="diagnostic_aggregator" type="aggregator_node" name="diagnostic_
    aggregator" >
32.         <rosparam command="load" file="$(find schunk_lwa4p)/config/diagnostics
    _analyzers.yaml" />
33.     </node>
34.     <node pkg="pr2_dashboard_aggregator" type="dashboard_aggregator.py" name="
    dashboard_aggregator" />
35.
36.     <!-- upload script server parameters -->
37.     <rosparam command="load" ns="/script_server/arm" file="$(find schunk_lwa4p
    )/config/joint_configurations.yaml"/>
38.
39. </launch>
```

14.2.4. ki2arm_follow.launch

```
1.  <?xml version="1.0"?>
2.  <launch>
3.
4.      <!-- send lwa4p urdf to param server -->
5.      <param name="robot_description" command="$(find xacro)/xacro.py '$(find sc
    hunk_lwa4p)/urdf/lwa4p.urdf.xacro'" />
6.
7.      <!-- robot state publisher -->
8.      <node pkg="robot_state_publisher" type="state_publisher" name="robot_state
    _publisher"/>
9.
10.     <!-- parameters for canopenmaster node -->
11.     <rosparam command="load" ns="canopen" file="$(find schunk_lwa4p)/config/ca
    nopen.yaml" />
12.
13.     <!--
     parameter description of the CAN modules and for the corresponding trajectory
     controller -->
14.     <rosparam command="load" ns="arm_controller" file="$(find schunk_lwa4p)/co
    nfig/controller.yaml" />
15.
16.     <!-- this is the CANopen ROS wrapper node -->
17.     <node ns="canopen" name="canopen_ros" pkg="ipa_canopen_ros" type="canopen_
    ros" cwd="node" respawn="false" output="screen" />
```

```
18.
19.     <!--
     the trajectory controller listens for JointTrajectoryFollowAction and sends v
    elocity commands to the CANopen node -->
20.     <node ns="arm_controller" name="joint_trajectory_controller" pkg="cob_traj
    ectory_controller" type="cob_trajectory_controller" cwd="node" respawn="false"
     output="screen" />
21.
22.     <!-- start diagnostics -->
23.     <node pkg="diagnostic_aggregator" type="aggregator_node" name="diagnostic_
    aggregator" >
24.         <rosparam command="load" file="$(find schunk_lwa4p)/config/diagnostics
    _analyzers.yaml" />
25.     </node>
26.     <node pkg="pr2_dashboard_aggregator" type="dashboard_aggregator.py" name="
    dashboard_aggregator" />
27.
28.     <!-- upload script server parameters -->
29.     <rosparam command="load" ns="/script_server/arm" file="$(find schunk_lwa4p
    )/config/joint_configurations.yaml"/>
30.
31.     <!-- skeletal tracking -->
32.     <include file="$(find ki2_arm)/launch/skeleton.launch"/>
33.
34.     <!-- nodos ki2_arm -->
35.     <node name="kinect_tf_listener_pub" pkg="ki2_arm" type="kinect_tf_listener
    _pub.py" />
36.     <node name="kinect2_arm" pkg="ki2_arm" type="kinect2_arm.py" />
37.
38. </launch>
```

14.2.5. ki2arm_follow_sim.launch

```
1.  <?xml version="1.0"?>
2.  <launch>
3.
4.      <!-- launch an empty world -->
5.      <include file="$(find gazebo_worlds)/launch/empty_world.launch" />
6.
7.       <!-- send lwa4p urdf to param server -->
8.      <param name="robot_description" command="$(find xacro)/xacro.py '$(find sc
    hunk_lwa4p)/urdf/lwa4p.urdf.xacro'" />
9.
10.     <!-- spawn robot in gazebo -->
11.     <node name="spawn_gazebo_model" pkg="gazebo" type="spawn_model" args="-
    urdf -param robot_description -model lwa4p -
    z 0.01 " respawn="false" output="screen" />
12.
13.     <!-- robot state publisher -->
14.     <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot
    _state_publisher">
15.         <param name="publish_frequency" type="double" value="50.0" />
16.         <param name="tf_prefix" type="string" value="" />
17.     </node>
18.
19.     <!-- Fake Calibration -->
20.     <node pkg="rostopic" type="rostopic" name="fake_joint_calibration" args="p
    ub /calibrated std_msgs/Bool true" />
```

```
21.
22.     <!-- load controller -->
23.     <rosparam file="$(find schunk_lwa4p)/config/controller_sim.yaml" command="
    load"/>
24.     <node name="arm_controller_spawner" pkg="pr2_controller_manager" type="spa
    wner" args="arm_controller" />
25.     <group ns="arm_controller">
26.         <node name="arm_joint_trajectory_action_node" pkg="joint_trajectory_ac
    tion" type="joint_trajectory_action" />
27.     </group>
28.
29.     <!-- start diagnostics -->
30.     <node pkg="pr2_mechanism_diagnostics" type="pr2_mechanism_diagnostics" nam
    e="pr2_mechanism_diagnostics" />
31.     <node pkg="diagnostic_aggregator" type="aggregator_node" name="diagnostic_
    aggregator" >
32.         <rosparam command="load" file="$(find schunk_lwa4p)/config/diagnostics
    _analyzers.yaml" />
33.     </node>
34.     <node pkg="pr2_dashboard_aggregator" type="dashboard_aggregator.py" name="
    dashboard_aggregator" />
35.
36.     <!-- upload script server parameters -->
37.     <rosparam command="load" ns="/script_server/arm" file="$(find schunk_lwa4p
    )/config/joint_configurations.yaml"/>
38.
39.     <!-- skeletal tracking -->
40.     <include file="$(find ki2_arm)/launch/skeleton.launch"/>
41.
42.     <!-- nodos ki2_arm -->
43.     <node name="kinect_tf_listener_pub" pkg="ki2_arm" type="kinect_tf_listener
    _pub.py" />
44.     <node name="kinect2_arm" pkg="ki2_arm" type="kinect2_arm.py" />
45.
46. </launch>
```

14.3. Archivos URDF

14.3.1. `lwa4p.urdf.xacro`

```
1.  <?xml version="1.0"?>
2.  <robot xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sens
    or"
3.         xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#
    controller"
4.         xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#i
    nterface"
5.         xmlns:xacro="http://ros.org/wiki/xacro">
6.
7.     <include filename="$(find schunk_description)/urdf/lwa4p/lwa4p.gazebo.xacr
    o" />
8.     <include filename="$(find schunk_description)/urdf/lwa4p/lwa4p.transmissio
    n.xacro" />
9.
10.     <xacro:macro name="schunk_lwa4p" params="parent name *origin">
11.
12.         <!-- joint between base_link and arm_0_link -->
```

```
13.         <joint name="${name}_0_joint" type="fixed" >
14.             <insert_block name="origin" />
15.             <parent link="${parent}" />
16.             <child link="${name}_0_link" />
17.         </joint>
18.
19.         <link name="${name}_0_link">
20.             <inertial>
21.                 <origin xyz="0 0 0" rpy="0 0 0" />
22.                 <mass value="0.29364"/>
23.                 <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0.
    01" />
24.             </inertial>
25.
26.             <visual>
27.                 <origin xyz="0 0 0" rpy="0 0 0" />
28.                 <geometry>
29.                     <mesh filename="package://schunk_description/meshes/lwa4p/
    arm_0_link.stl" />
30.                 </geometry>
31.                 <material name="Schunk/LightGrey" />
32.             </visual>
33.
34.             <collision>
35.                 <origin xyz="0 0 0" rpy="0 0 0" />
36.                 <geometry>
37.                     <mesh filename="package://schunk_description/meshes/lwa4p/
    arm_0_link.stl" />
38.                 </geometry>
39.             </collision>
40.         </link>
41.
42.         <!-- joint between arm_0_link and arm_1_link -->
43.         <joint name="${name}_1_joint" type="revolute">
44.             <origin xyz="0 0 0.11" rpy="0 0 0"/>
45.             <parent link="${name}_0_link"/>
46.             <child link="${name}_1_link"/>
47.             <axis xyz="0 0 1"/>
48.             <calibration rising="${arm_1_ref}"/>
49.             <dynamics damping="10" />
50.             <limit effort="370" velocity="2.0" lower="-
    6.2831853" upper="6.2831853"/>
51.             <safety_controller k_position="20" k_velocity="50" soft_lower_limi
    t="${-6.2831853 + 0.01}" soft_upper_limit="${6.2831853 - 0.01}" />
52.         </joint>
53.
54.         <link name="${name}_1_link">
55.             <inertial>
56.                 <origin xyz="0 0 0" rpy="0 0 0" />
57.                 <mass value="0.29364"/>
58.                 <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0.
    01" />
59.             </inertial>
60.
61.             <visual>
62.                 <origin xyz="0 0 0" rpy="0 0 0" />
63.                 <geometry>
64.                     <mesh filename="package://schunk_description/meshes/lwa4p/
    arm_1_link.stl" />
65.                 </geometry>
66.                 <material name="Schunk/Blue" />
67.             </visual>
```

```
68.
69.             <collision>
70.                 <origin xyz="0 0 0" rpy="0 0 0" />
71.                 <geometry>
72.                     <mesh filename="package://schunk_description/meshes/lwa4p/
    arm_1_link.stl" />
73.                 </geometry>
74.             </collision>
75.         </link>
76.
77.         <!-- joint between arm_1_link and arm_2_link -->
78.         <joint name="${name}_2_joint" type="revolute">
79.             <origin xyz="0 0 0" rpy="1.5708 0 0"/>
80.             <parent link="${name}_1_link"/>
81.             <child link="${name}_2_link"/>
82.             <axis xyz="0 0 1"/>
83.             <calibration rising="${arm_2_ref}"/>
84.             <dynamics damping="10" />
85.             <limit effort="370" velocity="2.0" lower="-
    2.0943951" upper="2.0943951"/>
86.             <safety_controller k_position="20" k_velocity="50" soft_lower_limi
    t="${-2.0943 + 0.01}" soft_upper_limit="${2.0943 - 0.01}" />
87.         </joint>
88.
89.         <link name="${name}_2_link">
90.             <inertial>
91.                 <origin xyz="0 0 0" rpy="0 0 0"/>
92.                 <mass value="1.68311"/>
93.                 <inertia ixx="0.03" ixy="0" ixz="0" iyy="0.03" iyz="0" izz="0.
    03" />
94.             </inertial>
95.
96.             <visual>
97.                 <origin xyz="0 0 0" rpy="0 0 0" />
98.                 <geometry>
99.                     <mesh filename="package://schunk_description/meshes/lwa4p/
    arm_2_link.stl" />
100.                    </geometry>
101.                    <material name="Schunk/LightGrey" />
102.                </visual>
103.
104.                <collision>
105.                    <origin xyz="0 0 0" rpy="0 0 0" />
106.                    <geometry>
107.                        <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_2_link.stl" />
108.                    </geometry>
109.                </collision>
110.            </link>
111.
112.            <!-- joint between arm_2_link and arm_3_link -->
113.            <joint name="${name}_3_joint" type="revolute">
114.                <origin xyz="0 0.350 0" rpy="0 3.14159 0"/>
115.                <parent link="${name}_2_link"/>
116.                <child link="${name}_3_link"/>
117.                <axis xyz="0 0 1"/>
118.                <calibration rising="${arm_3_ref}"/>
119.                <dynamics damping="5" />
120.                <limit effort="176" velocity="2.0" lower="-
    6.2831853" upper="6.2831853"/>
121.                <safety_controller k_position="20" k_velocity="25" soft_low
    er_limit="${-6.2831853 + 0.01}" soft_upper_limit="${6.2831853 - 0.01}" />
```

```
122.                </joint>
123.
124.                <link name="${name}_3_link">
125.                    <inertial>
126.                        <origin xyz="0 0 0" rpy="0 0 0"/>
127.                        <mass value="2.1"/>
128.                        <inertia ixx="0.03" ixy="0" ixz="0" iyy="0.03" iyz="0"
    izz="0.03" />
129.                    </inertial>
130.
131.                    <visual>
132.                        <origin xyz="0 0 0" rpy="0 0 0" />
133.                        <geometry>
134.                            <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_3_link.stl" />
135.                        </geometry>
136.                        <material name="Schunk/Blue" />
137.                    </visual>
138.
139.                    <collision>
140.                        <origin xyz="0 0 0" rpy="0 0 0" />
141.                        <geometry>
142.                            <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_3_link.stl" />
143.                        </geometry>
144.                    </collision>
145.                </link>
146.
147.                <!-- joint between arm_3_link and arm_4_link -->
148.                <joint name="${name}_4_joint" type="revolute">
149.                    <origin xyz="0 0 0" rpy="-1.5708 3.14159 0" />
150.                    <parent link="${name}_3_link"/>
151.                    <child link="${name}_4_link"/>
152.                    <axis xyz="0 0 1" />
153.                    <calibration rising="${arm_4_ref}"/>
154.                    <dynamics damping="5" />
155.                    <limit effort="176" velocity="2.0" lower="-
    2.0943951" upper="2.0943951"/>
156.                    <safety_controller k_position="20" k_velocity="25" soft_low
    er_limit="${-2.0943951 + 0.01}" soft_upper_limit="${2.0943951 - 0.01}" />
157.                </joint>
158.
159.                <link name="${name}_4_link">
160.                    <inertial>
161.                        <origin xyz="0 0 0" rpy="0 0 0"/>
162.                        <mass value="1.68311"/>
163.                        <inertia ixx="0.03" ixy="0" ixz="0" iyy="0.03" iyz="0"
    izz="0.03" />
164.                    </inertial>
165.
166.                    <visual>
167.                        <origin xyz="0 0 0" rpy="0 0 0" />
168.                        <geometry>
169.                            <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_4_link.stl" />
170.                        </geometry>
171.                        <material name="Schunk/LightGrey" />
172.                    </visual>
173.
174.                    <collision>
175.                        <origin xyz="0 0 0" rpy="0 0 0" />
176.                        <geometry>
```

```
177.                         <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_4_link.stl" />
178.                         </geometry>
179.                     </collision>
180.                 </link>
181.
182.                 <!-- joint between arm_4_link and arm_5_link -->
183.                 <joint name="${name}_5_joint" type="revolute">
184.                     <origin xyz="0 0.005263 0.365" rpy="1.5708 0 3.14159" />
185.                     <parent link="${name}_4_link"/>
186.                     <child link="${name}_5_link"/>
187.                     <axis xyz="0 0 1" />
188.                     <calibration rising="${arm_5_ref}"/>
189.                     <dynamics damping="5" />
190.                     <limit effort="41.6" velocity="2.0" lower="-
    6.2831853" upper="6.2831853" />
191.                     <safety_controller k_position="20" k_velocity="25" soft_low
    er_limit="${-6.2831853 + 0.01}" soft_upper_limit="${6.2831853 - 0.01}" />
192.                 </joint>
193.
194.                 <link name="${name}_5_link">
195.                     <inertial>
196.                         <origin xyz="0 0 0" rpy="0 0 0"/>
197.                         <mass value="0.807"/>
198.                         <inertia ixx="0.03" ixy="0" ixz="0" iyy="0.03" iyz="0"
    izz="0.03" />
199.                     </inertial>
200.
201.                     <visual>
202.                         <origin xyz="0 0 0" rpy="0 0 0" />
203.                         <geometry>
204.                             <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_5_link.stl" />
205.                         </geometry>
206.                         <material name="Schunk/Blue" />
207.                     </visual>
208.
209.                     <collision>
210.                         <origin xyz="0 0 0" rpy="0 0 0" />
211.                         <geometry>
212.                             <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_5_link.stl" />
213.                         </geometry>
214.                     </collision>
215.                 </link>
216.
217.                 <!-- joint between arm_5_link and arm_6_link -->
218.                 <joint name="${name}_6_joint" type="revolute">
219.                     <origin xyz="0 0 0" rpy="-1.5708 0 0" />
220.                     <parent link="${name}_5_link"/>
221.                     <child link="${name}_6_link"/>
222.                     <axis xyz="0 0 1" />
223.                     <calibration rising="${arm_6_ref}"/>
224.                     <dynamics damping="5" />
225.                     <limit effort="20.1" velocity="2.0" lower="-
    2.0943951" upper="2.0943951" />
226.                     <safety_controller k_position="20" k_velocity="25" soft_low
    er_limit="${-2.0943951 + 0.01}" soft_upper_limit="${2.0943951 - 0.01}" />
227.                 </joint>
228.
229.                 <link name="${name}_6_link">
230.                     <inertial>
```

```
231.                        <origin xyz="0 0 0" rpy="0 0 0"/>
232.                        <mass value="0.819"/>
233.                        <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0"
    izz="0.01" />
234.                    </inertial>
235.
236.                    <visual>
237.                        <origin xyz="0 0 0" rpy="0 0 0" />
238.                        <geometry>
239.                            <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_6_link.stl" />
240.                        </geometry>
241.                        <material name="Schunk/LightGrey" />
242.                    </visual>
243.
244.                    <collision>
245.                        <origin xyz="0 0 0" rpy="0 0 0" />
246.                        <geometry>
247.                            <mesh filename="package://schunk_description/meshes
    /lwa4p/arm_6_link.stl" />
248.                        </geometry>
249.                    </collision>
250.                </link>
251.
252.
253.                <!-- extensions -->
254.                <xacro:schunk_lwa4p_gazebo name="${name}" />
255.                <xacro:schunk_lwa4p_transmission name="${name}" />
256.
257.            </xacro:macro>
258.
259.        </robot>
```