Universidad Zaragoza
1542

Trabajo Fin de Grado

Sistema multirobot para el transporte colaborativo de objetos

Multirobot system for collaborative object transport

Grado en Ingeniería Informática

Ingeniería de Computadores

Autor

Miguel Burgh Oliván

Director

Gonzalo López Nicolás

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2022

AGRADECIMIENTOS

Quiero agradecer en primer lugar a mis padres, mi madre María Jesús y mi padre Don, porque me han apoyado, acompañado y aguantado incondicionalmente durante todos estos años en la universidad. Gracias por estar siempre ahí, sin vuestro apoyo no hubiese podido llegar hasta aquí.

En segundo lugar, a mi director Gonzalo, por su ayuda en la planificación, organización y dedicación de su tiempo en este Trabajo Fin Grado. Sin sus consejos, acotación del trabajo y su atenta lectura y escucha de mis soluciones no muy realistas, no sabría cuando hubiese terminado.

También quiero agradecer a mis amigos, por entenderme y apoyarme cuando los necesitaba.

Por último, a la Universidad de Zaragoza, aparte de ofrecer la oportunidad de hacer este trabajo quiero agradecer a los profesores que he tenido en el grado, gracias a los conocimientos adquiridos, me han permitido afrontar y aportar soluciones propias al trabajo.

RESUMEN

Sistema multirobot para el transporte colaborativo de objetos

La robótica es una de las ramas más importantes dentro de la automatización y los robots colaborativos son una de sus incorporaciones estrella en la actualidad, especialmente porque su ámbito de trabajo no se limita únicamente a la industria debido a que puede trabajar junto a las personas. En particular, este Trabajo Fin de Grado se enfoca en el desarrollo de un sistema multirobot que pueda realizar tareas de forma cooperativa como es el transporte de objetos. No hay mucha documentación en cómo desarrollar un sistema en donde se controle varios robots simultáneamente de forma correcta en el entorno de ROS, el cual es ampliamente utilizado en la investigación y prototipado para realizar pruebas antes de su puesta en producción.

Se han diseñado, desarrollado, implementado y evaluado experimentalmente dos soluciones: la primera es mediante el paquete de ROS MoveIt!, en donde el trabajo se centra sobre todo en su configuración para permitir el control simultáneo de varios cobots; y la segunda es la creación o utilización de un planificador de terceros que envía las órdenes directamente a los controladores encargados de realizar los movimientos de los cobots y cada uno de ellos tienen incorporado una pinza que les permiten realizar diferentes tareas. También está incorporado al sistema el dispositivo Leap Motion que es capaz de detectar, rastrear y reconocer gestos de las manos del usuario como interfaz para el control simultáneo de hasta dos cobots, permitiendo la manipulación de objetos.

Para el desarrollo de las soluciones propuestas se ha hecho un análisis con sus ventajas y desventajas de las soluciones, junto a sus diseños, arquitecturas y configuración del sistema. Las pruebas se han realizado con uno, dos y cuatro cobots en el simulador de Gazebo para ambas soluciones, también se ha realizado pruebas en el robot físico UR10 para una de las soluciones propuestas con un robot el cual está controlado por Leap Motion.

Como resultado de este trabajo se proporciona una implementación disponible en un repositorio público. El sistema desarrollado en el entorno ROS permite interactuar al usuario mediante gestos con la mano con varios manipuladores robóticos en simulación. El sistema también permite interactuar en un entorno real con un robot UR10. El sistema desarrollado ha sido evaluado y validado experimentalmente mostrando un correcto funcionamiento en tiempo real.

Índice

Capítulo 1

Introducción y objetivos

1.1. Motivación

Este proyecto se encuentra enmarcado en el área de conocimiento denominada *robótica*, el cual es multidisciplinar, en donde las aportaciones de disciplinas como la mecánica, física, electrónica, informática, etc. se combinan para diseñar máquinas con la capacidad de realizar tareas.

En este caso, la *máquina* es un *cobot* (robot colaborativo), que es un robot manipulador con la capacidad de realizar tareas junto a los operarios. El cobot es parte del robot de la universidad conocido como *Campero*, es la fase previa a su modelo comercial (*RB-EKEN 16* [1]).

Actualmente, la sociedad se encuentra viviendo en la *industria 4.0*, por la *digitalización* tanto de la industria como en la vida cotidiana de las personas. Este efecto viene acompañado de la *COVID-19*, que acentúa esta digitalización causada por un mayor consumo de contenido digital y esto requiere de una infraestructura para afrontarlo.

La robótica es uno de los elementos más importantes en esta industria 4.0, la aparición de robots colaborativos han permitido la robotización de tareas peligrosas, repetitivas o aburridas. Permitiendo la liberación de los operarios para realizar otras tareas de más valor que se traduce en un aumento en la producción. Pero esto no es nada nuevo, lo innovador es que pueden trabajar junto a los operarios, eliminando las *jaulas*, son fáciles de trasladar y de programar.

Estas características junto a su precio han permitido la expansión de la robótica, que lleva al aumento en las tareas robotizadas y la incorporación de nuevas tecnologías en las tareas, aumentando su complejidad, como es la inteligencia artificial, visión por computadora, realidad virtual, IoT (internet of Things), etc.

El trabajo a desarrollar en este proyecto es precisamente la incorporación de nuevas tecnologías para la realización de tareas, como es el control de uno o más cobots mediante una interfaz usando Leap Motion y también se implementará el control de varios cobots simultáneamente para la realización de tareas repetitivas.

Este TFG se enmarca dentro de las actividades del proyecto COMMANDIA [2], Robótica móvil colaborativa de objetos deformables en aplicaciones industriales (COMMANDIA, 2019) el cual está cofinanciado por el Programa Interreg Sudoe y por el Fondo Europeo de Desarrollo Regional (FEDER).

1.2. Introducción

Actualmente, la tecnología está en todos lados y es una parte importante de nuestras vidas, concretamente estamos viviendo la cuarta revolución industrial [3] y debido a la pandemia de la COVID-19, la digitalización se ha visto muy acelerada [4]. Las fábricas convencionales evolucionando a Smart Factories, incorporando sistemas físicos cibernéticos (CPS) en sus instalaciones para aumentar rendimiento y reducir costes (ver Figura 1.1).

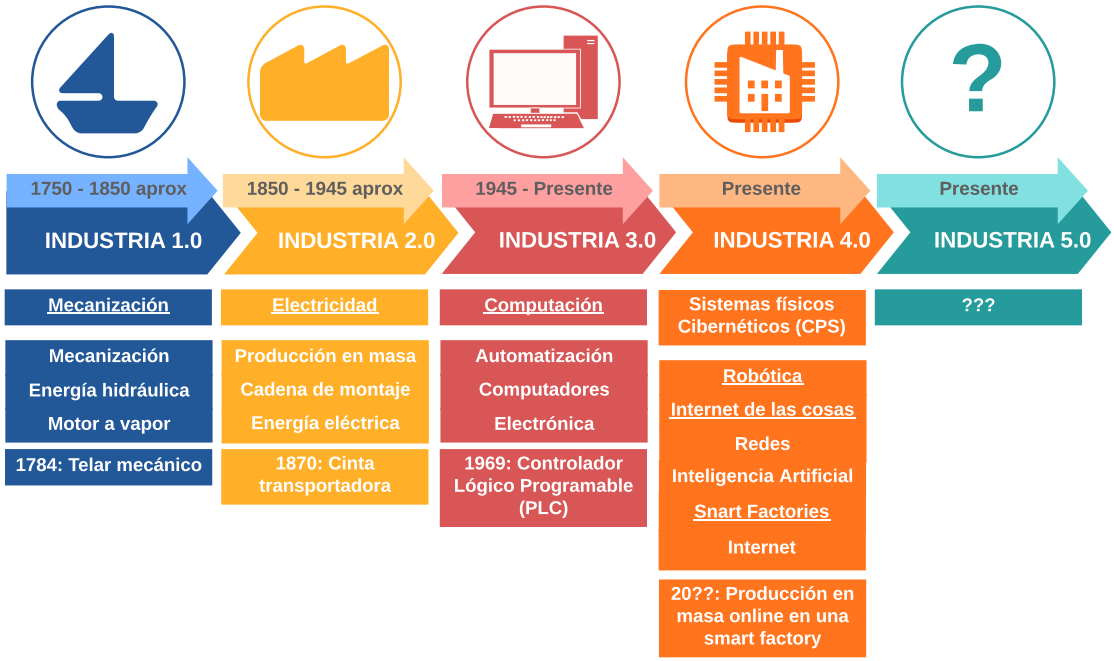


Figura 1.1: Industrialización 1.0 - 5.0

En estas fábricas, se han ido incorporando en los últimos años los cobots o robots colaborativos [5], y están siendo un componente muy importante. Pueden trabajar de

manera *segura* junto a los trabajadores, eliminando las jaulas y vallas de seguridad en sus entornos de trabajo, lo que permite un mejor aprovechamiento del espacio.

Los cobots son muy *flexibles*, ya que permiten ser programados para realizar diferentes tipos de trabajos sin mucha complejidad. También populares por su *rentabilidad* porque no es necesario una remodelación de las plantas de producción para usarlos, permitiendo a las empresas pequeñas y medianas automatizar sus instalaciones.

En el 2007 nació *ROS (Robot Operating System)* [6], es un *framework* para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo (Universidad de Standford).

Actualmente, es uno de los softwares más utilizados para trabajar con cobots por la cantidad de herramientas, bibliotecas, funcionalidad, así como una comunidad activa. No solamente utilizada para proyectos de investigación, sino también en las empresas que generan sus propios paquetes para ROS como es el caso de Universal Robots. Con esto, ROS se convierte en una pieza fundamental que permite el control de distintas marcas de robots/cobots en un único software.

1.3. Objetivos

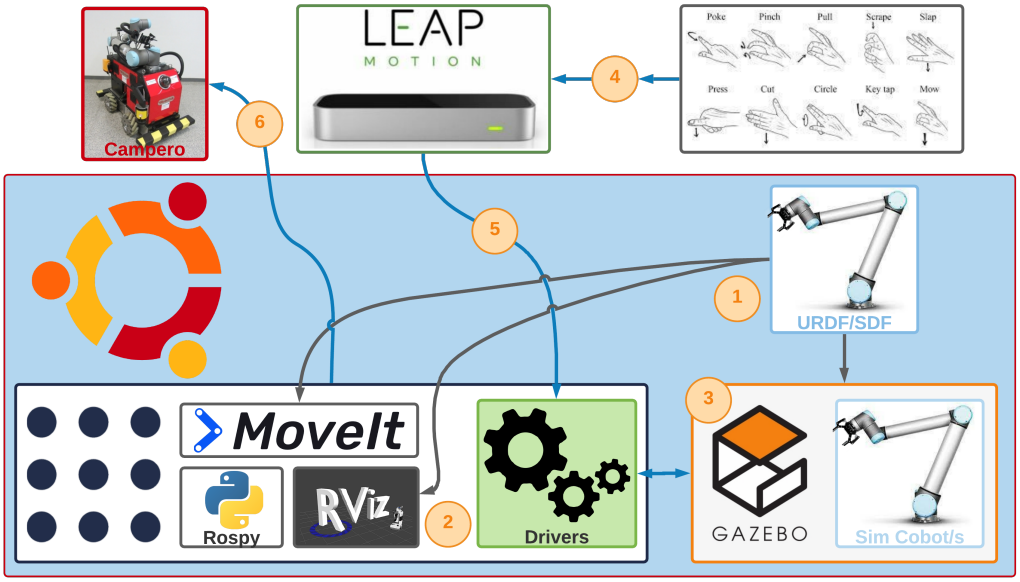


Figura 1.2: Esquema general del proyecto a implementar

El proyecto consiste en desarrollar un sistema en donde varios cobots interactúan con objetos, es decir, varios cobots que trabajan conjuntamente con objetos, que pueden estar o no compartidos, para realizar una tarea concreta mediante una interfaz.

Partiendo del esquema que se muestra en la Figura 1.2, se va a proceder a explicar los objetivos del trabajo realizado.

El proyecto puede ser muy complejo si se procede a realizarlo sin una planificación previa, por ello se ha dividido en pequeños módulos (objetivos), que están numeradas del 1 al 6 en el esquema de la Figura 1.2, que se irán conectándose entre ellos hasta generar el sistema completo. Se va a dividir los objetivos en varias fases:

- **Requisitos previos:** Instalación y familiarización con el nuevo entorno
  - Instalación del sistema operativo *Ubuntu 16.04*.
  - Instalación de *ROS Kinetic Kame*.
  - Documentación, conceptos y tutoriales de *ROS* y sus *herramientas*.
  - Documentación y pruebas de los *paquetes de ROS* que se puedan necesitar.
  - Aprendizaje y uso del lenguaje de programación *Python*.
  - Aprendizaje y uso de la interfaz y librería de *Leap Motion*.
- **Diseño:** Planteamiento de cómo implementar el sistema
  - Documentación e investigación de las herramientas disponibles para ROS.
  - Propuesta de posibles soluciones, sus problemas y coste de implementación.
- **Implementación:** División y desarrollo de las soluciones planteadas durante el diseño
  - Módulos de la implementación:
    1. Modificación del fichero URDF que modela el cobot, adaptándolo al proyecto a realizar.
    2. Creación, configuración y adaptación del paquete MoveIt! u otro planificador (es núcleo para el control del movimiento del cobot).
    3. Configuración y realización de pruebas en simulación con Gazebo.
    4. Configuración y realización de pruebas con Leap Motion.
    5. Configuración, creación y adaptación de los drivers necesarios para la comunicación entre Leap Motion y los módulos implementados previamente.
    6. Configuración y pruebas sobre el cobot físico.
  - Modificación de la configuración del sistema en los módulos del 1 al 3 para adaptarlos para dos y cuatro cobots.

- Modificación de la configuración del sistema en los módulos del 1 al 5 para adaptarlos para dos cobots controlados mediante la interfaz de Leap Motion.

– **Pruebas sobre el cobot físico:** Configuración del sistema para su funcionamiento en el cobot real de Universal Robots y controlado mediante Leap Motion.

– **Análisis:** Análisis sobre las soluciones implementadas, posibles mejoras, trabajo futuro y problemas encontrados.

1.4. Metodología

Para el desarrollo de cada una de las implementaciones así como de esta memoria se ha seguido la metodología que se muestra en la figura 1.3. En donde la zona de color rojo es donde el director del proyecto ha tenido más intervención mientras el resto lo ha realizado el estudiante.

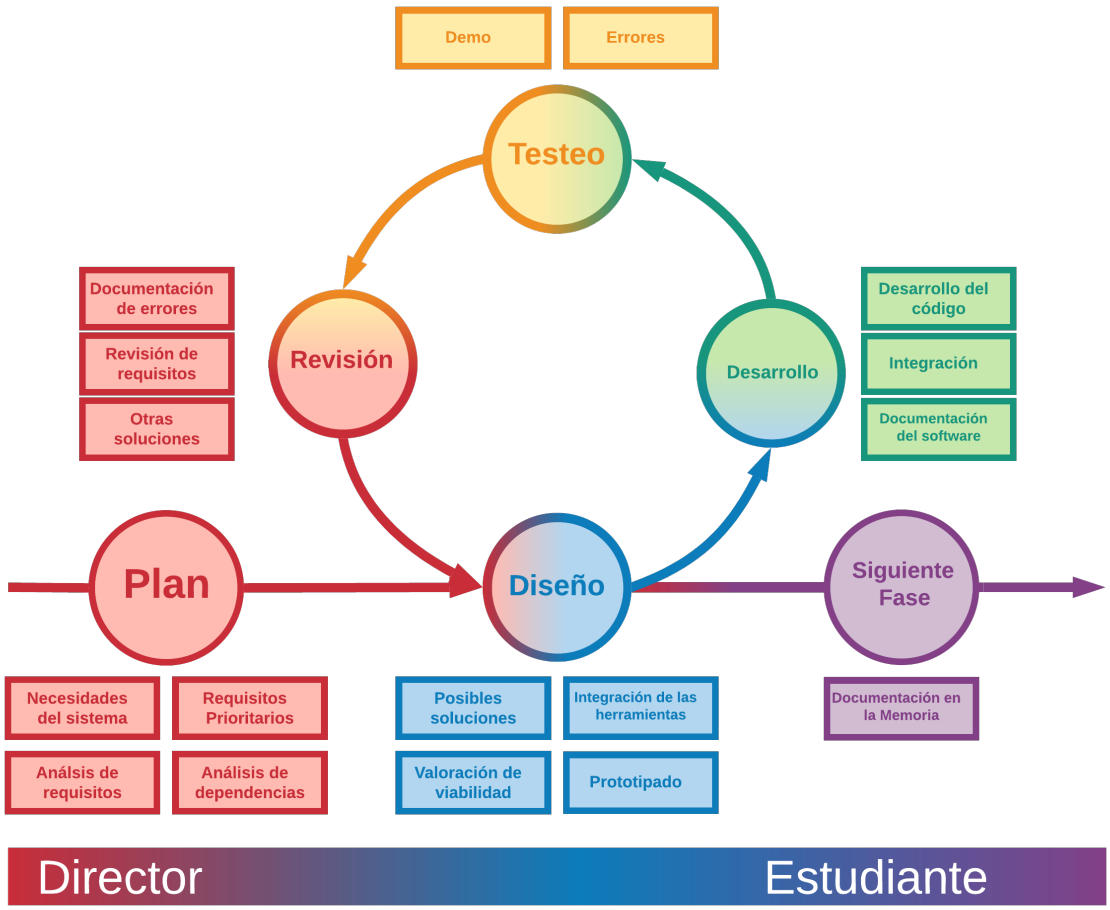


Figura 1.3: Metodología aplicada durante el desarrollo del TFG (Metodología Ágil[1])

Por tanto, es una interacción directa y constante entre el director y el estudiante, en el cual el director propone un plan junto al estudiante y es el estudiante quien diseña, implementa y realiza las pruebas para después realizar la revisión junto al director.

Durante la revisión el director puede dar el visto bueno a lo que el estudiante ha realizado y pasar a la siguiente fase en donde se realizaría de nuevo un plan, en caso de que no diese el visto bueno, es el estudiante el que principalmente propone posibles soluciones y es el director el que decide qué solución se seguirá tras analizar las propuestas del estudiante.

1.5. Estructura de la Memoria

Se va a realizar un breve resumen del contenido que abarca cada capítulo de esta memoria, el cual está estructurada en nueve capítulos:

- En este *Capítulo 1* se introduce el contexto, la metodología y los objetivos del trabajo.
- Después vienen los *Capítulos 2, 3 y 4* que documentan el entorno de desarrollo del trabajo, en donde se realiza una introducción a la disciplina de la robótica (destacando los robots colaborativos o cobots) y a la disciplina de interacción persona-ordenador para después introducir los elementos hardware (UR10 de Universal Robots, la pinza de Robotiq y el dispositivo Leap Motion) y los elementos Software (el entorno de Robot Operating System y sus paquetes) del sistema.
- Durante los *Capítulos 5 y 6* se explica el proceso de desarrollo del trabajo, en estos dos capítulos se explican las soluciones propuestas (diseño y análisis) y las soluciones desarrolladas (configuración, arquitectura, implementación e integración de Leap Motion).
- Los *Capítulos 7 y 8* trasladan la solución implementada al robot físico Campero y se muestran los resultados obtenidos de las soluciones desarrolladas.
- Y finalmente el Capítulo 9 contiene las conclusiones y trabajo futuro del trabajo desarrollado.

[1] *Metodología ágil*: el objetivo es el desarrollo de una tarea en poco tiempo al dividir el trabajo en tareas más pequeñas y en constante interacción con el cliente para mejorar la satisfacción y adaptación a los cambios que puedan surgir.

Capítulo 2

Estado de la materia

Este capítulo está dedicado a la robótica y poner en contexto el proyecto desarrollado. Se profundizará en los robots colaborativos especialmente, se introducirá las herramientas y software utilizados durante su implementación.

El proyecto se centra en la implementación de un sistema en donde varios cobots hagan un trabajo conjuntamente como es el *pick & place* de un objeto y que pueda ser controlado remotamente por una persona mediante *Leap Motion*.

2.1. Antecedentes

Con las características del proyecto en mente, se ha encontrado varios proyectos similares ya desarrollados, se van a nombrar algunos de los más actuales e interesantes:

- **Robótica 5G para telemedicina (2021):** Control de un brazo robótico mediante un dispositivo háptico, implementado con ROS y tecnología 5G para permitir su control en tiempo real [7].

- **Implementación de un Esquema de Teleoperación Utilizando el Sistema Operativo ROS en el Contexto de un Laboratorio Remoto (2012):** Control remoto de un brazo robótico (Jaco) mediante Leap Motion, con adicción de sensores conectados al sistema mediante Arduino[8].

- **Control teleoperado de un sistema multi-robot para aplicaciones quirúrgicas (2020):** Control remoto de dos robots IRB-120 de ABB mediante MATLAB, este proyecto no se utiliza ROS como las anteriores sino RobotStudio que actúa como servidor [9].

- **Mimic Kit: Human Skill – Machine Precision**

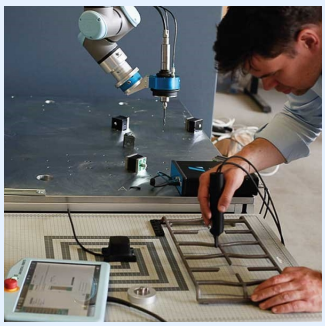


Figura 2.1: Mimic Kit de Nordbo Robotics [10]

En la figura 2.1 muestra un producto de Nordbo Robotics que es compatible con todos los modelos de Universal Robots [10].

La herramienta permite al cobot imitar los movimientos que el operario realiza en tiempo real y también pueden ser transferidos a otros cobots, es decir, puede controlar múltiples cobots simultáneamente [11].

Este producto es muy interesante porque es similar a lo que se quiere desarrollar y puede ser una solución alternativa para el control remoto de los cobots. Su salida en el mercado fue al principio del 2021 y su precio de unos 8000 € [12].

2.2. Introducción a la robótica

La robótica es una área multidisciplinar, en donde ingenieros industriales, mecánicos, electrónicos, eléctricos, informáticos..., ponen su conocimiento en común para su estudio, diseño, realización y manejo de los robots. El área de la robótica ha afectado principalmente a los entornos de trabajo en las industrias que actualmente es raro encontrar una que no esté influida por la tecnología que la robótica ha traído [13] [14].

Actualmente, las empresas se encuentran con el reto crear productos más complejos, en menor tiempo, que sean personalizados, con una rápida comercialización y su competitividad internacional. Para ello buscan que los procesos a la hora de elaborar sus productos sean flexibles, adaptables y fácilmente escalables [15].

Como respuesta a estos retos, la cooperación entre el operario y los robots es una de las respuestas a esos retos. Con la aparición de los *robots colaborativos* (cobots) que pueden trabajar junto a los operarios de forma segura, sin vallas y su alto grado de destreza, precisión, así como flexibilidad ha permitido aumentar la productividad de las empresas [15].

2.3. Historia de la robótica

El concepto de robots se puede encontrar en la antigüedad con los mitos de seres mecánicos [13][16], más tarde genios como *Leonardo da Vinci* dejaron su contribución [17]. Fue el escritor checo *Karel Capek* quién introduce la palabra robot con su obra *Rossum's Universal Robot* [16]. El primer robot industrial que fue puesto en operación

en 1961, se le llamó *Unimate* en la empresa de *General Motors* por *George Devol* y *Joe Engelberg* [17][18].

La introducción de los robots en la industria ha cambiado drásticamente el entorno de trabajo de los operarios, los procesos de producción y fabricación. Estos robots industriales han sustituido a los humanos en muchas de las tareas que requerían fuerza, eran repetitivos, peligrosas o simplemente imposibles para el ser humano realizarlas [16].

Pero las últimas dos décadas ha habido un cambio en la industria robótica con la introducción de los *robots colaborativos* o *cobots*, estos robots industriales se diferencian por su capacidad para trabajar junto a los operarios en donde la seguridad de este está garantizada. Los cobots aumentan las capacidades de los operarios, es decir, no son concebidos para sustituir al trabajador sino como un complemento, un aliado que cambiará el entorno de trabajo en las industrias de nuevo [16].

Las empresas actuales más importantes en la fabricación de cobots y sus primeras creaciones son: *KUKA* con su LBR 3 (2004), *Universal Robots* con su UR5 (2008), *Rethink Robotics* con su Baxter (2012), *FANUC* con su FANUC CR-35iA (2015) y *ABB* con su TuMi (2015) [19].

Pero dada la novedad de la tecnología, aún queda un largo camino, tanto para los operarios como para la directiva en la organización y distribución de tareas a realizar para crear el mejor entorno de trabajo y descubrir todas las posibilidades que esta nueva tecnología puede ofrecer [20]. No solo en las empresas, sino en la medicina, hogar, servicios públicos, investigación, etc [21]. Además, los cobots están siendo un elemento fundamental en la *industria 4.0* [20].

2.4. Clasificación de los robots – Arquitectura

Hay muchas formas de clasificar y categorizar a los robots porque son complejos con muchos elementos importantes que pueden ser tomados como foco principal para realizar su clasificación. Algunas de ellas son por su *arquitectura* como el que se presenta, pero existen por *generaciones*, *geometría*, *método de control*, *función que realizan*, etc [14][13].

2.4.1. Robots Poliarticulados

Figura 2.2: UR sobre un eje lineal eléctrico [37]

Karina Sanchez Madriz define a los robots poliarticulados como [22]:

> Robots de muy diversa forma y configuración cuya característica común es la de ser básicamente sedentarios (aunque excepcionalmente pueden ser guiados para efectuar desplazamientos limitados) y estar estructurados para mover sus elementos terminales en un determinado espacio de trabajo según uno o más sistemas de coordenadas y con un número limitado de grados de libertad.

En la Figura 2.2 se puede apreciar un cobot de *Universal Robots* de 6 grados de libertad (6 DOF), montado sobre un eje eléctrico dándole cierta libertad de movimiento y ampliando su espacio de trabajo.

2.4.2. Robots móviles

Figura 2.3: LD-250 de Omron [38]

Karina Sanchez Madriz define a los robots móviles como [22]:

> Robots con gran capacidad de desplazamiento, basados en carros o plataformas y dotados de un sistema locomotor de tipo rodante. Siguen su camino por telemando o guiándose por la información recibida de su entorno a través de sus sensores.

Se usan ampliamente para el transporte de mercancías, rescate, acceso a lugares complicados, exploración submarina, espacial, etc.

2.4.3. Robots Humanoides o Androides

Figura 2.4: Androide Erica de Hiroshi Ishiguro [23]

Karina Sanchez Madriz define a los robots humanoides como [22]:

> Robots que intentan reproducir total o parcialmente la forma y los comportamientos cinemáticos del ser humano. Actualmente, los androides son todavía dispositivos muy poco evolucionados y sin utilidad práctica, y destinados, fundamentalmente, al estudio y experimentación.

Erica fue creado en el 2015 y ha ido evolucionando desde entonces, el objetivo de su creado Hiroshi Ishiguro es que pueda mantener una conversación totalmente natural con una persona [23].

2.4.4. Robots Zoomórficos

Figura 2.5: Robot Spot de Boston Dynamics [24]

Karina Sanchez Madriz define a los robots zoomórficos de la siguiente manera [22]:

> Robots zoomórficos, que considerados en sentido no restrictivo podrían incluir también a los androides, constituyen una clase caracterizada principalmente por sus sistemas de locomoción que imitan a los diversos seres vivos.

Con la crisis del COVID-19, Boston Dynamics ha puesto a disposición de los sanitarios su robot *Spot* que provee de múltiples servicios a disposición del personal sanitario [24].

2.4.5. Robots Híbridos

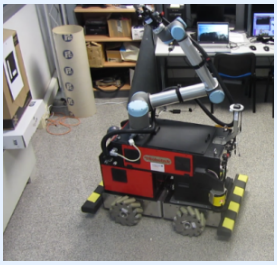


Figura 2.6: Robot Manipulador Móvil [39]

Karina Sanchez Madriz dice que los robots híbridos [22]

> Corresponden a aquellos de difícil clasificación cuya estructura se sitúa en combinación con alguna de las anteriores ya expuestas, bien sea por conjunción o por yuxtaposición.

El robot físico Campero del laboratorio encaja perfectamente en esta categoría, tiene un cobot UR10 sobre una plataforma móvil que es capaz de moverse de forma autónoma mientras que el brazo puede estar haciendo otra tarea, como es la de agarrar un producto que sería trasladado a otro sitio.

2.5. Robots Industriales y Robots colaborativos

Para saber qué son los robots colaborativos así como la razón de su nacimiento hay que conocer qué son los robots industriales primero, ambos tipos de robots conviven actualmente y tienen diferentes funciones y trabajos. Una breve descripción de ambos:

- **Robots industriales:** Son robots de gran envergadura y peso, por ello están ubicados en lugares muy concretos. Dado que suponen un peligro para los operarios, suelen estar *enjaulados*. Programados para trabajar en una tarea muy concreta y continuada. Sustituyen al operario en algunas tareas [25].

- **Robots colaborativos (cobots):** Son robots muy compactos y ligeros en comparación con el robot industrial y no necesitan de una zona seguridad. Permiten una sencilla reubicación, sencillos a la hora de programar diferentes tareas. Realizan tareas junto al operario como si fuese una extensión de ellos [25].

Resumiendo y remarcando las principales diferencias entre ambos tipos de robots:

Robot Industrial
Trabaja enjaulado sin la intervención directa de los operarios durante su trabajo
Son voluminosos y necesitan un lugar ser establecidos en un lugar fijo
Son creados específicamente para realizar una tarea en concreto
Son programados por un experto

Robot Colaborativo
Pueden realizar trabajo en colaboración con el operario sin jaulas
Son compactos, ocupan poco espacio y pueden cambiar de lugar fácilmente
Son creados para poder realizar diferentes tareas
Pueden ser programados por los operarios fácilmente

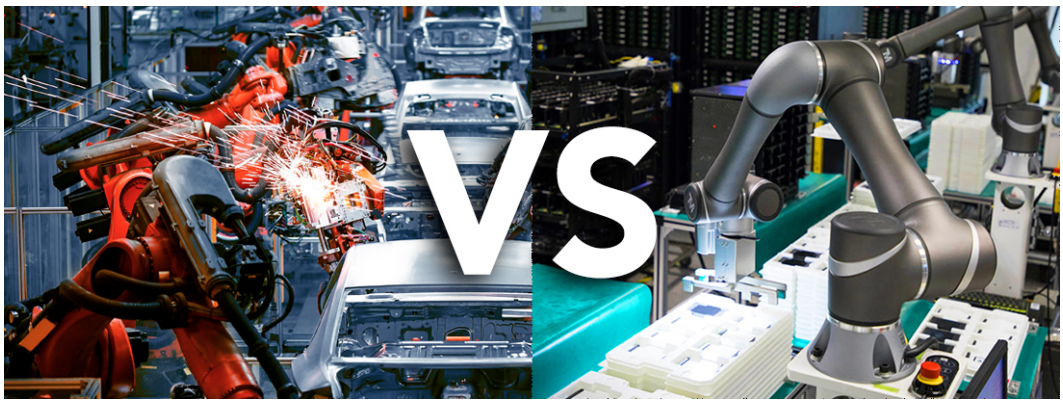


Figura 2.7: Robots Industriales VS Robots Colaborativos [40]

2.6. Interacción Persona-Ordenador: Leap Motion

"La interacción persona-computadora o persona-ordenador (IPO) es la disciplina dedicada a diseñar, evaluar e implementar sistemas informáticos interactivos para el uso humano, y a estudiar los fenómenos relacionados más significativos. ..." [26].

Figura 2.8: Interacción de una persona con un robot meadinte Leap Motion [41]

En la Figura 2.8 se muestra gráficamente los elementos que componen esta interacción. A la izquierda se encuentra el robot y el portátil, estos dos elementos conformarían el concepto de *ordenador*, después está el dispositivo de Leap Motion que haría su función de *interfaz* y finalmente la mano que realiza su función como *persona*.

Hay que tener en cuenta a la hora de diseñar la interfaz de comunicación es importante conocer a quién está dirigido, dado que el concepto de *persona* es muy amplio porque la persona puede tener distinta edad, físico, formación, etc. y se tiene que adaptar a los factores de cada persona o al grupo de personas al que es dirigido, de la misma manera, se debe adaptar al ordenador y la tecnología disponible.

La interacción persona-ordenador está muy integrada en la sociedad actual, tanto es así que puede parecer normal o pasar desapercibido si uno no se para a pensar el cómo se comunica con los computadores que le rodean. El elemento principal de la disciplina de la interacción persona-ordenador es la *interfaz*, que ha ido evolucionando a o largo de los años como se muestra en la Figura 2.9.

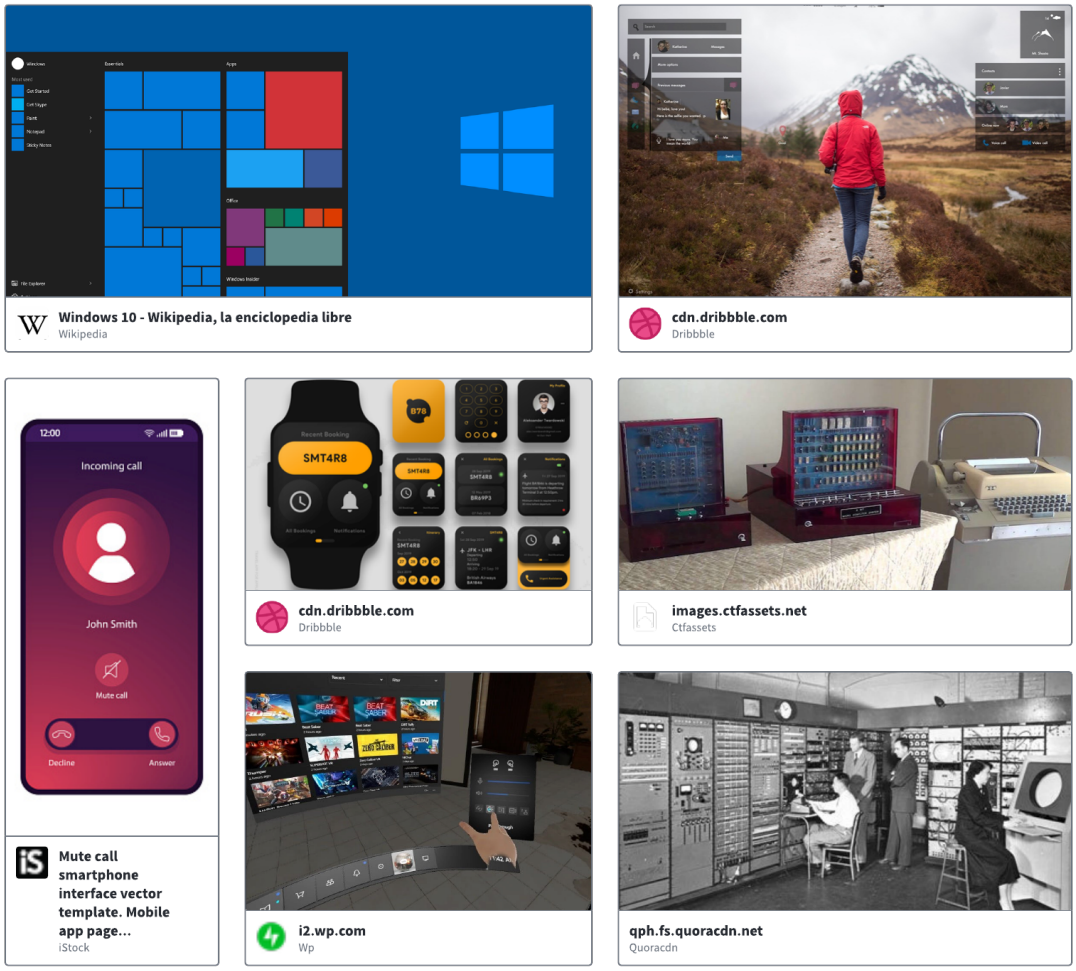


Figura 2.9: IPO: Algunas interfaces de usuario [42] [43] [44] [45] [46] [47] [48]

Se va a realizar un breve resumen de la evolución de esta disciplina basado en el artículo de Mireia Ribera Turró [27]:

– **Antes de 1967**

  Los ordenadores eran grandes máquinas que procesaban comandos en lenguaje máquina y hacían trabajo en modo *batch* con largos tiempos de espera como se ve en la Figura 2.10. La interacción con los ordenadores se realizaba mediante unas tarjetas perforadas que enviaba los comandos codificados al ordenador y el resultado se obtenía de una impresora, los usuarios eran los propios programadores [27].

  Es una etapa en donde los científicos crean diversas visiones, realización de pruebas experimentales, Bush pone la base del concepto de lo que se denominará *Memex* y es Ted Nelson quien crea las denominaciones, partiendo de la base que

propuso Bush, *hipertexto* e *hipermedia*. Licklider, propone dos conceptos que han dirigido la IPO desde entonces: el ordenador como asociado y como medio, en 1960 [27].

Figura 2.10: IPO: antiguos ordenadores [48]

En esta etapa también se estudia la interacción a tiempo real entre el usuario y el ordenador durante la ejecución de un proceso y la posibilidad de un sistema multiusuario, lo que es un gran avance con lo que en ese momento había (tareas realizadas en batch) [27].

Más tarde, Dough Engelbart realiza múltiples aportaciones prácticas al campo (pantalla con texto y gráficos, dispositivo apuntador, implementación del hipertexto, gestión de ventanas, etc.) y crea el primer procesador de textos [27]. Finalmente en 1963, Ivan Sutherland crea el programa *SketchPad* que permite la manipulación de gráficos, permitiendo realizar dibujos en la pantalla del ordenador [27].

– **De 1967 a 1977**

Como resultado de la investigación y experimentación de los años previos, los ordenadores experimentan un cambio importante, ahora cuentan con dispositivos

Figura 2.11: IPO: Antiguo ordenador soviético en 1967 [49]

de entrada y salida como es el teclado, una pantalla, impresora, etc. Su coste sigue siendo un problema a la hora de su expansión a un público más amplio, por tanto, sus principales usuarios siguen siendo los ingenieros e investigadores [27].

Aparecen en esta etapa las redes privadas a través de las líneas telefónicas y la interacción en tiempo real en 1971. La tecnología no está preparada para llegar a público en general, pero los diseñadores de la época señala esa posibilidad, concretamente el psicólogo Seymour Papert [27].

Más tarde Alan Kay realiza su tesis doctoral en 1969 sobre *el ordenador personal* que más tarde terminaría de desarrollarlo junto al equipo de PARC, es el padre de *SmallTalk* que permitió una programación incremental y la programación orientada a objetos después [27]. También aparece la primera hoja de cálculo (*VisiCalc*) y finalmente se expande las redes informáticas con la aparición del correo electrónico y la generación de pautas a la hora de diseñar interfaces [27].

– **De 1978 a 1988**

Esta es la etapa del ordenador personal en donde PARC tiene un papel principal al reunir a los principales talentos del área que fraguan los principales avances

(a) Celular (b) Apple Macintosh Plus 1986

Figura 2.12: IPO: Algunos inventos importantes de la época

tecnológicos e hipótesis del futuro. Gracias a la reducción del coste y tamaño de los ordenadores, los ordenadores llegan finalmente al público en general. El Macintosh Plus es uno de los ordenadores más exitosos de esta época (Figura 2.12b), los ordenadores estaban compuestos de una caja con el sistema central y sus dispositivos de entrada y salida, estos ordenadores son dirigidos principalmente al personal de oficina [27].

Aparte del éxito del ordenador personal, es una época muy rica en avances teóricos en el campo de la IPO:

- La incorporación de las teorías de la psicología cognitiva a la disciplina, los principales representantes son Card, Moran y Newell en 1983. Esta incorporación dio lugar al modelo GOMS (Goals, Operators, Methods y Selection rules) que es un modelo analítico que estima la bondad de un sistema interactivo [27].
- Se definen las bases teóricas de la usabilidad para crear interfaces más fáciles de usar (Bewley, Butler, Good, Spine, Whiteside, George) centrada en el usuario en vez de en la máquina [27].
- Establecimiento de las bases teóricas de la manipulación directa, reemplaza los comandos por acciones directas sobre objetos visibles, y su recopilación de diversas directrices heurísticas por Ben Shneiderman en 1982 [27].

Xerox lleva a la práctica la manipulación directa descrita por Shneiderman lo que

les llevó a definir la interfaz WIMP (Windows, Icons, Menus and Pointing device) que fue comercializado por Apple e imitado por Microsoft, WIMP es considerada el hito más importante para la disciplina de la IPO en esta etapa (ha prevalecido hasta la actualidad) [27].

La aparición de WIMP fue tan exitosa que frenó la investigación en otras interfaces de usuario e ideas con potencial, finalizando esta etapa con la adopción del modelo de desarrollo en espiral, en donde la especificación se realiza en múltiples iteraciones a partir de diseños previos o prototipos que se evalúan con la participación del usuario, y la redefinición de la usabilidad que tendrá también en cuenta el entorno y las situaciones concretas de la interacción e incorpora algunas técnicas de antropología [27].

– **De 1989 a 2005**

En esta etapa es difícil de sintetizar por la falta de perspectiva, pero se hará una división en tres apartados:

Figura 2.13: IPO: Programación ubicua [50]

- **La world wide web**

  Es uno de los cambios más importantes, la aparición de internet que aporta dos novedades:

  - Una interfaz centrada en el documento y no en la aplicación, que rompe los límites entre la información local y la remota [27].
  - Se convierte en un elemento clave en el desarrollo de la sociedad de la información, inmiscuyéndose en todos los aspectos de la vida social, tanto en la investigación como en la educación [27].

- **Continuismo y crisis en la disciplina**
  - Nielsen simplifica y divulga los métodos de usabilidad en 1993, basadas principalmente en evaluaciones cualitativas [27].
  - Aparición y consolidación de las redes inalámbricas en el área de la telecomunicación (Bluetooth, wifi, etc.) [27].
  - Miniaturización de los componentes informáticos que da paso a la aparición de dispositivos móviles como los portátiles y teléfonos móviles, su aparición en los electrodomésticos, relojes, coches, etc [27].
  - Aunque se sigue realizando investigaciones como es el reconocimiento del habla, en general no se realiza muchos avances ni aportaciones en el área de la IPO. Es una etapa en donde la disciplina realiza una autocrítica de lo conseguido y una mirada a las posibilidades que puede ofrecer con el avance de la tecnología sugiriendo nuevos posibles principios de diseño [27].
- **La computación ubicua, un nuevo paradigma**

  La computación ubicua (Figura 2.13), propuesta por Mark Weiser en PARC de Xerox, defiende que los ordenadores ya no están vinculados al escritorio, sino que están integrados en todos los aparatos y aspectos de la vida de una persona y que estas interactúan con ellas de forma transparente, lo que rompe con bastante de las presunciones de la base y las directrices del diseño en IPO [27].

– **De 2005 hasta la actualidad**

La tecnología ha avanzado dando pasos agigantados, cada día aparecen nuevas interfaces, aumento de la capacidad de almacenamiento, de banda ancha en las conexiones a Internet, nuevos paradigmas como la *computación en la nube*, etc [26].

Algunas de las investigaciones más importantes de la disciplina actualmente son:

- **Personalización del usuario**: "El desarrollo del usuario final estudia cómo los usuarios cotidianos podrían, rutinariamente, adaptar las aplicaciones a sus propias necesidades e inventar nuevas aplicaciones basadas en el entendimiento de sus propias capacidades." [26].
- **Computación incrustada**: "La computación está pasando de los ordenadores a cualquier objeto en el que se pueda aplicar. Los sistemas

incrustados hacen que el entorno esté vivo con pequeñas computaciones y procesos automatizados [...]" [26].

- **Realidad aumentada**: "La realidad aumentada se refiere a la noción de añadir información relevante a nuestra visión del mundo." [26].
- **Computación social**: "[...] investigaciones sobre ciencias sociales que se centran en interacciones cono la unidad de análisis." [26].
- **La interacción persona-computadora y las emociones**: "En la interacción entre personas y ordenadores, las investigaciones han estudiado cómo pueden los ordenadores detectar, procesar y reaccionar a las emociones humanas para desarrollar sistemas de información emocionalmente inteligentes." [26].

Para finalizar, decir la interacción entre la persona y el cobot, estaría enmarcada en el paradigma de la programación ubicua y la computación incrustada.

Capítulo 3

Herramientas del sistema – Hardware

En este capítulo se va a realizar una breve introducción de las herramientas que se van a usar o aplicar para el desarrollo del proyecto.

3.1. Cobot: Universal Robots

Figura 3.1: Logo de Universal Robots [10]

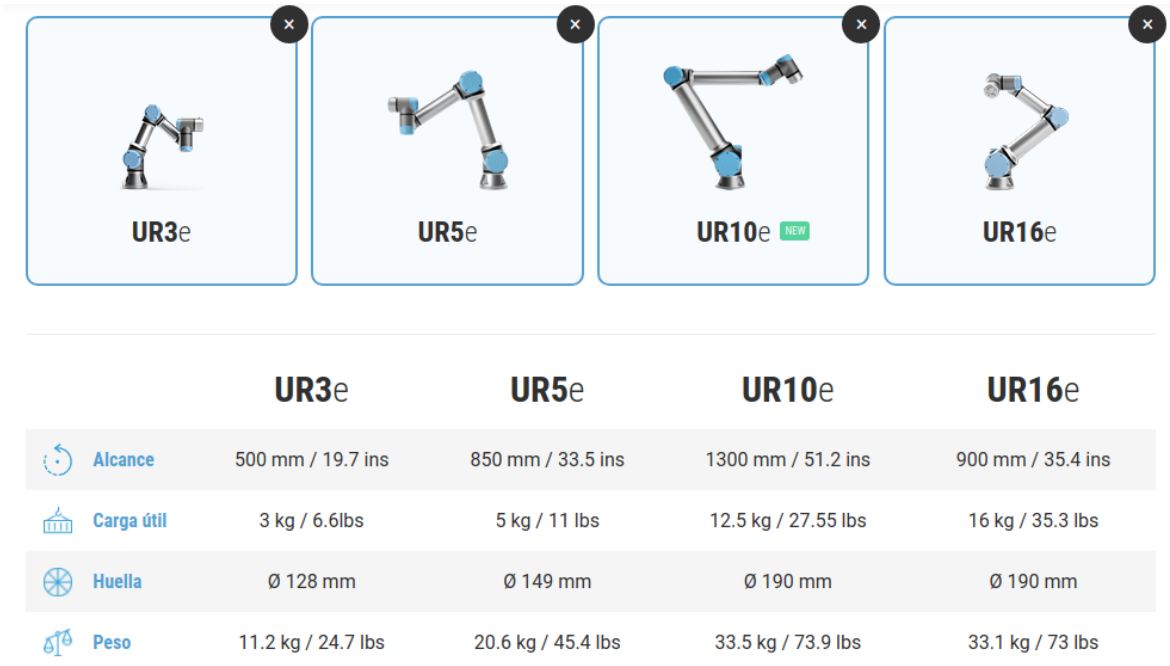
La empresa Universal Robots es actualmente una de las líderes en la producción y venta de robots colaborativos, esta empresa danesa fue fundada por oficialmente por Esben Østergaard, Kasper Støy y Kristian Kassow en 2005 con el objetivo de hacer la robótica accesible para las pymes [10].

En 2008 vendieron su primer cobot, el UR5 con seis grados de libertad, que se caracteriza porque es fácil de programar e instalar además de la flexibilidad para adaptarse al entorno de trabajo [10].

3.1.1. Tipos de cobots de Universal Robots

En la figura 3.2 se aprecia que Universal Robots tiene actualmente cuatro modelos: UR3, UR5, UR10 y UR16; y dos familias la tradicional y la e-serie (*e-serie* de *evolución, empoderamiento y easy to use*, en 2018). Los modelos e-serie viene con las mismas especificaciones que el original, pero se le han añadido más sensores y facilidades para su manejo [10].

Para este proyecto se va a utilizar el cobot UR10 tradicional, porque es el cobot que la universidad provee a sus alumnos para realizar sus investigaciones. Para el desarrollo del proyecto valdría cualquier otro cobot, porque las plataformas que se van a usar permiten fácilmente su incorporación.



	UR3e	UR5e	UR10e	UR16e
Alcance	500 mm / 19.7 ins	850 mm / 33.5 ins	1300 mm / 51.2 ins	900 mm / 35.4 ins
Carga útil	3 kg / 6.6lbs	5 kg / 11 lbs	12.5 kg / 27.55 lbs	16 kg / 35.3 lbs
Huella	Ø 128 mm	Ø 149 mm	Ø 190 mm	Ø 190 mm
Peso	11.2 kg / 24.7 lbs	20.6 kg / 45.4 lbs	33.5 kg / 73.9 lbs	33.1 kg / 73 lbs

Figura 3.2: Lista de modelos y características de Universal Robots [10]

3.1.2. Pinzas

En la figura 3.3 aparecen algunos de los end-effectors que proporciona Universal Robots, los end effectors son *pinzas* que realizan varios tipos de funciones. Se pueden encontrar *pinzas* de dos o más pinzas, pueden ser magnéticas, de succión, etc. Estas *pinzas* pueden estar combinadas con otras herramientas aumentando la adaptabilidad del cobot a la tarea que tenga asignada.

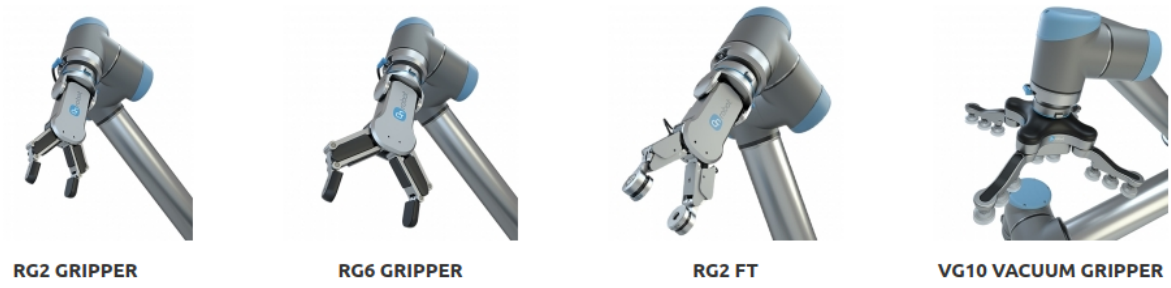


Figura 3.3: Algunos end-effectors de Universal Robots [10]

Para el proyecto, se utilizará una versión compatible para su simulación que se verá más adelante. La *pinza* será de dos pinzas, similar a los modelos RG2 y RG6 de la figura 3.3.

3.1.3. Robot manipulador - móvil: Campero (COMMANDIA)

Figura 3.4: Robot Campero del proyecto COMMANDIA [28]

En la figura 3.4 muestra el robot sobre el que se realizarán las pruebas del proyecto sobre un robot real, una vez que se termine de implementarlo.

El proyecto COMMANDIA, subvencionado por *Interreg* y *Sudoe* [28]:

> propone técnicas de control para manipuladores móviles (robots con brazos robóticos industriales) para llevar a cabo tareas industriales en las cuales han de manipularse productos blandos con destreza. Así, los operadores humanos no han de realizar manipulaciones repetitivas y agotadoras, pudiendo concentrarse en tareas más cerebrales.

Es un robot muy completo, compuesto, por una parte, móvil (controlado mediante un joystick) con ruedas que permite realizar movimientos laterales y sobre esta plataforma está montado un cobot UR10, sensores de movimiento y distancia, varias cámaras para la *visión por computadora*, dos servidores (uno que lleva el sistema operativo Ubutu 16.04 con ROS Kinetic Kame y el otro se encarga del robot UR10), una pinza de robotiq conectada al robot mediante USB, permite acceso remoto y local al robot.

3.2. Interfaz: Leap Motion

Figura 3.5: Leap Motion logo [51]

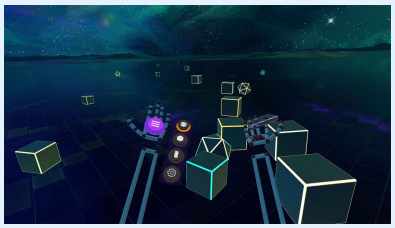


Figura 3.6: Leap Motion aplicación de Blocks [29]

El sistema de Leap Motion es capaz de reconocer y seguir los movimientos de las manos y los dedos. Se opera sin alejarse mucho del dispositivo, con mucha precisión y con una alta frecuencia de fotogramas reportando posiciones y movimientos discretos [29].

El software de Leap Motion se combina con los datos de sus sensores con un modelo interno de la mano que da apoyo a la hora de realizar el seguimiento de la mano real [29].

Posee de una API que permite el trabajo con los datos obtenidos para facilitar el desarrollo de aplicaciones con este dispositivo. Esta API se encuentra en varios lenguajes: JavaScript, Unity, C#, C++, Python, etc [29].

3.2.1. Leap Motion – Sistema de coordenadas

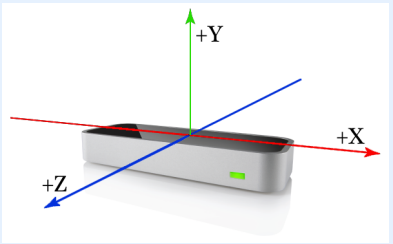


Figura 3.7: Leap Motion regla de la mano derecha [29]

El sistema de coordenadas de Leap Motion aplica la regla de la mano derecha. El origen está situado encima del centro del controlador Leap Motion [29].

En la figura 3.7 se aprecia que el eje X está situado paralelo al lado de mayor longitud del dispositivo, el eje Z incrementa su valor cuando la mano se mueve hacia el usuario y finalmente el eje Y sería el movimiento vertical con respecto al dispositivo, su valor se incrementa conforme la mano se aleja de este (hay que tener en cuenta que en el sistema que se usará, el valor de la Y disminuiría) [29].

Leap Motion – Datos de seguimiento de movimiento

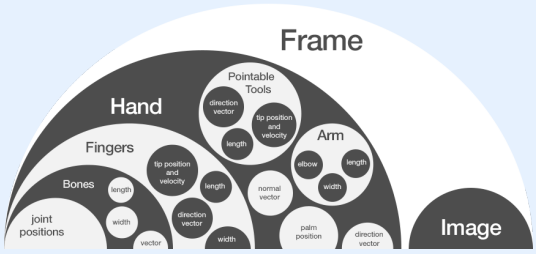


Figura 3.8: Composición de un *frame* de Leap Motion [52]

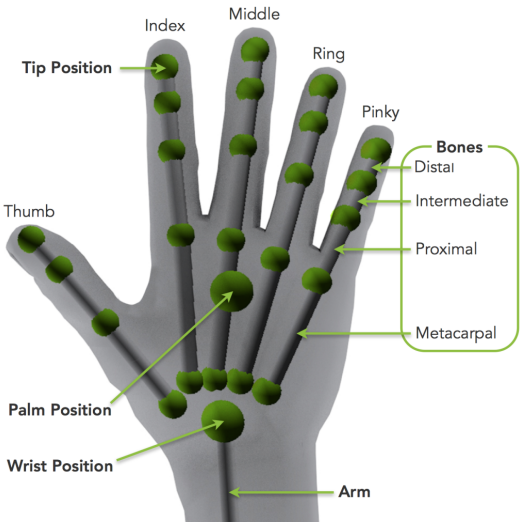


Figura 3.9: Composición del objeto *Hand* de Leap Motion [52]

Leap Motion provee actualizaciones de los datos mediante *frames*, que es un conjunto de datos que recoge la información de las manos que están dentro de su zona de trabajo. El tipo de información que es capaz de recoger incluye si es mano derecha o izquierda, su posición, el número de manos, la cantidad de dedos extendidos, que dedo está extendido, su orientación, distancia entre los dedos [29].

Capítulo 4

Herramientas del sistema – Software

4.1. Entorno: Robot Operating System – ROS

Figura 4.1: ROS logo [30]

En la página web oficial, definen ROS como: "ROS (Robot Operating System) is an open source software development kit for robotics applications. ROS offers a standard software platform to developers across industries that will carry them from research and prototyping all the way through to deployment and production." [30].

4.1.1. ROS – Kinetic Kame

Figura 4.2: ROS Kinetic Kame [31]

Esta versión de ROS fue lanzada en el 2016, la versión más actual es Noetic Ninjemys que fue lanzado en el 2020 [31]. Hay varias razones por las que se eligió Kinetic:

- El cobot UR10 funciona con ROS Kinetic
- Entre los requisitos de sistema operativo, hay que usar Ubuntu 16.04 y solamente ROS Kinetic puede funcionar sobre esta versión de Ubuntu correctamente.
- Otro requisito del proyecto es implementarlo usando Python, lo que ROS en sí cumple.
- La madurez de Kinetic frente a Noetic, tiene 5 años de recorrido en donde las herramientas, software, documentación, bugs, etc. están muy avanzadas en comparación con Noetic.

4.1.2. ROS – Arquitectura y Conceptos

En la Wiki de ROS, definen la arquitectura de ROS como:

> The ROS runtime "graph" is a peer-to-peer network of processes [...] that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

Siendo su objetivo principal de ROS dar apoyo a la reutilización de código en el desarrollo e investigación en la robótica [31].

ROS – Arquitectura: Roscore/Rosmaster

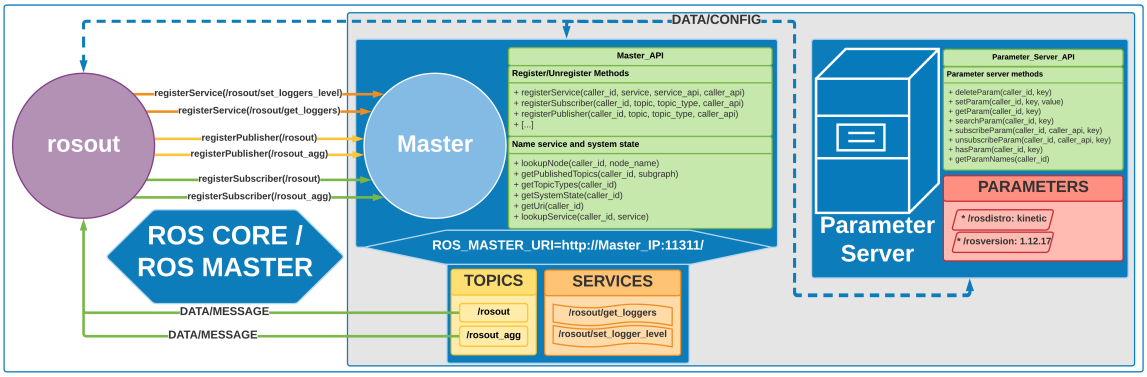


Figura 4.3: Esquema de Roscore/Rosmaster

Para poder explicar los conceptos de ROS, se ha decidido empezar por lo que hace el comando `roscore`. `roscore` ejecuta un conjunto de nodos (unidad más pequeña de computación en ROS) y programas (*Master*, *Parameter Server*, *rosout*), para dar soporte al sistema de ROS. Por ello `roscore` debe estar lanzado previamente a la adición de nuevos nodos para permitir su comunicación [31].

El concepto de **roscore** y **rosmaster** son diferentes, pero están íntimamente relacionadas porque es un requisito del sistema de ROS el ejecutar el comando `roscore` al inicializar, el cual ejecuta dos nodos, el nodo *Master* y *rosout*, siendo el nodo *Master* el que permite la comunicación entre los nodos. Por ello, roscore y rosmaster se utilizará en el futuro indistintamente para hacer referencia al nodo *Master*.

En la figura 4.3, se muestra un esquema simplificado del sistema al ejecutar el comando `roscore`, en ella se puede ver unos cuadros en color verde, que representan la API del *Master* y de *Parameter Server*, que provee los recursos que los nodos necesitan

como son: los servicios de registro (a services, topics, etc.), acceso a los datos del *Parameter Server*, dirección y comunicación con otros nodos, etc. Luego hay otros tres cuadros de color amarillo, naranja y rojo, representando los *Topics*, *Services* y *Parameters* respectivamente que se explicarán más adelante [31].

Hay que mencionar, que *Parameter Server* no es un nodo, porque está contenido dentro del nodo *Master* aunque en el esquema aparece separado por claridad. Su estructura datos es la de un diccionario, la información está compartida con todos los nodos y se aplica principalmente como parámetros de configuración [31].

Y finalmente el nodo *rosout*, todos los nodos al crearse también tienen una API genérica que provee información básica del nodo junto a la API que pueda proveer el propio desarrollador, pero por simplicar no se ha añadido en el esquema. *rosout* es un nodo que al ejecutarse, le pide al *Master* que cree dos *topics*, dos *Services* y después se suscribe a los *topics* creados, su finalidad es la de realizar un registro de los eventos del sistema [31].

ROS – Arquitectura: En una única máquina

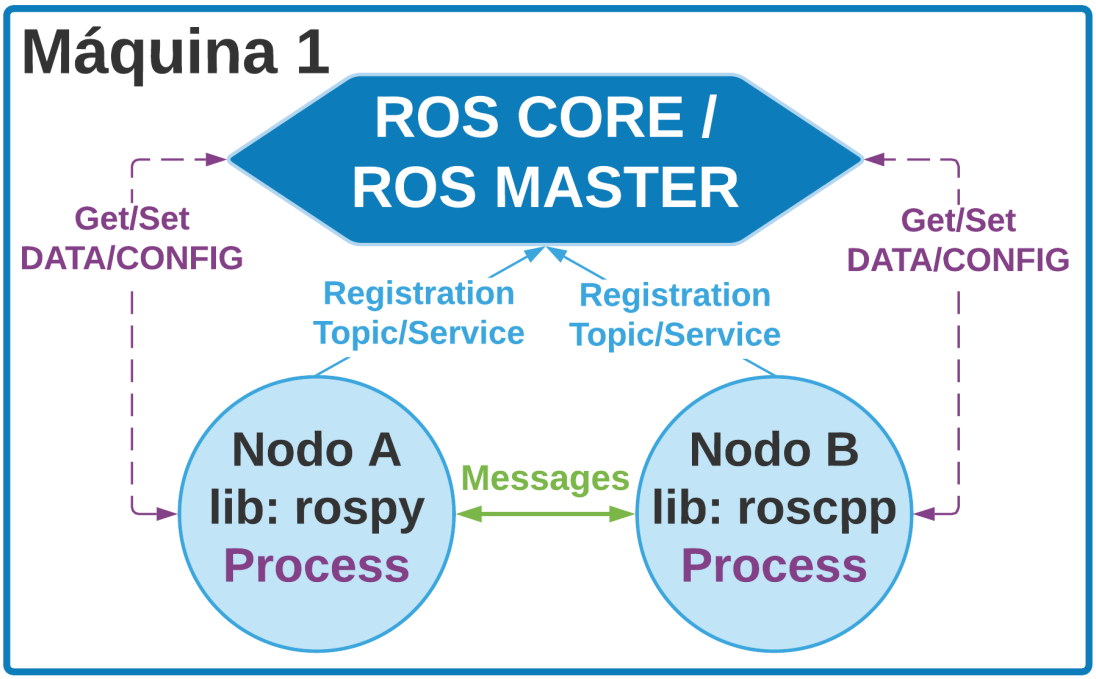


Figura 4.4: Esquema de la arquitectura de ROS en una única máquina

En el esquema realizado de la arquitectura de ROS que aparece en la figura 4.4, se aprecian varias cosas. Lo primero es que hay un único ROS Máster junto a N nodos, todos estos nodos tienen que estar registrados con el Máster al inicializarse,

los nodos son procesos independientes que no se conocen de antemano, pueden estar implementados en diferentes lenguajes (Python, C++, Lisp), que realizan ciertas acciones. Además, los nodos se pueden comunicar *asíncronamente* mediante la publicación y suscripción de/a topics o *síncronamente* mediante la llamada a servicios.

Se aprecia también que la arquitectura de ROS es centralizada, es decir, si el nodo Máster se cae, el sistema deja de funcionar correctamente. Se ha encontrado un repositorio que solventa el problema realizado por Pushyami Kaveti y Hanumant Singh, pero no está integrado en ROS oficialmente, ya que es el resultado de una investigación realizada por la Universidad de Cornell [32].

La versión oficial de ROS Kinect, no está preparada para tolerancia a fallos en ejecución y por ello no es adecuando aplicar ROS en producción pero sí para investigación, realizar prototipos, simulación, etc [33].

ROS – Arquitectura: Con varias máquinas

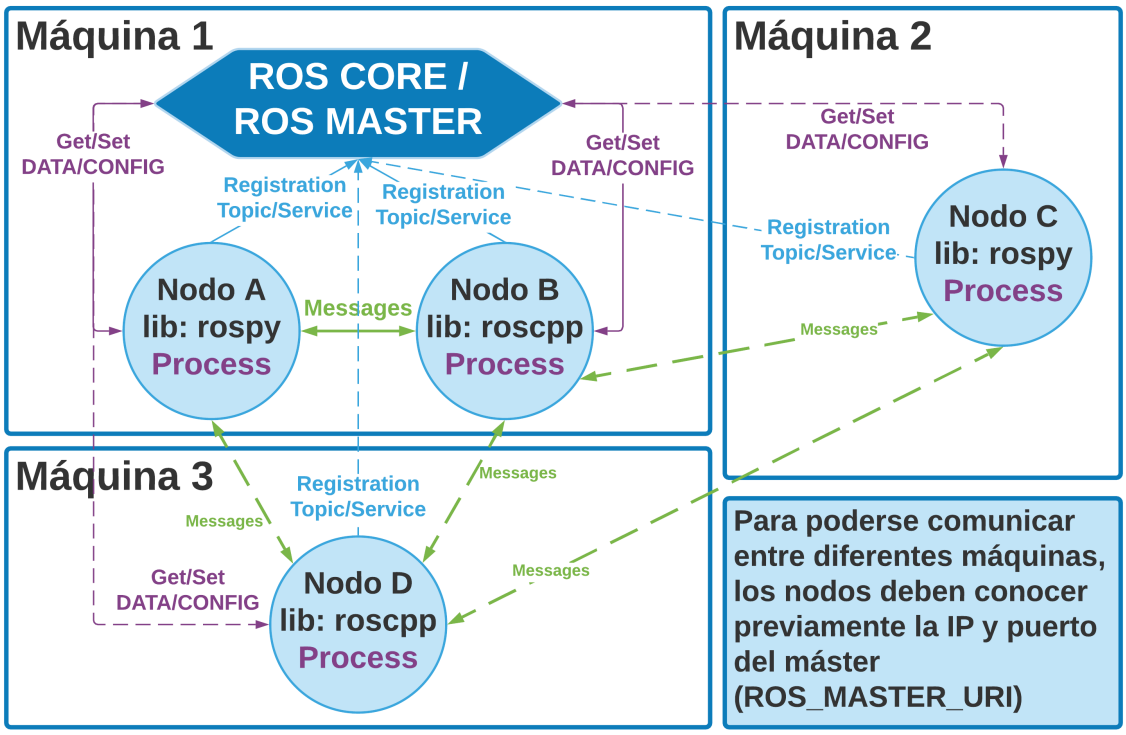


Figura 4.5: Esquema de la arquitectura de ROS con máquinas remotas

ROS permite la comunicación entre nodos que están en diferentes ordenadores, en el esquema de la figura 4.5, se aprecia cómo ROS máster está en una máquina con varios nodos en diferentes máquinas. Su funcionamiento es parecido al que se haría en una única máquina, la única diferencia es que necesita conocer previamente la dirección

URI del Máster para poder comunicarse con él, realizar su registro en el sistema y pedir los recursos que necesita para comunicarse con el resto de nodos.

Una de las diferencias más notables es que solamente existe un único máster para los nodos del sistema que se están comunicando, aunque estos nodos estén en diferentes máquinas [31].

ROS – Comunicación: Topics

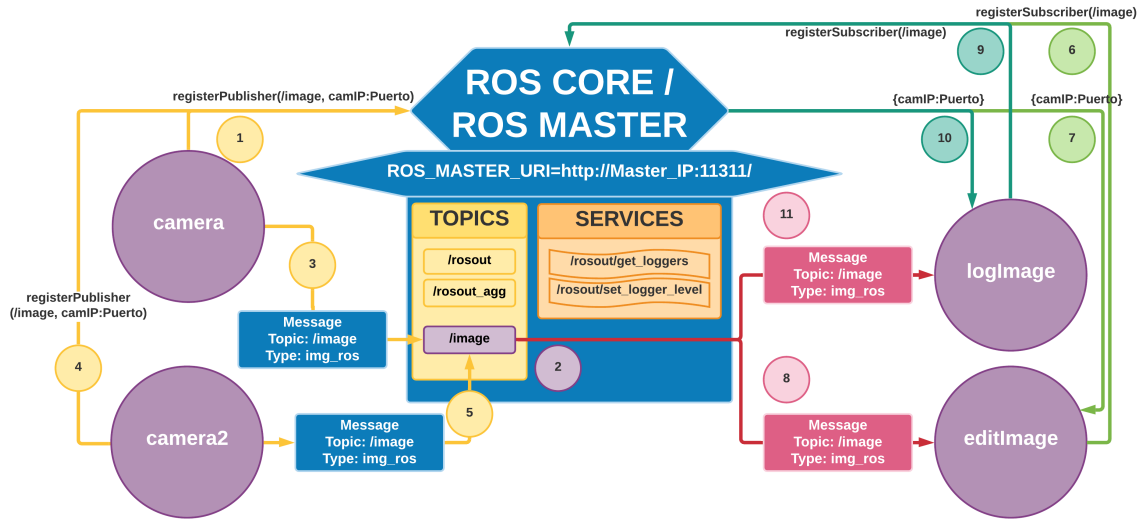


Figura 4.6: Esquema de la comunicación mediante *topics*

En el esquema de la figura 4.6, se presenta la comunicación asíncrona entre los nodos mediante la publicación y suscripción a los topics. Para saber qué es un topic, hay que saber qué es un *Mensaje*, un mensaje es una estructura de datos que tiene campos tipados (boolean, integer, float, etc.) los cuales pueden ser complejas, ya que permiten datos anidados. Sabiendo lo que es un *Mensaje*, un *topic* no es más que el nombre que se le da al canal por el que transcurre dicho tipo de mensajes que se publican [31].

El esquema de la Figura 4.6 se presenta un ejemplo de la comunicación, se comentará el proceso para el *nodo camara* que realiza publicaciones y el *nodo editImage* que está suscrito al topic, el proceso sería el mismo para los otros dos nodos.

Publisher	Subscriber
– **(1 y 4)**[1] Registra el topic */image* en el sistema realizando una petición al máster. – **(2)** El máster lo procesa y crea el canal para el topic */image*. – **(3)** El Topic ha sido creado correctamente, pero no realiza ninguna publicación ya que no existe suscriptores aún. – **(5)** Aparece la primera subscripción y se publica por el canal creado previamente.	– **(6 y 9)** Registra que quiere suscribirse al Topic */image* con el máster. – **(7 y 10)** El máster busca el Topic al que se quiere suscribir y le envía la información que necesita para realizar la suscripción y el máster comunica a los nodos que publican en el Topic que hay un nuevo nodo suscrito. – **(8 y 11)** Con la comunicación ya establecida, empieza a recibir mensajes por el Topic.

Puede existir N publicadores y N suscriptores para un Topic y cada nodo puede estar publicando a N y suscrito a N topics [31].

ROS – Comunicación: Services

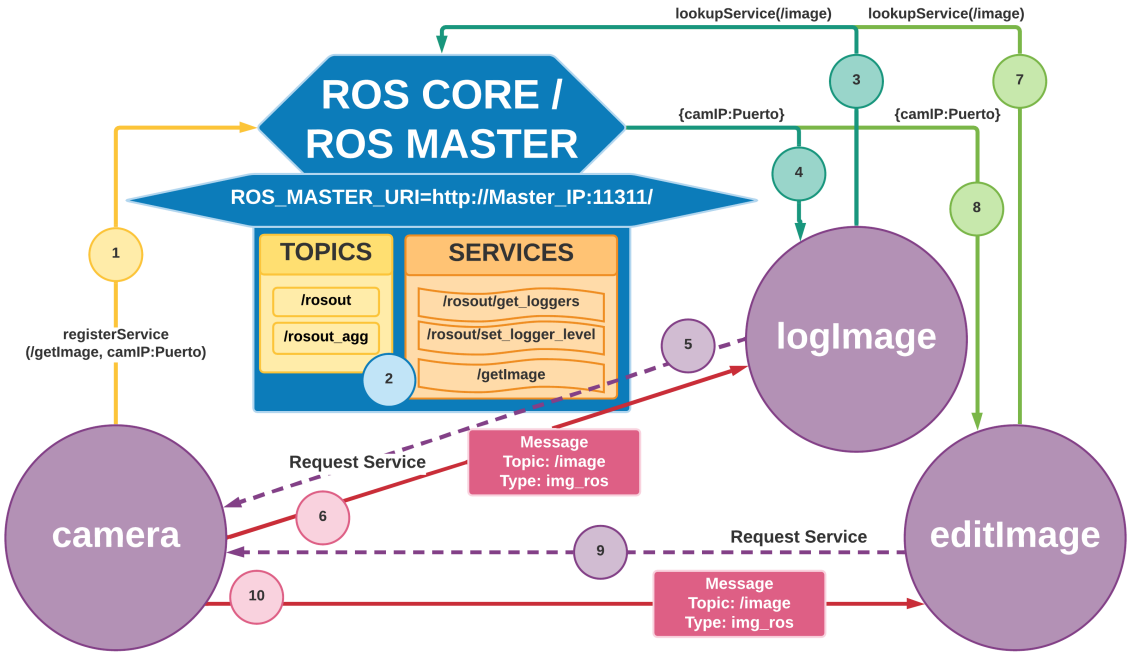


Figura 4.7: Esquema de la comunicación mediante *services*

En el esquema de la figura 4.7, se muestra la comunicación síncrona entre los nodos mediante los servicios.

[1]Los números representan las etapas que se muestran en la Figura 4.6

Los servicios permiten la comunicación entre los nodos como si fueran *cliente/servidor*, el cliente sería el nodo que solicita un servicio al servidor que es el nodo que provee de tal servicio, en este caso, el cliente es bloqueante en ROS, es decir, tiene que esperar a recibir la respuesta [31]. En el ejemplo del esquema, el *nodo camera* sería el servidor que provee de un servicio */getImage*, que es la de proveer una imagen. Lo primero que hace es registrar el servicio enviando una petición al máster y este lo añade en su lista de servicios disponibles.

Posteriormente, el *nodo editImage* quiere obtener una imagen por lo que procede a realizar su petición, pero no sabe que nodo provee de tal servicio y lo consulta con el máster. El máster le responde con la dirección del nodo y estable comunicación con el *nodo camera* y procede a realizar sus peticiones.

Finalmente, los servicios son únicos, es decir, dos nodos no pueden proveer un servicio con el mismo nombre, pero puede tener N nodos realizando peticiones al servicio creado.

ROS – Comunicación: Parameters

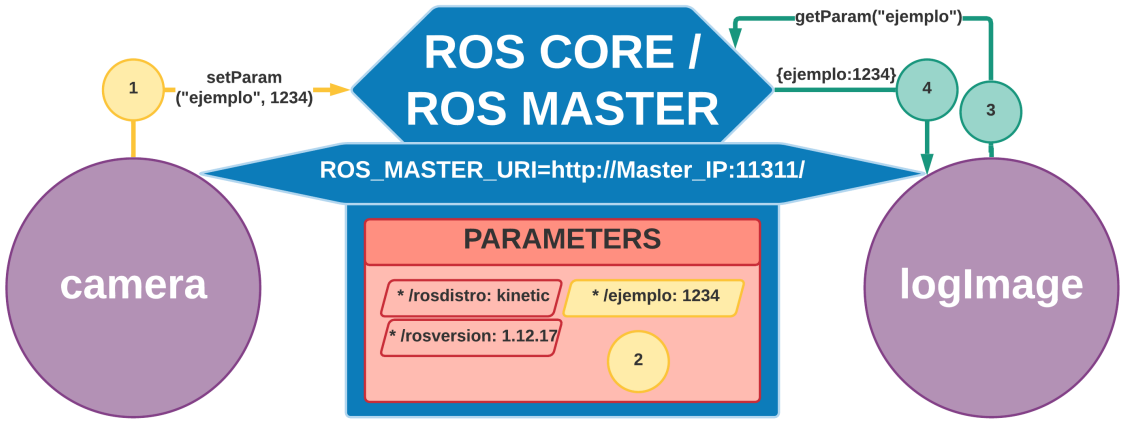


Figura 4.8: Esquema de la comunicación mediante *parameters*

En el esquema de la figura 4.8, se muestra el acceso y modificación de los datos almacenados en *Parameter Server*. El *Parameter Server* es una estructura de datos tipo diccionario, en el cual los nodos pueden almacenar y obtener datos en tiempo de ejecución, estos datos son visibles para todos los nodos y son accedidos mediante la API que se vio en la figura 4.3 [31].

En el ejemplo del esquema, el *nodo camera* crea un nuevo dato llamado `ejemplo` con valor `1234` en *Parameter Server* mediante la API. El dato almacenado es

posteriormente accedido por otro nodo, en este caso *nodo logImage*, obteniendo el dato que previamente se creó.

4.1.3. ROS – Packages

Los paquetes de ROS tienen el objetivo de proveer contenido, código, configuración, librerías, etc. para ser fácilmente reusados en nuevos proyectos. Hay una gran cantidad de paquetes disponibles para ROS, pero solamente se señalarán aquellos que son fundamentales para el desarrollo del proyecto.

Transforms – TF

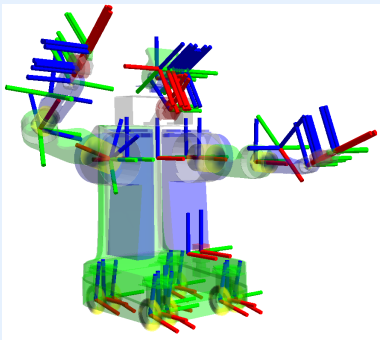


Figura 4.9: Imagen tf de un robot [31]

El objetivo de este paquete es permitir realizar un seguimiento de los cambios de múltiples coordenadas entre *frames* en el tiempo [31].

En robótica, la posiciones y orientaciones suelen ser expresadas como transformaciones de las coordenadas de un *frame* a otra. Hay que verlo como si fueran fotogramas de un vídeo, pero con la capacidad de obtener las coordenadas de cada punto, por tanto, de un fotograma a otro las coordenadas de esos puntos habrán cambiado. Lo que *tf* provee es justamente el poder realizar un seguimiento de esos cambios.

Universal Robotic Description Format – URDF

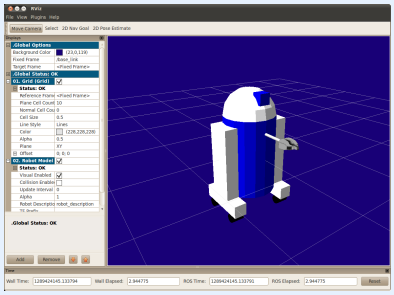


Figura 4.10: Vista del modelo URDF en Rviz [31]

El URDF es un paquete de ROS, concretamente es un formato de lenguaje para la descripción de robots siguiendo el formato del lenguaje XML, lo que permite a ROS simular y analizar el comportamiento del robot modelado [31].

La composición abstracta del modelo de un robot, se puede definir en enlaces y uniones, los archivos URDF justamente se estructuran de manera jerárquica (tiene una única raíz) para representar esos enlaces y uniones. Cada parte del robot se une mediante un *Joint*, que es representado mediante una relación padre-hijo en URDF, el cual puede tener otra relación padre-hijo [31].

robot_state_publisher

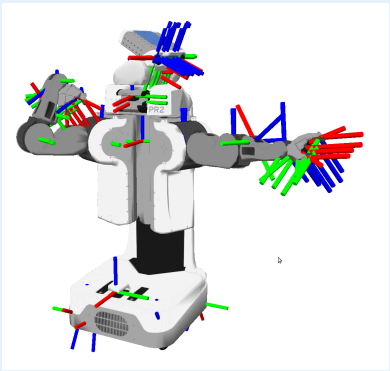


Figura 4.11: Imagen de paquete robot_state_publisher [31]

Permite al la publicación del estado de un robot al topic *tf*.

Este paquete usa el modelo URDF especificado en el parámetro `robot_description` y las posiciones de obtenidos en el topic `joint_states` para calcular el *forward kinematics* del robot y publicarlos por el topic *tf* [31].

MoveIt

MoveIt

Figura 4.12: MoveIt! logo [34]

El paquete *MoveIt!*, facilita el control de trayectorias para robots. Lo interesante de este paquete es que permite la planificación de los movimientos, manipulación, percepción 3D, cinemática, control y navegación [34]. Este paquete es usado ampliamente y se puede encontrar mucha documentación aparte del que el sitio oficial provee.

No se va a entrar en detalle sobre la arquitectura de *MoveIt!* ya que hay mucha documentación, por lo que se realizará un resumen de las funciones interesantes que este paquete puede ayudar en el desarrollo del proyecto.

- `moveit_commander`: Es una API en Python, que es el lenguaje en el que el proyecto debe desarrollarse. Hay que añadir que realmente es un *wrapper* sobre el código desarrollado en C++, es decir es una API en Python sobre la API ya realizada en C++.

- *Setup assistant*: El paquete provee de una forma sencilla la generación automática de la configuración del espacio de trabajo que el robot necesita, partiendo del URDF que se provee, en nuestro caso UR10.

- *Planning Scene*: Representa el mundo virtual, el cual el robot estará en simulación, permitiendo al robot interactuar con objetos.

– *Motion Planning*: Permite realizar planes, generando una trayectoria que el robot seguirá, ya sea especificando los valores de las articulaciones o la posición del *end-effector.*

– *Kinematics*: Permite al usuario desarrollar sus propios algoritmos Inverse/Forward Kinematics para transformar las coordenadas articulares a cartesianas y viceversa.

ur_modern_driver

Figura 4.13: Universal Robots Logo [10]

Este paquete está obsoleto, y solamente está para la versión de ROS Kinetic Kame. Es un driver para los controladores CB1 y CB2 para los robots UR5 y UR10 de Universal Robots únicamente [31].

Universal Robot

Figura 4.14: Universal Robots Logo [10]

Paquete únicamente para la versión de ROS Kinetic Kame, que da soporte a los robots de Universal Robot. Permite comunicarse con sus controladores, contiene varios modelos de sus robots (URDF) y su configuración para funcionar junto a *MoveIt!* [31].

Robotiq

Figura 4.15: Robotiq logo [53]

Paquete únicamente para la versión de ROS Kinetic Kame. Contiene drivers para *Robotiq Adaptive Grippers* y *Robotiq Force Troque Sensor*, el que se necesita para la implementación del proyecto son los drivers para el *Gripper* [31].

gazebo_ros_pkgs

Figura 4.16: Ros + Gazebo

Este paquete es una interfaz que permite a integración de ROS con el simulador Gazebo [31].

Este simulador es imprescindible para la simulación de varios robots que es el caso de este proyecto, ya que entre ellos no se ven, y Gazebo es capaz de simular varios robots con diferentes URDF simultáneamente, que no es el caso de Rviz, la siguiente herramienta.

Rosviz

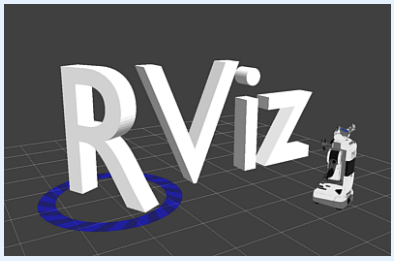


Figura 4.17: RViz logo [31]

Es un visualizador 3D para ROS. Permite la visualización del modelo del robot, captura y reproduce los datos capturados por los sensores (mediante los topics). Puede mostrar imágenes, nube de puntos, cámara, Mapas, láseres, etc.

Para este proyecto, interesa el pluging *Motion Planning* que permite establecer una posición de objetivo, realizar un plan y crear una trayectoria que después se puede visualizar su ejecución de forma interactiva [31].

leap_motion

Figura 4.18: Leap Motion logo [51]

Este paquete es un driver para los sensores que detecta los gestos de *Leap Motion* [31].

4.1.4. Gazebo – Integración en ROS

Figura 4.19: GAZEBO logo [35]

Gazebo es un simulador que permite simular el entorno y el robot sobre el que se trabajará. Aunque *Gazebo* utiliza modelos en formato SDF, permite transformar modelos en URDF a SDF para su posterior simulación.

La función principal de *Gazebo* es la de proveer de un entorno controlado para realizar pruebas de algoritmos, diseñar modelos de robots y también pruebas sobre un escenario *realista* del funcionamiento y comportamiento del robot.

Señalar que *Gazebo*, no es parte de *ROS*, ni es un paquete, es una entidad diferente llamada *Open Source Robotics Foundation* (OSRF), que mantiene el software gratuitamente para su uso y colabora con *ROS* siendo el simulador de preferencia por los usuarios de *ROS* [35].

Capítulo 5

Diseño del sistema

En este capítulo se realizará un análisis de los requisitos y las posibles soluciones del trabajo propuesto. Después de analizar las posibles soluciones se elegirá una como la solución definitiva y finalmente se diseñará esta solución para su posterior implementación.

Antes de proseguir con el diseño de la solución definitiva, se realizará un breve análisis de las herramientas disponibles, analizando brevemente sus capacidades y funciones que tendrían dentro de las soluciones definitivas a desarrollar. Hay que tener en cuenta que a la hora del desarrollo del proyecto se ha seguido la metodología ágil explicada en la Sección 1.4. Metodología del Capítulo 1 en la memoria. Lo que implica que durante el proceso pueden haber sucedido cambios que estarán debidamente señalados y razonados.

5.1. Propuesta

Partiendo de la información señala en el Capítulo 2, hay varios proyectos realizados e incluso comercializados que tienen unas características similares al que se quiere desarrollar (robot controlado por un dispositivo externo) como se señala en los antecedentes que se encuentra en la Sección 2.1. Antecedentes. Todos ellos tienen en común que están desarrollados y orientados para su aplicación a un único robot en cada momento.

Por tanto, la propuesta de este trabajo se basa en la misma idea, pero con la diferencia fundamental en que será enfocado para varios robots. Hay dos ideas que se quieren implementar, la primera es la demostración de que se pueden controlar varios robots haciendo un trabajo concurrentemente automatizado (sin intervención humana) y la segunda es la de controlar mediante un dispositivo externo (Leap Motion) varios robots simultáneamente (con intervención humana).

Con el desarrollo de este proyecto, se amplía la cantidad de tareas que se puede realizar así como el aumento de su complejidad, como podría ser la resolución de un

cubo de Rubik, ya que el cubo debe estar siempre con uno de los cobots sosteniéndolo, también se podría manipular objetos más pesados a su capacidad individual al tener dos cobots manipulando a la vez un único objeto y se evitaría problemas de sincronización si es controlado manualmente.

5.2. Análisis de los requisitos, herramientas y costes

Partiendo de los requisitos que el proyecto tiene, que han sido señalados previamente en el Capítulo 1 en la Sección 1.3. Objetivos en donde se señala los requisitos. Estos requisitos son: el tener sistema operativo Ubuntu 16.04 instalado, ROS Kinetic Kame será el entorno sobre que trabajará y sus diversos paquetes que se usará para controlar los cobots, hay que utilizar el lenguaje de programación Python, el simulador de Gazebo y como dispositivos externos se tiene el Leap Motion y el cobot de Campero en el caso de realizar pruebas sobre el Campero, una visión general de las dependencias se muestra en la Figura 5.1.

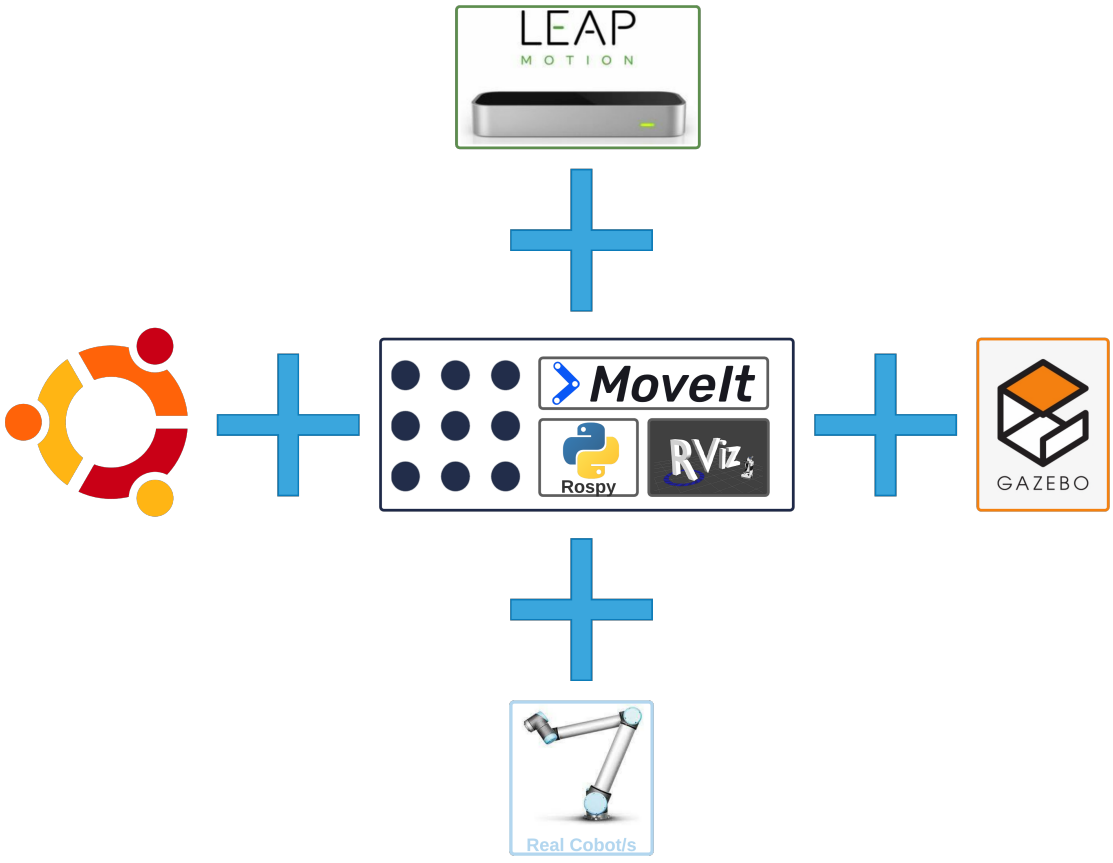


Figura 5.1: Requisitos del sistema

Estos requisitos limitan las soluciones posibles, ya que para comunicarse con el cobot

es obligatorio utilizar la versión de ROS Kinetic Kame, que es el que está instalado en el robot Campero, cuando se podría controlarlo con Matlab también.

Conociendo estas limitaciones el entorno de desarrollo es la que se ha mostrado en el Capítulo 4 que describen el entorno de ROS y los paquetes relacionados con el desarrollo del proyecto. Los paquetes están directamente relacionados con los elementos hardware descritos en el Capítulo 3, en este caso el cobot UR10 del robot Campero y el dispositivo externo Leap Motion.

Todos los paquetes de ROS pueden ser sustituidos por paquetes propios, pero si no hay necesidad no se reinventa la rueda. De estos paquetes el más importante es el planificador, es decir el paquete de MoveIt!, que permite el control de la trayectoria del cobot además de otras funciones integradas.

Como familiarizarse con el entorno de ROS es un requisito fundamental del proyecto, buscar soluciones fuera del entorno de ROS sería muy costoso, tanto en tiempo como en dificultad, en caso de tener que implementarlo de cero. Lo más efectivo es familiarizarse con todas las herramientas sugeridas en el Capítulo 4 y adaptar el contenido para desarrollar el proyecto.

Se ha de tener en cuenta que el estudiante, antes de comenzar este Trabajo Fin de Grado, no ha tenido contacto con ROS, ni ha cursado robótica durante el grado. Hay muchos elementos con los que se tiene que familiarizar y realizar sus propias investigaciones para poder adaptarse adecuadamente a los requisitos para la finalización del proyecto. Esto supone una limitación a la hora de sugerir otras posibles herramientas por la falta de conocimientos del área de la robótica, lo que implica la pérdida de un posible abanico de soluciones que se podrían aplicar.

5.3. Diseño de la solución

Partiendo de los requisitos de la sección anterior como se muestra en la Figura 5.1, se realiza el siguiente esquema general del sistema a desarrollar, en el cual se ha añadido flechas numeradas que representan las etapas o tareas que hay que realizar en ese orden con el objetivo de dividir el trabajo en pequeños módulos que se irán integrándose hasta llegar al robot real.

Hasta llegar al esquema general definitivo, se ha tenido que realizar un estudio de ROS y su entorno bastante intensivo, efectuar muchas modificaciones, pruebas y documentación de manera iterativa hasta solucionar cada uno de las etapas que se muestra en el esquema.

Aparte de la documentación de la parte enmarcada propia de la ingeniería informática, se ha buscado documentación para comprender cómo funciona el brazo del cobot, sobre todo porque es necesario el uso de las trayectorias cartesianas, las transformadas entre los componentes del brazo, pasar de valores articulares a cartesianas (Inverse Kinematic) y de cartesianas a articulares (Forward Kinematic). Esta documentación ha ayudado mucho a comprender cómo está configurado el sistema, ya que todo gira en torno al *cobot UR10*, desde los *topics* con los valores de los *joints*, los *tf* que muestran continuamente las posiciones de los *links*, la configuración que los drivers necesitan y a solventar los errores que aparecían más rápidamente.

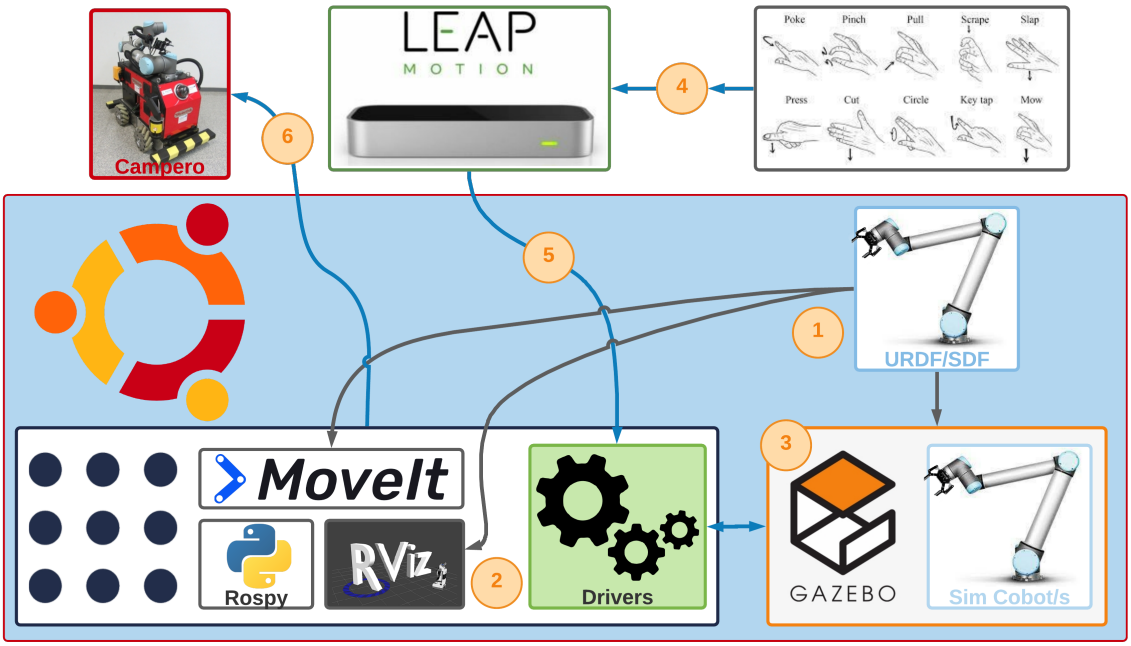


Figura 5.2: Diseño general de la arquitectura del sistema

Se va a describir las etapas que aparecen en la Figura 5.2, que serán las fases por las que pasará la implementación del sistema hasta llegar a la solución final:

1. Descripción del modelo del robot en formado URDF, el entorno de ROS lo utiliza para representar el robot en Gazebo así como en Rviz de forma realista. También es necesario para configurar correctamente MoveIt!, por tanto, también afecta a la configuración correcta del sistema.

   MoveIt! concretamente lo usa al lanzar su *Set Up Assistant* y para describir el contexto del planning, es decir, las posibles colisiones durante sus movimientos y ser consciente de los elementos que componente el modelo.

2. Esta es la fase más importante, es el cuerpo del sistema a desarrollar, en esta fase hay que instalar y configurar los paquetes, así como los ficheros `launch` que se

encargan de automatizar la configuración que el robot necesitará para funcionar correctamente. En esta fase no hay simulación, pero se puede visualizar el robot mediante la herramienta Rviz.

Se configuran los controladores tanto para el brazo como para la pinza, al principio puede ser una de las partes más costosas de entender por la cantidad, complejidad y poca claridad de los ficheros de configuración. También se crearán los drivers y scripts de Python que sean necesarios para el control del robot, en esta fase se puede ya realizar pruebas sobre el planificador para comprobar los movimientos del brazo y de la pinza.

Una vez que esta parte esta configurada, el resto de componentes se comunican con lo que se ha realizado hasta aquí, es decir, cada vez que se añada un nuevo componente que interactúa con el sistema, se debe adecuar la configuración realizada en esta fase de nuevo.

3. Rviz no es suficiente para realizar las simulaciones necesarias, ya que no es capaz de representar varios robots independientes porque cada robot tiene su propio URDF, por ello es necesario Gazebo para representar todos los robots en las simulaciones, además de esta razón, permite añadir y simular objetos.

   Gazebo, necesita lanzar los drivers que controlará el modelo descrito en el URDF que se le pasa, y esto puede ser confuso cuando se utiliza el paquete de MoveIt! también, da la sensación de que se esté duplicándolos, pero tiene sentido si los interpretas como interfaces de comunicación entre dos programas diferentes (que es lo que son, pero la mezcla de contenido en los ficheros de configuración puede confundir fácilmente al principio).

   También hay que añadir y configurar el plugin para agarrar objetos de Gazebo, si se simula únicamente un robot no es necesario, pero al añadir varios robots, sin el plugin bien configurado únicamente un robot agarraría el objeto lo que produciría una mala simulación del trabajo realizado.

4. Una vez se ha llegado a esta fase, quiere decir que toda la parte de simulación con los robots sin intervención del usuario se ha completado con éxito. Aquí se plantea el diseño de la interfaz entre el usuario y el sistema desarrollado hasta el momento.

Primero, hay que pensar cómo será esa interacción, qué movimientos y gestos de la mano son cómodos de realizar y qué representan. Después hay que usar la API de Leap Motion para captarlos y traducir esos gestos a variables que serán guardados en una estructura de datos para su futura comunicación el sistema.

5. Una vez completado el diseño de la interfaz de Leap Motion, se procede a crear los nodos de ROS necesarios para la comunicación entre ambos sistemas. Los datos transmitidos por Leap Motion que se codificaron previamente son publicados en un *topic* al cual estará suscrito otro nodo que traducirá los datos publicados y realizará una tarea según esos datos, en este caso controlar el cobot.

6. Finalmente, realizar las modificaciones necesarias para realizar las pruebas en el Campero, hay que señalar que para esta fase se elimina Gazebo en la comunicación y es sustituido por un Driver que se encarga de comunicarse con el robot real.

   Hay que calibrar velocidades, posiciones, espacio de trabajo, etc., ya que la simulación y la realidad son diferentes.

5.4. Soluciones propuestas y análisis

Tras realizar el análisis de los requisitos y las herramientas que se van a utilizar en la Sección 5.2 de este capítulo y partiendo del diseño general de la Figura 5.2 como referencia se puede apreciar que hay poco margen para proponer soluciones a lo que se ve en ese diseño inicial, debido principalmente al requisito del utilizar el entorno de ROS. Se van a analizar las ventajas y desventajas de las soluciones que se van a proponer para poder escoger la solución final de la manera más crítica posible.

Partiendo del diseño general de la figura 5.2, facilita el desarrollo de posibles soluciones y su entendimiento, se puede apreciar cuatro elementos que no son modificables: el sistema operativo Ubuntu 16.04, ROS Kinetic Kame, Gazebo, cobot UR10, Python 2.7 y el dispositivo de Leap Motion.

Eliminando los elementos intocables del diseño, el esquema resultante es el que se muestra en la Figura 5.3, se obtiene que las soluciones deben salir de las combinaciones entre el *fichero URDF* que define el modelo del robot, los *paquetes de ros* que en la imagen está representado por el paquete de MoveIt! que su función principal es la de planificador y los *drivers* que permiten la comunicación con el entorno de ROS.

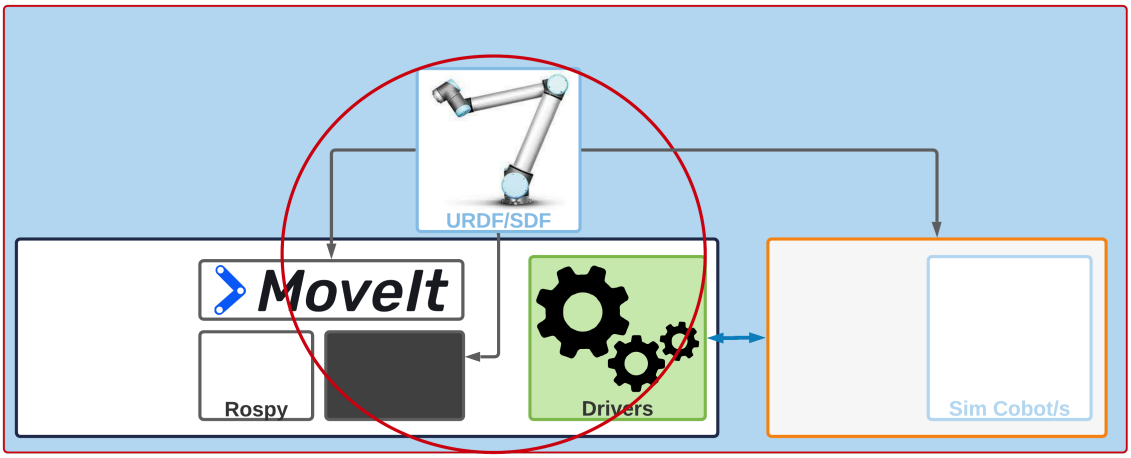


Figura 5.3: Herramientas y Drivers de ROS en el diseño

Las variaciones posibles son las siguientes:

– *Fichero URDF*: describe el modelo del robot, en este modelo se puede integrar varios robots, objetos o lo que se quiera modelar.

– *Paquetes de ROS*: Son los paquetes de ROS, pueden ser propios o instalados de terceros sobre el que se realizarán modificaciones para adaptarlos a la solución a desarrollar.

– *Drivers*: Es la interfaz entre los dispositivos externos y ROS para que se puedan comunicar entre ellos, no se realizará ninguna aportación o cambio de los drivers más allá de su instalación.

Simplificándolo, las soluciones que se proponen giran en torno a las modificaciones y combinaciones entre el modelado del robot (URDF) y los paquetes de ROS que se utilicen, por organización las soluciones propuestas se van a dividir en las soluciones que utilicen el paquete MoveIt! y las que no. Hay que tener en cuenta que el entorno de trabajo es complejo y hay muchos elementos que interaccionan o tienen dependencias entre sí, por lo que durante el desarrollo de los diseños propuestos pueden surgir problemas que no tienen solución o que el coste de corregirlos se expanda más allá de lo enmarcado en este Trabajo Fin de Grado.

Las soluciones propuestas son las siguientes:

– *Con el paquete MoveIt!*:

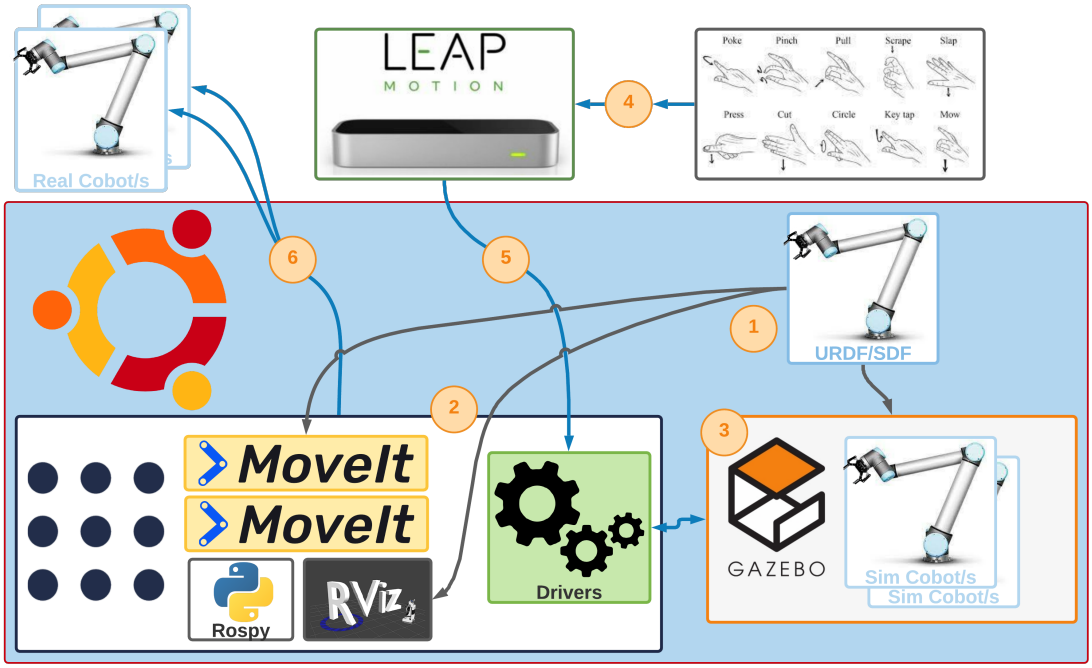


Figura 5.4: Propuesta de solución final con el paquete MoveIt! de ROS

El diseño de la solución de la Figura 5.4, el componente principal que realiza las funciones de planificador es el paquete MoveIt!. Este diseño se basa en la replicación de los componentes que el planificador necesita para su funcionamiento (el nodo principal es `move_group`). Cada replicación es un planificador asignado para cada cobot que están delimitados dentro de su propio espacio o *namespace*.

En la descripción del robot es únicamente necesario la descripción de un cobot y el número de robots que puede controlar simultáneamente está directamente relacionado con el número de replicaciones de planificador con diferentes *namespaces*. Los números en la imagen enumera el orden de las fases por las que pasa esta solución como en la Sección 5.3.

Las ventajas de esta propuesta son:

- La escalabilidad del sistema y del número de cobots.
- La potencia de las funciones que ofrece MoveIt! para cada robot individualmente.
- Una sencilla configuración, la mayoría se crea de forma automática con su *MoveIt Setup Assistant*.

- Flexibilidad a la hora de adaptarse o la incorporación de futuros elementos que puedan necesitar las funcionalidades de MoveIt!, como pueden ser dispositivos que se utilicen para el desarrollo de trabajo de *visión por computador*, *deep learning*, *navegación*, etc.

- Las trayectorias cartesianas están incluidas como una función de MoveIt!, necesario para la incorporación de Leap Motion al sistema.

- Permite el control de diferentes modelos y marcas de cobots simultáneamente.

- Hay una comunidad detrás del paquete de MoveIt! muy activa que da soporte a sus usuarios.

Las desventajas son:

- Poca documentación con respecto a la correcta configuración para el control de varios robots simultáneamente.

- Limitación de las funcionalidades del paquete MoveIt! a cada robot individualmente, pero no en conjunto, por ejemplo: *la funcionalidad planificar trayectorias evitando colisiones entre los robots no funcionaría adecuadamente porque tiene no conocimiento de ellos.*

- Pérdida de eficiencia conforme se escala: al escalar el sistema y aumentar el número de cobots que controla, se replica también las funcionalidades que en ese momento ocupan un recurso, pero no hacen ningún trabajo.

- Complicado realizar cambios en el código fuente de las funcionalidades del paquete de MoveIt!.

– *Sin el paquete MoveIt!*:

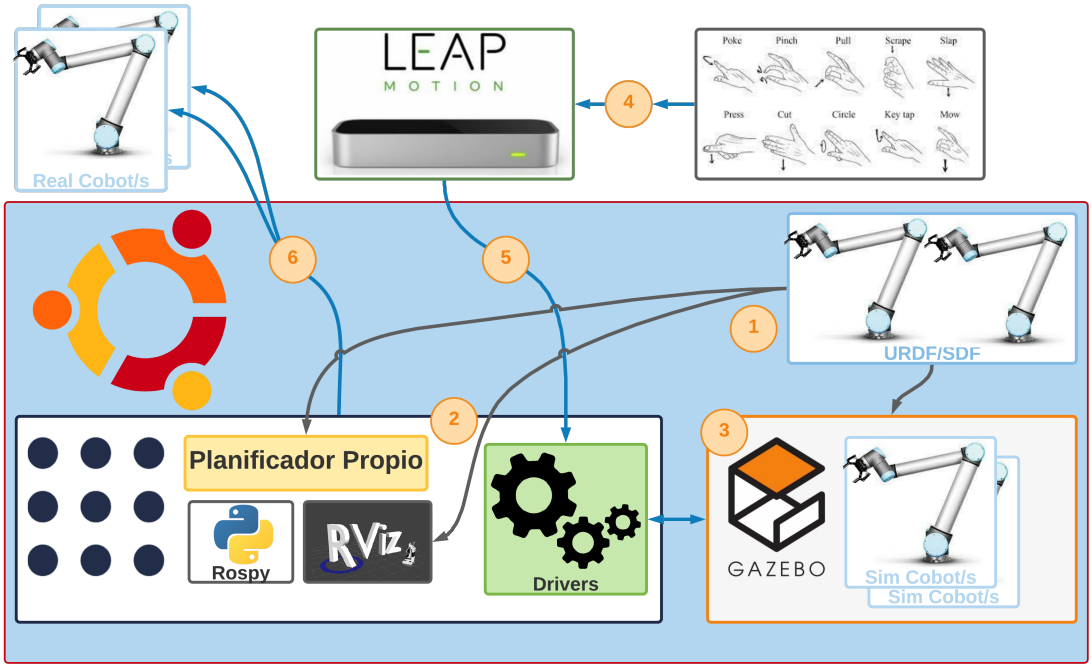


Figura 5.5: Propuesta de solución final sin el paquete de MoveIt! de ROS

En el diseño de la solución de la Figura 5.5, el componente principal que realiza las funciones de planificador debe ser implementado de cero, el planificador se comunicaría directamente con los controladores necesarios para realizar los movimientos del cobot, no es necesario que esté contenido dentro de un *namespace*. Es necesaria la descripción de todos los cobots que se quiera controlar simultáneamente en el fichero URDF que define el modelo. Los números en la imagen listan el orden de las fases por las que pasa esta solución como en la Sección 5.3.

Las ventajas de esta propuesta son:

- La escalabilidad del sistema y del número de cobots es algo más compleja que en el diseño de la Figura 5.4.
- Es muy eficiente en comparación con la solución de la Figura 5.4.
- Es sencillo realizar cambios en el código fuente implementado.
- Permite el control de diferentes modelos y marcas de cobots simultáneamente.

Las desventajas son:

- Poca adaptabilidad a la integración en otro proyecto, al ser una solución a medida.

- No tiene capacidad para planificar trayectorias evitando colisiones con otro cobots ni consigo mismo porque son funcionalidades proporcionadas por MoveIt!.
- Hay que implementar la funcionalidad para realizar movimientos cartesianos, es un requisito dada la futura incorporación de Leap Motion al sistema.
- La configuración puede ser tediosa (controladores, topics, tratamiento de los mensajes, interacción entre lo que se ha creado con lo creado por terceros, etc.), es necesario cierta familiaridad con el entorno.

– *Con el paquete MoveIt! y varios robots en el URDF*:

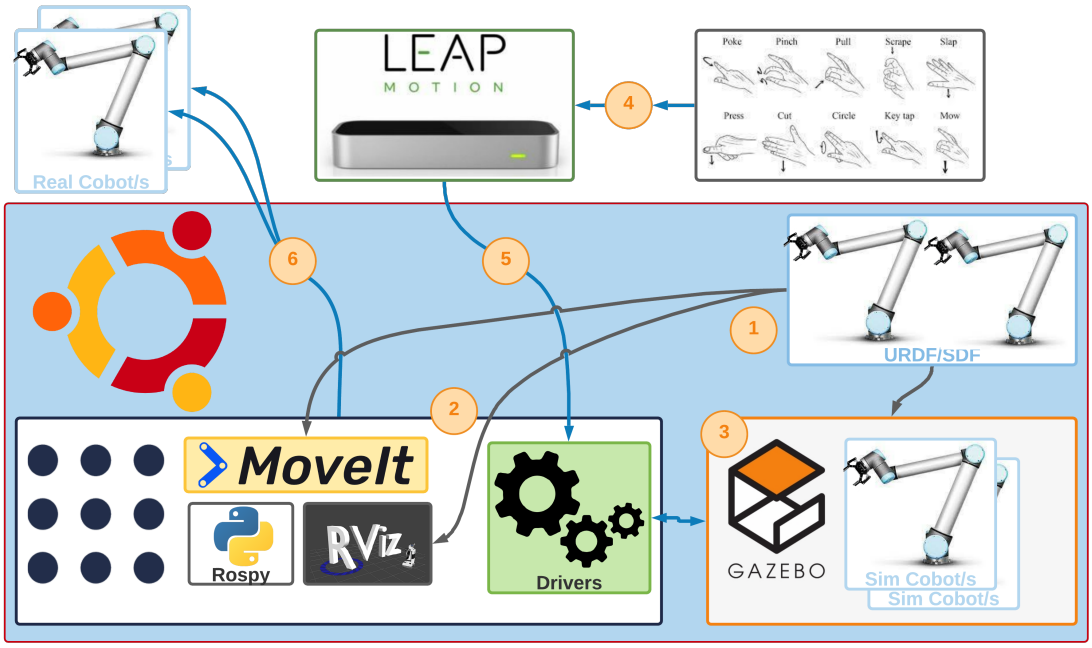


Figura 5.6: Propuesta de solución final añadiendo varios robots en el URDF y con el paquete MoveIt! de ROS

En el diseño de la solución de la Figura 5.6, el componente principal que realiza las funciones de planificador es el paquete MoveIt!, se basa en la adición de robots en el fichero URDF y el planificador de MoveIt! lo reconocería como un único robot, el cual obtendría los valores articulares con el planificador de movimiento de sus *end effectors* y se enviaría esos valores obtenidos como una única trayectoria a los controladores para que ejecute el movimiento deseado.

Las ventajas de esta propuesta son:

- La potencia de las funciones que ofrece MoveIt! para cada robot como en su conjunto.
- Una sencilla configuración, la mayoría se crea de forma automática con su *MoveIt Setup Assistant*.
- Flexibilidad a la hora de adaptarse o la incorporación de futuros elementos que puedan necesitar las funcionalidades de MoveIt!, como pueden ser dispositivos que se utilicen para desarrollo de trabajo de *visión por computador*, *deep learning*, *navegación*, etc.
- La funcionalidad de movimientos cartesianos está incluida como una función de MoveIt!.

Las desventajas son:

- Coste en tiempo a la hora de obtener la planificación de las trayectorias, ya que previamente debe obtener las trayectorias de cada *end effector* para después combinarlas si fuese el movimiento de un único robot.
- El único grupo que gestionaría los movimientos de los cobots habría que reconfigurarlo cada que se añadiese un robot al sistema, lo que afecta a la escalabilidad del sistema.
- No permite el control de diferentes modelos y marcas de cobots simultáneamente, ya que podrían necesitar diferentes controladores.

Esta solución tiene una ventaja muy importante y es que es consciente del movimiento de todos los robots y durante la planificación de las trayectorias puede evitarlos para llegar a la posición deseada.

Pero la arquitectura del diseño del paquete de MoveIt! no permite el control de varios robots con diferentes controladores. Si se quiere realizar el control de varios robots mediante este diseño, se podría conseguirlo mediante la secuenciación de los movimientos de cada cobot y no la simultaneidad que es lo que se busca. En caso de forzarlo (lanzando dos scripts que envían instrucciones para que planifique las trayectorias deseadas para cada controlador) el nodo de MoveIt!, `move_group`, envía un mensaje diciendo que la instrucción ha sido `PREEMTED`, es decir que el *Scheduler* del sistema lo ha retrasado como trabajo futuro.

Para sobrepasar este problema, habría que hacerle pensar al nodo `move_group` que maneja un único controlador. Para conseguir esto, hay que crear mínimo tres

grupos de manipuladores durante la configuración con el asistente de MoveIt!, uno para cada brazo junto a su *end_effector* y otro grupo que contiene a los dos grupos creados previamente. Cuando se tiene esta configuración, habría que realizar el plan de trayectorias de cada brazo sin ejecutarlo y añadir los valores de las articulaciones de cada cobot para esa trayectoria adecuadamente en el grupo que contiene a ambos cobots para ejecutarlo posteriormente.

No se ha llegado a comprobar su funcionamiento en detalle por la falta de conocimientos del estudiante al realizar las pruebas en ese momento, pero esta solución tiene un gran cuello de botella en el nodo de MoveIt!. Conforme aumenta el número de cobots en el sistema, aunque ROS permite que la mayoría de trabajo se realice en otros ordenadores y tuviese un ordenador dedicado a MoveIt!, habría un retraso importante en algún momento, por ejemplo: *si se tiene 10 cobots, hay que obtener la trayectoria de cada cobot, el nodo de MoveIt! tiene que agruparlos y después tiene que comprobar que no hay colisión y ejecutarlo (sin contar el coste de transmisión de los datos).* Durante el envío de las trayectorias hasta que se ejecuta, los robots están parados y también se encontraría con el problema de sí se ejecuta todos los movimientos o ninguno, en caso de detectar colisión en uno de los cobots, pero el resto de las trayectorias son válidas, todos deberán parar.

Por las razones anteriores, esta solución se desechó aunque cumpliese los requisitos del trabajo no es práctico para soluciones generales pero sí en casos concretos (robots humanoides).

Cabe añadir que durante la búsqueda de la solución al problema durante el desarrollo de los diseños, dada la poca información documentada sobre el problema y por la falta de conocimiento del estudiante sobre entorno de ROS y sus herramientas, pueden haber sido omitidas posibles soluciones. Las dos soluciones que se han seleccionado e implementado se describen en detalle en los siguientes capítulos.

Capítulo 6

Soluciones desarrolladas, arquitecturas y detalles de implementación

En este capítulo se va a entrar en detalle de las dos soluciones propuestas de la Sección 5.4. La razón por la que se presentan ambas soluciones es debido a los problemas que se ha tenido durante la configuración de la solución con el paquete de MoveIt!, por lo que se había desarrollado previamente la solución alternativa sin el paquete para cumplir con los requisitos de este Trabajo Fin de Grado. Fue a lo largo del desarrollo final durante un problema surgido al realizar las pruebas en el robot *Campero*, en donde se detectó que uno de los nodos publicaba información en el topic `\joint_states` y que estaba interfiriendo con la información que se recogía para realizar correctamente el cálculo de las trayectorias.

Una vez solucionado ese detalle en la configuración, la solución con el paquete de MoveIt! funciona correctamente y debido a las ventajas y potencial que esta solución provee se ha decidido explicar ambas soluciones. Aunque la solución sin el paquete de MoveIt! no provee de tantas ventajas y seguramente para la realización de trabajos futuros no se tome como referencia, demuestra la compresión y dominio del estudiante del trabajo realizado.

Para explicar ambas soluciones, se ha decidido dividirlo en dos secciones, y cada uno de ellos estará dividido en cinco fases, que corresponden con las primeras cinco etapas de la Figura 5.4 y la Figura 5.5. Además, en cada una de las secciones se comenzará explicando las fases de configuración para un único cobot y después se indicará las modificaciones que hay que realizar sobre lo ya desarrollado para dos o más cobots. También se añadirá los detalles de implementación correspondientes porque la mayor parte del trabajo desarrollado se basa en la integración, configuración y adaptación de paquetes de terceros.

Antes de comenzar a desarrollar las soluciones, se recomienda leer la configuración del entorno (ver Anexo E), ya que previamente se han instalado todos los paquetes de terceros de ROS mencionado en la Subsección 4.1.3 del Capítulo 4. Una vez realizado el *setup* de proyecto correctamente, la estructura del directorio de trabajo deberá ser el siguiente (ver Código Fuente 6.1) tras la instalación de todos los repositorios:

```
miguel@Omen:~/tfg_multirobot/src$ ls
CMakeLists.txt        geometry                  ros_control
gazebo-pkgs           leap_motion               roslint
gazebo_ros_pkgs       object_recognition_msgs   universal_robot
general-message-pkgs  robotiq_2finger_grippers  ur_modern_driver
```

Código Fuente 6.1: Directorio raíz `src` con todos los paquetes instalados

Los directorios están organizados de esta manera intencionalmente, tomando como ejemplo la estructura mostrada en la Figura 6.1, con los siguientes objetivos:

- Tener todos los paquetes originales localizados en un directorio común, permite rápidamente obtener y buscar los ficheros y da una idea de qué trata.
- Se creará un directorio llamado `tfg_project`, el cual hará una función de *workspace*, es decir, será como el directorio raíz de las soluciones a implementar, y se creará para organizar las diferentes soluciones otro directorio, por ejemplo: `one_arm_moveit` y es este directorio el que contendrá los paquetes de ROS que forman la solución, estos paquetes utilizarán los recursos del directorio raíz `src`.
- Los paquetes que forman la solución, replicarán únicamente los ficheros o directorios que necesiten de los recursos originales para su posterior modificación, lo que permite la reutilización de los recursos originales sin interferir con otras soluciones.
- También permite realizar pruebas rápidamente sin tener que reproducir todo el *setup* previamente realizado, únicamente replicar la solución, modificarlo y realizar las pruebas.
- Facilita la incorporación de paquetes al sistema sin que afecte a la configuraciones de las soluciones propuestas.
- Dada la estructura de los directorios impuestos por ROS, es fácil trasladar el proyecto de un sistema a otro, simplemente hay que copiar el directorio `src` y compilarlo.
- Cada directorio solución, por ejemplo `one_arm_moveit`, contiene hasta cinco paquetes:

- one_arm_moveit_*config*: Contiene toda la configuración necesaria del paquete MoveIt!, generado por su *Setup Assistant*.
- one_arm_moveit_*description*: Contiene los ficheros URDFs del modelado del cobot.
- one_arm_moveit_*gazebo*: Contiene toda la configuración del simulador Gazebo y controladores.
- one_arm_moveit_*leap_motion*: Contiene los scripts modificados basados en el paquete de Leap Motion para el control del cobot mediante el dispositivo externo de Leap Motion.
- one_arm_moveit_*manipulator*: Contiene los scripts creados para realizar el *pick & place* de forma automatizada.

Figura 6.1: Estructura de los directorios y organización

Teniendo esta información en mente como base y guía, se procede a explicar como se han desarrollado las soluciones propuestas. Si se quiere reproducir o inspeccionar en detalle las distintas implementaciones de las soluciones propuestas, toda la información necesaria está en una cuenta de *Github* creada con este propósito (ver Anexo F).

6.1. Solución desarrollada *con* el paquete de MoveIt!

En esta sección se presenta la solución utilizando el paquete de MoveIt!, para el control simultáneo de uno, dos o más cobots.

6.1.1. Desarrollo para un único cobot

Antes de comenzar, esta solución está localizada dentro del directorio `tfg_project` y en el directorio de la solución `one_arm_moveit` mostrada en la Figura 6.1. Se estructura el desarrollo de la solución para único cobot en las siguientes cinco fases:

– *Fase 1: Configuración del URDF para un único cobot*

El fichero URDF (United Robotics Description Format) modela el cobot utilizando el formato XML[1] el cual será utilizado por las diferentes aplicaciones que ROS necesite, pero principalmente para realizar una simulación del robot modelado.

El fichero está construido en forma de árbol, en donde hay tres etiquetas principales: <*robot*>, <*link*> y <*joint*>. Para explicarlo bien, se puede tomar como referencia el brazo del cuerpo humano. Si lo que se quiere modelar es el brazo de una persona, la etiqueta <*robot*> representaría al brazo en su conjunto. Este brazo está compuesto de varios huesos (húmero, cúbito y radio) que son representados por las etiquetas <*link*> y por una articulación que une esos huesos (codo) que es representado por la etiqueta <*joint*>. Además como en los huesos, estas etiquetas pueden ir con información adicional contenida en ellas que den información del tamaño, geometría, inercia, orientación etc. Finalmente, el modelado de un robot se puede unir a otro modelo y formar uno más complejo, que podría ser representado con la adición de la mano al brazo, con la muñeca como articulación que conectan ambos. Hay que tener en cuenta que las etiquetas <*joint*> conecta las etiquetas <*link*> a través de una relación padre-hijo.

Dicho esto, se realiza una representación de los componentes del robot que se aprecia en la Figura 6.2, estos componentes son la pinza y el brazo manipulador UR10 (ver Anexo A) y el código completo se encuentra en Github (ver Anexo F).

[1] XML (eXtensible Markup Language): lenguaje de marcado similar al HTML, de propósito general. El XML no está predefinido, por lo que se pueden definir etiquetas propias.

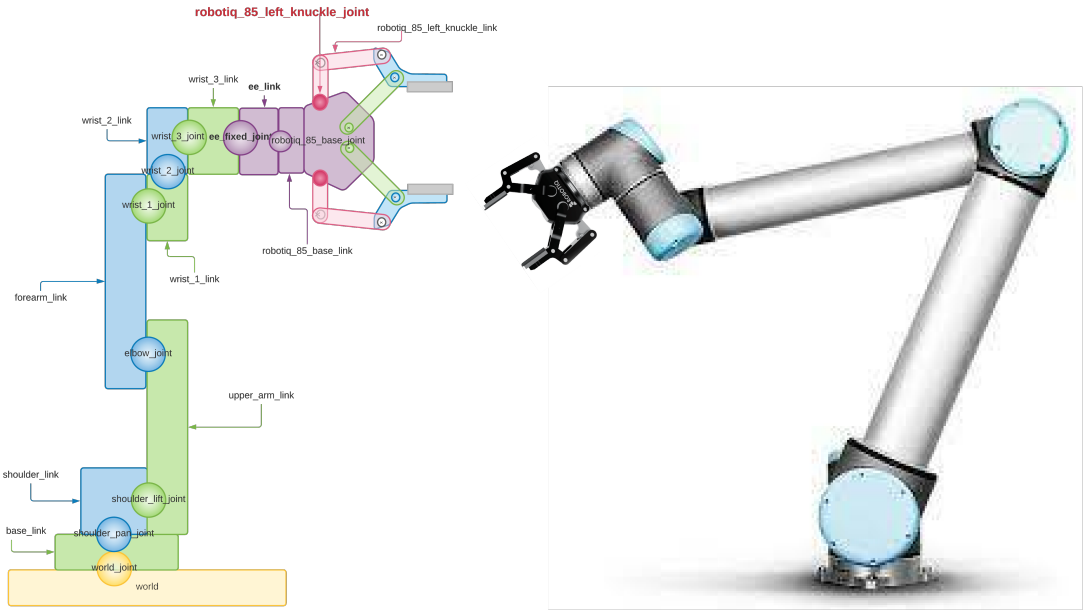


Figura 6.2: Fichero `ur10_robot.urdf.xacro`

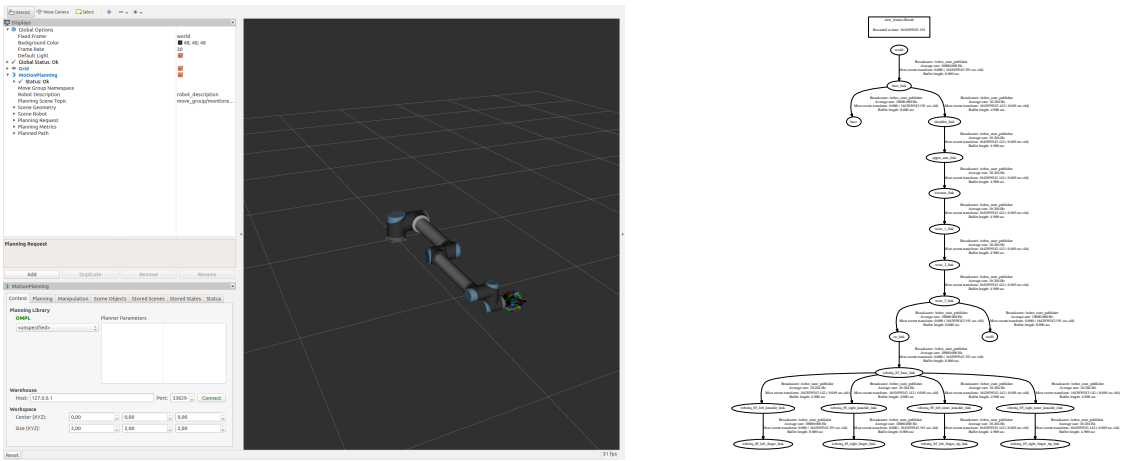
En el Código Fuente 6.2, se encuentra en el paquete de `one_arm_moveit_description` en su directorio `urdf` (basado en el contenido del fichero `ur10_robot.urdf.xacro` del paquete `universal_robot`), se puede ver cómo se conecta el componente del brazo `ur10_robot` con el link `world`, esto puede apreciarse en la Figura 6.2 representando `world` (color amarillo) y la base del brazo del UR10 `base_link` (color verde) situado justo encima, además el joint `world_joint` es la esfera de color amarillo situado entre ambos links. De la misma manera se tiene el componente de la pinza `robotiq_85_gripper`, está conectado al brazo del UR10 (`ur10_robot`), esto se aprecia en la Figura 6.2 en donde la esfera que representa el joint `robotiq_85_base_joint` que une ambos componentes (color morado), uniendo el link `robotiq_85_base_link` de la pinza con el link `ee_link` del brazo de UR10.

```
<link name="world" />

<joint name="world_joint" type="fixed">
  <parent link="world" />
  <child link = "base_link" />
  <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
</joint>

<!-- arm -->
<xacro:ur10_robot prefix="" joint_limited="false"
  transmission_hw_interface="$(arg transmission_hw_interface)"
/>

<!-- gripper -->
<xacro:robotiq_85_gripper prefix="" parent="ee_link" >
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:robotiq_85_gripper>
```

Código Fuente 6.2: Parte del fichero `ur10_robot.urdf.xacro`

– *Fase 2: Configuración de MoveIt! y visualización por Rviz*

La configuración del paquete de MoveIt! es bastante simple, una vez que se sabe cómo hay que hacerlo, pero antes de eso hay que crear el directorio `one_arm_moveit_config` en donde se almacenará lo generado de la configuración realizada como un paquete de ROS, como se aprecia en la Figura 6.1. Una vez preparado el directorio junto al fichero URDF que modela el cobot en la *Fase 1* de esta Sección 6.1, se procede a realizar la configuración del paquete MoveIt! lanzando su *Setup Assistant*, todo el proceso está en el Anexo B.



(a) Visualizacion en Rviz (b) Árbol de las transformadas

Figura 6.3: Fase 2: Visualización en Rviz y transformadas al lanzar MoveIt!

Al lanzar MoveIt! con el comando:

```
roslaunch one_arm_moveit_config demo.launch
```

Código Fuente 6.3: Comando para lanzar una demo del paquete MoveIt!

aparecerá Rviz (Figura 6.3a) en el cual se puede apreciar el brazo del cobot UR10 junto a la pinza de Robotiq. Además, para confirmar que se ha generado correctamente el árbol con las transformadas del modelo del robot Figura 6.3b y finalmente se va a utilizar la herramienta de ROS `rqt_grah` para ver las interacciones entre los nodos y los topics (Figura 6.4) y que se procede a explicar en más detalle (para ver las Figuras con más detalle ver Anexo C, Secciones C.1.1, C.1.3 y C.1.2).

En la Figura 6.3a, se visualiza en Rviz el modelado del robot que se realizó modificando el fichero URDF, Rviz es una herramienta muy potente que permite visualizar y representar la información del sistema de ROS que está en ejecución, se puede ver las transformadas, links, e incluso comunicarse con los servicios, en este caso con el planificador de MoveIt! mediante el plugin de MoveIt! *Motion Planning*.

Después en la Figura 6.3b, es la representación el árbol de las transformadas, muy útil para ver rápidamente si hay defectos en las interconexiones y el modelado del robot, se obtiene mediante la función `view_frames` del paquete *tf* de ROS.

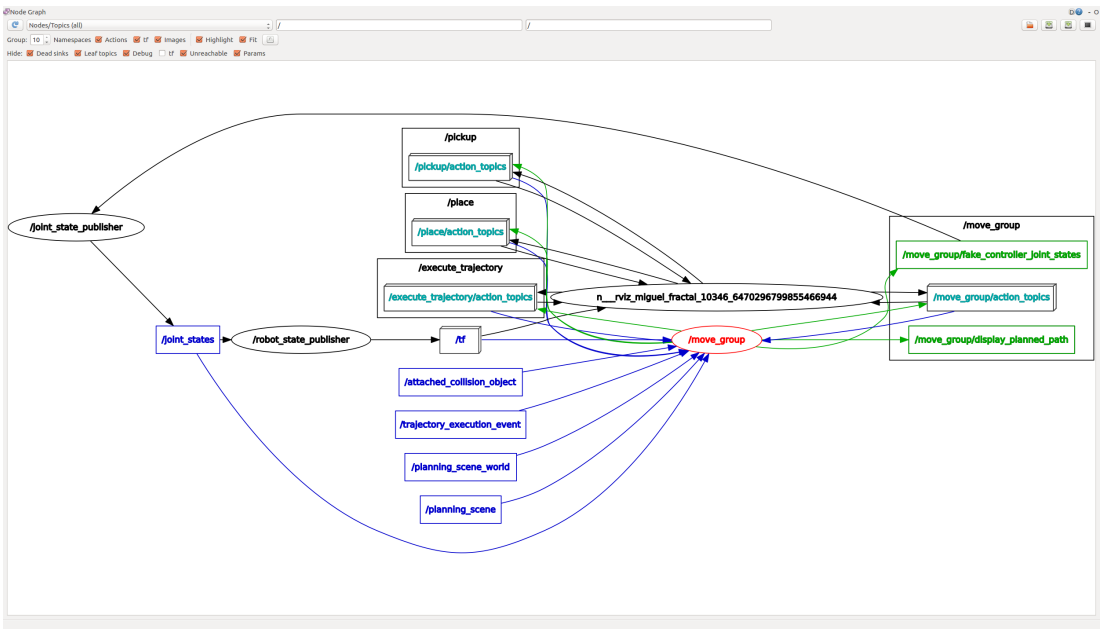


Figura 6.4: Fase 2: Visualización de la arquitectura en nodos y topics

Finalmente en la Figura 6.4, es una gráfica obtenida con la herramienta `rqt_graph` de ROS. Esta gráfica representa la arquitectura del sistema lanzando solamente MoveIt!, formado por cuatro nodos que son */joint_state_publisher*, */robot_state_publisher*, *rviz* y */move_group* en donde el nodo principal es el nodo

`move_group` (en color rojo en la Figura 6.4). Se va a explicar las funciones que realiza cada nodo:

- *joint_state_publisher*: Permite la obtención de los valores de todos los *joints* de un robot que no sean estáticos en el topic `/joint_states`. En la Figura 6.4 se ve que está suscrito al topic `/move_group/fake_controller_joint_states`, esto es debido a que el nodo *move_group* simula los valores de los joints de un cobot real. Hay que tener cuidado con no mezclar estos valores junto a los valores del cobot real al añadir este al sistema o introducirá ruido[2] y puede generar movimientos indeseados.

- *robot_state_publisher*: Obtiene la información del valor actual de los dinámicos del robot tras realizar un Forward Kinematic[3], la información necesaria para realizar las transformadas se recogen del parámetro del servidor `robot_description` y el resultado es enviado al topic `/tf`.

- *Rviz*: En el visualizador de Rviz, no se va a entrar en detalle porque únicamente se utiliza para la depuración del sistema y no es un elemento fundamental de la arquitectura, pero se ve que está suscrito y puede publicar en ciertas *acciones* del nodo *move_group*, esto es por los plugins del paquete MoveIt! que permite controlar el cobot desde la interfaz de Rviz y obtiene la información para representar el movimiento del cobot del topic `/tf`.

- *move_group*: Es el nodo principal de esta arquitectura e integra todas las funcionalidades del paquete MoveIt!, proveyendo de un conjunto de servicios y acciones que permite su comunicación con el cobot. En la figura de la Subsección C.1.4 del Anexo C se puede apreciar todos los servicios y acciones inactivos que el nodo tiene.

  Añadir que si no se lanza Rviz, no habría comunicación con los servicios o acciones `/pickup`, `/execute_trajectory` y `/place`. En la Figura 6.5 se muestra la interacción mínima al lanzar el nodo de `move_group` correctamente, es decir la arquitectura base del paquete MoveIt!.

[2]Ruido: en este caso, se refiere a la interferencia o la introducción de datos no deseados que hagan que el sistema pueda no funcionar correctamente o que conlleve una disminución de sus prestaciones.

[3]Foward kinematic o Cinemática directa: Calcula la posición del *end-effector* partiendo de los valores de los *joints* como parámetros.

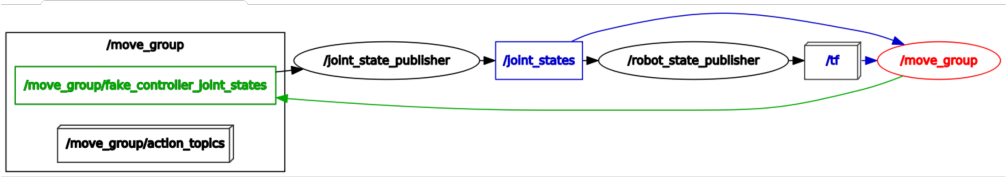


Figura 6.5: Fase 2: Visualización de la arquitectura en nodos y topics sin Rviz

– *Fase 3: Simulación en Gazebo y creación del script pick & place*

Esta fase tiene se divide en tres etapas:

- *Configuración de Gazebo y controladores*

  Lo primero que hay que hacer en esta fase es configurar Gazebo y los controladores para que pueda simular adecuadamente los movimientos del cobot. Se crea el paquete `one_arm_moveit_gazebo`, que contendrá toda la configuración relacionada con Gazebo, entre ellos los controladores. Una vez creada el paquete, hay que configurar los controladores que están almacenados en el directorio `controller`, aunque todos los controladores pueden estar definidos en un único fichero por claridad se ha distribuido en tres ficheros.

  Los controladores se definen en ficheros con extensión `yaml`[4], para definir estos controladores hay que darles un nombre y definir el tipo del controlador, los joints dinámicos que se quieren controlar, las restricciones que tiene, el ratio de publicación y otras opciones.

  A continuación se presenta el contenido de los ficheros de configuración de los controladores, todos estos controladores, en general, siguen la estructura mencionada. La definición de los controladores pueden ser contenidas en un único fichero, lo importante es que en Gazebo los cargue correctamente, se procede a explicar brevemente estos controladores:

  - Fichero `arm_controller_ur10.yaml`: En este fichero (Código Fuente 6.4) se define el controlador para el cobot UR10, aquí se define el nombre del controlador `arm_controller`, el tipo de controlador `position_controllers/JointTrajectoryController`, lo que implica la definición del tipo de mensajes y el formateo adecuado de la información necesaria para comunicarse con éste. Después está

[4]YAML: Es un fichero para la transmisión de datos serializados de una manera legible, ampliamente usado como ficheros de configuración.

el campo `joints`, que es donde se indica qué joints del cobot forma parte del controlador, todos estos joints son dinámicos. El resto de campos no se han tocado, pero hay que mantener la consistencia en cómo se nombran.

```
arm_controller:
  type: position_controllers/JointTrajectoryController
  joints:
     - shoulder_pan_joint
     - shoulder_lift_joint
     - elbow_joint
     - wrist_1_joint
     - wrist_2_joint
     - wrist_3_joint
  constraints:
      goal_time: 0.6
      stopped_velocity_tolerance: 0.05
      shoulder_pan_joint: {trajectory: 0.1, goal: 0.1}
      shoulder_lift_joint: {trajectory: 0.1, goal: 0.1}
      elbow_joint: {trajectory: 0.1, goal: 0.1}
      wrist_1_joint: {trajectory: 0.1, goal: 0.1}
      wrist_2_joint: {trajectory: 0.1, goal: 0.1}
      wrist_3_joint: {trajectory: 0.1, goal: 0.1}
  stop_trajectory_duration: 0.5
  state_publish_rate:  25
  action_monitor_rate: 10
```

Código Fuente 6.4: Definición del controlador UR10

- Fichero `gripper_controller_robotiq.yaml`: En este fichero (Código Fuente 6.5) se define el controlador para la pinza de Robotiq, aquí se define el nombre del controlador `gripper`, el tipo de controlador `position_controllers/JointTrajectoryController` que define el tipo de mensajes y la información necesaria para comunicarse con éste. Después está el campo `joints`, que es donde se indica qué joints del cobot forma parte del controlador en este caso un único joint `robotiq_85_left_knuckle_joint` porque el resto de joints del controlador imitan los movimientos de este. El resto de campos no se han tocado, pero hay que mantener la consistencia en cómo se nombran como en el caso anterior.

```
gripper:
  type: position_controllers/JointTrajectoryController
  joints:
     - robotiq_85_left_knuckle_joint
  constraints:
      goal_time: 0.6
      stopped_velocity_tolerance: 0.05
      robotiq_85_left_knuckle_joint: {trajectory: 0.1, goal: 0.1}
  stop_trajectory_duration: 0.5
  state_publish_rate:  25
  action_monitor_rate: 10
```

Código Fuente 6.5: Definición del controlador la pinza de Robotiq

- Fichero `joint_state_controller.yaml`: Lo que define este fichero (Código Fuente 6.6) realmente no es un controlador como tal, su función es la de una interfaz que traduce la información de los joints que viene del

cobot real y lo traduce a mensajes de tipo `JointState` para después publicarlo. Es fundamental para el correcto funcionamiento, tanto en simulación como con el robot real, forma parte del paquete de ROS *ros_control*.

```
joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
```

Código Fuente 6.6: Controlador de *ros_control* (definido por defecto)

Ahora hay que definir el mundo del robot, es decir el fichero de extensión `world` que define el entorno virtual de la simulación que será necesario para que quede funcional el *pick & place*, no se va a entrar en detalle porque no es complicado y la solución puede funcionar sin esta implementación, en Github (ver Anexo F), en el directorio `one_arm_moveit_gazebo/world/multiarm_bot.world` se puede ver la implementación y en el *setup* se explica con más detalle.

Con los controladores correctamente configurados, se procede a modificar el fichero `ur10.launch` del paquete `one_arm_moveit_gazebo` (basado en el contenido del fichero `ur10.launch` del paquete `ur_gazebo` del directorio `universal_robot`) para que cargue los controladores definidos previamente, para ello hay que añadir al final del fichero el siguiente el trozo de código (Código Fuente 6.7). El trozo de código lo que hace es cargar el contenido de los ficheros `yaml` en los parámetros del servidor y para lanzarlos pasa el *nombre del controlador* definido como argumento en los nodos (*arm_controller* y *gripper*).

```
<!-- start this controller -->
 <rosparam file="$(find one_arm_moveit_gazebo)/controller/
 arm_controller_ur10.yaml" command="load"/>
 <node name="arm_controller_spawner" pkg="controller_manager"
 type="controller_manager" args="spawn arm_controller" respawn="false"
 output="screen"/>

 <!-- robotiq_85_gripper controller -->
 <rosparam file="$(find one_arm_moveit_gazebo)/controller/
 gripper_controller_robotiq.yaml" command="load"/>
 <node name="gripper_controller_spawner" pkg="controller_manager" type="spawner"
 args="gripper" />
```

Código Fuente 6.7: Trozo de código que carga los controladores al fichero `ur10.launch`

Ahora mismo, el sistema tiene Gazebo y MoveIt! configurado, por tanto, se procede a simularlo y ver si todo funciona correctamente, en dos terminales diferentes se lanza uno de los comandos de que se muestra en Código Fuente 6.8.

```
roslaunch one_arm_moveit_gazebo ur10_joint_limited.launch
roslaunch one_arm_moveit_config demo.launch
```

Código Fuente 6.8: Comando para lanzar la demo del paquete MoveIt! y Gazebo

Al utilizar la herramienta `rqt_graph` para comprobar si ha cargado correctamente los controladores (Figura 6.6, para la Figura en detalle, ver la Subsección C.1.5 del Anexo C), se ve que Gazebo carga correctamente los controladores `arm_controller` y `gripper` pero no hay comunicación entre el nodo `move_group` y los controladores del nodo `gazebo`, el único punto en común es el topic `/joint_states`. Esto implica que si se planifica y ejecuta trayectorias con el plugin *Motion Planning* de MoveIt!, no sean representados en Gazebo pero sí en Rviz. Este no es el resultado que se busca, por ello se procede a modificar la configuración de MoveIt! para que pueda comunicarse con los controladores cargados en Gazebo.

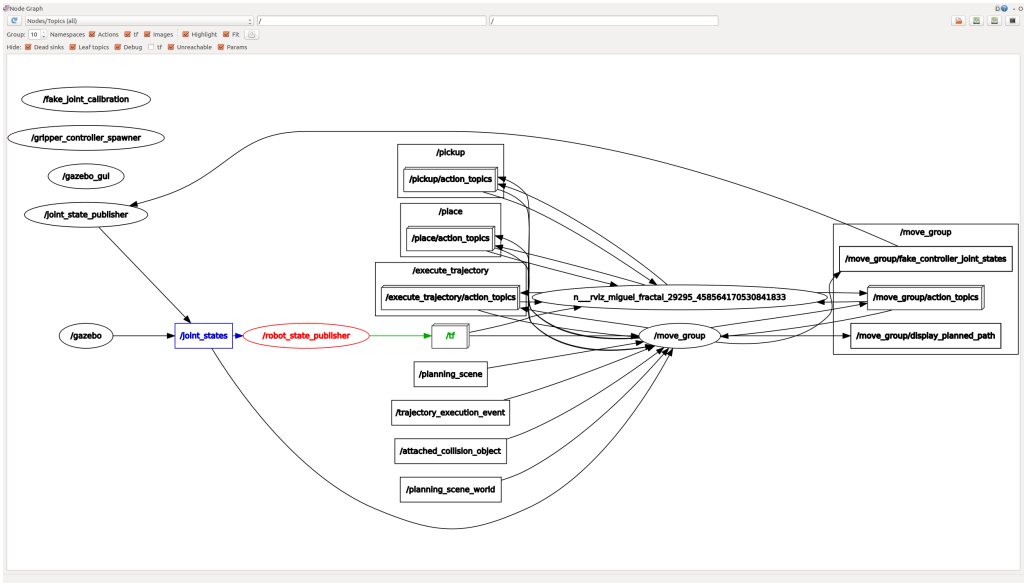


Figura 6.6: Fase 3: No hay comunicación entre MoveIt! y los controladores de Gazebo

- *Conexión entre Gazebo y Moveit!*

  Como se ve en la Figura 6.6, MoveIt! no está conectado con los controladores de Gazebo, en esta fase se procede a realizar los cambios necesarios para que sea posible. Primero se procede a crear un nuevo paquete `one_arm_moveit_manipulator` que contendrá, además de las modificaciones de la configuración original de MoveIt! realizada en la *Fase 1* de esta solución, los scripts para realizar el *pick & place*.

  Una vez creado el nuevo paquete, se crea el directorio `config` en donde se crearán los dos ficheros `controllers.yaml` (Código Fuente 6.9) y

`joint_names.yaml` (Código Fuente 6.10) que permitirán la interacción con los controladores de Gazebo, se va a realizar una breve descripción de los puntos importantes en el contenido de estos ficheros:

- Fichero `controllers.yaml`: La definición de los controladores de ROS en MoveIt! es similar al que se realizó para Gazebo, el nombre de los controladores debe coincidir con los nombres de los controladores descritos en Gazebo, se define el servidor de acciones `follow_joint_trajectory`, el tipo debe ser `FollowJointTrajectory` para que el tipo mensaje enviado entre ellos sean compatibles y finalmente los nombres de los joints involucrados deben ser idénticos también.

```
controller_list:
  - name: "arm_controller"
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - shoulder_pan_joint
      - shoulder_lift_joint
      - elbow_joint
      - wrist_1_joint
      - wrist_2_joint
      - wrist_3_joint
  - name: "gripper"
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - robotiq_85_left_knuckle_joint
```

Código Fuente 6.9: Definición de los controladores de moveIt!

- Fichero `joint_names.yaml`: Este fichero define el nombre de los joints del controlador del cobot, se almacenará como parámetro del servidor y será utilizado como parte de la configuración de MoveIt!.

```
controller_joint_names: [shoulder_pan_joint, shoulder_lift_joint,
elbow_joint, wrist_1_joint, wrist_2_joint, wrist_3_joint]
```

Código Fuente 6.10: Define el nombre de los joints del brazo manipulador (UR10)

Lo único que falta para que Gazebo y MoveIt! se comunique correctamente es modificar el fichero `launch`, se crea un nuevo fichero con extensión `launch` llamado `one_arm_moveit_execution.launch`, es similar al fichero `demo.launch` con la diferencia de que el objetivo deja de ser simular que hay un robot (`move_group/fake_controller_joint_states`) y se configura para la comunicación con un robot real, en este caso Gazebo es el que simula el robot, pero MoveIt! no es consciente de eso y lo trata como un robot real.

El contenido del fichero (Código Fuente 6.11) es algo largo, pero es importante describirlo bien. Se comienza cargando el `controller_joint_names` al servidor de parámetros y posteriormente el modelado del robot (URDF) al incluir `planning_context.launch`. Aquí es cuando se define que el topic de origen para obtener información del estado del robot será el topic `/joint_states` como si fuese un robot real por ello se elimina el nodo `joint_state_publisher` que había previamente para que no de interferencias, posteriormente se carga el nodo `move_group`, y el punto más importante de para la comunicación entre Gazebo y MoveIt!, hace un `remap` del topic `/follow_joint_trajectory` que es el servidor de acción definido en los controladores de MoveIt! (Código Fuente 6.9) por `/arm_controller/follow_joint_trajectory` que es topic del controlador en Gazebo y opcionalmente se puede añadir el nodo Rviz.

```
<rosparam command="load" file="$(find one_arm_moveit_manipulator)/config
/joint_names.yaml"/>

<include file="$(find one_arm_moveit_config)/launch/planning_context.launch">
  <arg name="load_robot_description" value="true"/>
</include>

<include file="$(find one_arm_moveit_manipulator)/launch/move_group.launch">
  <arg name="debug" default="$(arg debug)" />
  <arg name="publish_monitored_planning_scene" value="true"/>
  <!--arg name="info" value="true"/-->
</include>

<!-- Remap follow_joint_trajectory -->
<remap from="/follow_joint_trajectory"
to="/arm_controller/follow_joint_trajectory"/>

<include file="$(find one_arm_moveit_config)/launch/moveit_rviz.launch">
  <arg name="config" value="true"/>
  <arg name="debug" default="false"/>
</include>
```

Código Fuente 6.11: Fase 3: Nuevo fichero `launch` de MoveIt! para comunicarse con Gazebo

Pero aún queda cargar los controladores definidos para MoveIt!, para ello hay que modificar el fichero `ur10_moveit_controller_manager.launch.xml` (Código Fuente 6.12), eliminar lo que había y cargar los controladores que se definieron previamente (Código Fuente 6.9):

```
<rosparam file="$(find one_arm_moveit_manipulator)/config/controllers.yaml"/>
<param name="use_controller_manager" value="false"/>
<param name="trajectory_execution/execution_duration_monitoring" value="false"/>
<param name="moveit_controller_manager"
value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
```

Código Fuente 6.12: Fase 3: Fichero `launch` de MoveIt! que carga los nuevos controladores

Con esto los cambios realizados, se comprueba de nuevo si la comunicación entre MoveIt! con los controladores de Gazebo se han establecido correctamente, por ello se lanza de nuevo en dos terminales diferentes uno de los comandos de que se muestra en Código Fuente 6.13.

```
roslaunch one_arm_moveit_gazebo ur10_joint_limited.launch
roslaunch one_arm_moveit_manipulator one_arm_moveit_execution.launch
```

Código Fuente 6.13: Comando para lanzar MoveIt! modificado y Gazebo

Se comprueba que esta vez sí que hay comunicación entre MoveIt! y los controladores como se puede apreciar en la Figura 6.7 (para verlo en detalle la imagen, ver la Subsección C.1.6 del Anexo C), en donde el nodo `move_group` realiza publicaciones a los controladores y es Gazebo el que está a la escucha de lo que se publica para ejecutar los movimientos deseados. Estos movimientos modifican el estado actual del robot que es publicado al topic `/jont_states` y esa información es transmitida al nodo `robot_state_publisher` y al nodo `move_group`. El nodo `move_group` puede volver a calcular una nueva trayectoria con la información que le llega de los topics `/tf` y `/jont_states`.

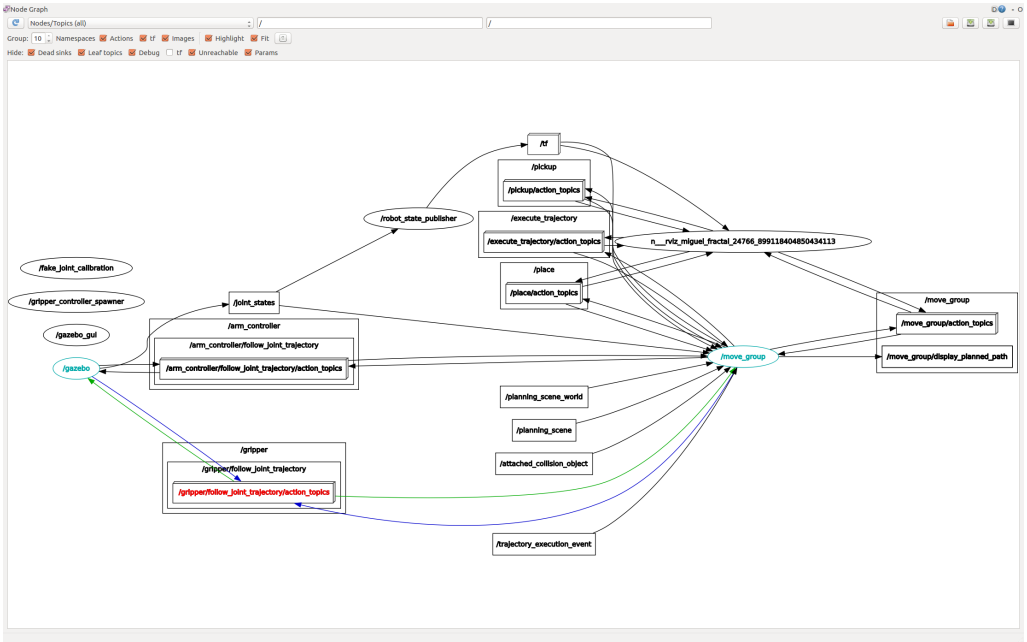
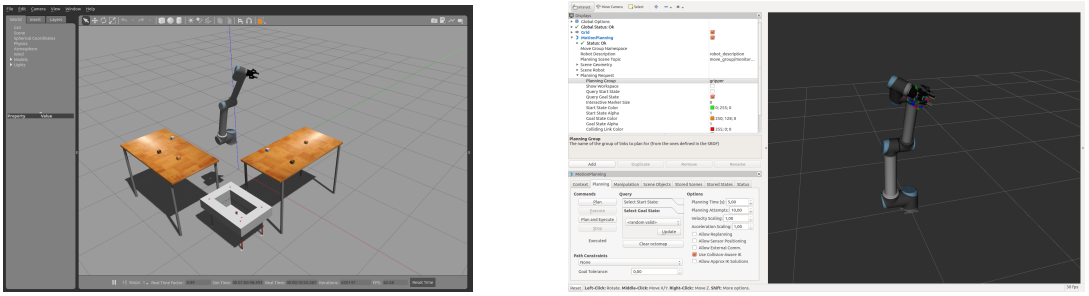


Figura 6.7: Fase 3: Comunicación entre MoveIt! y los controladores de Gazebo

- *Simulación: pick & place usado script en Python*

Para realizar el script de *pick & place* en Python, se utiliza la interfaz de Python `moveit_commander` para comunicarse con el nodo `move_group` y sus servicios y acciones. No se va a entrar en detalle porque para el control de un único robot no es muy problemático y hay una buena documentación, por ello se describirá el script en detalle para la solución con dos o más cobots, pero si se quiere ver el código completo, se encuentra en Github (ver Anexo F).



(a) Simulación en Gazebo (b) Visualización en Rviz

Figura 6.8: Fase 3: Visualización del *pick & place*

En la Figura 6.8 se muestra los resultados en Gazebo y en Rviz y en la Figura 6.9 se muestra la arquitectura del sistema cuando se añade el nodo *pick & place* al sistema (para ver la Figura C.10 en detalle, ver la Subsección C.1.7 del Anexo C).

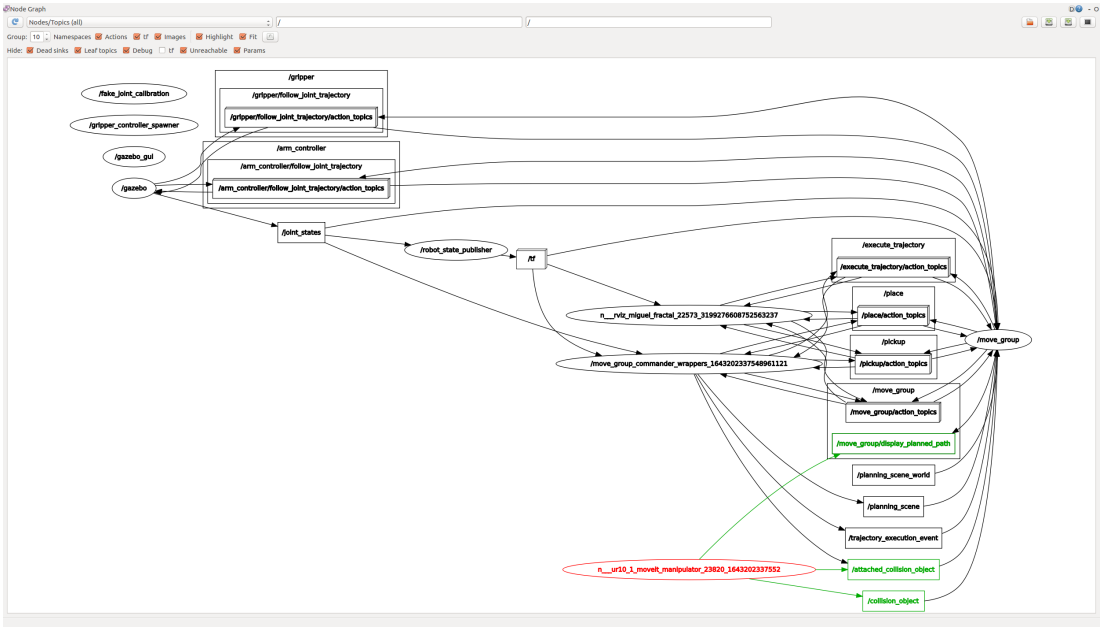


Figura 6.9: Fase 3: Comunicación entre MoveIt! y Gazebo, realizando un simple *pick & place*

– *Fase 4: Diseño de la interfaz de Leap Motion*

Una vez completado la configuración de MoveIt!, Gazebo y comprobado que realiza correctamente el *pick & place*, se procede a realizar el diseño de la interfaz de Leap Motion. Se va a describir las etapas para las que ha pasado para realizar la interfaz de Leap Motion:

- *Funcionamiento general*: Utilizando las librerías de Leap Motion se identifican los gestos y se obtiene los datos necesarios para el control de la pinza, los movimientos del cobot y las orientaciones del *end-effector*. La Figura 6.10 muestra un esquema del funcionamiento general, se creará el fichero `leap_interface.py`, utilizando la librería `Leap` de Leap Motion estará a la escucha de eventos (frames de leap motion) y en cada evento obtendrá de ellos los datos que se necesitan y los almacenará en un objeto que luego es accedido por el nodo `sender` mediante la interfaz creada para acceder a ese objeto.

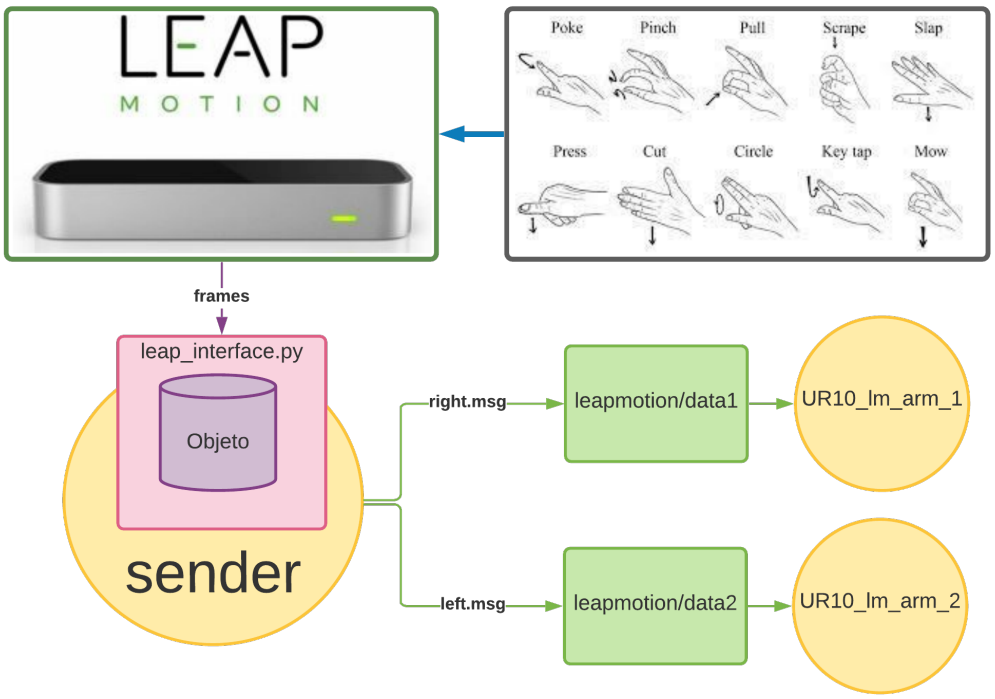


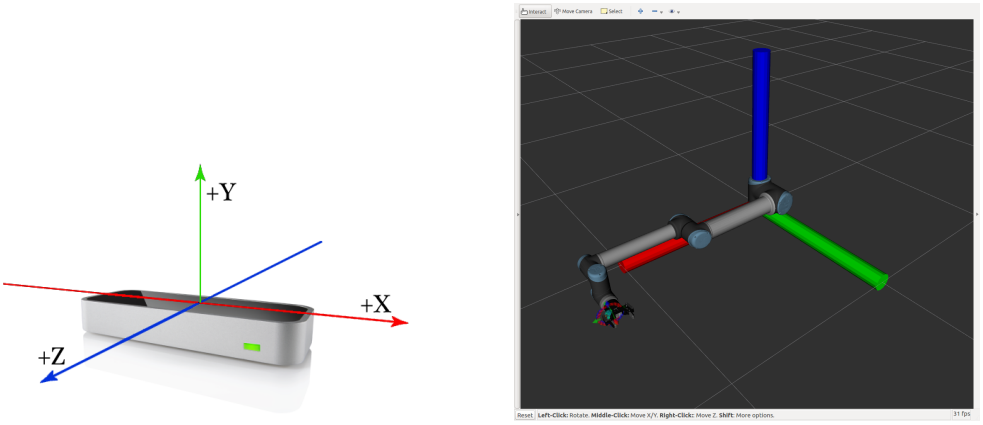
Figura 6.10: Fase 4: Funcionamiento general de Leap Motion integrado a ROS

El nodo `sender` es el que obtiene la información, los almacena adecuadamente en un mensaje y los publica por el topic `leapmotion/data1`, a este topic estará suscrito el nodo `UR10_lm_arm_1`, que con la información obtenida del topic, envía las órdenes al cobot. Finalmente, el nodo `UR10_lm_arm_1` es básicamente el script que realiza el

pick & place, pero la introducción de datos pasa a ser por lo que obtiene del topic `leapmotion/data_1` en vez de ser introducidos manualmente.

En el esquema de la Figura 6.10, se ve que hay dos topics que sale del sender, esto es debido a que se ha tenido en cuenta durante el diseño que Leap Motion puede identificar hasta dos manos, por esta razón se han separado los datos de la mano derecha (`right.msg`) y de la izquierda (left.msg) enviándolos por diferentes topics, no se ha metido toda la información en un único mensaje porque esto permite jugar con el ratio de las publicaciones, da más claridad y es más sencilla la depuración.

- *Sistemas de coordenadas diferentes*: Hay que tener en cuenta durante el diseño que las coordenadas de referencia de ROS y las que utiliza Leap Motion son distintas (como se muestra en la Figura 6.11), por ello durante la implementación hay que adaptaras adecuadamente.



(a) Coordenadas de referencia de Leap Motion

(b) Coordenadas de referencia de ROS

Figura 6.11: Fase 4: Distintos sistemas de coordenadas de referencia

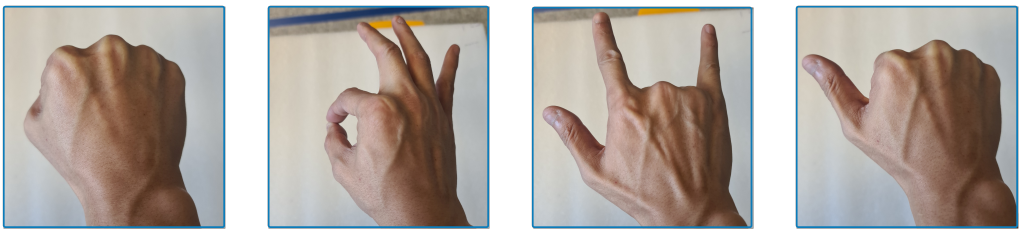
- *Workspace de Leap Motion*: También hay que tener en cuenta que la zona de trabajo de Leap Motion bastante pequeña en comparación con la del cobot UR10, por ello según que tareas se quiera realizar hay que tenerlo en cuenta, pero realizar un simple `pick & place` como es en este caso, no hay problema.
- *Formas de controlar de control mediante Leap Motion*: Los datos que se obtienen del Leap Motion, permiten implementar de manera sencilla dos formas de control:
  - *Joystick*: Este tipo de control tiene una zona muerta, que toma un origen como referencia y en esa zona muerta no se realizará ningún

movimiento, en el momento en que se sale de esa zona muerta, se va incrementando/decrementando el valor en esa coordenada dependiendo de la distancia a la que esté del origen de referencia. Esto debe ser calibrado para no realizar movimientos bruscos.

- *Imitación*: Esta es la solución que se ha escogido porque es más intuitivo, consiste en tener el origen de referencia de Leap Motion y el origen de referencia del *end-effector* del cobot, a partir de ahí el cobot seguirá los movimientos de la mano, igualmente hay que calibrarlo adecuadamente para evitar movimientos bruscos.

- *Identificación de Gestos*: Durante la identificación de los gestos hay que tener en cuenta que si los gestos que se utilizan son muy similares, Leap Motion puede realizar falsos positivos en la identificación de los gestos así como partes ocultas de la mano al moverse puede hacerle pensar que ha reconocido un gesto que no se ha realizado.

  Se ha implementado cuatro tipos de gestos que son los que se muestran en la Figura 6.12, el puño indica que hay parar de enviar instrucciones, el gesto de la pinza es para controlar la *pinza* del cobot, el gesto *thumb up* indica que está preparado y el gesto de *rock* indica que toma la posición actual de la mano como origen de referencias. Se ha preparado para la implementación para el control de la orientación del *end-effector*, pero dado que da problemas de identificación de los gestos se decidió que es mejor tener una orientación fija.



(a) Gesto: Puño - Stop (b) Gesto: Pinza - Open/Close pinza (c) Gesto: Rock - set referencia (d) Gesto: Thumb up - ready

Figura 6.12: Fase 4: Distintos sistemas de coordenadas de referencia

En el fichero `leap_interface.py` es donde se define los gestos, se va a analizar la parte del código que identifica un gesto como ejemplo. La implementación del Código Fuente 6.14 trata de comprobar cada vez que Leap Motion envía un *frame* si se ha realizado el gesto *thumb up*, para

ello del frame que entra, comprueba si es de la mano derecha o izquierda, después se comprueba lo cerrada que esté la mano comprobando si el valor del atributo `grab_strength`, en caso de ser mayor que lo definido identifica que la mano está cerrada. Sabiendo que la mano está cerrada se quiere saber si el pulgar está extendido o no y eso lo obtiene comprobando si el atributo `thumb_finger.extended()` es igual a `1` y esta es la manera de identificar gestos con Leap Motion.

```
def on_frame(self, controller):
    frame = controller.frame()

    for hand in frame.hands:
        handType = "Left hand" if hand.is_left else "Right hand"

        if handType == "Right hand":
            if hand.grab_strength > self.GRAB_STRENGTH_THRESHOLD:
                thumb_finger = hand.fingers.finger_type(0)
                for _ in thumb_finger:
                    if len(thumb_finger.extended()) == 0:
                        # print("fist in right hand: Gesture to stop controling
                        # the cobot A")
                        self.right_hand_fist = True
                        self.right_hand_thumb_up = False
                    elif len(thumb_finger.extended()) == 1:
                        # print("Pulgar extendido en la mano derecha")
                        self.right_hand_thumb_up = True
                        self.right_hand_fist = False
        else:
            if hand.grab_strength > self.GRAB_STRENGTH_THRESHOLD:
                thumb_finger = hand.fingers.finger_type(0)
                for _ in thumb_finger:
                    if len(thumb_finger.extended()) == 0:
                        # print("fist in left hand: Gesture to stop controling
                        # the cobot B")
                        self.left_hand_fist = True
                        self.left_hand_thumb_up = False
                    elif len(thumb_finger.extended()) == 1:
                        # print("Pulgar extendido en la mano izquierda")
                        self.left_hand_fist = False
                        self.left_hand_thumb_up = True
```

Código Fuente 6.14: Fase 4: Parte del contenido del fichero `leap_interface.py`

- *Calibración de la velocidad de movimiento*: Para evitar movimientos bruscos, se ha utilizado la velocidad del *end-effector* como limitante, para ello se ha cogido como velocidad máxima *0.05 rad/s* y la distancia se basa en el mayor error entre los valores de los joints actuales y de los joints de la posición futura, realizando una división se obtiene el tiempo que debe durar ese movimiento.
- *Creación de los `msgs` que transmitirán la información*: Se han creado dos tipos de mensajes para su implementación, una para identificar los movimientos y gestos de la mano derecha y la otra para la mano izquierda que también se utilizarán para definir el tipo de dato que se transmitirá por ese topic. Estos ficheros con extensión `msg` tienen que ser compilados dentro de un directorio llamado `msg` para ser integrados en el sistema ROS o no

lo encontrará. Concretamente, el contenido del Código Fuente 6.15 es para la mano derecha, para la mano izquierda sería igual pero sustituyendo *right* por *left*.

```
Header header

# Right hand information
bool is_right_hand                      # Right hand detected
geometry_msgs/Point right_hand_palmpos  # Palm's position
bool right_hand_fist                    # Fist gesture recognize
bool right_hand_thumb_up                # Thumb up gesture recognize
bool right_hand_pinch                   # Pinch gesture recognize
float32 right_hand_pinch_value                  # Pinch gesture value
bool right_hand_origin_frame            # Reference frame set
bool right_hand_set_origin_frame_detected # Detect gesture
float32 right_hand_rotate_value         # Values between [-1..0..1] rads
float32 right_hand_turn_value           # Values between [-1..0..1] rads
float32 right_hand_swing_value          # Values between [-1..0..1] rads
```

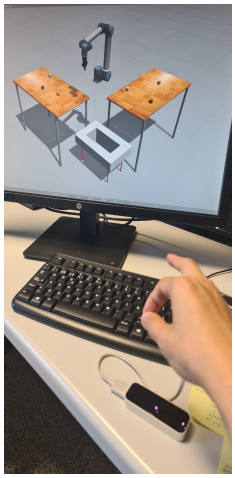
Código Fuente 6.15: Fase 4: Fichero `leapcobotright.msg`

- *Obtención y publicación de datos por el topic*: Para la obtención de los datos de Leap Motion, se utiliza la interfaz (*por ejemplo* `li.get_is_right_hand()`) definida en el fichero `leap_interface.py` que permite acceder al *objeto* que almacena los datos que interesan como se muestra en el Código Fuente 6.16 y una vez que el mensaje está formado, se envía por el topic `leapmotion/data` con el tipo de mensaje `leapcobotright`.

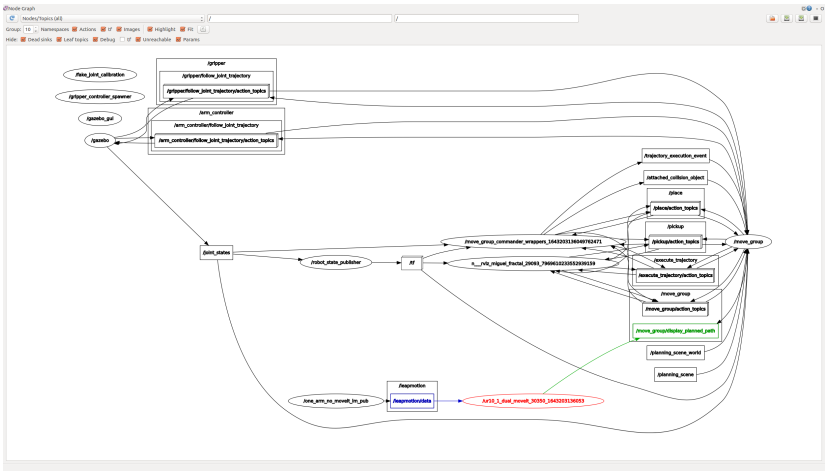
```
pub_ros_right   = rospy.Publisher('leapmotion/data', leapcobotright, queue_size=1)
    while not rospy.is_shutdown():

        # Right hand information
        msg_right = leapcobotright()
        msg_right.is_right_hand = li.get_is_right_hand()    # Right hand detected
        msg_right.right_hand_palmpos.x = right_hand_palm_pos_[0]
        msg_right.right_hand_palmpos.y = right_hand_palm_pos_[1]
        msg_right.right_hand_palmpos.z = right_hand_palm_pos_[2]

        msg_right.right_hand_fist = li.get_right_hand_fist()    # Fist gesture recognize
        [...]
        pub_ros_right.publish(msg_right)
```

Código Fuente 6.16: Fase 4: Fichero `sender.py`

– *Fase 5: Integración de Leap Motion en el sistema* Para la integración de Leap Motion en el sistema de ROS y formar parte de la solución diseñada es muy sencillo, simplemente en el script que controle los movimientos del cobot este debe estar suscrito al topic `leapmotion/data` (en el caso para dos cobots, uno de los scripts se suscribirá al topic que envíe información de la mano derecha y otro al que envíe información de la mano izquierda) y utilice esa de entrada de datos adecuadamente para gestionar los movimientos del cobot como se muestra en la Figura 6.13b, y en la Figura 6.13a se puede ver el gesto de la mano para cerrar la pinza y como la pinza en Gazebo está cerrada (para ver las imágenes de la Figura 6.13 en detalle, ver la Subsección C.1.8 del Anexo C).



(a) Comunicación entre MoveIt!, Gazebo y Leap Motion (1/2)



(b) Comunicación entre MoveIt!, Gazebo y Leap Motion (2/2)

Figura 6.13: Fase 5: Comunicación entre MoveIt!, Gazebo y Leap Motion

6.1.2. Desarrollo para dos o más cobots

Se estructura el desarrollo de la solución para dos o más cobots en las siguientes cinco fases:

– *Fase 1: Configuración del URDF para dos o más cobots*

Es idéntico a la primera fase de la subsección 6.1.1.

– *Fase 2: Configuración de MoveIt! y visualización por Rviz*

Es idéntico a la segunda fase de la subsección 6.1.1.

– *Fase 3: Simulación en Gazebo y creación del script pick & place*

Para poder controlador varios dos o más cobots simultáneamente y con diferentes controladores, lo que implica que pueden ser cobots de diferentes marcas y modelos, se realiza mediante la replicación del nodo de MoveIt! y del robot simulado en Gazebo.

Si se quiere realizar una replicación correcta hay que aplicar el concepto de *namespace*, se puede ver como fuese un directorio que contiene nodos, topics o incluso otros directorios (*namespace*s) lo que permite también una organización jerarquizada y ROS permite ejecutar instancias del mismo nodo siempre y cuando estén dentro de diferentes *namespace*s. Partiendo de lo realizado hasta la Fase 2 de la solución para un único cobot con MoveIt! (Subsección 6.1.1) se realiza cambios en los paquetes de `one_arm_moveit_gazebo` y `one_arm_moveit_manipulator` que contiene las modificaciones realizadas sobre el paquete configurado por el *setup assistant* de MoveIt! (`one_arm_moveit_config`), aunque los nombres de los paquetes no represente la solución actual, para esta explicación se mantendrán los nombres de los paquetes de la solución para un único cobot. Se va a dividir el proceso de la configuración en dos, configuración realizada en Gazebo y en MoveIt!.

- *Configuración de Gazebo*:

  Hay que tener en cuenta que se debe tener dos o más cobots en simulación, por ello se va a explicar qué cambios hay que realizar en los ficheros `launch` para permitir la adicción de dos o más cobots en la simulación correctamente. La idea es crear un fichero externo que replique (lance instancias) de tantos cobots como se quieran añadir en la simulación. Por ello primero se preparan los ficheros de Gazebo del paquete `one_arm_moveit_gazebo`.

- Fichero `ur10_joint_limited.launch`: El contenido del fichero contendrá el Código Fuente 6.17, respecto al fichero original se le ha añadido dos argumentos, el nombre del robot (`robot_name`) y la posición inicial (`init_pose`). Estos dos argumentos serán obtenidos del fichero que incluirá este `launch`.

```
<?xml version="1.0"?>
<launch>
  <arg name="robot_name"/>
  <arg name="init_pose"/>

  <include file="$(find one_arm_moveit_gazebo)/launch/ur10.launch">
    <arg name="limited" value="true"/>
    <!--arg name="gui" value="$(arg gui)"/-->
    <arg name="robot_name" value="$(arg robot_name)"/>
    <arg name="init_pose" value="$(arg init_pose)"/>
  </include>
</launch>
```

Código Fuente 6.17: Fase 3: Fichero `ur10_joint_limited.launch` modificado

- Fichero `ur10.launch`: De forma similar al fichero `launch` anterior (`ur10_joint_limited.launch`), a este fichero se le ha añadido los dos argumentos, el nombre del robot (`robot_name`) y la pose y posición inicial (`init_pose`), que se definirán del fichero que lo incluya. Aparte de la adicción de los argumentos, se ha eliminado la instanciación del mundo virtual de Gazebo y la carga del modelo del cobot en el servidor de parámetros (`robot_description`) como se puede ver en el Código Fuente 6.18.

```
<?xml version="1.0"?>
<launch>
  <arg name="limited" default="false"  doc="If true, limits joint range
  [-PI, PI] on all joints." />
  <arg name="paused" default="false" doc="Starts gazebo in paused mode" />
  <!--arg name="gui" default="true" doc="Starts gazebo gui" /-->
  <arg name="robot_name"/>
  <arg name="init_pose"/>

  <!-- push robot_description to factory and spawn robot in gazebo -->
  <node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model"
  respawn="false" output="screen" args="-urdf  -param /robot_description
  -model $(arg robot_name) $(arg init_pose)" />

  <include file="$(find one_arm_moveit_gazebo)/launch/controller_utils.launch"/>

  <!-- start this controller -->
  <rosparam file="$(find one_arm_moveit_gazebo)/controller/arm_controller_ur10.yaml"
  command="load"/>
  <node name="arm_controller_spawner" pkg="controller_manager"
  type="controller_manager" args="spawn arm_controller" respawn="false"
  output="screen"/>

  <!-- robotiq_85_gripper controller -->
  <rosparam file="$(find one_arm_moveit_gazebo)/controller/
  gripper_controller_robotiq.yaml" command="load"/>
  <node name="gripper_controller_spawner" pkg="controller_manager"
  type="spawner" args="gripper" />

  <!-- load other controllers -->
  <node name="ros_control_controller_manager" pkg="controller_manager"
  type="controller_manager" respawn="false" output="screen"
```

```
    args="load joint_group_position_controller" />

</launch>
```

Código Fuente 6.18: Fase 3: Fichero `ur10.launch` modificado

- Fichero `controller_utils.launch`: En este fichero simplemente hay que comentar el parámetro `tf_prefix`, su valor por defecto es una cadena vacía, pero eso interfiere a la hora de modificar su valor por defecto. Como se muestra en el Código Fuente 6.19, simplemente hay que dejar comentar ese trozo de código respecto al su contenido original, el resto se deja igual.

```
[...]
  <!-- Robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
  name="robot_state_publisher" respawn="true" output="screen">
    <param name="publish_frequency" type="double" value="50.0" />
    <!-- param name="tf_prefix" type="string" value="" / -->
  </node>
[...]
```

Código Fuente 6.19: Fase 3: Contenido de la parte modificada del Fichero `controller_utils.launch`

Una vez configurado los ficheros del paquete de Gazebo `one_arm_moveit_gazebo`, se procede a instanciar varios cobots en este, para ello se crea dentro del paquete `one_arm_moveit_manipulator` (puede ser cualquier otro paquete) un fichero `launch` llamado `two_arm_moveit_gazebo.launch`.

El contenido del fichero es el siguiente Código Fuente 6.20, lo primero que se aprecia es que carga el modelo del robot en el servidor de parámetros e instancia el mundo virtual de Gazebo, lo que se eliminó del fichero `ur10.launch`, esto debe estar aquí porque solo se quiere instanciar una vez el mundo de Gazebo.

Después aparecen dos grupos, `ur10_1` y `ur10_2`, esta es la forma de definir los `namespaces`, la configuración de estos cobots es idéntica excepto por tres cosas, el valor del parámetro `tf_prefix` que será el prefijo que irá en las transfromadas, es importante que *coincida* con el nombre del grupo (*namespace*), el nombre (`robot_name`) y su posición inicial (`init_pose`). Si se quiere añadir más cobots al sistema, simplemente hay que copiar el contenido de grupo y modificar el contenido adecuadamente. Y las últimas dos líneas de código unen la base de los cobots con el *frame* `world`.

```
<launch>
    <param name="/use_sim_time" value="true"/>
    <arg name="robot_name"/>
    <arg name="init_pose"/>
    <arg name="paused" default="false"/>
    <arg name="gui" default="true"/>
    <arg name="limited" default="true"  doc="If true, limits joint range
    [-PI, PI] on all joints." />


    <!-- send robot urdf to param server -->
    <include file="$(find one_arm_moveit_description)/launch/ur10_upload.launch">
      <arg name="limited" value="$(arg limited)"/>
    </include>

    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <!--arg name="world_name" default="worlds/empty.world"/-->
        <arg name="verbose" value="true"/>
        <arg name="world_name" default="$(find one_arm_moveit_gazebo)/world/
        multiarm_bot.world"/>
        <arg name="paused" value="$(arg paused)"/>
        <!--arg name="gui" value="$(arg gui)"/-->
        <arg name="gui" value="$(arg gui)"/>
    </include>

    <group ns="ur10_1">
        <param name="tf_prefix" value="ur10_1" />
        <include file="$(find one_arm_moveit_gazebo)/launch/
        ur10_joint_limited.launch">
            <arg name="init_pose" value="-x 0.6 -y -0.6 -z 1.1"/>
            <arg name="robot_name" value="ur10_1"/>
        </include>
    </group>

    <group ns="ur10_2">
        <param name="tf_prefix" value="ur10_2" />
        <include file="$(find one_arm_moveit_gazebo)/launch/
        ur10_joint_limited.launch">
            <arg name="robot_name" value="ur10_2"/>
            <arg name="init_pose" value="-x 0.6 -y 1.38 -z 1.1"/>
        </include>
    </group>

    <node pkg="tf" type="static_transform_publisher"
    name="world_frames_connection_1" args="0 0 0 0 0 0
    /world /ur10_1/world 100"/>

    <node pkg="tf" type="static_transform_publisher"
    name="world_frames_connection_2" args="0 0 0 0 0 0
    /world /ur10_2/world 100"/>

</launch>
```

Código Fuente 6.20: Fase 3: Contenido del Fichero `two_arm_moveit_gazebo.launch`

Con esto, Gazebo ya está configurado, se procede a lanzar el fichero `two_arm_moveit_gazebo` para comprobar que se han realizado las modificaciones adecuadamente. En la Figura 6.14 se ve que está simulando dos cobots correctamente. Hay más resultados en el Anexo C, en la Subsección C.2.1 está el árbol de transformadas, en donde se puede ver cómo se han definido los *namespaces* correctamente y están representadas ambos cobots y finalmente está la representación de los nodos y los topics

en la Subsección C.2.2, lo más importante de aquí es la aparición de dos grandes recuadros que contienen a cada uno de los cobots y sus controladores sin tener que configurarlo de nuevo, finalmente se puede apreciar que ambos cobots están conectados al simulador Gazebo, la configuración se ha realizado correctamente.

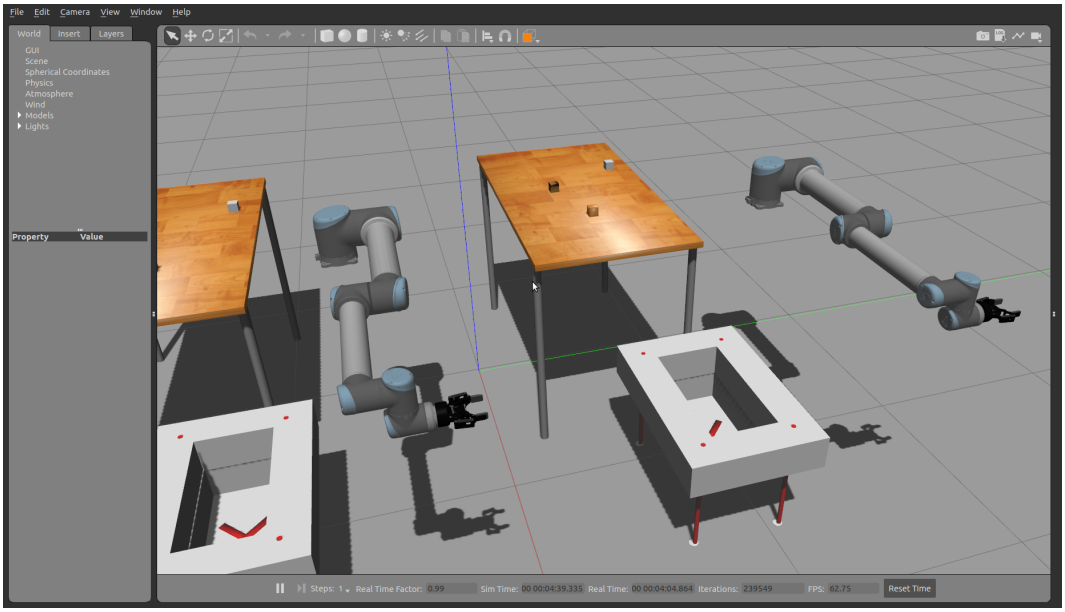


Figura 6.14: Fase 3: Visualización de dos cobots en Gazebo

- *Configuración de MoveIt!*:

  Los cambios realizados para la configuración de MoveIt! son muy pequeñas, básicamente hay que agrupar el código de lo que se había implementado para un único cobot bajo un *namespace*, adecuar el `remap` con el *namespacing* y posteriormente replicar el proceso tantas veces como cobots se esté simulando. El Código Fuente 6.21 si se compara con El Código Fuente 6.11 de la solución desarrollada para un único cobot, esencialmente se ha agrupado todo en un *namespace* y se ha añadido el nombre del *namespace* como prefijo en los nombres de los topics del `remap` para la correcta comunicación con los controladores.

```
<launch>
    <arg name="sim" default="false" />
    <arg name="debug" default="false" />

    <group ns="ur10_1">
      <rosparam command="load" file="$(find one_arm_moveit_manipulator)/config
      /joint_names.yaml"/>

      <include file="$(find one_arm_moveit_manipulator)/launch/
      planning_context_ur10_1.launch">
        <arg name="load_robot_description" value="true"/>
      </include>

      <include file="$(find one_arm_moveit_manipulator)/launch/move_group.launch">
        <arg name="debug" default="$(arg debug)" />
        <arg name="publish_monitored_planning_scene" value="true"/>
        <!--arg name="info" value="true"/-->
      </include>

      <!-- Remap follow_joint_trajectory -->
      <remap from="/ur10_1/follow_joint_trajectory"
      to="/ur10_1/arm_controller/follow_joint_trajectory"/>

      <include file="$(find one_arm_moveit_config)/launch/moveit_rviz.launch">
        <arg name="config" value="true"/>
        <arg name="debug" default="false"/>
      </include>
    </group>

    <group ns="ur10_2">
      <rosparam command="load" file="$(find one_arm_moveit_manipulator)/config/
      joint_names.yaml"/>

      <include file="$(find one_arm_moveit_manipulator)/launch/
      planning_context.launch">
        <arg name="load_robot_description" value="true"/>
      </include>

      <include file="$(find one_arm_moveit_manipulator)/launch/move_group.launch">
        <arg name="debug" default="$(arg debug)" />
        <arg name="publish_monitored_planning_scene" value="true"/>
        <!--arg name="info" value="true"/-->
      </include>

      <!-- Remap follow_joint_trajectory -->
      <remap from="/ur10_2/follow_joint_trajectory"
      to="/ur10_2/arm_controller/follow_joint_trajectory"/>

      <include file="$(find one_arm_moveit_config)/launch/moveit_rviz.launch">
        <arg name="config" value="true"/>
        <arg name="debug" default="false"/>
      </include>
    </group>
</launch>
```

Código Fuente 6.21: Fase 3: Contenido del Fichero `two_arm_moveit_execution.launch`

Se procede a comprobar si la comunicación entre Gazebo y las réplicas de MoveIt! se comunican correctamente. Se puede ver dos grandes agrupaciones, en donde cada cobot se está comunicando con su `move_group` asignado, la comunicación con los controladores y el camino de las transformadas y valores de los joints tiene un único origen que es el nodo de Gazebo, esto

es muy importante para evitar movimientos extraños dependiendo de la frecuencia en que se produzcan esas interferencias.

El camino parte del nodo de `gazebo` es el único que publica a los topics `/joint_states` de ambos *namespaces*, el nodo `move_group` está suscrito a este topic también, después llega al nodo `robot_state_publisher` que realiza las transformadas y se las envía por el topic `/tf` el cual el nodo `move_group` también está suscrito, esto es importante porque `move_group` utiliza la información que proviene de ambos para realizar la planificación de las trayectorias.

Las imágenes obtenidas de la herramienta `rqt_graph` tienen ya muchos nodos y las líneas de comunicación no se aprecian bien, en la Subsección C.2.3 del Anexo C están las imágenes, pero se recomienda reproducirlo (ver Anexo F) y obtener la gráfica con la herramienta para verlo en detalle.

Finalmente, se va a explicar el script que controla a un cobot y realiza la tarea de *pick & place*, no se ha tenido que explicar previamente debido a que no había *namespacings* y la configuración era sencilla, pero ahora hay que tenerlo en cuenta o no se comunicará correctamente con el nodo `move_group` y sus servicios, se puede hacer complicado por la cantidad de formas que MoveIt! te permite configurarlo, por tanto, a la hora de realizar cambios hay que ir con cuidado.

Se va a realizar la explicación para uno, porque el segundo y sucesivos scripts son una copia de este primero con algunos datos modificados adecuadamente para que se comunique con su MoveIt! correspondiente. La idea es lanzarlos en terminales diferentes para confirmar que se están ejecutando simultáneamente, lo ideal en este caso sería tenerlos ejecutando en diferentes ordenadores.

El código es relativamente largo, por lo que solamente se va a explicar lo más importante, que es la configuración para comunicarse con la API de `move_group` cuando se tiene *namespaces*. El Código Fuente 6.22 está bastante claro, pero no está bien documentado si es la primera vez que se trabaja con MoveIt!, por ello se va a explicarlo en detalle.

```
1 def main():
2     moveit_commander.roscpp_initializer.roscpp_initialize(sys.argv)
3     rospy.init_node('ur10_1_arm_moveit',
4                     anonymous=True)
5 
6     PLANNING_GROUP_GRIPPER = "gripper"
7     PLANNING_GROUP_ARM = "manipulator"
8     PLANNING_NS = "/ur10_1/"
9     REFERENCE_FRAME = "/ur10_1/world"
10 
11     ## Instantiate a RobotCommander object.  This object is an interface to
12     ## the robot as a whole.
13     robot = moveit_commander.RobotCommander(
14         "%srobot_description"%PLANNING_NS,
15         ns="/ur10_1/"
16     )
17 
18     arm = moveit_commander.move_group.MoveGroupCommander(
19         PLANNING_GROUP_ARM,
20         "%srobot_description"%PLANNING_NS,
21         ns="/ur10_1/"
22     )
23 
24     gripper = moveit_commander.move_group.MoveGroupCommander(
25         PLANNING_GROUP_GRIPPER,
26         "%srobot_description"%PLANNING_NS,
27         ns="/ur10_1/"
28     )
29 
30     ## We create this DisplayTrajectory publisher which is used below to publish
31     ## trajectories for RVIZ to visualize.
32     display_trajectory_publisher = rospy.Publisher(
33                                         '/move_group/display_planned_path',
34                                         moveit_msgs.msg.DisplayTrajectory,
35                                         queue_size=10
36                                         )
37 
38     rospy.sleep(2)
39 
40     arm.set_num_planning_attempts(15)
41     arm.set_planning_time(5)
42     arm.allow_looking(True)
43     arm.allow_replanning(True)
44     arm.set_pose_reference_frame(REFERENCE_FRAME)
45     arm.set_goal_position_tolerance(0.001)
46     arm.set_goal_orientation_tolerance(0.001)
47 
48     gripper.set_num_planning_attempts(15)
49     gripper.allow_replanning(True)
50     gripper.allow_looking(True)
51 
52     pick_place(arm, gripper)
53 
54     ## When finished shut down moveit_commander.
55     moveit_commander.roscpp_shutdown()
```

Código Fuente 6.22: Fase 3: Parte del contenido del Fichero `two_arm_moveit_1.py`

Como se puede ver en el Código Fuente 6.22, se ha definido la configuración en la función `main`, pero no es necesario que sea aquí, es más es preferible modularlo y pasarle la configuración por parámetros, pero para esta explicación es suficiente.

- *Línea 2:* Lo primero es inicializar `moveit_commander` que es una API

sobre la interfaz desarrollada en C++, lo que es definido como Wrappers, este provee la mayoría de las funcionalidades que provee la interfaz de la versión para C++ pero no están todas las funcionalidades de MoveIt!. Es necesario porque entre sus funcionalidades permite calcular trayectorias cartesianas que es la funcionalidad que principalmente se requiere.

- *Línea 3:* Inicializa el nodo al que se le ha nombrado ur10_1_arm_moveit.
- *Líneas 6-9*: Se definen las constantes para facilitar la configuración, las dos primeras líneas (6 y 7) definen los nombres que se pusieron a los grupos de planificación, esto se puede comprobar en el Anexo B, en este caso se les llamó *gripper* para la pinza y *manipulator* para el brazo del cobot UR10. Las líneas 8 y 9 definen la información necesaria para configurar el planificador, PLANNING_NS contiene el nombre del *namespace* donde está el nodo move_group con el que se quiere comunicar y REFERENCE_FRAME es el link que se tomará como referencia para realizar los cálculos de trayectoria con respecto al end-effector (el end-effector es ee_link), en este caso daría igual que sea /ur10_1/world pero lo correcto sería tomar el link /ur10_1/base_link como referencia.
- *Línea 13*: Inicializa el RobotCommander, como en el código indica, se utiliza para controlar el robot, hay que pasarle como parámetros el *namespace* y qué robot_description debe utilizar para definir el robot, porque en el momento hay tres descripciones, que son el de Gazebo (robot_description) y los otros dos definidos en el fichero planning_context.launch que son instanciadas dentro de sus corresponientes *namespaces* por ello se tiene /ur10_1/robot_description y /ur10_2/robot_description.
- *Línea 18 y 24*: Se crea una interfaz para un conjunto de Joints, en este caso la interfaz arm para el conjunto de joints dinámicos del brazo del cobot UR10 y la interfaz gripper para el joint que controla la pinza. A través de estas interfaces se realizarán las planificaciones de las trayectorias y su ejecución (es posible realizarlo con la interfaz de robot, ya que contiene a estas dos interfaces, pero es más claro y cómodo realizarlo de esta manera.
- *Línea 32*: Como dice en el comentario, display_trajectory_publisher realiza publicaciones al topic /move_group/display_planned_path y el cual Rviz se suscribe para visualizar las trayectorias, no es necesario, pero para depuración es recomendable.

- *Línea 38*: Simplemente espera dos segundos, se asegura que las instancias anteriores se han cargado correctamente en el sistema, hay que tener en cuenta alguno de ellos instancian nodos y si el ordenador sobre el que se está lazando es lento puede provocar una situación de no deseada.
- *Líneas 40-50*: Aquí se está configurando algunas opciones del planificador que se va a utilizar, por defecto es `RTT`, pero se puede modificar. Los más importantes son los últimos en las líneas 44, 45 y 46. En la línea 44 se define el link de referencia que se utilizará para realizar las planificaciones, la línea 45 y 45 definen el margen de error aceptable del resultado obtenido del planificador para la posición y la orientación, hay que tener cuidado porque cuanto más pequeño es el error que se defina más tiempo tardará el planificador en dar una respuesta, está definidos para permitir errores de milímetros. El mismo proceso para la interfaz de la pinza (`gripper`).
- *Línea 52*: aquí se lanza la tarea de *pick & place*.
- *Línea 55*: Una vez que termine la tarea *pick & place* finaliza.

Tras la descripción detallada del código, para que el script controlase otro cobot que esté en otro *namespace* es tan sencillo como cambiarle el valor de la variable `PLANNING_NS` por la del nombre del *namespace* donde esté definido el cobot objetivo (el código completo está en Github, ver Anexo F).

Si se ejecuta el `pick & place` con ambos brazos, se aprecia que realizan la tarea simultáneamente y con un poco de retraso entre ellos debido a que uno empieza un poco más tarde que otro, se puede realizar pruebas a lanzar los scripts a destiempo y comprobar que efectivamente se mueven simultáneamente. En el siguiente Capítulo 8 se mostrará unas imágenes que dan una idea de cómo funcionan durante la simulación.

– *Fase 4: Diseño de la interfaz de Leap Motion*

Es idéntico a la cuarta fase de la subsección 6.1.1.

– *Fase 5: Integración de Leap Motion en el sistema*

Los pasos son los mismos que en lo descrito en la subsección 6.1.1, la única diferencia es que ahora se controla dos cobots, pero únicamente se tiene que suscribir a uno de los topics que el nodo `sender` publica y tratar la información que le llega adecuadamente. Y la configuración para la comunicación con cada cobot para que envíen las trayectorias correctamente está descrito en la fase anterior de esta solución.

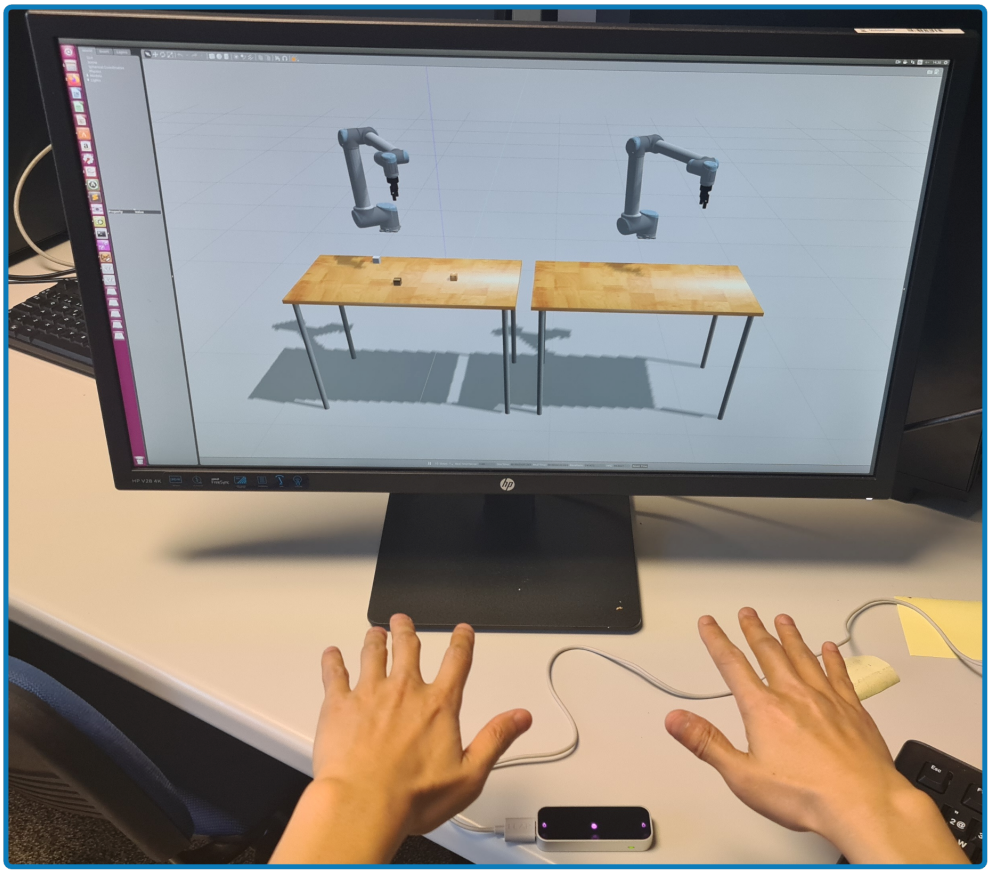


Figura 6.15: Fase 5: Control de dos cobots simultáneamente con Leap Motion en el simulador de Gazebo

6.2. Solución desarrollada *sin* el paquete de MoveIt!

En esta sección se presenta la solución sin utilizar el paquete de MoveIt!, para el control simultáneo de uno, dos y más cobots.

6.2.1. Desarrollo para un único cobot

Se estructura el desarrollo de la solución para único cobot en las siguientes cinco fases:

– *Fase 1: Configuración del URDF para un único cobot*

  Es idéntico a la primera fase de la subsección 6.1.1.

– *Fase 2: Creación del planificador y nodos auxiliares*

En la Figura 6.16, está el esquema del diseño de la arquitectura del planificador que se ha implementado. Se puede apreciar dos divisiones Gazebo (son los nodos de color rojo y los topics de color naranja) y el planificador (son los nodos de color azul y topics de color verde), se ha dividido así para facilitar la explicación.

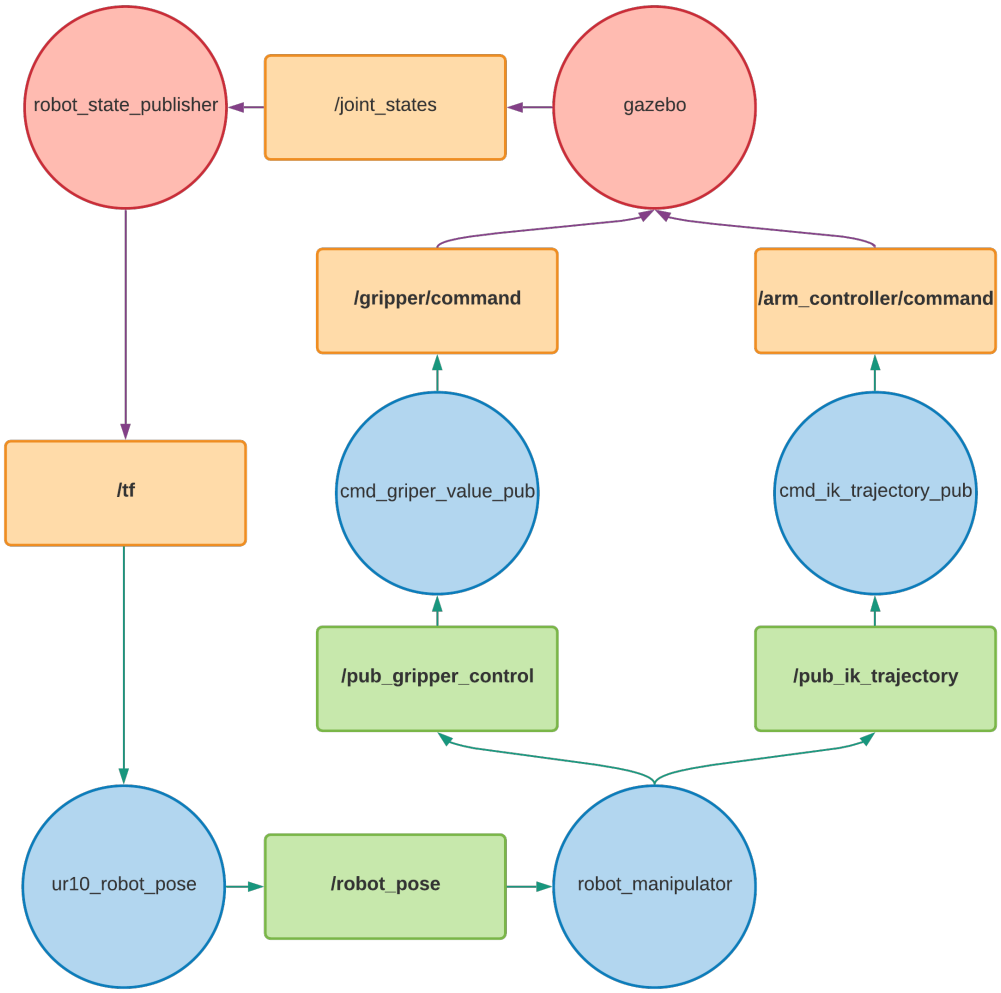


Figura 6.16: Fase 2: Arquitectura del planificador propio

El grupo de nodos y topics del grupo de Gazebo, no se han tenido que tocar y se ha tomado ventaja de su existencia para obtener la información necesaria que el planificador necesita para realizar su tarea. Este grupo se comunica con el grupo del planificador mediante tres topics `/tf` que contiene información de las transformadas del robot y los topics `/gripper/command` y `/arm_controller/command` que reciben la información necesaria para realizar movimientos en el robot.

Teniendo conocimiento de esos nodos y topics del grupo de Gazebo, la solución diseñada para el planificador es comunicarse con ellos para

obtener la información que necesita. El nodo `ur10_robot_pose` obtiene la información de las transformadas y le envía la posición del *end-effector* al nodo `robot_manipulator`, que realiza dos funciones principales, la primera es el control de la pinza y la segunda es planificar la trayectoria del brazo. Los nodos `cmd_gripper_value_pub` y `cmd_ik_trajectory_pub` obtienen las órdenes del nodo `robot_manipulator` y se lo envía a los topic de los controladores de Gazebo directamente (`/gripper/command` y `/arm_controller/command`).

Una vez explicado el funcionamiento general de la solución se va a explicar lo que hace cada nodo con más detalle (el código fuente está disponible en Github, ver Anexo F):

- `ur10_robot_pose` script: Este script crea un nodo que únicamente publica la posición del *end-effector* con cierta frecuencia (10 Hz) mediante la librería `tf` de ROS, es decir, obtiene la posición del *end-effector* de la información del topic `/tf` que es publicada en el topic`/robot_pose`. La posición del *end-effector* es obtenida con la diferencia entre las posiciones del `/base_link` y `/ee_link` (es el link que conecta con la pinza).

  Realmente no es necesario este nodo, el nodo `robot_manipulator` podría gestionarlo de propio, aunque sea redundante, pero muy útil durante la depuración y el entendimiento de cómo funciona el planificador.

- `pub_gripper_cmd` script: El script crea un nodo que únicamente recibe y transmite las órdenes recibidas por el topic `/pub_gripper_control` al topic del controlador de la pinza `/gripper/command`. Aunque la funcionalidad es sencilla, este nodo permite cambios del valor que se le envía al controlador en cualquier momento, porque no realiza ninguna comprobación de que se ha realizado el movimiento correctamente, esto es un comportamiento deseado. Y si no recibe ninguna orden sigue publicando la orden anterior manteniendo el valor de la pinza y si recibe una nueva orden, desecha inmediatamente el valor anterior permitiendo realizar cambios durante la ejecución del movimiento anterior.

- `pub_ik_trajectory` script: Funciona de la misma manera que lo descrito para el nodo `cmd_gripper_value_pub`, este script instancia el nodo `cmd_ik_trajectory_pub`. Recibe las órdenes del topic `/pub_ik_trajectory` al cual está suscrito y los transmite al topic del

controlador del brazo `/arm_controller/command`. También permite realizar cambios durante la ejecución de una trayectoria, lo que permite cambios bruscos de dirección sin tener que esperar a que termine de llegar a la posición previamente designada.

- `robot_manipulator` script: Este script instancia el nodo `robot_manipulator` que obtiene la posición actual del *end-effector* del topic `/robot_pose`, esta información es necesaria para poder obtener los valores de cada uno de las articulaciones necesarias para llegar a la posición en cartesiano que se desea. Una vez obtenido el valor que deben tener las articulaciones se publican por el topic `/pub_ik_trajectory`.

  Se ha tenido que implementar un script que hace la función de librería, esta librería está compuesta de varias funciones para realizar los cálculos necesarios en la obtención de la cinemática directa (forward kinematic) y la cinemática inversa (inverse kinematic), el proceso de obtención de estas funciones está documentado en el Anexo D.

  Para el caso del gripper, como solamente hay que controlar el valor de una articulación, no es necesario ningún cálculo, simplemente se le pasa el valor deseado.

  Durante la implementación hay que tener en cuenta el tipo de los mensajes que deben recibir los controladores para construirlos adecuadamente o no realizarán ningún movimiento o realizará movimientos no deseados. Esta solución pueda dar problemas debido a la aparición de singularidades[5], esto es debido a que pueden existir divisiones por cero durante el cálculo de las ecuaciones, principalmente debido cuando dos articulaciones están alineadas, para evitarlo, en este caso, simplemente se ha limitado el entorno de trabajo del robot.

– *Fase 3: Simulación en Gazebo y creación del script pick & place*

  Por no repetir de nuevo la configuración de Gazebo y sus controladores porque la configuración es igual para un único cobot, el proceso para configurarlo está en: *Configuración de Gazebo y controladores*.

  Partiendo de la configuración realizada de Gazebo la única diferencia es la adición al fichero `controller_utils.launch` los nodos implementados, para

[5] *Singularidad cinemática*: punto en el espacio de trabajo donde el robot pierde su capacidad de mover el efector final en alguna dirección, sin importar cómo mueva sus articulaciones.

no tener que lanzarlos manualmente como se muestra en el Código Fuente 6.23.

```
<!-- get the robot position [Own Script]-->
<node name="ur10_robot_pose" pkg="one_arm_no_moveit_gazebo"
type="ur10_robot_pose.py" respawn="true" />

<!-- send the arms commands [Own Script]-->
<node name="cmd_ik_trajectory_pub" pkg="one_arm_no_moveit_gazebo"
type="pub_ik_trajectory.py" respawn="true" />

<!-- send the gripper commands [Own Script]-->
<node name="cmd_gripper_value_pub" pkg="one_arm_no_moveit_gazebo"
type="pub_gripper_cmd.py" respawn="true" />
```

Código Fuente 6.23: Fase 3: Parte del contenido del Fichero `controller_utils.launch`

Al lanzar la configuración junto al script que realiza el *pick & place*, se obtiene la gráfica de nodos y topics con la herramienta `rqt_graph` de la Figura 6.17 que efectivamente se comunica de la forma deseada con los controladores del cobot y con Gazebo (para visualizar las imágenes con más detalle, ver la Subsección C.2.4 del Anexo C).

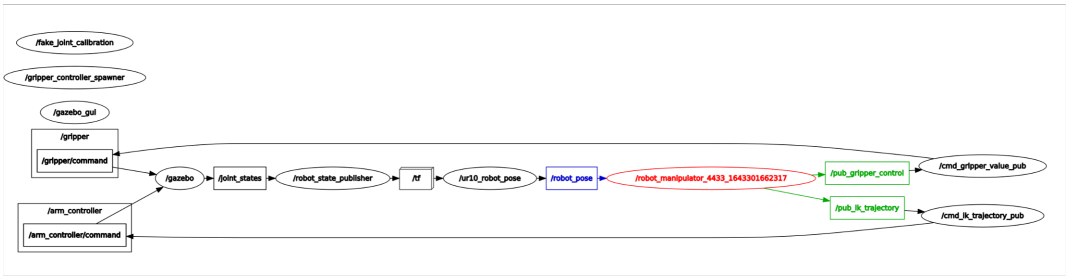


Figura 6.17: Fase 3: Nodos y topics del planificador propio para un cobot

– *Fase 4: Diseño de la interfaz de Leap Motion*

  Es idéntico a la cuarta fase de la subsección 6.1.1.

– *Fase 5: Integración de Leap Motion en el sistema*

  Es idéntico a la quinta fase de la subsección 6.1.1.

6.2.2. Desarrollo para dos o más cobot

Se estructura el desarrollo de la solución para dos o más cobots en las siguientes cinco fases:

– *Fase 1: Configuración del URDF para dos o más cobots*

Es parecido a lo realizado en la primera fase de la subsección 6.1.1, pero con la diferencia de que esta vez son dos cobots UR10 y no uno, como se muestra en la Figura 6.18, si se quiere ver el código completo, está disponible en Github (ver Anexo F).

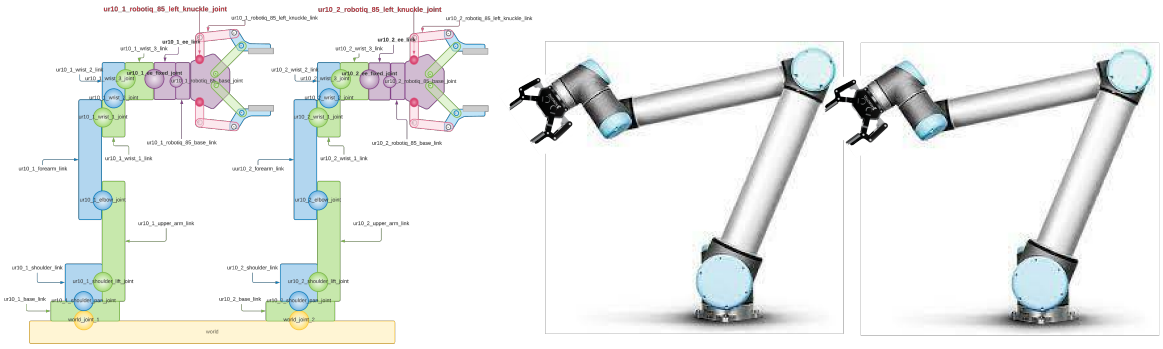


Figura 6.18: Fichero `ur10_robot.urdf.xacro`

En el Código Fuente 6.24, se puede ver el cómo se conectan los componentes de los brazos `ur10_robot` con el link `world`, esto se puede apreciar en la Figura 6.2 representado `world` (color amarillo) y las bases de los brazos de los UR10s `ur10_1_base_link` y `ur10_2_base_link` (color verde) situado justo encima, además los joints `ur10_1_world_joint` y `ur10_2_world_joint` son las esferas de color amarillo situado entre ambos links. De la misma manera, los componentes de las pinzas `robotiq_85_gripper` están conectados a los brazos de los UR10s (`ur10_robot`), esto se aprecia en la Figura 6.18 en donde las esferas que representan los joints `ur10_1_robotiq_85_base_joint` y `ur10_2_robotiq_85_base_joint` que unen ambos componentes son de color morado, uniendo los links `ur10_1_robotiq_85_base_link` y `ur10_2_robotiq_85_base_link` de las pinzas con los links `ur10_1_ee_link` y `ur10_2_ee_link` de los brazos de los UR10s.

```
  <link name="world" />

  <joint name="ur10_1_world_joint" type="fixed">
    <parent link="world" />
    <child link = "ur10_1_base_link" />
    <origin xyz="0.6 -0.6 1.1" rpy="0.0 0.0 0.0" />
  </joint>

  <!-- arm -->
  <xacro:ur10_robot prefix="ur10_1_" joint_limited="false"
    transmission_hw_interface="$(arg transmission_hw_interface)"
  />

  <!-- gripper -->
  <xacro:robotiq_85_gripper prefix="ur10_1_" parent="ur10_1_ee_link" >
```

```
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:robotiq_85_gripper>


<joint name="ur10_2_world_joint" type="fixed">
  <parent link="world" />
  <child link = "ur10_2_base_link" />
  <origin xyz="0.6 1.38 1.1" rpy="0.0 0.0 0.0" />
</joint>

<!-- arm -->
<xacro:ur10_robot prefix="ur10_2_" joint_limited="false"
  transmission_hw_interface="$(arg transmission_hw_interface)"
/>

<!-- gripper -->
<xacro:robotiq_85_gripper prefix="ur10_2_" parent="ur10_2_ee_link" >
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:robotiq_85_gripper>
```

Código Fuente 6.24: Parte del fichero `ur10_robot.urdf.xacro` (no moveit)

Para añadir más de dos robots al sistema, hay que simplemente copiar y pegar los componentes `ur10_robot` y `robotiq_85_gripper`, modificando el valor en sus atributos `prefix` por uno nombre único (prefijo que afectará en la configuración de los controladores) y fijarlo al mundo realizando las mismas operaciones con ese joint y modificar el origen de este joint, o habrá solapamiento de los cobots al simularlo. Es decir, añadiendo la siguiente porción de código (Código Fuente 6.25) al Código Fuente 6.24 anterior, se obtendría tres cobots.

```
<joint name="ur10_3_world_joint" type="fixed">
  <parent link="world" />
  <child link = "ur10_3_base_link" />
  <origin xyz="0.6 3.36 1.1" rpy="0.0 0.0 0.0" />
</joint>

<!-- arm -->
<xacro:ur10_robot prefix="ur10_3_" joint_limited="false"
  transmission_hw_interface="$(arg transmission_hw_interface)"
/>

<!-- gripper -->
<xacro:robotiq_85_gripper prefix="ur10_3_" parent="ur10_3_ee_link" >
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:robotiq_85_gripper>
```

Código Fuente 6.25: Ejemplo de la porción de código a añadir a `ur10_robot.urdf.xacro` para obtener un tercer cobot

– *Fase 2: Creación del planificador y nodos auxiliares*

La explicación es la misma que en la Fase 2 de la Subsección 6.2.1. La diferencia es visible, hay una duplicación de los nodos, debido a la duplicación de los robots a controlar, aunque la organización se puede mejorar para facilitar la escalabilidad del sistema, el funcionamiento esencialmente es el mismo. En la Figura 6.19 se puede apreciar que lo dicho anteriormente, se han duplicado los nodos de propia creación así como el el número de controladores de Gazebo.

Hay que tener cuidado a la hora de escalar en esta solución, el nombre de las articulaciones deben ser únicos, esto es debido a la configuración del modelo del robot. Al no estar dentro de un *namespace* que generaría automáticamente nombres únicos para las articulaciones, se tiene que realizar los cambios manualmente y esto afecta también a los nombres de los topics, lo que implica su duplicación, también hay que modificar los scripts para que publiquen y se suscriban a los topics correctos.

Esto no implica que el sistema en sí sea poco escalable, porque es sencillo hacer que los scripts reciban un argumento que tomen como prefijo lo que generaría nombres únicos de manera automática desde los ficheros `launch`, como lo haría un *namespace*.

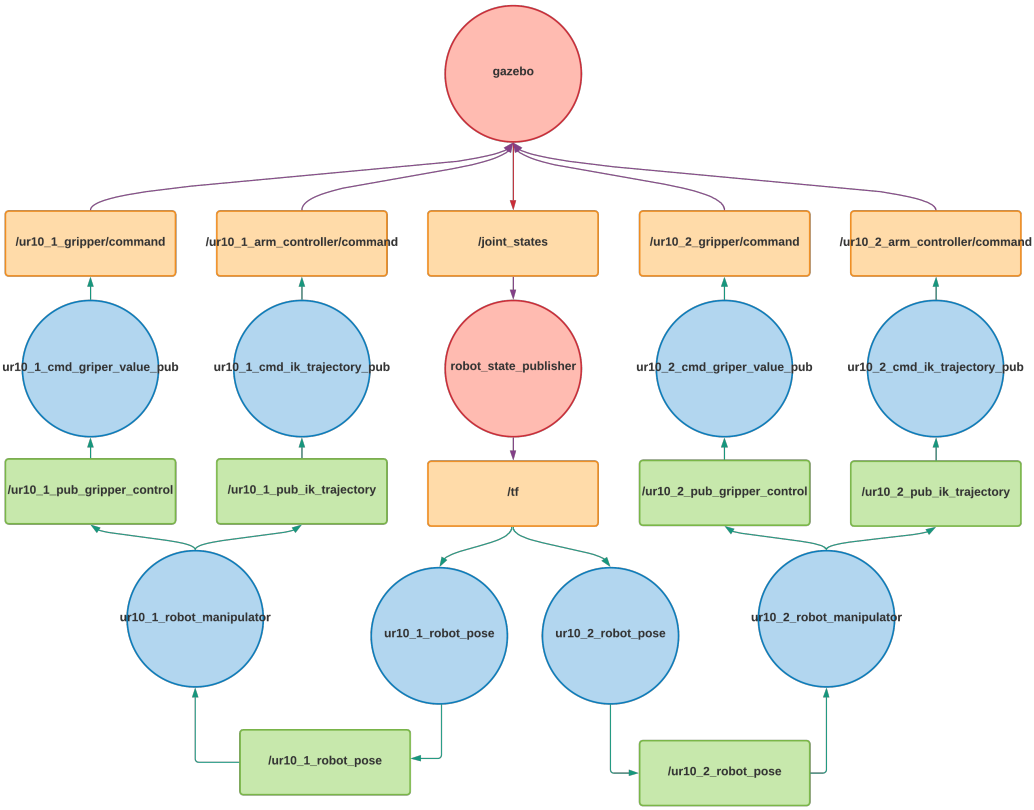


Figura 6.19: Fase 2: Arquitectura del planificador propio para dos cobots

– *Fase 3: Simulación en Gazebo y creación del script pick & place*

Por no repetir de nuevo la configuración de Gazebo y sus controladores porque la configuración es igual para un único cobot, el proceso para configurarlo está en: *Configuración de Gazebo y controladores*. Sobre esta configuración, hay que redefinir los ficheros que configuran los controladores de Gazebo, básicamente

es duplicarlos y añadirles el prefijo `ur10_1_` y `ur10_2_` al nombre de las articulaciones, tanto para la pinza como para el brazo de UR10.

De manera similar hay que añadir los nodos duplicados y adaptados para su comunicación con los controladores del cobot en Gazebo y del nodo que obtiene la posición actual de cada cobot del topic `/tf` deben ser también añadidos al fichero `controller_utils.launch`.

Al lanzar la configuración junto al script que realiza el *pick & place* en los dos cobots, se obtiene la gráfica de nodos y topics con la herramienta `rqt_graph` de la Figura 6.20 que efectivamente se comunica de la forma deseada con los controladores de los cobots y con Gazebo (para visualizar las imágenes con más detalle, ver la Subsección C.2.5 del Anexo C).

Figura 6.20: Fase 3: Nodos y topics del planificador propio para dos cobots

– *Fase 4: Diseño de la interfaz de Leap Motion*

  Es idéntico a la cuarta fase de la subsección 6.1.1.

– *Fase 5: Integración de Leap Motion en el sistema*

  Es idéntico a la quinta fase de la subsección 6.1.2.

Capítulo 7

Pruebas en el Robot Real: Campero

Tras desarrollar todas las soluciones, se procede a realizar pruebas en el robot real Campero, que es la fase 6 del diseño como se puede ver en la Figura 7.1, tal y como se planteó durante la división en la realización del proyecto en la Sección 5.3. Diseño de la solución en el Capítulo 5. La solución que se va a probar en el robot Campero es la solución desarrollada sin el paquete MoveIt! para un único cobot, las fases para su desarrollo en el campero es igual que el seguido para todas las soluciones propuestas, reutilizando todo lo que se pueda reutilizar.

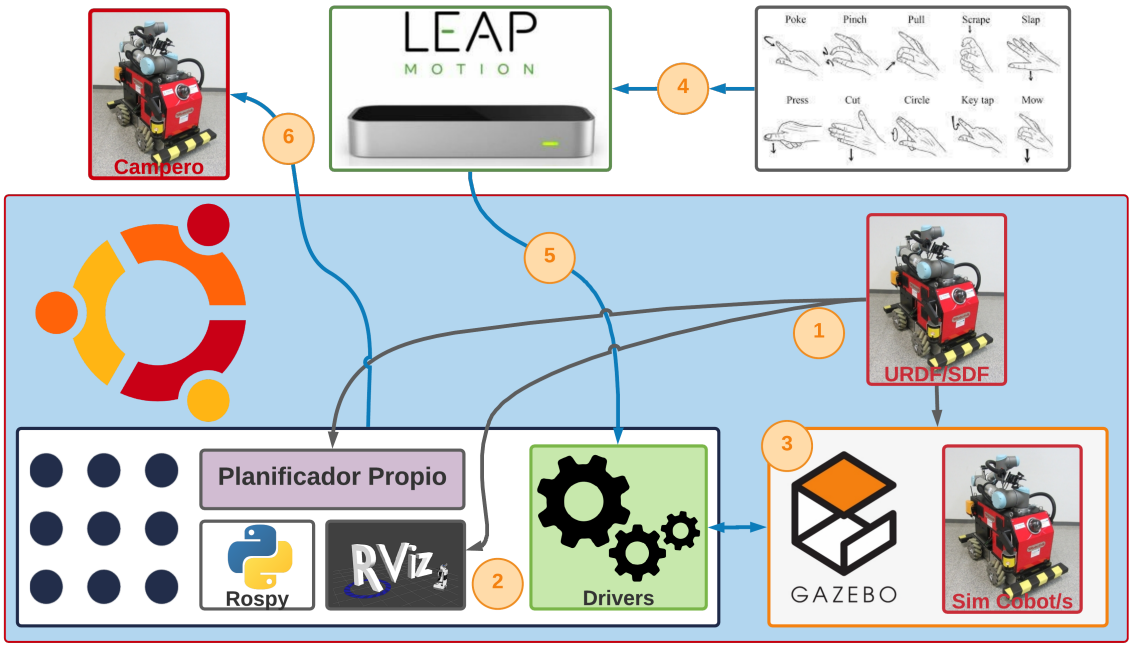


Figura 7.1: Diseño general de la arquitectura del sistema

La estructura del proyecto para el desarrollo en el robot Campero es la misma que se ha seguido para todas las soluciones. No se va a entrar en profundidad en las explicaciones porque la mayoría ya está explicado el Capítulo 6. Añadir la configuración

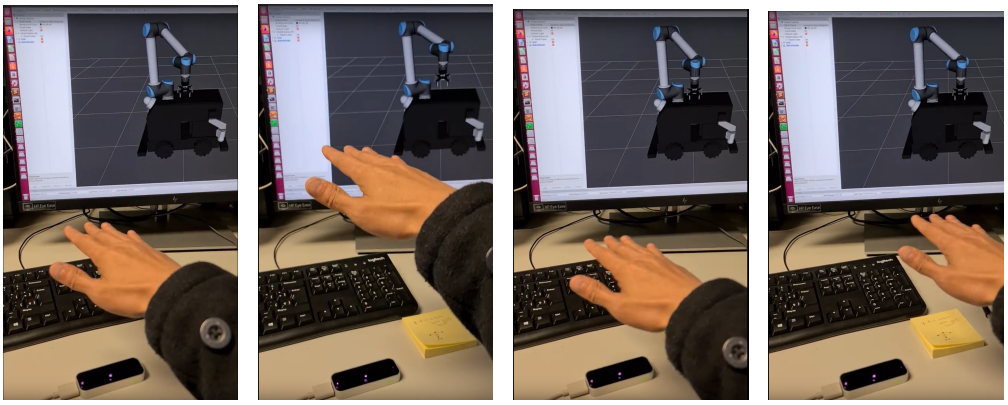
para que el robot funcione ya está implementando y lo único que hay que hacer es adaptar lo desarrollado a lo ya existente.

El robot campero está compuesto por dos servidores, uno es el servidor principal que es donde el sistema operativo de Ubuntu 16.04 y ROS está instalado y el otro está conectado al brazo del cobot UR10. Los ficheros de configuración de ROS están en el directorio `catkin_ws` y de ahí se copia los paquetes que consideremos necesarios partiendo del fichero `campero_complete.launch` (contenido en el paquete `campero_bringup`) que es el fichero que se instancia por defecto al ponerse en marcha. Si se traza el contenido de este fichero `launch`, se puede saber cómo está configurado y los paquetes y scripts que utiliza.

Una vez que se conoce cómo ha sido organizado el sistema, se puede empezar a adaptar lo desarrollado al robot Campero. Se crea un nuevo directorio de trabajo para ROS con la herramienta de *catkin*, en donde se copiarán los que se necesiten e instalar los paquetes que falten. Se compila para asegurarse que toda la configuración es correcta, esta primera parte corresponde con el *setup* del entorno de ROS que se describió en el Capítulo 6.

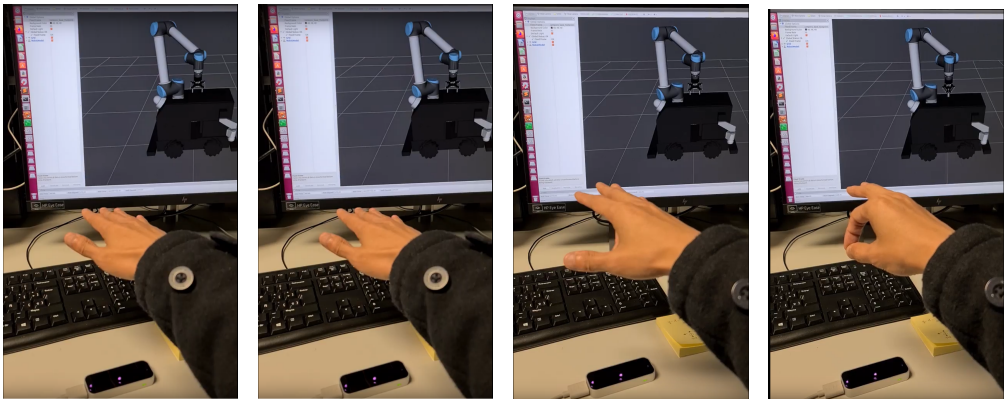
Una vez que se ha realizado el *setup* de sistema correctamente con todo compilado, se procede a crear el directorio donde se copiará la solución deseada, en este caso la solución para un único cobot sin utilizar el paquete MoveIt!. Con esto se tendría la Fase 2 de la figura 7.1, pero se ha saltado la Fase 1, por ello hay que obtener el fichero `URDF` del robot Campero y adaptarlo al contenido de la solución que se quiere desarrollar.

Una vez que se tiene el modelado del robot resuelto, hay que solucionar el control de la pinza en el campero, porque para controlarlo hay que establecer conexión con el servidor de acciones que provee esa funcionalidad. Para ello se mira el paquete de *robotiq* y se configura para que instancie el nodo que se comunica con la pinza y hace de API para el control de la pinza. Con lo anterior arreglado, ya se puede pasar a la Fase 3 de la Figura 7.1, hay que simularlo en Gazebo primero para comprobar que efectivamente todo funciona correctamente antes de probarlo en el robot Campero. La configuración es igual a la que se ha estado desarrollando, por lo que hay que adaptarlo, una vez adaptado, las Fases 4 y 5 de la Figura 7.1 ya están desarrolladas previamente, por ello se puede pasar directamente a realizar las pruebas en Gazebo con el robot Campero simulado y el control de este con Leap Motion, el resultado de las pruebas se muestran en las Figuras 7.2 y 7.3, en las imágenes aparece Rviz porque Gazebo no simulaba correctamente la pinza, pero la simulación es en Gazebo.



(a) Movimiento: Arriba/Abajo (1/2) (b) Movimiento: Arriba/Abajo (2/2) (c) Movimiento: Derecha/Izquierda (1/2) (d) Movimiento: Derecha/Izquierda (2/2)

Figura 7.2: Leap Motion sin Moveit y Campero: Movimientos - Arriba/Abajo y Derecha/Izquierda



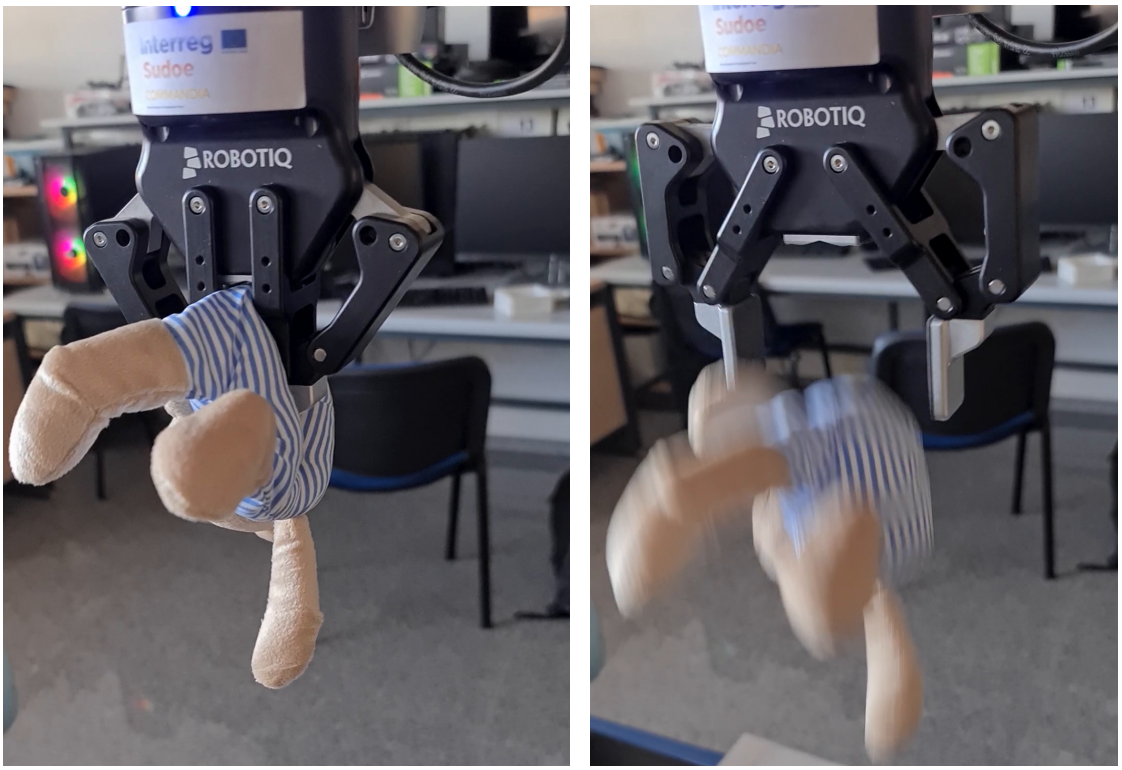
(a) Movimiento: Dentro/Fuera (1/2) (b) Movimiento: Dentro/Fuera (2/2) (c) Movimiento: Pinza Abierta (1/2) (d) Movimiento: Pinza Cerrada (2/2)

Figura 7.3: Leap Motion sin Moveit y Campero: Movimientos - Dentro/Fuera y Pinza Abierta/Cerrada

Una vez que las pruebas realizadas son satisfactorias, se procede a la Fase 6 de la Figura 7.1, hay que configurar el fichero `launch` *bringup*, este fichero es similar al configurado para Gazebo (controladores (incluido la conexión al USB de la pinza), descripción del robot (URDF), ...), pero a diferencia que para Gazebo, para configurar el fichero *bringup* que se comunicará con el robot físico, hay que introducir información importante como la IP del robot Campero, puerto o fichero con la calibración de la cinemática del robot Campero que el simulador de Gazebo no necesita. Y esta configuración es cargado en al driver del paquete *ur_robot_driver*, que es el que hará de interfaz entre el Hardware del robot real Campero y ROS, básicamente se sustituye Gazebo por el driver de *ur_robot_driver* para la interacción con el robot real Campero.

Y finalmente queda probar el funcionamiento en el robot y calibrar la solución desarrollada. Para ello tras lanzar el *bringup*, hay que activar la comunicación con el robot Campero, ya que está en parada de emergencia por defecto. Hay que abrir la aplicación de *VNC Client* que permitirá remover la parada de emergencia (hay que apretar en el mando de seguridad el botón verde también que le permite volver al estado normal al robot) y el script *URcap* que hará de interfaz entre ROS y el robot real.

Primero se realiza pruebas para controlar la pinza como se ve en la Figura 7.4.



(a) Agarra el muñeco (b) Suelta el muñeco

Figura 7.4: Campero: Pruebas de abrir y cerrar la pinza

(a) Coge muñeco (1/3) (b) Coge muñeco (2/3) (c) Coge muñeco (3/3)

Figura 7.5: Campero: Coge y deja muñeco

y finalmente realizar movimientos con el brazo del cobot UR10 en la Figura 7.5,

siempre con el botón de parada de emergencia preparado ya que puede realizar movimientos inesperados en cualquier momento, durante estas pruebas se ha calibrado la velocidad máxima que puede moverse el cobot UR10 a 0.05 rad/s, ya que la velocidad para a simulación era muy rápida.

Capítulo 8

Resultados: simulación en Gazebo y en el robot Campero

Los resultados de las pruebas realizadas con las soluciones con el paquete MoveIt! y sin el paquete, son las mismas. Dado que el objetivo de este Trabajo Fin de Grado es un sistema multirobot, se va a proveer de los resultados de las soluciones para dos y cuatro brazos.

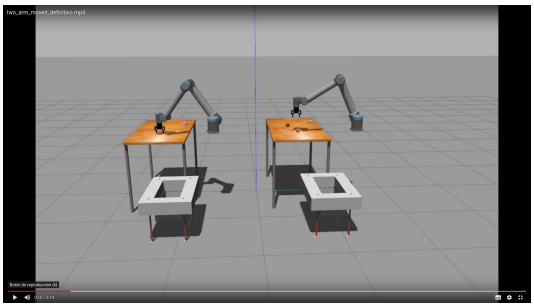
Los resultados se ven mejor en un vídeo que mediante imágenes estáticas, por ello se ha creado una carpeta en *Drive* que contiene el resultado todas las simulaciones realizadas incluidos las realizadas para un único cobot.

El enlace para acceder al *Drive* es el siguiente: `https://drive.google.com/drive/folders/1WP0bFIbLk2onSgx8xH_fq7PJwbzoZK8T?usp=sharing`

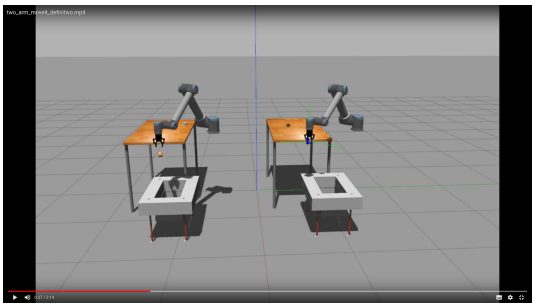
8.1. Pruebas de *pick & place* realizadas en el simulador de Gazebo para dos y cuatro cobots

Ahora se procede a mostrar los resultados al realizar la tarde de *pick & place* utilizado para realizar las pruebas con las soluciones obtenidas. Se puede apreciar un poco de retraso e incluso parecer que va en secuencia en las imágenes de las Figuras 8.1, 8.2, 8.3, 8.4, 8.5 y 8.6, pero la razón es debido a que se han ejecutado de forma secuencial la misma tarea creando un poco de retraso entre ellos lo que da la sensación de que se mueven secuencialmente. Con esto se confirma que el objetivo de controlar varios robots que se mueven de forma simultánea mientras realizan una tarea ha sido exitosa.

– Solución desarrollada *con & sin* el paquete de MoveIt! para dos cobots:

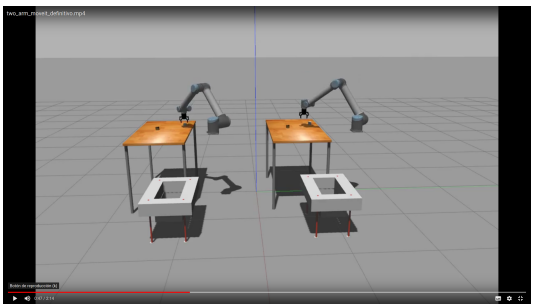


(a) Cogen cubo 1

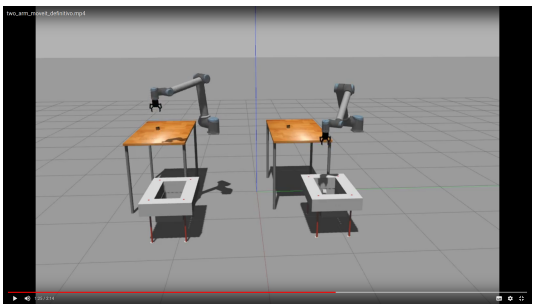


(b) Dejan cubo 1

Figura 8.1: Imágenes del vídeo simulado en Gazebo para dos cobots (1/3)

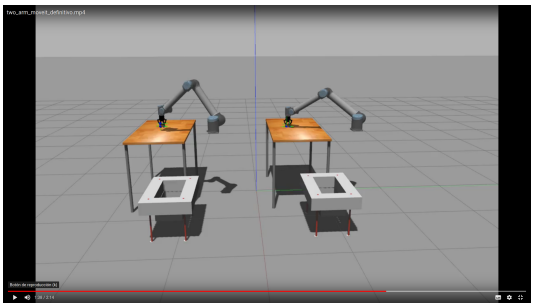


(a) Cogen cubo 2

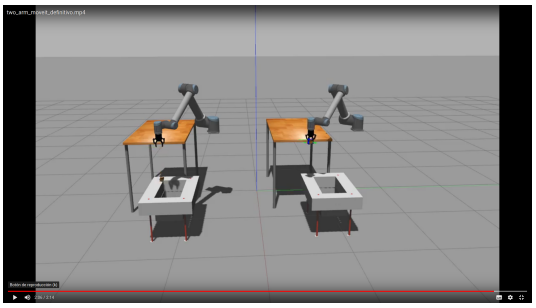


(b) Dejan cubo 2

Figura 8.2: Imágenes del vídeo simulado en Gazebo para dos cobots (2/3)



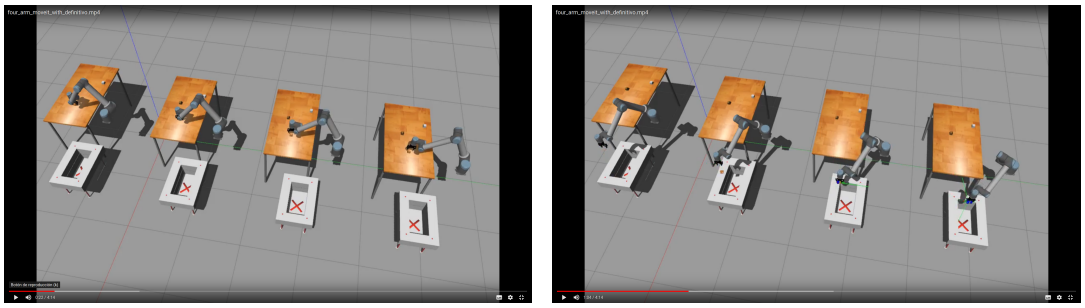
(a) Cogen cubo 3



(b) Dejan cubo 3

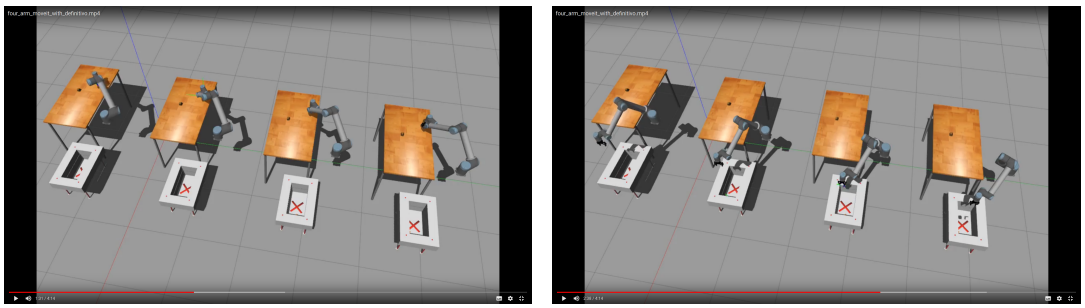
Figura 8.3: Imágenes del vídeo simulado en Gazebo para dos cobots (2/3)

– Solución desarrollada *con & sin* el paquete de MoveIt! para 4 cobots:



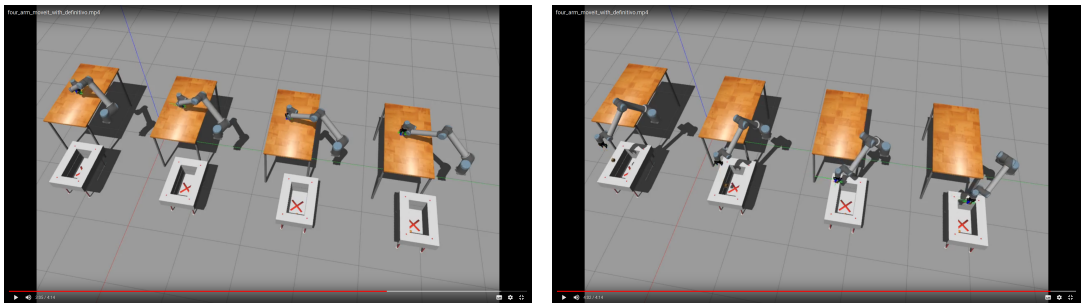
(a) Cogen cubo 1 (b) Dejan cubo 1

Figura 8.4: Imágenes del vídeo simulado en Gazebo para cuatro cobots (1/3)



(a) Cogen cubo 2 (b) Dejan cubo 2

Figura 8.5: Imágenes del vídeo simulado en Gazebo para cuatro cobots (2/3)



(a) Cogen cubo 3 (b) Dejan cubo 3

Figura 8.6: Imágenes del vídeo simulado en Gazebo para cuatro cobots (2/3)

8.2. Pruebas de *Leap Motion* realizadas en el simulador de Gazebo para dos cobots

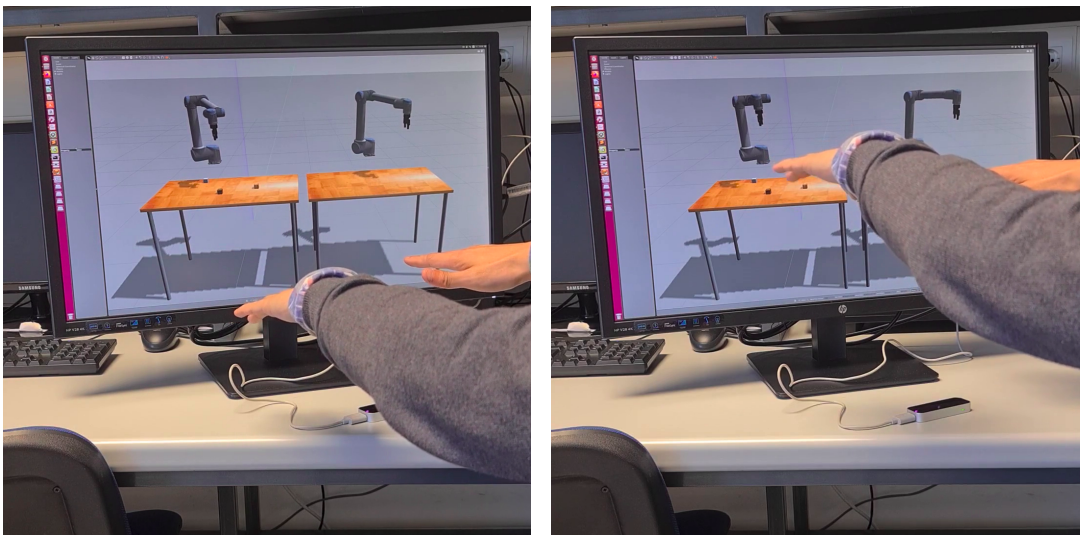
En las pruebas realizadas con Leap Motion y dos cobots en el simulador de Gazebo (las imágenes de las Figuras 8.7, 8.8, 8.9, 8.10, 8.11 y 8.12) hay bastante diferencia entre el controlado con MoveIt! y sin MoveIt! pero ambas simulaciones realizan los movimientos exitosamente. En el vídeo de la simulación, se aprecia que la solución sin MoveIt! sigue mejor los movimientos de la mano, las razones pueden ser diversas, algunas de ellas pueden ser:

- La configuración del Planning de MoveIt!, si se requiere mucha precisión, esto puede afectar al tiempo que tarde en obtener los valores de las trayectorias lo que produce en una retraso en los movimientos del cobot.

- Que MoveIt! no permita realizar el cálculo de otra trayectoria y enviarla hasta que no se haya ejecutado la anterior completamente, lo que hace que pierda movimientos de la mano mientras está ejecutando una trayectoria.

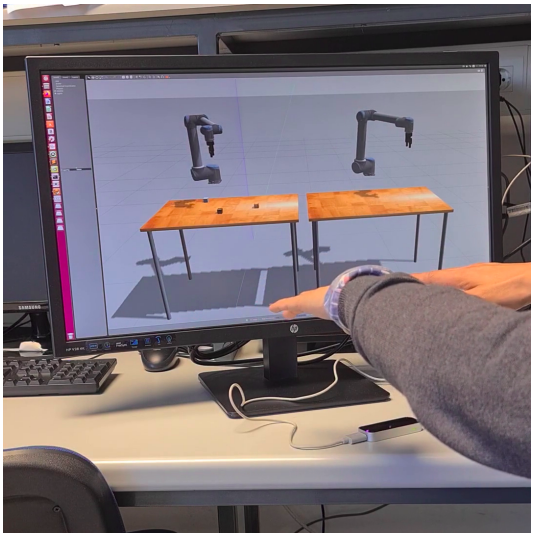
La primera razón es la más probable, realmente no se requiere de mucha precisión durante la planificación lo que le permitiría seguir mejora la mano.

- Solución desarrollada *con* el paquete de MoveIt! para dos cobots controlado mediante Leap Motion:

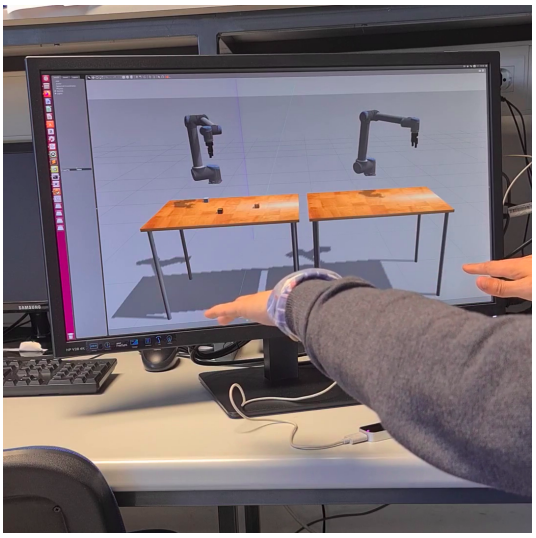


(a) Movimiento: Arriba/Abajo (1/2) (b) Movimiento: Arriba/Abajo (2/2)

Figura 8.7: Leap Motion con Moveit: Movimiento - Arriba/Abajo

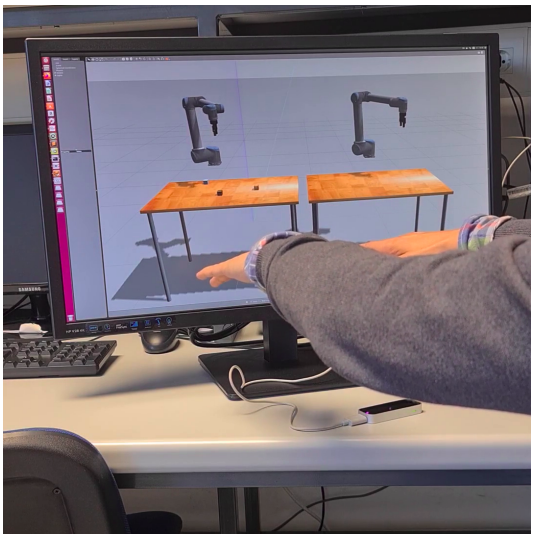


(a) Movimiento: Derecha/Izquierda (1/2)

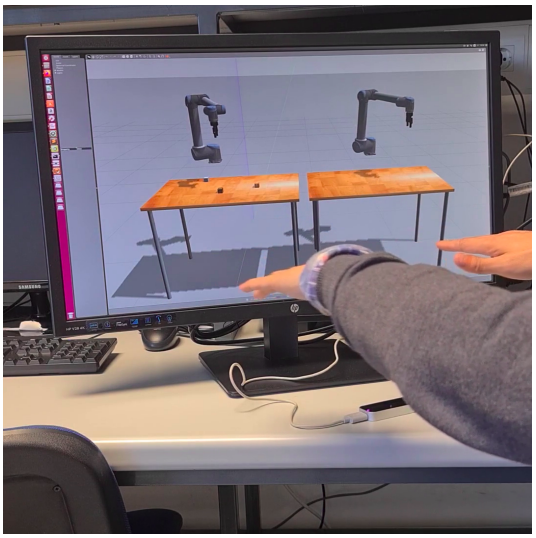


(b) Movimiento: Derecha/Izquierda (2/2)

Figura 8.8: Leap Motion con Moveit: Movimiento - Derecha/Izquierda



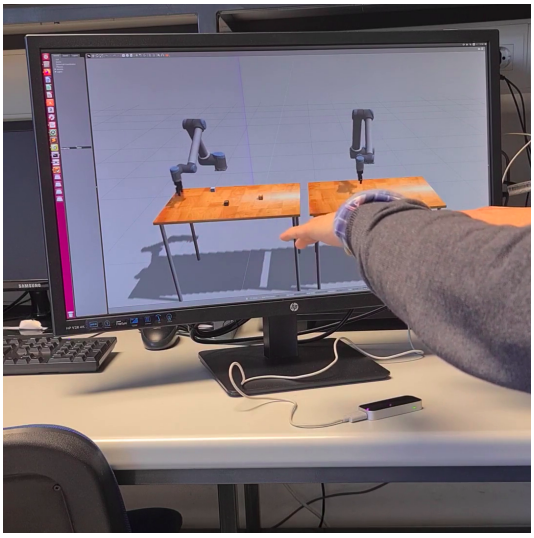
(a) Movimiento: Dentro/Fuera (1/2)



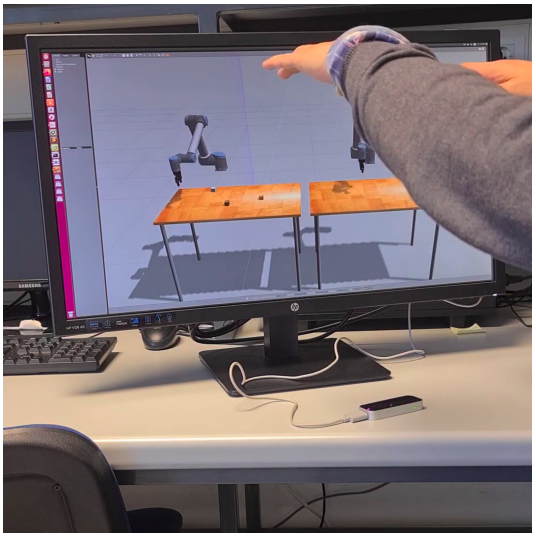
(b) Movimiento: Dentro/Fuera (2/2)

Figura 8.9: Leap Motion con Moveit: Movimiento - Dentro/Fuera

– Solución desarrollada *sin* el paquete de MoveIt! para dos cobots controlado mediante Leap Motion:

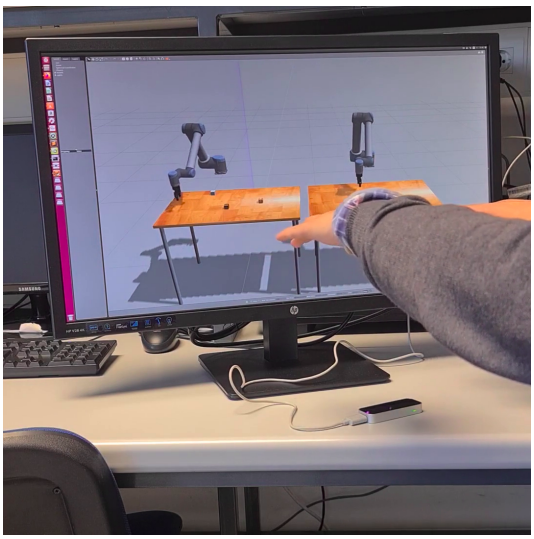


(a) Movimiento: Arriba/Abajo (1/2)

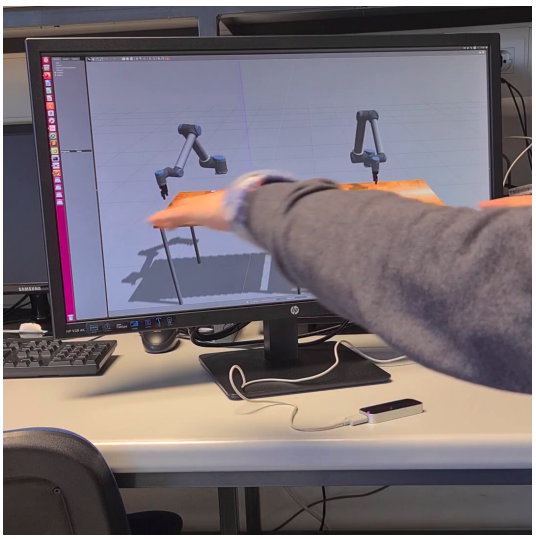


(b) Movimiento: Arriba/Abajo (2/2)

Figura 8.10: Leap Motion sin Moveit: Movimiento - Arriba/Abajo

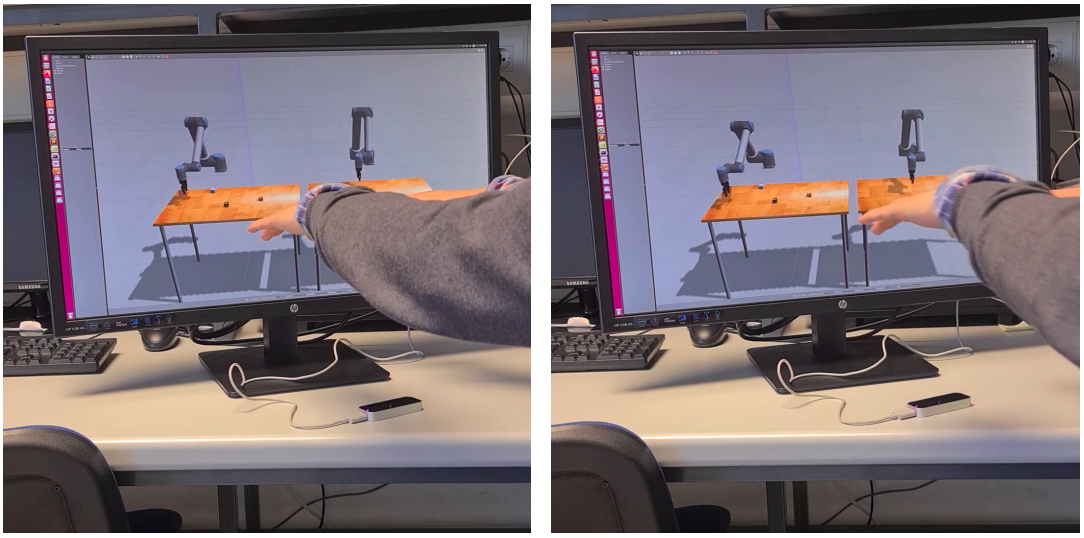


(a) Movimiento: Derecha/Izquierda (1/2)



(b) Movimiento: Derecha/Izquierda (2/2)

Figura 8.11: Leap Motion sin Moveit: Movimiento - Derecha/Izquierda



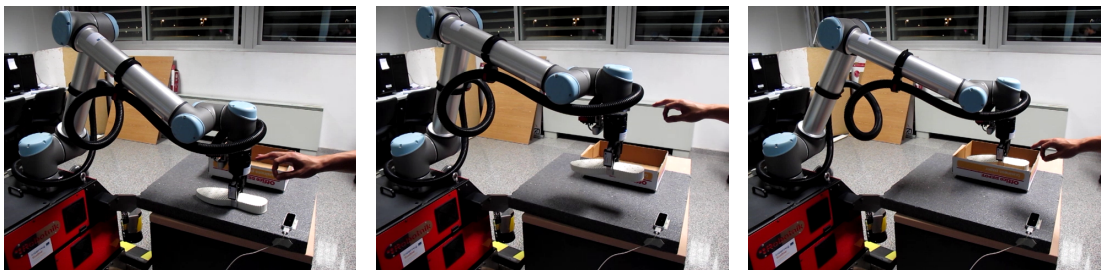
(a) Movimiento: Dentro/Fuera (1/2) (b) Movimiento: Dentro/Fuera (2/2)

Figura 8.12: Leap Motion sin Moveit: Movimiento - Dentro/Fuera

8.3. Pruebas de *Leap Motion* realizadas en el robot Campero

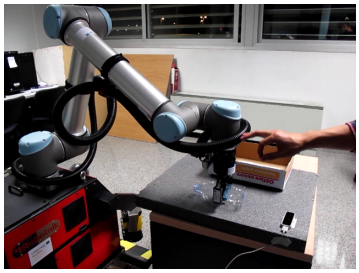
En las Figuras 8.13, 8.14 y 8.15, se muestra el *pick & place* realizado mediante Leap Motion con el brazo UR10 del robot Campero. Sin tener en cuenta los problemas que podría dar expuestas durante el desarrollo de la interfaz de Leap Motion (Diseño de la interfaz de Leap Motion), las pruebas en el robot Campero han sido realizadas con éxito.

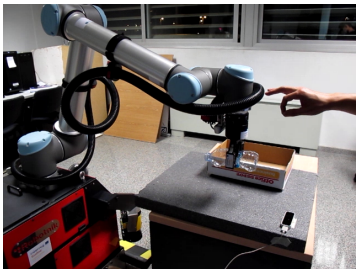
– Solución desarrollada *con* el paquete de MoveIt! para dos cobots controlado mediante Leap Motion:



(a) Coge suela (1/3) (b) Coge suela (2/3) (c) Coge suela (3/3)

Figura 8.13: Campero: Coge y deja suela


(a) Coge botella (1/3)


(b) Coge botella (2/3)

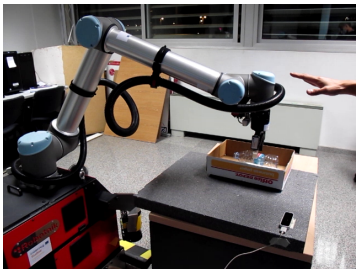
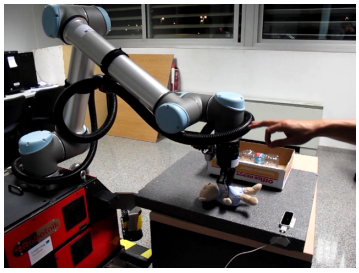

(c) Coge botella (3/3)

Figura 8.14: Campero: Coge y deja botella


(a) Coge muñeco (1/3)


(b) Coge muñeco (2/3)

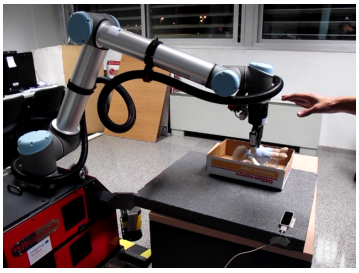

(c) Coge muñeco (3/3)

Figura 8.15: Campero: Coge y deja muñeco

Capítulo 9

Conclusiones y trabajo futuro

9.1. Conclusiones

Este Trabajo Fin de Grado demuestra de que es posible el control de múltiples robots simultáneamente haciendo diferentes tareas en el entorno de ROS. Se ha tenido muchas dificultades para configurar adecuadamente MoveIt! para su correcto funcionamiento con varios robots, tanto es así que se ha desarrollado una solución alternativa sin MoveIt! que también cumple con todos los requisitos.

Todos los objetivos propuestos al inicio del proyecto se han cumplido:

- Durante el proceso, se ha familiarizado con el entorno de ROS, mediante la creación, documentación y pruebas hasta llegar a las soluciones finales propuestas. No solo se ha familiarizado con ROS y sus herramientas, sino también con una cantidad importante de paquetes de terceros como los de MoveIt!, Universal Robots, Robotiq, Gazebo y Leap Motion principalmente. Todos los scripts creados han sido íntegramente realizados en el lenguaje de programación de Python 2.7.

- Se ha hecho varias propuestas de diseño como solución, concretamente tres con las herramientas del que se tiene conocimiento, se ha efectuado un análisis de cada uno de ellos con sus ventajas e inconvenientes, tanto en el aspecto actual como su posible integración en trabajos futuros.

- Durante la realización de pruebas en el robot real, se descubrió dónde estaba el problema en la *solución desarrolla mediante MoveIt!* cuando la *solución con un planificador propio* ya estaba resuelto. Por las grandes ventajas que provee MoveIt! se ha decidido hacer la documentación del desarrollo de ambas soluciones y han pasado por todas las etapas de desarrollo establecidas al inicio del proyecto menos probarlo en el robot real.

- Se han desarrollado ambas soluciones propuestas para uno, dos y 4 robots colaborativos, es decir, para N robots. Cabe añadir que las soluciones permiten

el control simultáneo de N robots diferentes, no tiene por qué ser únicamente de Universal Robots.

- Se ha incorporado Leap Motion para el control de hasta dos robots así como el diseño y reconocimiento de los gestos que controlarán simultáneamente hasta dos robots para ambas soluciones.
- En el desarrollo de la *solución con un planificador propio*, se ha implementado una librería en Python para calcular la cinemática directa y la cinemática inversa. Se han creado nodos desde cero para la obtención y transmisión de órdenes a los controladores de ROS (estás transmisiones son directas, es decir se ha tratado el formato del mensaje manualmente) que controlan la pinza y el brazo del UR10. Y también una documentación del paquete *tf* de ROS para poder realizar la implementación de un nodo que esté siempre publicando la posición actual del *end-effector*.
- Se han desarrollado scripts de Python para las dos soluciones, la tarea de *pick & place* para uno, dos y cuatro robots, demostrando que el sistema puede escalar hasta N robots ejecutando la misma o diferentes tareas.
- Se ha configurado, calibrado y efectuado pruebas en el robot físico (Campero) del laboratorio para la *solución con un planificador propio*, ejecutando la tarea de *pick & place*, controlado por el dispositivo externo Leap Motion.
- Finalmente, se ha creado en GitHub dos repositorios con el código fuente y documentación para la reproducción del trabajo hecho.

Se ha encontrado poca documentación relacionada con la configuración de MoveIt! para el control de varios robots. Dejan caer la idea de que hay que hacerlo mediante el uso de *namespaces* o que realices un control directo estos como se ha desarrollado en la *solución con planificador propio*, pero no se ha encontrado ni un ejemplo, ni siquiera en la documentación de MoveIt! que a veces es incluso confusa. Debido a esto una gran parte del tiempo dedicado durante el desarrollo del trabajo ha sido dedicado a la documentación y la realización de pruebas durante la configuración del sistema.

Finalmente como opinión personal, la robótica es una de las ramas más importantes dentro de las aplicaciones industriales y los robots colaborativos son una de sus incorporaciones estrella. Su ámbito de trabajo no se limita únicamente en la industria debido a que puede trabajar junto a las personas. Y esta característica de trabajo puede permitir su inclusión en áreas que requieran el control de múltiples robots controlados simultáneamente, alejado de la industria.

9.2. Trabajo futuro

Hay muchas opciones para trabajo futuro, pero siguiendo en la línea de lo desarrollado se proponen algunas de ellas:

- Mejora del planificador mediante inteligencia artificial, mediante la predicción de posibles movimientos, al tenerlos pre-procesados se puede mover más rápido, así como mapear las trayectorias que ya se han calculado previamente.
- Incorporación de nuevas interfaces para la comunicación o control de los robots.
- Desarrollo de robots móviles con dos o más cobots para realizar tareas que serían complejas con un único cobot.
- Mejora del reconocimiento de los gestos de las manos mediante inteligencia artificial para un mejor control de los movimientos del cobot.
- Solventar el problema de que el cobot no sea consciente del movimiento de otros cobots, una posible solución en MoveIt! es configurando adecuadamente el fichero `planning_context.launch` teniendo en cuenta los *namespacings* y cómo el parámetro del servidor `robot_description` está definido y utilizado el sistema. Esta solución permitiría efectuar movimientos más complejos, como son movimientos en donde se cruzan o se entrelazan sin la necesidad de dispositivos de reconocimiento.

Capítulo 10

Bibliografía

[1] "Rb-eken - robot manipulador para la industria: Robotnik®." [Online]. Available: https://robotnik.eu/es/productos/manipuladores-moviles/rb-eken/

[2] "Commandia." [Online]. Available: http://commandia.unizar.es/es/lo-basico-de-commandia/

[3] "Industria 4.0 la cuarta revolución industrial inteligente," https://www.cic.es/industria-40-revolucion-industrial/, Jan. 2017, accessed: 2021-11-3.

[4] K. V. B. Del Villar, "Avanzamos hacia la industria 5.0 - IV congreso de industria conectada 4.0," https://cic40.es/cic40/avanzamos-hacia-la-industria-5-0/, Sep. 2021, accessed: 2021-11-3.

[5] J. Ochoa and Universidad de Sonora, "LA CUARTA REVOLUCIÓN INDUSTRIAL," *rvcs*, no. 2, pp. 113–114, 2017.

[6] Wikipedia contributors, "Robot operating system," https://es.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=137501462, accessed: 2021-11-3.

[7] P. Trelis Molina, "Robótica 5g para telemedicina: Control remoto en tiempo real de un brazo colaborativo mediante un háptico para el telediagnóstico clínico." Ph.D. dissertation, Universitat Politècnica de València, 2021.

[8] S. J. Tosco, F. Corteggiano, and M. Broll, "Implementación de un esquema de teleoperación utilizando el sistema operativo ros en el contexto de un laboratorio remoto," *Mecánica Computacional*, vol. 31, no. 23, pp. 3741–3749, 2012.

[9] D. B. León, "Control teleoperado de un sistema multi-robot para aplicaciones quirúrgicas," *Control teleoperado de un sistema multi-robot para aplicaciones*

quirúrgicas, 2020. [Online]. Available: https://zaguan.unizar.es/record/96635/files/TAZ-TFG-2020-1629.pdf

[10] U. Robots, "Robots colaborativos: Universal robots." [Online]. Available: https://www.universal-robots.com/es/

[11] "The birth of mimic: Human skills, machine precision," Mar 2021. [Online]. Available: https://nordbo-robotics.com/the-birth-of-mimic-human-skills-machine-precision/

[12] "Mimic kit for ur npr-r50-ur," Nov 2021. [Online]. Available: https://unchainedrobotics.de/en/products/teaching-kits-en/mimic-kit-for-ur-npr-r50-ur/

[13] G. Almeida, "Unidad 1: Fundamentos generales de la robotica," Universidad técnica de Ambato, Tech. Rep., 2009.

[14] J. R. de Garibay Pascual, "Robótica: Estado del arte," *Universidad de Deuston. Número. Fecha*, p. 54, 2006.

[15] A. C. Simões, A. L. Soares, and A. C. Barros, "Factors influencing the intention of managers to adopt collaborative robots (cobots) in manufacturing organizations," *Journal of Engineering and Technology Management*, vol. 57, p. 101574, 2020.

[16] M. Li, A. Milojević, and H. Handroos, "Robotics in manufacturing—the past and the present," in *Technical, Economic and Societal Effects of Manufacturing 4.0*. Palgrave Macmillan, Cham, 2020, pp. 85–95.

[17] F. Reyes, *Robótica-control de robots manipuladores*. Alfaomega grupo editor, 2011.

[18] Wikipedia contributors, "Historia de los robots," https://es.wikipedia.org/w/index.php?title=Historia_de_los_robots&oldid=138927569, accessed: 2021-11-8.

[19] ——, "Cobot," https://es.wikipedia.org/w/index.php?title=Cobot&oldid=134246375, accessed: 2021-11-8.

[20] F. Sherwani, M. M. Asad, and B. Ibrahim, "Collaborative robots and industrial revolution 4.0 (ir 4.0)," in *2020 International Conference on Emerging Trends in Smart Technologies (ICETST)*, 2020, pp. 1–5.

[21] A. Weiss, A.-K. Wortmeier, and B. Kubicek, "Cobots in industry 4.0: A roadmap for future practice studies on human–robot collaboration," *IEEE Transactions on Human-Machine Systems*, vol. 51, no. 4, pp. 335–345, 2021.

[22] K. Sanchez Madriz, "Clasificación de los robots según su arquitectura - el avance de la robótica," Oct 2011. [Online]. Available: https://sites.google.com/site/elavancedelarobotica/clasificacion-de-los-robots/clasificacion-de-los-robots-su

[23] Ad, "Robot humanoide erica creado en japón," Dec 2019. [Online]. Available: https://www.robotsjaponeses.net/robot-humanoide-erica-feminina-japon/

[24] "Boston dynamics pone sus robots al servicio de los sanitarios que luchan contra el coronavirus," Apr 2020. [Online]. Available: https://www.niusdiario.es/ciencia-y-tecnologia/tecnologia/coronavirus-spot-robot-ayudar-sanitarios_18_2935545259.html

[25] Robotnik, "Robots colaborativos vs robots industriales, ¿qué necesita mi empresa?" https://www.interempresas.net/Robotica-industrial/Articulos/346419-Robots-colaborativos-vs-robots-industriales-que-necesita-mi-empresa.html, 29 2021, accessed: 2021-11-3.

[26] Wikipedia, "Interacción persona-computadora — wikipedia, la enciclopedia libre," 2021, [Internet; descargado 15-enero-2022]. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Interacci%C3%B3n_persona-computadora&oldid=140367503

[27] M. Ribera Turró, "Evolución y tendencias en la interacción persona–ordenador," *El profesional de la información*, vol. 15, no. 6, pp. 414–422, Noviembre - Diciembre 2005.

[28] "Programa interreg sudoe - robótica móvil colaborativa de objetos deformables en aplicaciones industriales," Abril 2018. [Online]. Available: https://www.interreg-sudoe.eu/proyectos/los-proyectos-aprobados/188-robotica-movil-colaborativa-de-objetos-deformables-en-aplicaciones-industriales

[29] "Api overview." [Online]. Available: https://developer-archive.leapmotion.com/documentation/python/devguide/Leap_Overview.html

[30] "Robot operating system." [Online]. Available: https://www.ros.org/

[31] "Wiki." [Online]. Available: https://wiki.ros.org/

[32] P. Kaveti and H. Singh, "Ros rescue : Fault tolerance system for robot operating system," Oct 2019. [Online]. Available: https://arxiv.org/abs/1910.01078

[33] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Engineering adaptive fault-tolerance mechanisms for resilient computing on ros," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016, pp. 94–101.

[34] "Moving robots into the future." [Online]. Available: https://moveit.ros.org/

[35] Osrf, "Why gazebo?" [Online]. Available: http://gazebosim.org/

[36] A. Cain, T. Furutani, and R. Nagpal, "A platform to learn ros-based advanced robotics online," Nov 2021. [Online]. Available: https://www.theconstructsim.com/

Capítulo 11

Bibliografía de Imágenes

[37] H. GmbH, "Eje lineal eléctrico by hiwin gmbh: Directindustry." [Online]. Available: https://www.directindustry.es/prod/hiwin-gmbh/product-14370-2419274.html

[38] Novologística, "Omron desarrolla su robot móvil ld-250," Nov 2019. [Online]. Available: https://novologistica.com/manutencion-y-almacenaje/omron-desarrolla-su-robot-movil-ld-250/

[39] D. Barrera, "Robot campero," http://webdiis.unizar.es/~raragues/wp/wp-content/uploads/davidBarrera_Campero2-300x285.png, 2020, accessed: 2021-11-8.

[40] K. Jackson, "What is the difference between a robot and a cobot?" https://www.alliedelec.com/expert/difference-between-robot-and-cobot, Jun. 2019, accessed: 2021-11-3.

[41] Robotshop, "Explore virtual reality with leap motion 3d motion controller." [Online]. Available: https://www.robotshop.com/community/blog/show/explore-virtual-reality-with-leap-motion-3d-motion-controller

[42] [Online]. Available: https://cdn.dribbble.com/users/450489/screenshots/6053160/bahur78-apple-watch-travel-app_4x.png?compress=1&resize=400x300

[43] bsd555, "Mute call smartphone interface vector template. mobile app page..." [Online]. Available: https://www.istockphoto.com/es/vector/silenciar-plantilla-vectorial-de-interfaz-de-tel%C3%A9fono-inteligente-de-llamada-gm1185653916-334221969

[44] [Online]. Available: https://cdn.dribbble.com/users/1234549/screenshots/3200527/drib.jpg

[45] Wikipedia, "Windows 10 — wikipedia, la enciclopedia libre," 2022, [Internet; descargado 16-enero-2022]. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Windows_10&oldid=140891046

[46] [Online]. Available: https://images.ctfassets.net/freurdme1ae3/67HOeZ66Ec0ZiI2ZjbLM9V/45f3c34b99e2e4e23a649f308cff1d63/maxresdefault.jpeg?w=2048&h=1152&q=80&fit=fill&f=face

[47] [Online]. Available: https://i2.wp.com/www.affinityvr.com/wp-content/uploads/2019/11/642895-oculus-rift-s.jpg

[48] [Online]. Available: https://qph.fs.quoracdn.net/main-qimg-dd18f01fffcd88f5bfc61ac53f96794a-lq

[49] S. P. Library, "Early soviet computer, 1967 - stock image - c048/7680." [Online]. Available: https://www.sciencephoto.com/media/1104340/view/early-soviet-computer-1967

[50] L. Yusuf, "Ubiquitous computing in the jet age: its characteristics and challenges - scientific figure on researchgate." [Online]. Available: https://www.researchgate.net/figure/Ubiquitous-Computing-Environment-for-the-Future_fig3_259648636

[51] Ultraleap, "Tracking: Leap motion controller." [Online]. Available: https://www.ultraleap.com/product/leap-motion-controller/

[52] "Getting started with the leap motion sdk," Aug 2015. [Online]. Available: https://blog.leapmotion.com/getting-started-leap-motion-sdk/

[53] "Inicie la producción más rápido." [Online]. Available: https://robotiq.com/es/

Lista de Figuras

Lista de Códigos fuentes

Anexos A

Representaciones URDF

Anexo con las representaciones de los ficheros URDF que componen el modelo del robot usado en el desarrollo de las soluciones del Trabajo Fin de Grado.

A.1. Representación del fichero URDF del Gripper

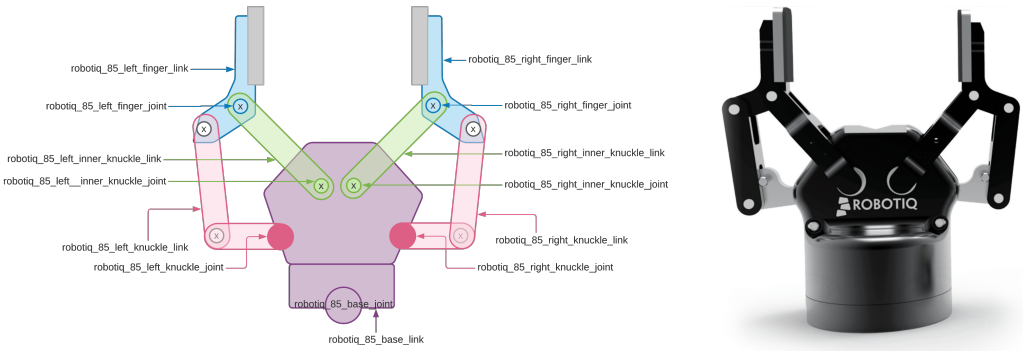


Figura A.1: Fichero `robotiq_85_gripper.urdf.xacro`

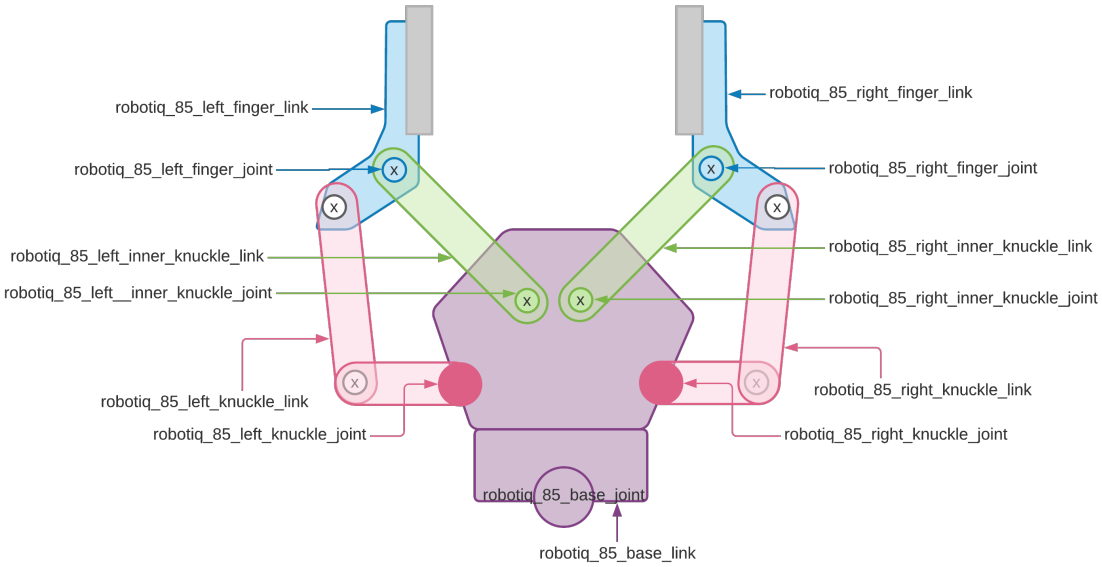


Figura A.2: Fichero `robotiq_85_gripper.urdf.xacro`

A.2. Representación del fichero URDF de un único UR10

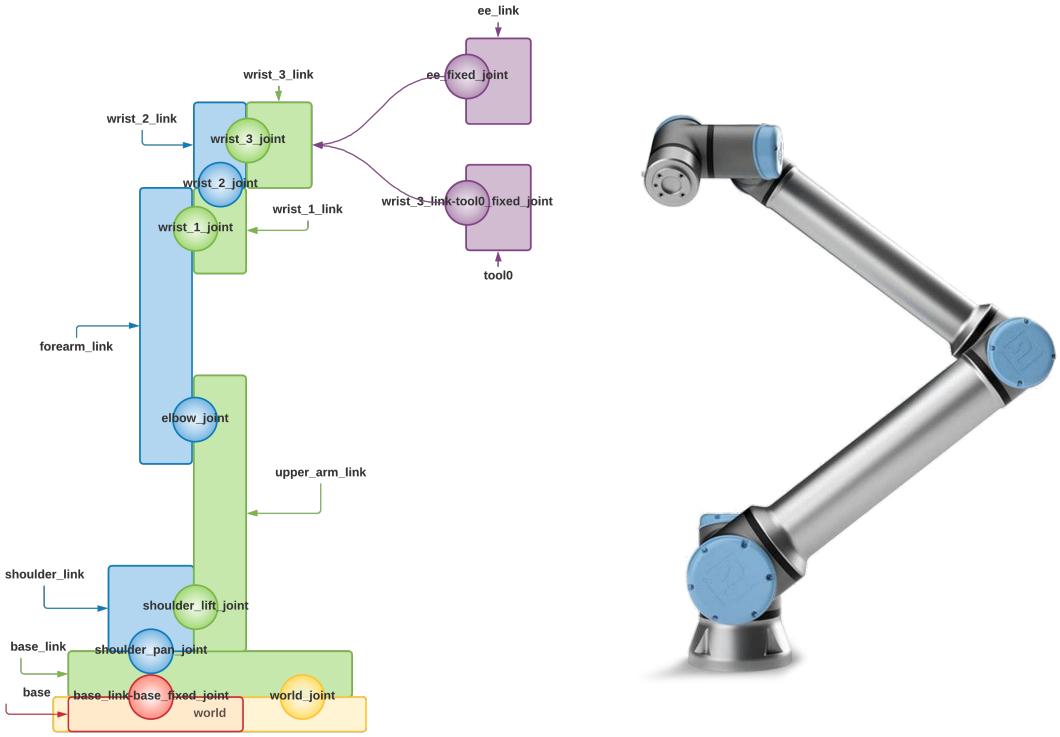


Figura A.3: Fichero `ur10.urdf.xacro`

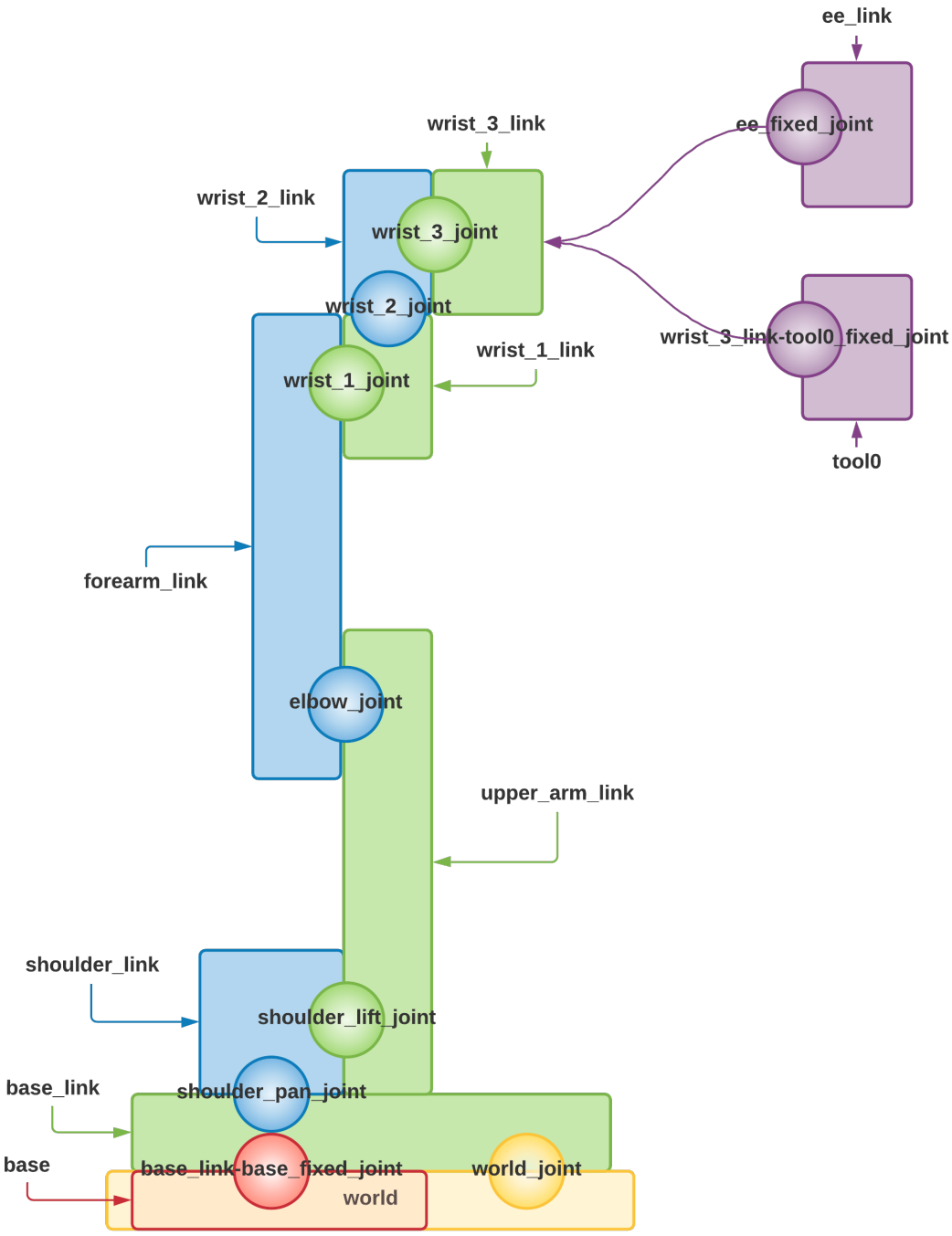


Figura A.4: Fichero `ur10.urdf.xacro`

A.3. Representación del fichero URDF de un único UR10 y gripper

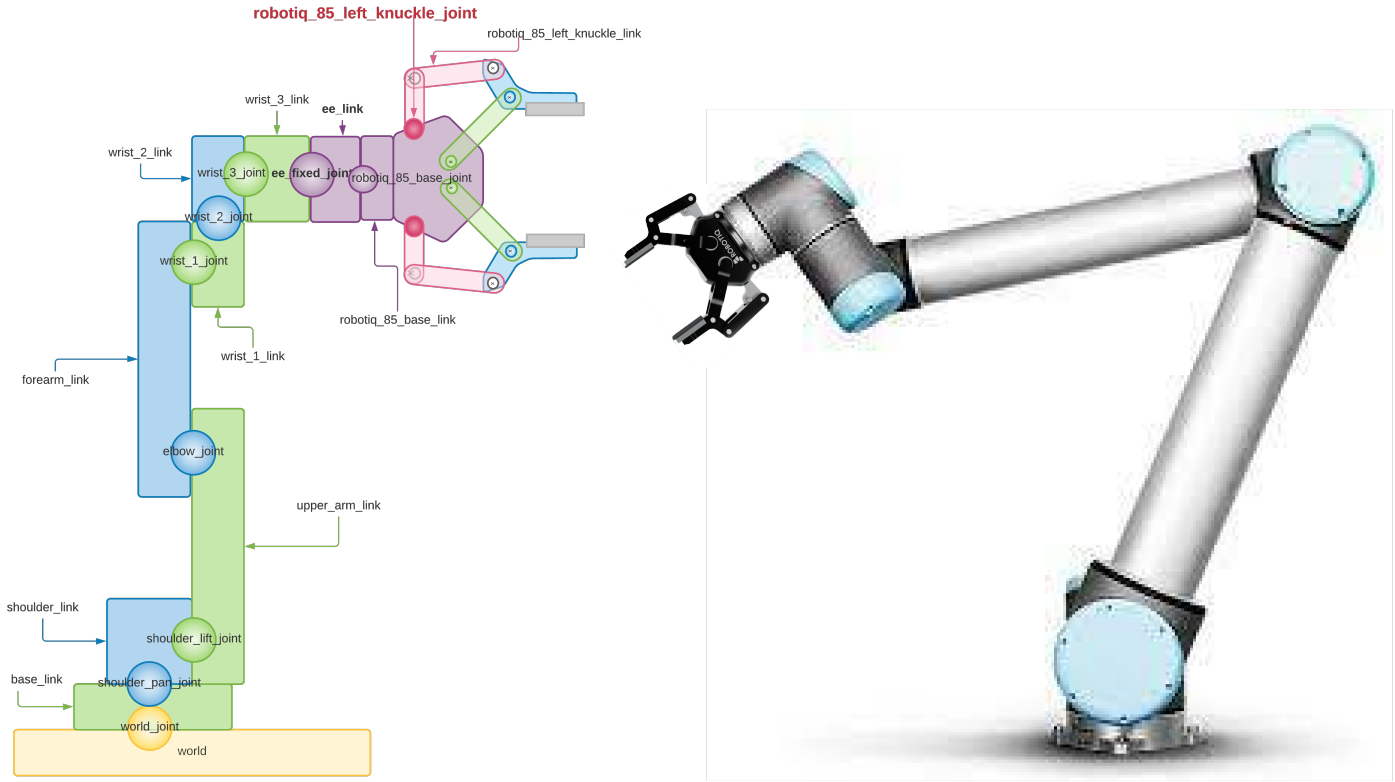


Figura A.5: Fichero `ur10_robot.urdf.xacro`

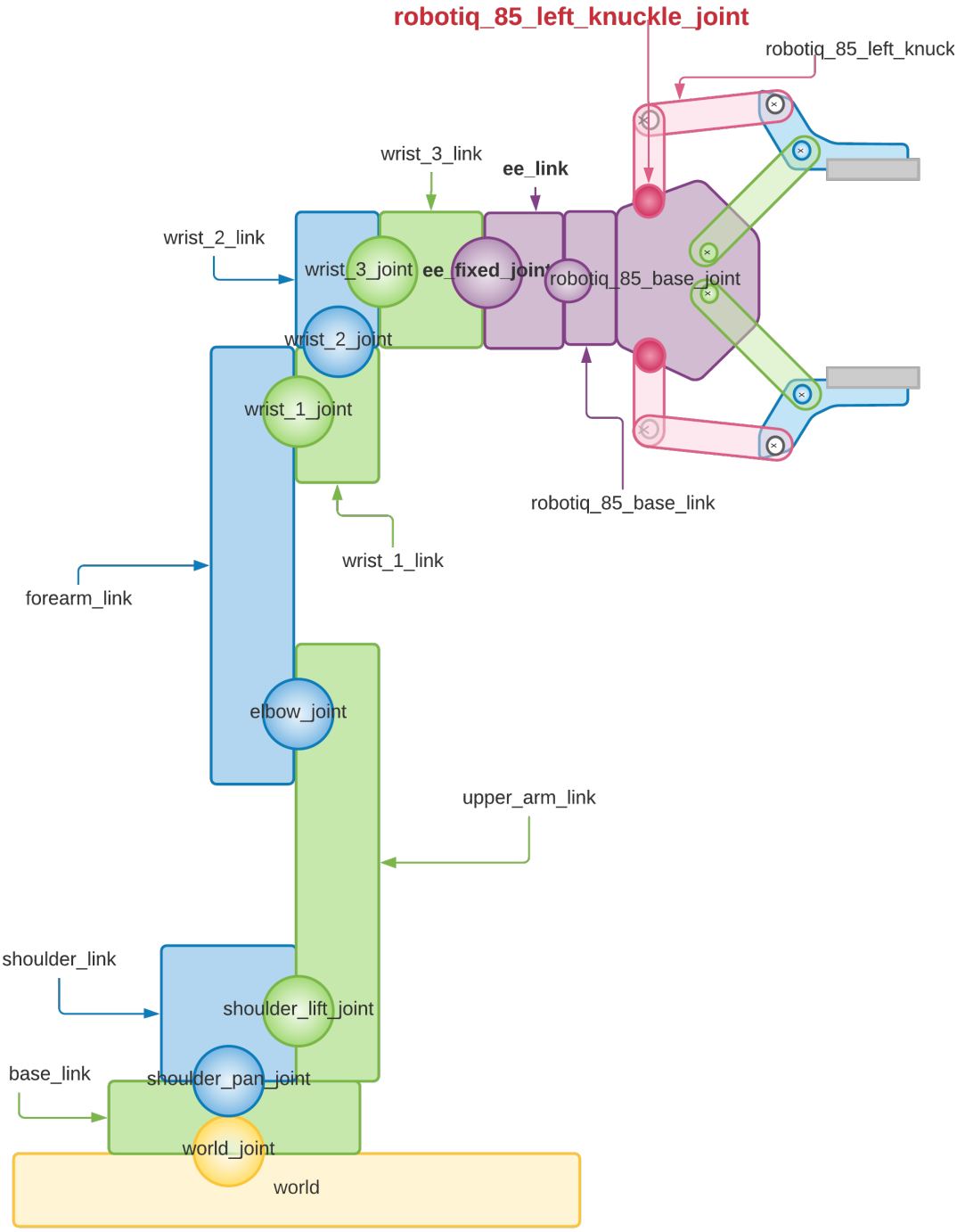


Figura A.6: Fichero `ur10_robot.urdf.xacro`

A.4. Representación del fichero URDF de dos UR10s y grippers

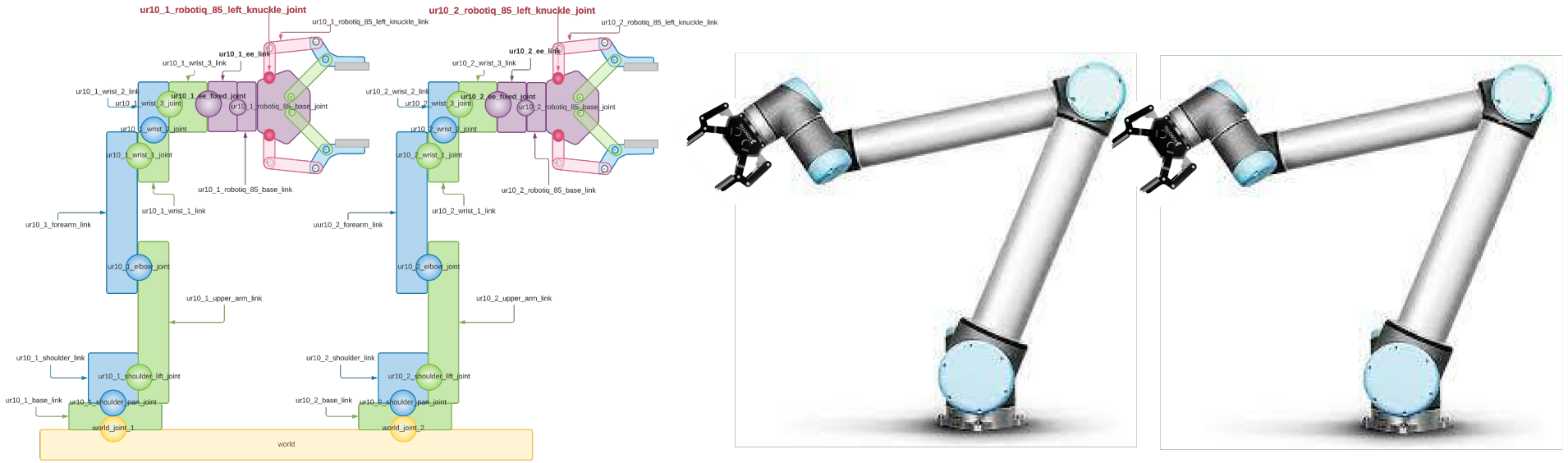


Figura A.7: Fichero `ur10_robot.urdf.xacro`

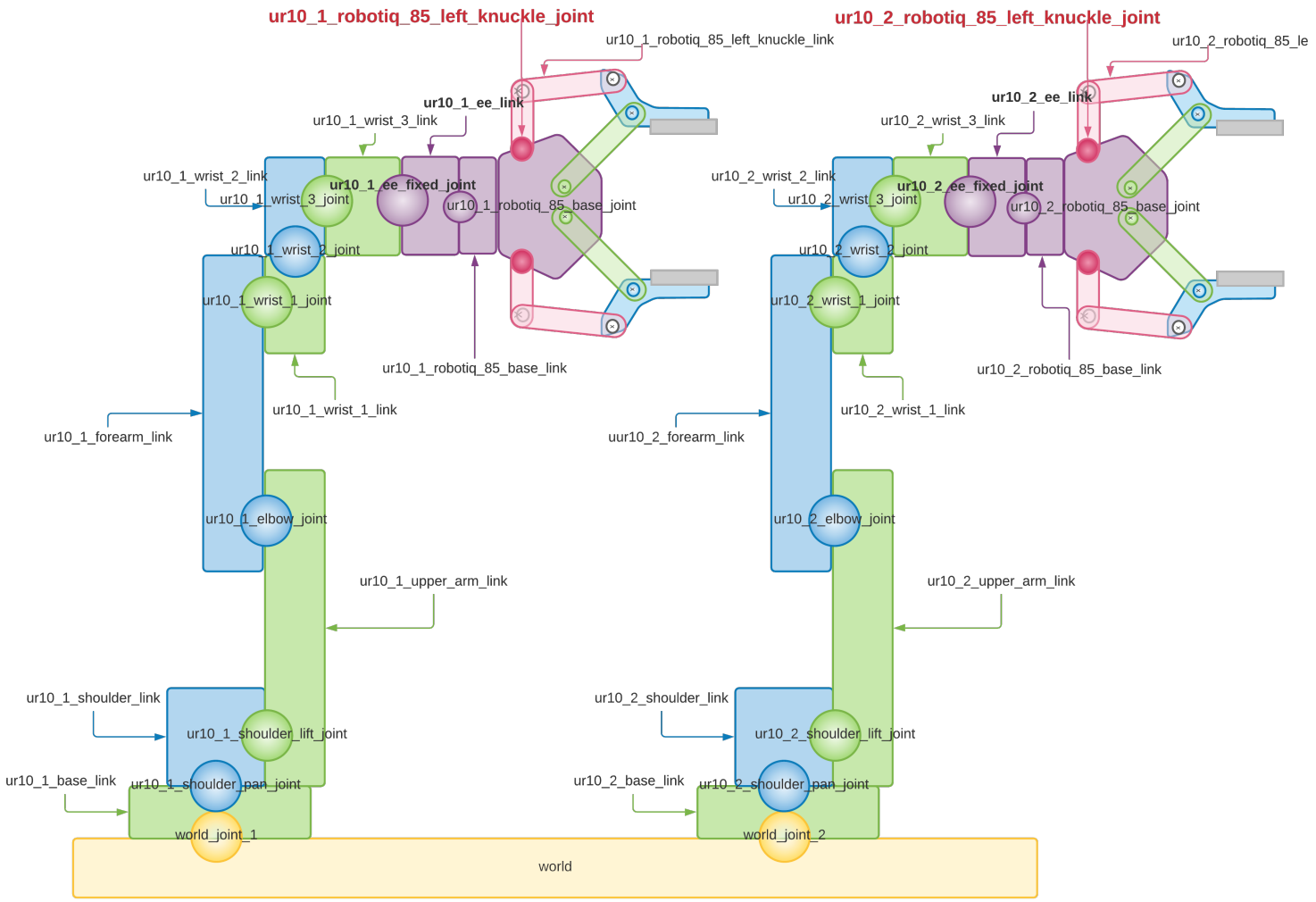


Figura A.8: Fichero `ur10_robot.urdf.xacro`

Anexos B

Moveit! Setup Assistant: Configuración para un cobot

Antes de realizar la configuración con el Setup Assitant hay que tener el URDF bien definido previamente, con ese hecho se lanza el asistente de configuración con el siguiente comando en la terminal:

```
$ mkdir one_arm_moveit_config
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

Código Fuente B.1: Preparación del directorio que almacenará la configuración de moveit y lanzar el Setup Assistant

Se va a escoger como modelo del robot el fichero URDF: `ur10_joint_limited_robot.urdf.xacro` (podría ser perfectamente `ur10_robot.urdf.xacro`).

– Comenzando la configuración del *Setup Assistant*, hay que decirle donde está el fichero `ur10_joint_limited_robot.urdf.xacro`, es decir, el modelo del robot que se quiere configurar:

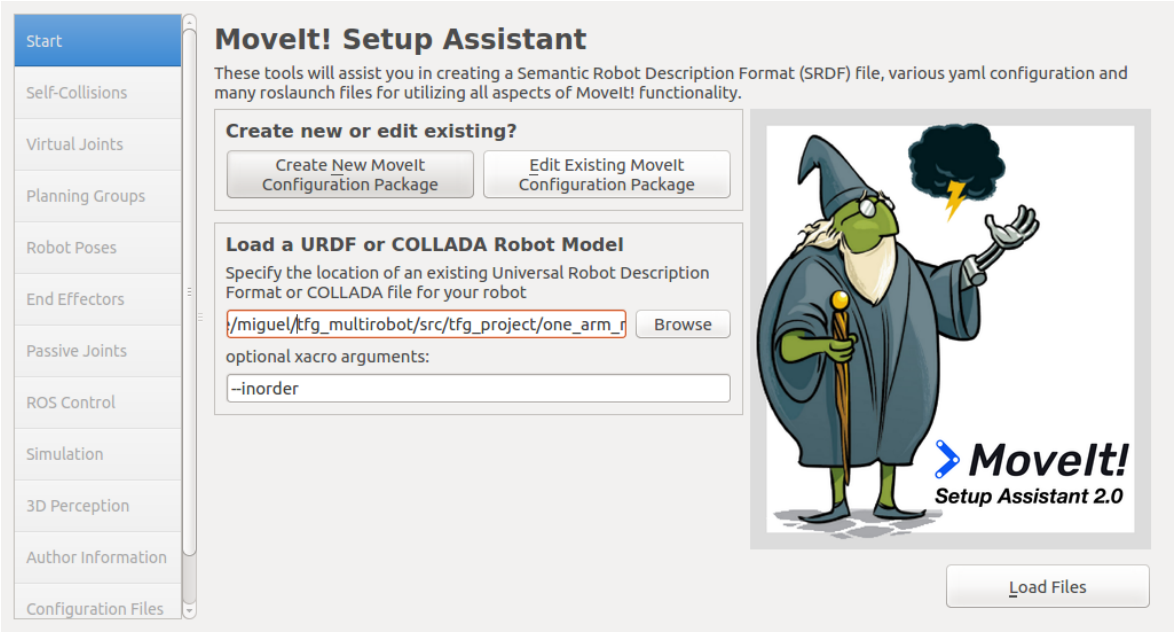


Figura B.1: Cargar el modelo URDF del robot UR10

– Posteriormente, se le da al botón *Load Files*.

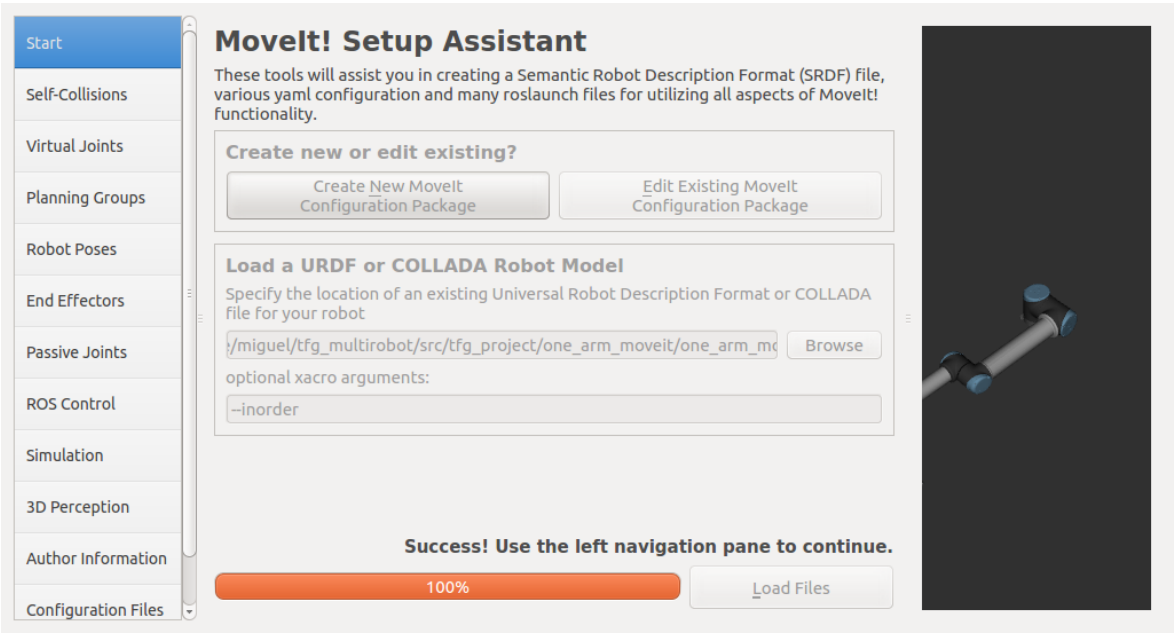


Figura B.2: Cargado el modelo URDF del robot UR10

– En la pestaña *Self-Collisions*, darle al botón *Generate Collision Matrix*, lo que generará una matriz entre los diferentes componentes del robot que puedan generar autocolisiones durante la planificación de las trayectorias:

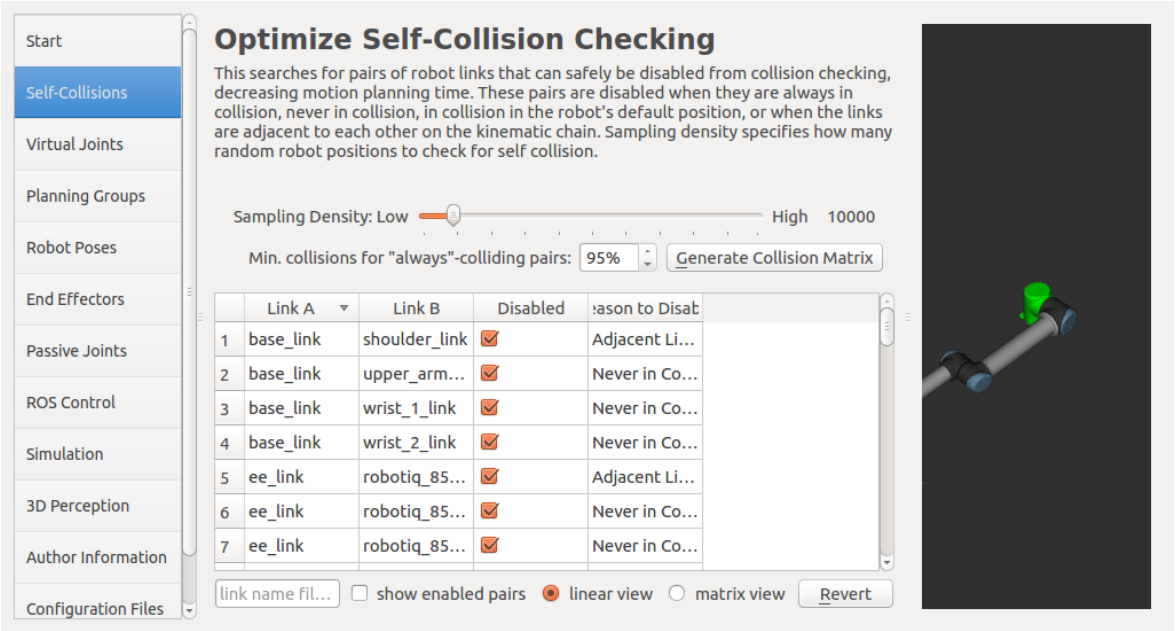


Figura B.3: Generación de la matriz de colisiones

– En la pestaña *Virtual Joints* está dirigida especialmente para brazos robóticos instalados sobre una base móvil, en este caso no afecta en la configuración, ya que la base es fija, pero se configurará igualmente para configuraciones futuras, hay que crear un joint entre el robot *base_link* y el *frame world*, siendo la configuración la siguiente:

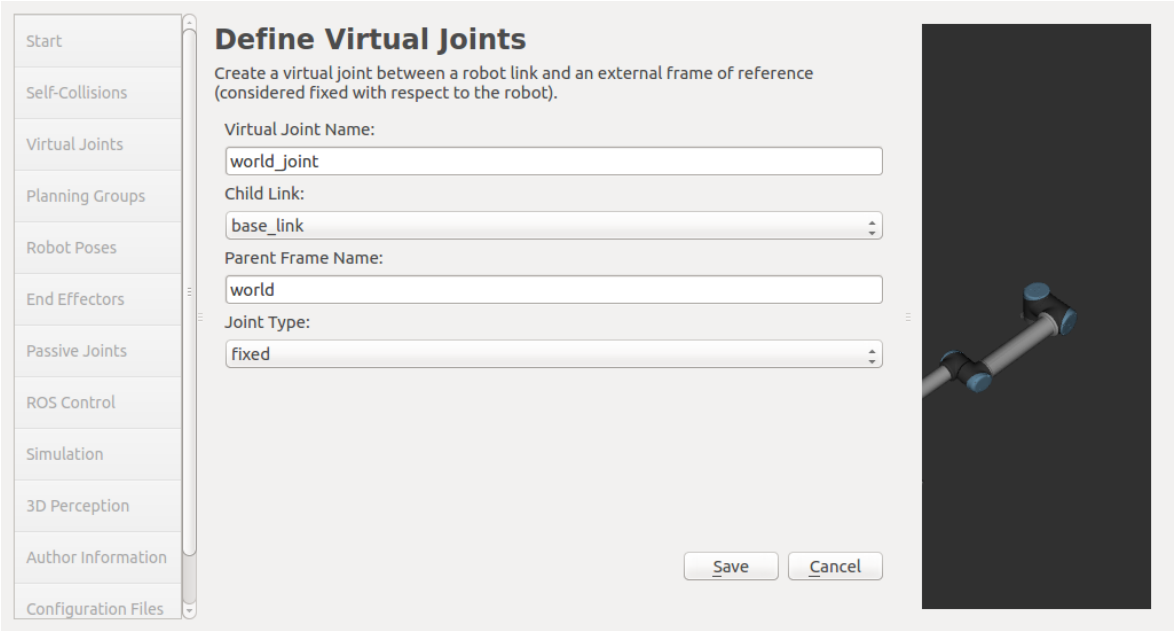


Figura B.4: Definiendo Virtual Joint

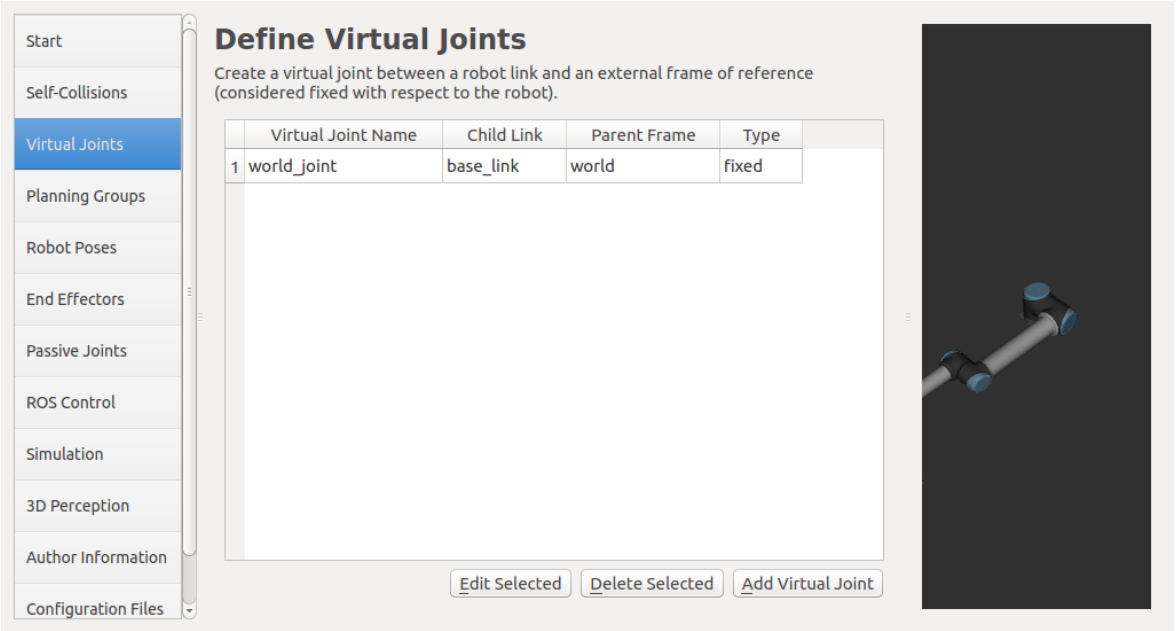


Figura B.5: Definido Virtual Joint

– Una de las pestañas más importantes es definir bien los *Planning groups*, en este caso se tiene dos grupos, el grupo *manipulator* que controlará el brazo del robot y el grupo *gripper* que controlará la pinza:

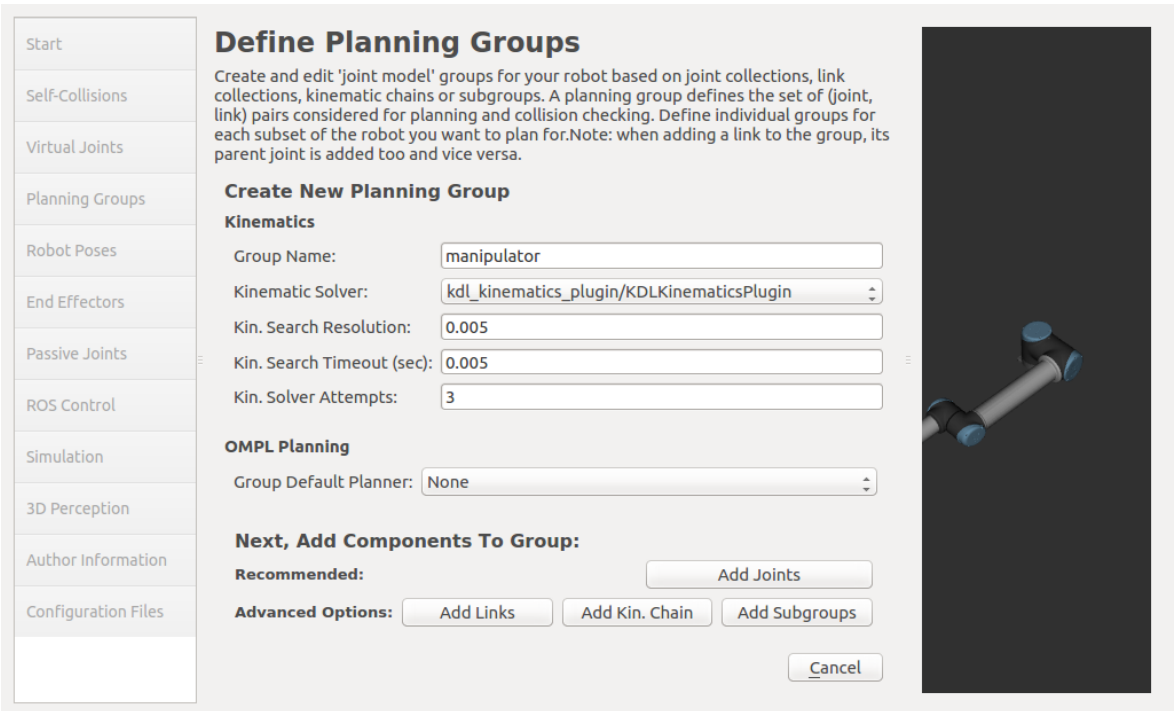


Figura B.6: Manipulator: kdl_kinematics_plugin

– Después hay que darle al botón *Add Kin. Chain*:

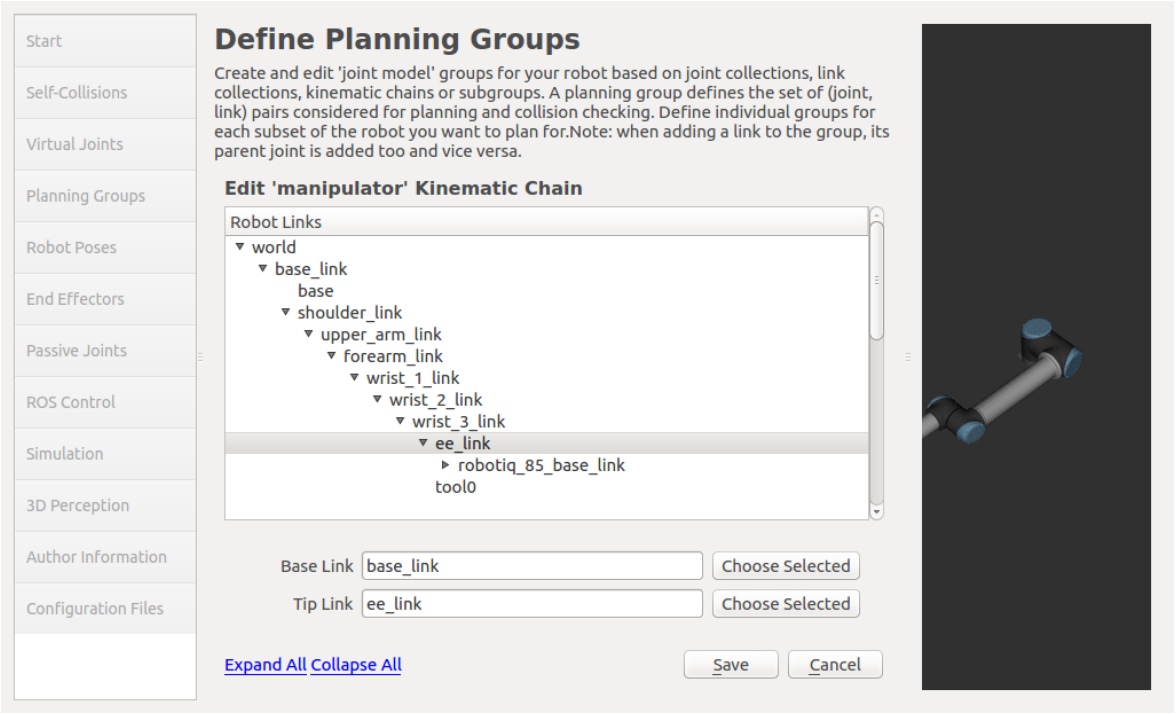


Figura B.7: Manipulator: Kinetic Chain configuración

– Finalmente, se guarda la configuración:

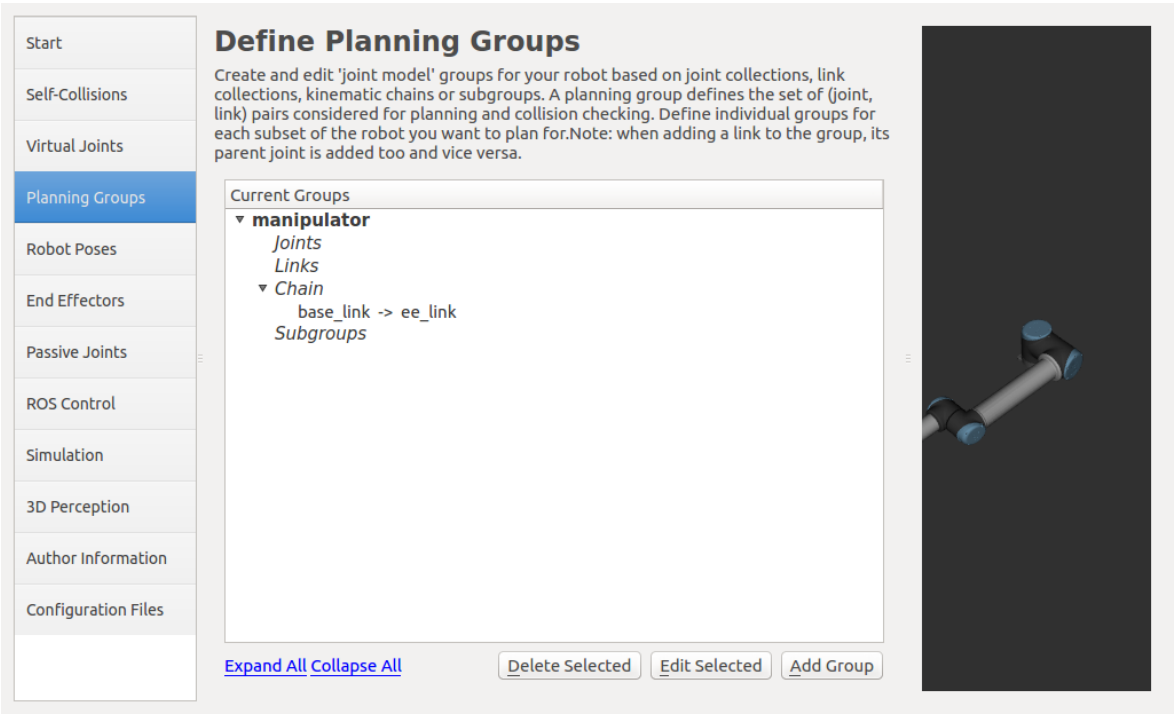


Figura B.8: Grupo manipulator configurado

– Ahora hay que hacerlo para el grupo *Gripper* que controlará la pinza, se pulsa el botón *Add Group*, se rellena con el nombre del grupo y se pone *kdl* como *Kinematic Solver*:

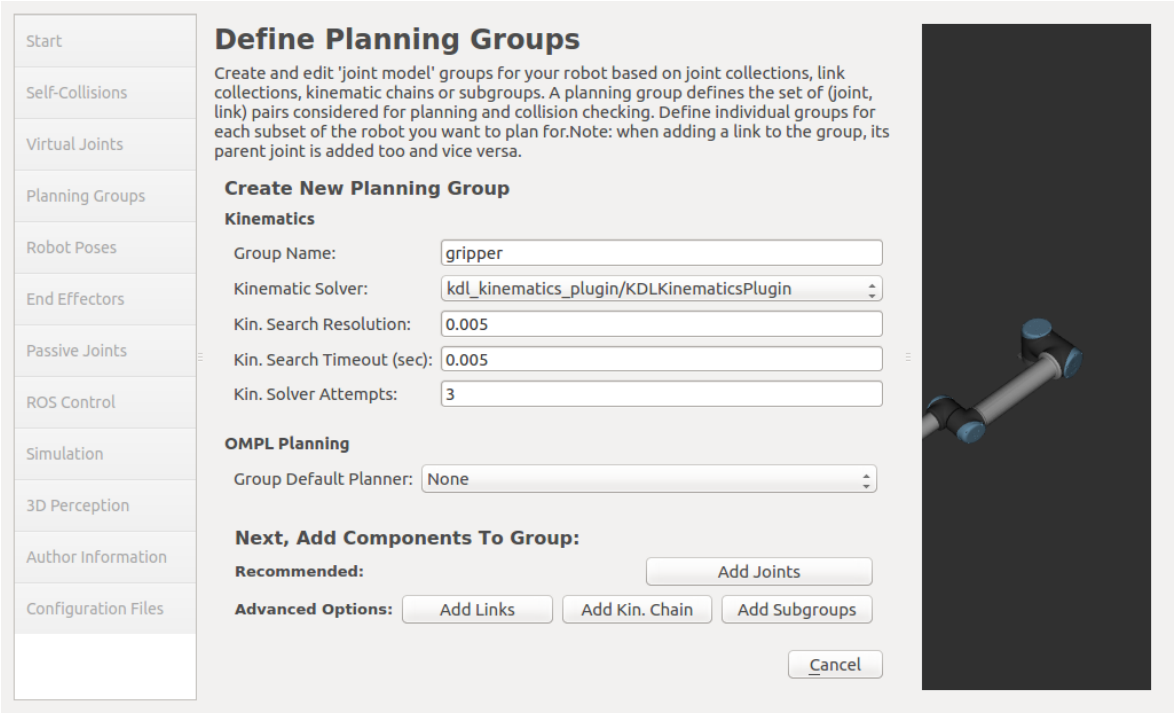


Figura B.9: Gripper: kdl_kinematics_plugin

– Tras darle al botón *Add Joints*, hay que buscar por el joint `robotiq_85_left_knucle_joint` y añadirlo a con la flecha > y se guarda la configuración:

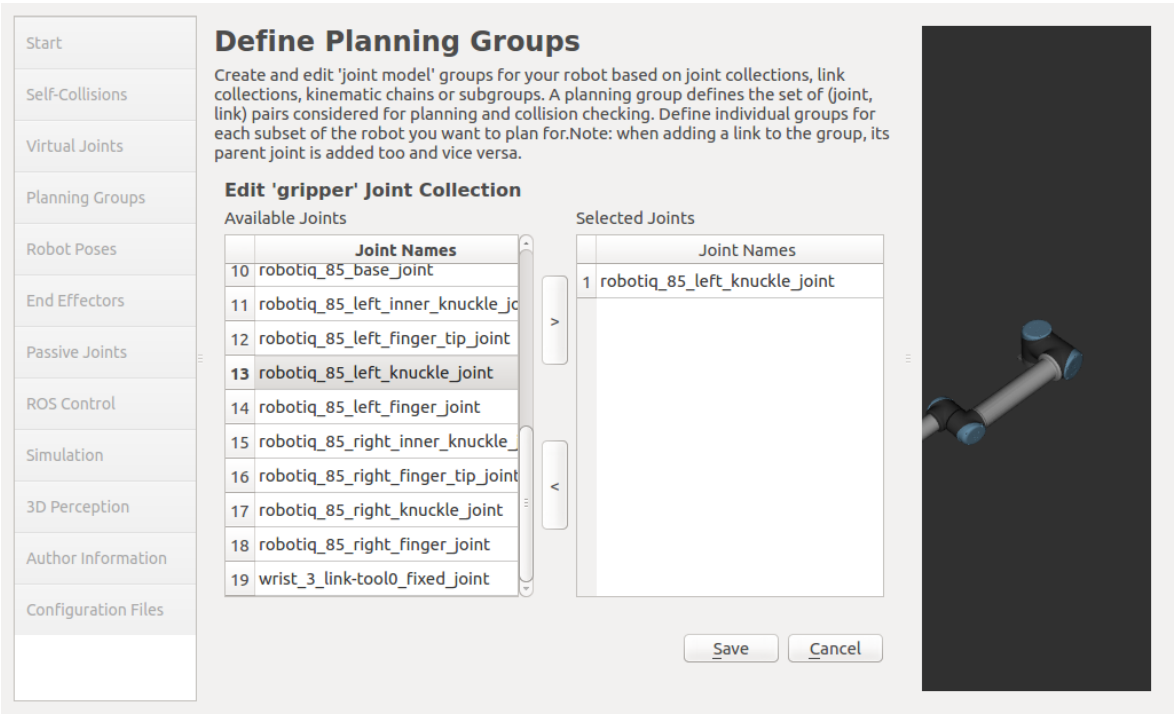


Figura B.10: Gripper: Configuración del Joint del cual los demás imitan

– Tras guardar, el resultado en la pestaña de *Planning Group* debería ser la

siguiente:

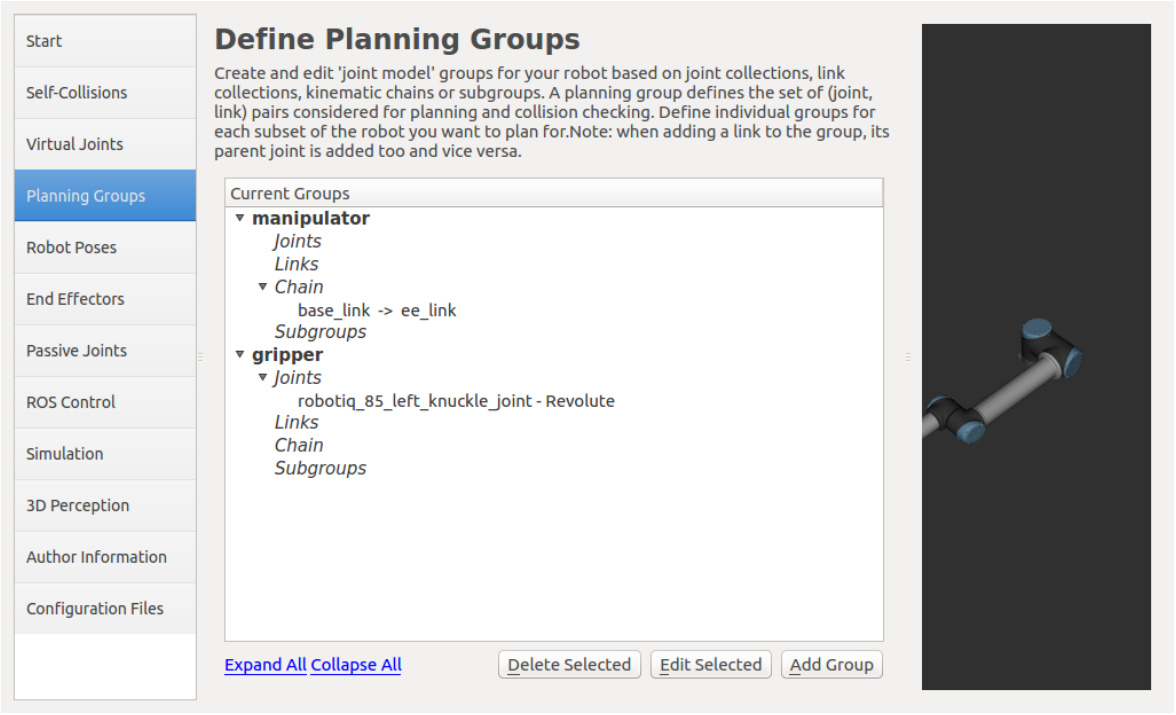


Figura B.11: Configuración final del Planning Group

– En la pestaña *Robot Poses* es donde se configura poses fijas de antemano en el robot, se va a configurar la pose *home* el cual reflejará la posición inicial del cobot:

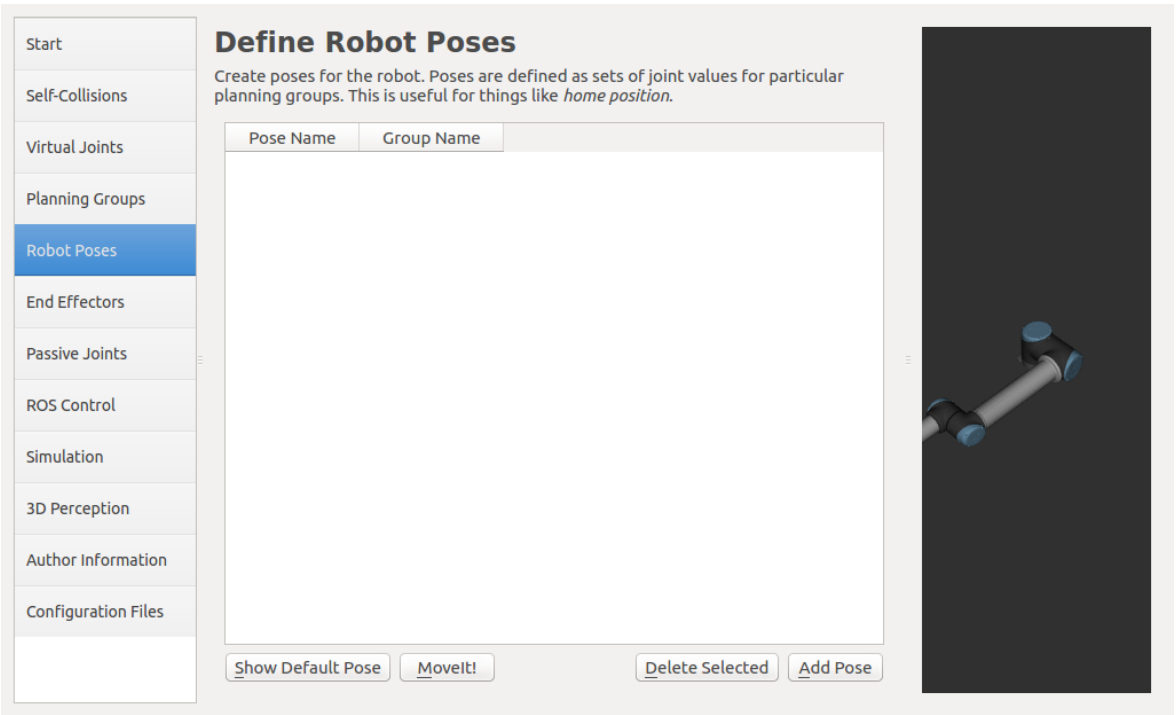


Figura B.12: Configurando pose *home* 1/3

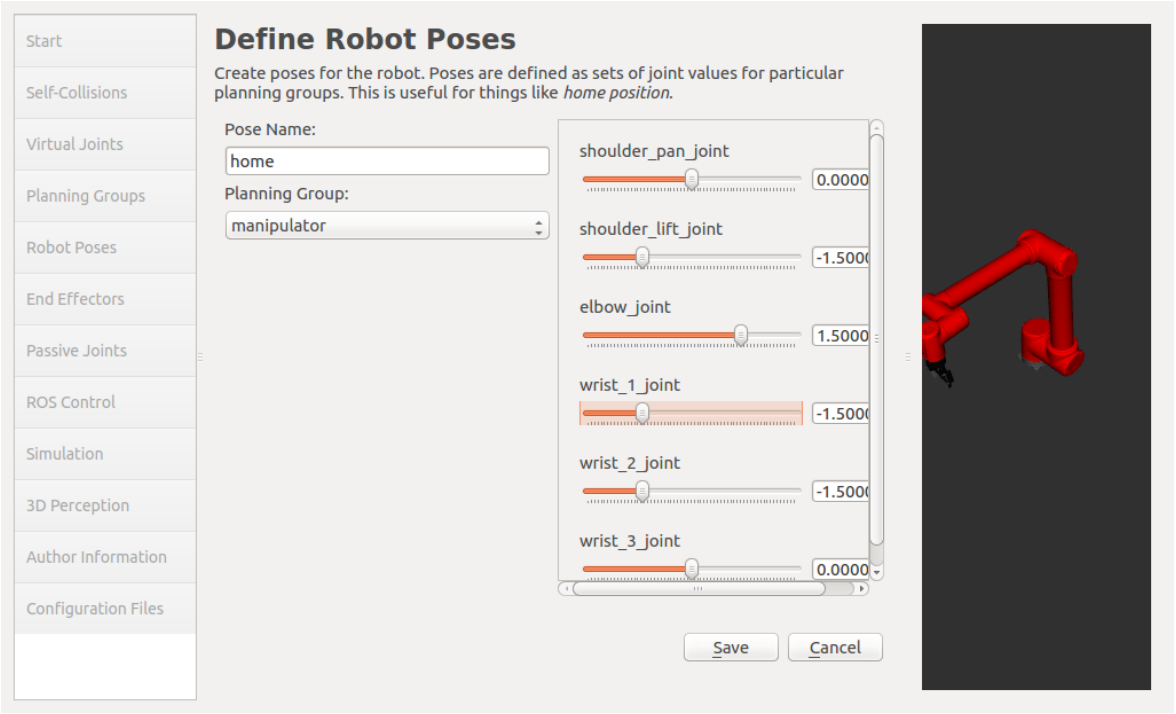


Figura B.13: Configurando pose *home* 2/3

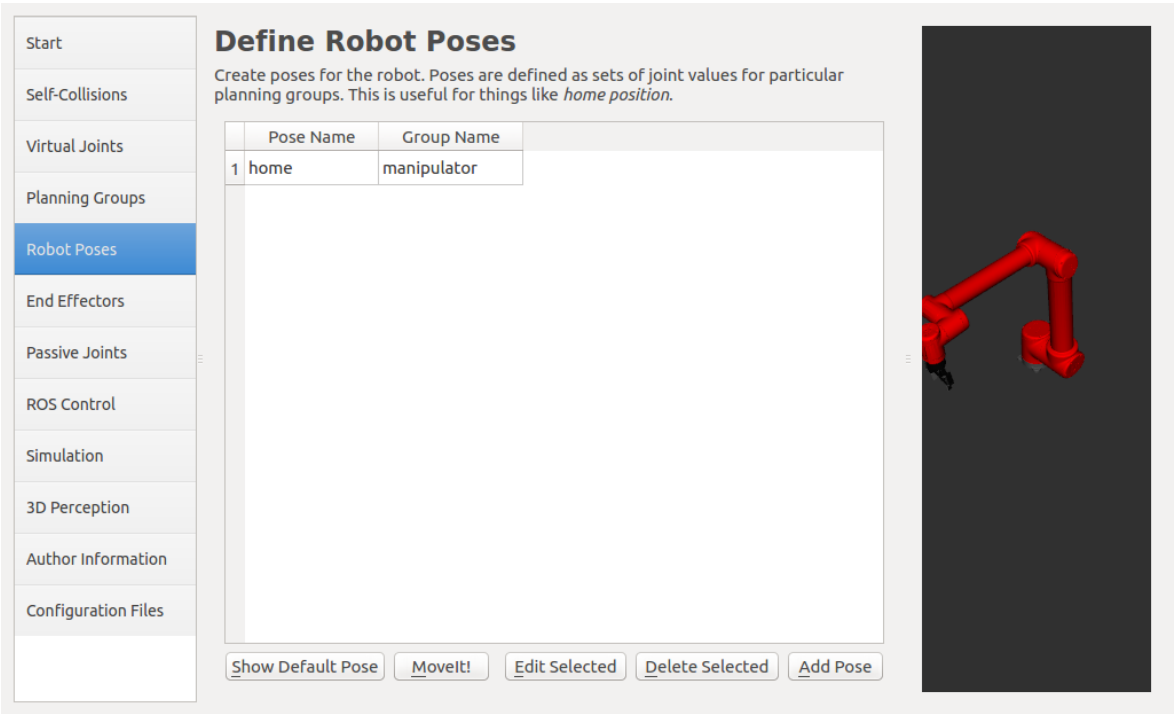


Figura B.14: Configurando pose *home* 3/3

– En la pestaña *Robot Poses* se va a configurar la pose *gripper_open*:

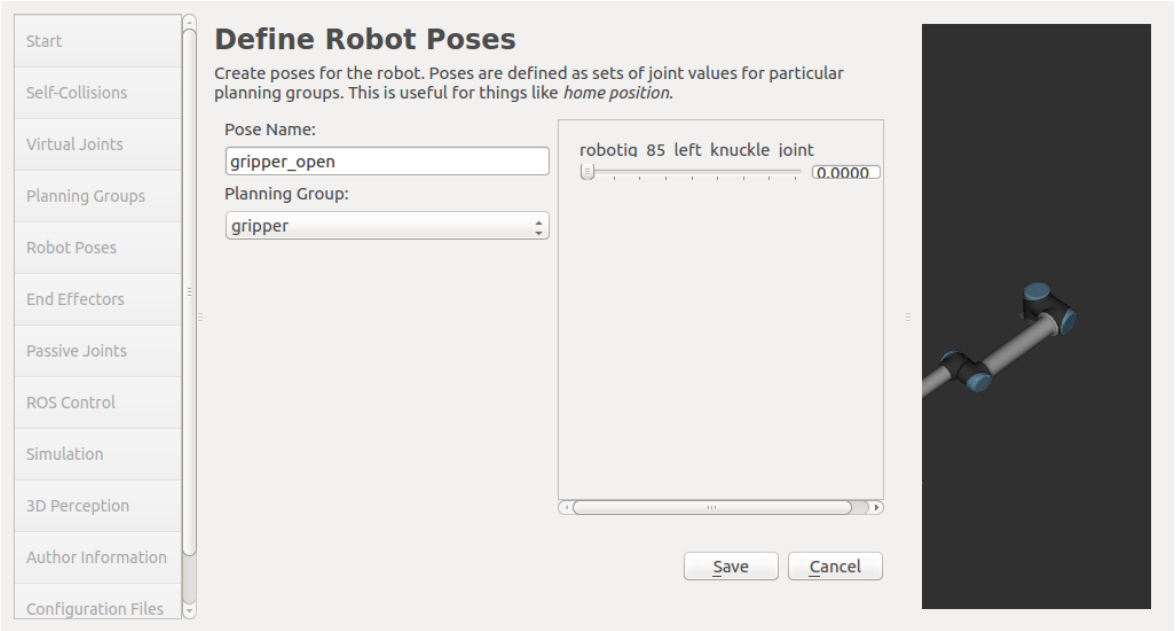


Figura B.15: Configurando pose *gripper_open*

– En la pestaña *Robot Poses* se va a configurar la pose *gripper_close*:

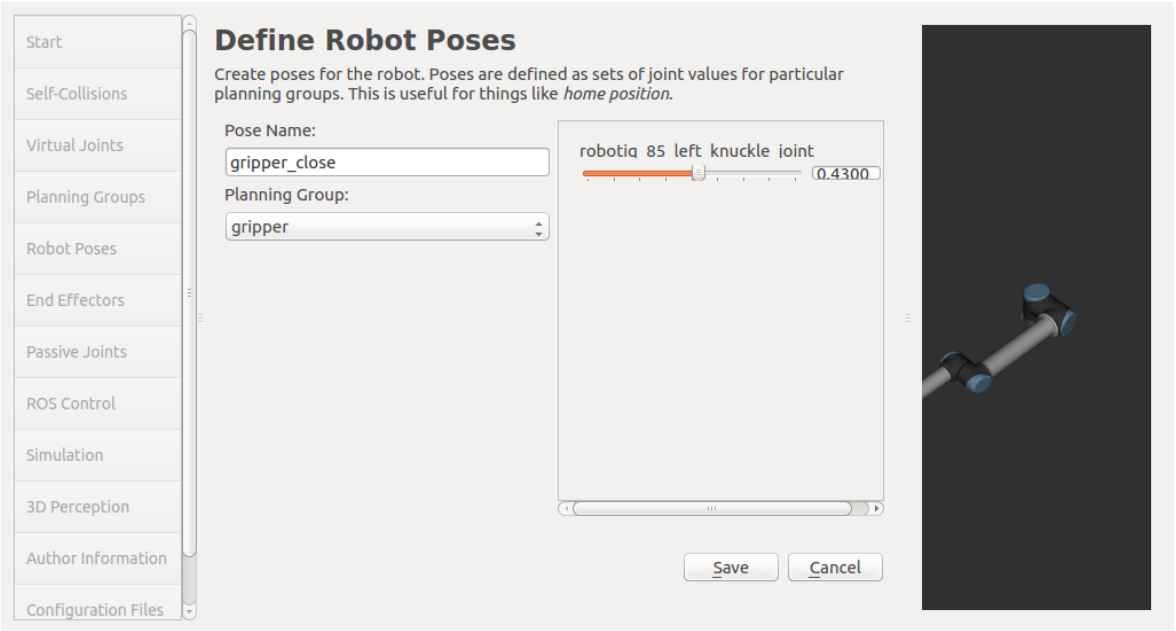


Figura B.16: Configurando pose *gripper_close*

– En la pestaña *End-Effectors* se va a añadir el gripper:

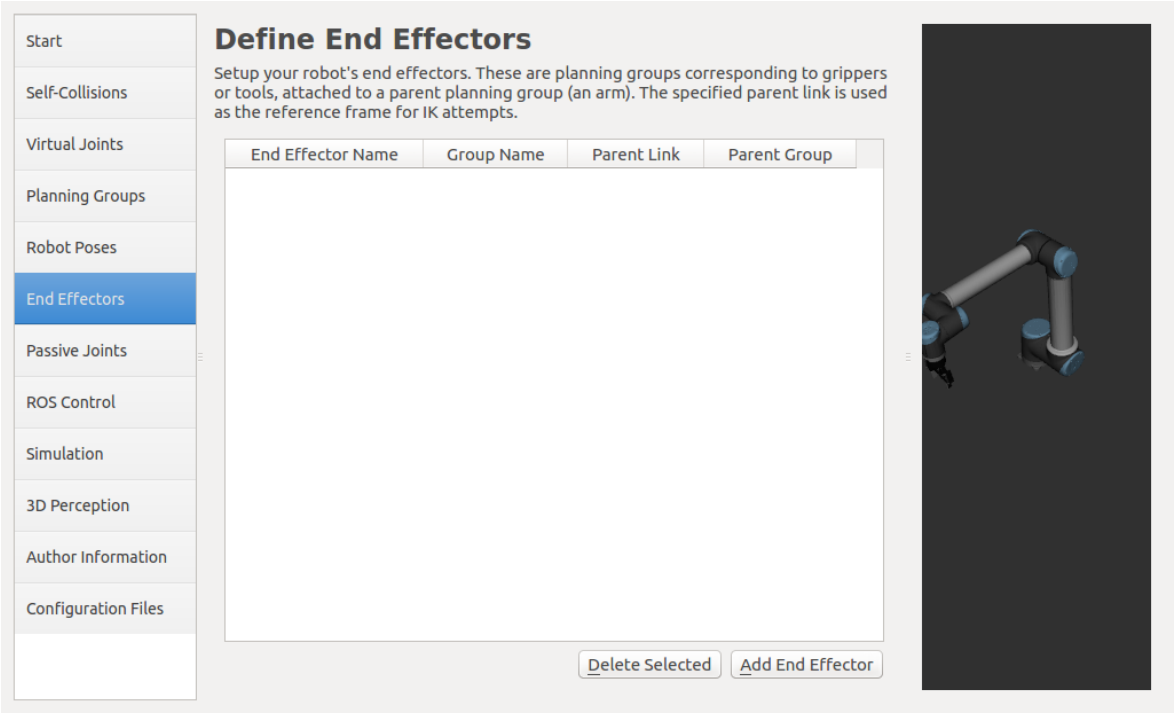


Figura B.17: Configurando end-effector 1/3

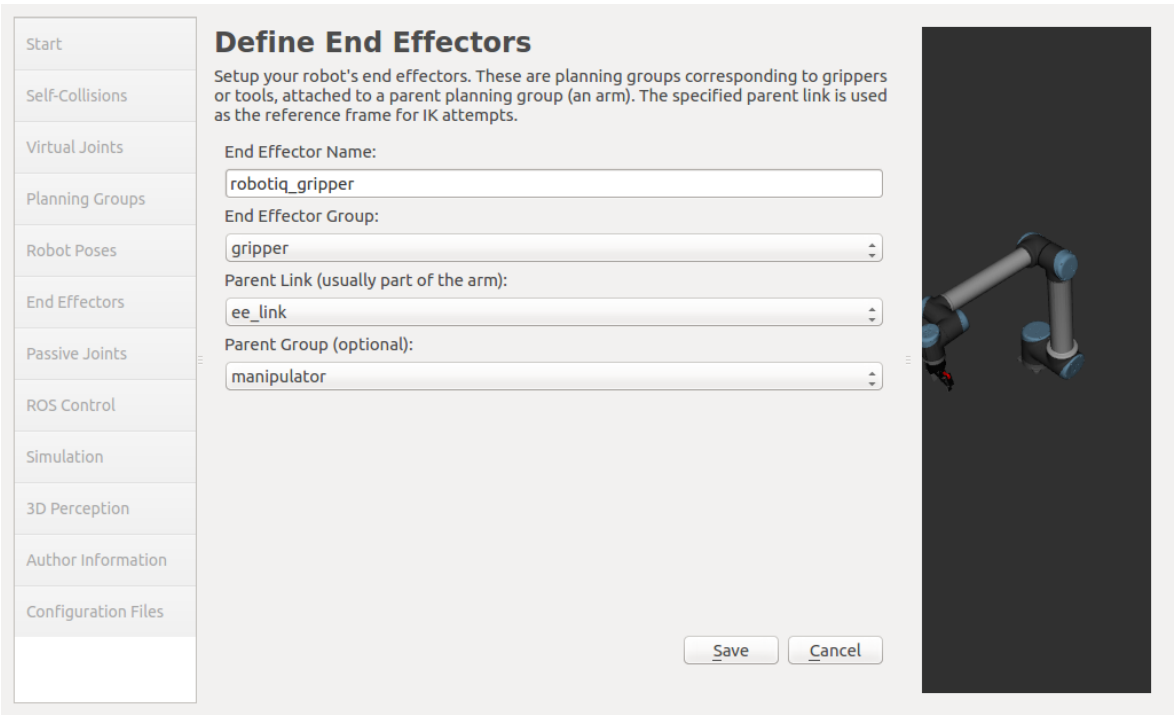


Figura B.18: Configurando end-effector 2/3

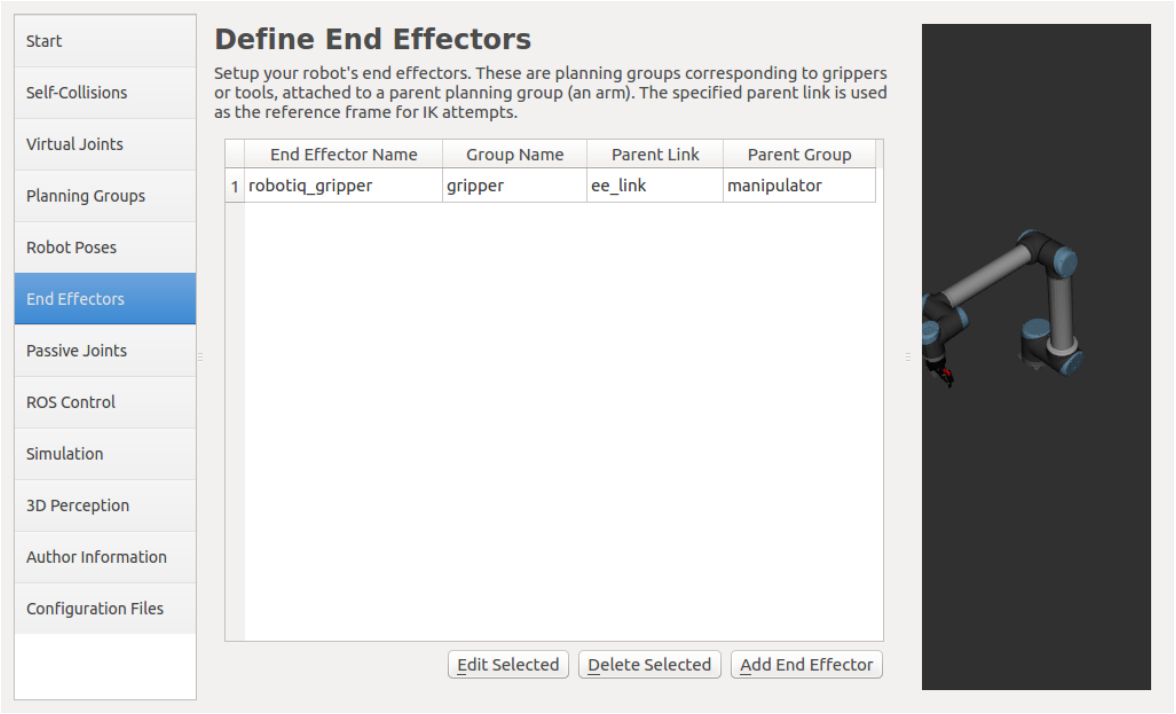


Figura B.19: Configurando end-effector 3/3

– En la pestaña *Passive Joints* representa las articulaciones que no se pueden mover activamente, en el caso del *gripper* definido en el su fichero URDF son los joints que imitan el movimiento de otro joint. Los joints definidos como pasivos no se tendrán en cuenta en la planificación, para este caso la configuración es la siguiente:

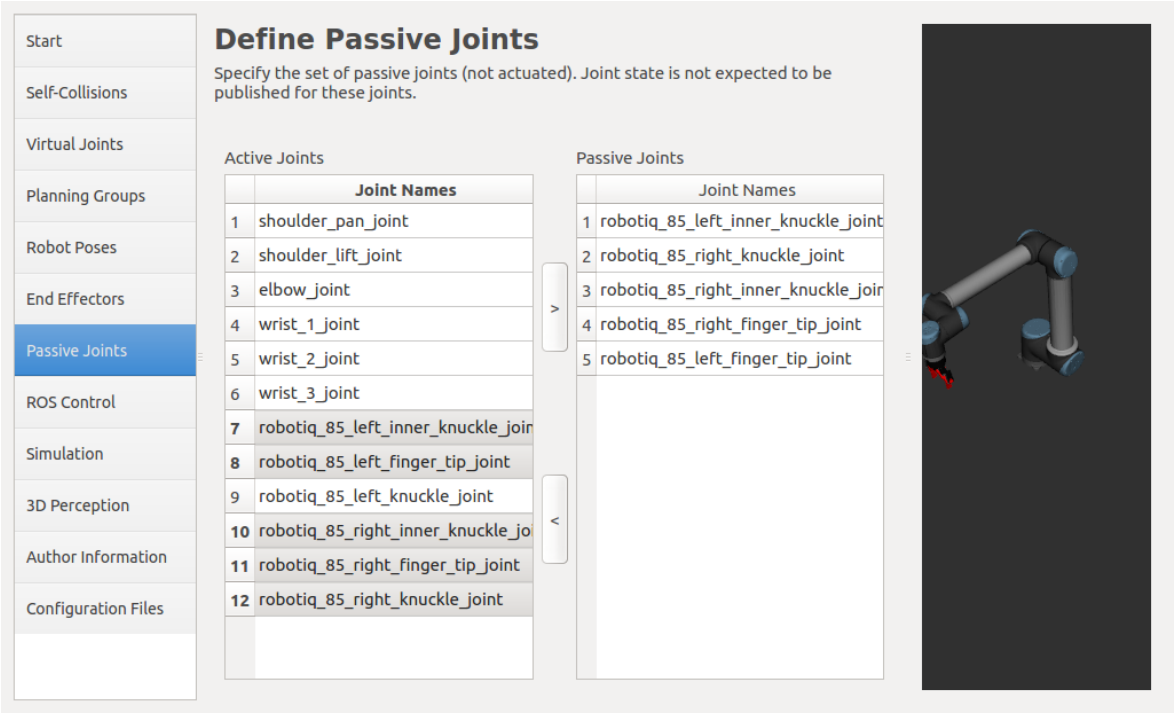


Figura B.20: Configurando Passive Joints

– En la pestaña *ROS Control*, se añadirá de forma automática, los ficheros generados, serán posteriormente modificados manualmente:

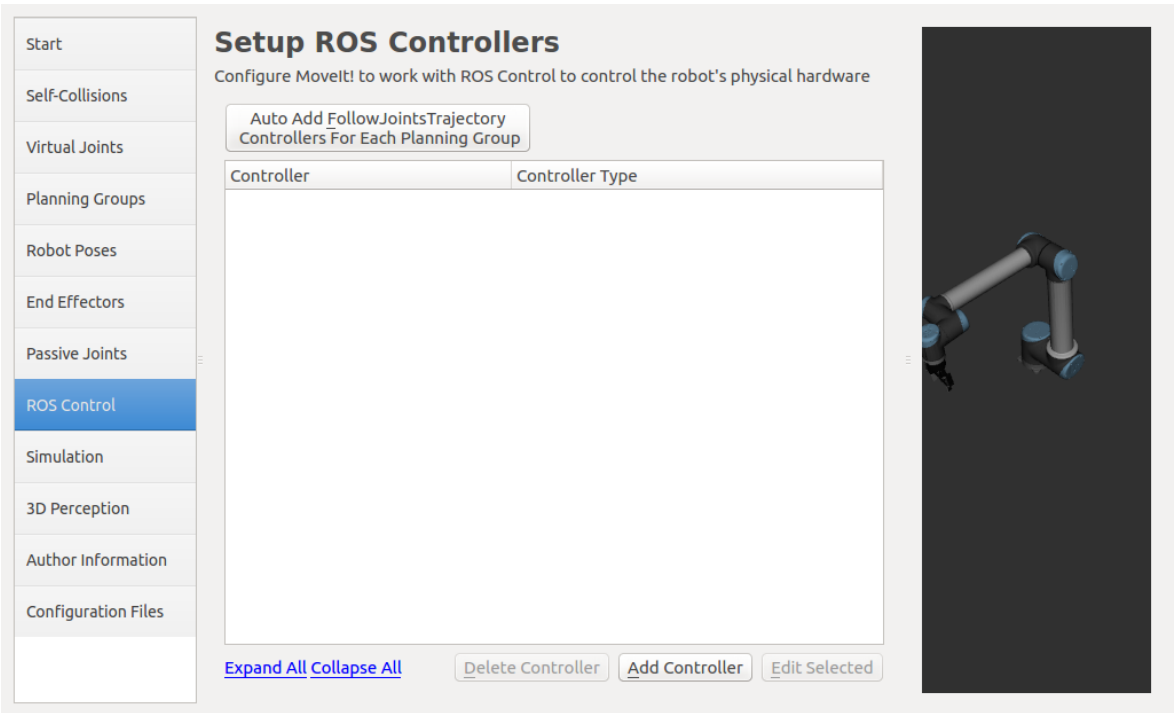


Figura B.21: Configurando ROS Control

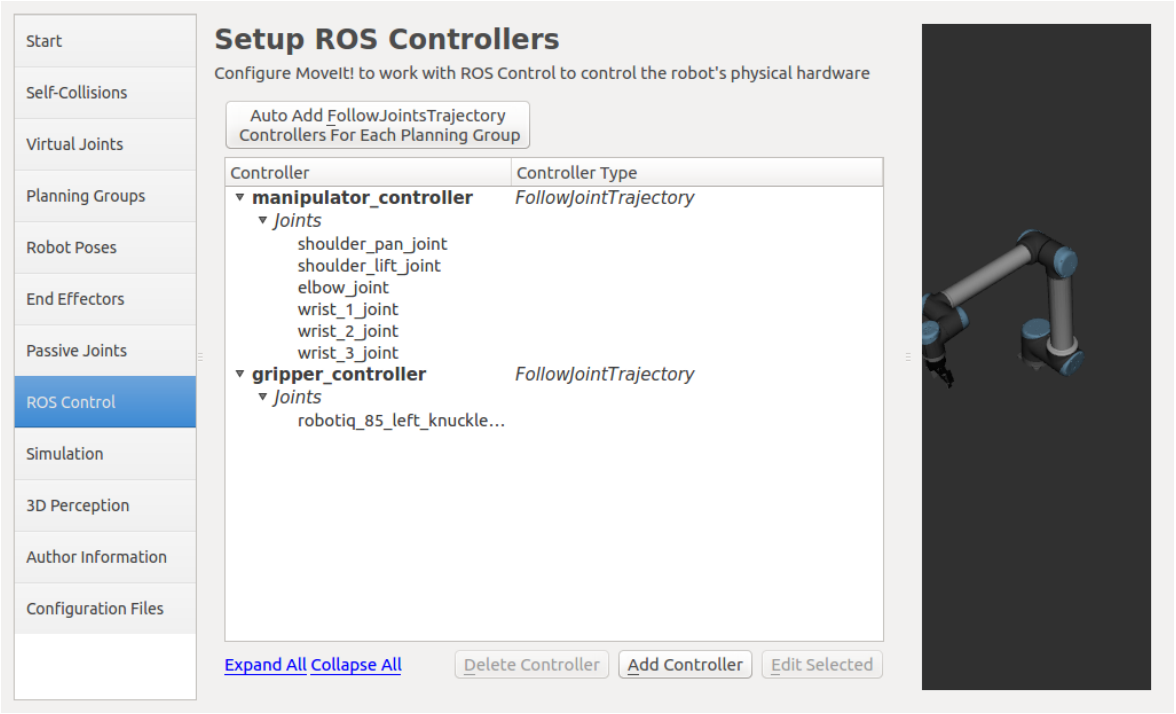


Figura B.22: Configurando ROS Control

– Hay que rellenar la pestaña *Author Information* para que la configuración pueda terminar:

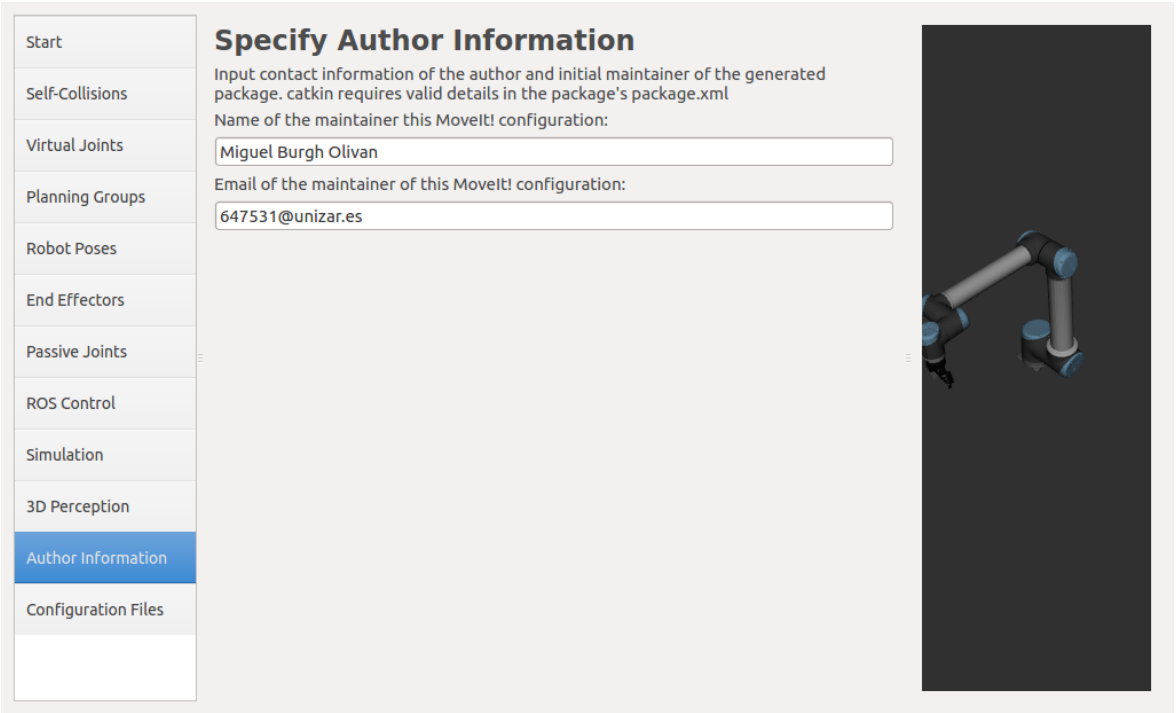


Figura B.23: Configurando Author Information

– La última pestaña *Configuration Files*, permite decidir dónde se guardará la configuración de MoveIt!, en este caso en `one_arm_moveit_config` creado

previamente, se genera la configuración mediante el botón *Generate Package* y finalmente *Exit Setup Assistant* para terminar:

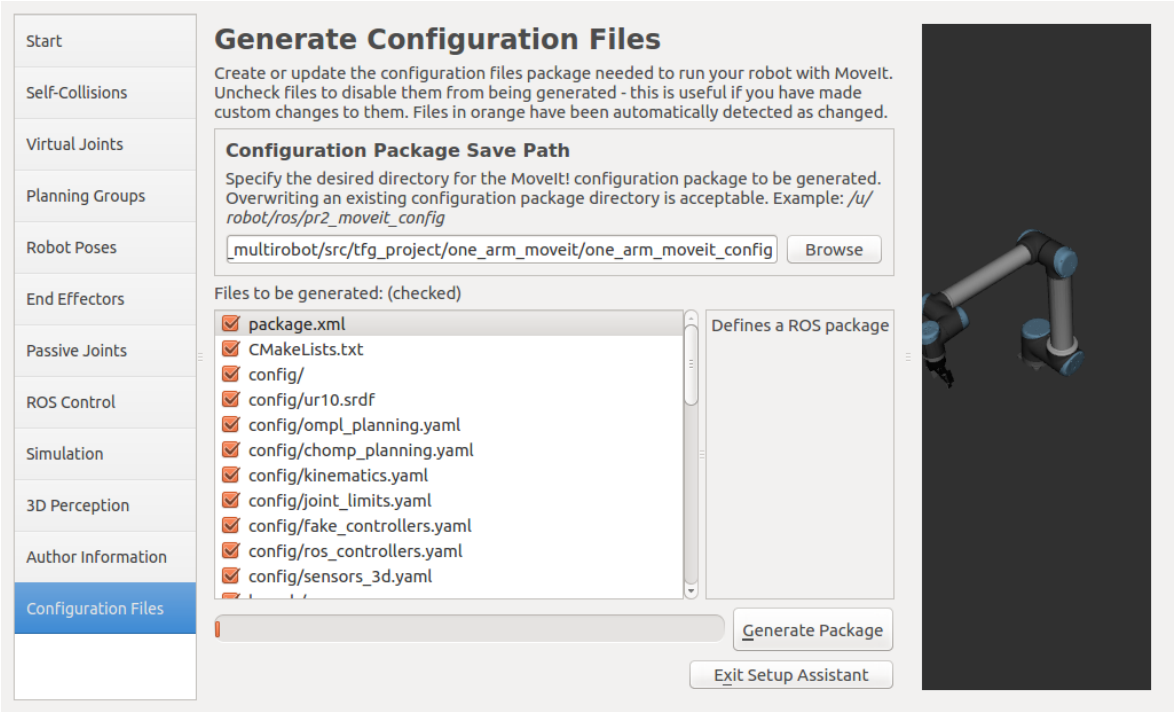


Figura B.24: Generando los ficheros de configuración

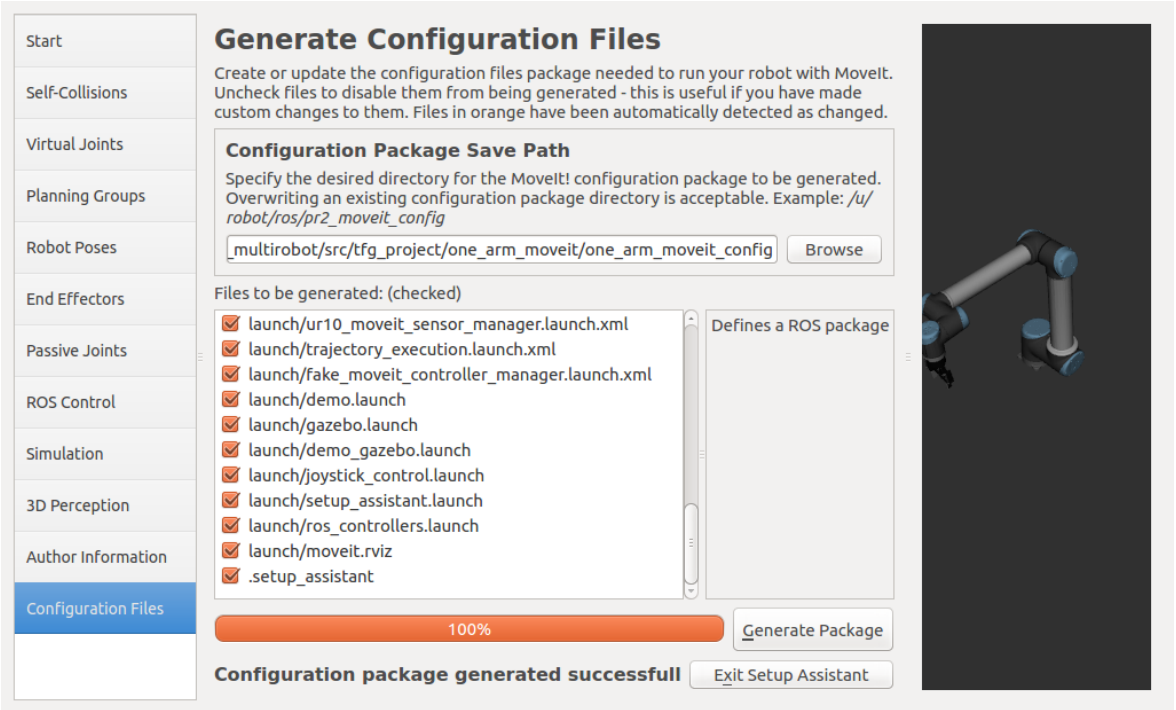


Figura B.25: Generado los ficheros de configuración

Anexos C

Gráficas de las soluciones propuestas

C.1. Solución desarrollada con el paquete de MoveIt! para un único cobot

Gráficos en detalle de la solución desarrollada para un único cobot en la Subsección 6.1.1 de la Sección 6.1 del Capítulo 6, de la solución desarrollada con el paquete MoveIt!.

C.1.1. Fase 2: Visualización en Rviz de la solución con Moveit!

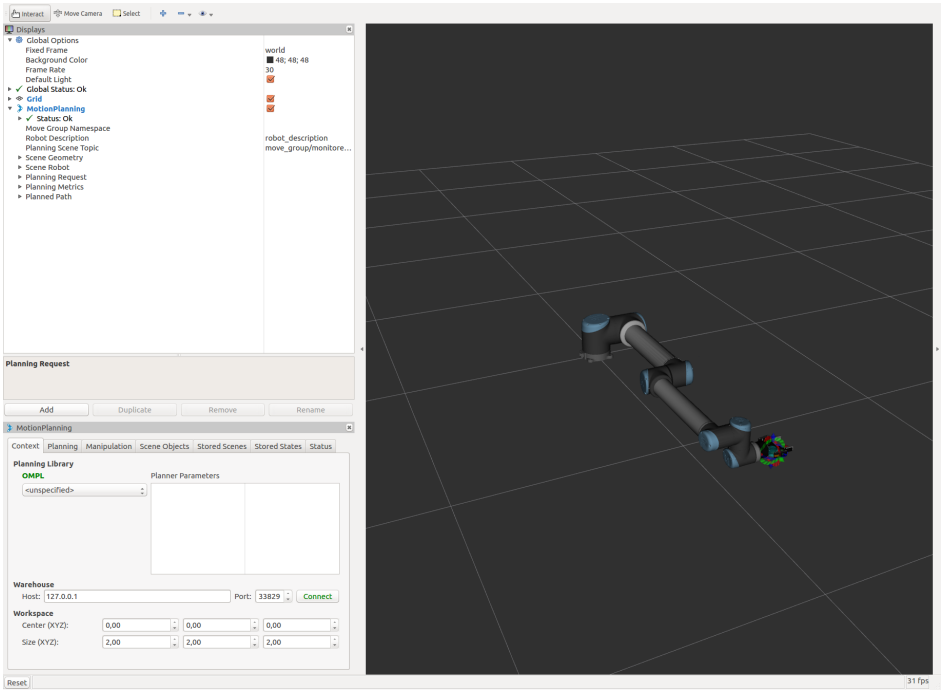


Figura C.1: Fase 2: Visualización en Rviz al lanzar MoveIt!

C.1.2. Fase 2: Nodos y topics de la solución con Moveit!

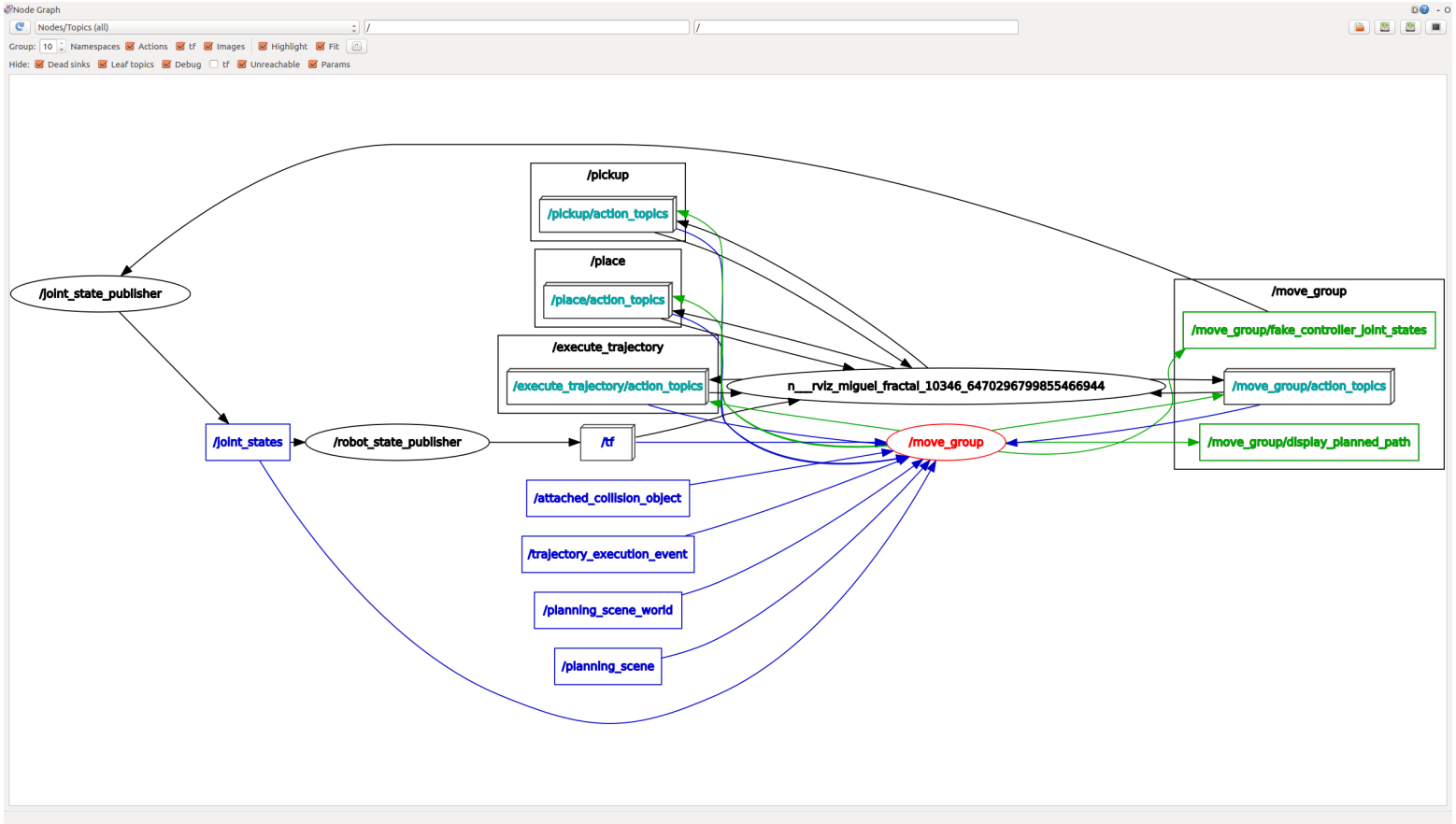


Figura C.2: Fase 2: Gráfico de nodos y topics usando la herramienta `rqt_graph`

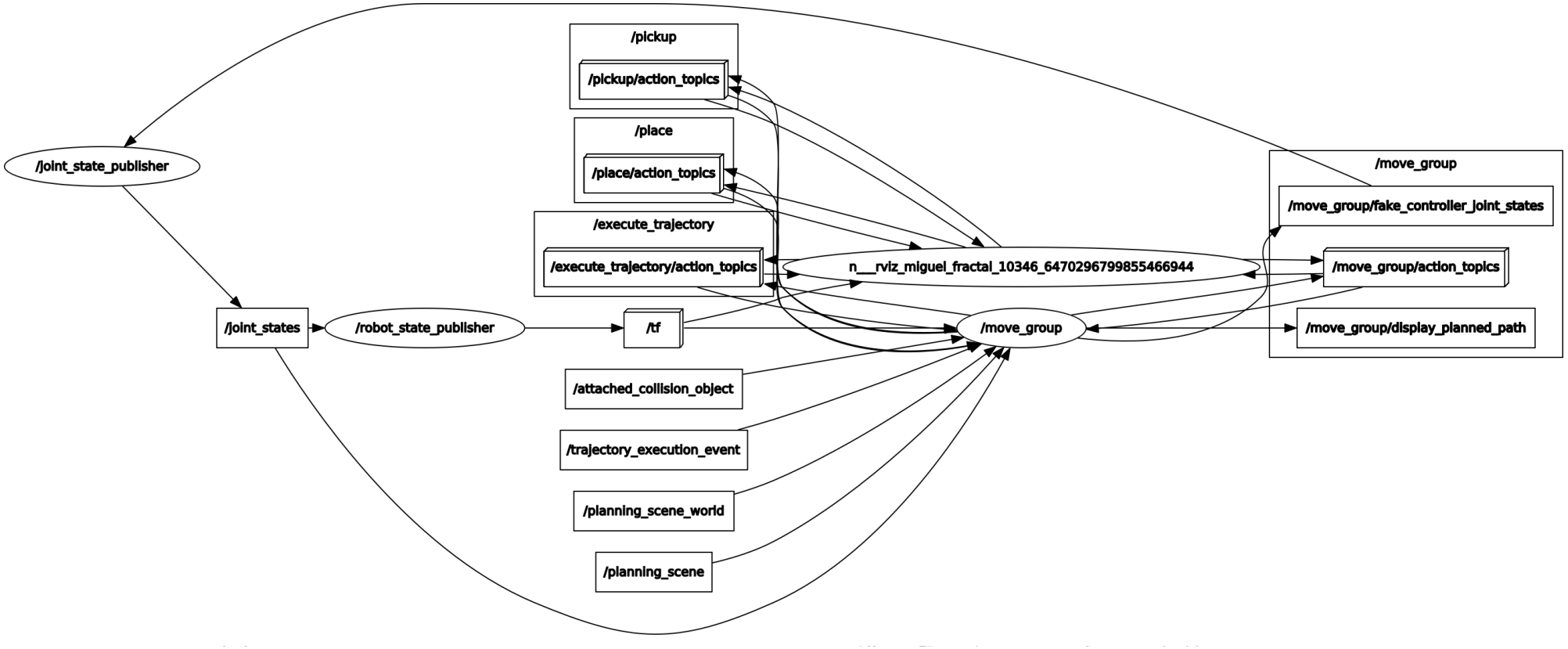


Figura C.3: Fase 2: Gráfico de nodos y topics usando la herramienta `rqt_graph` (mejor definición)

C.1.3. Fase 2: Árbol de las transformadas de la solución con MoveIt!

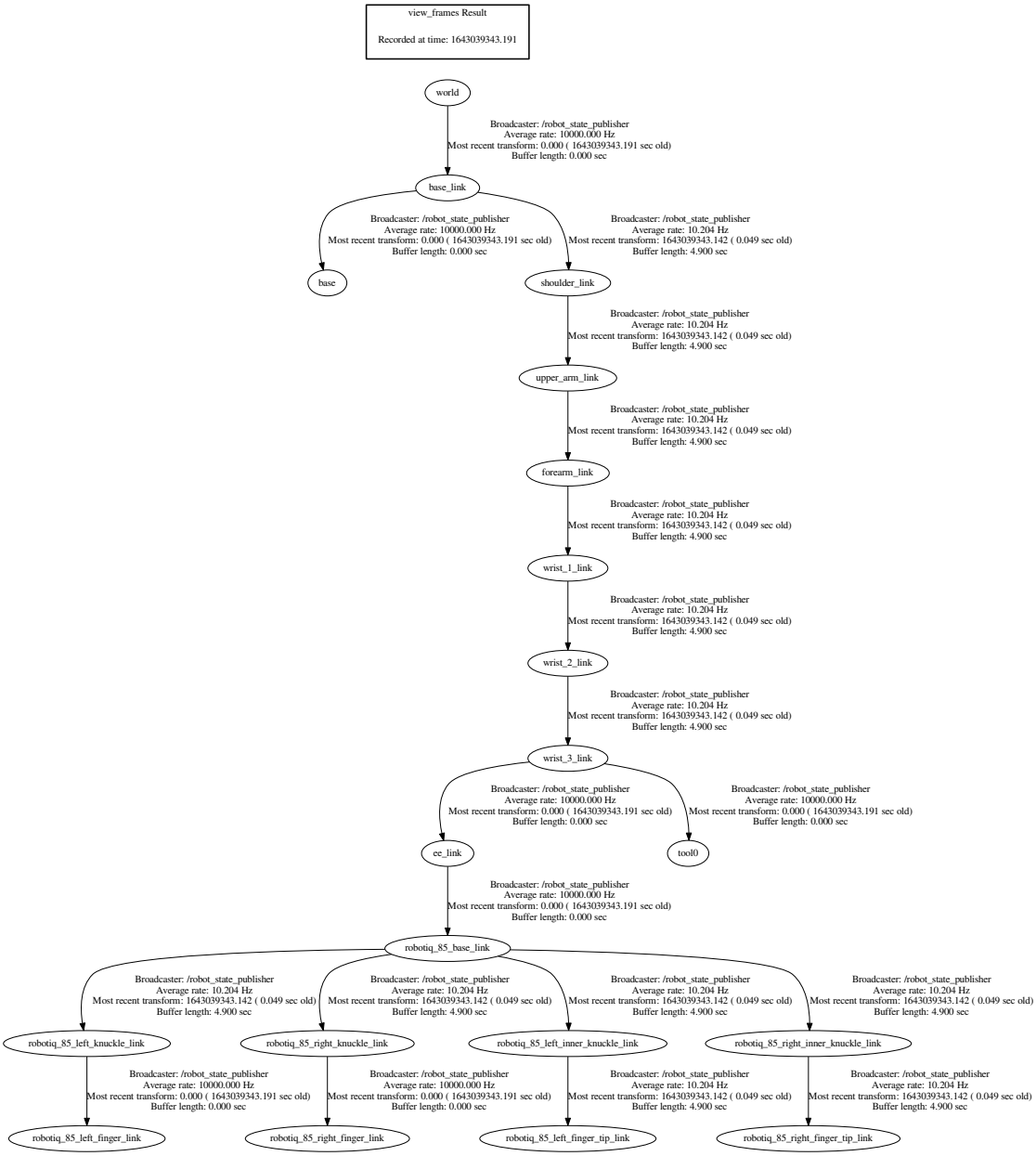


Figura C.4: Herramienta rqt_graph: Gráfica de nodos y topics de la *Fase 2*

C.1.4. Fase 2: Nodos y topics de la solución con Moveit! junto a todos los servicios

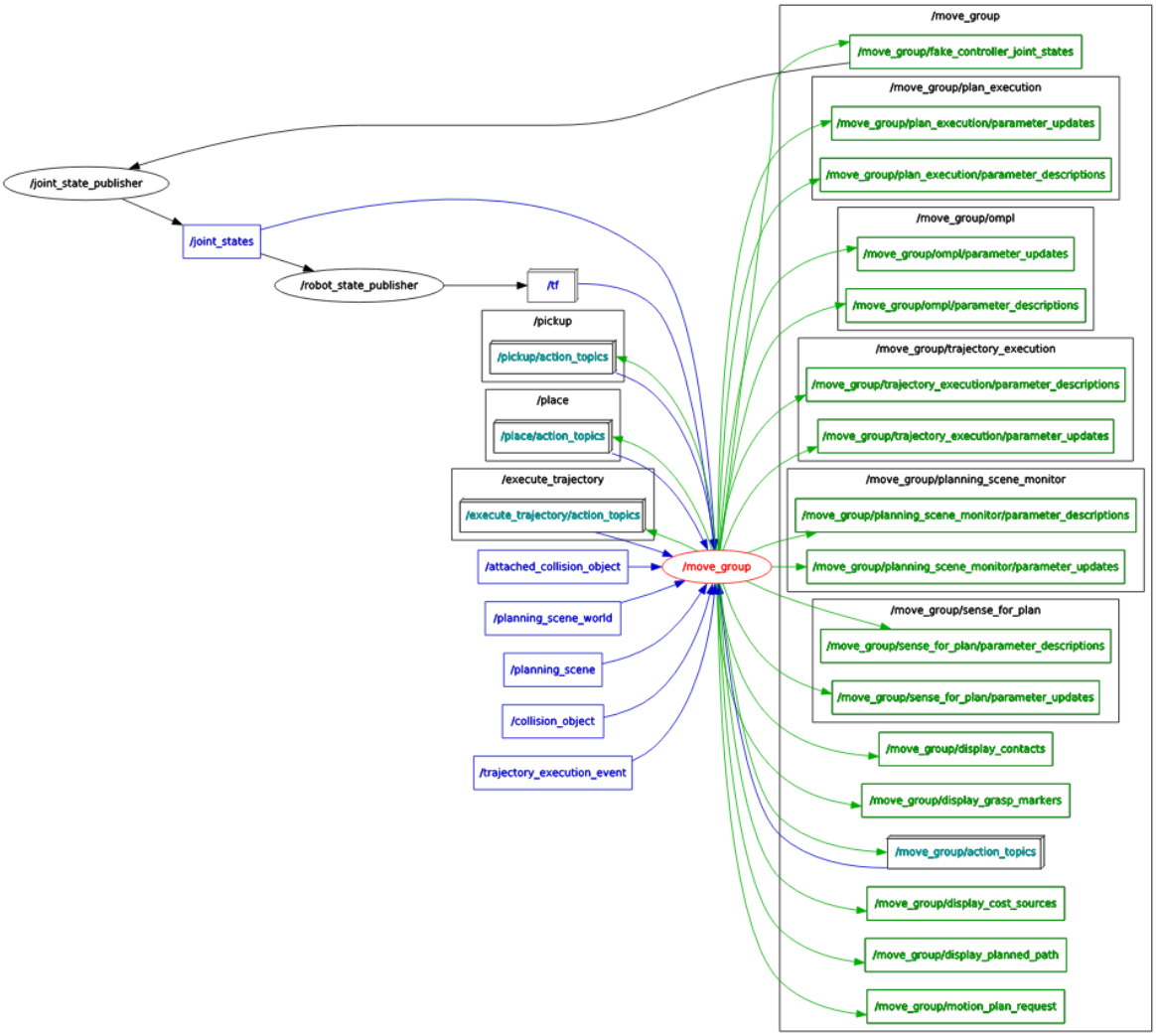


Figura C.5: Fase 2: Gráfico de `move_group` y todos los servicios usando la herramienta `rqt_graph`

C.1.5. Fase 3: Nodos y topics de la solución con Moveit! conexión con Gazebo fallida

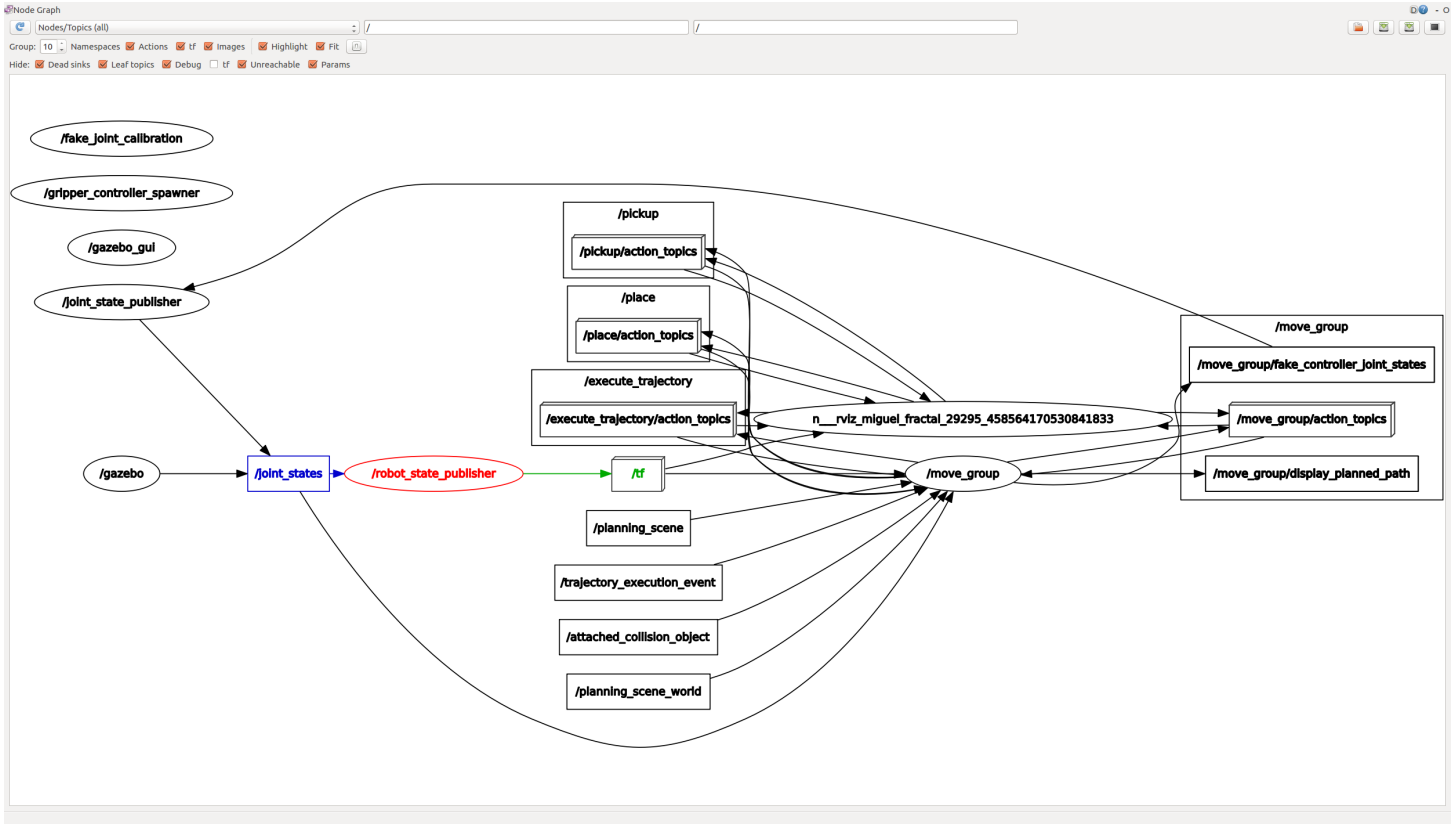


Figura C.6: Fase 3: Nodos y topics de la solución con Moveit! conexión con Gazebo fallida

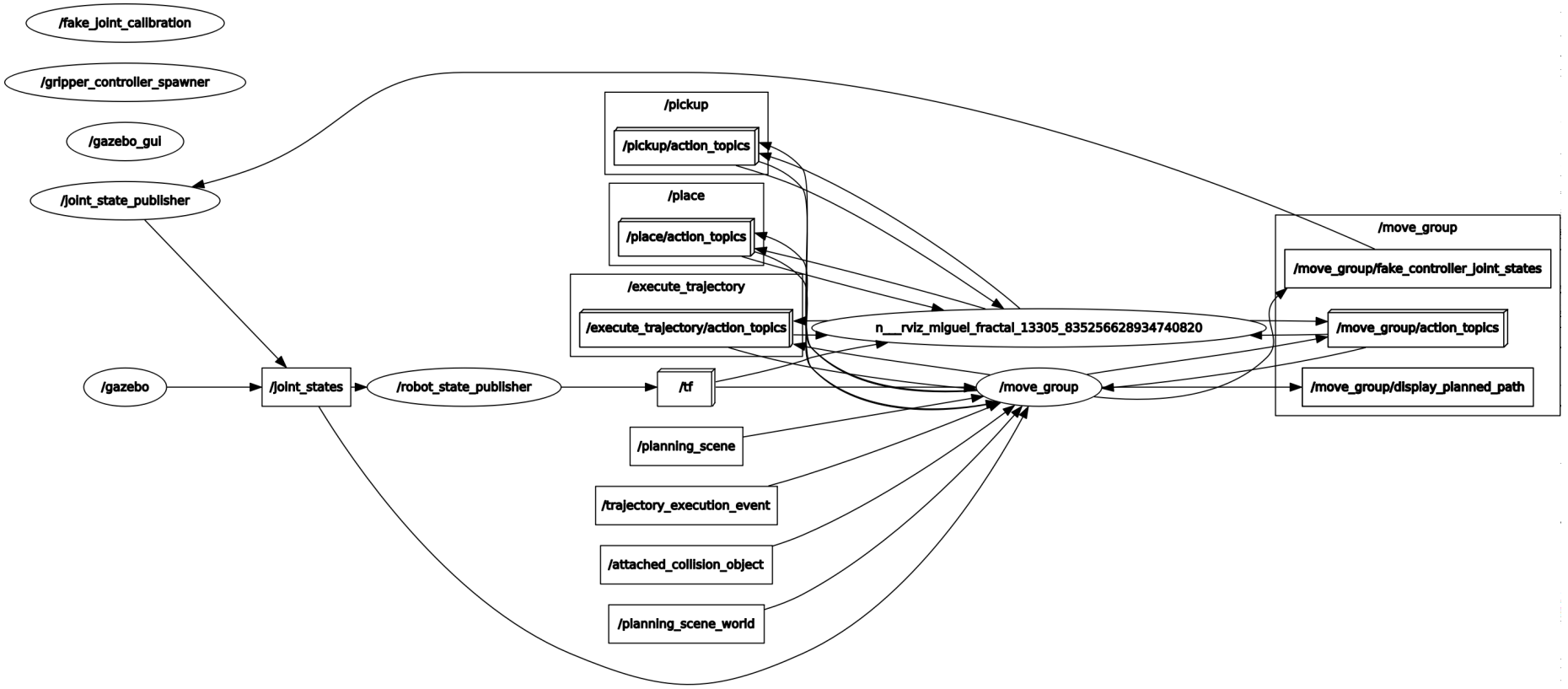


Figura C.7: Fase 3: Nodos y topics de la solución con Moveit! conexión con Gazebo fallida (mejor definición)

C.1.6. Fase 3: Comunicación entre MoveIt! y los controladores de Gazebo

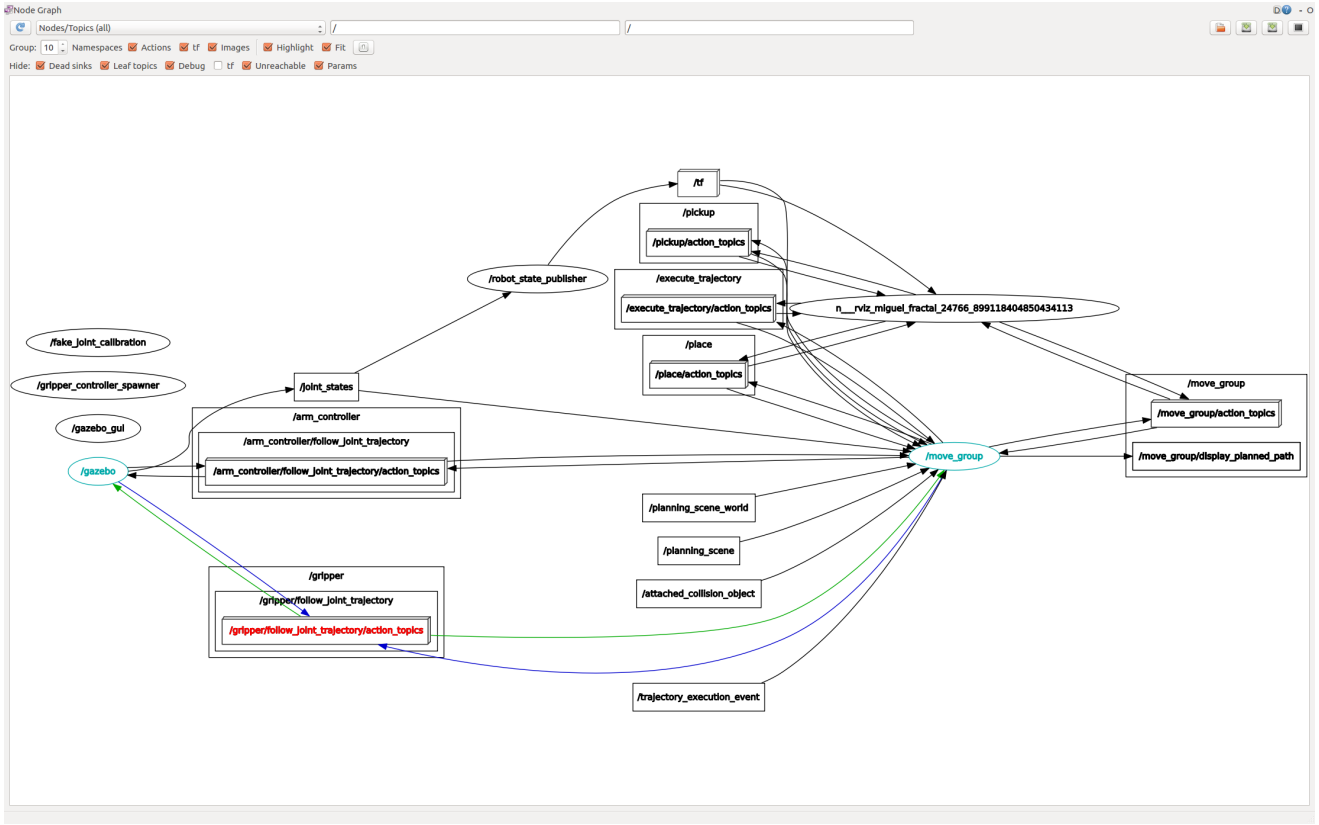


Figura C.8: Fase 3: Comunicación entre MoveIt! y los controladores de Gazebo

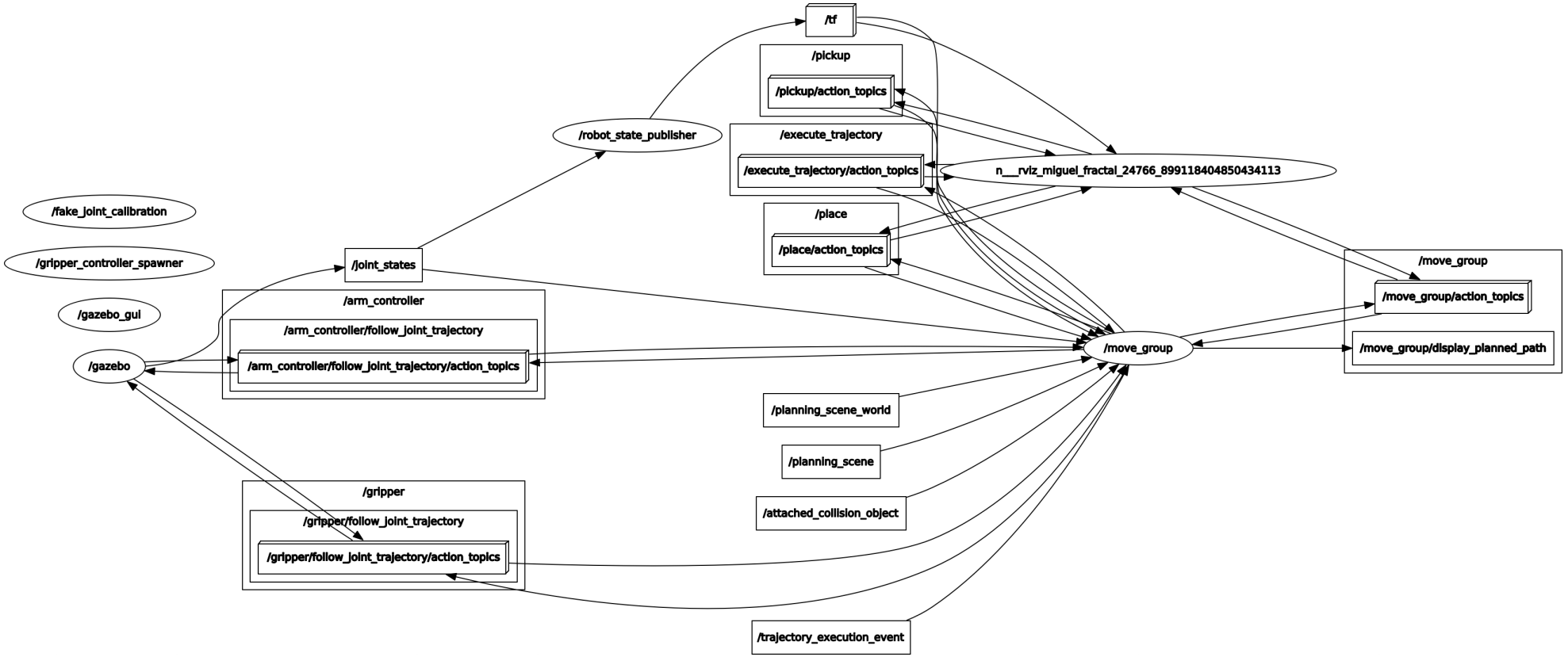


Figura C.9: Fase 3: Comunicación entre MoveIt! y los controladores de Gazebo (mejor definición)

C.1.7. Fase 3: Comunicación entre MoveIt! y Gazebo, realizando un simple *pick & place*

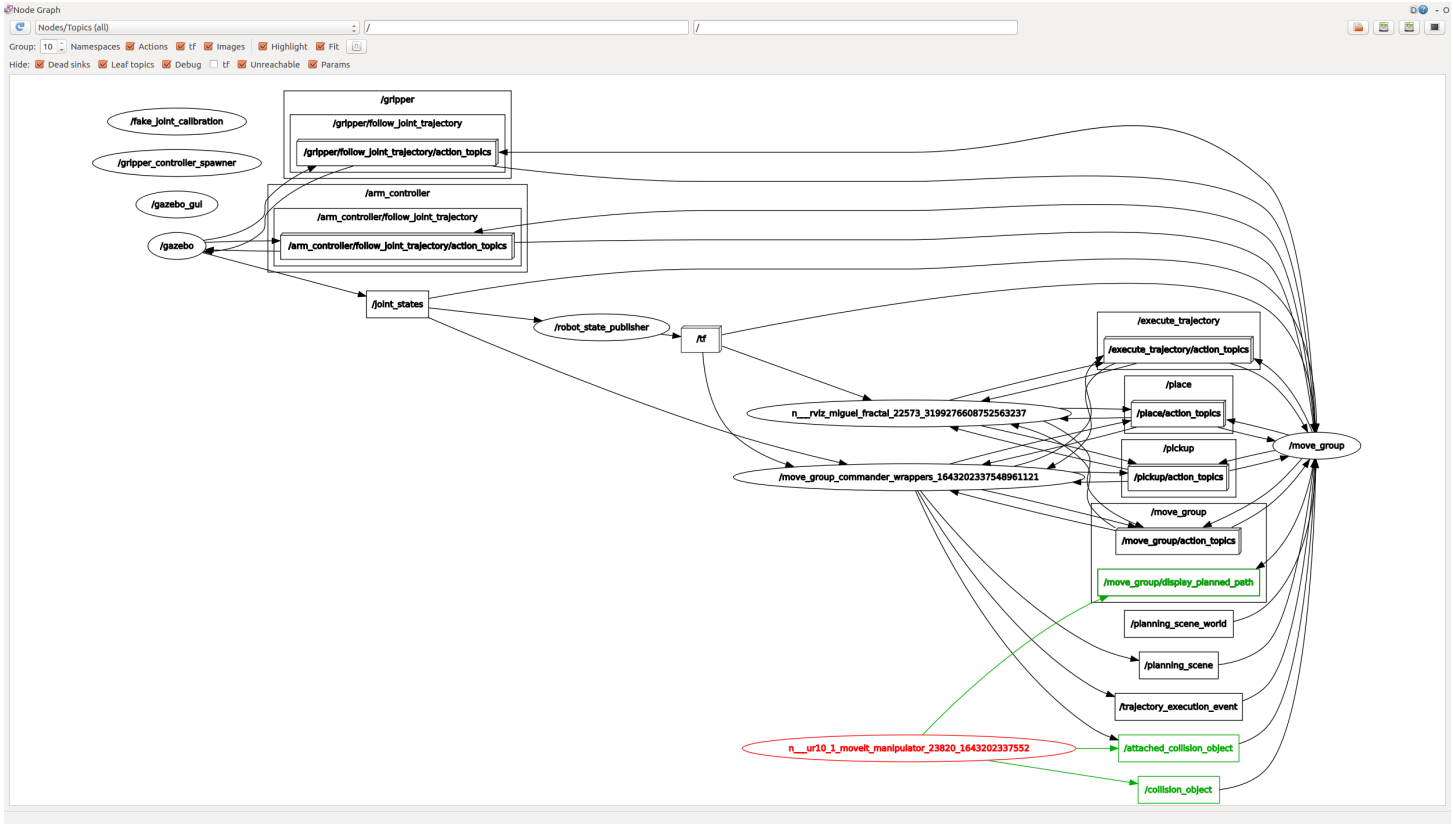


Figura C.10: Fase 3: Comunicación entre MoveIt! y Gazebo, realizando un simple *pick & place*

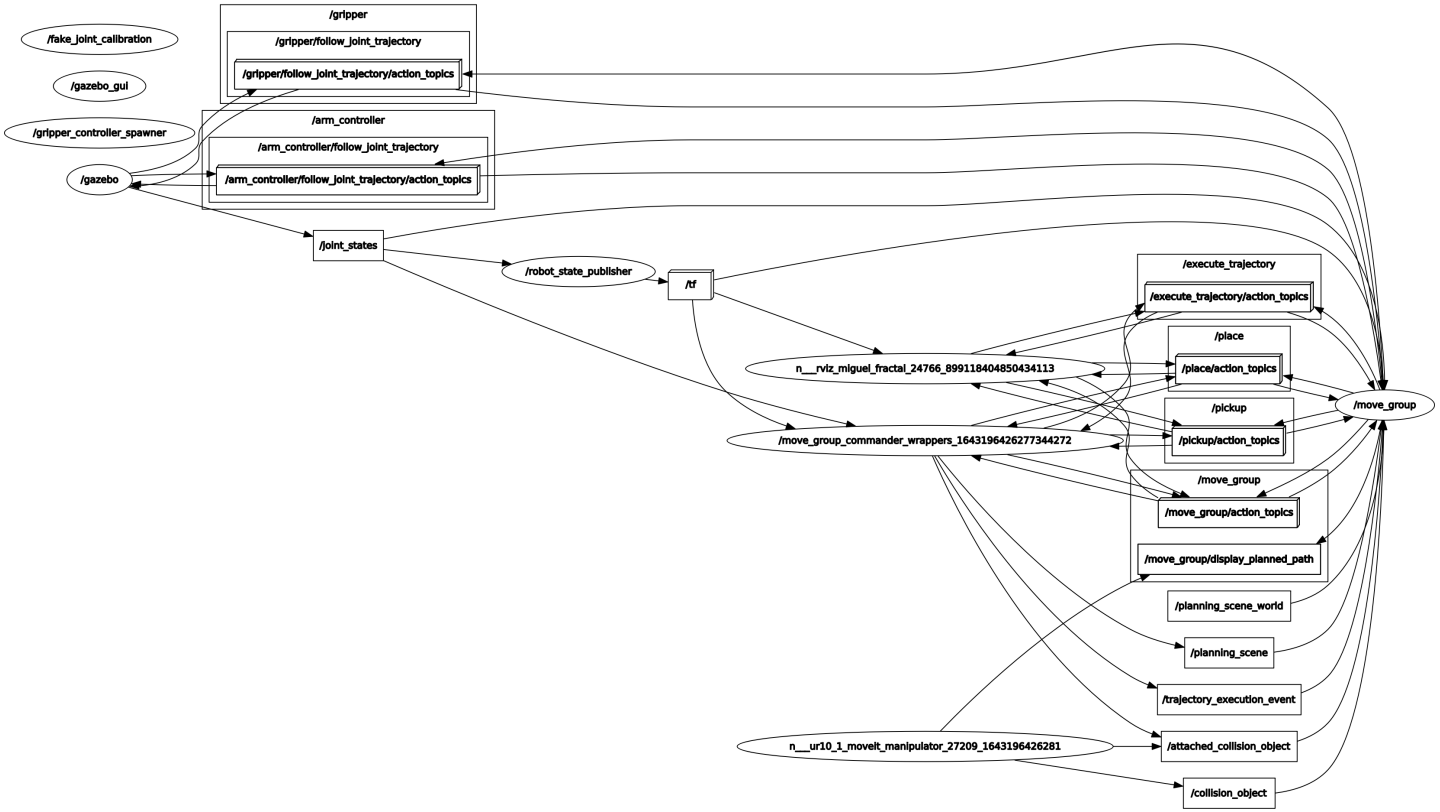


Figura C.11: Fase 3: Comunicación entre MoveIt! y Gazebo, realizando un simple *pick & place* (mejor definición)

C.1.8. Fase 5: Comunicación entre MoveIt!, Gazebo y Leap Motion

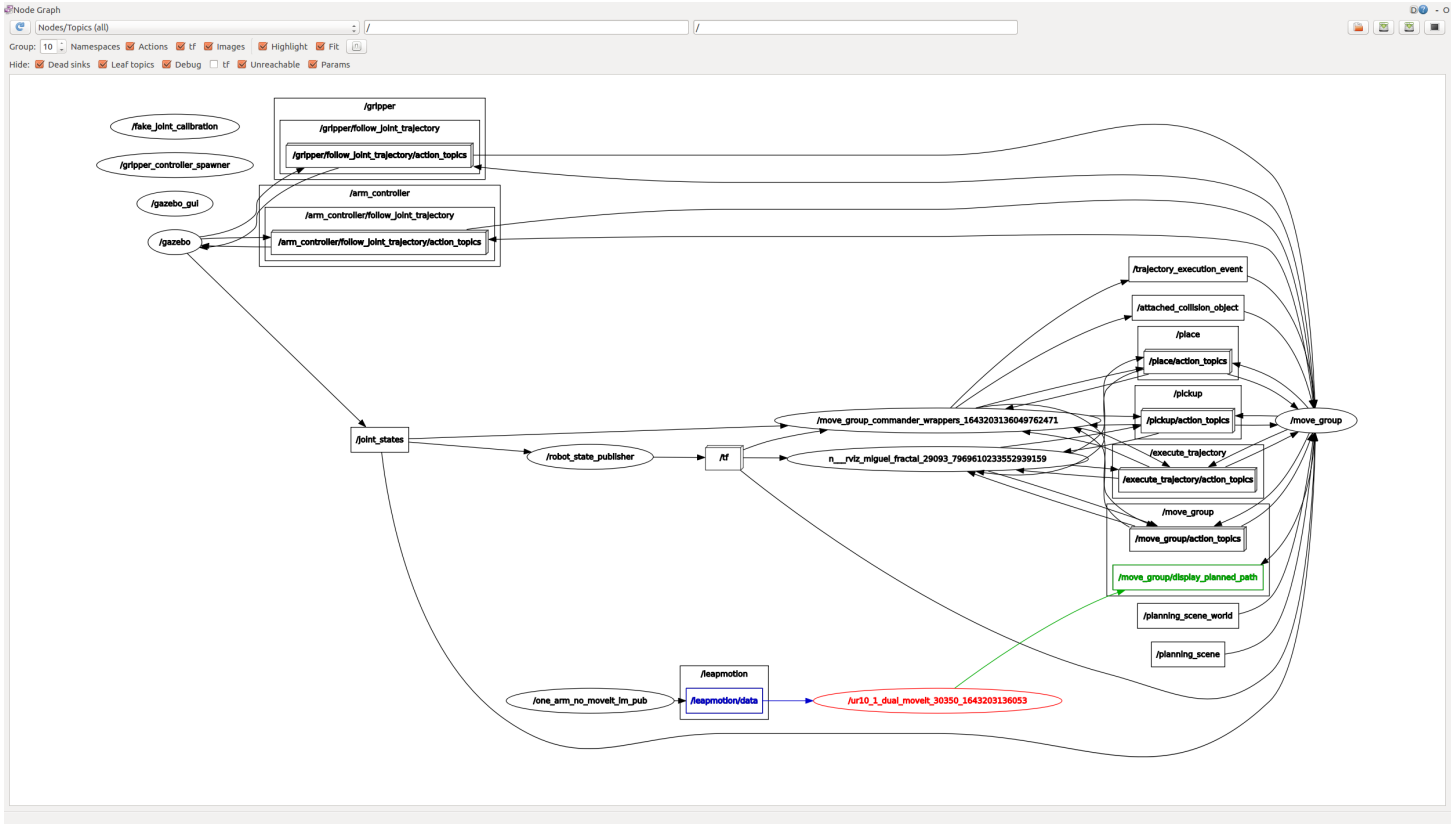


Figura C.12: Fase 5: Comunicación entre MoveIt!, Gazebo y Leap Motion

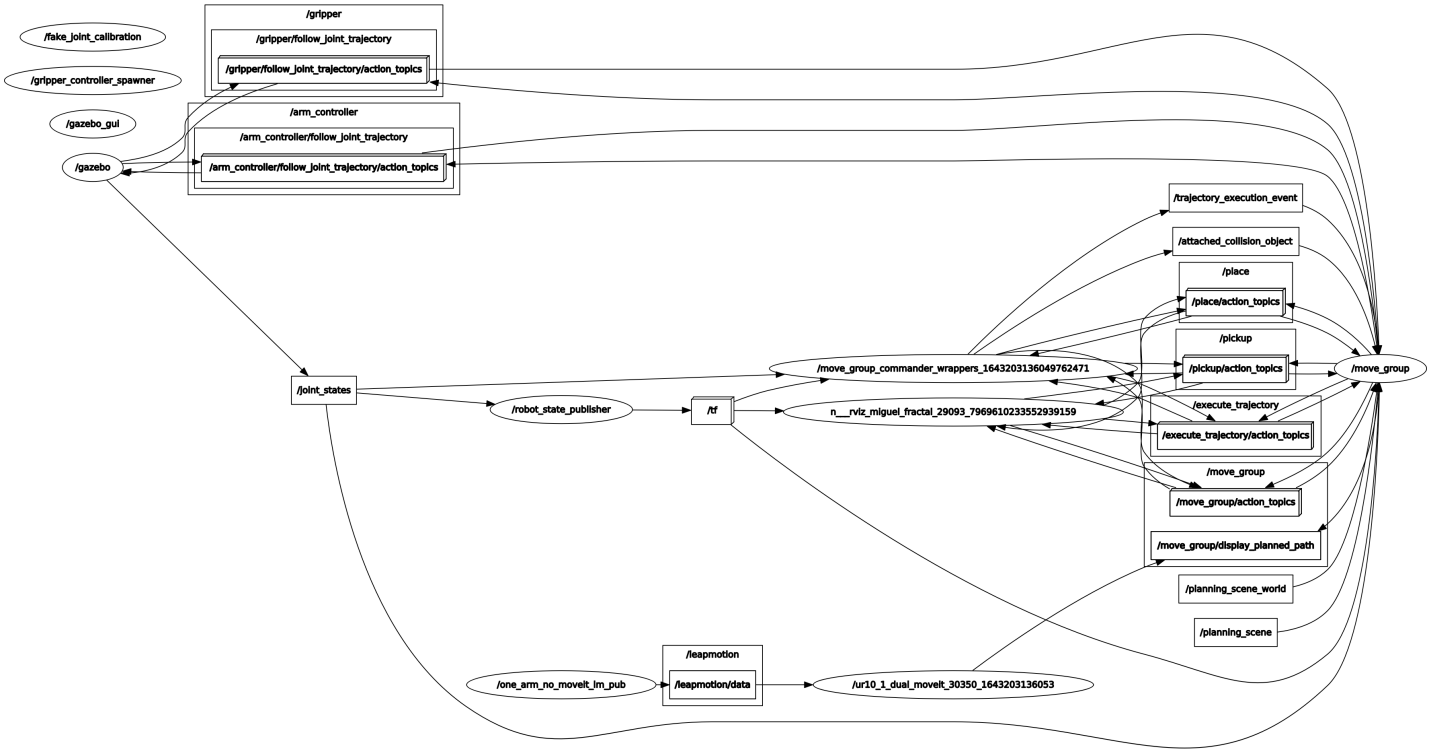


Figura C.13: Fase 5: Comunicación entre MoveIt!, Gazebo y Leap Motion (mejor definición)

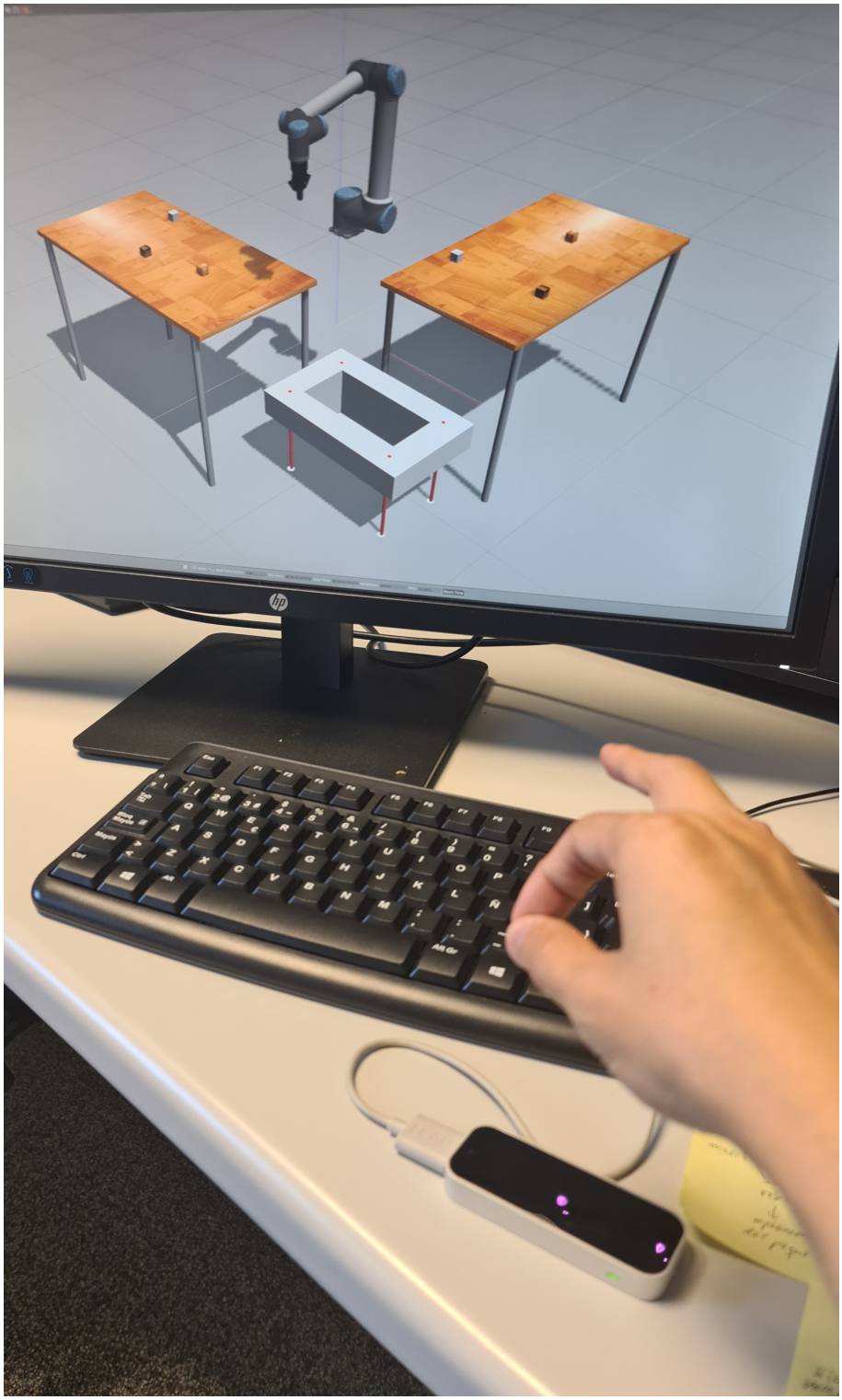


Figura C.14: Fase 5: Comunicación visual entre MoveIt!, Gazebo y Leap Motion

C.2. Solución desarrollada con el paquete de MoveIt! para dos o más cobots

Gráficos en detalle de la solución desarrollada para dos o más cobots en la Subsección 6.1.2 de la Sección 6.1 del Capítulo 6, de la solución desarrollada con el paquete MoveIt!.

C.2.1. Fase 3: Visualización del árbol de transformadas para dos cobots en la solución con Moveit!

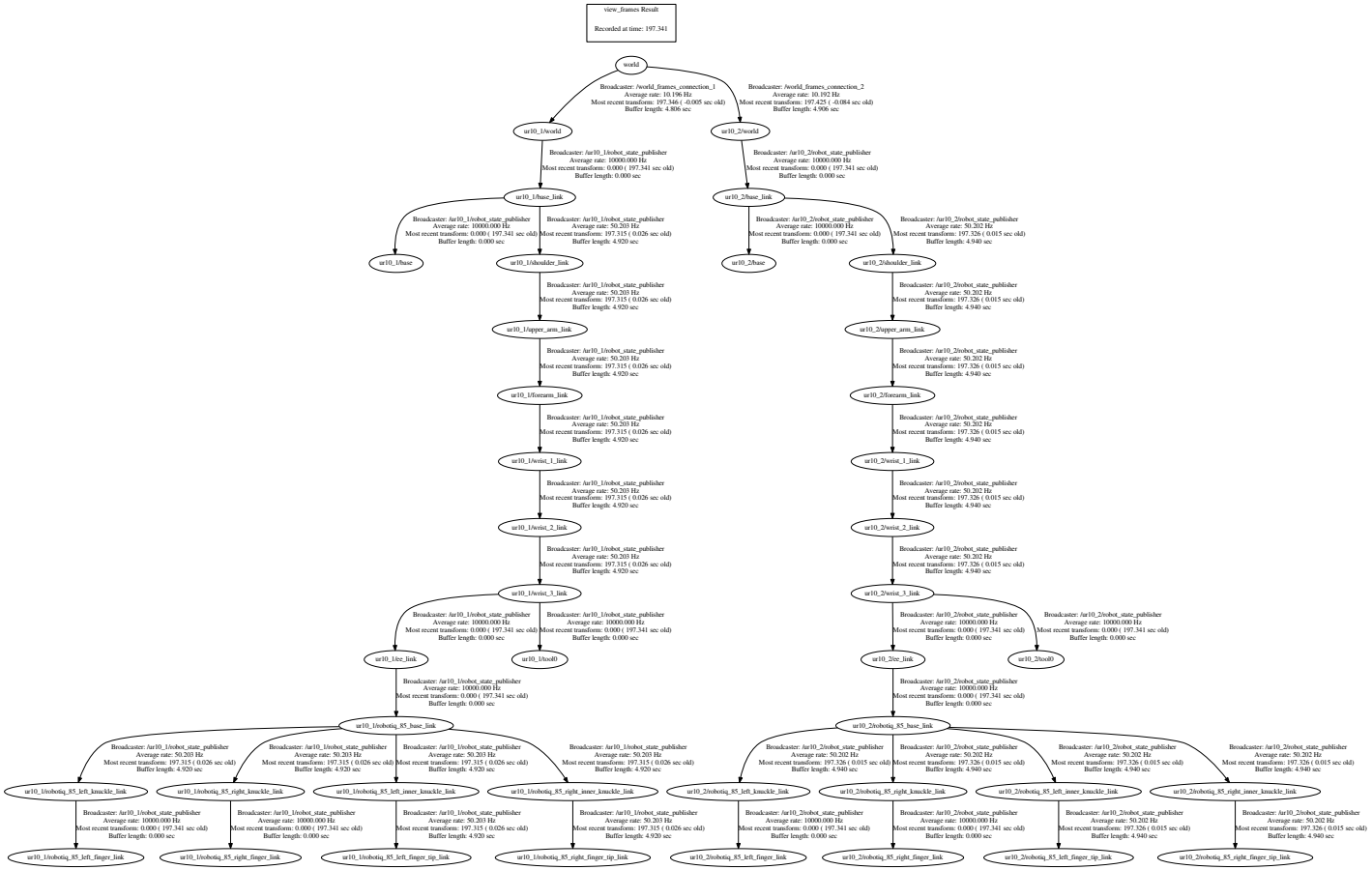


Figura C.15: Fase 3: Visualización del árbol de transformadas para dos cobots en la solución con Moveit!

C.2.2. Fase 3: Nodos y topics de Gazebo para dos cobots en la solución con Moveit!

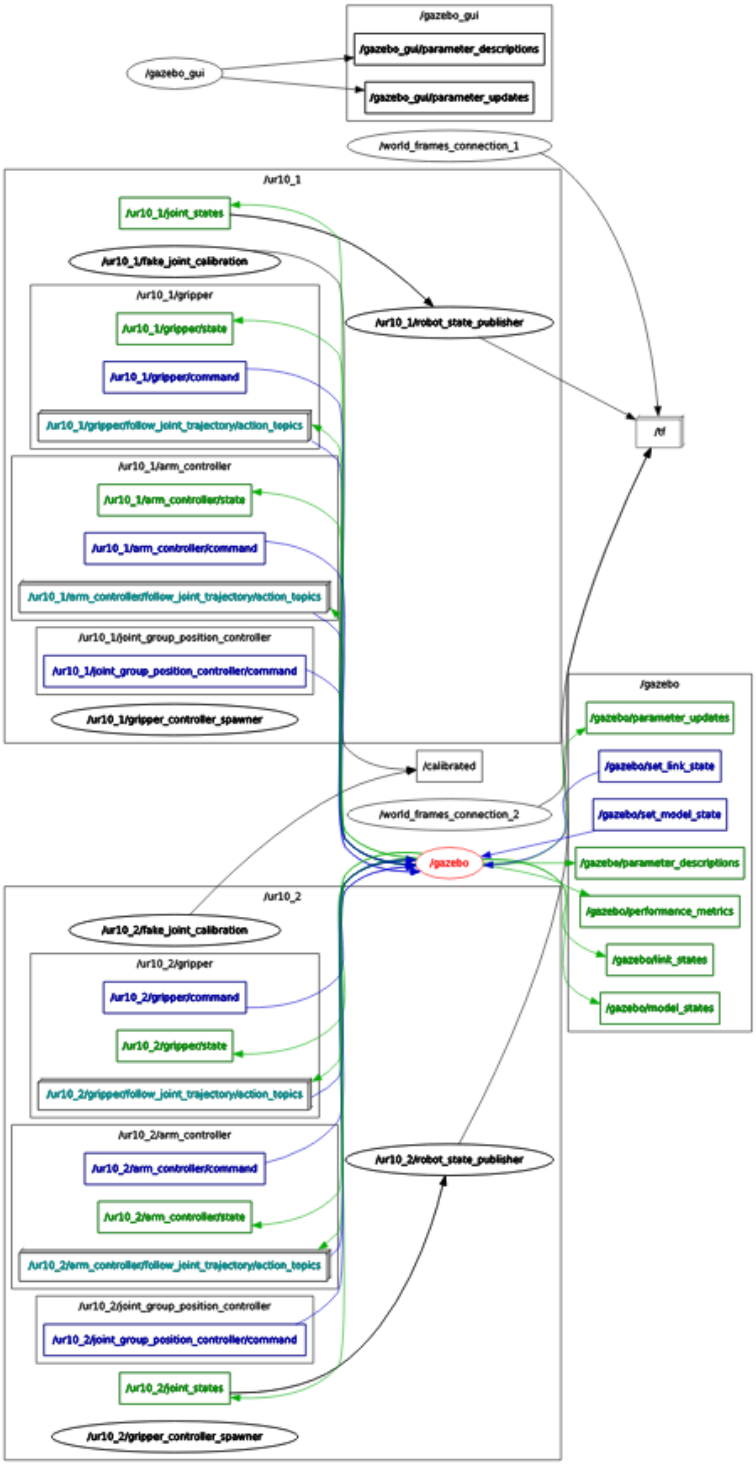


Figura C.16: Fase 3: Nodos y topics de Gazebo para dos cobots en la solución con Moveit!

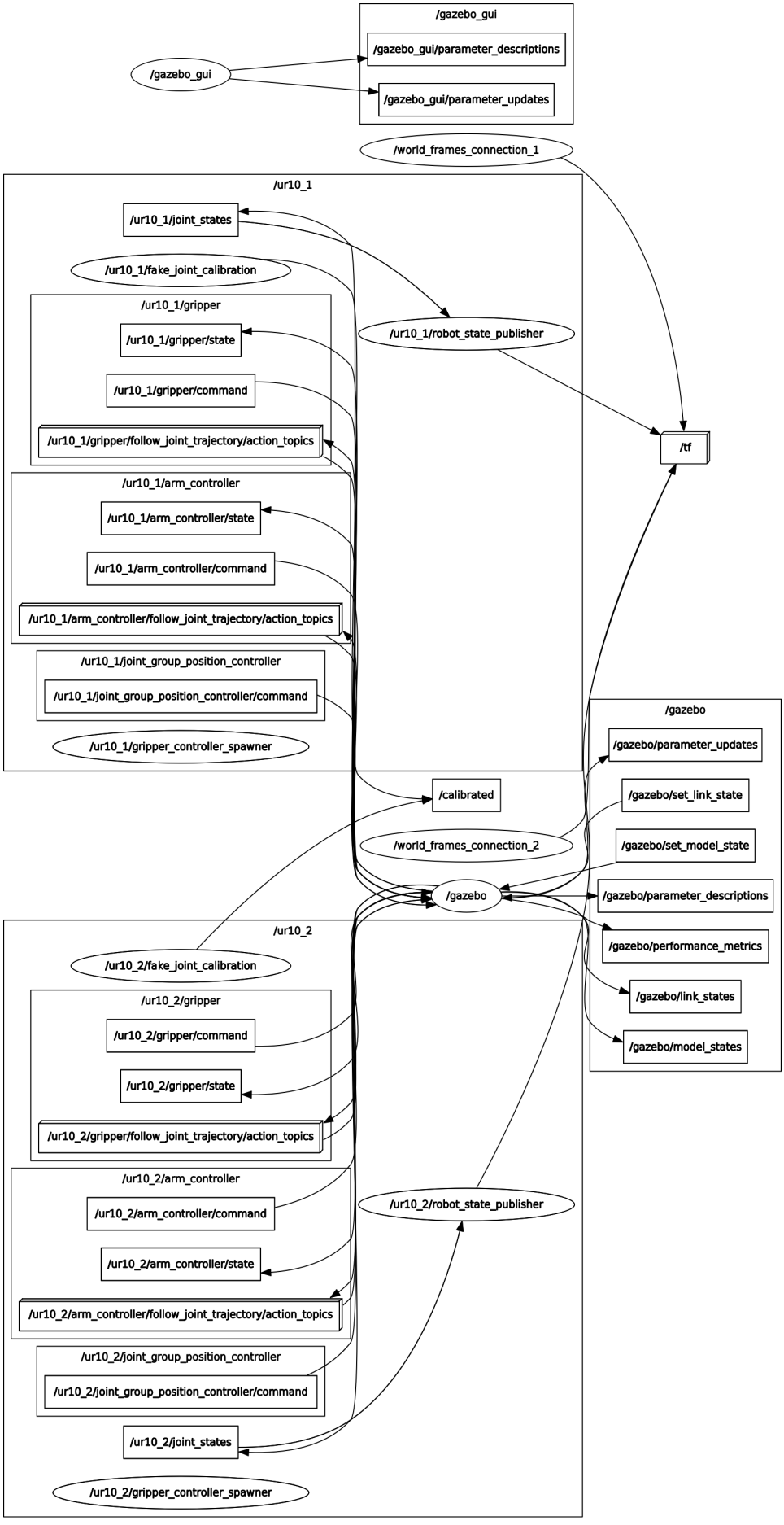


Figura C.17: Fase 3: Nodos y topics de Gazebo para dos cobots en la solución con Moveit! (mejor definición)

C.2.3. Fase 3: Nodos y topics de la comunicación entre Gazebo y MoveIt! para dos cobots en la solución con Moveit!

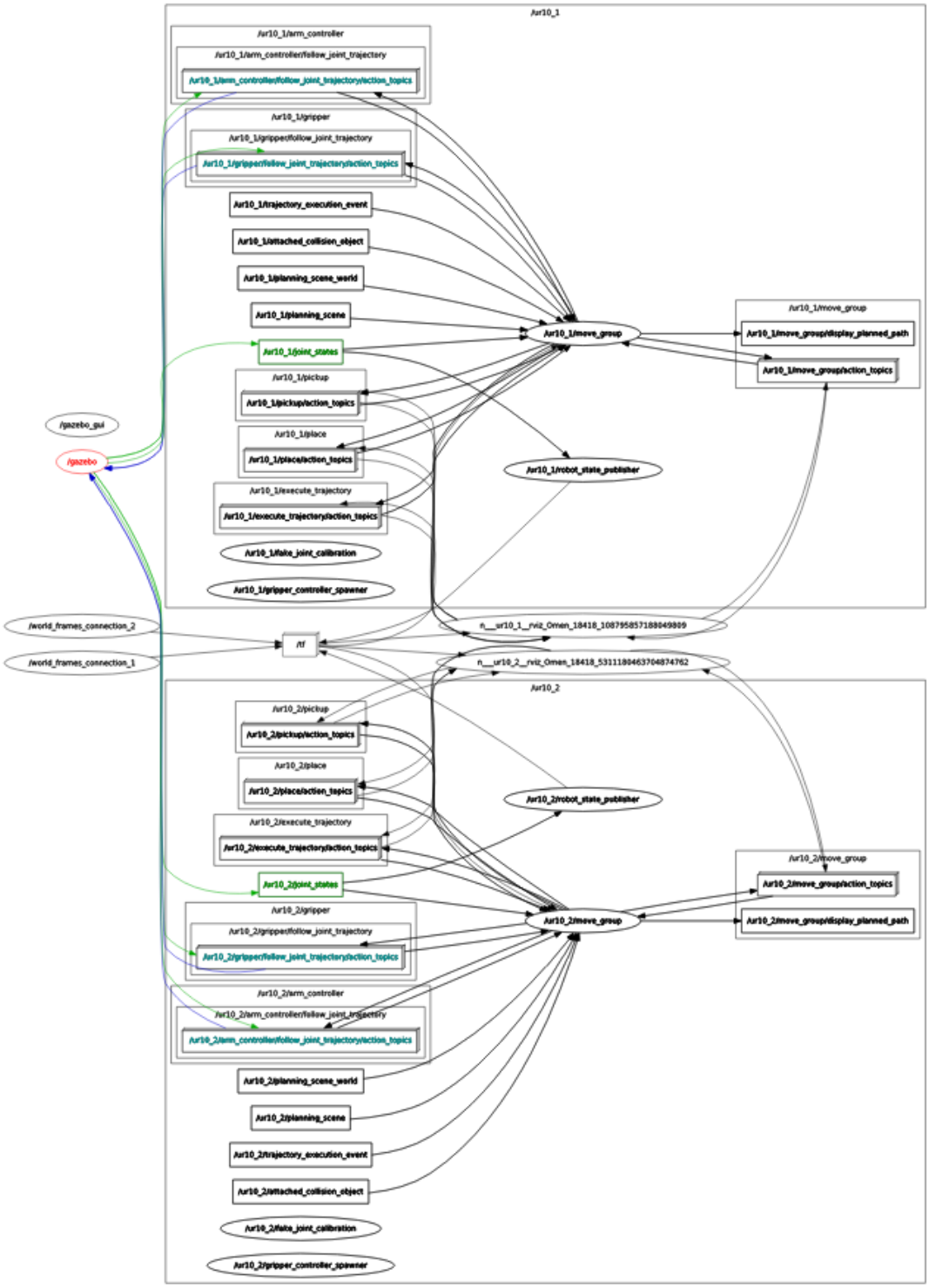


Figura C.18: Fase 3: Nodos y topics de la comunicación entre Gazebo y MoveIt! para dos cobots en la solución con Moveit!

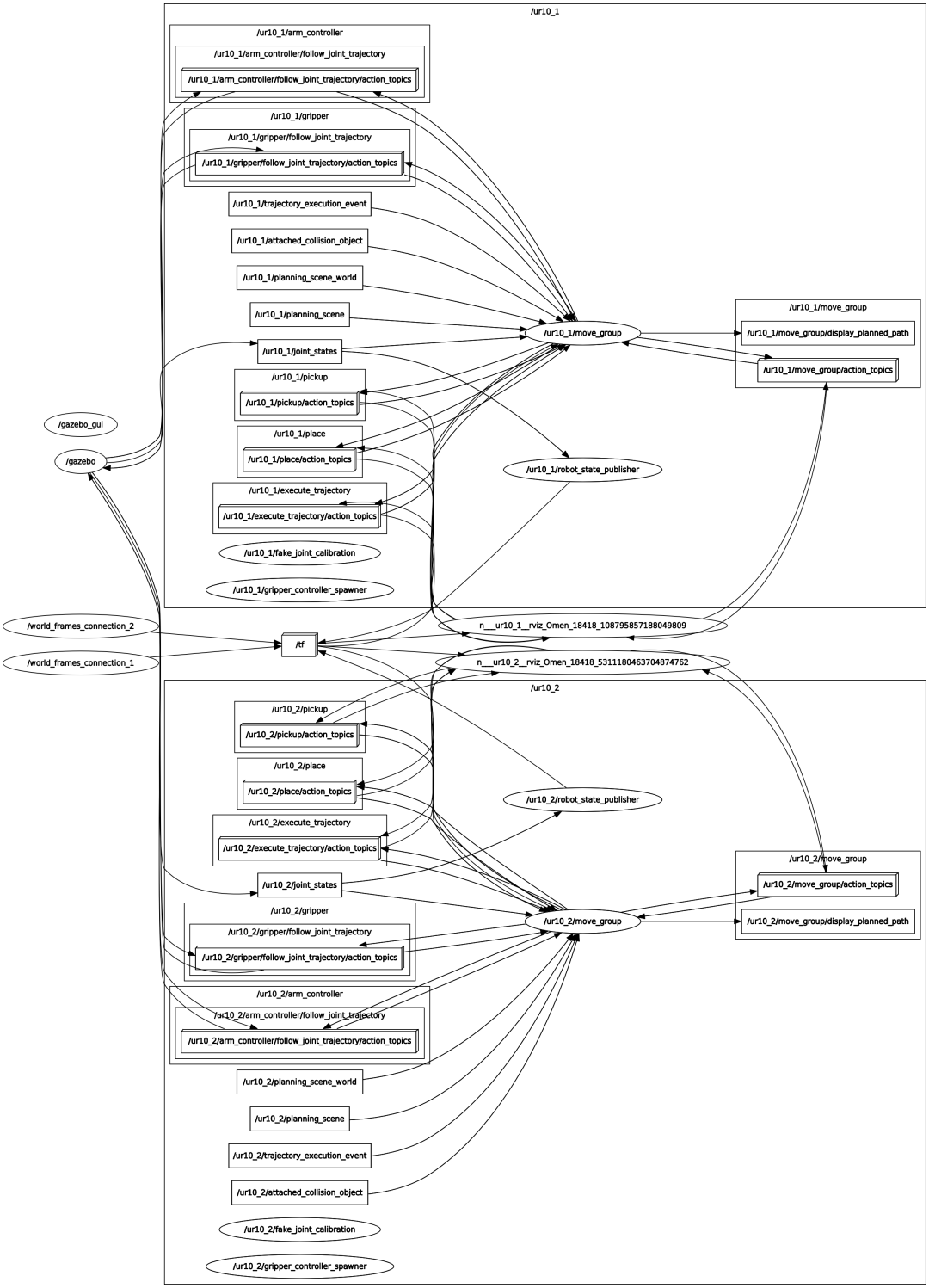


Figura C.19: Fase 3: Nodos y topics de la comunicación entre Gazebo y MoveIt! para dos cobots en la solución con Moveit! (mejor definición)

C.2.4. Fase 3: Nodos y topics de la comunicación entre Gazebo y el planificador! para un único cobot en la solución sin Moveit!

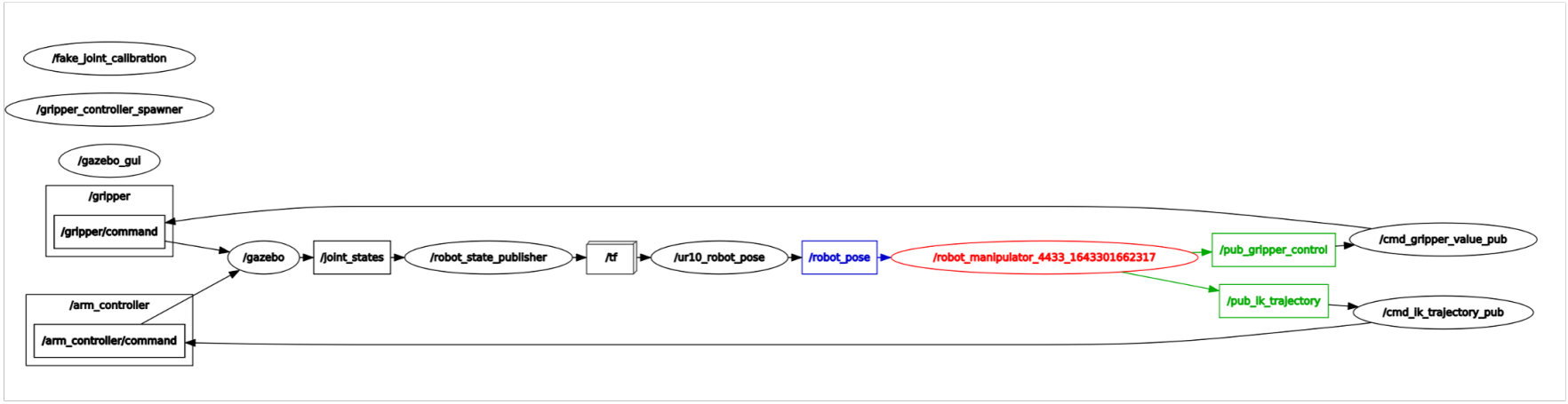


Figura C.20: Fase 3: Nodos y topics del planificador propio para un único cobot

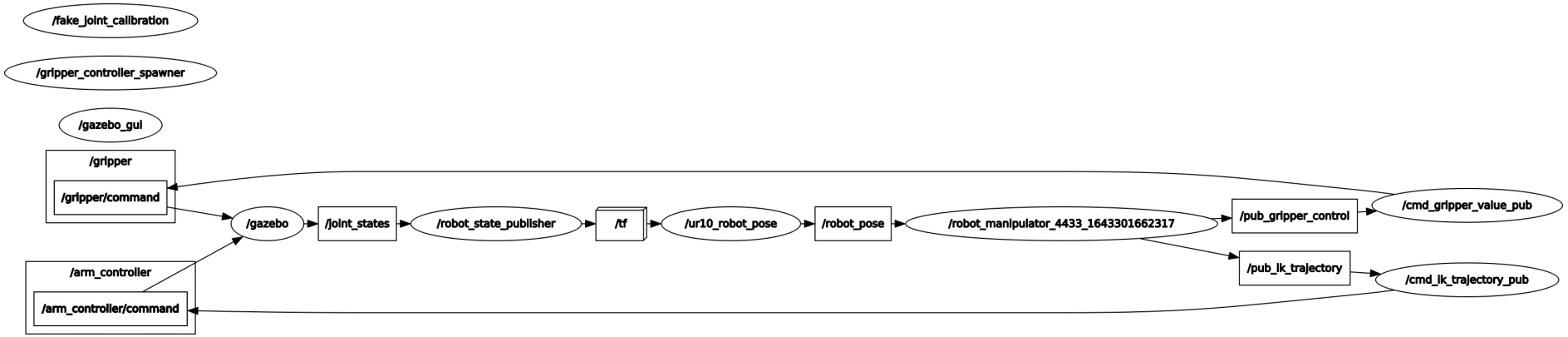


Figura C.21: Fase 3: Nodos y topics del planificador propio para un único cobot (mejor definición)

C.2.5. Fase 3: Nodos y topics de la comunicación entre Gazebo y el planificador! para dos cobots en la solución sin Moveit!

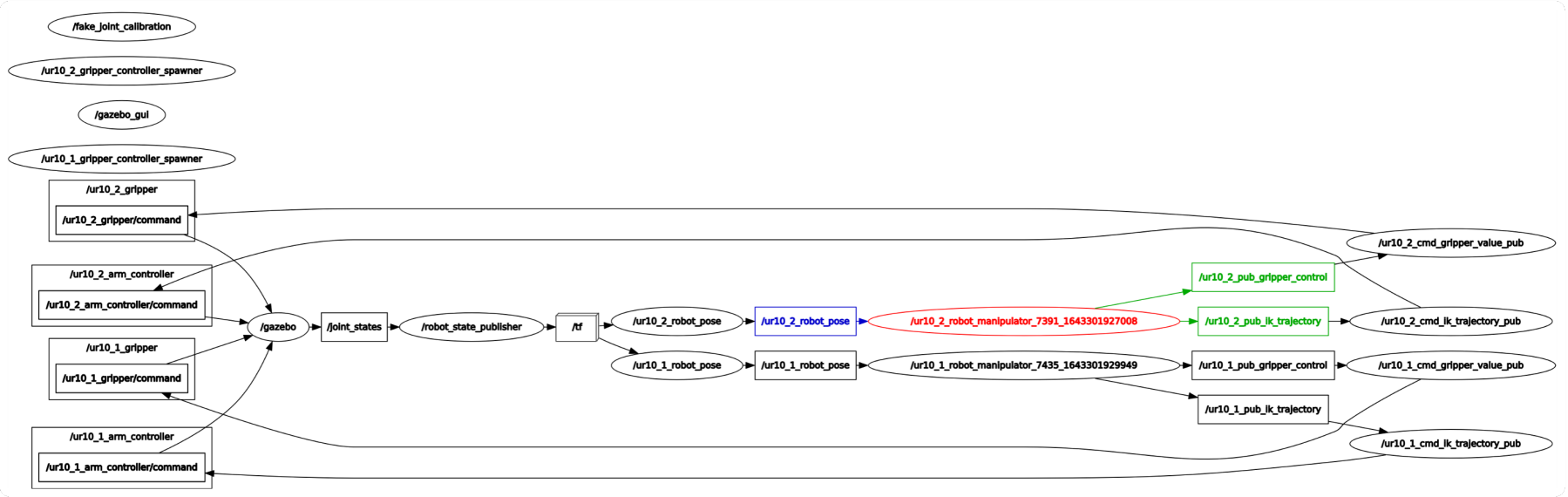


Figura C.22: Fase 3: Nodos y topics del planificador propio para dos cobots

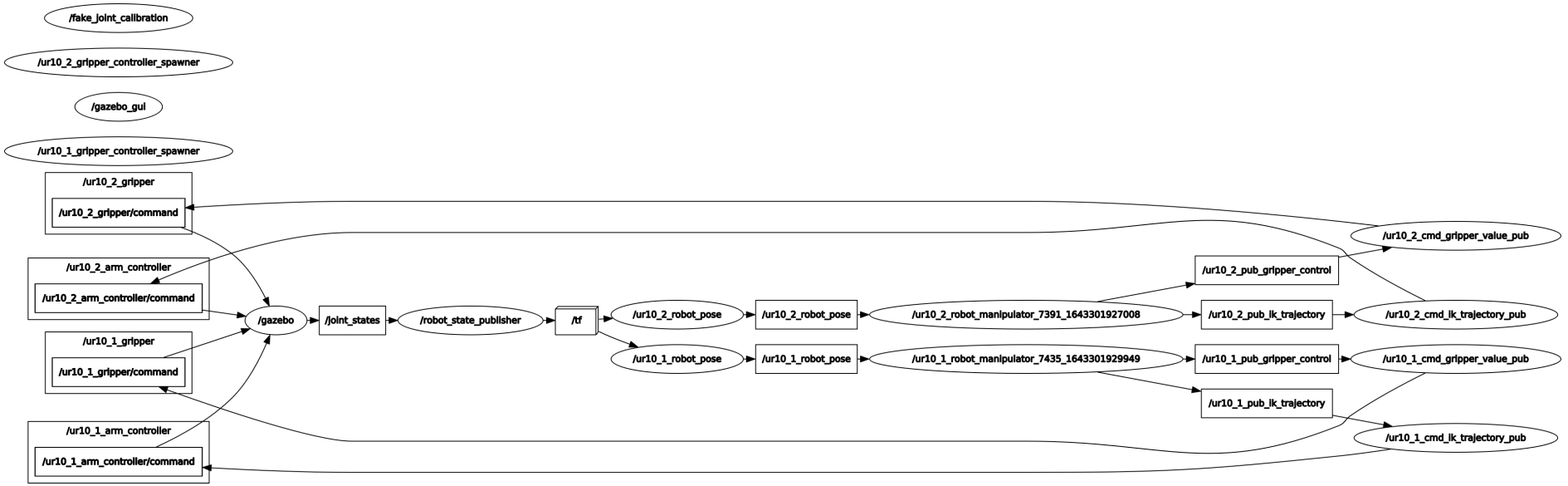


Figura C.23: Fase 3: Nodos y topics del planificador propio para dos cobots (mejor definición)

Anexos D

Forward and inverse Kinematics

Para poder implementar el planificar es imprescindible controlar la cinemática, pero no es el objetivo de este trabajo. Solo es necesario comprender qué hace, los datos que necesita de entrada y lo que se obtiene. Por ello el código se ha obtenido de un curso realizado en *The Construct* [36], el título del curso es *Basic Arm Kinematic Real Robot Project*. En este anexo se va a seguir la explicación que se dio en el curso para la implementación de la cinemática pero adaptándolo para el cobot UR10.

Los pasos que se siguieron fueron los siguientes:

– Localizar los link y los joints del brazo del UR10 e identificar los ejes para cada uno de los joints.

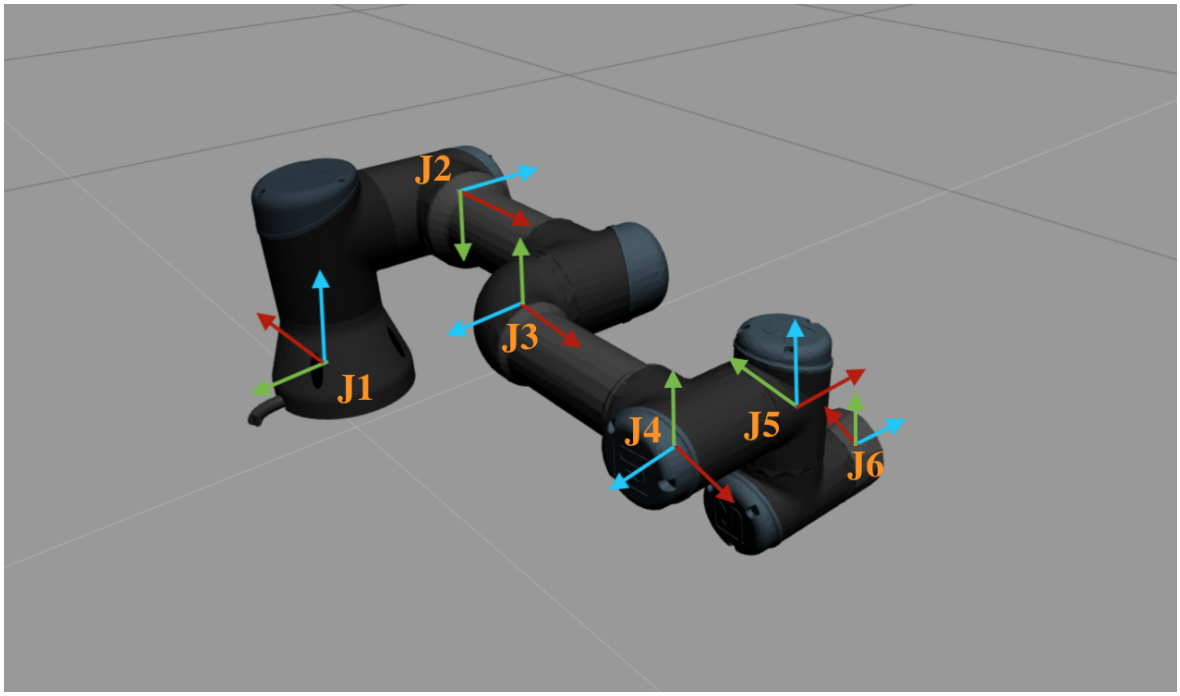


Figura D.1: Representación de los ejes de cada uno de los joints del UR10

– Crear la tabla de los parámetros DH para el UR10: Esta información se encuentra en el URDF del robot UR10:

```
# DH Parameters from ur10.urdf.xacro file
<xacro:property name="d1" value="0.1273" />
<xacro:property name="a2" value="-0.612" />
<xacro:property name="a3" value="-0.5723" />
<xacro:property name="d4" value="0.163941" />
<xacro:property name="d5" value="0.1157" />
<xacro:property name="d6" value="0.0922" />
```

Código Fuente D.1: Información de los valores de los parámetros DH del fichero `ur10.urdf.xacro`

– Generar las matrices de las transformadas para cada uno de los *frames* del robot, y generar la matriz de la transformada para cada uno de los seis joints.

$$T_0^1 = \begin{bmatrix} cos(\theta_1) & -sin(\theta_1) & 0 & 0 \\ sin(\theta_1) & cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad T_2^3 = \begin{bmatrix} cos(\theta_3) & -sin(\theta_3) & 0 & a_2 \\ sin(\theta_3) & cos(\theta_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad T_4^5 = \begin{bmatrix} cos(\theta_5) & -sin(\theta_5) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ sin(\theta_5) & cos(\theta_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$T_1^2 = \begin{bmatrix} cos(\theta_2) & -sin(\theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -sin(\theta_2) & -cos(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad T_3^4 = \begin{bmatrix} cos(\theta_4) & -sin(\theta_4) & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ -sin(\theta_4) & -cos(\theta_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad T_5^6 = \begin{bmatrix} cos(\theta_6) & -sin(\theta_6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -sin(\theta_6) & -cos(\theta_6) & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Figura D.2: Matrices de las transformadas para cada uno de los joints del UR10

– Traduciendo las matrices a Python, se obtiene el siguiente código que es el *Forward Kinematic*:

```
1  #!/usr/bin/env python
2
3  import sys
4  import copy
5  import rospy
6
7  import numpy as np
8  import tf.transformations as tf
9  from math import *
10 import cmath
11 from geometry_msgs.msg import Pose, Quaternion
12
13 # d (unit: mm)
14 d1 = 0.1273
15 d2 = d3 = 0
16 d4 = 0.163941
17 d5 = 0.1157
18 d6 = 0.0922
19
20 # a (unit: mm)
21 a1 = a4 = a5 = a6 = 0
22 a2 = -0.612
23 a3 = -0.5723
24
25
26 # List type of D-H parameter
27 d = np.array([d1, d2, d3, d4, d5, d6]) # unit: mm
```

```
a = np.array([a1, a2, a3, a4, a5, a6]) # unit: mm
alpha = np.array([pi/2, 0, 0, pi/2, -pi/2, 0]) # unit: radian


# Auxiliary Functions

def ur2ros(ur_pose):
    """Transform pose from UR format to ROS Pose format.
    Args:
        ur_pose: A pose in UR format [px, py, pz, rx, ry, rz]
        (type: list)
    Returns:
        An HTM (type: Pose).
    """

    # ROS pose
    ros_pose = Pose()

    # ROS position
    ros_pose.position.x = ur_pose[0]
    ros_pose.position.y = ur_pose[1]
    ros_pose.position.z = ur_pose[2]

    # Ros orientation
    angle = sqrt(ur_pose[3] ** 2 + ur_pose[4] ** 2 + ur_pose[5] ** 2)
    direction = [i / angle for i in ur_pose[3:6]]
    np_T = tf.rotation_matrix(angle, direction)
    np_q = tf.quaternion_from_matrix(np_T)
    ros_pose.orientation.x = np_q[0]
    ros_pose.orientation.y = np_q[1]
    ros_pose.orientation.z = np_q[2]
    ros_pose.orientation.w = np_q[3]

    return ros_pose


def ros2np(ros_pose):
    """Transform pose from ROS Pose format to np.array format.
    Args:
        ros_pose: A pose in ROS Pose format (type: Pose)
    Returns:
        An HTM (type: np.array).
    """

    # orientation
    np_pose = tf.quaternion_matrix([ros_pose.orientation.x, ros_pose.orientation.y, \
                                    ros_pose.orientation.z, ros_pose.orientation.w])

    # position
    np_pose[0][3] = ros_pose.position.x
    np_pose[1][3] = ros_pose.position.y
    np_pose[2][3] = ros_pose.position.z

    return np_pose


def np2ros(np_pose):
    """Transform pose from np.array format to ROS Pose format.
    Args:
        np_pose: A pose in np.array format (type: np.array)
    Returns:
        An HTM (type: Pose).
    """

    # ROS pose
    ros_pose = Pose()

    # ROS position
    ros_pose.position.x = np_pose[0, 3]
    ros_pose.position.y = np_pose[1, 3]
    ros_pose.position.z = np_pose[2, 3]

    # ROS orientation
```

```
    np_q = tf.quaternion_from_matrix(np_pose)
    ros_pose.orientation.x = np_q[0]
    ros_pose.orientation.y = np_q[1]
    ros_pose.orientation.z = np_q[2]
    ros_pose.orientation.w = np_q[3]

    return ros_pose

def filter_sols(q_sols):
    is_sol = False

    shoulder_pan_lower_limit = 0
    shoulder_pan_upper_limit = 4*pi
    shoulder_lift_lower_limit = -pi
    shoulder_lift_upper_limit = 0
    elbow_joint_lower_limit = -pi
    elbow_joint_upper_limit = pi
    wrist_1_lower_limit = -pi
    wrist_1_upper_limit = pi
    wrist_2_lower_limit = -pi
    wrist_2_upper_limit = pi
    wrist_3_lower_limit = -pi
    wrist_3_upper_limit = pi

    if q_sols[0] > shoulder_pan_lower_limit:
        if q_sols[0] < shoulder_pan_upper_limit:
            if q_sols[1] > shoulder_lift_lower_limit:
                if q_sols[1] < shoulder_lift_upper_limit:
                    if q_sols[2] > elbow_joint_lower_limit:
                        if q_sols[2] < elbow_joint_upper_limit:
                            if q_sols[3] > wrist_1_lower_limit:
                                if q_sols[3] < wrist_1_upper_limit:
                                    if q_sols[4] > wrist_2_lower_limit:
                                        if q_sols[4] < wrist_2_upper_limit:
                                            if q_sols[5] > wrist_3_lower_limit:
                                                if q_sols[5] < wrist_3_upper_limit:
                                                    is_sol = True

    print(is_sol)
    return is_sol

def select(q_sols, q_d, w=[1]*6):
    """Select the optimal solutions among a set of feasible joint value
       solutions.
    Args:
        q_sols: A set of feasible joint value solutions (unit: radian)
        q_d: A list of desired joint value solution (unit: radian)
        w: A list of weight corresponding to robot joints
    Returns:
        A list of optimal joint value solution.
    """

    error = []
    #print("in select function")
    #print q_sols
    for q in q_sols:
        #print("in select function 2")
        #if filter_sols(q):
            #print("in select function 3")
            #print q
        error.append(sum([w[i] * (q[i] - q_d[i]) ** 2 for i in range(6)]))

#    print("in select function 4")
    return q_sols[error.index(min(error))]


def HTM(i, theta):
    """Calculate the HTM between two links.
    Args:
        i: A target index of joint value.
        theta: A list of joint value solution. (unit: radian)
    Returns:
        An HTM of Link l w.r.t. Link l-1, where l = i + 1.
```

```
174     """
175
176     Rot_z = np.matrix(np.identity(4))
177     Rot_z[0, 0] = Rot_z[1, 1] = cos(theta[i])
178     Rot_z[0, 1] = -sin(theta[i])
179     Rot_z[1, 0] = sin(theta[i])
180
181     Trans_z = np.matrix(np.identity(4))
182     Trans_z[2, 3] = d[i]
183
184     Trans_x = np.matrix(np.identity(4))
185     Trans_x[0, 3] = a[i]
186
187     Rot_x = np.matrix(np.identity(4))
188     Rot_x[1, 1] = Rot_x[2, 2] = cos(alpha[i])
189     Rot_x[1, 2] = -sin(alpha[i])
190     Rot_x[2, 1] = sin(alpha[i])
191
192     A_i = Rot_z * Trans_z * Trans_x * Rot_x
193
194     return A_i
195
196
197 # Forward Kinematics
198
199 def fwd_kin(theta, i_unit='r', o_unit='n'):
200     """Solve the HTM based on a list of joint values.
201     Args:
202         theta: A list of joint values. (unit: radian)
203         i_unit: Output format. 'r' for radian; 'd' for degree.
204         o_unit: Output format. 'n' for np.array; 'p' for ROS Pose.
205     Returns:
206         The HTM of end-effector joint w.r.t. base joint
207     """
208
209     T_06 = np.matrix(np.identity(4))
210
211     if i_unit == 'd':
212         theta = [radians(i) for i in theta]
213
214     for i in range(6):
215         T_06 *= HTM(i, theta)
216
217     if o_unit == 'n':
218         return T_06
219     elif o_unit == 'p':
220         return np2ros(T_06)
```

Código Fuente D.2: Forward Kinematic en Python

– Calcular su *Invers Kinematic*, es decir obtener el valor del ángulo de cada uno de los joints:

$$\theta_1 = atan2(p_x, p_y)$$

$$\theta_2 = \theta_{23} - \theta_3$$

$$\theta_3 = -atan2(K, \pm\sqrt{d_4^2 - K^2}$$

$$\theta_4 = atan2(-r13 * s1 + r_{23} * c1, -r_{13} * c1 * c23 - r_{23} * s1 * c23 + r_{33} * s23)$$

$$\theta_5 = atan2(s5, c5)$$

$$\theta_6 = atan2(s6, c6)$$

Figura D.3: Cálculo de cada uno de los ángulos de los joints del UR10

– Traduciendo las ecuaciones a Python, se obtiene el siguiente código que es el *Inverse Kinematic*:

```
# Inverse Kinematics

def inv_kin(p, q_d, i_unit='r', o_unit='r'):
    """Solve the joint values based on an HTM.
    Args:
        p: A pose.
        q_d: A list of desired joint value solution
             (unit: radian).
        i_unit: Output format. 'r' for radian; 'd' for degree.
        o_unit: Output format. 'r' for radian; 'd' for degree.
    Returns:
        A list of optimal joint value solution.
    """

    # Preprocessing
    if type(p) == Pose: # ROS Pose format
        T_06 = ros2np(p)
    elif type(p) == list: # UR format
        T_06 = ros2np(ur2ros(p))

    if i_unit == 'd':
        q_d = [radians(i) for i in q_d]

    # Initialization of a set of feasible solutions
    theta = np.zeros((8, 6))

    # theta1
    P_05 = T_06[0:3, 3] - d6 * T_06[0:3, 2]
    phi1 = atan2(P_05[1], P_05[0])
```

```
    phi2 = acos(d4 / sqrt(P_05[0] ** 2 + P_05[1] ** 2))
    theta1 = [pi / 2 + phi1 + phi2, pi / 2 + phi1 - phi2]
    theta[0:4, 0] = theta1[0]
    theta[4:8, 0] = theta1[1]

    # theta5
    P_06 = T_06[0:3, 3]
    theta5 = []
    for i in range(2):
        theta5.append(
            acos((P_06[0] * sin(theta1[i]) - P_06[1] * cos(theta1[i]) - d4) / d6)
                    )
    for i in range(2):
        theta[2*i, 4] = theta5[0]
        theta[2*i+1, 4] = -theta5[0]
        theta[2*i+4, 4] = theta5[1]
        theta[2*i+5, 4] = -theta5[1]

    # theta6
    T_60 = np.linalg.inv(T_06)
    theta6 = []
    for i in range(2):
        for j in range(2):
            s1 = sin(theta1[i])
            c1 = cos(theta1[i])
            s5 = sin(theta5[j])
            theta6.append(atan2((-T_60[1, 0] * s1 + T_60[1, 1] * c1) / s5,
                                (T_60[0, 0] * s1 - T_60[0, 1] * c1) / s5))
    for i in range(2):
        theta[i, 5] = theta6[0]
        theta[i+2, 5] = theta6[1]
        theta[i+4, 5] = theta6[2]
        theta[i+6, 5] = theta6[3]

    # theta3, theta2, theta4
    for i in range(8):
        # theta3
        T_46 = HTM(4, theta[i]) * HTM(5, theta[i])
        T_14 = np.linalg.inv(HTM(0, theta[i])) * T_06 * np.linalg.inv(T_46)
        P_13 = T_14 * np.array([[0, -d4, 0, 1]]).T - np.array([[0, 0, 0, 1]]).T
        if i in [0, 2, 4, 6]:
            theta[i, 2] = -cmath.acos(
                    (np.linalg.norm(P_13) ** 2 - a2 ** 2 - a3 ** 2) / (2 * a2 * a3)
                                    ).real

            theta[i+1, 2] = -theta[i, 2]

        # theta2
        theta[i, 1] = -atan2(P_13[1], -P_13[0]) + asin(
                            a3 * sin(theta[i, 2]) / np.linalg.norm(P_13)
                                                    )

        # theta4
        T_13 = HTM(1, theta[i]) * HTM(2, theta[i])
        T_34 = np.linalg.inv(T_13) * T_14
        theta[i, 3] = atan2(T_34[1, 0], T_34[0, 0])

    theta = theta.tolist()

    # Select the most close solution
    q_sol = select(theta, q_d)

    # Output format
    if o_unit == 'r': # (unit: radian)
        return q_sol
    elif o_unit == 'd': # (unit: degree)
        return [degrees(i) for i in q_sol]
```

Código Fuente D.3: Inverse Kinematic en Python

Y este es el planificador utilizado en este Trabajo Fin de Grado.

Anexos E

Puesta en marcha del entorno en Ubuntu 16.04

Para la reproducción del proyecto realizado y teniendo en cuenta la posibilidad de una continuación del trabajo, se ha procedido a explicar el proceso paso a paso de la instalación base del sistema con el que se ha trabajado.

E.1. Instalación de ROS (Robotic Operating System)

El único *requisito* necesario para comenzar la instalación es tener el sistema operativo *Ubuntu 16.04* preferiblemente recién instalado.

Desde la terminal en Ubuntu 16.04 y con conexión a Internet, hay que ejecutar los siguientes comandos para instalar *ROS Kinetic Kame*:

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

$ sudo apt install curl # if you haven't already installed curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -

$ sudo apt-get update

$ sudo apt-get install ros-kinetic-desktop-full

$ sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool
build-essential

$ sudo apt install python-rosdep
$ sudo rosdep init
$ rosdep update

$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ source /opt/ros/kinetic/setup.bash
```

Código Fuente E.1: Comandos realizados para la instalación de ROS Kinetic Kame

Para la instalación de las dependencias necesarias antes de instalar los paquetes desde sus *sources*:

```
$ sudo apt-get install ros-kinetic-$(dependencia)
```

Código Fuente E.2: Comando para la instalación dependencias, por ejemplo: *sudo apt-get install ros-kinetic-ros-controllers*

Listado de las dependencias instaladas de ROS:

Dependencias instaladas	
– moveit_ros_planning – moveit_kinematics – moveit_simple_controller_manager – position_controllers – moveit_fake_controller_manager – gazebo_ros_control – joint_trajectory_controller	– moveit_ros_visualization – moveit_planners_ompl – joint_state_controller – effort_controllers – ros_controllers – industrial_msgs

Hay que tener en cuenta que esta instalación es en el año 2022, con el tiempo puede ser necesario otras dependencias no señaladas. Esta es la configuración base antes de instalar los repositorios que se usará en la implementación del proyecto.

E.2. Instalación de los paquetes del proyecto

Se procede a la instalación de los repositorios comunes a todas las soluciones propuestas en la memoria. Los repositorios pueden ser modificados porque siguen activos, por tanto, hay tener cuidado con la versión que se está instalando por posibles problemas de compatibilidad.

Primero se crea el directorio de trabajo:

```
$ mkdir -p ~/tfg_multirobot/src
$ cd  ~/tfg_multirobot/src
```

Código Fuente E.3: Comando para la creación del directorio de trabajo (Workspace)

Una vez que se tiene creado el directorio, se procede a instalar los repositorios de terceros utilizados total o parcialmente:

E.2.1. Universal Robots

Este repositorio de Universal Robots se utilizará como base para realizar las modificaciones necesarias que el proyecto requiera. Los ficheros modificados se señalarán debidamente.

Clonar el repositorio:

```
$ cd ~/tfg_multirobot/src
$ git clone -b kinetic-devel https://github.com/ros-industrial/universal_robot.git
```

Código Fuente E.4: Clonación del repositorio Universal Robots almacenado en Github

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.5: Instalación de las dependencias que el paquete Universal Robots requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.6: Compilación del entorno de trabajo junto al paquete Universal Robots instalado

E.2.2. Robotiq_2finger_grippers

Este repositorio de Robotiq_2finger_grippers se utilizará para añadir la pinza que utilizará el robot real (campero) y los controladores necesarios para su funcionamiento.

Clonar el repositorio:

```
$ cd ~/tfg_multirobot/src
$ git clone https://github.com/Danfoa/robotiq_2finger_grippers.git
```

Código Fuente E.7: Clonación del repositorio Robotiq_2finger_grippers almacenado en Github

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.8: Instalación de las dependencias que el paquete Robotiq_2finger_grippers requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.9: Compilación del entorno de trabajo junto al paquete Robotiq_2finger_grippers instalado

E.2.3. Robotiq_85_gripper

Este repositorio de Robotiq_85_gripper se utilizará para añadir la pinza que se utilizará al robot durante la simulación y los controladores necesarios para su funcionamiento.

Clonar el repositorio:

```
$ cd ~/tfg_multirobot/src
$ git clone -b develop https://github.com/PickNikRobotics/robotiq_85_gripper.git
```

Código Fuente E.10: Clonación del repositorio Robotiq_85_gripper almacenado en Github

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.11: Instalación de las dependencias que el paquete Robotiq_85_gripper requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.12: Compilación del entorno de trabajo junto al paquete Robotiq_85_gripper instalado

E.2.4. ros_control

Este repositorio contiene: Un conjunto de paquetes que incluye controladores de interfaces, gestor de controladores, transmisiones e interfaces del hardware.

Clonar el repositorio:

```
$ cd ~/tfg_multirobot/src
$ git clone -b kinetic-devel https://github.com/ros-controls/ros_control.git
```

Código Fuente E.13: Clonación del repositorio ros_control almacenado en Github

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.14: Instalación de las dependencias que el paquete ros_control requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.15: Compilación del entorno de trabajo junto al paquete ros_control instalado

E.2.5. ur_modern_driver

Este repositorio está obsoleto, pero por compatibilidad con ROS Kinetic Kame hay que utilizarlo, contiene drivers para los robots de Universal Robots (UR3/UR5/UR10) y es compatible con el paquete de ros_control.

Clonar el repositorio:

```
$ cd ~/tfg_multirobot/src
$ git clone -b kinetic-devel https://github.com/ros-industrial/ur_modern_driver.git
```

Código Fuente E.16: Clonación del repositorio ur_modern_driver almacenado en Github

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.17: Instalación de las dependencias que el paquete ur_modern_driver requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.18: Compilación del entorno de trabajo junto al paquete ur_modern_driver instalado

Puede dar error de compilación para ROS Kinetic Kame, para solucionarlo hay que modificar el fichero *ur_hardware_interface.cpp*, para realizarlo rápido, sustituir el contenido del fichero por el contenido que hay en al siguiente link: `https://github.com/iron-ox/ur_modern_driver/blob/883070d0b6c0c32b78bb1ca7155b8f3a1ead416c/src/ur_hardware_interface.cpp` y volverlo a compilar.

E.2.6. gazebo-pkgs

Es una colección de *plugings* para Gazebo, principalmente nos interesa el pluging que permite agarrar objetos.

Dependencias del paquete:

Dependencias del paquete gazebo-pkgs	
– gazebo_ros – eigen_conversions – object_recognition_msgs	– roslint – general-message-pkgs

Instalación de dependencias del paquete gazebo-pkgs:

– Ros Kinetic Kame usa Gazebo 7.x

```
$ sudo apt-get install -y libgazebo7-dev
```

Código Fuente E.19: Instalación de libgazebo7-dev por dependencia del paquete gazebo-pkgs

– Clonar el repositorio de gazebo_ros_pkgs

```
$ cd ~/tfg_multirobot/src
$ git clone -b kinetic-devel https://github.com/ros-simulation/gazebo_ros_pkgs.git
```

Código Fuente E.20: Clonación del repositorio gazebo_ros_pkgs almacenado en Github que el paquete gazebo-pkgs depende

– Clonar el repositorio de eigen_conversions

```
$ cd ~/tfg_multirobot/src
$ git clone -b indigo-devel https://github.com/ros/geometry.git
```

Código Fuente E.21: Clonación del repositorio eigen_conversions almacenado en Github que el paquete gazebo-pkgs depende

– Clonar el repositorio de object_recognition_msgs

```
$ cd ~/tfg_multirobot/src
$ git clone -b master https://github.com/wg-perception/object_recognition_msgs.git
```

Código Fuente E.22: Clonación del repositorio object_recognition_msgs almacenado en Github que el paquete gazebo-pkgs depende

– Clonar el repositorio de roslint

```
$ cd ~/tfg_multirobot/src
$ git clone -b master https://github.com/ros/roslint.git
```

Código Fuente E.23: Clonación del repositorio roslint almacenado en Github que el paquete gazebo-pkgs depende

– Clonar el repositorio de general-message-pkgs

```
$ cd ~/tfg_multirobot/src
$ git clone -b master https://github.com/JenniferBuehler/general-message-pkgs.git
```

Código Fuente E.24: Clonación del repositorio general-message-pkgs almacenado en Github que el paquete gazebo-pkgs depende

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep check --from-paths . --ignore-src --rosdistro kinetic
$ rosdep install --from-paths . --ignore-src --rosdistro kinetic -y
```

Código Fuente E.25: Instalación de las dependencias de los paquetes clonados para poder instalar el paquete gazebo-pkgs

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.26: Compilación del entorno de trabajo junto a las dependencias del paquete gazebo-pkgs

Clonar el repositorio gazebo-pkgs:

```
$ cd ~/tfg_multirobot/src
$ git clone https://github.com/JenniferBuehler/gazebo-pkgs.git
```

Código Fuente E.27: Clonación del repositorio gazebo-pkgs almacenado en Github

Instalación de dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.28: Instalación de las dependencias que el paquete gazebo-pkgs requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.29: Compilación del entorno de trabajo junto al paquete gazebo-pkgs instalado

E.2.7. Posible error durante la compilación de los paquetes del proyecto

Durante la compilación puede aparecer un *error* que pide que debe estar compilado con el estándar de c++11:

```
/usr/include/c++/5/bits/c++0x_warning.h:32:2: error: #error This file requires compiler and
    library support for the ISO C++ 2011 standard. This support must be enabled with the
    -std=c++11 or -std=gnu++11 compiler options.
```

Código Fuente E.30: Error: This file requires compiler and library support for the ISO C++ 2011 standard

Para solventarlo, hay que modificar el fichero *CMakeLists.txt* de *catkin* al compilar el proyecto, simplemente hay que modificar el fichero ~/tfg_multirobot/src/CMakeLists.txt y añadir `add_compile_options(-std=c++11)` al principio del fichero, como se muestra a continuación:

```
# toplevel CMakeLists.txt for a catkin workspace
# catkin/cmake/toplevel.cmake

cmake_minimum_required(VERSION 3.0.2)

project(Project)

set(CATKIN_TOPLEVEL TRUE)
add_compile_options(-std=c++11)
[...]
```

Código Fuente E.31: Solución: This file requires compiler and library support for the ISO C++ 2011 standard

E.2.8. Leap Motion

Driver de ROS para el controlador de Leap Motion, para la correcta instalación y configuración del controlador de Leap Motion en ROS Kinetic Kame, hay que realizar un poco más de trabajo que en los repositorios previos.

Lo primero es reemplazar o crear el fichero que dar servicio al controlador: */lib/systemd/system/leapd.service*

```
[Unit]
Description=LeapMotion Daemon
After=syslog.target

[Service]
Type=simple
ExecStart=/usr/sbin/leapd

[Install]
WantedBy=multi-user.target
```

Código Fuente E.32: Configurando manualmente el servicio de Leap Motion

Después hay que crear un acceso directo en el directorio */etc/systemd/system/* y lanzarlo:

```
$ sudo ln -s /lib/systemd/system/leapd.service /etc/systemd/system/leapd.service
$ sudo systemctl daemon-reload
```

Código Fuente E.33: Creación del acceso directo al servicio de Leap Motion

Clonar el repositorio:

```
$ cd ~/tfg_multirobot/src
$ git clone https://github.com/ros-drivers/leap_motion.git
```

Código Fuente E.34: Clonación del repositorio leap_motion almacenado en Github

En los pasos de instalación dice de mover el directorio LeapSDK al directorio `$HOME`, pero se va a mantener en el repositorio original y se modificarán los `PATHS` adecuadamente:

```
# 64-bit operating system
$ echo "export PYTHONPATH=$PYTHONPATH:$HOME/tfg_multirobot/src/leap_motion/LeapSDK/lib
:$HOME/tfg_multirobot/src/leap_motion/LeapSDK/lib/x64" >> ~/.bashrc
$ source ~/.bashrc
```

Código Fuente E.35: Configuración del PATH al SDK de Leap Motion

Instalación del paquete de ROS de Leap Motion:

```
$ sudo apt-get install ros-kinetic-leap-motion
```

Código Fuente E.36: Instalación del paquete de ROS de Leap Motion

Instalación de las dependencias que puedan faltar para ROS Kinetic Kame:

```
$ cd ~/tfg_multirobot
$ rosdep update
$ rosdep install --rosdistro kinetic --ignore-src --from-paths src
```

Código Fuente E.37: Instalación de las dependencias que el paquete leap_motion requiera

Compilación:

```
$ cd ~/tfg_multirobot
$ catkin_make
```

Código Fuente E.38: Compilación del entorno de trabajo junto al paquete leap_motion instalado

Si surgen errores o se para durante el uso de los drivers de Leap Motion, con reiniciar el servicio suele ser suficiente:

```
$ sudo service leapd restart
```

Código Fuente E.39: Reiniciar el servicio de Leap Motion

E.2.9. Activación del entorno de trabajo actual

```
$ cd ~/tfg_multirobot
$ source ~/tfg_multirobot/devel/setup.bash
```

Código Fuente E.40: Activación del entorno de trabajo actual

E.3. Tratamiento de los warnings durante la instalación

– Warnings ignorados: *ur_modern_driver*

```
WARNING: Package 'ur_modern_driver' is deprecated (This package has been deprecated.
Users of CB3 and e-Series controllers should migrate to ur_robot_driver.)

CMake Warning at /opt/ros/kinetic/share/catkin/cmake/catkin_package.cmake:418
(message):
  catkin_package() include dir
  '/home/miguel/tfg_multirobot/build/gazebo-pkgs/gazebo_grasp_plugin/..'
  should be placed in the devel space instead of the build space
Call Stack (most recent call first):
  /opt/ros/kinetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  gazebo-pkgs/gazebo_grasp_plugin/CMakeLists.txt:31 (catkin_package)

/home/miguel/tfg_multirobot/src/ros_control/hardware_interface/include/
hardware_interface/internal/interface_manager.h:69:85: warning: type qualifiers
ignored on function return type [-Wignored-qualifiers]
  static const void callConcatManagers(typename std::vector<T*>& managers, T* result)
```

Código Fuente E.41: WARNING: Package 'ur_modern_driver' is deprecated

– Warnings resuelto:*gazebo_version_helpers*

```
CMake Warning at /opt/ros/kinetic/share/catkin/cmake/catkin_package.cmake:166
(message):
  catkin_package() DEPENDS on 'gazebo' but neither 'gazebo_INCLUDE_DIRS' nor
  'gazebo_LIBRARIES' is defined.
Call Stack (most recent call first):
  /opt/ros/kinetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  gazebo-pkgs/gazebo_version_helpers/CMakeLists.txt:26 (catkin_package)
```

Código Fuente E.42: Warning: catkin_package() DEPENDS on 'gazebo' but neither 'gazebo_INCLUDE_DIRS' nor 'gazebo_LIBRARIES' is defined

Modificar el fichero ~/tfg_multirobot/src/gazebo-pkgs/gazebo_version_helpers/CMakeLists.txt, a partir de la línea 26 por lo siguiente, y se soluciona de forma similar este tipo de warnings:

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES gazebo_version_helpers
  CATKIN_DEPENDS gazebo_ros roscpp
  DEPENDS GAZEBO
)
```

Código Fuente E.43: Solución: catkin_package() DEPENDS on 'gazebo' but neither 'gazebo_INCLUDE_DIRS' nor 'gazebo_LIBRARIES' is defined

E.4. Pruebas y comprobación de la configuración base

Tras realizar previamente toda la instalación y configuración del sistema se puede proceder a realizar pruebas para comprobar su funcionamiento antes de proceder a realizar otras modificaciones. No se va a indicar que pruebas se puede realizar, pero los repositorios de origen tiene indicaciones para ejecutar pequeñas demostraciones que son muy útiles para comprender lo que pueden realizar.

El entorno de trabajo debería quedarse de la siguiente manera tras la instalación de todos los repositorios:

```
miguel@Omen:~/tfg_multirobot/src$ ls
CMakeLists.txt        geometry                 ros_control
gazebo-pkgs           leap_motion              roslint
gazebo_ros_pkgs       object_recognition_msgs  universal_robot
general-message-pkgs  robotiq_2finger_grippers ur_modern_driver
```

Código Fuente E.44: Directorio raíz con todos los paquetes instalados

Estos repositorios, serán la base para la implementación del proyecto, es decir, los recursos del sistema que se implementará.

Anexos F

Github

Se han creado dos repositorios de *Github*, en donde se explica paso a paso cómo recrear las soluciones realizas durante este Trabajo Fin de Grado. Este repositorio está disponible de forma pública de tal manera que cualquier persona puede consultarla, descargarla y modificarla como sea conveniente.

El objetivo es permitir que futuros proyectos de la Universidad de Zaragoza pueda usarlo como referencia, realizar proyectos más complejos en menos tiempo y sobre todo que el trabajo realizado les sea de utilidad.

El proyecto consta de dos repositorios:

- **MultiCobot-UR10-Gripper**: Este repositorio contiene la documentación principal de toda la implementación de las soluciones propuestas en este Trabajo Fin de Grado, se ha intentado organizarlo lo mejor posible para ser fácilmente reusable.

  Los ficheros en los directorios *moveit* y *no_moveit*, son las soluciones desarrolladas utilizando y sin utilizar el paquete MoveIt!, realizado exactamente la misma tarea. En esos ficheros se encuentra el *setup* de cada solución paso a paso partiendo de la previa instalación de ROS y sus dependencias indicado en el fichero *proyect_setup.md*. Las imágenes que se muestran en los *setups* se encuentran en el directorio *imgs_md*.

  Los ficheros dentro de la carpeta *src* del directorio raíz son los paquetes que el sistema se ha apoyado para su desarrollo, y el directorio *tfg_proyect* contiene las implementaciones y modificaciones propias realizadas.

  URL: `https://github.com/Serru/MultiCobot-UR10-Gripper`

- **MultiCobot-UR10-Gripper-Campero**: Este directorio es un ejemplo de cómo se ha trasladado lo realizado en simulación a un robot real, en este caso el robot Campero de la Universidad de Zaragoza.

URL: `https://github.com/Serru/MultiCobot-UR10-Gripper-Campero`

En caso de que el proyecto desapareciese o dejase de estar disponible para el público, así como cualquier otra duda o intriga que pudiese surgir, debe ponerse en contacto con el director de este Trabajo Fin de Grado.

Con su permiso, aquí esta su información pública de contacto:

Gonzalo López Nicolás
E-mail: gonlopez@unizar.es Despacho: D.0.12

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura